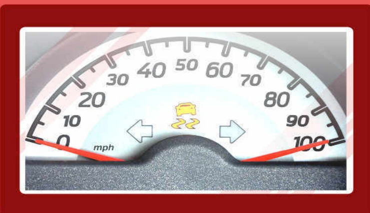
SERIE: PROGRAMACIÓN



De 0 a 99
en un solo libro

Un viaje desde la programación estructurada en pseudocódigo hasta las estructuras de datos avanzadas en lenguaje C

A. M. Vozmediano

ensegundapersona.es

Aprender a programar en C:

de 0 a 99 en un solo libro

Un viaje desde la programación estructurada en pseudocódigo hasta las estructuras de datos avanzadas en C.

A. M. Vozmediano

© 2005, 2017 Alfredo Moreno Vozmediano

Primera edición en este formato, abril de 2017.

Todos los derechos reservados.

Esta obra está protegida por las leyes de copyright y tratados internacionales.

http://ensegundapersona.es

ANTES DE EMPEZAR

Gracias por adquirir este manual. Ha sido puesto a disposición del público para tu aprovechamiento personal. Nada te impide copiarlo y compartirlo con otras personas y no podemos hacer nada por evitarlo. Sin embargo, si te parece que su contenido merece la pena y que el autor debe ser compensado, te

rogaríamos que no lo hicieras. Por el contrario, puedes recomendar a otros su compra. Al fin y al cabo, tampoco es que cueste una fortuna.

Gracias.

¿A quién va dirigido este libro?

Este libro está destinado a aquellas personas que quieren aprender a programar en lenguaje C. No es necesario que hayas programado previamente, aunque si ya sabes

programar el proceso de aprendizaje del lenguaje C te resultará mucho más sencillo, como es lógico. De hecho, gran parte de lo que se mencionará en la primera parte del libro te resultará familiar.

El libro NO da por hecho que el lector o lectora tenga conocimientos previos sobre programación ni sobre el lenguaje C, pero eso no quiere decir que se trate solo de un libro de introducción. Lo es, pero va más allá que la mayoría de los textos introductorios sobre cualquier

lenguaje, y se adentra en aspectos avanzados como las estructuras de datos dinámicas, los ficheros indexados o las bibliotecas gráficas.

¿QUÉ ENCONTRARÁS Y QUÉ NO ENCONTRARÁS AQUÍ?

En este libro encontrarás toda la información necesaria para convertirte en un programador o programadora de lenguaje C, desde las cuestiones más básicas, como la forma en la que opera

el sistema de numeración binario o las diferencias entre compiladores e intérpretes, hasta aspectos avanzados como los que mencionábamos antes.

Sin embargo, este libro no contiene TODO lo que se puede saber sobre C. Su propósito no es enciclopédico. Por eso hemos incluido en el subtítulo la expresión "de 0 a 99" y no "de 0 a 100". Seguramente habrá quien piense que 99 tampoco es la cifra exacta, pero no quisiéramos convertir esto en una discusión aritmética.

Este libro tampoco es una referencia del lenguaje. Hay excelentes manuales de referencia en el mercado, pero este no es uno de ellos. Los manuales de referencia son libros enormes donde se incluye un listado detallado de todas y cada una de las funciones de la librería estándar y, ocasionalmente, de algunas otras librerías de uso frecuente. Son libros que no se leen de principio a fin, sin que se usan para consultar puntualmente el uso de esta o de aquella función. No será necesario aclarar que, en la actualidad, los manuales de

referencia han sido sustituidos casi en su totalidad por referencias online que se pueden consultar de forma inmediata y actualizada en internet.

Por el contrario, el libro que tienes en las manos (o en la pantalla) es un *manual de aprendizaje*. Eso significa que está diseñado para leerlo de principio a fin e ir aprendiendo las técnicas del lenguaje mediante un enfoque constructivista, es decir, construyendo nuevos aprendizajes sobre los anteriores. Eso no quiere decir, por

supuesto, que sea obligatorio leerlo en ese orden. Es perfectamente posible leer solo las partes que te interesen, o volver sobre ellas cuando sea necesario, pero el programador novel encontrará seguramente más claro y productivo empezar por el principio e ir avanzando a lo largo del texto.

Por último, diremos algo más que este libro no es: una colección de ejercicios. Es habitual en los libros de programación incluir montañas de ejercicios para los aprendices. Suelen

ser ejercicios aburridísimos del tipo de "escribe un programa que calcule los cien primeros números primos" o "escribe un programa que convierta una cadena de caracteres a mayúsculas". Soy muy escéptico sobre la utilidad de tales ejercicios, aunque sé que esta idea es minoritaria. En mi opinión, esos ejercicios bienintencionados son aburridos, poco realistas y producen el efecto contrario al que pretenden: desincentivan el aprendizaje y desmotivan al aprendiz. Pienso que es mucho más productivo que el

programador ponga en práctica sus conocimientos (muchos o pocos) escribiendo los programas que él mismo desee, es decir, los que surjan de una motivación interna y no externa.

ENTONCES, ¿ESTE LIBRO NO TRAE EJERCICIOS?

Pues no, y sí. No los trae directamente pero, si estás empeñado en hacer ejercicios, te facilitamos una colección de cientos de ellos (algunos resueltos y otros solo propuestos) sin coste

económico adicional.

Ya hemos justificado por qué consideramos contraproducentes los ejercicios. No obstante, mucha gente se siente incómoda con este enfoque y prefiere disponer de una serie de ejercicios clásicos guiados. Para ese tipo de lector, hemos preparando una completa colección de ejercicios (ya sabes, del tipo de "escribe un programa que calcule los cien primeros números primos", y así), incluyendo una buena batería de casos resueltos. **Con la**

adquisición de este libro tienes acceso a esa colección de ejercicios gratis. Si estás interesado/a, solo tienes que dejarnos tu correo electrónico y te enviaremos el libro de ejercicios en formatos PDF junto con el código fuente de multitud de ejercicios resueltos. Nos puedes facilitar tu correo electrónico en esta dirección:

http://ensegundapersona.es/programacion-en-c

¿Por qué sé que este

LIBRO FUNCIONA?

He sido profesor de informática durante más de quince años. En todo este tiempo, aunque he tocado muchos palos, me he centrado sobre todo en la enseñanza de la programación. Conforme los planes de estudio se actualizaban he pasado por múltiples lenguajes: C, C++, Visual Basic, Java, PHP… Pero siempre he tenido una predilección especial hacia C, el lenguaje con el que aprendí a programar en serio en mi juventud y el primero con

el que me topé en mi carrera como docente.

Durante todo este tiempo he desarrollado multitud de materiales para mi alumnado, algunos tan breves como un apunte en un blog, otros visuales como una presentación de diapositivas, y otros mucho más elaborados y extensos como este libro que estás leyendo ahora.

Este libro es, básicamente, una revisión a fondo del texto que elaboré hace años para mi alumnado de ciclo formativo de

Administración de Sistemas Informáticos. En aquellos tiempos gloriosos, se consideraba importante que los administradores de sistemas tuviesen una buena base de programación, y la administración educativa exigía que esta base se adquiriese con el lenguaje C.

Utilicé este texto con éxito durante varios años, y muchos estudiantes aprendieron a programar en C, incluso aunque ellos mismos no se consideraban "programadores". Pero no te voy a

engañar: hubo algunos que no lo consiguieron. Mejor dicho, hubo algunos que no lo intentaron y tiraron la toalla al poco de empezar. Si tú perteneces al primer o al segundo grupo es algo que te corresponderá a ti decidir, y ningún libro ni ningún profesor podrá tomar la decisión por ti. Aprender a programar desde cero no es fácil. Hay gente que lo consigue en poco tiempo, y hay otros a quienes les cuesta un gran trabajo. Pero estoy convencido de que, con el esfuerzo debido, cualquier persona puede aprender a programar. Lo he visto

muchas veces.

Piensa que, por muy difícil que resulte, por muy arcana que parezca la labor del programador, los lenguajes de programación no dejan de ser más que invenciones humanas, artefactos creados por gente como tú o como yo que cada día usan a lo largo y ancho del globo millones de personas como tú o como yo. Y si ellos han podido aprender, tú también puedes.

La edición actual es fruto de muchas revisiones y refinamientos del texto

original. He puesto en ella todo el material que he ido elaborando sobre el lenguaje C a lo largo de los años y la he organizado del modo que mejor demostró funcionar con mi alumnado. De modo que este texto ha sido probado con personas reales. Por eso sé que este libro funciona.

Y ahora, ¿estás preparado (o preparada) para empezar?

ORGANIZACIÓN DEL LIBRO

Este es un libro largo. Es difícil encontrar un manual extensivo de programación que no alcance la categoría "pisapapeles". Hemos procurado resumir todo lo posible sin prescindir de la legilibilidad, pero aún así nos han salido casi cuatrocientas páginas en la edición impresa... y eso después de reducir el cuerpo de la tipografía principal de 12 a 11 puntos y hemos sacado a parte los ejercicios (que sumarían otras doscientas páginas con facilidad).

Para ayudarte a orientarte en este territorio tan amplio, te ofrecemos aquí un breve esquema de qué encontrarás en los siguientes capítulos y dónde encontrarlo.

Hemos dividido el libro en seis partes de nivel de dificultad creciente, de modo que el libro pueda leerse de principio a fin, pero también consultarse con facilidad por partes.

En la PRIMERA PARTE hacemos una introducción a las ciencias de la computación, empezando desde cero.

Aquí explicaremos qué es el sistema binario, el código ASCII o las unidades de medida de información. Luego hablaremos de los distintos lenguajes de programación que existen, para detenernos en los lenguajes de programación imperativos y en sus estructuras de control. Expondremos también los tipos de datos simples, las expresiones y los operadores, los fundamentos de la programación modular mediante funciones y procedimientos, y construiremos nuestros primeros algoritmos sencillos

con pseudocódigo y diagramas de flujo.

La SEGUNDA PARTE estará dedicada por entero al lenguaje C. Aprenderemos a traducir a ese lenguaje las estructuras que estudiamos en la parte anterior. Veremos como gestiona C la entrada y salida de datos y la interacción con el usuario y haremos una introducción a los pasos que debes seguir para desarrollar tus propios programas. Si ya sabes programar pero nunca has programado en C, quizá sea prefieras saltarte la primera parte y empezar por la segunda

directamente.

La TERCERA PARTE se dedica a las estructuras de datos estáticas. Estudiaremos los arrays en C en todas sus formas (cadenas, vectores, matrices, arrays multidimensionales), y también los utilísimos registros o *structs*. No nos dejaremos nada en el tintero, así que luego hablaremos de las uniones y las enumeraciones. Terminaremos con una introducción a las técnicas de definición de nuestros propios tipos de datos.

En la CUARTA PARTE nos centraremos

en los ficheros. Hablaremos de la estructura interna de los ficheros y de los diferentes tipos de fichero que podemos construir y manejar, centrándonos, sobre todo, en tres los ficheros secuenciales, los de acceso aleatorio y los indexados. Estudiaremos como maneja los flujos de datos el lenguaje C, y todas las funciones necesarias para manipular ficheros binarios y de texto desde nuestros programas.

La QUINTA PARTE versará sobre las

estructuras de datos dinámicas. Aquí hablaremos en detalle de los míticos punteros de C y aprenderemos a construir estructuras de datos muy útiles como listas, pilas, colas y árboles. También haremos un inciso para hablar de los algoritmos recursivos, estrechamente emparentados con los árboles.

En la SEXTA PARTE recogemos algunos aspectos avanzados del lenguaje C que no tienen cabida en las partes anteriores sin menoscabar la

comprensibilidad del texto. Encontrarás información sobre la compilación de proyectos grantes, la creación de librerías, los espacios con nombre o el preprocesador de C. También hablaremos de los argumentos de la línea de comandos y de las capacidades del lenguaje para la manipulación de datos a nivel de bits.

Por último, en los APÉNDICES encontraremos información de referencia sobre diversos aspectos importantes, pero no imprescindibles.

Por eso se han incluido en forma de apéndices, de manera que el lector decida cuál de ellos le resulta útil. Hay una lista no exhaustiva de funciones de uso frecuente de ANSI C, una breve guía de usuario del compilador Dev-C++ para Windows y otra del legendario compilador gcc para Linux (incluiremos aquí las instrucciones para la construcción de los Makefiles), seguidas de unas breves pero prácticas introducciones a las librerías ncurses y SDL, con las que podemos dotar de colores y gráficos a nuestros programas.

¿Alguna sugerencia?

Ningún texto está por completo libre de errores, y este, sin duda, no es una excepción. Si encuentras cualquier cosa que te chirríe, o simplemente quieres hacer alguna sugerencia, puedes escribirnos a admin@ensegundapersona.es. Te aseguramos que lo tendremos muy en cuenta para futuras ediciones.

PRIMERA PARTE: PROGRAMACIÓN ESTRUCTURADA

Nuestra historia comienza por donde suelen comenzar todas las historias: por el principio. En esta primera parte hablaremos de la programación estructurada clásica. Si ya has programado antes, es posible que prefieras saltar directamente a la

segunda parte, donde aplicaremos todo lo que se dice aquí al lenguaje C.

Lo primero, y especialmente dedicado a las personas recién llegadas, será aclarar algunos conceptos fundamentales: qué es un programa informático, qué son el código binario y el hexadecimal, cómo operan y por qué son tan importantes en el ámbito informático, en qué demonios se diferencian un bit y un byte, un kilobyte (KB) y un megabyte (MB), un gigabit (Gb) y un gigabyte (GB), y no digamos

un kilobyte (KB) y un kibibyte (KiB); y, después de ese trabalenguas, aclararemos cuál es el papel del programador en el engranaje de desarrollo del software.

Nos adentraremos más adelante en los tipos de datos simples, las constantes y las variables, las expresiones y los operadores. Luego pasaremos a los estilos de programación clásicos, hasta desembocar en el teorema de la programación estructurada (que no cunda el pánico: no habrá abstrusas

demostraciones matemáticas). Aprenderemos a usar las estructuras secuencial, condicional e iterativa con pseudocódigo y diagramas de flujo, y más adelante las aplicaremos a la programación modular, introduciendo las funciones y los procedimientos, el paso de parámetros y la devolución de resultados.

Terminaremos esta parte refiriéndonos a algunas reglas de estilo que todo programador novel debería conocer. He visto muchas veces un buen algoritmo

arruinado por una pésima escritura, así que merece la pena dedicar unos minutos a asimilar las cuatro normas básicas para adquirir buenos hábitos de escritura desde el principio.

Para empezar, los fundamentos

Es este apartado vamos a hablar de algunas cuestiones básicas que debes tener claras antes de empezar a programar. Del mismo modo que nadie se lanzaría a conducir un coche sin

conocer al menos lo más importante del código de circulación, ni nadie construiría su propia casa sin saber algo sobre materiales e instalaciones, es demasiado aventurado pensar que podemos programar un ordenador sin saber algunas cosas importantes sobre esos silenciosos compañeros de escritorio.

Sin embargo, si tienes tiene cierta experiencia con ordenadores o ya has programado antes, es posible que prefieras saltarte este capítulo y pasar

directamente al siguiente.

¿Qué es un programa de ordenador?

Empecemos por definir qué es un ordenador como lo entendemos hoy en día, teniendo en cuenta que ésta sólo es una de las muchas definiciones válidas:

“Un ordenador es una máquina digital y sincrónica, con cierta capacidad de cálculo numérico y lógico, controlada por un programa almacenado y con

posibilidad de comunicación con el mundo exterior" (DE GUISTI, Armando; Algoritmos, datos y programas, Prentice-Hall, 2001)

Veamos cada aspecto de la definición por separado para intentar comprenderla bien:

- Máquina digital: el ordenador sólo maneja señales eléctricas que representan dos estados de información. Estos dos estados, en binario, son el 0

y el 1.

- Máquina sincrónica: todas las operaciones se realizan coordinadas por un único reloj central que envía pulsos a todos los elementos del ordenador para que operen al mismo tiempo.
- Tienen cierta capacidad de cálculo: los ordenadores, normalmente, sólo son capaces de realizar operaciones muy simples, ya

sean aritméticas (sumas, restas, productos, etc) o lógicas (comparaciones de números)

- Está controlada por un programa almacenado: significa que los ordenadores tienen guardado internamente un conjunto de instrucciones y las obedecen en el orden establecido por el programador, que es quien ha escrito esas instrucciones.

- Se comunica con el mundo exterior a través de diferentes dispositivos periféricos de entrada (como el teclado, el ratón, el escáner...) o de salida (monitor, impresora...)

Según esta definición de ordenador, podemos deducir que un programa de ordenador es un conjunto de instrucciones ordenadas y comprensibles para un ordenador, además de un conjunto de datos

manipulados por esas instrucciones, de manera que el ordenador realice alguna tarea.

Todos los programas deben tener una función específica, es decir, una tarea que realizar. Por ejemplo, gestionar las facturas de una empresa (si es un programa de facturación) o acabar con todo bicho viviente (si es un videojuego ultraviolento). Normalmente, el programa deberá alcanzar su objetivo en un tiempo finito, es decir, empieza en un momento dado y termina en otro

momento posterior.

Los programas utilizan datos. Un dato es una representación de algún objeto del mundo real relacionado con la tarea que trata de realizar el programa.
Representar los datos en un ordenador suele ser complicado porque, debido a su naturaleza digital, todos los datos deben tener forma binaria, cuando está claro que el mundo real no es binario en absoluto. Por lo tanto, para representar objetos reales en un programa es necesario transformarlos el objetos

binarios. Éstos objetos binarios son los que llamamos datos.

Por ejemplo, en el programa que gestiona las facturas de una empresa, uno de los muchos objetos del mundo real que se han de manejar es el nombre de los clientes. ¿Cómo representar un nombre compuesto por letras en un ordenador que sólo admite código binario, es decir, ceros y unos? Este es uno de los problemas a los que se enfrenta el programador. Y la cosa se complica con objetos más complejos,

como imágenes, sonidos, etc.

Resumiendo: los ordenadores son herramientas muy potentes que pueden resolver problemas muy diversos, pero es necesario programarlas, es decir, proporcionarles las instrucciones y los datos adecuados. Y eso es lo que vamos a aprender a hacer a lo largo de este libro.

Codificación de la información

El ordenador es una máquina digital, es

decir, binaria. Antes de proseguir, es conveniente repasar el código binario y sus implicaciones. Los programadores de alto nivel no necesitan conocer cómo funciona el código binario ni otros códigos relacionados (como el hexadecimal), pero, en muchas ocasiones, resulta muy conveniente que estén familiarizados con ellos. Los programadores en C, en cambio, sí que lo necesitan. Esto es debido a que C es un lenguaje de más bajo nivel que otros. Ojo, que esto no es ningún insulto. En este contexto, "bajo nivel" significa

"más próximo al hardware". Eso tiene sus ventajas y sus inconvenientes, como veremos más adelante. Por ahora, baste decir que todos los programadores en C deben conocer bien cómo funcionan los sistemas de codificación binario y hexadecimal porque son los lenguajes nativos del hardware del ordenador.

Dedicaremos, pues, el resto de este apartado a conocer esos sitemas de codificación.

Códigos

Un código es un método de representación de la información. Se compone de un conjunto de símbolos, llamado alfabeto, y de un conjunto de reglas para combinar esos símbolos de forma correcta.

- Ejemplo 1: la lengua castellana es un código. Su alfabeto es el abecedario (a, b, c, d, e ... z), pero los símbolos del alfabeto no se pueden combinar a lo loco, sino que existen unas reglas,

y sólo siguiendo esas reglas se codifica correctamente la información, dando lugar a mensajes con sentido. Esas reglas las adquiriste hace años, cuando aprendiste a leer y escribir.

- Ejemplo 2: el código morse también es un código. Su alfabeto es mucho más reducido, puesto que solo se compone de dos símbolos: el punto (.) y la raya (–). Pero

combinando los dos símbolos correctamente, se puede transmitir cualquier información.

- Ejemplo 3: el sistema de numeración decimal también es un código. Tiene un alfabeto de 10 símbolos (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9). Combinándolos según ciertas reglas, puede usarse para transmitir información. Pero ojo, no cualquier

información, solamente información numérica. Hemos dicho que los códigos sirven para representar información, pero no que tengan que servir para representar toda la información posible. Aunque sólo sirva para los números, el sistema de numeración también es un código.

Código binario

El sistema de numeración binario es muy

parecido al sistema de numeración decimal; por lo tanto, también es un código. La única diferencia con el sistema decimal es la cantidad de símbolos del alfabeto. Si el decimal tiene diez, el binario sólo tiene dos: el 0 y el 1. En todo lo demás son iguales, así que el sistema binario también sirve para representar información numérica.

Pero, ¿puede representarse cualquier número con sólo dos símbolos?

La respuesta es sí. El modo de hacerlo consiste en combinar los símbolos 0 y 1

adecuadamente, igual que hacemos con los números decimales. En el sistema decimal contamos así: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Cuando queremos pasar a la siguiente cantidad, empezamos a agrupar los dígitos de dos en dos: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19. Al volver a terminar las unidades, vamos incrementando las decenas: 20, 21, 22, etc.

(Esto se debe a que, en los sistemas de numeración, cada dígito tiene un valor posicional, es decir, tiene un valor

diferente dependiendo del lugar que ocupe en el número general. Por ejemplo, en el número 283, el 3 tiene valor de tres, pero el 8 no tiene valor de ocho, sino de ochenta, y el 2 no tiene valor de dos, sino de doscientos)

En binario, el razonamiento es el mismo. Empezamos a contar por 0 y 1, pero entonces ya hemos agotado los símbolos, así que empezamos a agruparlos: 10, 11. Como hemos vuelto a agotarlos, seguimos combinándolos: 100, 101, 110, 111, 1000, 1001, 1010, y

así sucesivamente.

Así, los 16 primeros números binarios comparados con sus equivalentes decimales son:

Decimal	Binario
0	0
1	1
2	10
3	11

4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011

12	1100
13	1101
14	1110
15	1111

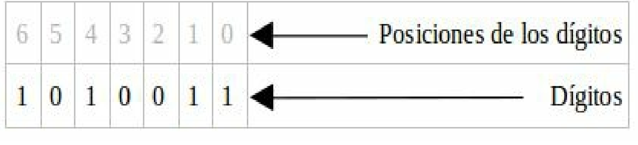
Los números escritos en código binario tienen el mismo valor que en decimal, y sólo cambia la representación. Es decir, “15” en decimal y “1111” en binario representan exactamente a la misma idea: quince. Son, en realidad, dos

formas distintas de expresar lo mismo. O, dicho de otro modo, la misma cosa escrita en dos idiomas diferentes.

Convertir números binarios a decimales

Para obtener la representación decimal de un número binario hay que proceder del siguiente modo:

1) Numeramos la posición que ocupa cada dígito binario de derecha a izquierda, empezando por 0. Por ejemplo, en el número binario 1010011, numeraremos las posiciones así:



2) Multiplicamos cada dígito binario por 2 elevado a la posición del dígito y sumamos todos los resultados. Con el número del ejemplo anterior:

6	5	4	3	2	1	0
1	0	1	0	0	1	1

1×2^6	0×2^5	1×2^4	0×2^3	0×2^2	1×2^1	1×2^0
64	0	16	0	0	2	1

Ahora sólo nos quedaría sumar los resultados de todas las multiplicaciones:

$$64 + 0 + 16 + 0 + 0 + 2 + 1 = 83$$

Por lo tanto, el número binario 1010011 es equivalente al número decimal 83. Es habitual indicar con un subíndice el sistema de numeración en el que está escrito cada número, así:

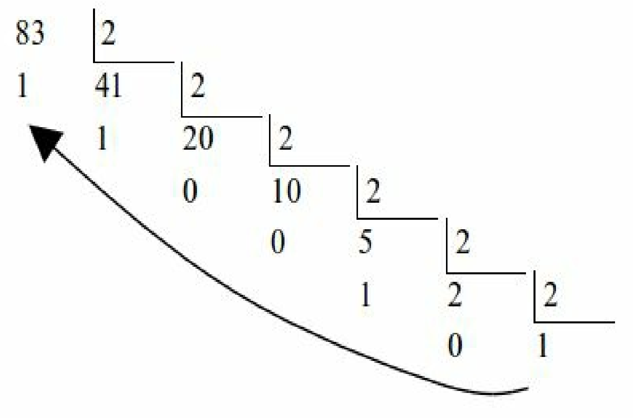
$$1010011_2 = 83_{10}$$

Convertir números decimales a binarios

El proceso contrario se realiza

dividiendo sucesivamente el número decimal entre dos, y cogiendo el último cociente y todos los restos en el orden inverso al que los obtuvimos.

Por ejemplo, vamos hallar la representación binaria del número decimal 83:



Tomando el último cociente (que siempre es 1) y todos los restos desde el último hacia el primero (es decir, 010011, siguiendo la dirección de la flecha), obtenemos el número binario 1010011. Por lo tanto, podemos decir

que:

$$83_{10} = 1010011_2$$

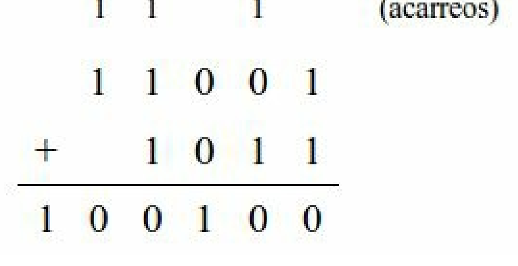
Operaciones aritméticas binarias

La operaciones aritméticas binarias se realizan exactamente igual que las decimales, aunque teniendo la precaución de usar sólo los dos símbolos permitidos (0 y 1), lo que puede parecernos un poco extraño al principio.

Por ejemplo, para realizar una suma de dos números binarios, podemos usar el

algoritmo de suma que aprendimos en la escuela: escribiremos ambos números uno encima de otro, alineados a la derecha, como hacíamos cuando éramos tiernos infantes y nos enseñaron a sumar. Luego, iremos sumando los dígitos de derecha a izquierda, como haríamos con dos números decimales, con la precaución de sumar también el acarreo ("me llevo una") cuando se produzca.

Vamos a sumar los números binarios 11001 y 1011:



Del mismo modo, pueden realizarse otras operaciones aritméticas como restas, productos o divisiones. No te preocupes demasiado por ello. Aunque es perfectamente posible aprender a operar en binario, no tiene una gran utilidad práctica. El sistema binario es tremendamente incómodo para nosotros, los humanos, y no necesitas hacer raices

cuadradas en binario para entenderte con tu ordenador. Con lo que hemos visto hasta ahora será suficiente.

Código ASCII

Hasta ahora hemos visto que mediante el código binario se pueden representar números, pero no sabemos cómo se las apaña un ordenador para representar las letras, o, dicho en terminilogía informática, los caracteres alfanuméricos (que incluyen números, letras y otros símbolos habituales, como los signos de puntuación).

El código ASCII consiste en una correspondencia entre números binarios de 8 dígitos y caracteres alfanuméricos. Así, por ejemplo, al número decimal 65 (en binario, 01000001, y observa que escribimos 8 dígitos, rellenando con ceros a la izquierda si es necesario) se le hace corresponder la letra A, al 66 la B, al 67 la C, etc. De este modo, el ordenador puede también manejar letras, y lo hace del mismo modo en que maneja números: mediante combinaciones de ceros y unos.

Es importante resaltar que los códigos ASCII siempre tienen 8 dígitos binarios, rellenándose con ceros a la izquierda si fuera necesario. Así ocurre en el caso de la letra A, que, como hemos dicho, se representa con el código binario 01000001.

Originalmente, la codificación ASCII solo usaba 7 bits, lo que proporcionaba un juego de 128 caracteres. Eso era suficiente para la lengua inglesa, pero, cuando se intentaron introducir los símbolos de otras lenguas, se hizo

necesario añadir el octavo bit. Por eso, a veces al código ASCII de 8 bits se le llama código ASCII extendido.

Aún así, el código ASCII tiene serias limitaciones para representar todos los símbolos de todas las lenguas. De hecho, existen tablas de código ASCII para el español, tablas de código ASCII para el ruso, tablas de código ASCII para el griego, y así sucesivamente con otros idiomas. En la actualidad, la International Organization for Standardization (ISO) ha propuesto

diversos estándares que engloban al antiguo ASCII. Así, el ISO-8859-1, también llamado ISO-Latin-1, se corresponde con la vieja tabla ASCII para los idiomas español, catalán, gallego y euskera, además de otros como inglés, alemán, francés, italiano, etc.

El código ASCII no es, desde luego, el único que existe para representar letras en binario, pero sí el más popular por motivos históricos. En la actualidad, se ha impuesto con fuerza el sistema de codificación UTF-8 de Unicode, con el

cual se consiguen representar todos los caracteres de todas las lenguas del mundo, incluyendo todos los dialectos de chino, árabe o japonés (¡e incluso las lenguas muertas!) sin ambigüedad posible. Pues bien, los primeros 256 caracteres del UTF-8 se corresponden con el ISO-8859-1, es decir, con la vieja tabla ASCII que incluye los símbolos en español. Por eso mismo, y para propósitos prácticos, las tablas clásicas de código ASCII siguen siendo vigentes.

Código hexadecimal

Es importante conocer y saber manejar el código binario al ser el método de codificación que emplean los ordenadores digitales, pero este código tiene dos serios inconvenientes:

- Primero, resulta difícil de manipular para cerebros que, como los nuestros, están habituados a pensar en decimal (o habituados a no pensar en absoluto, que también se da el caso).
- Segundo, los números

binarios pueden llegar a tener cantidades enormes de dígitos (es habitual trabajar con números de 16, 32 ó 64 dígitos binarios), lo cual los convierte en inmanejables.

Por este motivo, suelen usarse, en programación, otros dos sistemas de numeración llamados octal y hexadecimal. El octal maneja 8 símbolos distintos y, el hexadecimal, 16. Sin duda, el más utilizado es el hexadecimal y por este motivo nos

vamos a detener en él, aunque haciendo notar que el octal funciona de la misma manera, sólo que empleando los dígitos del 0 al 7.

Si el sistema binario utiliza dos símbolos (0 y 1) y el decimal utiliza 10 (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9), el hexadecimal emplea 16 símbolos, que son: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F.

En hexadecimal, por tanto, es normal ver números cuyos dígitos son letras del alfabeto. Por ejemplo: 2AF5 es un

número válido escrito en hexadecimal (exactamente, el 10997 en decimal). La forma de contar, por supuesto, es la misma: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, y después empezamos a agrupar los símbolos: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F. Seguiríamos con 20, 21, 22, etc.

Recuerda: "20" en hexadecimal no significa "veinte", sino que significa "treinta y dos". Y, por su puesto, se escribe "32" en decimal. Tienes que ver el hexadecimal y el decimal como dos

idiomas distintos para expresar los mismos números. Del mismo modo que un mismo concepto, por ejemplo "mesa", se puede escribir como "table" (en inglés) o "tavolo" (en italiano), otro concepto, como "treinta y dos", se puede escribir como "20" (en hexadecimal) o como "32" en decimal. Si nosotros, pobres humanos, vemos "32" y pensamos automáticamente "treinta y dos", es porque estamos acostumbrados al sistema de numeración decimal, pero te aseguro que, si tuviéramos dieciseis dedos en lugar de diez, nos habríamos

acostumbrado igual de bien al hexadecimal.

Podemos construir una tabla para comparar los primeros números en los tres sistemas de numeración que conocemos. Hemos rellenado los primeros números binarios con ceros a la izquierda por razones que pronto se verán, pero en realidad los números no cambian (recuerda que un cero a la izquierda no tiene ningún valor, ni en binario ni en el resto de sistemas)

Decimal	Binario	Hexadecimal

0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6

7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E

15	1111	F

Si te fijas, cada dígito hexadecimal se corresponde exactamente con una combinación de 4 dígitos binarios. Así, por ejemplo, el número binario 1001 1101 se puede escribir más resumidamente como 9D en hexadecimal. Y esa es la gran utilidad del sistema hexadecimal: permite manipular números binarios de forma más escueta y resumida, de manera que nos sean más fáciles de manejar a

nosotros, los humanos, que somos muy propensos a cometer errores.

Convertir números hexadecimales a decimales

El mecanismo es el mismo que ya utilizamos para convertir números binarios, sólo que cambiando la base del sistema de numeración de 2 a 16, ya que ahora vamos a manejar números hexadecimales. Por lo tanto, los pasos a seguir son:

1) Numeramos las posiciones que ocupa cada dígito hexadecimal de derecha a

izquierda, empezando por 0. Por ejemplo, en el número hexadecimal 2AF, numeraremos las posiciones así:

2 1 0 (posiciones)

2 A F (dígitos)

2) Multiplicamos cada dígito hexadecimal por 16 elevado a la posición del dígito y sumamos todos los resultados. Con el número 2AF lo haríamos así:

$$2 \times 16^2 + A \times 16^1 + F \times 16^0$$

Según la tabla de anterior, tenemos que el dígito hexadecimal A equivale a 10 en decimal, y que F equivale a 15. Por lo tanto la operación quedaría así:

$$2 \text{ x } 16^2 + 10 \text{ x } 16^1 + 15 \text{ x } 16^0$$

Ahora sólo nos falta resolver la operaciones y sumar:

$$2 \text{ x } 256 + 10 \text{ x } 16 + 15 \text{ x } 1 = 687$$

Por lo tanto, el número hexadecimal 2AF es equivalente al número decimal 687. Indicándolo con subíndices, lo expresaríamos así:

$$2AF_{16} = 687_{10}$$

Convertir números decimales a hexadecimales

El proceso también es idéntico al realizado con números binarios, pero sustituyendo la división entre 2 por divisiones entre 16, que es la base del sistema hexadecimal. Como suponemos que todo el mundo sabe dividir, nos ahorraremos otro aburrido ejemplo de divisiones sucesivas.

En realidad, convertir números de decimal a hexadecimal y a la inversa es

algo que solo hacemos los humanos para nuestra comodidad, porque a veces nos da grima ver esos números hexadecimales tan raros. Pero, a un ordenador, el sistema decimal le importa un pimiento. Lo suyo es el binario, pero también el hexadecimal, porque, en realidad, ambos sistemas, binario y hexadecimal, están directamente emparentados.

Relación entre números hexadecimales y binarios

La verdadera utilidad del sistema

hexadecimal es que se puede utilizar en lugar del binario, siendo más fácil de manejar. Para que ello sea posible, el paso de hexadecimal a binario y viceversa debe poder hacerse con mucha rapidez.

Para convertir un número hexadecimal a binario, basta con sustituir cada dígito hexadecimal por sus cuatro cifras binarias correspondientes, según la tabla de la página anterior. Por ejemplo:

$$2AF_{16} = 0010\ 1010\ 1111_2$$

Del mismo modo, para convertir un

número binario a hexadecimal, lo agruparemos en bloques de 4 cifras binarias (empezando por la derecha) y buscaremos la correspondencia en la tabla. Por ejemplo, el número binario 100100 se convierte así:

$$0010\ 0100_2 = 24_{16}$$

Observa que hemos rellenado con ceros a la izquierda para obtener bloques de 4 dígitos binarios sin alterar la esencia del número. Por supuesto, no es obligatorio hacerlo, pero las primeras veces puede facilitar las cosas. Con un poco de

práctica conseguirás convertir binarios a hexadecimales y viceversa de un sólo vistazo y sin necesidad de consultar la tabla.

Unidades de medida de información

Como hemos visto, el código binario es el fundamento del funcionamiento de los ordenadores: toda la información que el ordenador maneja, ya sea numérica o alfanumérica, se encuentra codificada en binario.

Del mismo modo que para medir distancias se utiliza el metro, o para medir masas se utiliza el gramo, para medir la cantidad de información almacenada o procesada en un ordenador existe otra unidad de medida. Como el ordenador representa toda la información en binario, la unidad fudamental es el dígito binario (es decir, 0 ó 1), también llamado BIT (de BInary digiT)

Un bit es realmente muy poca cantidad de información. Recuerda que, por

ejemplo, para almacenar un sólo carácter en código ASCII son necesarios 8 bits. ¡Para un único carácter! Del mismo modo que el metro dispone de múltiplos (el decámetro, el hectómetro, el kilómetro, etc), también los tiene el bit, y son los siguientes:

- Byte: 1 byte equivale a 8 bits. Cuidado con el nombre, porque se parecen y es un error común confundir el bit con el byte.
- Kilobyte (KB): 1 kilobyte

son 1000 bytes. Sí, hemos dicho 1000. Si te parece que es un error y que deberíamos haber dicho 1024, sigue leyendo y quizá logremos sorprenderte.

- Megabyte (MB): 1 megabyte equivale a 1000 kilobytes.
- Gigabyte (GB): 1 gigabyte equivale a 1000 megabytes.
- Terabyte (TB): 1 terabyte equivale a 1000 gigabytes

Tal vez hayas oído decir que los múltiplos de las unidades de información no equivalen a 1000 unidades de la unidad anterior, sino a 1024. Es cierto. Esto se debe a que 1024 es la potencia de 2 (2 elevado a 10) más próxima a 1000, el múltiplo clásico del sistema métrico decimal.

Históricamente, 1 KB equivalía a 1024 Bytes y no a 1000 Bytes, y 1 MB eran 1024 KB, y así sucesivamente. Todavía mucha gente lo considera así. Sin embargo, otras personas, en especial fabricantes de hardware y

comercializadoras de fibra óptica, utilizan potencias de 10 por la sencilla razón de que así parece que te venden más por menos.

Me explico: si compras un disco duro de 1 TB, ¿cuál es se capacidad real? Porque no es lo mismo que lo hayan calculado con potencias de 10 que con potencias de 2:

- Con potencias de 10, la capacidad es: 1 x 1000 x 1000 x 1000 x 1000 Bytes.
- Con potencias de 2, la

capacidad es: 1 x 1024 x 1024 x 1024 x 1024 Bytes.

No es necesario poseer un doctorado en matemáticas para darse cuenta de que la segunda cantidad es mayor, es decir, que te pueden vender dos discos duros de 1 TB y, cuando llegues a tu casa, comprobar que en realidad tienen diferente capacidad. Ni que decir tiene que los fabricantes suelen usar las potencias de 10 para anunciar la capacidad de sus productos, porque así parece que tienen más.

Por ese motivo, la ISO introdujo una nueva familia de medidas de información que se diferencias de las anteriores porque llevan una letra "i" intercalada y porque se calculan como potencias de 2 y no de diez. Son las siguientes:

- Byte: Esta no cambia. 1 byte sigue siendo igual a 8 bits.
- Kibibyte (KiB): 1 kibibyte son 1024 bytes.
- Mebibyte (MiB): 1 mebibyte equivale a 1024 kibibytes.

- Gibibyte (GiB): 1 gibibyte equivale a 1024 mebibytes.
- Tebibyte (TiB): 1 tebibyte equivale a 1024 gibibytes

Por último, ten en cuenta que el Byte suele abreviarse con la B mayúscula y el bit con la b minúscula, y que el primero es 8 veces mayor que el segundo. Así que, si tu compañía telefónica te está ofertando una conexión a interntet de 300 Mbps, se trata de Megabits (potencias de 10 y bits). Si fueran 300 MiBps (potencias de 2 y Bytes), tendrías

un ancho de banda mucho mayor:

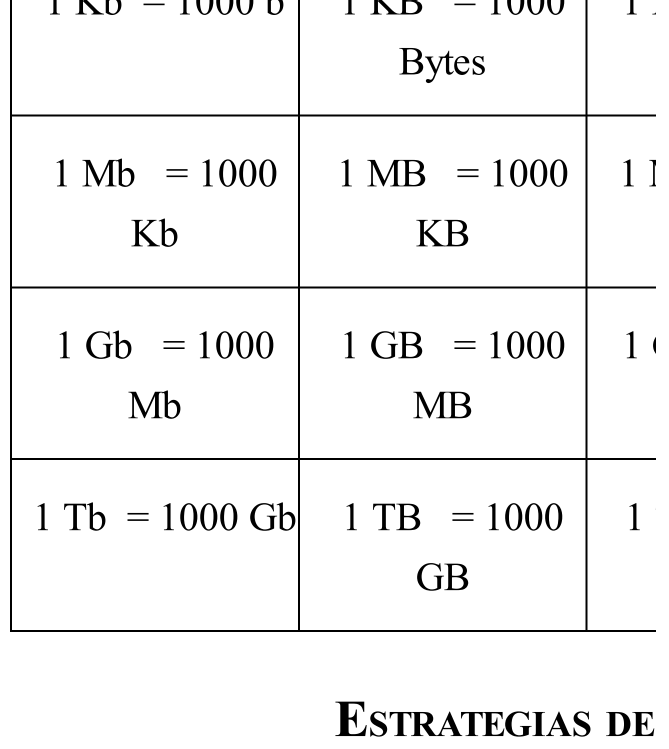
- 300 Mpbs = 300 x 1000 x 1000 = 300.000.000 bits por segundo.
- 300 MiBps = 300 x 1024 x 1024 x 8 = 2.516.582.400 bits por segundo.

La segunda cifra es 8,4 veces mayor que la primera, así que la diferencia no es ninguna tontería.

Finalizamos esta sección con un pequeño cuadro resumen con las

unidades de medida de cantidad de información más habituales. Ten en cuenta que existen múltiplos aún mayores, como el EB (Exabyte) o el PB (Petabyte), pero que, de momento, no tienen gran aplicación práctica.

Potencias de 10		
Múltiplos de bit (b)	**Múltiplos del Byte (B)**	**Mú**
	1 Byte = 8 bits	



1 Kb = 1000 b	1 KB = 1000 Bytes	1
1 Mb = 1000 Kb	1 MB = 1000 KB	1
1 Gb = 1000 Mb	1 GB = 1000 MB	1
1 Tb = 1000 Gb	1 TB = 1000 GB	1

ESTRATEGIAS DE

RESOLUCIÓN DE PROBLEMAS

Después de este necesario paréntesis dedicado al sistema binario y el hexadecimal, retomamos el hilo de nuestro discurso: la programación de ordenadores como método para resolver problemas del mundo real.

Ingeniería del software

Los programas de ordenador son

productos realmente complejos (y caros) de diseñar y construir. Al principio, con los primeros ordenadores de la historia, esto no era así. Aquellos ordenadores eran tan elementales que sus programas no podían ser demasiado complicados, y podían ser desarrollados por cualquiera con algunos conocimientos del funcionamiento de la máquina.

Pero, a lo largo de la década de 1970, el avance de la tecnología provocó que los ordenadores tuvieran cada vez más capacidad de cálculo y, por lo tanto, que

los programas fueran cada vez más complejos. Llegó un momento en el que se hizo evidente que ningún ser humano era capaz de construir un programa tan complejo que aprovechase todas las posibilidades de hardware de los ordenadores de esa época. A esto se le llamó *crisis del software*, y estancó la industria informática durante varios años.

El problema era que, hasta entonces, se programaba sin método ni planificación. A nadie se le ocurriría, por ejemplo,

construir un avión sin haber hecho antes, cuidadosamente, multitud de cálculos, estudios, planos, diseños, esquemas, etc. Pues bien, un programa de ordenador puede ser tan complejo, o más, que un avión o cualquier otro artefacto industrial, y, por lo tanto, es necesario construirlo con los mismos procesos de ingeniería.

Surgió así el concepto de ingeniería del software, que podemos definir como el conjunto de procedimientos y técnicas encaminadas a diseñar y desarrollar

programas informáticos y su documentación interna y externa.

Actualmente, los procesos de la ingeniería del software (que son muchos y variados) se aplican en todas las empresas y organismos en los que se desarrolla software de forma profesional y rigurosa, porque no hay otro modo de asegurar que el producto se va a terminar dentro de los plazos y costes previstos, y que éste va a funcionar correctamente y se va a ajustar a los niveles de calidad que el mercado

exige.

Ciclo de vida clásico

Una de las primeras enseñanzas de la ingeniería del software fue que, al ser el proceso de producción de software tan complicado, debía descomponerse en varias etapas para poder abordarlo.

El conjunto de estas etapas, o fases, constituyen lo que se denomina el ciclo de vida del software.

Dependiendo de diversos factores (como el tipo de software que se va a

desarrollar, el sistema en el que va a funcionar, o las propias preferencias de los ingenieros o de la empresa desarrolladora), se puede elegir entre varios tipos de ciclos de vida que han demostrado su eficacia a lo largo de los años. Pero la mayoría de ellos, con ligeras variaciones, constan de las siguiente fases:

- Análisis del problema
- Diseño de una solución
- Especificación de los módulos

- Codificación
- Pruebas
- Mantenimiento

A continuación se describen las fases del ciclo de vida, pero antes dejemos claro que este manual se centrará, principalmente, en las fases de especificación de módulos y codificación. También nos adentraremos, aunque sea superficialmente y desde un punto de vista práctico, en las etapas anteriores y posteriores, pero no es nuestra tarea.

Para eso existen unos profesionales, generalmente muy bien pagados, llamados analistas y diseñadores (busca ofertas de empleo para analistas en cualquier portal de empleo, y compara su sueldo con el de los programadores; ya verás qué diferencia). La buena noticia es que gran parte de los analistas han comenzado siendo programadores y, con el tiempo y la experiencia, han ido extendiendo su saber hacer al resto de fases del ciclo de vida.

Nosotros, pues, nos centraremos en lo

que compete al programador: las fases de especificación de módulos y la codificación. Pero antes, debemos saber en dónde nos estamos metiendo para conocer bien cuál es nuestro papel en el engranaje. De modo que, sin más dilación, pasamos a describir someramente en qué consisten las otras fases.

Análisis

La fase de análisis busca averiguar QUÉ problema vamos a resolver. Parece una obviedad, pero la experiencia demuestra

que no sólo no es así, sino que el análisis suele ser la etapa que más problemas causa y a la que más tiempo se le debería dedicar.

Es imprescindible partir de una especificación de requisitos lo más exacta y detallada posible. El resultado debe ser un modelo preciso del entorno del problema, de los datos y del objetivo que se pretende alcanzar. Pero expliquémoslo todo con más detenimiento:

El mundo real, por definición, es muy

complejo. Cuando pretendemos traspasar una parte de ese mundo a un ordenador es necesario extraer sólo los aspectos esenciales del problema, es decir, lo que realmente afecta a esa parte del mundo, desechando todo lo demás. El proceso de comprensión y simplificación del mundo real se denomina análisis del problema, y la simplificación obtenida como resultado del análisis se llama modelo.

Por ejemplo, si lo que pretendemos es realizar un programa que calcule la

trayectoria de un proyectil lanzado por un cañón de artillería (el clásico problema del tiro oblicuo, ¿recuerdas tus clases de física en el instituto?), lo lógico es que simplifiquemos el problema suponiendo que el proyectil es lanzado en el vacío (por lo que no hay resistencia del aire) y que la fuerza de la gravedad es constante. El resultado será muy aproximado al real, aunque no exacto. Esto es así porque nos hemos quedado con los aspectos esenciales del problema (la masa del proyectil, su velocidad, etc), desechando los menos

importantes (la resistencia del aire, la variación de la gravedad). Es decir, hemos realizado un modelo del mundo real.

En este ejemplo, el modelo del tiro oblicuo es muy fácil de construir ya que se basa en fórmulas matemáticas perfectamente conocidas. Necesitaremos saber algunos datos previos para que el modelo funcione: la velocidad del proyectil, su masa y su ángulo de salida. Con eso, nuestro programa podría calcular fácilmente la altura y la

distancia que el proyectil alcanzará.

Sin embargo, las áreas de aplicación de la Informática van más allá de la Física, por lo que la modelización suele ser bastante más difícil de hacer que en el ejemplo anterior.

Por ejemplo, en el programa de facturación de una empresa: ¿qué datos previos necesitamos conocer? ¿Qué fórmulas o cálculos matemáticos debemos realizar con ellos? ¿Qué resultado se espera del programa? Estas cuestiones deben quedar muy claras

antes de la modelización porque, de lo contrario, el modelo no será adecuado para resolver el problema y todo el proceso de programación posterior dará como fruto un programa que no funciona o no hace lo que se esperaba de él.

Para que el modelo sea acertado, por lo tanto, es necesario tener muy clara la naturaleza del problema y de los datos que le afectan. A este respecto, es imprescindible establecer lo que se denomina una especificación de requisitos, que no es más que una

definición lo más exacta posible del problema y su entorno. Sin una especificación detallada, es imposible comprender adecuadamente el problema y, por lo tanto, también es imposible hacer bien el análisis y construir un modelo que sea válido.

Los analistas cuentan con un montón de herramientas a su disposición, muchas de ellas en forma de diagramas, para hacer su trabajo, pero describirlas escapa a los propósitos de esta obra.

Diseño de soluciones

Una vez establecido el modelo del mundo real, y suponiendo que el problema sea computable, es necesario decidir CÓMO se va a resolver el problema, es decir, crear una estructura de hardware y software que lo resuelva (en este libro únicamente nos interesaremos por la parte del software)

Diseñar una solución para un modelo no es una tarea sencilla y sólo se aprende a hacerlo con la práctica. Típicamente, el diseño se resuelve mediante la técnica del diseño descendente (top-down), que

consiste en dividir el problema en subproblemas más simples, y estos a su vez en otros más simples, y así sucesivamente hasta llegar a problemas lo bastante sencillos como para ser resueltos con facilidad.

Nuevamente, los diseñadores cuentan con una gran batería de herramientas a su disposición, que no podemos describir aquí, para realizar el trabajo. Al final del proceso de análisis y diseño, deberíamos tener a nuestra disposición algo así como los planos del

edificio que vamos a construir. Los analistas han sido los arquitectos. Ahora llega el turno de los albañiles. Sí, lo has acertado: los albañiles somos nosotros, los programadores.

Especificación de módulos y codificación

Para cada subproblema planteado en el diseño hay que inventarse una solución lo más eficiente posible, es decir, crear un algoritmo. Veremos qué son los algoritmos más adelante, y dedicaremos el resto del libro a describir algoritmos

para todo tipo de problemas. Cada algoritmo que resuelve un subproblema se llama módulo.

Posteriormente, cada módulo debe ser traducido a un lenguaje comprensible por el ordenador, tecleado y almacenado. Estos lenguajes se llaman lenguajes de programación.

Los lenguajes de programación son conjuntos de símbolos y de reglas sintácticas especialmente diseñados para transmitir órdenes al ordenador. Existen multitud de lenguajes para hacer

esto. Hablaremos de ellos más adelante y centraremos el resto del libro en aprender a utilizar uno de ellos, ya sabes: el lenguaje C.

Pruebas

Una vez que el programa está introducido en la memoria del ordenador, es necesario depurar posibles errores. La experiencia demuestra que hasta el programa más sencillo contiene errores y, por lo tanto, este es un paso de vital importancia.

Los errores más frecuentes son los sintácticos o de escritura, por habernos equivocado durante la codificación. Para corregirlos, basta con localizar el error (que generalmente nos marcará el propio ordenador) y subsanarlo.

Más peliagudos son los errores de análisis o diseño. Un error en fases tan tempranas dará lugar a un programa que, aunque corre en la máquina, no hace lo que se esperaba de él y, por lo tanto, no funciona. Estos errores obligan a revisar el análisis y el diseño y, en

consecuencia, a rehacer todo el trabajo de especificación, codificación y pruebas. La mejor forma de evitarlos es realizar un análisis y un diseño concienzudos antes de lanzarnos a teclear código como posesos.

Existen varias técnicas, relacionadas con los controles de calidad, para generar software libre de errores y diseñar baterías de prueba que revisen los programas hasta el límite de lo posible, pero que quede claro: ningún programa complejo está libre de errores

al 100% por más esfuerzos que se hayan invertido en ello.

Mantenimiento

Cuando el programa está en uso, y sobre todo si se trata de software comercial, suele ser preciso realizar un mantenimiento. El mantenimiento puede ser de varios tipos: correctivo (para enmendar errores que no se hubieran detectado en la fase de pruebas), perfectivo (para mejorar el rendimiento o añadir más funciones) o adaptativo (para adaptar el programa a otros

entornos).

El coste de la fase de mantenimiento ha experimentado un fuerte incremento en los últimos años. Así, se estima que la mayoría de las empresas de software que dedican alrededor del 60% de sus recursos exclusivamente a mantener el software que ya tienen funcionando, empleando el 40% restante en otras tareas, entre las que se incluye el desarrollo de programas nuevos. Esto es una consecuencia lógica del elevado coste de desarrollo del software.

Nadie es perfecto

Las fases del ciclo de vida que hemos mencionado son las del llamado "ciclo de vida clásico" o en cascada. En teoría, se aplican una detrás de otra. Es decir, primero se hace en análisis y, cuando está completo, se pasa al diseño, luego a la codificación, etc. Esto solo funciona así en un mundo ideal de casitas de caramelo y chocolate. En la práctica, las fases se solapan unas con otras, se vuelve atrás cuando es necesario, o se pasa varias veces por todas ellas

construyendo la aplicación en una espiral creciente. Todo ello da lugar a diferentes tipos de ciclo de vida (con vuelta atrás, en espiral, basado en prototipos, etc), que, a su vez, no dejan de ser construcciones más o menos teóricas que los desarrolladores pueden tomar como referencia pero no seguir a rajatabla.

Lo que queremos decir es que hemos hablado del ciclo de vida clásico para ilustrar cómo el trabajo del programador es solo una parte del desarrollo de un

programa. Planificar y gestionar el ciclo de vida de un proyecto software no es tarea del programador, ni siquiera del analista, sino del jefe de proyecto, de modo que nosotros nos detendremos aquí. Para saber más sobre ciclos de vida del software, tendrás que consultar algún manual de ingeniería del software.

El papel del programador

La figura del programador artesanal que, poseído por una idea feliz repentina, se

lanza a teclear como un poseso y, tras algunas horas de pura inspiración, consigue componer un programa para acceder, digamos, a las bases de datos de la CIA, es, digámoslo claro, pura fantasía romántica. El programador de ordenadores es una pieza más, junto con los analistas, diseñadores, jefes de proyecto, usuarios, controladores de calidad, etc., del complejo engranaje de la ingeniería del software.

Como es lógico, toda la maquinaria de esta ingeniería es excesiva si lo que

pretendemos es realizar programas pequeños y sencillos, del mismo modo que no usamos un helicóptero para ir a comprar el pan a la esquina.

El programador, pues, debe estar capacitado para elaborar programas relativamente sencillos basándose en las especificaciones de los analistas y diseñadores. Esto no quiere decir que un programador no pueda ser, a la vez, analista y diseñador (en realidad, a menudo ejerce varias de estas funciones, dependiendo de su experiencia y

capacidad y de la organización de la empresa en la que trabaje). Sin embargo, en este libro no nos ocuparemos de esas otras actividades y nos centraremos únicamente en las capacidades propias del programador puro. Nuestros programas serán forzosamente de tamaño modesto, aunque al final del camino estaremos en condiciones de escribir programas lo bastante complejos como para atisbar las dificultades que condujeron a la crisis del software de la década de 1970, y empezaremos a notar en nuestras propias

carnes la necesidad de una adecuada planificación previa. Ese será el momento en el que debas dar un salto cualitativo y aprender algo más sobre ingeniería del software.

Estilos de programación

Programación desestructurada

Un programa de ordenador, como hemos dicho, es un conjunto de instrucciones

que el ordenador puede entender y que ejecuta en un determinado orden. Generalmente, el orden de ejecución de las instrucciones es el mismo que el orden en el que el programador las escribió, pero en ocasiones, como veremos, es imprescindible repetir un conjunto de instrucciones varias veces (a esto se le llama técnicamente loop o bucle), o saltar hacia delante o hacia atrás en la lista de instrucciones.

La programación clásica desestructurada utiliza indistintamente bucles y saltos

entremezclados hasta conseguir el correcto funcionamiento del programa. Debido a esto, este tipo de programación es farragosa, confusa, e implica una alta probabilidad de errores. Estos defectos se hacen más patentes cuanto más grande es el programa, llegando a un punto en que el código se hace inmanejable (es lo que se suele denominar *código spaghetti*, una metáfora de lo más afortunada).

Este tipo de programación, con saltos continuos aquí y allá, cayó en desuso

tras la crisis del software de los años 70. Hoy se considera una mala práctica y debe ser evitada.

Programación estructurada

E. W. Dijkstra, de la Universidad de Eindhoven, introdujo este concepto en los años 70 del siglo XX con el fin de eliminar las limitaciones de la programación clásica.

La programación estructurada es una técnica de programación que utiliza una

serie de estructuras específicas que optimizan los recursos lógicos y físicos del ordenador. Estas estructuras (de ahí viene el nombre de programación estructurada) y las reglas de uso que implican las veremos en más adelante y las pondremos en práctica a lo largo de todo el libro.

Programación modular

Esta otra técnica de programación no es excluyente de la anterior, sino que se

pueden utilizar conjuntamente. Es decir, un programa puede ser a la vez modular y estructurado.

La programación modular consiste en dividir un programa complejo en varios programas sencillos que interaccionan de algún modo. Cada programa sencillo se llama módulo. Los módulos deben ser independientes entre sí, es decir, no deben interferir con otros módulos, aunque sí cooperar con ellos en la resolución del problema global. Las técnicas de programación modular

también las estudiaremos un poco más adelante y las aplicaremos repetidamente a lo largo de todo el libro.

Los datos

Como vimos al definir qué es un programa de ordenador, tan importantes son las instrucciones de que consta un programa como los datos que maneja.

Los datos son representaciones de los objetos del mundo real. Por ejemplo, en un programa de gestión de nóminas de

una empresa, existen multitud de datos: los nombres de los empleados, el dinero que ganan, los impuestos que pagan, etc. Cada programa, pues, tiene su propia colección de datos.

Tipos de datos

Se llama tipo de datos a una clase concreta de objetos. Cada tipo de datos, además, tiene asociado un conjunto de operaciones para manipularlos.

Cada tipo de datos dispone de una representación interna diferente en el

ordenador; por eso es importante distinguir entre tipos de datos a la hora de programar.

Tipos de datos simples

Existen tipos de datos simples y tipos complejos. Entre los simples tenemos:

- Números enteros
- Números reales
- Caracteres
- Lógicos

Así, por ejemplo, en el caso del

programa de gestión de nóminas, la edad de los empleados será un dato de tipo número entero, mientras que el dinero que gana al mes será un dato de tipo número real.

Los tipos de datos complejos, también llamados estructuras de datos, los estudiaremos más adelante. Por ahora nos centraremos en los tipos simples. Esto es lógico: hay que empezar por lo más sencillo.

Números enteros

Los datos de tipo entero sólo pueden

tomar como valores:

..., -4, -3, -2, -1, 0, 1, 2, 3, 4, ...

Como el ordenador tiene una memoria finita, la cantidad de valores enteros que puede manejar también es finita y depende del número de bits que emplee para ello (recuerda que el ordenador, internamente, representa todos los datos en binario).

Además, los enteros pueden ser con signo y sin signo. Si tienen signo, se admiten los números negativos; si no lo tienen, los números sólo pueden ser

positivos (sería más correcto llamarlos números naturales).

(Los enteros con signo se almacenan en binario en complemento a uno o en complemento a dos. No vamos a describir estas formas de representación interna, pero, si tienes curiosidad, puedes encontrar abundante información en internet)

Por lo tanto:

- Si se utilizan 8 bits para codificar los números enteros, el rango de valores

permitido irá de 0 a 255 (sin signo) o de -128 a +127 (con signo).

- Si se utilizan 16 bits para codificar los números enteros, el rango será de 0 a 65535 (sin signo) o de -32768 a 32767 (sin signo).
- Si se utilizan 32, 64, 128 bits o más, se pueden manejar números enteros mayores.

Números reales

El tipo de dato número real permite representar números con decimales. La cantidad de decimales de un número real puede ser infinita, pero al ser el ordenador una máquina finita es necesario establecer un número máximo de dígitos decimales significativos.

La representación interna de los números reales se denomina coma flotante (también existe la representación en coma fija, pero no es habitual). La coma flotante es una generalización de la notación científica

convencional, consistente en definir cada número con una mantisa y un exponente.

La notación científica es muy útil para representar números muy grandes economizando esfuerzos. Por ejemplo, el número 129439000000000000000 tiene la siguiente representación científica:

$$1{,}29439 \times 10^{20}$$

Pero el ordenador representaría este número siempre con un 0 a la izquierda de la coma, así:

$$0{,}129439 \times 10^{21}$$

La mantisa es el número situado en la posición decimal (129439) y el exponente es 21.

La notación científica es igualmente útil para números muy pequeños. Por ejemplo, el número 0,00000000000000000000259 tiene esta notación científica:

$$2{,}59 \times 10^{-23}$$

Pero el ordenador lo representará así:

$$0{,}259 \times 10^{-22}$$

Siendo 259 la mantisa y -22 el exponente.

Internamente, el ordenador reserva varios bits para la mantisa y otros más para el exponente. Como en el caso de los números reales, la magnitud de los números que el ordenador pueda manejar estará directamente relacionada con el número de bits reservados para su almacenamiento.

Overflow

Cuando se realizan operaciones con números (tanto enteros como reales), es

posible que el resultado de una de ellas dé lugar a un número fuera del rango máximo permitido. Por ejemplo, si tenemos un dato de tipo entero sin signo de 8 bits cuyo valor sea 250 y le sumamos 10, el resultado es 260, que sobrepasa el valor máximo (255).

En estos casos, estamos ante un caso extremo denominado overflow o desbordamiento. Los ordenadores pueden reaccionar de forma diferente ante este problema, dependiendo del sistema operativo y del lenguaje

utilizado. Algunos lo detectan como un error de ejecución del programa, mientras que otros lo ignoran, convirtiendo el número desbordado a un número dentro del rango permitido pero que, obviamente, no será el resultado correcto de la operación, por lo que el programa probablemente fallará.

Caracteres y cadenas

El tipo de dato carácter sirve para representar datos alfanuméricos. El conjunto de elementos que puede representar está estandarizado según el

código ASCII, que, como ya vimos, consiste en una combinación de 8 bits asociada a un carácter alfanumérico concreto.

Las combinaciones de 8 bits dan lugar a un total de 255 valores distintos (desde 0000 0000 hasta 1111 1111), por lo que esa es la cantidad de caracteres diferentes que se pueden utilizar. Entre los datos de tipo carácter válidos están:

- Las letras minúsculas: 'a', 'b', 'c' ... 'z'
- Las letras mayúsculas: 'A',

'B', 'C' ... 'Z'

- Los dígitos: '1', '2', '3' ...
- Caracteres especiales: '$', '%', '¿', '!' ...

Nótese que no es lo mismo el valor entero 3 que el carácter '3'. Para distinguirlos, usaremos siempre comillas para escribir los caracteres.

Los datos tipo carácter sólo pueden contener UN carácter. Una generalización del tipo carácter es el tipo cadena de caracteres (string, en

inglés), utilizado para representar series de varios caracteres. Éste, sin embargo, es un tipo de datos complejo y será estudiado más adelante. Sin embargo, las cadenas se utilizan tan a menudo que no podremos evitar usarlas en algunos casos antes de estudiarlas a fondo.

Datos lógicos

El tipo dato lógico, también llamado booleano en honor de George Boole, el matemático británico que desarrolló una rama entera del álgebra llamada lógica de Boole, es un dato que sólo puede

tomar un valor entre dos posibles. Esos dos valores son:

- Verdadero (en inglés, true)
- Falso (en inglés, false)

Este tipo de datos se utiliza para representar alternativas del tipo sí/no. En algunos lenguajes, el valor true se representa con el número 1 y el valor false con el número 0. Es decir, los datos lógicos contienen información binaria. Esto ya los hace bastante importantes, pero la mayor utilidad de los datos lógicos viene por otro lado:

son el resultado de todas las operaciones lógicas y relacionales, como veremos en el siguiente epígrafe.

Tipos de datos complejos

Los tipos de datos complejos se componen a partir de agrupaciones de otros datos, ya sean simples o complejos. Por ejemplo, una lista ordenada de números enteros (datos simples) constituyen lo que se llama un vector de números enteros (dato complejo)

Como los datos complejos son muy importantes, dedicaremos a ellos gran parte de este libro (de la tercera parte en adelante). Por ahora, sin embargo, utilizaremos sólo los datos simples hasta que tengamos un dominio suficiente sobre los mecanismos de la programación estructurada.

Operaciones con datos

Como dijimos más atrás, los tipos de datos se caracterizan por la clase de

objeto que representan y por las operaciones que se pueden hacer con ellos. Los datos que participan en una operación se llaman operandos, y el símbolo de la operación se denomina operador. Por ejemplo, en la operación entera 5 + 3, los datos 5 y 3 son los operandos y "+" es el operador.

Podemos clasificar las operaciones básicas con datos en dos grandes grupos: las operaciones aritméticas y las operaciones lógicas.

Operaciones aritméticas

Son análogas a las operaciones matemáticas convencionales, aunque cambian los símbolos. Sólo se emplean con datos de tipo entero o real (aunque puede haber alguna excepción):

Operación	Operador
suma	+
resta	-
multiplicación	*
división entera	div

división	/
módulo (resto)	%
exponenciación	^

No todos los operadores existen en todos los lenguajes de programación. Por ejemplo, en lenguaje Fortran no existe la división entera, en C no existe la exponenciación, y, en Pascal, el operador "%" se escribe "mod".

Señalemos que la división entera (div) se utiliza para dividir números enteros,

proporcionando a su vez como resultado otro número entero, es decir, sin decimales. La operación módulo (%) sirve para calcular el resto de estas divisiones enteras.

El tipo del resultado de cada operación dependerá del tipo de los operandos. Por ejemplo, si sumamos dos números enteros, el resultado será otro número entero. En cambio, si sumamos dos números reales, el resultado será un número real. La suma de un número entero con otro real no está permitida en

muchos lenguajes, así que intentaremos evitarla.

Por último, decir que las operaciones "div" y "%" sólo se pueden hacer con números enteros, no con reales, y que la operación "/" sólo se puede realizar con reales, no con enteros.

Aquí tenemos algunos ejemplos de operaciones aritméticas con números enteros y reales:

Operandos	Operador	Operación	Resu

35 y 9 (enteros)	+	35 + 9	44 (e
35 y 9 (enteros)	–	35 – 9	26 (e
35 y 9 (enteros)	*	35 * 9	315 (
35 y 9 (enteros)	div	35 div 9	3 (en
35 y 9 (enteros)	%	35 % 9	8 (en

35 y 9 (enteros)	^	35 ^ 9	ove
8,5 y 6,75 (reales)	+	8,5 + 6,75	15,25
8,5 y 6,75 (reales)	-	8,5 - 6,75	1,75
8,5 y 6,75 (reales)	*	8,5 * 6,75	57,37
8,5 y 6,75 (reales)	/	8,5 / 6,75	1,259

8,5 y 6,75 (reales)	^	8,5 ^ 6,75	1,877 (re

Nótese que el operador "–" también se usa para preceder a los números negativos, como en el álgebra convencional.

Operaciones lógicas (o booleanas)

Estas operaciones sólo pueden dar como resultado verdadero o falso, es decir, su resultado debe ser un valor lógico.

Hay dos tipos de operadores que se

utilizan en estas operaciones: los operadores de relación y los operadores lógicos.

Operadores de relación

Los operadores de relación son los siguientes:

Operación	Operador
menor que	<
mayor que	>
igual que	==

menor o igual que	<=
mayor o igual que	>=
distinto de	!=

Muchos lenguajes prefieren el símbolo "< >" para "distinto de". En realidad, es un asunto de notación que no tiene mayor importancia.

Los operadores de relación se pueden usar con todos los tipos de datos simples: entero, real, carácter o lógico. El resultado será verdadero si la

relación es cierta, o falso en caso contrario.

Aquí tienes algunos ejemplos:

Operandos	Operador	Operación	Resu
35, 9 (enteros)	>	35 > 9	verd
35, 9 (enteros)	<	35 < 9	fa
35, 9 (enteros)	==	35 == 9	fa

35, 9 (enteros)	!=	35 != 9	verd
5, 5 (enteros)	<	5 < 5	fa
5, 5 (enteros)	<=	5 <= 5	verd
5, 5 (enteros)	!=	5 != 5	fa
"a", "c" (caracteres)	==	'a' = 'c'	fa
"a", "c" (caracteres)	>=	'a' > 'c'	fa

"a", "c" (caracteres)	<=	'a' <= 'c'	verd

En cuanto a los datos lógicos, se considera que "falso" es menor que "verdadero". Por lo tanto:

Operadores lógicos

Los operadores lógicos son and (y), or (o) y not (no). Sólo se pueden emplear con tipos de datos lógicos.

El operador and, que también podemos llamar y, da como resultado verdadero sólo si los dos operandos son

verdaderos:

Operandos	Operador	Operación
verdadero, falso	y	verdadero y fa
falso, verdadero	y	falso y verdad
verdadero, verdadero	y	verdadero y verdadero
falso, falso	y	falso y fals

El operador or (también nos vale o) da como resultado verdadero cuando al menos uno de los dos operandos es verdadero:

Operandos	Operador	Operación
verdadero, falso	o	verdadero o fa
falso, verdadero	o	falso o verdad
verdadero, verdadero	o	verdadero verdadero

falso, falso	o	falso o fals

El operador not (o no) es uno de los escasos operadores que sólo afectan a un operando (operador monario), no a dos (operador binario). El resultado es la negación del valor del operando, es decir, que le cambia el valor de verdadero a falso y viceversa:

Operando	Operador	Operación
verdadero	no	no verdader

falso	no	no falso

Prioridad de los operadores

Es habitual encontrar varias operaciones juntas en una misma línea. En estos casos es imprescindible conocer la prioridad de los operadores, porque las operaciones se calcularán en el orden de prioridad y el resultado puede ser muy distinto del esperado. Por ejemplo, en la operación 6 + 4 / 2, no es lo mismo calcular primero la operación 6 + 4 que calcular primero la operación 4 / 2.

La prioridad de cálculo respeta las reglas generales del álgebra. Así, por ejemplo, la división y la multiplicación tienen más prioridad que la suma o la resta. Pero el resto de prioridades pueden diferir de manera imporante de un lenguaje de programación a otro. Como nosotros vamos a usar C, emplearemos las prioridades de C, que son las siguientes:

Operador	Prioridad
^	máxima

*, /, div, %	
no	
+, -	
<, >, <=, >=	
==, !=	
y	
o	mínima

La prioridad del cálculo se puede

alterar usando paréntesis, como en álgebra. Los paréntesis se pueden anidar tantos niveles como sean necesarios. Por supuesto, a igualdad de prioridad entre dos operadores, la operación se calcula de izquierda a derecha, en el sentido de la lectura de los operandos.

Aquí tenemos algunos ejemplos de operaciones conjuntas y su resultado según el orden de prioridad que hemos visto:

Operación	Resultado

6 + 4 / 2	8
(6 + 4) / 2	5
(33 + 3 * 4) / 5	9
2 ^ 2 * 3	12
3 + 2 * (18 – 4 ^ 2)	7
5 + 3 < 2 + 9	verdadero
2 + 3 < 2 + 4 y 7 > 5	verdadero
"A" > "Z" o 4 / 2 + 4 > 6	falso

"A" > "Z" o 4 / (2 + 2) <= 6	verdadero

Funciones

Además de todas estas operaciones artiméticas, lógicas y relacionales, los lenguajes de programación disponen de mecanismos para realizar operaciones más complejas con los datos, como, por ejemplo, calcular raices cuadradas, logaritmos, senos, cosenos, redondeo de números reales, etc.

Todas estas operaciones (y muchas más) se realizan a través de operadores

especiales llamados funciones de biblioteca. Cuando llegue el momento, ya explicaremos en detalle qué son las funciones de biblioteca, e incluso aprenderemos a hacer las nuestras. Por ahora nos basta saber que sirven para hacer cálculos más complejos y que varían mucho de unos lenguajes a otros, aunque hay cierto número de ellas que estarán disponibles en cualquier lenguaje y que, por lo tanto, podemos usar si las necesitamos.

Las funciones suelen tener al menos un

argumento (pueden tener más de uno), que es el valor sobre el que realizan la operación. Los argumentos se indican entre paréntesis a continuación del nombre de la función.

Estas son algunas de las funciones que encontraremos en todos los lenguajes de programación:

Función	Descripción
abs(x)	valor absoluto de x

sen(x)	seno de x
cos(x)	coseno de x
exp(x)	ex
ln(x)	logaritmo neperiano de x
log10(x)	logaritmo decimal de x

redondeo(x)	redondea el número x al valor entero más próximo
trunc(x)	trunca el número x, es decir, le elimina la parte decimal
raiz(x)	raiz cuadrada de x
cuadrado(x)	x2
aleatorio(x)	genera un número al azar entre 0 y x

Aquí tienes algunos ejemplos de aplicación de estas funciones sobre datos reales:

Operación	Resultado
abs(-5)	5
abs(6)	6
redondeo(5.7)	6
redondeo(5.2)	5

trunc(5.7)	5
trunc(5.2)	5
cuadrado(8)	64
raiz(64)	8

Constantes y variables

Se define un dato constante (o, simplemente, "una constante") como un dato de un programa cuyo valor no

cambia durante la ejecución. Por el contrario, un dato variable (o, simplemente, "una variable") es un dato cuyo valor sí cambia en el transcurso del programa.

Identificadores

A los datos variables se les asigna un identificador alfanumérico, es decir, un nombre. Por lo tanto, es necesario disinguir entre el identificador de una variable y su valor. Por ejemplo, una variable llamada X puede contener el valor 5. En este caso, X es el

identificador y 5 el valor de la variable.

Los identificadores o nombres de variable deben cumplir ciertas reglas que, aunque varían de un lenguaje a otro, podemos resumir en que:

- Deben empezar por una letra y, en general, no contener símbolos especiales excepto el subrayado ("_")
- No deben coincidir con alguna palabra reservada del lenguaje

Identificador	¿Es válido?
x	Sí
5x	No, porque no empieza por
x5	Sí
pepe	sí
_pepe	No, porque no empieza por
pepe_luis	Sí
pepe!luis	No, porque contiene caract

	(!)
raiz	No, porque coincide con la

Las constantes también pueden tener un identificador, aunque no es estrictamente obligatorio. En caso de tenerlo, ha de cumplir las mismas reglas que los identificadores de variable.

Declaración y asignación

Las variables tienen que ser de un tipo de datos determinado, es decir, debemos indicar explícitamente qué tipo de datos

va a almacenar a lo largo del programa. Esto implica que, en algún punto del programa (luego veremos dónde) hay que señalar cual va a ser el identificador de la variable, y qué tipo de datos va a almacenar. A esto se le llama declarar la variable.

Una declaración de variables será algo así:

```
X es entero
Y es real
letra es carácter
```

X, Y y letra son los identificadores de variable. Es necesario declararlas

porque, como vimos, el ordenador maneja internamente cada variable de una forma diferente: en efecto, no es lo mismo una variable entera de 8 bits sin signo que otra real en coma flotante. El ordenador debe saber de antemano qué variables va a usar el programa y de qué tipo son para poder asignarles la memoria necesaria.

Para adjudicar un valor a una variable, se emplea una sentencia de asignación, que tienen esta forma:

```
X = 5
Y = 7.445
```

```
LETRA = 'J'
```

A partir de la asignación, pueden hacerse operaciones con las variables exactamente igual que se harían con datos. Por ejemplo, la operación X + X daría como resultado 10. A lo largo del programa, la misma variable X puede contener otros valores (siempre de tipo entero) y utilizarse para otras operaciones. Por ejemplo:

```
X es entero
Y es entero
Z es entero
X = 8
Y = 2
```

```
Z = X div Y
X = 5
Y = X + Z
```

Después de esta serie de operaciones, realizadas de arriba a abajo, la variable X contendrá el valor 5, la variable Y contendrá el valor 9 y, la variable Z, el valor 4.

En cambio, las constantes no necesitan identificador, ya que son valores que nunca cambian. Esto no significa que no se les pueda asociar un identificador para hacer el programa más legible. En ese caso, sólo se les puede asignar valor

una vez, ya que, por su propia naturaleza, son invariables a lo largo del programa.

Expresiones

Una expresión es una combinación de constantes, variables, operadores y funciones. Es decir, se trata de operaciones aritméticas o lógicas como las que vimos en el apartado anterior, pero en las que, además, pueden aparecer variables.

Por ejemplo:

```
(5 + X) div 2
```

En esta expresión, aparecen dos constantes (5 y 2), una variable (X) y dos operadores (+ y div), además de los paréntesis, que sirven para alterar la prioridad de las operaciones. Lógicamente, para resolver la expresión, es decir, para averiguar su resultado, debemos conocer cuál es el valor de la variable X. Supongamos que la variable X tuviera el valor 7. Entonces, el resultado de la expresión es 6. El cálculo del resultado de una expresión se suele denominar evaluación de la

expresión.

Otro ejemplo:

```
( - b  + raiz(b^2 - 4 * a *
c)) / (2 * a)
```

Esta expresión, más compleja, tiene tres variables (a, b y c), 4 operadores (–, +, ^ y *, aunque algunos aparecen varias veces), 2 constantes (2 y 4, apareciendo el 2 dos veces) y una función (raiz, que calcula la raiz cuadrada). Si el valor de las variables fuera a = 2, c = 3 y b = 4, al evaluar la expresión el resultado sería –0.5

La forma más habitual de encontrar una expresión es combinada con una sentencia de asignación a una variable. Por ejemplo:

```
Y = (5 + X) div 2
```

En estos casos, la expresión (lo que hay a la derecha del signo "=") se evalúa y su resultado es asignado a la variable situada a la izquierda del "=". En el ejemplo anterior, suponiendo que la variable X valiera 7, la expresión (5 + X) div 2 tendría el valor 6, y, por lo tanto, ese es el valor que se asignaría a

la variable Y.

Los algoritmos

Concepto de algoritmo

Para realizar un programa es necesario idear previamente un algoritmo. Esto es importante hasta el extremo de que, sin algoritmo, no existiría el programa.

Un algoritmo es una secuencia ordenada de pasos que conducen a la solución de un problema. Los algoritmos tienen tres

características fundamentales:

- Son precisos, es decir, deben indicar el orden de realización de los pasos.
- Están bien definidos, es decir, si se sigue el algoritmo dos veces usando los mismos datos, debe proporcionar la misma solución.
- Son finitos, esto es, deben completarse en un número determinado de pasos.

Por ejemplo, vamos a diseñar un algoritmo simple que determine si un número N es par o impar:

```
1. Inicio
2. Si N es divisible entre 2,
entonces ES PAR
3. Si N no es divisible entre
2, entonces NO ES PAR
4. Fin
```

Si te fijas bien, este algoritmo cumple las tres condiciones enumeradas anteriormente (precisión, definición y finitud) y resuelve el problema planteado. Lógicamente, al ordenador no le podemos dar estas instrucciones tal y

como las hemos escrito, sino que habrá que expresarlo en un lenguaje de programación, pero esto es algo que trataremos más adelante.

Notación de algoritmos

Los algoritmos deben representarse con algún método que permita independizarlos del lenguaje de programación que luego se vaya a utilizar. Así se podrán traducir más tarde a cualquier lenguaje. En el ejemplo que

acabamos de ver hemos especificado el algoritmo en lenguaje español, pero existen otras formas de representar los algoritmos. Entre todas ellas, destacaremos las siguientes:

- Lenguaje español
- Diagramas de flujo
- Diagramas de Nassi-Schneiderman (NS)
- Pseudocódigo

Nosotros utilizaremos, principalmente, el pseudocódigo y los diagramas de

flujo. El pseudocódigo es un lenguaje de especificación de algoritmos basado en la lengua española que tiene dos propiedades que nos interesarán: facilita considerablemente el aprendizaje de las técnicas de programación y logra que la traducción a un lenguaje de programación real sea casi instantánea. Los diagramas de flujo son representaciones gráficas de los algoritmos que ayudan a comprender su funcionamiento.

Dedicaremos todo el apartado siguiente

a aprender las técnicas básicas de programación usando pseudocódigo y diagramas de flujo, pero, como adelanto, ahí va el algoritmo que determina si un número N es par o impar, escrito en pseudocódigo. Es recomendable que le eches un vistazo para intentar entenderlo y para familiarizarte con la notación en pseudocódigo:

```
algoritmo par_impar
variables
  N es entero
  solución es cadena
inicio
  leer (N)
  si (N div 2 == 0) entonces
```

```
solución = "N es par"
  si_no solución = "N es
impar"
  escribir (solución)
fin
```

Escritura inicial del algoritmo

Una vez superadas las fases de análisis y diseño, es decir, entendido bien el problema y sus datos y descompuesto en problemas más sencillos, llega el momento de resolver cada problema sencillo mediante un algoritmo.

Muchos autores recomiendan escribir una primera versión del algoritmo en

lenguaje natural (en nuestro caso, en castellano), siempre que dicha primera version cumpla dos condiciones:

- Primera: que la solución se exprese como una serie de instrucciones o pasos a seguir para obtener una solución al problema
- Segunda: que las instrucciones haya que ejecutarlas de una en una, es decir, una instrucción cada vez

Por ejemplo, consideremos un problema sencillo: el cálculo del área y del perímetro de un rectángulo. Evidentemente, tenemos que conocer su base y su altura, que designaremos con dos variables de tipo real. Una primera aproximación, en lenguaje natural, podría ser:

```
1. Inicio
2. Preguntar al usuario los
valores de base y altura
3. Calcular el área como área
= base * altura
4. Calcular el perímetro como
perímetro = 2 * base + 2 *
altura
```

```
5. Fin
```

Describir un algoritmo de esta forma puede ser útil si el problema es complicado, ya que puede ayudarnos a entenderlo mejor y a diseñar una solución adecuada. Pero esto sólo es una primera versión que puede refinarse añadiendo cosas. Por ejemplo, ¿qué pasa si la base o la altura son negativas o cero? En tal caso, no tiene sentido averiguar el área o el perímetro. Podríamos considerar esta posibilidad en nuestro algoritmo para hacerlo más completo:

```
1. Inicio
2. Preguntar al usuario los
valores de base y altura
3. Si base es mayor que cero y
altura también, entonces:
   3.1. Calcular el área como
área = base * altura
   3.2. Calcular el perímetro
como perímetro = 2 * base + 2
* altura
4. Si no:
   4.1. No tiene sentido
calcular el área ni el
perímetro
5. Fin
```

Estos refinamientos son habituales en todos los algoritmos y tienen la finalidad de conseguir una solución lo más

general posible, es decir, que pueda funcionar con cualquier valor de "base" y "altura".

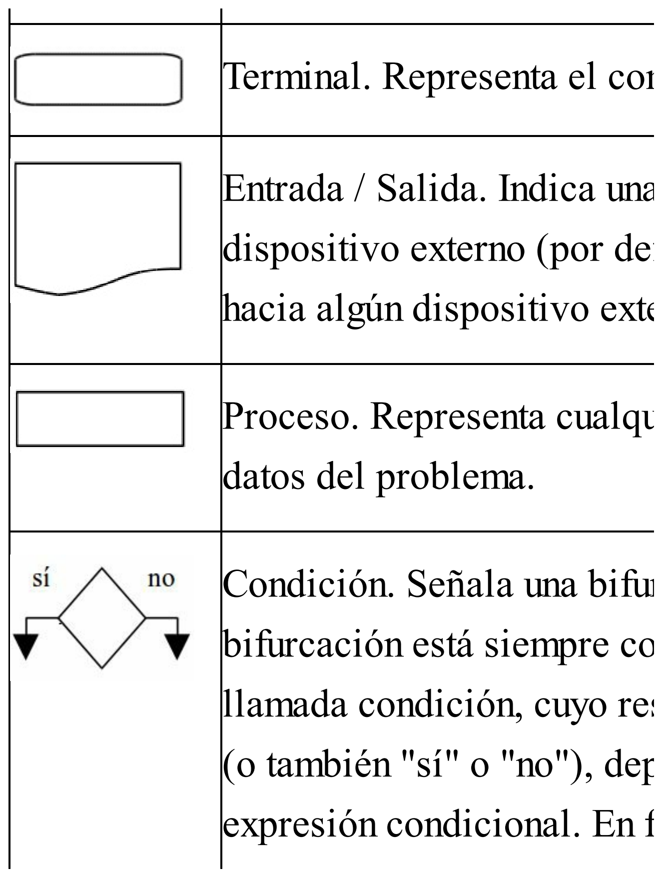
Diagramas de flujo

El diagrama de flujo es una de las técnicas de representación de algoritmos más antigua y también más utilizada, al menos entre principiantes y para algoritmos sencillos. Con la práctica comprobaremos que, cuando se trata de problemas complejos, los diagramas de flujo se hacen demasiado grandes y complicados.

Un diagrama de flujo o flowchart es un gráfico en el que se utilizan símbolos (o cajas) para representar los pasos del algoritmo. Las cajas están unidas entre sí mediante flechas, llamadas líneas de flujo, que indican el orden en el que se deben ejecutar para alcanzar la solución.

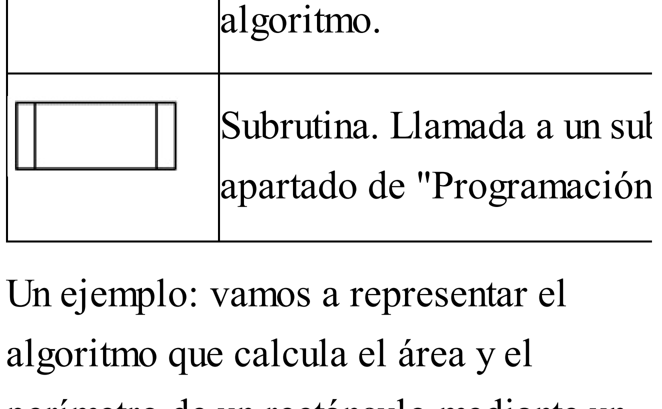
Los símbolos de las cajas están estandarizados y son muy variados. En la tabla siguiente tienes los más habituales, aunque existen algunos otros que no vamos a utilizar.

Símbolo	Función



	Terminal. Representa el co
	Entrada / Salida. Indica una dispositivo externo (por de hacia algún dispositivo ext
	Proceso. Representa cualqu datos del problema.
sí no	Condición. Señala una bifu bifurcación está siempre co llamada condición, cuyo re (o también "sí" o "no"), de expresión condicional. En

	el flujo de ejecución contin las dos a la vez)
	Condición múltiple. Sirve p varias ramas, no sólo en una booleana, sino entera.
	Conector. Para enlazar un fr fragmento situado en la mis muy grande y no puede dibu
	Conector. Como el anterior, diagrama con otro fragment



	Dirección del flujo. Indica algoritmo.
	Subrutina. Llamada a un sub apartado de "Programación

Un ejemplo: vamos a representar el algoritmo que calcula el área y el perímetro de un rectángulo mediante un diagrama de flujo. Antes, tengamos en cuenta que:

- los valores de "base" y "altura" los introducirá el usuario del programa a través

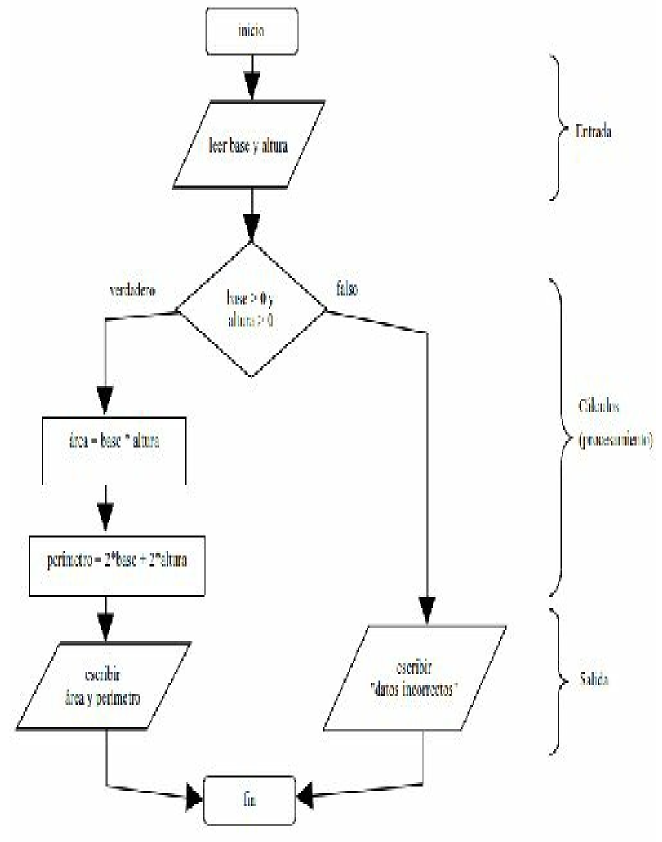
del teclado; así, el programa servirá para cualquier rectángulo

- después se realizarán los cálculos necesarios
- los resultados, "área" y "perímetro", deben mostrarse en un dispositivo de salida (por defecto, la pantalla) para que el usuario del programa vea cuál es la solución

Esta estructura en 3 pasos es muy típica

de todos los algoritmos: primero hay una entrada de datos, luego se hacen cálculos con esos datos, y por último se sacan los resultados.

El diagrama de flujo será más o menos así:


inicio
leer base y altura
Entrada
verdadero
base > 0 y
altura > 0
falso
área = base * altura
perímetro = 2*base + 2*altura
Cálculos
(procesamiento)
escribir
área y perímetro
escribir
"datos incorrectos"
Salida
fin

Pseudocódigo

El pseudocódigo es un lenguaje de descripción de algoritmos. El paso desde el pseudocódigo hasta el lenguaje de programación real (por ejemplo, C), es relativamente fácil. Además, la descripción de algoritmos en pseudocódigo ocupa mucho menos espacio que su equivalente con un diagrama de flujo, por lo que lo preferiremos a la hora de diseñar algoritmos complejos.

El pseudocódigo es bastante parecido a la mayoría de los lenguajes de programación reales, pero no tiene unas reglas tan estrictas, por lo que el programador puede trabajar en la estructura del algoritmo sin preocuparse de las limitaciones del lenguaje final que, como veremos al estudiar C, son muchas y variopintas.

El pseudocódigo utiliza ciertas palabras reservadas para representar las acciones del programa. Estas palabras originalmente están en inglés (y se

parecen mucho a las que luego emplean los lenguajes de programación), pero por suerte para nosotros su traducción española está muy extendida entre la comunidad hispanohablante.

Lista de instrucciones del pseudocódigo

Las instrucciones del pseudocódigo son relativamente pocas, pero, como iremos aprendiendo a lo largo del libro, con un conjunto bastante reducido de instrucciones, correctamente combinadas, podemos construir

programas muy complejos.

A continuación presentamos una tabla-resumen con todas las palabras reservadas del pseudocódigo, y en los siguientes apartados iremos viéndolas una a una.

Instrucción	Significa
`algoritmo nombre`	Marca el adjudica
`inicio`	Marca el instruccio

fin	Marca el instrucci
variables **nombre_var es tipo_de_datos**	Declarac identifica que se va
constantes **nombre_const = expresión**	Declarac expresión asigna a l puede mo programa
	Entrada d

leer (variable)	dato desd no se indi asignando
escribir (variable)	Salida de programa dispositiv otra cosa,
variable = expresión	Asignació resultado
	Instrucció ordenado

si (condición) **entonces** **inicio** ** acciones-1** **fin** **si_no** **inicio** ** acciones-2** **fin**	debe ser verdader es falsa, Instrucci pero care modo qu se realiza continúa
según (expresión) **hacer**	Instrucci utiliza cu condicio falso) . S

inicio **valor1: acciones-1** **valor2: acciones-2** **...** **valor3: acciones-N** **si_no: acciones-si_no** **fin**	suele ser valor en l valorN qu realizánd valor coi Si ningún la expres acciones
mientras (condición) hacer **inicio** **acciones** **fin**	Bucle mie en tanto la expresión condición bloque de

	pueden no

`repetir`
`inicio`
`   acciones`
`fin`
`mientras que`
`(condición)`
	Bucle rep
	en tanto q
	una expre
	parece m
	condición
	por lo qu
	mínimo, u

`para variable desde`
`expr-ini hasta expr-`
`fin hacer`
`inicio`
	Bucle par
	expr-ini,
	se asigna
	variable s

``` acciones fin ```	cada repe acciones variable a

Las instrucciones básicas del pseudocódigo

Hay varias instrucciones de pseudocódigo que son muy simples, así que las vamos a explicar ahora mismo, junto con un ejemplo para ir acostumbrándonos al aspecto de los algoritmos:

- algoritmo: sirve para ponerle

un nombre significativo al algoritmo

- inicio: marca el principio de un proceso, de un módulo o, en general, de un conjunto de instrucciones
- fin: marca el fin de un proceso, módulo o conjunto de instrucciones. "Inicio" y "fin" siempre van por parejas, es decir, cuando aparezca un "Inicio", debe existir un "fin" en algún sitio

más abajo. Y al revés: todo "fin" se corresponde con algún "Inicio" que aparecerá más arriba.

- = (asignación): se utiliza para asociar un valor a una variable, como vimos en el apartado anterior.
- leer: sirve para leer un dato de un dispositivo de entrada (típicamente, el teclado)
- escribir: sirve para enviar un dato a un dispositivo de

salida (si no se indica otra cosa, la pantalla)

Ejemplo: Volvemos al algoritmo del área y el perímetro de un rectángulo:

```
algoritmo rectángulo
inicio
  leer (base)
  leer (altura)
  área = base * altura
  perímetro = 2 * base + 2 *
altura
  escribir (área)
  escribir (perímetro)
fin
```

Recuerda que los programas se ejecutan de arriba a abajo, una instrucción cada

vez.

Cuando este programa se haya introducido en un ordenador y le pidamos que lo ejecute, la máquina irá mirando las instrucciones en el orden en que el programador las introdujo y las irá ejecutando. Veamos, instrucción por instrucción, qué acciones provocan en el ordenador:

- algoritmo rectángulo: simplemente, le pone título al algoritmo y marca su principio (esta instrucción no

hace nada "útil")

- Inicio: marca el comienzo de las instrucciones (por lo tanto, ni esta instrucción ni la anterior realizan ninguna tarea: sólo son marcas)
- leer(base): el ordenador se queda a la espera de que el usuario del programa introduzca algún dato a través del teclado. Cuando el usuario lo hace, ese dato queda almacenado en la

variable "base". Supongamos que el usuario teclea un 7: será como haber hecho la asignación base = 7.

- leer(altura): vuelve a ocurrir lo mismo, pero ahora el dato tecleado se guarda en la variable "altura". Supongamos que se teclea un 2. Por lo tanto, altura = 2.
- área = base * altura: según vimos en el apartado anterior, se evalúa la

expresión situada a la derecha del símbolo "=". El resultado de la misma será 7 * 2, es decir, 14. Ese valor se asigna a la variable situada a la izquierda del "=". Por lo tanto, área = 14.

- perímetro = 2 * base + 2 * altura: en esta ocasión, la evaluación de la expresión da como resultado 18, que se asigna a la variable perímetro, o sea, perímetro =

18.

- escribir(área): el ordenador muestra en la pantalla el valor de la variable área, que es 14.
- escribir(perímetro): el ordenador muestra en la pantalla el valor de la variable perímetro, es decir, 18.
- Fin: marca el punto final del algoritmo

Podemos concluir que el algoritmo presentado resuelve el problema de calcular el área y el perímetro de un rectángulo y posee las tres cualidades básicas de todo algoritmo: precisión, definición y finitud.

Declaración de variables y constantes

Como regla general, diremos que todas las variables deben ser declaradas ANTES de usarse por primera vez. Recuerda que la declaración se usa para comunicar al ordenador el tipo y el identificador de cada variable.

La sintaxis de estas declaraciones es como sigue:

```
variables
   nombre_de_variable  es
tipo_de_datos
```

Ejemplo: Si te fijas en el ejemplo anterior, no hemos declarado ninguna de las variables del algoritmo y, por lo tanto, éste no es del todo correcto. Vamos a completarlo:

```
algoritmo rectángulo
variables
  base es real
  altura es real
  área es real
  perímetro es real
```

```
inicio
  leer (base)
  leer (altura)
  área = base * altura
  perímetro = 2 * base + 2 *
altura
  escribir (área)
  escribir (perímetro)
fin
```

Fíjate que hemos definido las variables antes del inicio de las instrucciones del algoritmo.

A veces, también es útil declarar ciertas constantes para usar valores que no van a cambiar en todo el transcurso del programa. Las constantes se deben

declarar también antes del inicio de las instrucciones del programa.

Ejemplo de declaración de constantes:

```
algoritmo ejemplo
constantes
  pi = 3.141592
  g = 9.8
  txt = "En un lugar de La
Mancha"
inicio
  ...instrucciones...
fin
```

LA PROGRAMACIÓN ESTRUCTURADA

Ya tenemos en nuestra mano los ladrillos fundamentales con los que empezar a construir programas de verdad. El siguiente paso es proporcionarles una estructura para que, al colocarlos unos junto a otros formando un edificio, no se nos vengan abajo.

Teorema de la programación estructurada

El término programación estructurada se refiere a un conjunto de técnicas que han

ido evolucionando desde los primeros trabajos del holandés E. Dijkstra. Pese a ser físico, Dijkstra se convirtió en uno de los más importantes científicos de la computación hasta su muerte en 2002. Una de sus frases más famosas es: “la pregunta de si un computador puede pensar no es más interesante que la pregunta de si un submarino puede nadar”.

Las técnicas que propuso Dijkstra aumentan la productividad del programador, reduciendo el tiempo

requerido para escribir, verificar, depurar y mantener los programas.

Allá por mayo de 1966, Böhm y Jacopini, en un texto clásico sobre programación ("Flow diagrams, turing machines and lenguages only with two formation rules", Communications of the ACM, vol.9, nº 5, pg. 366-371, 1966) demostraron que se puede escribir cualquier programa propio utilizando solo tres tipos de estructuras de control: la secuencial, la selectiva (o condicional) y la repetitiva. A esto se le

llama *Teorema de la programación estructurada*, y define un programa propio como un programa que cumple tres características:

- Posee un sólo punto de inicio y un sólo punto de fin
- Existe al menos un camino que parte del inicio y llega hasta el fin pasando por todas las partes del programa
- No existen bucles infinitos

Realmente, el trabajo de Dijkstra basado

en este teorema fue revolucionario, porque lo que venía a decir es que, para construir programas más potentes y en menos tiempo, lo que había que hacer era simplificar las herramientas que se utilizaban para hacerlos, en lugar de complicarlas más. Este regreso a la simplicidad, unido a las técnicas de ingeniería del software, acabó con la crisis del software de los años 70.

Por lo tanto, los programas estructurados deben limitarse a usar tres estructuras:

- Secuencial
- Selectiva (o condicional)
- Repetitiva

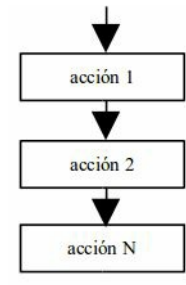
Vamos a estudiar cada estructura detenidamente y veremos cómo se representan mediante diagramas de flujo y pseudocódigo.

Estructura secuencial

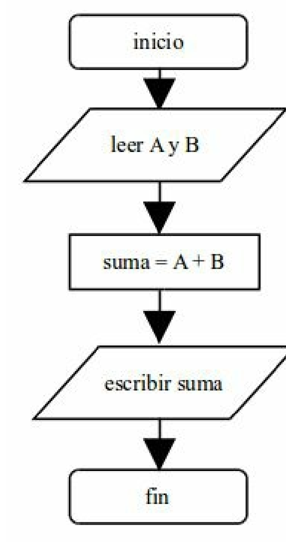
La estructura secuencial es aquélla en la que una acción sigue a otra (en

secuencia). Esta es la estructura algorítmica básica, en la que las instrucciones se ejecutan una tras otra, en el mismo orden en el que fueron escritas.

La estructura secuencial, por lo tanto, es la más simple de las tres estructuras permitidas. A continuación vemos su representación mediante diagrama de flujo y pseudocódigo:



Ejemplo: Vamos a escribir un algoritmo completamente secuencial que calcule la suma de dos números, A y B. Recuerda que, generalmente, los algoritmos se dividen en tres partes: entrada de datos, procesamiento de esos datos y salida de resultados.



Estructuras selectivas (condicionales)

Los algoritmos que usan únicamente estructuras secuenciales están muy limitados y no tienen ninguna utilidad real. Esa utilidad aparece cuando existe la posibilidad de ejecutar una de entre varias secuencias de instrucciones dependiendo de alguna condición asociada a los datos del programa.

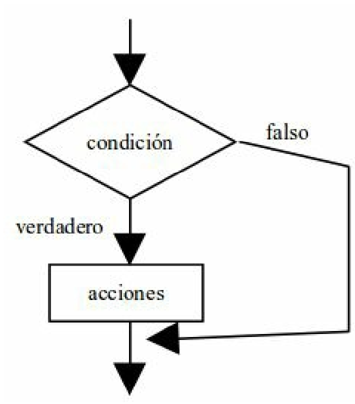
Las estructuras selectivas pueden ser de tres tipos:

- simples
- dobles

- múltiples

Condicional simple

La estructura condicional simple tiene esta representación:



La condición que aparece entre "si" y "entonces" es siempre una expresión lógica, es decir, una expresión cuyo resultado es "verdadero" o "falso". Si el resultado es verdadero, entonces se ejecutan las acciones situadas entre "inicio" y "fin". Si es falso, se saltan las acciones y se prosigue por la siguiente instrucción (lo que haya debajo de "fin")

Ejemplo: Recuperemos algoritmo del área y el perímetro del rectángulo para mostrar la condicional simple en pseudocódigo.

```
algoritmo rectángulo
variables
   base, altura, área,
perímetro son reales
inicio
   leer (base)
   leer (altura)
   si (área > 0) y (altura >
0) entonces
   inicio
      área = base * altura
      perímetro = 2 * base + 2
* altura
      escribir (área)
```

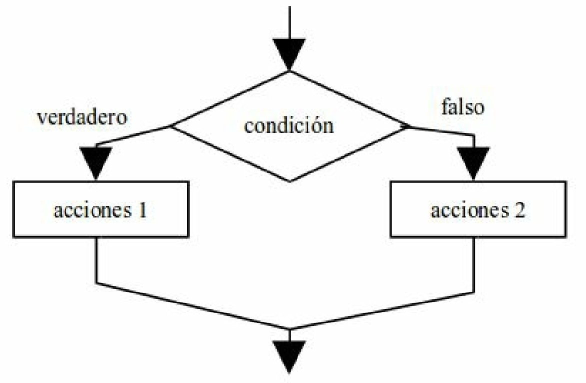
```
        escribir (perímetro)
    fin
    si (área <= 0) o (altura
<=0) entonces
    inicio
        escribir ('Los datos son
incorrectos')
    fin
fin
```

Observa que, en la primera instrucción condicional (si (área > 0) y (altura > 0) entonces) se comprueba que los dos datos sean positivos; en caso de serlo, se procede al cálculo del área y el perímetro mediante las acciones situadas entre inicio y fin. Más abajo

hay otra condicional (si (área <= 0) o (altura <=0) entonces) para el caso de que alguno de los datos sea negativo o cero: en esta ocasión, se imprime en la pantalla un mensaje de error.

Condicional doble

La forma doble de la instrucción condicional es:



En esta forma, la instruccción funciona del siguiente modo: si el resultado de la condición es verdadero, entonces se ejecutan las acciones de la primera parte, es decir, las acciones-1. Si es falso, se ejecutan las acciones de la

parte "si_no", es decir, las acciones-2.

Ejemplo: Podemos reescribir nuestro algoritmo del rectángulo usando una alternativa doble:

```
algoritmo rectángulo
variables
   base, altura, área,
perímetro son reales
inicio
   leer (base)
   leer (altura)
   si (área > 0) y (altura >
0) entonces
    inicio
      área = base * altura
      perímetro = 2 * base + 2
* altura
      escribir (área)
```

```
        escribir (perímetro)
      fin
    si_no
      inicio
        escribir ('Los datos de
entrada son incorrectos')
      fin
fin
```

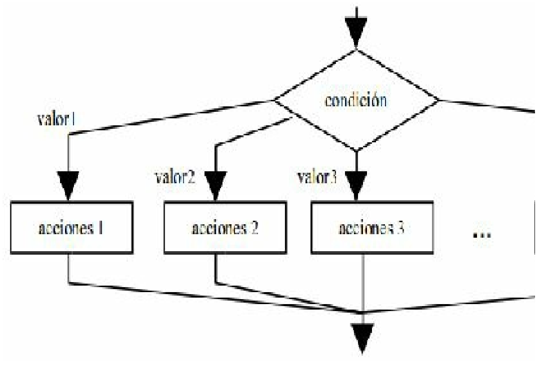
Lo más interesante de este algoritmo es compararlo con el anterior, ya que hace exactamente lo mismo. ¡Siempre hay varias maneras de resolver el mismo problema! Pero esta solución es un poco más sencilla, al ahorrarse la segunda condición, que va implícita en el si_no.

Condicional múltiple

En algunas ocasiones nos encontraremos con selecciones en las que hay más de dos alternativas (es decir, en las que no basta con los valores "verdadero" y "falso"). Siempre es posible plasmar estas selecciones complejas usando varias estructuras si-entonces-si_no anidadas, es decir, unas dentro de otras, pero, cuando el número de alternativas es grande, esta solución puede plantear grandes problemas de escritura y legibilidad del algoritmo.

Sin embargo, hay que dejar clara una cosa: cualquier instrucción condicional múltiple puede ser sustituida por un conjunto de instrucciones condicionales simples y dobles totalmente equivalente.

La estructura condicional múltiple sirve, por tanto, para simplificar estos casos de condiciones con muchas alternativas. Su sintaxis general es:



Su funcionamiento es el siguiente: se evalúa expresión, que en esta ocasión no puede ser de tipo lógico, sino entero, carácter, etc. Sin embargo, no suele admitirse una expresión de tipo real por motivos en los que ahora no nos vamos a

detener. Lo más habitual es que sea de tipo entero. El resultado de expresión se compara con cada uno de los valores valor1, valor2... valorN. Si coincide con alguno de ellas, se ejecutan únicamente las acciones situadas a la derecha del valor coincidente (acciones-1, acciones-2... acciones-N). Si se diera el caso de que ningún valor fuera coincidente, entonces se ejecutan las acciones-si_no ubicadas al final de la estructura. Esta última parte de la estructura no es obligatorio que aparezca.

Ejemplo: Construyamos un algoritmo que escriba los nombres de los días de la semana en función del valor de una variable entera llamada "día". Su valor se introducirá por teclado. Los valores posibles de la variable "día" serán del 1 al 7: cualquier otro valor debe producir un error.

```
algoritmo día_semana
variables
  día es entero
inicio
  leer (día)
  según (día) hacer
  inicio
    1: escribir('lunes')
```

```
      2: escribir('martes')
      3: escribir('miécoles')
      4: escribir('jueves')
      5: escribir('viernes')
      6: escribir('sábado')
      7: escribir('domingo')
      si_no: escribir('Error: el
  día introducido no existe')
    fin
  fin
```

Hay dos cosas interesantes en este algoritmo. Primera, el uso de la instrucción selectiva múltiple: la variable día, una vez leída, se compara con los siete valores posibles. Si vale 1, se realizará la acción escribir('lunes'); si vale 2, se realiza escribir('martes'); y así

sucesivamente. Por último, si no coincide con ninguno de los siete valores, se ejecuta la parte si_no. Es conveniente que pienses cómo se podría resolver el mismo problema sin recurrir a la alternativa múltiple, es decir, utilizando sólo alternativas simples y dobles.

El otro aspecto digno de destacarse no tiene nada que ver con la alternativa múltiple, sino con la sintaxis general de pseudocódigo: no hemos empleado inicio y fin para marcar cada bloque de

instrucciones. Lo más correcto hubiera sido escribirlo así:

```
    según día hacer
    inicio
      1: inicio
           escribir('lunes')
         fin
      2: inicio
           escribir('martes')
         fin
  ..etc..
```

Sin embargo, cuando el bloque de instrucciones consta sólo de UNA instrucción, podemos prescindir de las marcas de inicio y fin y escribir directamente la instrucción.

Estructuras repetitivas (bucles)

Los ordenadores se diseñaron inicialmente para realizar tareas sencillas y repetitivas. El ser humano es de lo más torpe acometiendo tareas repetitivas: pronto le falla la concentración y comienza a tener descuidos. Los ordenadores programables, en cambio, pueden realizar la misma tarea muchas veces por segundo durante años y nunca se aburren (o, al menos, hasta hoy no se ha

tenido constancia de ello)

La estructura repetitiva, por tanto, reside en la naturaleza misma de los ordenadores y consiste, simplemente, en repetir varias veces un conjunto de instrucciones. Las estructuras repetitivas también se llaman bucles, lazos o iteraciones. Nosotros preferiremos la denominación "bucle".

Los bucles tienen que repetir un conjunto de instrucciones un número finito de veces. Si no, nos encontraremos con un bucle infinito y el algoritmo no

funcionará. En rigor, ni siquiera será un algoritmo, ya que no cumplirá la condición de finitud.

El bucle infinito es un peligro que acecha constantemente a los programadores y nos toparemos con él muchas veces a lo largo de nuestra experiencia como programadores. Para conseguir que el bucle se repita sólo un número finito de veces, tiene que existir una condición de salida del mismo, es decir, una situación en la que ya no sea necesario seguir repitiendo las

instrucciones.

Por tanto, los bucles se componen, básicamente, de dos elementos:

- un cuerpo del bucle o conjunto de instrucciones que se ejecutan repetidamente
- una condición de salida para dejar de repetir las instrucciones y continuar con el resto del algoritmo

Dependiendo de dónde se coloque la condición de salida (al principio o al

final del conjunto de instrucciones repetidas), y de la forma de realizarla, existen tres tipos de bucles, aunque hay que resaltar que, con el primer tipo, se puede programar cualquier estructura iterativa. Pero con los otros dos, a veces el programa resulta más claro y legible. Los tres tipos de bucle se denominan:

- Bucle "mientras": la condición de salida está al principio del bucle.
- Bucle "repetir": la condición de salida está al final del

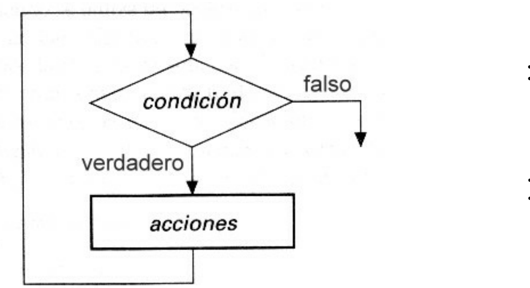
bucle.

- Bucle "para": la condición de salida está al principio y se realiza con un contador automático.

Bucle "mientras"

El bucle "mientras" es una estructura que se repite mientras una condición sea verdadera. La condición, en forma de expresión lógica, se escribe en la cabecera del bucle, y a continuación aparecen las acciones que se repiten

(cuerpo del bucle):



Cuando se llega a una instrucción mientras, se evalúa la condición. Si es verdadera, se realizan las acciones y, al terminar el bloque de acciones, se regresa a la instrucción mientras (he aquí el bucle o lazo). Se vuelve a

evaluar la condición y, si sigue siendo verdadera, vuelve a repetirse el bloque de acciones. Y así, sin parar, hasta que la condición se haga falsa.

Ejemplo: Escribir un algoritmo que muestre en la pantalla todos los números enteros entre 1 y 100

```
algoritmo contar
variables
  cont es entero
inicio
  cont = 0
  mientras (cont <= 100) hacer
  inicio
    cont = cont + 1
    escribir (cont)
```

```
    fin
  fin
```

Aquí observamos el uso de un contador en la condición de salida de un bucle, un elemento muy común en estas estructuras. Observa la evolución del algoritmo:

- cont = 0. Se le asigna el valor 0 a la variable cont (contador)
- mientras (cont <= 100) hacer. Condición de salida del bucle. Es verdadera porque

cont vale 0, y por lo tanto es menor o igual que 100.

- cont = cont + 1. Se incrementa el valor de cont en una unidad. Como valía 0, ahora vale 1.

- escribir(cont). Se escribe el valor de cont, que será 1.

Después, el flujo del programa regresa a la instrucción mientras, ya que estamos en un bucle, y se vuelve a evaluar la condición. Ahora cont vale 1, luego sigue siendo verdadera. Se repiten las

intrucciones del bucle, y cont se incrementa de nuevo, pasando a valer 2. Luego valdrá 3, luego 4, y así sucesivamente.

La condición de salida del bucle hace que éste se repita mientras cont valga menos de 101. De este modo nos aseguramos de escribir todos los números hasta el 100.

Lo más problemático a la hora de diseñar un bucle es, por lo tanto, pensar bien su condición de salida, porque si la condición de salida nunca se hiciera

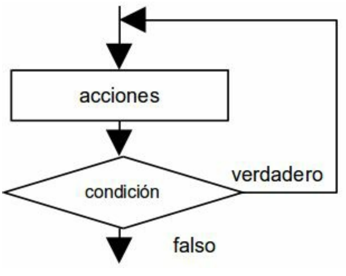
falsa, caeríamos en un bucle infinito. Por lo tanto, la variable implicada en la condición de salida debe sufrir alguna modificación en el interior del bucle; si no, la condición siempre sería verdadera. En nuestro ejemplo, la variable cont se modifica en el interior del bucle: por eso llega un momento, después de 100 repeticiones, en el que la condición se hace falsa y el bucle termina.

Bucle "repetir"

El bucle de tipo "repetir" es muy similar

al bucle "mientras", con la salvedad de que la condición de salida se evalúa al final del bucle, y no al principio, como a continuación veremos. Todo bucle "repetir" puede escribirse como un bucle "mientras", pero al revés no siempre sucede.

La forma de la estructura "repetir" es la que sigue:



Cuando el ordenador encuentra un bucle de este tipo, ejecuta las acciones escritas entre inicio y fin y, después, evalúa la condición, que debe ser de tipo lógico. Si el resultado es falso, se vuelven a repetir las acciones. Si el resultado es verdadero, el bucle se repite. Si es falso, se sale del bucle y se continúa ejecutando la siguiente

instrucción.

Existe, pues, una diferencia fundamental con respecto al bucle "mientras": la condición se evalúa al final. Por lo tanto, las acciones del cuerpo de un bucle "repetir" se ejecutan al menos una vez, cuando en un bucle "mientras" es posible que no se ejecuten ninguna (si la condición de salida es falsa desde el principio)

Ejemplo: Diseñar un algoritmo que escriba todos los números enteros entre 1 y 100, pero esta vez utilizando un

bucle "repetir" en lugar de un bucle "mientras"

```
algoritmo contar
variables
  cont es entero
inicio
  cont = 0
  repetir
    inicio
      cont = cont + 1
      escribir (cont)
    fin
  mientras que (cont <= 100)
fin
```

Observa que el algoritmo es básicamente el mismo que en el ejemplo anterior, pero hemos cambiado el lugar

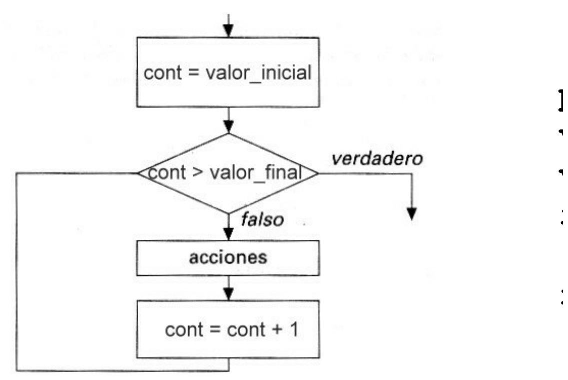
de la condición de salida.

Bucle "para"

En muchas ocasiones se conoce de antemano el número de veces que se desean ejecutar las acciones del cuerpo del bucle. Cuando el número de repeticiones es fijo, lo más cómodo es usar un bucle "para", aunque sería perfectamente posible sustituirlo por uno "mientras".

La estructura "para" repite las acciones del bucle un número prefijado de veces

e incrementa automáticamente una variable contador en cada repetición. Su forma general es:



cont es la variable contador. La primera vez que se ejecutan las acciones situadas entre inicio y fin, la variable cont tiene

el valor especificado en la expresión valor_inicial. En la siguiente repetición, cont se incrementa en una unidad, y así sucesivamente, hasta alcanzar el valor_final. Cuando esto ocurre, el bucle se ejecuta por última vez y después el programa continúa por la instrucción que haya a continuación.

El incremento de la variable cont siempre es de 1 en cada repetición del bucle, salvo que se indique otra cosa. Por esta razón, la estructura "para " tiene una sintaxis alternativa:

```
para cont desde valor_inicial
hasta valor_final inc | dec
paso hacer
inicio
   acciones
fin
```

De esta forma, se puede especificar si la variable cont debe incrementarse (inc) o decrementarse (dec) en cada repetición, y en qué cantidad (paso).

Ejemplo 1: Diseñar un algoritmo que escriba todos los números enteros entre 1 y 100, utilizando un bucle "para"

```
algoritmo contar
variables
  cont es entero
```

```
inicio
  para cont desde 1 hasta 100
hacer
  inicio
    escribir (cont)
  fin
fin
```

De nuevo, lo más interesante es observar las diferencias de este algoritmo con los dos ejemplos anteriores. Advierte que ahora no es necesario asignar un valor inicial de 0 a cont, ya que se hace implícitamente en el mismo bucle; y tampoco es necesario incrementar el valor de cont en el cuerpo del bucle (cont = cont + 1), ya

que de eso se encarga el propio bucle "para". Por último, no hay que escribir condición de salida, ya que el bucle "para" se repite hasta que cont vale 100 (inclusive)

Ejemplo 2: Diseñar un algoritmo que escriba todos los números enteros impares entre 1 y 100, utilizando un bucle "para"

```
algoritmo contar
variables
  cont es entero
inicio
  para cont desde 1 hasta 100
inc 2 hacer
```

```
    inicio
      escribir (cont)
    fin
  fin
```

Este ejemplo, similar al anterior, sirve para ver el uso de la sintaxis anternativa del bucle "para". La variable cont se incrementará en 2 unidades en cada repetición del bucle.

Contadores, acumuladores, conmutadores

Asociadas a los bucles se encuentran a

menudo algunas variables auxiliares. Como siempre se utilizan de la misma manera, las llamamos con un nombre propio (contador, acumulador, etc.), pero hay que dejar claro que no son más que variables comunes, aunque se usan de un modo especial.

Contadores

Un contador es una variable (casi siempre de tipo entero) cuyo valor se incrementa o decrementa en cada repetición de un bucle. Es habitual llamar a esta variable "cont" (de

contador) o "i" (de índice).

El contador suele usarse de este modo:

Primero, se inicializa antes de que comience el bucle. Es decir, se le da un valor inicial. Por ejemplo:

```
cont = 5
```

Segundo, se modifica dentro del cuerpo del bucle. Lo más habitual es que se incremente su valor en una unidad. Por ejemplo:

```
cont = cont + 1
```

Esto quiere decir que el valor de la

variable "cont" se incrementa en una unidad y es asignado de nuevo a la variable contador. Es decir, si cont valía 5 antes de esta instrucción, cont valdrá 6 después.

Otra forma típica del contador es:

```
cont = cont - 1
```

En este caso, la variable se decrementa en una unidad; si cont valía 5 antes de la instrucción, tendremos que cont valdrá 4 después de su ejecución.

El incremento o decremento no tiene por qué ser de una unidad. La cantidad que

haya que incrementar o decrementar vendrá dada por la naturaleza del problema.

Y, tercero, se utiliza en la condición de salida del bucle. Normalmente, se compara con el valor máximo (o mínimo) que debe alcanzar el contador para dejar de repetir las instrucciones del bucle.

Ejemplo: Escribir un algoritmo que escriba la tabla de multiplicar hasta el 100 de un número N introducido por el usuario

```
algoritmo tabla_multiplicar
variables
  cont es entero
  N es entero
inicio
  leer (N)
  cont = 1
  mientras (cont <= 100) hacer
  inicio
    escribir (N * cont)
    cont = cont + 1
  fin
fin
```

El uso de contadores es casi obligado en bucles "mientras" y "repetir" que deben ejecutarse un determinado número de veces. Recuerda que siempre hay que asignar al contador un valor inicial para

la primera ejecución del bucle (cont = 1 en nuestro ejemplo) e ir incrementándolo (o decrementándolo, según el algoritmo) en cada repetición con una instrucción del tipo cont = cont + 1 en el cuerpo del bucle. De lo contrario habremos escrito un bucle infinito.

Por último, hay que prestar atención a la condición de salida, que debe estar asociada al valor del contador en la última repetición del bucle (en nuestro caso, 100). Mucho cuidado con el

operador relacional (<, >, <=, >=, etc) que usemos, porque el bucle se puede ejecutar más o menos veces de lo previsto. En general, suele ser buena idea evitar el operador "==" en las condiciones de salida de los bucles si estamos trabajando con números reales: la precisión limitada de estos números puede hacer que dos variables que deberían valer lo mismo difieran en realidad en una cantidad infinitesimal y, por tanto, la condición "==" nunca se haga verdadera.

Acumuladores

Las variables acumuladoras tienen la misión de almacenar resultados sucesivos, es decir, de acumular resultados, de ahí su nombre.

Las variables acumuladores también debe ser inicializadas. Si llamamos "acum" a un acumulador, escribiremos antes de iniciar el bucle algo como esto:

```
acum = 0
```

Por supuesto, el valor inicial puede cambiar, dependiendo de la naturaleza del problema. Más tarde, en el cuerpo

del bucle, la forma en la que nos la solemos encontrar es:

```
acum = acum + N
```

...siendo N otra variable. Si esta instrucción va seguida de otras:

```
acum = acum + M
acum = acum + P
```

... estaremos acumulando en la variable "acum" los valores de las variables M, N, P, etc, lo cual resulta a veces muy útil para resolver ciertos problemas repetitivos.

Ejemplo: Escribir un algoritmo que

pida 10 números por el teclado y los sume, escribiendo el resultado

```
algoritmo sumar10
variables
  cont es entero
  suma es entero
  N es entero
inicio
  suma = 0
  para cont desde 1 hasta 10
hacer
  inicio
    leer (N)
    suma = suma + N
  fin
  escribir (suma)
fin
```

En este algoritmo, cont es una variable

contador típica de bucle. Se ha usado un bucle "para", que es lo más sencillo cuando conocemos previamente el número de repeticiones (10 en este caso). La variable N se usa para cada uno de los números introducidos por el teclado, y la variable suma es el acumulador, donde se van sumando los diferentes valores que toma N en cada repetición.

Observa como, al principio del algoritmo, se le asigna al acumulador el valor 0. Esta es una precaución

importante que se debe tomar siempre porque el valor que tenga una variable que no haya sido usada antes es desconocido (no tiene por qué ser 0)

Conmutadores

Un conmutador (o interruptor) es una variable que sólo puede tomar dos valores. Pueden ser, por tanto, de tipo booleano, aunque también pueden usarse variables enteras o de tipo carácter.

La variable conmutador recibirá uno de los dos valores posibles antes de entrar

en el bucle. Dentro del cuerpo del bucle, debe cambiarse ese valor bajo ciertas condiciones. Utilizando el conmuntador en la condición de salida del bucle, puede controlarse el número de repeticiones.

Ejemplo: Escribir un algoritmo que sume todos los números positivos introducidos por el usuario a través del teclado. Para terminar de introducir números, el usuario tecleará un número negativo.

```
algoritmo sumar
variables
```

```
  suma es entero
  N es entero
  terminar es lógico
inicio
  suma = 0
  terminar = falso
  mientras (terminar == falso)
    inicio
      escribir ('Introduce un
número (negativo para
terminar)')
      leer (N)
      si (N >= 0) entonces
         suma = suma + N
      si_no
         terminar = verdadero
    fin
  fin
  escribir (suma)
fin
```

Este algoritmo es una variación del ejemplo con acumuladores que vimos en el apartado anterior. Entonces el usuario introducía 10 números, y ahora puede ir introduciendo números indefinidamente, hasta que se canse. ¿Cómo indica al ordenador que ha terminado de introducir números? Simplemente, tecleando un número negativo.

El bucle se controla por medio de la variable "terminar": es el conmutador. Sólo puede tomar dos valores: "verdadero", cuando el bucle debe

terminar, y "falso", cuando el bucle debe repetirse una vez más. Por lo tanto, "terminar" valdrá "falso" al principio, y sólo cambiará a "verdadero" cuando el usuario introduzca un número negativo.

A veces, el conmutador puede tomar más de dos valores. Entonces ya no se le debe llamar, estrictamente hablando, conmutador. Cuando la variable toma un determinado valor especial, el bucle termina. A ese "valor especial" se le suele denominar valor centinela.

Ejemplo: Escribir un algoritmo que

sume todos los números positivos introducidos por el usuario a través del teclado. Para terminar de introducir números, el usuario tecleará un número negativo.

```
algoritmo sumar
variables
  suma es entero
  N es entero
inicio
  suma = 0
  repetir
    inicio
      escribir ('Introduce un
número (negativo para
terminar)')
      leer (N)
      si (N >= 0) entonces
```

```
            suma = suma + N
      fin
    mientras que (N >= 0)
    escribir (suma)
  fin
```

Tenemos aquí un ejemplo de cómo no siempre es necesario usar contadores para terminar un bucle "mientras" (o "repetir"). Las repeticiones se controlan con la variable N, de modo que el bucle termina cuando N < 0. Ese es el valor centinela.

PROGRAMACIÓN MODULAR

Podemos definir la programación modular como aquélla que afronta la solución de un problema descomponiéndolo en subproblemas más simples, cada uno de los cuales se resuelve mediante un algoritmo o módulo más o menos independiente del resto (de ahí su nombre: "programación modular")

Las ventajas de la programación modular son varias:

- Facilita la comprensión del problema y su resolución

escalonada

- Aumenta la claridad y legibilidad de los programas
- Permite que varios programadores trabajen en el mismo problema a la vez, puesto que cada uno puede trabajar en uno o varios módulos de manera bastante independiente
- Reduce el tiempo de desarrollo, reutilizando módulos previamente

desarrollados

- Mejora la fiabilidad de los programas, porque es más sencillo diseñar y depurar módulos pequeños que programas enormes
- Facilita el mantenimiento de los programas

Resumiendo, podemos afirmar sin temor a equivocarnos que es virtualmente imposible escribir un programa de grandes dimensiones si no procedemos a dividirlo en fragmentos más pequeños,

abarcables por nuestro pobre intelecto humano.

Recuérdese que la programación modular y la estructurada no son técnicas incompatibles, sino más bien complementarias. Todos los programas que desarrollemos de ahora en adelante serán, de hecho, al mismo tiempo modulares y estructurados.

Pero expliquemos más despacio que es eso de "descomponer un problema en subproblemas simples..."

Descomposición modular: ¡divide y vencerás!

La forma más habitual de diseñar algoritmos para resolver problemas de cierta envergadura se suele denominar, muy certeramente, divide y vencerás (en inglés, divide and conquer o simplemente DAC). Fíjate que hemos dicho "diseñar" algoritmos: estamos adentrándonos, al menos en parte, en la fase de diseño del ciclo de vida del software. Si no recuerdas lo que es,

repasa el apartado correspondiente.

El método DAC consiste en dividir un problema complejo en subproblemas, y tratar cada subproblema del mismo modo, es decir, dividiéndolo a su vez en subproblemas. Así sucesivamente hasta que obtengamos problemas lo suficientemente sencillos como para escribir algoritmos que los resuelvan. Dicho de otro modo: problemas que se parezcan en complejidad a los que hemos venido resolviendo hasta ahora. Llamaremos módulo a cada uno de estos

algoritmos que resuelven los problemas sencillos.

Una vez resueltos todos los subproblemas, es decir, escritos todos los módulos, es necesario combinar de algún modo las soluciones para generar la solución global del problema.

Esta forma de diseñar una solución se denomina diseño descendente o top-down. No es la única técnica de diseño que existe, pero sí la más utilizada. Resumiendo lo dicho hasta ahora, el diseño descendente debe tener dos

fases:

- La identificación de los subproblemas más simples y la construcción de algoritmos que los resuelvan (módulos)
- La combinación de las soluciones de esos algoritmos para dar lugar a la solución global

La mayoría de lenguajes de programación, incluyendo por supuesto a C, permiten aplicar técnicas de diseño descendente mediante un proceso muy

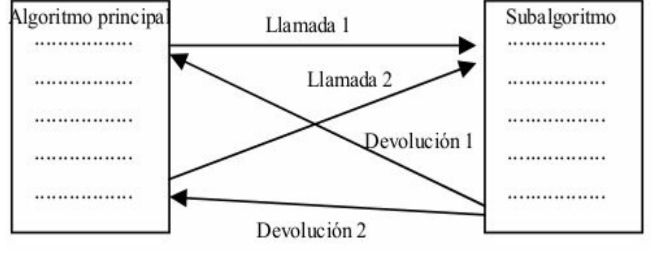
simple: independizando fragmentos de código en subprogramas o módulos denominados procedimientos y funciones, que más adelante analizaremos en profundidad.

Algoritmo principal y subalgoritmos

En general, el problema principal se resuelve en un algoritmo que denominaremos algoritmo o módulo principal, mientras que los subproblemas sencillos se resolverán en subalgoritmos, también llamados módulos a secas. Los subalgoritmos

están subordinados al algoritmo principal, de manera que éste es el que decide cuándo debe ejecutarse cada subalgoritmo y con qué conjunto de datos.

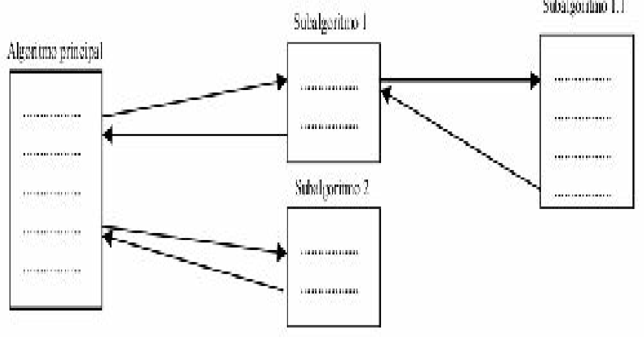
El algoritmo principal realiza llamadas o invocaciones a los subalgoritmos, mientras que éstos devuelven resultados a aquél. Así, el algoritmo principal va recogiendo todos los resultados y puede generar la solución al problema global.



Cuando el algoritmo principal hace una llamada al subalgoritmo (es decir, lo invoca), se empiezan a ejecutar las instrucciones del subalgoritmo. Cuando éste termina, devuelve los datos de salida al algoritmo principal, y la ejecución continúa por la instrucción siguiente a la de invocación. También se dice que el subalgoritmo devuelve el

control al algoritmo principal, ya que éste toma de nuevo el control del flujo de instrucciones después de habérselo cedido temporalmente al subalgoritmo.

El programa principal puede invocar a cada subalgoritmo el número de veces que sea necesario. A su vez, cada subalgoritmo puede invocar a otros subalgoritmos, y éstos a otros, etc. Cada subalgoritmo devolverá los datos y el control al algoritmo que lo invocó.



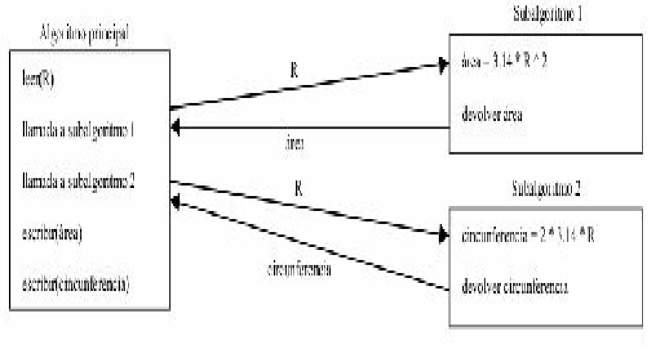
Los subalgoritmos pueden hacer las mismas operaciones que los algoritmos, es decir: entrada de datos, proceso de datos y salida de datos. La diferencia es que los datos de entrada se los proporciona el algoritmo que lo invoca, y los datos de salida son devueltos

también a él para que haga con ellos lo que considere oportuno. No obstante, un subalgoritmo también puede, si lo necesita, tomar datos de entrada desde el teclado (o desde cualquier otro dispositivo de entrada) y enviar datos de salida a la pantalla (o a cualquier otro dispositivo de salida).

Ejemplo: Diseñar un algoritmo que calcule el área y la circunferencia de un círculo cuyo radio se lea por teclado. Se trata de un problema muy simple que puede resolverse sin aplicar el método

divide y vencerás, pero lo utilizaremos como ilustración.

Dividiremos el problema en dos subproblemas más simples: por un lado, el cálculo del área, y, por otro, el cálculo de la circinferencia. Cada subproblema será resuelto en un subalgoritmo, que se invocará desde el algoritmo principal. La descomposición en algoritmos y subalgoritmos sería la siguiente (se indican sobre las flechas los datos que son interrcambiados entre los módulos):



Lógicamente, los subalgoritmos deben tener asignado un nombre para que puedan ser invocados desde el algoritmo principal, y también existe un mecanismo concreto de invocación/devolución. Todo esto lo veremos con detalle en los siguientes

epígrafes.

Técnicas de descomposición modular

Nivel de descomposición modular

Los problema complejos, como venimos diciendo, se descomponen sucesivamente en subproblemas más simples cuya solución combinada dé lugar a la solución general. Pero, ¿hasta dónde es necesario descomponer? O, dicho de otro modo, ¿qué se puede considerar un “problema simple” y qué no?

La respuesta se deja al sentido común y a la experiencia del diseñador del programa. Como regla general, digamos que un módulo no debería constar de más de 30 ó 40 líneas de código. Si obtenemos un módulo que necesita más código para resolver un problema, probablemente podamos dividirlo en dos o más subproblemas. Por supuesto, esto no es una regla matemática aplicable a todos los casos. En muchas ocasiones no estaremos seguros de qué debe incluirse y qué no debe incluirse en un módulo.

Tampoco es conveniente que los módulos sean excesivamente sencillos. Programar módulos de 2 ó 3 líneas daría lugar a una descomposición excesiva del problema, aunque habrá ocasiones en las que sea útil emplear módulos de este tamaño.

Diagrama de estructura modular

La estructura modular, es decir, el conjunto de módulos de un programa y la forma en que se invocan unos a otros, se puede representar gráficamente mediante un diagrama de estructura

modular. Estos diagramas se usan profusamente en la etapa de diseño del ciclo de vida. Esto es particularmente útil si el programa es complejo y consta de muchos módulos con relaciones complicadas entre sí. Nosotros, que no pretendemos convertirnos en diseñadores, los emplearemos sólo cuando nos enfrentemos con diseños modulares complejos. También se denominan Diagramas de Cuadros de Constantine, debido al clásico libro sobre diseño estructurado: CONSTANTINE, L.; YOURDON, E.,

Structured design: fundamentals of a discipline of computer programs and system design, Prentice-Hall, 1979.

En el diagrama se representan los módulos mediante cajas, en cuyo interior figura el nombre del módulo, unidos por líneas, que representan las interconexiones entre ellos. En cada línea se pueden escribir los parámetros de invocación y los datos devueltos por el módulo invocado.

El diagrama de estructura siempre tiene forma de árbol invertido. En la raíz

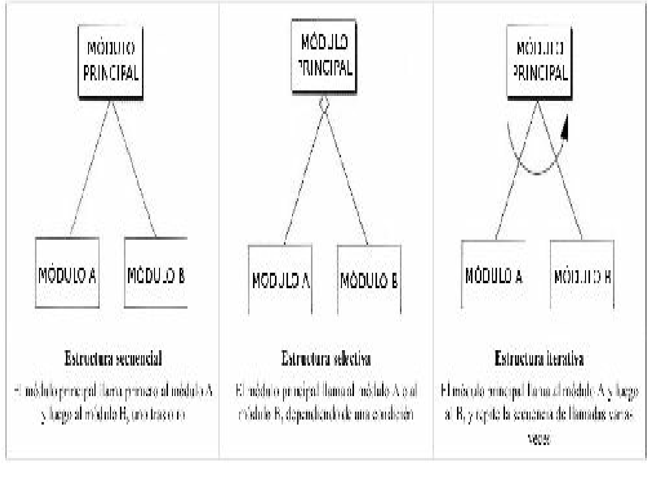
figura el módulo principal, y de él “cuelgan” el resto de módulos en uno o varios niveles.

En el diagrama también se puede representar el tipo de relación entre los módulos. Las relaciones posibles se corresponden exactamente con los tres tipos de estructuras básicas de la programación estructurada:

- Estructura secuencial: cuando un módulo llama a otro, después a otro, después a otro, etc.

- Estructura selectiva: cuando un módulo llama a uno o a otro dependiendo de alguna condición
- Estructura iterativa: cuando un módulo llama a otro (o a otros) en repetidas ocasiones

Las tres estructuras de llamadas entre módulos se representan con tres símbolos diferentes:

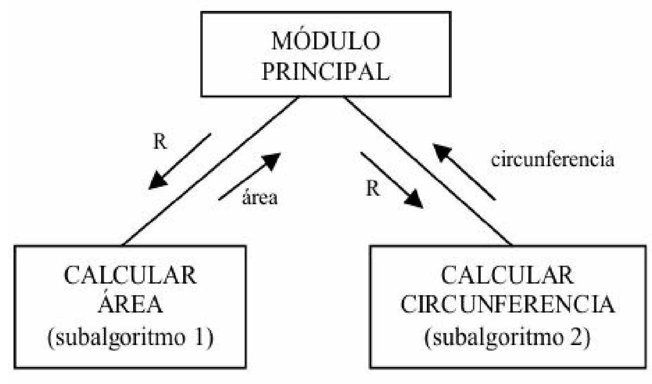


Ejemplo: Diagrama de estructura del algoritmo que calcula el área y la circunferencia de un círculo, que vimos como ejemplo unas páginas más atrás. La descomposición modular que hicimos en aquel ejemplo consistía en un

algoritmo principal que llamaba a dos subalgoritmos: uno para calcular el área y otro para calcular la circunferencia.

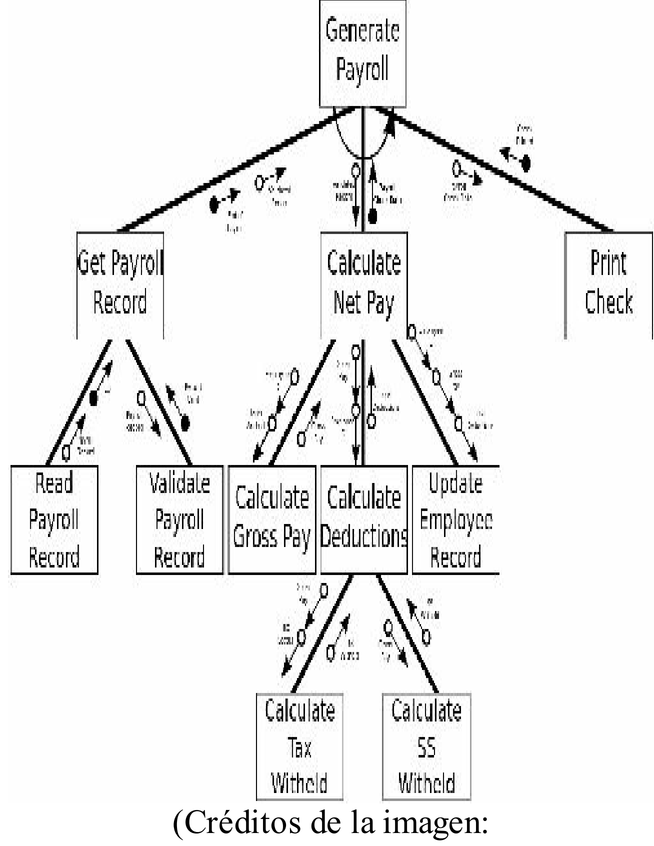
Los dos subalgoritmos (o módulos) son llamados en secuencia, es decir, uno tras otro, por lo que lo representaremos con la estructura secuencial. El módulo principal pasará a los dos subalgoritmos el radio (R) del círculo, y cada subalgoritmo devolverá al módulo principal el resultado de sus cálculos. Observa que ese trasiego de información también puede representarse en el

diagrama de estructura:



Es este otro ejemplo podemos ver un caso más elaborado de diagrama de estructura. Se trata de un ejemplo real (modificado) de dominio público. Solo te lo mostramos con el propósito de que conozcas el aspecto que tiene un

diagrama de estructura real:



(Créditos de la imagen:

Structured_Chart_Example.jpg: Sandia National Laboratoriesderivative work: Pluke (talk) - Structured_Chart_Example.jpg, Public Domain, https://commons.wikimedia.org/w/index.php?curid=16283788)

Escritura del programa

Una vez diseñada la estructura modular, llega el momento de escribir los algoritmos y subalgoritmos mediante pseudocódigo, diagramas de flujo o cualquier otra herramienta. Lo más conveniente es comenzar por los módulos (subalgoritmos) de nivel inferior e ir ascendiendo por cada rama

del diagrama de estructura.

Lo dicho hasta ahora respecto de algoritmos y subalgoritmos se puede traducir en programas y subprogramas cuando pasemos del pseudocódigo a lenguajes de programación concretos, como C. Los subprogramas, en general, se pueden dividir en dos tipos, muy similares pero con alguna sutil diferencia: las funciones y los procedimientos, que estudiaremos en los siguientes epígrafes del tema.

Funciones

Las funciones son subalgoritmos (o módulos) que resuelven un problema sencillo y devuelven un resultado al algoritmo que las invoca.

Las funciones pueden tener argumentos, aunque no es obligatorio. Los argumentos son los datos que se proporcionan a la función en la invocación, y que la función utilizará para sus cálculos.

Además, las funciones tienen, obligatoriamente, que devolver un

resultado. Este resultado suele almacenarse en una variable para usarlo posteriormente.

Ejemplo: Paso de parámetros a funciones de biblioteca. Cuando utilizamos las funciones matemáticas de biblioteca, siempre escribimos algún dato entre paréntesis para que la función realice sus cálculos con ese dato. Pues bien, ese dato es el argumento o parámetro de entrada:

```
A = raiz(X)
B = redondeo(7.8)
N = aleatorio(100)
```

En estas tres instrucciones de asignación, se invoca a las funciones raiz(), redondeo()y aleatorio(), pasándoles los argumentos X y 7.8. Recuerda que estas son funciones que los lenguajes de programación incorporan por defecto, junto con muchas otras que iremos descubriendo con el uso.

Ambas funciones devuelven un resultado; el resultado de la función raiz() se almacena en la variable A, el de redondeo() en la variable B y el de la

función aleatorio() en la variable N.

Declaración de funciones

No sólo de funciones de biblioteca vive el programador. Como es lógico, también podemos crear nuestras propias funciones para invocarlas cuando nos sea necesario.

Recuerda que una función no es más que un módulo, es decir, un subalgoritmo que depende, directamente o a través de otro subalgoritmo, del algoritmo principal. Por tanto, su estructura debe ser similar

a la de los algoritmos que hemos manejado hasta ahora.

La sintaxis en pseudocódigo de una función es:

```
tipo_resultado función
nombre_función(lista_de_argumen
constantes
   lista_de_constantes
variables
   lista_de_variables
inicio
   acciones
   devolver (expresión)
fin
```

Observa que es exactamente igual que los algoritmos que conoces, excepto por

la primera línea, que ya no contiene la palabra "algoritmo" e incluye algunos elementos nuevos:

- El tipo_resultado es el tipo de datos del resultado que devuelve la función
- El nombre_función es el identificador de la función
- La lista_de_argumentos es una lista con los parámetros que se le pasan a la función

También aparece una nueva sentencia,

devolver(expresión), justo al final de la función. La expresión se evalúa y su resultado es devuelto al algoritmo que invocó a la función. El tipo de la expresión debe coincidir con el de tipo_resultado.

De todos estos elementos nuevos, el más complejo con diferencia es la lista de argumentos, ya que pueden existir argumentos de entrada, de salida y de entrada/salida. El problema de los argumentos lo trataremos en profundidad un poco más adelante, cuando ya nos

hayamos acostumbrado al uso de funciones. Por ahora, diremos que es una lista de esta forma:

```
parámetro_1 es
tipo_de_datos_1, parámetro_2
es tipo_de_datos_2, etc.
```

Ejemplo: Declaración de una función que calcule el área de un círculo. El radio se pasa como argumento de tipo real.

```
real función área_círculo
(radio es real)
variables
   área es real
inicio
   área = 3.14 * radio ^ 2
```

```
    devolver (área)
  fin
```

Fíjate en que la función no es más que un algoritmo normal y corriente, salvo por dos detalles:

- La primera línea. En ella aparece más información: el tipo de valor devuelto por la función (real, puesto que calcula el área del círculo), el nombre de la función (área_círculo) y la lista de argumentos. En esta función sólo hay un argumento,

llamado radio. Es de tipo real.

- La penúltima línea (antes de fin). Contiene el valor que la función devuelve. Debe ser una expresión del mismo tipo que se indicó en la primera línea (en este ejemplo, real).

Invocación de funciones

Para que las instrucciones escritas en una función sean ejecutadas es necesario que la función se llame o invoque desde

otro algoritmo.

La invocación consiste en una mención al nombre de la función seguida, entre paréntesis, de los valores que se desan asignar a los argumentos. Deben aparecer tantos valores como argumentos tenga la función, y además coincidir en tipo. Estos valores se asignarán a los argumentos y se podrán utilizar, dentro de la función, como si de variables se tratase.

Como las funciones devuelven valores, es habitual que la invocación aparezca

junto con una asignación a variable para guardar el resultado y utilizarlo más adelante.

Ejemplo 1: Escribir un algoritmo que calcule el área de un círculo mediante el empleo de la función vista en el ejemplo anterior. La función área_círculo() que acabamos de ver puede ser invocada desde otro módulo, igual que invocamos las funciones de biblioteca como raiz() o redondeo()

```
algoritmo círculo
variables
   A, B, R son reales
```

```
inicio
   leer(R)
   A = área_círculo(R)
   escribir(A)
fin
```

Este fragmento de código invocará la función área_círculo() con el argumento R. La función se ejecutará con el valor de R asociado al identificador radio, exactamente igual que si éste fuera una variable y hubiéramos hecho la asignación radio = R. Una vez calculado el resultado, la función lo devuelve al módulo que la invocó, y por tanto el valor del área se asigna a la variable A.

Por último, el valor de A se escribe en la pantalla.

Ejemplo 2: Escribir un algoritmo que calcule el cuadrado y el cubo de un valor X introducido por teclado, utilizando funciones. Aunque el algoritmo es simple y podría resolverse sin modularidad, forzaremos la situación construyendo dos funciones, cuadrado() y cubo():

```
algoritmo cuadrado_cubo
variables
   N, A, B son reales
inicio
   leer(N)
```

```
   A = cuadrado(N)
   B = cubo(N)
   escribir("El cuadrado es ",
A)
   escribir("El cubo es ", B)
fin
real función cuadrado (número
es real)            // Devuelve
el cuadrado de un número
inicio
   devolver (número ^ 2)
fin
real función cubo (número es
real)            // Devuelve el
cubo de un número
inicio
   devolver (número ^ 3)
fin
```

Fíjate en que hemos escrito las

funciones después del algoritmo principal. Esto puede variar dependiendo del lenguaje utilizado.

Procedimientos

Las funciones son muy útiles como herramientas de programación, pero tienen una seria limitación: sólo pueden devolver un resultado al algoritmo que las invoca. Y en muchas ocasiones es necesario devolver más de un resultado.

Para eso existen los procedimientos, también llamados subrutinas, que son, en

esencia, iguales a las funciones, es decir:

- son algoritmos independientes que resuelven algún problema sencillo
- pueden recibir datos de entrada del algoritmo que los invoca
- el algoritmo que los invoca queda momentáneamente en suspenso mientras se ejecuta el procedimiento y, cuando éste termina, el algoritmo

principal continúa ejecutándose

Pero existe una diferencia fundamental entre las funciones y los procedimientos: los procedimientos pueden devolver 0, 1 o más resultados, mientras que las funciones siempre devuelven uno.

Los procedimientos son, por lo tanto, módulos más generales que las funciones. La declaración de un procedimiento es similar a la de una función, pero sustituyendo la palabra función por procedimiento y sin indicar

el tipo de datos del resultado; tampoco tienen sentencia devolver al final del código:

```
procedimiento
nombre_procedimiento(lista_de_a
constantes
   lista_de_constantes
variables
   lista_de_variables
inicio
   acciones
fin
```

Pero, si no tienen sentencia devolver, ¿cómo devuelve un procedimiento los resultados al algoritmo que lo invoca? La única posibilidad es utilizar los

argumentos como puerta de dos direcciones, es decir, que no solo sirvan para que el algoritmo comunique datos al subalgoritmo, sino también para comunicar datos desde el subalgoritmo hacia el algoritmo.

Para ello necesitamos saber más cosas sobre el paso de parámetros, que es lo que estudiamos en el siguiente epígrafe:

Paso de parámetros

El paso de parámetros, o comunicación de datos del algoritmo invocante al

subalgoritmo invocado, puede hacerse mediante, al menos, dos métodos:

- Paso de parámetros por valor, que es la forma más sencilla pero no permite al subalgoritmo devolver resultados en los parámetros.
- Paso de parámetros por referencia, que es más complejo pero permite a los subalgoritmos devolver resultados en los parámetros.

Veamos cada método detenidamente.

Paso de parámetros por valor

Los subalgoritmos/subprogramas, como hemos visto, pueden tener una serie de parámetros en su declaración. Estos parámetros se denominan parámetros formales.

Ejemplo: Una función que calcula la potencia de un número elevado a otro

```
real función potencia(base es
real, exponente es real)
inicio
   devolver (base ^ exponente)
fin
```

En esta declaración de función, base y

exponente son parámetros formales.

Cuando el subalgoritmo es invocado, se le pasan entre paréntesis los valores de los parámetros. A éstos se les denomina parámetros actuales; por ejemplo:

```
A = 5
B = 3
C = potencia(A,B)
```

En esta invocación de la función potencia(), los parámetros actuales son A y B, es decir, 5 y 3.

Al invocar un subalgritmo, los parámetros actuales son asignados a los

parámetros formales en el mismo orden en el que fueron escritos. Dentro del subalgoritmo, los parámetros se pueden utilizar como si fueran variables. Así, en el ejemplo anterior, dentro de la función potencia(), el parámetro base puede usarse como una variable a la que se hubiera asignado el valor 5, mientras que exponente es como una variable a la que se hubiera asignado el valor 3.

Cuando el subalgoritmo termina de ejecutarse, sus parámetros formales base y exponente dejan de existir y se

devuelve el resultado (en nuestro ejemoplo, 53), que se asigna a la variable C.

Paso de parámetros por referencia

En el paso de parámetros por referencia se produce una ligadura entre el parámetro actual y el parámetro formal, de modo que si el parámetro formal se modifica dentro del subalgoritmo, el parámetro actual, propio del algoritmo principal, también será modificado.

Los argumentos pasan sus parámetros

por valor excepto cuando indiquemos que el paso es por referencia colocando el símbolo * (asterisco) delante del nombre del argumento.

Ejemplo: Escribiremos el mismo subalgoritmo de antes, pero utilizando un procedimiento (que, en principio, no devuelve resultados) en lugar de una función.

```
procedimiento potencia(base es
real, exponente es real,
*resultado es real)
inicio
   resultado = base ^
exponente
```

```
fin
```

Observa el símbolo * delante del nombre del argumento resultado: esa es la señal de que el paso de parámetros será por referencia para ese argumento. Si no aparece el símbolo *, el paso será por valor, como es el caso de los argumentos base y exponente.

La invocación del subalgoritmo se hace del mismo modo que hasta ahora, pero delante del parámetro que se pasa por referencia debe colocarse el símbolo &:

```
A = 5
B = 3
```

```
C = 0
potencia(A, B, &C)
```

En este caso, pasamos tres parámetros actuales, ya que el subalgoritmo tiene tres parámetros formales. El tercero de ellos, C, se pasa por referencia (para señalar esta circunstancia, se antepone el símbolo &), y por lo tanto queda ligado al parámetro formal resultado.

El parámetro formal es modificado en la instrucción resutado = base ^ exponente, y como está ligado con el parámetro actual C, el valor de la variable C también se modifica. Por lo tanto, C

toma el valor 53.

Cuando el subalgoritmo termina de ejecutarse, dejan de existir todos sus parámetros formales (base, exponente y resultado), pero la ligadura de resultado con la variable C hace que esta variable conserve el valor 53 incluso cuando el parámetro resultado ya no exista.

Diferencias entre los métodos de paso de parámetros

La utilidad del método de paso de parámetros por referencia es evidente:

un subalgoritmo puede devolver tantos resultados como argumentos tenga, y no tiene que limitarse a un único resultado, como en el caso de las funciones.

El paso de parámetros por referencia suele, por lo tanto, usarse en procedimientos que tienen que devolver muchos resultados al algoritmo que los invoca. Cuando el resultado es sólo uno, lo mejor es emplear una función. Esto no quiere decir que las funciones no puedan tener argumentos pasados por referencia: al contrario, a veces es muy

útil.

Expresado de otro modo:

- el paso por valor es unidireccional, es decir, sólo permite transmitir datos del algoritmo al subalgoritmo a través de los argumentos.
- el paso por referencia es bidireccional, es decir, permite transmitir datos del algoritmo al subalgoritmo, pero también permite al subalgoritmo transmitir

resultados al algoritmo.

El problema del ámbito

Variables locales

Se llama ámbito de una variable a la parte de un programa donde dicha variable puede utilizarse.

En principio, todas las variables declaradas en un algoritmo son locales a ese algoritmo, es decir, no existen fuera del algoritmo, y, por tanto, no pueden

utilizarse más allá de las fronteras marcadas por inicio y fin. El ámbito de una variable es local al algoritmo donde se declara.

Cuando el algoritmo comienza, las variables se crean, reservándose un espacio en la memoria RAM del ordenador para almacenar su valor. Cuando el algoritmo termina, todas sus variables se destruyen, liberándose el espacio en la memoria RAM. Todos los resultados que un algoritmo obtenga durante su ejecución, por lo tanto, se

perderán al finalizar, salvo que sean devueltos al algoritmo que lo invocó o sean dirigidos a algún dispositivo de salida (como la pantalla). Esta forma de funcionar ayuda a que los algoritmos sean módulos independientes entre sí, que únicamente se comunican los resultados de sus procesos unos a otros.

Ejemplo: Calcular el cuadrado de un valor X introducido por teclado utilizando diseño modular.

```
algoritmo cuadrado
variables
   N, result son reales
```

```
inicio
   leer(N)
   calcular_cuadrado()
   escribir("El cuadrado es ",
result)
fin
procedimiento cacular_cuadrado
()             // Calcula el
cuadrado de un número
inicio
   result = N ^ 2
fin
```

En este algoritmo hay un grave error, ya que se han intentado utilizar las variables result y N, que son locales al algoritmo principal, en el subalgoritmo cuadrado(), desde donde no son

accesibles.

Es importante señalar que en algunos lenguajes de programación, y bajo determinadas circunstancias, cuando un algoritmo invoca a un subalgoritmo, puede que todas las variables locales del algoritmo estén disponibles en el subalgoritmo. Así, el ejemplo anterior podría llegar a ser correcto. Esto no ocurre en C, debido a que no se pueden anidar funciones dentro de funciones, pero debe ser tenido en cuenta por el alumno/a si en algún momento debe

programar en otro lenguaje. El problema que surge en esas situaciones es similar al de las variables globales que tratamos a continuación.

Variables globales

En ocasiones es conveniente utilizar variables cuyo ámbito exceda el del algoritmo donde se definen y puedan utilizarse en varios algoritmos y subalgoritmos. Las variables globales implican una serie de riesgos, como veremos más adelante, por lo que no deben utilizarse a menos que sea

estrictamente necesario. A pesar de los riesgos, la mayoría de los lenguajes de programación disponen de algún mecanismo para manejar variables globales.

Aunque ese mecanismo varía mucho de un lenguaje a otro, diremos como regla general que las variables globales deben declararse en el algoritmo principal, anteponiendo el identificador global al nombre de la variable, siendo entonces accesibles a todos los algoritmos y subalgoritmos que conformen el

programa.

Ejemplo: Calcular el cuadrado de un valor X introducido por teclado utilizando diseño modular.

```
algoritmo cuadrado
variables
   global N es real
   global result es reales
inicio
   leer(N)
   calcular_cuadrado()
   escribir("El cuadrado es ",
result)
fin
procedimiento cacular_cuadrado
()             // Calcula el
cuadrado de un número
inicio
```

```
    result = N ^ 2
  fin
```

El error que existía antes ya no ocurre, porque ahora las variables result y N han sido declaradas como globales en el algoritmo principal, y por lo tanto pueden utilizarse en cualquier subalgoritmo, como cuadrado().

Pudiera ocurrir que una variable global tenga el mismo nombre que una variable local. En ese caso, el comportamiento depende del lenguaje de programación (los hay que ni siquiera lo permiten), pero lo habitual es que la variable local

sustituya a la global, haciendo que ésta última sea inaccesible desde el interior del subalgoritmo. Al terminar la ejecución del subalgoritmo y destruirse la variable local, volverá a estar accesible la variable global que, además, habrá conservado su valor, pues no ha podido ser modificada desde el subalgoritmo.

De todas formas, y puestos a evitar la utilización de variables globales (a menos que no quede otro remedio), con más razón aún evitaremos usar variables

locales que tengan el mismo nombre que las globales.

Los efectos laterales

Al utilizar variables globales, muchas de las ventajas de la programación modular desaparecen.

Efectivamente, la filosofía de la programación modular consiste en diseñar soluciones sencillas e independientes (llamadas módulos) para problemas sencillos, haciendo que los módulos se comuniquen entre sí sólo

mediante el paso de parámetros y la devolución de resultados.

Cuando empleamos variables globales como en el ejemplo anterior, se crea una comunicación alternativa entre módulos a través de la variable global. Ahora un módulo puede influir por completo en otro modificando el valor de una variable global. Los módulos dejan de ser "compartimentos estanco" y pasan a tener fuertes dependencias mutuas que es necesario controlar. Cuando el programa es complejo y consta de muchos

módulos, ese control de las dependencias es cada vez más difícil de hacer.

Cualquier comunicación de datos entre un algoritmo y un subalgoritmo al margen de los parámetros y la devolución de resultados se denomina efecto lateral. Los efectos laterales, como el ilustrado en el ejemplo anterior, son peligrosísimos y fuente habitual de malfuncionamiento de los programas. Por esa razón, debemos tomar como norma:

- Primero, evitar la utilización de variables globales.
- Segundo, si no quedara más remedio que emplear variables globales, no hacer uso de ellas en el interior de los procedimientos y las funciones, siendo preferible pasar el valor de la variable global como un parámetro más al subalgoritmo.
- Por último: si, a pesar de todo, decidimos usar

variables globales por algún motivo, asegurarnos de que sabemos por qué lo hacemos y de que lo documentamos bien.

La reutilización de módulos

El diseño modular tiene, entre otras ventajas, la posibilidad de reutilizar módulos previamente escritos. Es habitual que, una vez resuelto un problema sencillo mediante una función

o un procedimiento, ese mismo problema, o uno muy parecido, se nos presente más adelante, durante la realización de otro programa. Entonces nos bastará con volver a utilizar esa función o procedimiento, sin necesidad de volver a escribirlo.

Es por esto, entre otras razones, que los módulos deben ser independientes entre sí, comunicándose con otros módulos únicamente mediante los datos de entrada (paso de parámetros por valor) y los de salida (devolución de

resultados – en las funciones – y paso de parámetros por referencia). Los módulos que escribamos de este modo nos servirán probablemente para otros programas, pero no así los módulos que padezcan efectos laterales, pues sus relaciones con el resto del programa del que eran originarios serán diferentes y difíciles de precisar.

Es habitual agrupar varios algoritmos relacionados (por ejemplo: varios algoritmos que realicen diferentes operaciones matemáticas) en un mismo

archivo, formando lo que se denomina una biblioteca de funciones. Cada lenguaje trata las librerías de manera distinta, de modo que volveremos sobre este asunto al estudiar el lenguaje C.

Por último, señalemos que, para reutilizar con éxito el código, es importante que esté bien documentado. En concreto, en cada algoritmo deberíamos documentar claramente:

- la función del algoritmo, es decir, explicar qué hace
- los parámetros de entrada

- los datos de salida, es decir, el resultado que devuelve o la forma de utilizar los parámetros por referencia

Ejemplo: Documentaremos la función potencia(), que hemos utilizado como ejemplo en otras partes de este capítulo. Es un caso exagerado, pues la función es muy sencilla y se entiende sin necesidad de tantos comentarios, pero ejemplifica cómo se puede hacer la documentación de una función.

```
{ Función: potencia() -->
Calcula una potencia de
```

```
números enteros
  Entrada: base         --> Base
de la potencia
           exponente  -->
Exponente de la potencia
  Salida:  base elevado a
exponente }
real función potencia(base es
real, exponente es real)
inicio
   devolver (base ^ exponente)
fin
```

ALGUNAS REGLAS DE ESTILO

No podemos finalizar esta primera parte del libro sin referirnos a algunas reglas

de estilo básicas que deben observarse a la hora de escribir código fuente. Y es que la escritura de un algoritmo debe ser siempre lo más clara posible, ya se esté escribiendo en pseudocódigo o en un lenguaje de programación real. La razón es evidente: los algoritmos pueden llegar a ser muy complejos, y si a su complejidad le añadimos una escritura sucia y desordenada, se volverán ininteligibles.

Esto es un aviso para navegantes: todos los programadores han experimentado la

frustración que se siente al ir a revisar un algoritmo redactado pocos días antes y no entender ni una palabra de lo que uno mismo escribió. Multiplíquese esto por mil en el caso de revisión de algoritmos escritos por otras personas.

Por esta razón, y ya desde el principio, debemos acostumbrarnos a respetar ciertas reglas básicas en cuanto al estilo de escritura. Por supuesto, un programa puede funcionar correctamente sin aplicar ninguna de las cosas que vamos a mencionar aquí, pero no es a la

corrección a lo que nos referimos ahora, sino al estilo.

Por cierto: cada programador desarrollará con el tiempo su estilo de codificación propio, pero debería hacerlo siempre dentro de un marco aceptado por la mayoría, salvo que piense desarrollar su carrera como programador en Saturno.

Partes de un algoritmo

Los algoritmos deberían tener siempre

una estructura en tres partes:

```
1 - Cabecera
2 - Declaraciones
3 - Acciones
```

Algunos lenguajes, C entre ellos, son lo bastante flexibles como para permitir saltarse a la torera esta estructura, pero es una buena costumbre respetarla siempre:

- La cabecera: contiene el nombre del programa o algoritmo.
- Las declaraciones: contiene

las declaraciones de variables y constantes que se usan en el algoritmo

- Las acciones: son el cuerpo en sí del algoritmo, es decir, las instrucciones

Puedes observar esta estructura en todos los ejemplos que hemos visto hasta ahora.

Documentación

La documentación del programa comprende el conjunto de información

interna y externa que facilita su posterior mantenimiento.

- La documentación externa la forman todos los documentos ajenos al programa: guías de instalación, guías de usuario, etc.
- La documentación interna es la que acompaña al programa. Nosotros sólo nos ocuparemos, por ahora, de esta documentación.

La forma más habitual de plasmar la

documentación interna es por medio de comentarios significativos que acompañen a las instrucciones del algoritmo o programa. Los comentarios son líneas de texto insertadas entre las instrucciones, o bien al lado, que se ignoran durante la ejecución del programa y aclaran el funcionamiento del algoritmo a cualquier programador que pueda leerlo en el futuro.

Para que el ordenador sepa qué debe ignorar y qué debe ejecutar, los comentarios se escriben precedidos de

determinados símbolos que la máquina interpreta como "principio de comentario" o "fin de comentario".

Los símbolos que marcan las zonas de comentario dependen del lenguaje de programación, como es lógico. Así, por ejemplo, en Pascal se escriben encerrados entre los símbolos (* y *):

(* Esto es un comentario en Pascal *)

El lenguaje C, sin embargo, utiliza los símbolos /* y */ para marcar los comentarios. Además, C++ permite emplear la doble barra (/ /) para

comentarios que ocupen sólo una línea. Nosotros usaremos indistintamente estos dos métodos:

/* Esto es un comentario en C */

// Esto es un comentario en C++

Ejemplo: Escribir un algoritmo que sume todos los números naturales de 1 hasta 1000

```
algoritmo sumar1000
/* Función: Sumar los números
naturales entre 1 y 1000
   Autor:   Nombre y apellidos
   Fecha:   08-11-17 */
variables
  cont es entero              /*
```

```
variable contador */
  suma es entero                /*
variable acumulador */
  N es entero
inicio
  suma = 0                /* se
pone el acumulador a 0 */
  para cont desde 1 hasta 1000
hacer            /* repetir
1000 veces */
  inicio
    suma = suma +
cont           /* los números
se suman al acumulador */
  fin
  escribir (suma)
fin
```

Este es un ejemplo de algoritmo comentado. Observa que los

comentarios aparecen a la derecha de las instrucciones, encerrados entre llaves. A efectos de ejecución, se ignora todo lo que haya escrito entre los símbolos /* y */, pero a efectos de documentación y mantenimiento, lo que haya escrito en los comentarios puede ser importantísimo.

Una buena e interesante costumbre es incluir un comentario al principio de cada algoritmo que explique bien la función del mismo y, si se considera necesario, el autor, la fecha de

modificación y cualquier otra información que se considere interesante.

Pero ¡cuidado! Comentar un programa en exceso no sólo es tedioso para el programador, sino contraproducente, porque un exceso de documentación lo puede hacer más ilegible. Sólo hay que insertar comentarios en los puntos que se considere que necesitan una explicación. En este sentido, el algoritmo del ejemplo está demasiado comentado.

Estilo de escritura

A lo largo de este capítulo has podido ver diversos ejemplos de algoritmos. Si te fijas en ellos, todos siguen ciertas convenciones en el uso de la tipografía, las sangrías, los espacios, etc. Escribir los algoritmos cumpliendo estas reglas es una sana costumbre.

Sangrías

Las instrucciones que aparezcan debajo de "inicio" deben tener una sangría mayor que dicha instrucción. Ésta

sangría se mantendrá hasta la aparición del "fin" correspondiente. Esto es particularmente importante cumplirlo si existen varios bloques inicio–fin anidados. Asimismo, un algoritmo es más fácil de leer si los comentarios tienen todos la misma sangría.

Ejemplo: Escribir un algoritmo que determine, entre dos números A y B, cuál es el mayor o si son iguales. Observa bien las sangrías de cada bloque de instrucciones, así como la posición alineada de los comentarios.

```
algoritmo comparar
// Función: Comparar dos
números A y B
variables
  A,B son enteros
inicio
  leer (A)           // leemos
los dos números del teclado
  leer (B)
  si (A == B)
entonces             // los
números son iguales
  inicio
     escribir ('Los dos
números son iguales')
  fin
  si_no             // los
números son distintos, así que
  inicio             // vamos a
compararlos entre sí
     si (A > B)
```

```
entonces
     inicio               // A es
mayor
        escribir ('A es mayor
que B')
     fin
     si_no
     inicio               // B es
mayor
        escribir ('B es mayor
que A')
     fin
  fin
fin
```

Cuándo prescindir de "inicio" y "fin"

Cuando un bloque de instrucciones sólo contiene una instrucción, podemos

escribirla directamente, sin necesidad de encerrarla entre un "inicio" y un "fin". Esto suele redundar en una mayor facilidad de lectura.

Ejemplo: Repetiremos el mismo ejemplo anterior, prescindiendo de los "inicio" y "fin" que no sean necesarios. Fíjate en que el algoritmo es más corto y, por lo tanto, más fácil de leer y entender.

```
algoritmo comparar
// Función: Comparar dos
números A y B
variables
  A,B son enteros
```

```
inicio
  leer (A)                // leemos
los dos números del teclado
  leer (B)
  si (A == B)
entonces                // los
números son iguales
     escribir ('Los dos
números son iguales')
  si_no                // los
números son distintos, así que
  inicio                // vamos a
compararlos entre sí
     si (A > B)
entonces                // A es
mayor
        escribir ('A es mayor
que B')
     si_no                // B es
mayor
        escribir ('B es mayor
```

```
  que A')
    fin
  fin
```

Tipografía

En todos los ejemplos del tema hemos resaltado las palabras del pseudocódigo en negrita, para distinguirlas de identificadores de variable, símbolos, etc. Esto también aumenta la legibilidad del algoritmo, pero, cuando utilicemos un lenguaje de programación real, no será necesario hacerlo, ya que los editores de texto que se usan en programación suelen estar preparados

para resaltar las palabras reservadas.

Ahora bien, si vas a escribir un algoritmo con un procesador de texto normal o usando pseudocódigo, es conveniente que uses una fuente de tamaño fijo o monoespaciada (el tipo Courier New es el que hemos empleado en la versión impresa de este texto; si lo estás leyendo en un e-reader, el tipo concreto dependerá del dispositivo). A veces se distinguen en **negrita** las palabras clave del lenguaje para facilitar la lectura de los algoritmos.

Los editores de texto orientados a la programación (hablaremos de ellos más adelante) hacen algo parecido: siempre usan un tipo de fuente monoespaciado, y colorean el código para distinguir de un solo vistazo palabras reservadas, números, literales y otros elementos del lenguaje, de modo que facilitan enormemente la legibilidad.

Espacios

Otro elemento que aumenta la legibilidad es espaciar suficientemente (pero no demasiado) los distintos

elementos de cada instrucción. Por ejemplo, esta instrucción ya es bastante complicada y difícil de leer:

```
si (a > b) y (c > d * raiz(k)
) entonces a = k + 5.7 * b
```

Pero se lee mucho mejor que esta otra, en la que se han suprimido los espacios (excepto los imprescindibles):

```
si (a>b)y(c>d*raiz(k))entonces
a=k+5.7*b
```

Al ordenador le dará igual si escribimos (a > b) o (a>b), pero a cualquier programador que deba leer nuestro código le resultará mucho más cómoda

la primera forma.

Por la misma razón, también es conveniente dejar líneas en blanco entre determinadas instrucciones del algoritmo cuando se considere que mejora la legibilidad.

Identificadores

A la hora de elegir identificadores de variables (o de constantes) es muy importante utilizar nombres que sean significativos, es decir, que den una idea de la información que almacena esa

variable. Por ejemplo, si en un programa de nóminas vamos a guardar en una variable la edad de los empleados, es una buena ocurrencia llamar a esa variable "edad", pero no llamarla "X", "A" o "cosa".

Ahora bien, dentro de esta política de elegir identificadores significativos, es conveniente optar por aquellos que sean lo más cortos posible, siempre que sean descifrables. Así, un identificador llamado "edad_de_los_empleados" es engorroso de escribir y leer, sobre todo

si aparece muchas veces en el algoritmo, cuando probablemente "edad_empl" proporciona la misma información. Sin embargo, si lo acortamos demasiado (por ejemplo "ed_em") llegará un momento en el quede claro lo que significa.

Toda esta idea de significación de los identificadores es extensible a los nombres de los algoritmos, de las funciones, de los procedimientos, de los archivos y, en general, de todos los objetos relacionados con un programa.

Por último, señalar que muchos lenguajes de programación distinguen entre mayúsculas y minúsculas, es decir, que para ellos no es lo mismo el identificador "edad" que "Edad" o "EDAD". Es conveniente, por tanto, ir acostumbrándose a esta limitación. Nosotros preferiremos usar identificadores en minúscula, por ser lo más habitual entre los programadores de lenguaje C.

SEGUNDA PARTE: EL LENGUAJE C

Ha llegado el momento de hablar del lenguaje C. Si has llegado hasta aquí, ya conoces las tres estructuras básicas de la programación estructurada (secuencial, condicional e iterativa), así como los tipos de datos simples, las expresiones y operadores, y las técnicas básicas de programación modular. Lo

que aún no sabes es cómo escribir todo esto en C.

En realidad, esa es lo más fácil. Si ya tienes experiencia programando y lo has hecho en más de un lenguaje, habrás observado que pasar de un lenguaje imperativo a otro es bastante sencillo: en unos pocos días, estás programando en el nuevo lenguaje como si llevaras toda la vida haciéndolo.

C es el padre (o quizá sería mejor decir el abuelo) de la mayor parte de los lenguajes imperativos que aún existen en

la actualidad. Es más, como C es el padre de C++, no es exagerado decir que C también es el abuelo (o el tío-abuelo, si nos ponemos exquisitos con los parentescos) de muchos de los lenguajes orientados a objetos modernos. Eso quiere decir que las expresiones sintácticas típicas de C están heredadas en la mayor parte de los lenguajes más populares, de modo que, si has programado, digamos, en Java, en Python, en PHP, en Javascript, en C# o en Perl, por citar solo unos cuantos, la sintaxis de C te resultará muy familiar.

Pero antes de empezar con C es conveniente que te proporcionemos un pequeño mapa para que te orientes en las confusas aguas de los lenguajes de programación, porque existen, literalmente, cientos de lenguajes. Tantos, que para hablar de ellos no nos queda más remedio que categorizarlos y así tratar de responder a esta pregunta: ¿en qué se diferencian unos de otros?

Los lenguajes de programación

Podemos definir un lenguaje de programación como **un conjunto de símbolos que se combinan de acuerdo con una sintaxis bien definida para posibilitar la transmisión de instrucciones a la CPU** (definición extraída de QUERO, E., *Fundamentos de programación*, Ed. Paraninfo, 2003).

Dicho de otro modo: el lenguaje de programación es el código con el que podemos transmitir al ordenador las órdenes de un programa. Hasta ahora hemos usado pseudocódigo (y, en menor

medida, diagramas de flujo) para escribir esas órdenes. Ahora llega el momento de traducir ese pseudocódigo en un código real, el lenguaje de programación, comprensible por la máquina.

Lenguajes de programación hay muchos, cada uno con sus ventajas e inconvenientes. Conviene, por tanto, clasificarlos en categorías. Nosotros haremos dos clasificaciones:

- La primera, atendiendo al nivel de abstracción del

lenguaje, distinguirá entre lenguajes de bajo nivel y de alto nivel.

- La segunda, según el proceso de traducción a código máquina, distinguirá entre lenguajes interpretados, compilados y ensamblados.

Lenguajes de alto y bajo nivel

El ordenador, como es sabido, solo puede manejar ceros y unos, es decir,

código o lenguaje binario. Los seres humanos, por el contrario, utilizamos un lenguaje mucho más complejo, con montones de símbolos y reglas sintácticas y semánticas, que denominaremos lenguaje natural.

Entre estos dos extremos (lenguaje binario y lenguaje natural) se encuentran los lenguajes de programación. Tienen cierto parecido con el lenguaje natural, pero son mucho más reducidos y estrictos en su sintaxis y semántica, para acercarse a las limitaciones del lenguaje

binario.

Hay lenguajes de programación muy próximos al lenguaje binario: a éstos los llamamos lenguajes de bajo nivel de abstracción. Y los hay más próximos al lenguaje natural: son los lenguajes de alto nivel de abstracción.

Lenguajes de bajo nivel

Son los lenguajes más cercanos a la máquina. Los programas directamente escritos en código binario se dice que están en lenguaje máquina que, por lo

tanto, es el lenguaje de más bajo nivel que existe.

Las instrucciones del lenguaje máquina realizan tareas muy sencillas, como, por ejemplo, sumar dos números, detectar qué tecla se ha pulsado en el teclado o escribir algo en la pantalla del ordenador. Cuando se combinan adecuadamente muchas de estas instrucciones sencillas se obtiene un programa de ordenador que puede realizar tareas muy complejas.

A pesar de la simplicidad de las

instrucciones del lenguaje máquina, la forma de escribirlas es muy complicada, ya que hay que hacerlo en binario. En los primeros años de la informática los ordenadores se programaban directamente en lenguaje máquina, lo cual convertía la tarea de programar en una verdadera pesadilla. Por ejemplo, una instrucción para sumar dos números en lenguaje máquina puede tener este aspecto:

```
1101001001011100101000100010011
```

Cuando los ordenadores fueron

haciéndose más potentes, pronto se vio que con el lenguaje máquina no se podrían crear programas que aprovechasen esa potencia por la sencilla razón de que era demasiado difícil programar así: no se podía hacer nada demasiado complicado porque el cerebro humano no está “diseñado” para pensar en binario.

Surgió entonces la idea de utilizar el propio ordenador como traductor: ¿por qué no escribir una instrucción como la anterior, que suma dos números, de una

forma más parecida al lenguaje humano y que luego un pequeño programa de ordenador se encargue de traducir esa instrucción a su correspondiente ristra de ceros y unos? Así apareció el lenguaje ensamblador, cuyas instrucciones son equivalentes a las del lenguaje máquina, pero se escriben con palabras similares a las del lenguaje humano. Por ejemplo, para sumar dos números, la instrucción en ensamblador puede ser algo como:

```
ADD  D1, D2
```

Los lenguajes de bajo nivel se caracterizan por ser dependientes del hardware de la máquina. Es decir: un programa escrito en lenguaje máquina o en ensamblador para un procesador con arquitectura x86 no funcionará, por ejemplo, en un smartphone con arquitectura ARM, a menos que sea modificado sustancialmente. Incluso puede tener serios problemas para funcionar en máquinas de la misma familia pero con el resto del hardware diferente, o con un sistema operativo distinto.

Lenguajes de alto nivel

Siguiendo el razonamiento anterior (utilizar el propio ordenador como traductor), en los años sesenta se empezaron a desarrollar lenguajes cada vez más complejos, en los que cada instrucción ya no se correspondía exactamente con una instrucción del lenguaje máquina, sino con varias. Estos son los lenguajes de alto nivel o, simplemente, L.A.N. (no confundir con "red de área local")

Lógicamente, la traducción desde un

lenguaje de alto nivel a lenguaje máquina es mucho más compleja que desde lenguaje ensamblador, por lo que los traductores se han hecho cada vez más complicados.

Una característica muy importante de los lenguajes de alto nivel es que son independientes del hardware, lo que implica que los programas desarrollados con estos lenguajes pueden ser ejecutados en ordenadores con hardware totalmente distinto. A esto se le llama portabilidad.

Los programas encargados de traducir el código de alto nivel a código máquina se llaman compiladores e intérpretes. Son programas muy complejos que generan el código binario equivalente al código de alto nivel para una máquina concreta. Por lo tanto, el programa de alto nivel, que es portable de un hardware a otro, debe ser traducido a código máquina en cada tipo de máquina en la que se pretenda ejecutar.

Los ejemplos de lenguajes de alto nivel son innumerables, y la lista incluye casi

todos de los que has oído hablar alguna vez: Basic, Cobol, Fortran, Ada, C, PHP, Python, Java, Perl, etc.

Comparación entre los lenguajes de alto y bajo nivel

LENGUAJES DE BAJO NIVEL	LI
Ventajas	
Son comprensibles directamente por la máquina (aunque el ensamblador necesita una pequeña traducción)	Neces compl e intér

Los programas se ejecutan muy rápidamente (si están bien escritos, claro)	La tra alto ni genera si se e binari
Ocupan menos espacio en memoria	Ocupa
Permiten controlar directamente el hardware, por lo que son apropiados para la programación	En ger hardw operat entonc

de sistemas	operat direct
Inconvenientes	
Son completamente dependientes del hardware. Un programa escrito para determinado tipo de máquina no funcionará en un ordenador con diferente arquitectura.	Son p del ha una m con ha vuelva máqui
Incluso los programas más	Los pr que un

sencillos son largos y farragosos	equiva binari
Los programas son difíciles de escribir, depurar y mantener	Los pr escrib
Es imposible resolver problemas muy complejos	Es pos proble

Enfrentando las ventajas e inconvenientes de unos y otros, se concluye que, en general, es preferible usar lenguajes de alto nivel para el desarrollo de aplicaciones, reservando

los de bajo nivel para casos muy concretos en los que la velocidad de ejecución o el control del hardware sean vitales. Por ejemplo, los sistemas operativos más conocidos, como Windows, MacOS o Linux, están programados casi en su totalidad con lenguajes de alto nivel (generalmente C o C++), reservando un pequeño porcentaje del código a rutinas en ensamblador.

También hay que destacar que no todos los lenguajes de alto nivel son iguales.

Los hay de "más alto nivel" que otros. C tiene sin duda menor nivel de abstracción que, por ejemplo, Visual Basic; pero, por eso mismo, los programas en C son más rápidos y eficientes que los escritos en Visual Basic, aunque también pueden llegar a ser más difíciles de escribir y depurar.

Categorías dentro de los lenguajes de alto nivel

Para terminar con esta vista preliminar sobre el mundo de los lenguajes de programación, mencionaremos que los

lenguajes de alto nivel se suelen subdividir en categorías tales como:

- Lenguajes de tercera generación (o imperativos), en los que el programador escribe una secuencia de instrucciones que el ordenador debe ejecutar en un orden preestablecido. Son los lenguajes que nosotros vamos a manejar. Todos los lenguajes "clásicos" pertenecen a esta categoría:

C, Basic, Cobol, Fortran, etc.

- Lenguajes de cuarta generación (o 4GL), dirigidos a facilitar la creación de interfaces con el usuario y con otras aplicaciones, como las bases de datos. Un ejemplo de estos lenguajes es SQL.
- Lenguajes orientados a objetos, que son una evolucuión de los lenguajes de tercera generación y que

permiten construir con mayor facilidad y robustez programas modulares complejos. Ejemplos de lenguajes orientados a objetos son C++, Java, Python, PHP o Ruby. Algunos de ellos son multiparadigma, es decir, permiten programar con orientación a objetos pero también permiten hacer programación estructurada clásica, sin objetos.

- Lenguajes declarativos y funcionales, propios de la inteligencia artificial, como Prolog o Lisp.
- Otos tipos más específicos: lenguajes concurrentes, paralelos, distribuidos, etc.

En general, podemos decir que un programador acostumbrado a trabajar con un lenguaje de tercera generación puede aprender con poco esfuerzo cualquier otro lenguaje de tercera generación, y, con algo más de trabajo,

un lenguaje orientado a objetos. Sin embargo, el "salto" a otros tipos de lenguajes, como los declarativos, cuesta más porque la raíz misma de estos lenguajes es diferente.

Ensambladores, compiladores e intérpretes

Cuando programamos en un lenguaje distinto del lenguaje máquina, nuestro código debe ser traducido a binario para que el ordenador pueda entenderlo y

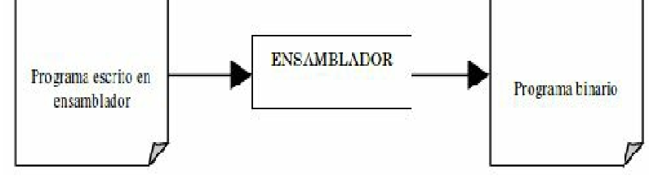
ejecutarlo. Existe un programa específico encargado de hacer esa traducción y que, dependiendo del lenguaje en el que hayamos escrito nuestro programa, puede ser un ensamblador, un compilador o un intérprete.

Ensambladores

Se llaman ensambladores los programas encargados de traducir los programas escritos en ensamblador a código binario.

Fíjate que tanto el programa traductor como el lenguaje se llaman del mismo modo: ensamblador.

Como el lenguaje ensamblador es muy próximo al binario, estos traductores son programas relativamente sencillos.



Compiladores

El compilador es un programa que

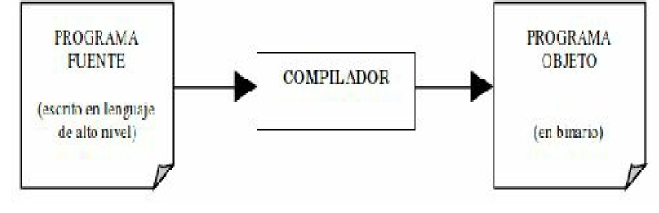
traduce el código de alto nivel a código binario. Es, por tanto, parecido al ensamblador, pero mucho más complejo, ya que las diferencias entre los lenguajes de alto nivel y el código binario son muy grandes.

El programa escrito en lenguaje de alto nivel se denomina programa fuente o código fuente. El programa traducido a código binario se llama programa objeto o código objeto. Por lo tanto, el compilador se encarga de convertir el programa fuente en un programa objeto.

Una vez que se ha obtenido el programa objeto ya no es necesario volver a realizar la traducción (o compilación), a menos que se haga alguna modificación en el programa fuente, en cuyo caso habría que volver a compilarlo.

El programa objeto, una vez generado, puede ejecutarse en la máquina en la que fue compilado, o en otra de similares características (procesador, sistema operativo, etc.). Cuando se usa programación modular, puede ser necesario un proceso previo de enlace

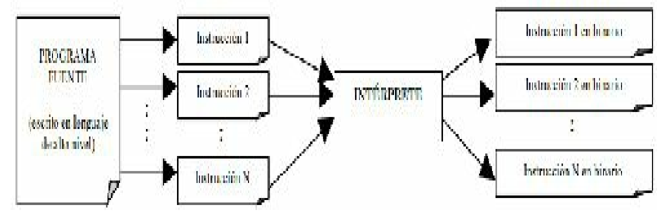
de los diferentes módulos, pero de esto ya hablaremos más adelante.



Intérpretes

El intérprete es un programa que traduce el código de alto nivel a código binario pero, a diferencia del compilador, lo hace en tiempo de ejecución. Es decir, no se hace un proceso previo de traducción de todo el programa fuente a

binario, sino que se va traduciendo y ejecutando instrucción por instrucción.



Compiladores frente a intérpretes

El intérprete es notablemente más lento que el compilador, ya que realiza la traducción al mismo tiempo que la ejecución. Además, esa traducción se lleva a cabo siempre que se ejecuta el programa, mientras que el compilador

sólo la hace una vez. Por estos motivos, un mismo programa interpretado y compilado se ejecuta mucho más despacio en el primer caso.

La ventaja de los intérpretes es que hacen que los programas sean más portables. Así, un programa compilado en una máquina PC bajo Windows no funcionará en un Macintosh, o en un PC bajo Linux, a menos que se vuelva a compilar el programa fuente en el nuevo sistema. En cambio, un programa interpretado funcionará en todas las

plataformas, siempre que dispongamos del intérprete en cada una de ellas.

JavaScript es un ejemplo de lenguaje interpretado. Esto permite que los programas JavaScript (llamados comúnmente *scripts*) puedan funcionar en cualquier máquina que disponga de un navegador de Internet capaz de interpretarlos. En cambio, C/C++ es un lenguaje compilado, lo que hace que los programas desarrollados con estos lenguajes se ejecuten más rápido que sus equivalentes en JavaScript, aunque

obliga a volver a compilarlos si se desea ejecutarlos en una máquina con diferente hardware o diferente sistema operativo.

Por último, hay ciertos lenguajes que pueden ejecutarse de forma interpretada o de forma compilada. No es el caso de C. El lenguaje C está orientado a obtener la velocidad de ejecución más alta posible, y por eso siempre se compila. Pero ya es hora de empezar a hablar del lenguaje C, ¿no es así?

Vamos a ello.

INTRODUCCIÓN AL LENGUAJE C

Características básicas de C

C es un lenguaje compilado de alto nivel (aunque a veces se le denomina "de nivel medio" debido a sus características a medio camino entre una y otra categoría) cuyas propiedades fundamentales son:

- Es un lenguaje eficiente.

- Es un lenguaje con muy pocas órdenes (comandos).
- Los operadores de C son más numerosos que en la mayoría de los lenguajes de programación anteriores y contemporáneos suyos.
- Muchas de las sentencias de decisión y de bucles han servido de referencia para el diseño de todos los lenguajes creados en estos últimos años, de modo especial los

populares Java y Visual Basic (no confundir Java con JavaScript: son dos lenguajes muy diferentes. Tampoco se debe confundir Visual Basic con el antiguo Basic)

- C es un lenguaje muy eficiente, casi tanto como el ensamblador, por lo que es adecuado para desarrollar software en el que la velocidad de ejecución sea importante: sistemas

operativos, sistemas en tiempo real, compiladores, software de comunicaciones, etc.

- C es altamente portable, más que otros lenguajes de alto nivel, ya que existen compiladores para lenguaje C estándar en todas las plataformas imaginables
- Es un lenguaje muy popular y, por lo tanto, existen multitud de librerías de funciones ya

programadas que se pueden reutilizar, así como documentación abundante.

- C es más críptico que la mayoría de los otros lenguajes de programación de alto nivel. Su naturaleza críptica proviene de la enorme cantidad de operadores y un número pequeño de palabras clave o palabras reservadas. El lenguaje C estándar (ANSI

C) tiene solamente 32 palabras reservadas, un número extremadamente pequeño comparado a otros lenguajes como Visual Basic.

Breve historia de C

En 1972, los laboratorios Bell necesitaban un nuevo sistema operativo. Hasta ese momento, la mayoría de los sistemas operativos estaban escritos en lenguaje ensamblador ya que los lenguajes de alto nivel no generaban

programas lo suficientemente rápidos. Pero los programas escritos en ensamblador son difíciles de mantener y Bell quería que su nuevo sistema operativo se pudiera mantener y modificar con facilidad. Por lo tanto, se decidieron a inventar un lenguaje de alto nivel nuevo con el que programar su sistema operativo. Este lenguaje debía cumplir dos requisitos: ser tan manejable como cualquier otro lenguaje de alto nivel (para que los programas fueran fáciles de mantener) y generar un código binario tan rápido como el

escrito directamente en ensamblador.

Brian Kerningham y Dennis Ritchie, dos ingenieros de laboratorios Bell, tras varios intentos (C procede de otro lenguaje llamado B, que a su vez procede de otro anterior), terminaron de diseñar el lenguaje C en un ordenador DEC PDP-11. El lenguaje C pasó a convertirse y conocerse como "un lenguaje de programación de alto-bajo nivel". Eso significa que soporta todas las construcciones de programación de cualquier lenguaje de alto nivel,

incluyendo construcciones de programación estructurada, y al mismo tiempo se compila en un código eficiente que corre casi tan rápidamente como un lenguaje ensamblador.

Los laboratorios Bell terminaron de construir su sistema operativo Unix y su lenguaje de programación por excelencia, C. El tándem C – Unix ha sido la referencia fundamental en el mundo de la programación en el último medio siglo, y C se ha convertido en uno de los lenguajes de programación más

populares y longevos de la historia de la informática. C creció en popularidad muy rápidamente y sigue siendo uno de los lenguajes fundamentales tanto en el mundo educativo como en el mundo profesional.

El lenguaje C como tal aparece descrito por primera vez en el libro "The C Programming Language" (Prentice-Hall, 1978), auténtica biblia de la programación escrita por Kerningham y Ritchie. Todavía se pueden encontrar ediciones recientes de ese texto, y existe

incluso una edición digital oficial. El lenguaje se extendió rápidamente y surgieron diferentes implementaciones con ligeras diferencias entre sí hasta que el instituto de estándares americano (ANSI) formó un comité en 1983 para definir un estándar del lenguaje.

El primer estándar ANSI C apareció en 1990 y fue revisado en 1999. Una evolución de C fue el lenguaje C++ que, a parte de todas las características del ANSI C, incluye la posibilidad de orientación a objetos, una técnica de

programación ligeramente diferente de la programación estructurada. En el año 2000, Microsoft patentó el lenguaje C#, otra evolución de C++ orientada al desarrollo de aplicaciones para la plataforma .NET de esta compañía.

En la actualidad son muchos los fabricantes de compiladores C, y todos cumplen con la norma ANSI C, por lo que el código escrito para un compilador es altamente portable a otros. Algunos de ellos son Visual C++ (o C#) de Microsoft, Embarcadero C++

Builder (antiguo Borland C++), el Intel C++ Compiler y, por supuesto, el legendario compilador gcc con licencia GNU en sus diferentes versiones.

Un lenguaje para programadores

H. Schildt, en su "Manual de referencia de Turbo C/C++" editado por McGraw-Hill, se hace una interesante reflexión que al principio puede resultar sorprendente: "pero... ¿no son todos los lenguajes para programadores? La

respuesta es sencillamente: no."

Analizando un poco más las razones del autor para tan rotunda negativa, se llega a la conclusión de que existen determinados lenguajes (algunos clásicos, como Basic, Cobol o Fortran, y otros más actuales, como Visual Basic, Python o PHP) que han sido diseñados para permitir que los no programadores puedan leer y comprender los programas y, presumiblemente, aprender a escribir los suyos propios para resolver problemas sencillos.

Por el contrario, C fue creado, influenciado y probado en vivo por programadores profesionales. El resultado es que C da al programador lo que muchos programadores piden: unas pocas y bien escogidas palabras clave, una biblioteca poderosa y estandarizada, unas mínimas restricciones y un máximo control sobre lo que sucede en el interior de la máquina. Si a esto unimos que el código objeto generado por C es casi tan eficiente como el ensamblador, se entenderá por qué lleva medio siglo siendo uno de los lenguajes más

populares entre los programadores profesionales.

Ahora bien, C también tiene sus detractores que lo acusan de ser confuso, críptico y demasiado flexible. En efecto, con C se pueden desarrollar las técnicas de programación estructurada, pero también se puede programar "código espagueti". Esto, sin embargo, ocurre con todos los lenguajes: incluso los que tienen una sintaxis más estilizada y elegante, como Python o Ruby, pueden generar código

absolutamente ininteligible en manos de un programador manazas.

Un lenguaje estructurado y modular

C es un lenguaje estructurado porque contiene las estructuras de control básicas que hemos estudiado con anterioridad. También permite romper las estructuras y escribir programas no estructurados, pero nosotros evitaremos hacerlo.

C es un lenguaje estrictamente modular. Todos los algoritmos se escriben en forma de funciones, incluido el algoritmo principal (cuya función siembre recibe el mismo nombre: main()). En C no existen los procedimientos, pero se pueden escribir funciones que no devuelvan ningún valor, es decir, funciones que en realidad son procedimientos.

VARIABLES, OPERADORES Y

Generalidades sintácticas de C

Antes de profundizar en la programación en C, debemos conocer algunas normas básicas del lenguaje:

- Los bloques de código se marcan con las llaves {...}. Son equivalentes al inicio y fin que usábamos en pseudocódigo.

- Todas las instrucciones terminan con un punto y coma (;)
- Los identificadores de variables, funciones, etc., no pueden empezar con un número ni contener espacios o símbolos especiales, salvo el de subrayado (_)
- Los caracteres se encierran entre comillas simples ('…')
- Las cadenas de caracteres se encierran entre comillas

dobles ("...")

- El lenguaje es sensitivo a las mayúsculas. Es decir, no es lo mismo escribir main() que MAIN() o Main()

Tipos de datos simples

Los tipos fundamentales o simples de datos admitidos por C son los siguientes:

Denominación	Tipo de datos	Ta

		bit
char	Carácter	8
int	Número entero	16
float	Número real de precisión simple	32
double	Número real de precisión doble	64
void	Tipo vacío	0

Esta tabla es sólo una orientación, ya

que pueden existir variaciones entre compiladores. Por ejemplo, el viejo compilador Borland C++ para Windows utilizaba enteros de 16 bits, pero el compilador mingw integrado con el Dev-C++ utiliza enteros de 32 bits (en realidad, interpreta que todos los “int” son “long int”; véase el modificador “long” más abajo).

El programador debe estar al tanto de los límites que utiliza el compilador que esté usando para evitar los overflows. Una forma sencilla de hacerlo es

utilizando el operador sizeof(tipo). Por ejemplo, sizeof(int) nos devuelve la cantidad de bytes que ocupa un dato de tipo int.

El tipo char se usa normalmente para variables que guardan un único carácter, aunque lo que en realidad guardan es un código ASCII, es decir, un número entero de 8 bits sin signo (de 0 a 255). Los caracteres se escriben siempre entre comillas simples ('…'). Por lo tanto, si suponemos que x es una variable de tipo char, estas dos asignaciones tienen

exactamente el mismo efecto, ya que 65 es el código ASCII de la letra A:

```
x = 'A';
x = 65;
```

Mucho cuidado con esto, porque las cadenas de caracteres se escriben con comillas dobles ("…") a diferencia de las comillas simples de los caracteres sueltos.

El tipo int se usa para números enteros, mientras que los tipos float y double sirven para números reales. El segundo permite representar números mayores, a

costa de consumir más espacio en memoria.

El tipo void tiene tres usos. El primero es para declarar funciones que no devuelven ningún valor (procedimientos); el segundo, para declarar funciones sin argumentos; el tercero, para crear punteros genéricos. En posteriores epígrafes se discutirán los tres usos.

Observa que en C no existe el tipo de dato lógico. Se utiliza en su lugar el tipo int, representando el 0 el valor falso y

cualquier otra cantidad (normalmente 1) el valor verdadero.

Modificadores de tipo

Existen, además, unos modificadores de tipo que pueden preceder a los tipos de datos char e int. Dichos modificadores son:

- signed: obliga a que los datos se almacenen con signo
- unsigned: los datos se almacenan sin signo

- long: los datos ocuparán el doble de espacio en bits del habitual, y, por lo tanto, aumentará su rango de valores
- short: los datos ocuparán la mitad del espacio habitual, y, por lo tanto, disminuirá su rango de valores

De este modo, nos podemos encontrar, por ejemplo, con estos tipos de datos (suponiendo que un "int" normal ocupe 16 bits):

- unsigned int: Número entero de 16 bits sin signo. Rango: de 0 a 65535.
- signed int: Número entero de 16 bits con signo. No tiene sentido, porque el tipo int ya es con signo por definición, pero es sintácticamente correcto.
- signed char: Carácter (8 bits) con signo. Rango: de –128 a 127
- long int: Número entero de

32 bits. Rango: de –
2147483648 a 2147483647

Incluso podemos encontrar combinaciones de varios modificadores. Por ejemplo:

- unsigned long int: Número entero de 32 bits sin signo. Rango: de 0 a 4294967295

Variables: ámbito y asignación

Todas las variables deben declararse

antes de ser usadas. La sintaxis de la declaración incluye su tipo y su nombre (identificador):

```
tipo_de_datos
lista_de_variables;
```

Por ejemplo:

```
int cont;
char respuesta;
float x, y, resultado;
```

En C no está delimitado el lugar del algoritmo donde deben declararse las variables, siendo la única condición que se declaren antes de ser usadas por primera vez. Sin embargo, nosotros

recomendamos, al menos al principio, hacer la declaración inmediatamente después de abrir el bloque algorítmico, antes de la primera instrucción.

Todas las variables son, salvo que se indique otra cosa, locales a la función donde estén definidas, dejando de existir al finalizar la función. Las variables globales se declaran fuera del cuerpo de todas las funciones y antes de la función main(), que es el algoritmo principal. Recuerda que debes evitar el uso de variables globales a menos que sea

estrictamente necesario.

Se pueden aplicar ciertos modificadores a las variables que modifican la forma en la que almacenan y/o su ámbito. Estos modificadores puedes consultarlos en la sexta parte del libro. Por ahora, no te van a hacer falta.

Para asignar un valor a una variable se utiliza la sentencia de asignación, exactamente igual que en pseudocódigo. Por ejemplo:

```
cont = cont + 1;
respuesta = 'S';
x = 5.33;
```

Constantes

Recuerda que también se pueden usar identificadores para asociarlos a valores constantes, es decir, valores que no cambiarán nunca durante la ejecución del programa.

Para declarar una constante y asignarle un valor se utiliza el modificador const delante de la declaración:

```
const tipo_de_datos
nombre_constante = valor;
```

Por ejemplo:

```
const float pi = 3.141592;
```

El valor de la constante pi no podrá ser modificado a lo largo del programa.

Otra forma de definir constantes es mediante una directiva del compilador:

```
#define PI = 3.141592
```

Las directivas no son instrucciones de C, sino consignas comunicadas al compilador para que sepa que, si encuentra el símbolo PI en el código fuente, debe sustituirlo por 3.141592. Puedes leer más detalles sobre las directivas en los apéndices de este libro. Por ahora nos basta saber que

existen estas dos formas de declarar constantes.

Conversiones de tipo

C es un lenguaje débilmente tipado, es decir, no hace comprobaciones estrictas de tipos a la hora de asignar un valor a una variable o de comparar dos expresiones.

Por ejemplo, estas instrucciones son correctas:

```
float a;
int b;
b = 5;
```

```
a = b;
```

Se ha asignado un valor entero a la variable "a", que es de tipo float. En otros lenguajes esto no está permitido, pero en C se realizan conversiones automáticas de tipo cuando en una misma expresión aparecen datos de tipos diferentes. Esto, que en principio es una ventaja, pues elimina algunas limitaciones engorrosas, otras veces es peligroso porque algunos datos pueden cambiar extrañamente de valor al hacerse esa conversión automática.

La conversión puede ser de dos clases:

- Asignación de un valor a una variable que permita más precisión. Por ejemplo, asignar un número entero a una variable float. En este caso, el número se convierte a real añadiendo ".0" a la parte decimal. No hay pérdida de información.
- Asignación de un valor a una variable que permita menos precisión. Por ejemplo,

asignar un número long int a una variable de tipo int. En este caso, el número se recorta, perdiendo sus bits más significativos, es decir, los que están a la izquierda, y por lo tanto hay pérdida de información. Hay que tener mucho cuidado con este tipo de conversiones porque pueden producir resultados imprevisibles

Además de las conversiones automáticas

de tipo, el programador puede forzar la conversión de tipos a voluntad utilizando moldes. Un molde es una expresión de un tipo de datos entre paréntesis que aparece delante de un dato. Entonces, antes de evaluar la expresión, el dato es convertido al tipo especificado en el molde. Por ejemplo:

```
float a;
int b;
a = 5;
b = (float)a/2;
```

Sin el molde (float), la división a/2 sería entera, ya que a es una variable de

tipo int, y se perdería la parte decimal. Al aplicar el molde, se convierte momentáneamente el valor entero 5 al valor real 5.0 y se evalúa la expresión, que ahora sí se realiza como división real, conservando sus decimales.

Operadores y expresiones

C es un lenguaje muy rico en operadores, por lo que ahora solo hablaremos de los más habituales, dejando otros muy específicos para

temas posteriores.

Operadores aritméticos

Igual que en pseudocódigo, en C existen los operadores aritméticos típicos, y alguno más que más abajo comentaremos:

Operador	Operación
+	Suma
-	Resta

*	Multiplicación
/	División
%	Módulo
++	Incremento
--	Decremento

Se pueden utilizar paréntesis ( ) para cambiar el orden de las operaciones, pero no corchetes [ ], que C se reserva para otros usos.

Observa que no existe el operador potencia. En C, las potencias se calculan con funciones de librería.

Tampoco existe el operador de división entera. En C se utiliza el mismo símbolo para la división entera y la real: la barra (/). Simplemente, si los operandos son de tipo entero, C realiza una división entera, y si son de tipo real, la división será con decimales.

Los operadores incremento y decremento no suelen existir en otros lenguajes, pero son muy prácticos.

Sirven para abreviar las expresiones típicas de los contadores:

cont++;	es equivalente a	cont = cont + 1
cont--;	es equivalente a	cont = cont – 1

Los operadores de incremento y decremento pueden escribirse antes o después de la variable. Es decir, que estas dos expresiones son correctas y realizan la misma operación:

```
cont++;
++cont;
```

Ahora bien, no son exactamente iguales cuando aparecen como parte de una expresión, ya que la primera se realiza después de evaluar la expresión, y, la segunda, antes.

Esto quiere decir que, en este caso, tanto la variable x como la variable y tomarán el valor 11:

```
x = 10;
y = ++x;
```

Pero, escrito de esta otra forma, la variable x toma el valor 11, pero y se

queda con 10, ya que el incremento (x++) se realiza después de evaluar la expresión y asignarla a la variable y:

```
x = 10;
y = x++;
```

Operadores relacionales

Los operadores relacionales no ofrecen ninguna dificultad porque son exactamente iguales a los que hemos utilizado en pseudocódigo. Sólo hay que hacer una salvedad: el C, como se ha dicho, no existe el tipo de dato lógico, sino que se emplean números enteros.

Falso se representa con el valor 0. Verdadero se representa con cualquier valor distinto de cero, aunque preferentemente se usa el 1.

- Falso = 0
- Verdadero = 1 (o cualquier número distinto de 0)

Los operadores relacionales en C son:

Operador	Operación
>	Mayor que

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

Operadores lógicos

Los operadores lógicos de C también son los mismos que usamos en pseudocódigo, aunque se escriben de

manera diferente. Recuerda que el resultado de las operaciones lógicas, en C, no es verdadero o falso, sino 1 ó 0.

<table>
<tr><th>Operador</th><th>Operación</th></tr>
<tr><td>&&</td><td>Y</td></tr>
<tr><td>||</td><td>O</td></tr>
<tr><td>!</td><td>No</td></tr>
</table>

Otros operadores de C

C dispone de otros operadores sobre los

que el lector puede obtener información en cualquier manual de programación en C. Aquí nos limitaremos a mencionarlos, apuntando que algunos de ellos los estudiaremos en temas posteriores, cuando nos sean necesarios.

- Operadores a nivel de bits: & (and), | (or), ^ (xor), ~ (complemento a uno), >> (desplazamiento a la derecha) y << (desplazamiento a la izquierda). Actúan

directamente sobre los bits de la representación binaria de un dato.

- Operador condicional: ? (puede sustituir a condicionales simples y dobles)
- Operadores puntero: & (dirección) y * (contenido). Los estudiaremos detenidamente en el tema de estructuras dinámicas, aunque empezaremos a manejarlos

antes.

- Operador en tiempo de compilación: sizeof (longitud en bytes de un identificador). También los estudiaremos en el tema de estructuras dinámicas.
- Operadores de acceso a elementos de estructuras: . (acceso directo) y -> (acceso por puntero). Estos los veremos en dos temas posteriores: el dedicado a las

estructuras de datos estáticas
y el de estructuras dinámicas.

Precedencia de operadores y conversión de tipos en expresiones

Las expresiones en C son similares a las que hemos estado usando en pseudocódigo: combinaciones de variables, constantes y operadores. Las expresiones se evalúan, es decir, se calcula su resultado, aplicando las reglas de precedencia de operadores, que pueden alterarse mediante el uso de paréntesis. Las reglas de precedencia

son las mismas que aplicamos en pseudocódigo.

En una expresión es posible que aparezcan variables y/o constantes de diferentes tipos de datos. Cuando esto ocurre, C convierte todos los datos al tipo más grande. Por ejemplo, si aparecen datos de tipo short int, int y long int, todos ellos se convertirán a long int antes de realizar las operaciones.

ESTRUCTURAS DE

CONTROL

Las estructuras de control en C son muy similares a las que hemos utilizado en pseudocódigo, cambiando ligeramente la notación empleada. Pasamos a continuación a presentarlas todas, con su equivalente en pseudocódigo. Para una explicación más extensa de cada una de ellas, puedes volver a leer la primera parte de este libro.

Condicional simple

Pseudocódigo	

`si condición entonces`
`inicio`
`   acciones`
`fin`
	`if`
	`{`
	
	`}`

Observa que, en C, la condición debe escribirse entre paréntesis y que no se emplea la palabra "entonces".

Condicional doble

Pseudocódigo	

`si condición entonces`
`  inicio`
`    acciones-1`
	`if`
	`{`

`    fin`	`}`
`  si_no`	`el`
`    inicio`	`{`
`      acciones-2`	
`    fin`	`}`

Condicional múltiple

Pseudocódigo	
`según expresión hacer`	`sw`
`inicio`	`{`
`  valor1: acciones-1`	
`  valor2: acciones-2`	
`  valor3: acciones-3`	
`    ...`	
`  valorN: acciones-N`	
`  si_no: acciones-si_no`	

fin	
	}

Esta estructura presenta algunas peculiaridades, a saber:

- La expresión discriminante debe escribirse entre paréntesis y ser de un tipo ordinal (int, char o similar). No pueden usarse cadenas ni números reales.

- Los valores no pueden ser

expresiones, sino constantes, es decir, números o caracteres fijos.

- ¡Cuidado! Las acciones no son bloques de instrucciones, es decir, no van encerradas entre { y }. Eso quiere decir que, si se ejecutan las acciones-2, después se ejecutarán automáticamente las acciones-3, luego las acciones-4, etc. La forma de evitar esto es insertar la

instrucción break al final de cada bloque de instrucciones.

Bucle mientras

Pseudocódigo	
mientras condición hacer **inicio** **acciones** **fin**	**wh** **{** **}**

Bucle repetir

Pseudocódigo	
repetir	**do**

inicio
   acciones
fin
mientras que condición
	{
	   a
	}
	whil

Bucle para

Pseudocódigo	
para cont desde valor_inicial hasta valor_final **inicio** **acciones** **fin**	

Cuidado con este tipo de bucle porque es algo diferente a como lo hemos visto en pseudocódigo. Ya se ha dicho en

varias ocasiones que C es a veces un poco críptico. El bucle para (o bucle for) es un ejemplo típico de ello ya que:

- La variable contador debe ser inicializada con una asignación dentro de la instrucción for.
- El valor final debe ser expresado en forma de condición, como haríamos en un bucle mientras.
- El incremento del contador hay que indicarlo

explícitamente.

Por ejemplo, el siguiente bucle en pseudocódigo:

```
para cont desde 1 hasta 100
inc 2 hacer
inicio
    acciones
fin
```

Tendría esta traducción en C:

```
for (cont = 1; cont <= 100;
cont = cont + 2)
{
    acciones
}
```

Funciones. La función

MAIN()

Como se ha dicho anteriormente, C es un lenguaje modular hasta el extremo de que todas las líneas de código deben pertenecer a alguna función, incluyendo las instrucciones del algoritmo principal, que se escriben en una función llamada principal (main en inglés)

Funciones

La declaración de funciones se hace de forma similar a la empleada en

pseudocódigo:

```
tipo_devuelto nombre_función
(parámetros_formales)
{
   ...instrucciones...
   return expresión;
}
```

Observa que las únicas diferencias con el pseudocódigo son que no se usa la palabra "función", que las llaves { y } sustituyen a inicio y fin, y que se emplea la palabra return en lugar de devolver.

Procedimientos

Si el tipo_devuelto es void, se considera

que la función no devuelve ningún valor y que, por lo tanto, es un procedimiento. Entonces, un procedimiento se declara así:

```
void nombre_procedimiento
(parámetros_formales)
{
   ...instrucciones...
}
```

Paso de parámetros

Los parámetros formales son, como en pseudocódigo, una lista de tipos e identificadores que se sustituirán por los parámetros actuales y se usarán como

variables dentro de la función.

Los parámetros se pasan normalmente por valor, pero también se pueden pasar por referencia. El paso de parámetros por referencia admite dos sitaxis ligeramente diferentes en C: anteponiendo el operador * (asterisco) al nombre del parámetro (como hemos hecho en pseudocódigo) o anteponiendo el operador &.

Paso de parámetros por valor

Por ejemplo, en esta función el paso de

parámetros es por valor:

```
int funcion1 (int x, int y)
```

Esto quiere decir que la función1 recibirá únicamente el valor de los dos parámetros, x e y. Podrá utilizar esos valores a lo largo de su código, e incluso podrá cambiarlos. Pero cualquier cambio en x e y no afectará a los parámetros actuales, es decir, a los parámetros del programa que llamó a función1.

Paso de parámetros por referencia con el operador *

En la siguiente función, el paso del parámetro "x" es por valor y el del parámetro "y", por referencia:

```
int funcion2 (int x, int *y)
```

¡OJO! Recuerda que cada vez que se vaya a usar el parámetro "y" dentro del código de la función, será necesario acompañarlo del asterisco. Por ejemplo:

```
*y = 5;
x = 17 + *y;
```

(Hay algunas excepciones a esta regla, pero ya las veremos cuando surjan más adelante)

Por último, también en la llamada a la función hay que indicar explícitamente si alguno de los parámetros se está pasando por referencia, utilizando el operador &, como en pseudocódigo. Por lo tanto, para llamar a la funcion2 del ejemplo anterior con los parámetros A y B habrá que escribir:

```
resultado = funcion2 (A, &B);
```

Observa que el segundo parámetro (el que se pasa por referencia), lleva delante el operador &.

Una última observación: en realidad, *en*

C no existe en paso por referencia propiamente dicho, sino que se pasa a la función la dirección de memoria del parámetro actual por valor. Es decir, se pasa un puntero por valor (véanse los punteros más adelante en este mismo texto). Como la función accede directamente a esa posición de memoria, puede cambiar la variable del programa principal. En la práctica, el resultado es el mismo que si se pasase por referencia, y para el programador no apreciará diferencia alguna.

Paso de parámetros por referencia con el operador &

Otra forma de pasar un argumento por referencia es usar el operador & en los parámetros formales, así:

```
int funcion3 (int x, int &y)
```

En esta función, el parámetro x se pasa por valor y el parámetro "y" se pasa por referencia. Utilizando esta sintaxis no es necesario añadir asteriscos cada vez que se usa la "y" en el cuerpo de la función, ni tampoco usar "&" en la llamada a la función.

Esta tercera forma de paso por referencia no es estándar en C, sino que es propia de C++, por lo que evitaremos utilizarla.

Juntándolo todo en un ejemplo

En el siguiente ejemplo se ilustra los dos tipos de paso de parámetros y, en el paso por referencia, las dos sintaxis alternativas de que dispone C.

El ejemplo muestra tres funciones muy similares que reciben dos parámetros, a y b. Las tres intentan *intercambiar* el

valor de a y b mediante una tercera variable, tmp. Sin embargo, en la primera de ellas el intercambio no tiene ningún efecto en el programa main(), ya que los parámetros están pasados por valor. En las otras dos funciones sí que se consigue el intercambio, ya que los parámetros está pasados por referencia.

Lo más interesante de este ejemplo es mostrar cuál es la sintaxis correcta en cada tipo de paso de parámetros.

```
#include <stdio.h>
// Paso de parámetros por
valor.
```

```
// En este ejemplo, esta
función no tendrá el efecto
deseado, porque las variables
// del programa principal no
se verán afectadas.
void intercambiar1(int a, int
b)
{
     int tmp = a;
     a = b;
     b = tmp;
}
// Paso de parámetros por
referencia, sintaxis 1.
// Esta función sí que
consigue intercambiar los
valores de las variables
// del programa principal.
void intercambiar2(int *a, int
*b)
{
```

```
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
// Paso de parámetros por
referencia, sintaxis 2.
// Esta función también
consigue su objetivo. A todos
los efectos,
// es idéntica a la función
anterior.
void intercambiar3(int &a, int
&b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
// Programa principal
int main()
{
```

```
    int dato1 = 30, dato2 = 90;

    printf("Antes de la llamada a
las funcioens: dato1 = %i, dato2
= %i\n", dato1, dato2);
    intercambiar1(dato1, dato2);
    printf("Después de
intercambiar1: dato1 = %i, dato2
= %i\n", dato1, dato2);
    intercambiar2(&dato1,
&dato2);
    printf("Después de
intercambiar2: dato1 = %i, dato2
= %i\n", dato1, dato2);
    intercambiar3(dato1, dato2);
    printf("Después de
intercambiar3: dato1 = %i, dato2
= %i\n", dato1, dato2);

    return 0;
}
```

La función main()

La función main() contiene el algoritmo o módulo principal del programa. La ejecución de un programa siempre empieza por la primera línea de la función main()

La función main(), como todas las funciones de C, puede devolver un valor. El valor devuelto por main() debe ser de tipo entero. Esto se utiliza para pasar algún valor al programa que haya llamado al nuestro, que suele ser el sistema operativo. Si main() no

devuelve un número entero al sistema operativo mediante una sentencia return, entonces nuestro programa devolverá un número desconocido. Moraleja: es una buena idea incluir un return al final de la función main(). Generalmente, la devolución de un 0 indica al sistema operativo que el programa a finalizado sin problemas, mientras que cualquier otro valor señala que se ha producido algún error.

Por lo tanto, la forma habitual de la función main() será:

```
int main(void)
{
   ...instrucciones del
algoritmo principal...
   return 0;
}
```

Observa que main() no tiene argumentos, por lo que aparece el identificador void entre paréntesis en la declaración. También se pueden utilizar argumentos en main(), pero eso es algo que trataremos en otro momento.

Prototipos de funciones

En C no es necesario escribir las funciones (subalgoritmos) antes de su primera invocación. El mecanismo de compilación y enlace de C permite, de hecho, que las funciones puedan estar físicamente en un archivo distinto del lugar desde el que se invocan.

En la práctica, esto plantea un problema: C no tiene forma de saber si la llamada a una función se hace correctamente, es decir, si se le pasan los argumentos debidos y con el tipo correcto, ni si el resutado devuelto es asignado a una

variable del tipo adecuado.

Para conseguir que C realice esas comprobaciones se utilizan los prototipos de función. Un prototipo de función es la declaración de una función. Consiste, simplemente, en la primera línea del código la función. El prototipo debe aparecer antes de que la función se invoque por primera vez, aunque el código completo de la función esté en otra parte. Los prototipos permiten al compilador comprobar que los argumentos de la función coinciden en

tipo y número con los de la invocación de la misma, y que el tipo devuelto es el correcto.

Los prototipos suelen aparecer al principio del programa, antes de la función main(). Observa, en el siguiente ejemplo, que el prototipo de la función calcular_area() se coloca delante de main(). Sin embargo, el código concreto de esta función no aparece hasta después (incluso podría estar situado en otro archivo diferente):

```
float calcular_area (float
base, float
```

```
altura);              //
Prototipo de la función
int
main()
Algoritmo principal
{
   ...instrucciones...
   area = calcular_area (x,y);
   ...más instrucciones...
   return 0;
}
float calcular_area(float
base, float
altura)             // Código de
la función
{
   ... instrucciones...
}
```

Cuando se vayan a usar funciones de

librería, como fabs() (valor absoluto), sqrt() (raíz cuadrada) o cualquier otra, hay que escribir sus prototipos antes de la función main(). Sin embargo, como estas funciones no las hemos escrito nosotros, desconocemos cuales son sus prototipos.

En C se soluciona este problema con los archivos de cabecera, que son archivos proporcionados con el compilador de C que incluyen en su interior los prototipos de las funciones de librería, entre otras cosas. Como funciones de librería hay

muchas, también hay muchos archivos de cabecera. Por ejemplo, el archivo math.h tiene los prototipos de todas las funciones matemáticas. Todos los archivos de cabecera tienen la extensión .h en su nombre (h de "header").

Para incluir un archivo de cabecera en nuestro programa se utiliza #include, que no es exactamente una instrucción de C, sino una directiva de compilación. Más adelante veremos qué significa eso. Ya hemos visto otra directiva de compilación: #define, que usábamos

para definir constantes. Las directivas de compilación se detallan en uno de los apéndices de este libro, y puedes ir ahora allí si quieres ampliar esta información.

Por ejemplo, esta línea de código sirve para incluir todos los prototipos de las funciones de librería matemática en nuestro programa:

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

Al final del tema, en el apéndice dedicada a las funciones ANSI, encontrarás una lista con las funciones

utilizadas más habitualmente y sus correspondientes archivos de cabecera. Cada vez que necesites usar una de esas funciones en un programa, debes escribir al principio del mismo el #include del archivo de cabecera para disponer así del prototipo.

Estructura general de un programa en C

Visto todo esto, ya estamos en condiciones de echar un vistazo a cual

será el aspecto de (casi) todos los programas que escribamos en C.

Todo programa C, desde el más pequeño hasta el más complejo, tiene un programa principal ubicado en la función main(). Además, por encima de main() deben aparecer los prototipos de funciones (y esto implica a los archivos de cabecera, si se utilizan funciones de librería) y las variables y constantes globales. Por debajo de main() encontraremos el código de las funciones de usuario.

Por lo tanto, la estructura habitual de nuestros programas en C debería ser esta:

```
/* Comentario inicial: nombre
del programa, del programador,
fecha, etc */
/* Archivos de cabecera
(prototipos de funciones de
librería) */
#include <archivo_cabecera.h>
#include <archivo_cabecera.h>
/* Prototipos de funciones
escritas por nosotros */
float función1 (argumentos);
float función2 (argumentos);
/* Variables globales */
int variable_global1;
char variable_global2;
/* Algoritmo principal */
```

```
int main(void)
{
   /* Variables locales del
algoritmo principal */
   int a, b;
   float x, y;
   ...
   ...
   /* Instrucciones del
algoritmo principal */
   ...
   función1(argumentos);
   ...
   función2(argumentos);
   ...
   return 0;
}
/* Código completo de las
funciones escritas por
nosotros */
float función1 (argumentos)
```

```
{
   /* Variables locales e
instrucciones de este
subalgoritmo */
}
float función2 (argumentos)
{
   /* Variables locales e
instrucciones de este
subalgoritmo */
}
```

ENTRADA Y SALIDA ESTÁNDAR

La entrada y salida de datos en C, es decir, la traducción de las instrucciones leer() y escribir() de pseudocódigo, es

uno de los aspectos más difíciles (y criticables) de C.

El estándar ANSI C dispone de muchas funciones para hacer las entradas y salidas de datos. En concreto, dispone de un subconjunto de ellas para hacer la entrada y salida por consola, es decir, por teclado y pantalla.

Podemos clasificar estas funciones de E/S en dos grupos:

- Funciones de E/S simples: getchar(), putchar(), gets(), puts()

- Funciones de E/S con formato: printf(), scanf()

Las más utilizadas y versátiles son sin duda las segundas, así que nos detendremos más en ellas.

E/S con formato

Salida de datos: printf()

La función printf() (de "print" = imprimir y "f" = formato) sirve para escribir datos en el dispositivo de salida estándar (generalmente la pantalla) con

un formato determinado por el programador. La forma general de utilizarla es la siguiente:

```
printf(cadena_de_formato,
datos);
```

El prototipo de printf() se encuentra en el archivo de cabecera stdio.h (de "std" = standard e "io" = input/output, es decir, entrada/salida; por lo tanto, "stdio" es un acrónimo de "entrada/salida estándar")

El primer argumento, la cadena_de_formato, especifica el modo

en el que se deben mostrar los datos que aparecen a continuación. Esta cadena se compone de una serie de códigos de formato que indican a C qué tipo de datos son los que se desean imprimir. Todos los códigos están precedidos del símbolo de porcentaje ("%"). Por ejemplo, el código "%i" indica a la función que se desea escribir un número de tipo int, y el código "%f", que se desea escribir un número real de tipo float.

La forma más simple de utilizar printf()

es:

```
int a;
a = 5;
printf("%i", a);
```

Esto escribirá el valor de la variable entera a en la pantalla, es decir, 5. Fíjate que el primer argumento de printf() es una cadena (y, por lo tanto, se escribe entre comillas) cuyo contenido es el código del tipo de dato que se pretende escribir. El segundo argumento es el dato mismo.

En una sola instrucción printf() pueden escribirse varios datos. Por ejemplo:

```
int a;
float x;
a = 5;
x = 10.33;
printf("%i%f", a, x);
```

Observa detenidamente la cadena de formato: primero aparece "%i" y luego "%f". Esto indica que el primer dato que debe imprimirse es un entero, y el segundo, un real. Después, aparecen esos datos separados por comas y exactamente en el mismo orden que en la cadena de formato: primero a (la variable entera) y luego x (la variable real). El resultado será que en la

pantalla se escribirán los números 5 y 10.33.

Los códigos de formato que se pueden utilizar en printf() son:

Código	Tipo del dato que se escribe
%c	Carácter
%d	Número entero
%i	Número entero
%e	Número real con notación

	científica
%f	Número real
%g	Usar %e o %f, el más corto
%o	Número octal
%s	Cadena de caracteres
%u	Entero sin signo
%x	Número hexadecimal
%p	Puntero

Algunos de estos códigos sirven para imprimir tipos de datos que aún no conocemos, pero que iremos viendo en las siguientes páginas.

Hay códigos que admiten modificadores. Por ejemplo:

- Los códigos numéricos "%i", "%d", "%u" (para números enteros) y "%f", "%e" y "%g" (para números reales), permiten insertar modificadores de longitud "l" (longitud doble) y "h"

(longitud corta). Así, por ejemplo, "%ld" indica que se va a imprimir un entero de longitud doble (long int); "%hu" sirve para enteros cortos sin signo (unsigned short int); "%lf" indica que se imprimirá un número real de longitud doble (double), etc.

- El código "%f" (números reales) se pueden usar con un modificador de posiciones decimales que se desean

mostrar. Por ejemplo, con "%10.4f" obligamos a que se impriman diez dígitos a la izquierda de la coma decimal y cuatro a la derecha. La escritura se ajusta a la derecha. Para ajustarla a la izquierda se utiliza el modificador "-", de esta forma: "%-10.4f"

- El código "%s" (cadenas de caracteres) se puede combinar con un

especificador de longitud máxima y mínima de la cadena. Por ejemplo, "%4.8s" escribe una cadena de al menos cuatro caracteres y no más de ocho. Si la cadena tiene más, se pierden los que excedan de ocho. También se puede utilizar el modificador "-" para alinear el texto a la izquierda.

Además de los códigos de formato, en la cadena de formato puede aparecer

cualquier texto entremezclado con los códigos. A la hora de escribir en la pantalla, los códigos serán sustituidos por los datos correspondientes. Por ejemplo:

```
int a;
float x;
a = 5;
x = 10.33;
printf("El número entero es %i
y el real es %f", a, x);
```

Lo que aparecerá en la pantalla al ejecutar este fragmento de código será:

```
El número entero es 5 y el
real es 10.33
```

Una última observación sobre printf(). Hay ciertos caracteres que no son directamente imprimibles desde el teclado. Uno de ellos es el salto de línea. Para poder ordenar a printf() que escriba un salto de línea (o cualquier otro carácter no imprimible) se utilizan los códigos de barra invertida, que con códigos especiales precedidos del carácter "\".

En concreto, el carácter "salto de línea" se indica con el código "\n". Observa las diferencias entre estos dos bloques de

instrucciones para intentar comprender la importancia del salto de línea:

```
int a;
a = 5;
printf("La variable a vale
%i", a);
a = 14;
printf("La variable a vale
%i", a);
```

El resultado en la pantalla de la ejecución de estas instrucciones es:

```
La variable a vale 5La
variable a vale 14
```

Veamos el mismo ejemplo usando el código del salto de línea (\n):

```
int a;
a = 5;
printf("La variable a vale
%i\n", a);
a = 14;
printf("La variable a vale
%i", a);
```

El resultado en la pantalla será:

```
La variable a vale 5
La variable a vale 14
```

Entrada de datos: scanf()

La función scanf() es, en muchos sentidos, la inversa de printf(). Puede leer desde el dispositivo de entrada estándar (normalmente el teclado) datos

de cualquier tipo de los manejados por el compilador, convirtiéndolos al formato interno apropiado. Funciona de manera análoga a printf(), por lo que su sintaxis es:

```
scanf(cadena_de_formato,
datos);
```

El prototipo de scanf() se encuentra en el archivo de cabecera stdio.h (de "std" = standard e "io" = input/output, es decir, entrada/salida)

La cadena_de_formato tiene la misma composición que la de printf(). Los

datos son las variables donde se desea almacenar el dato o datos leidos desde el teclado. ¡Cuidado! Con los tipos simples, es necesario utilizar el operador & delante del nombre de la variable, porque esa variable se pasa por referencia a scanf() para que ésta pueda modificarla.

Por ejemplo:

```
int a, b;
float x;
scanf("%d", &a);
scanf("%d%f", &b, &x);
```

La primera llamada a scanf() sirve para

leer un número entero desde teclado y almacenarlo en la variable a. La segunda llamada lee dos números: el primero, entero, que se almacena en b; y, el segundo, real, que se almacena en x.

La función scanf() tiene alguna otra funcionalidad añadida para el manejo de cadenas de caracteres que ya veremos en su momento.

Ejemplo de uso de scanf() y printf()

Debido a la relativa complejidad de estas funciones de entrada y salida,

vamos a presentar un pequeño ejemplo de traducción de pseudocódigo a C. Se trata de un algoritmo que lee dos números enteros, A y B. Si A es mayor que B los resta, y en otro caso los suma.

Observa detenidamente la correspondencia entre cada pareja de instrucciones, especialmente las de entrada y salida.

Pseudocódigo
algoritmo suma_y_resta **variables** **   a y b son enteros** **inicio**

```
    escribir ("Introduzca dos
números enteros")
    leer(a, b)
    si (a < b) entonces
        escribir("La suma de
a y b es:", a+b)
    si_no
        escribir("La resta de
a menos b es:", a-b)
fin
```

E/S simple por consola

Técnicamente, con printf() y scanf() es posible escribir y leer cualquier tipo de

datos desde cualquier dispositivo de salida o entrada, no solo la pantalla y el teclado, como de hecho comprobaremos cuando estudiemos los ficheros.

En la práctica, aunque printf() resulta bastante efectiva y versátil, scanf() puede darte muchos dolores de cabeza. Para hacerte una idea, sólo tienes que probar a hacer un scanf() de un número entero e inmediatamente después otro scanf() de una cadena de caracteres. El segundo scanf() fallará. La razón es bastante rocambolesca: el flujo de

entrada no consumirá el carácter de retorno de carro al leer el número entero, por lo que dicho carácter se adjudicará al segundo scanf() automáticamente.

Por suerte, existe otro grupo de funciones en ANSI C específicamente diseñadas para hacer la E/S por consola, es decir, por teclado y pantalla, de manera más simple. Las resumimos en el siguiente cuadro.

Los prototipos de estas funciones se encuentran en el archivo de cabecera

stdio.h (de "std" = standard e "io" = input/output, es decir, "entrada/salida")

Función	Utilidad
getchar()	Espera a que se pulse seguida de INTRO y d su valor. Muestra el ec pantalla, es decir, la te pulsada aparece en la
putchar(carácter)	Escribe un carácter en pantalla
	Lee del teclado una ca

gets(cadena)	caracteres seguida de
puts(cadena)	Escribe una cadena de caracteres en la pantal

Para evitar los problemas que a menudo causa scanf(), podemos recurrir a gets() para leer las cadenas de caracteres. Si necesitamos leer un número, podemos usar gets() y luego convertir la cadena a un tipo de dato numérico con las funciones de conversión atoi() y atof(),

como se muestra en el siguiente ejemplo:

```
char cadena[50];
int a;
float x;
gets(cadena);
Leemos una cadena de
caracteres
a =
atoi(cadena);
Convertimos la cadena en un
número entero
x =
atof(cadena);
Convertimos la cadena en un
número real
```

Las funciones de conversión atoi() y atof() tratarán de convertir la cadena en

un número, si ello es posible (es decir, si la cadena realmente contiene números). Estas funciones, junto con muchas otras, se describen en el apéndice I de este capítulo.

Usar la combinación de gets() con atoi() o atof() es más costoso que utilizar scanf(). Primero, porque necesitamos una variable auxiliar de tipo cadena. Y, segundo, porque gets() es una función peligrosa: si se teclean más caracteres de los que caben en la cadena, el resultado es imprevisible (a menudo el

programa se cuelga). Esto también tiene solución utilizando en su lugar la función fgets().

Tal vez pienses que resulta demasiado complicado hacer algo muy simple como una entrada de datos por teclado. Tienes razón. Pero ten en cuenta dos cosas: las entradas de datos *nunca* son simples (son el punto donde el usuario interacciona con más libertad con el programa, y los usuarios humanos tendemos a hacer cosas impredecibles y complicadas), y C es un lenguaje de

nivel intermedio, por lo que muchas tareas de alto nivel, simplemente, no las resolverá por nosotros. En ese sentido, C requiere del programador prestar atención a ciertos detalles que podría obviar en otros lenguajes. Es por esto, entre otras cosas, por las que C tiene exaltados detractores pero también incondicionales entusiastas.

Por último, mencionaremos que los compiladores de Borland tienen dos variaciones muy útiles de la función getchar() llamadas getche() y getch().

Estas funciones, no definidas en el estándar ANSI de C, son como getchar() pero sin necesidad de pulsar INTRO detrás del carácter. La primera muestra el eco, es decir, escribe en la pantalla la tecla pulsada, y la segunda no. Los prototipos de estas funciones se encuentran en conio.h (de "con" = consola e "io" = input/output)

FLUJO DE TRABAJO PROGRAMANDO CON LENGUAJE C

Como vimos más arriba, el ciclo de vida de desarrollo del software consta de una serie de etapas. En esta parte del libro nos estamos centrando a la etapa de implementación o codificación del software, ya que estamos aprendiendo un lenguaje de programación concreto.

Antes de continuar profundizando en el lenguaje, vamos a hacer un pequeño paréntesis para especificar cuál suele ser el flujo de trabajo en el desarrollo de programas con lenguaje C.

Cuando se trabaja con C, la

implementación de un programa suele dividirse en varias subfases: edición, compilación, enlace y depuración. Pasamos a describirlas brevemente a continuación.

Edición del código fuente

Editar consiste en escribir el código fuente del programa en el lenguaje seleccionado, en nuestro caso C.

Para escribir el código nos puede servir cualquier procesador de textos que

permita guardar el documento en forma de texto ASCII plano (sin códigos de control y formato propios de los procesadores avanzados, como MS Word).

Existen multitud de procesadores de texto plano para programar en lenguaje C. Solo tienes que hacer una pequeña búsqueda en internet para encontrar una pléyade de candidatos. La ventaja de estos procesadores es que resaltan, en diferentes colores y tipografías, las palabras clave, las funciones, las

cadenas, los comentarios, etc, haciendo de este modo mucho más legible el código fuente. Necesitarás probar unos cuantos antes de decidir cuál es el que más te gusta.

Además, es habitual que los compiladores de C incluyan también un editor. Por ejemplo, los compiladores de Borland (como Turbo C/C++, Borland C/C++ o C++ Builder) poseen un entorno integrado de desarrollo, que es un programa que une al editor de texto, al compilador y al depurador en

una sola aplicación controlada por un único interfaz, lo cual facilita mucho el trabajo. Estos editores con funcionalidades añadidas suelen denominarse IDEs (Integrated Development Environment, entorno integrado de desarrollo), y nos referiremos a ellos con más detalle en los apéndices de este libro.

Mi recomendación es que, al menos al principio, intentes usar un editor simple, que no incluya el compilador. Esto te obligará a hacer un trabajo adicional al

tener que compilar y enlazar manualmente tu programa, pero es la mejor forma de comprender realmente en qué consiste la compilación y el enlace. En el futuro, esta comprensión te puede ayudar a resolver muchos errores de compilación y enlace.

En cualquier caso, las recomendaciones que hemos de seguir durante la edición del código fuente son:

- No empezar a teclear código sin haber entendido bien el problema que se nos plantea.

Si éste es complejo, es imprescindible plantear antes su descomposición modular en papel, resolviendo los módulos con pseudocódigo.

- Recuerda: comenzar a teclear a lo loco y sin pensar antes la solución detenidamente es la manera más segura de tardar el mayor tiempo posible en desarrollar un programa que, además, no funcione bien.
- Realizar un diseño modular

previo del programa. Recuerda que un módulo de más de 30 ó 40 líneas (aproximadamente) empieza a ser demasiado largo.

- Evitar las variables globales.
- Elegir bien el nombre de los identificadores (variables, constantes, funciones...). Que sean significativos y no excesivamente largos.
- Identar el texto, es decir, dejar las sangrías necesarias

para facilitar su comprensión.

- Usar espacios y líneas en blanco siempre que se considere que facilita la lectura.
- Ser generosos documentando el código fuente. Mejor que sobren comentarios que no que falten.
- Guardar el código fuente en archivos de texto cuya extensión sea ".c" (por

ejemplo: "ejercicio.c")

Compilación

El proceso de compilación, como sabes, consiste en que un programa, llamado compilador, traduzca el código fuente en lenguaje C a código binario. La compilación, por lo tanto, no es más que una traducción.

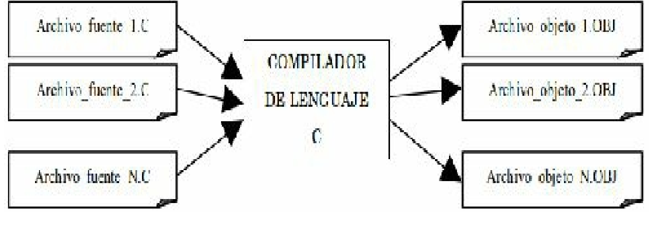
El resultado de la compilación es el mismo programa traducido a código binario. Como el programa fuente estaba almacenado en un archivo con extensión

.C, el compilador suele guardar el programa objeto en otro archivo con el mismo nombre y extensión .OBJ.

Los programas cortos se guardan en un único archivo fuente que se traducirá a un único archivo objeto. Pero cuando los programas crecen, es habitual distribuir el código fuente en varios archivos con el objetivo de manipularlo mejor.

Los compiladores de C usan compilación separada. Esto significa que, si un programa largo está escrito en

varios archivos fuente, no es necesario compilarlos todos cada vez que se modifica algo. Basta con volver a compilar el archivo modificado. Por eso, dividir un programa fuente largo en varios archivos más cortos también sirve para mejorar los tiempos de compilación.



Cuando tenemos varios archivos fuente

es normal que existan dependencias entre ellos. Por ejemplo, cuando en un archivo A1 se utiliza (con la directiva #include) un archivo de cabecera A2. Si modificamos el archivo A2 es necesario volver a compilar el archivo A1, aunque A1 no haya sido modificado en absoluto. Se dice entonces que existe una dependencia entre los archivos A1 y A2.

Controlar las dependencias es un trabajo tedioso y propenso a errores. Por fortuna, los compiladores se encargan de controlarlas por sí mismos. Así que no

te extrañes si, al volver a compilar un archivo fuente después de modificarlo, se compilan automáticamente algunos otros archivos, aunque no los hayas tocado. El control de las dependencias lo puede realizar el compilador de manera automática o semiautomática (mediante archivos de dependencias o makefiles escritos por el programador), como veremos en los apéndices.

Los compiladores, en fin, son programas muy complejos que, además, tienen multitud de opciones de compilación.

Algunas de esas opciones también las veremos en los apéndices del libro. Allí encontrarás información sobre cómo compilar tus programas con Mingw, un compilador de C/C++ para Windows, y con gcc, el compilador nativo de Linux.

Enlace (link)

Cuando existen varios programas objeto es necesario combinarlos todos para dar lugar al programa ejecutable definitivo. Este proceso se denomina enlace.

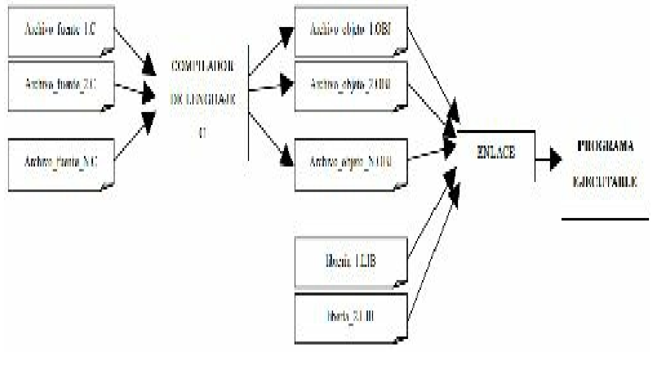
El código objeto de las funciones de

librería de C se encuentra almacenado en varios archivos (cuya extensión es .LIB) situados en ubicaciones conocidas por el enlazador. De este modo, el código objeto de las funciones de librería que hayamos utilizado en nuestro programa puede unirse con el código objeto del programa durante en enlace, generándose el programa ejecutable.

Por lo tanto, es necesario hacer el enlace cuando el programa se encuentra distribuido en varios archivos, o cuando

dentro del programa se utilizan funciones de librería. Esto quiere decir que, en la práctica, el enlace hay que hacerlo siempre.

El enlazador o linker, es decir, el programa encargado de hacer el enlace, es en realidad diferente del compilador, que sólo hace la traducción. Sin embargo, la mayoría de los compiladores de C lanzan automáticamente el enlazador al finalizar la compilación para que el programador no tenga que hacerlo.



El enlace de nuestro código objeto con las funciones de librería puede hacerse de dos maneras:

- Enlace estático. Consiste en unir durante el enlace el código objeto de las librerías con el código del programa,

generando así el ejecutable. El programa ejecutable crece notablemente de tamaño respecto de los archivos objeto, ya que incorpora el código de todas las funciones de las librerías. El enlace estático es el que normalmente se utiliza a menos que indiquemos otra cosa.

- Enlace dinámico. El código de las librerías no se une al

del programa, sino que se busca durante la ejecución, únicamente cuando es requerido. El enlace dinámico produce, por lo tanto, ejecuciones más lentas, ya que cada vez que se use una función de librería dinámica es necesario buscar el archivo en el que se encuentra y ejecutar su código. Además, pueden producirse errores de enlace durante la ejecución del

programa. Sin embargo, el enlace dinámico tiene las ventajas de reducir el tamaño del archivo ejecutable y permitir la compartición de librerías entre diferentes aplicaciones.

Depuración

La depuración del programa consiste en localizar y corregir los errores que se hayan podido producir durante el desarrollo. El objetivo es conseguir un

programa que funcione lo más correctamente posible, aunque hay que tener presente que ningún programa complejo está libre de errores al 100%

Los errores pueden ser de tres tipos:

- Errores en tiempo de compilación. Se producen al traducir el código fuente a código objeto. El compilador los detecta y marca en qué línea se han producido, y de qué tipo son, por lo que son relativamente fáciles de

corregir. Los errores de compilación más frecuentes son:

- Errores sintácticos: escribir mal alguna instrucción o algún identificador, u olvidarnos del punto y coma que debe terminar cada instrucción.
- Errores de tipos: intentar asignar a una variable de cierto tipo un valor de otro tipo incompatible, o invocar

a una función con argumentos de tipo equivocado. Recuerda que C puede hacer conversiones de tipo automáticas, por lo que estos errores pueden quedar enmascarados.

- Errores de identificadores no reconocidos: ocurren cuando se intenta utilizar una variable o una constante que no ha sido declarada, o cuyo ámbito no llega al lugar

donde se intenta utilizar.

- Avisos. Además de los errores, el compilador puede dar avisos (warnings) en lugares donde potencialmente puede existir un error de compilación. Es conveniente revisar todos los avisos y tratar de corregirlos antes de continuar con la ejecución.
- Errores en tiempo de enlace. Cuando el compilador termina la traducción se

produce el enlace de todos los archivos objeto. En este momento se resuelven todas las llamadas a funciones, de modo que si alguna función no está presente en el conjunto de archivos objeto, el enlazador fallará y explicará la causa del error.

- Errores en tiempo de ejecución. Si la compilación y el enlace terminan sin novedad, se genera un

archivo ejecutable (con extensión .EXE en sistemas Windows). Es el momento de comprobar que el programa realmente hace lo que se espera que haga. Para ello hay que probarlo con diversos conjuntos de datos de entrada; la elaboración de estos juegos de pruebas es una técnica que excede nuestras pretensiones.

Los errores que surgen en tiempo de

ejecución son los más complicados de corregir, ya que muchas veces no está clara la causa del error. En el peor de los casos, puede ser necesario rediseñar la aplicación por completo.

Simplificando mucho, podemos encontrarnos con estos errores en tiempo de ejecución:

- Errores lógicos. Se producen cuando alguna condición lógica está mal planteada. Entonces, el flujo del programa puede ir por la

rama "si_no" cuando debería ir por la rama "si", o puede salirse de un bucle cuando debería repetir una vez más, o entrar en un bucle infinito, etc.

- Errores aritméticos. Ocurren cuando una variable se desborda (overflow), o se intenta una operación de división entre cero, o alguna operación aritmética está mal planteada.

- Errores de punteros. Los punteros son herramientas muy potentes que permiten la manipulación dinámica de la memoria, pero también conllevan grandes riesgos porque un puntero "descontrolado" puede hacer auténticas locuras en la memoria del ordenador, hasta el punto de colgar sistemas poco fiables (Windows 9x)
- Errores de conversión

automática de tipos. Se producen cuando C realiza una conversión automática que no teníamos prevista. Entonces el dato puede cambiar y dar al traste con la lógica del programa.

- Errores de diseño. Ocurren cuando el programa no está bien diseñado y realiza tareas diferentes de las que se pretendían. Son los peores errores, porque obligarán a

modificar una parte (o la totalidad) del trabajo realizado, debiendo, en ocasiones, volver a las primeras fases del ciclo de vida para repetir todo el proceso.

Estos y otros errores en tiempo de ejecución pueden manifestarse con distintas frecuencias:

- Siempre que se ejecuta el programa: son los más fáciles de localizar y

corregir.

- Solo cuando se introducen determinados datos de entrada: puede ser complicado dar con la secuencia de datos de entrada que provocan el error, pero una vez que la encontramos, puede localizarse con facilidad.
- Al azar: algunas veces, los programas fallan sin motivo aparente, cuando han estado

funcionando en el pasado con el mismo conjunto de datos. Son los errores más difíciles de localizar, porque ni siquiera se sabe bajo qué circunstancias ocurren.

El depurador

El depurador es un programa independiente del editor, el compilador y el enlazador. La mayoría de los compiladores disponen de uno y, además, suele estar integrado con los otros tres, de modo que desde el editor

se puede lanzar cualquiera de los otros.

El depurador es una herramienta fundamental para localizar y corregir errores en tiempo de ejecución. Para que la depuración sea cómoda, hay que activar una opción específica del compilador que permita hacer la depuración sobre el código fuente. Si no se activa, la depuración se hará sobre el código binario o, como mínimo, será imposible acceder a los identificadores de variables, lo cual complica mucho la depuración. Muchos compiladores

tienen esta opción activada por defecto.

Cada depurador tiene sus propias opciones y características, pero todos suelen coincidir en varios aspectos:

- Permiten ejecutar paso a paso cada instrucción del programa, deteniéndose antes de ejecutar la siguiente para permitirnos ver el estado de las variables o de los dispositivos de E/S.
- Permiten ver y manipular el contenido de las variables en

cualquier punto del programa.

- Permiten ver y manipular la estructura de la memoria asignada al programa y de los registros del microprocesador.
- Permiten insertar puntos de ruptura (breakpoints), es decir, puntos donde la ejecución se detendrá momentáneamente para que hagamos alguna

comprobación de las anteriormente expuestas.

Haciendo correcto uso de estas posibilidades, podemos localizar rápidamente cualquier error en tiempo de ejecución y afrontar la tarea de corregirlo.

Si quieres ver ejemplos concretos de depuradores para Windows y para Linux, puedes consultar los apéndices del libro en este momento.

Documentación

La documentación no es exactamente una fase del desarrollo del software, sino una actividad que debe practicarse a lo largo de todo el desarrollo.

La documentación que debe haberse generado al terminar un producto software es de dos tipos:

- La documentación externa la forman todos los documentos ajenos al programa: guías de instalación, guías de usuario, etc.
- La documentación interna es

la que acompaña al programa; básicamente, los comentarios.

La que más nos afecta a nosotros, como programadores, es la documentación interna, que debe elaborarse al mismo tiempo que el programa. Pero también debemos conocer la documentación externa; a veces, porque el programador debe consultarla para realizar su trabajo; otras veces, porque debe colaborar en su elaboración o modificación.

El manual técnico

El manual técnico es un documento donde queda reflejado el diseño de la aplicación, la codificación de los módulos y las pruebas realizadas. Está destinado al personal técnico (analistas y programadores) y tiene el objeto de facilitar el desarrollo y el mantenimiento del software.

El manual técnico se compone de tres grupos de documentos:

- El cuaderno de carga: es el conjunto de documentos

donde se refleja el diseño de la aplicación a partir de la fase de análisis. Entronca, pues, con la fase de diseño del ciclo de vida. Está destinado a los programadores de la aplicación, que lo utilizarán para saber qué módulos tienen que codificar, qué función realiza cada uno y cómo se comunican con los otros módulos. Es un documento fundamental para

permitir que varios programadores puedan trabajar en el mismo proyecto sin pisarse el trabajo unos a otros.

- El programa fuente: el código fuente completo también suele incluirse en la guía técnica, y debe ir autodocumentado, es decir, con comentarios dentro del código realizados por el programador.

- Juego de pruebas: se trata de un documento en el que se detallan las pruebas que se han realizado a la aplicación o a partes de la misma. Las pruebas pueden ser de tres tipos: unitarias (se prueba un módulo por separado), de integración (se prueban varios módulos que se llaman unos a otros) y de sistema (pruebas de toda la aplicación). Se debe detallar en qué ha consistido la

prueba, cuáles han sido los datos de entrada y qué resultado ha producido el programa.

El primero de los documentos anteriores, que hemos llamado cuaderno de carga, suele estar, a su vez, dividido en varias secciones:

- Tratamiento general: consiste en una descripción de las tareas que la aplicación tiene que llevar a cabo, una descripción del hardware y

del software de las máquinas donde va a funcionar y una planificación del trabajo (tiempo de desarrollo, distribución de tareas, etc)

- Diseño de datos: se trata de una especificación de los datos utilizados en la aplicación: descripciones detalladas de archivos, de tablas y relaciones (si se maneja una base de datos), etc.

- Diseño de la entrada/salida: es una descripción del interfaz con el usuario. Se detallan las pantallas, los formularios, los impresos, los controles que se deben realizar sobre las entradas de datos, etc.

- Diseño modular: consiste en una descripción de los módulos que conforman el programa y las relaciones entre ellos (quién llama a

quién, en qué orden, y qué datos se pasan unos a otros). Se utilizan diagramas de estructura, que vimos en la primera parte del libro, y descripciones de los módulos. También se debe indicar en qué archivo se almacenará cada módulo.

- Diseño de programas: es una descripción detallada de cada uno de los programas y subprogramas de la

aplicación. Puede hacerse, por ejemplo, con pseudocódigo.

El manual de usuario

Este es un documento destinado al usuario de la aplicación. La información del manual de usuario proviene del manual técnico, pero se presenta de forma comprensible para el usuario, centrándose sobre todo en los procesos de entrada/salida.

Debe estar redactado en un estilo claro,

evitando en lo posible el uso de terminología técnica. En general, todo manual de usuario debe contar con estos apartados:

- Índice de los temas
- Forma de uso de la guía
- Especificaciones hardware y software del sistema donde se vaya a usar la aplicación
- Descripción general de la aplicación
- Forma de ejecutar la

aplicación

- Orden en el que se desarrollan los procesos
- Descripción de las pantallas de entrada de datos
- Descripción de todas las pantallas y de la forma en que se pasa de una a otra
- Controles que se realizan sobre los datos y posibles mensajes de error
- Descripción de los informes

impresos

- Ejemplos de uso
- Solución de problemas frecuentes durante el uso del programa
- Ayuda en línea
- Realización de copias de seguridad de los datos

La guía de instalación

Es un documento destinado a informar al usuario o al administrador del sistema

sobre cómo poner en marcha la aplicación y cuáles son las normas de explotación.

TERCERA PARTE: ESTRUCTURAS DE DATOS ESTÁTICAS

Los tipos de datos vistos hasta ahora (enteros, reales, caracteres y lógicos) se denominan simples porque no pueden descomponerse en otros datos más simples aún.

Los tipos de datos complejos son aquellos que se componen de varios

datos simples y, por lo tanto, pueden dividirse en partes más sencillas. A los tipos de datos complejos se les llama también estructuras de datos.

Las estructuras de datos pueden ser de dos tipos:

- Estáticas: son aquéllas que ocupan un espacio determinado en la memoria del ordenador. Este espacio es invariable y lo especifica el programador durante la escritura del código fuente.

- Dinámicas: sin aquéllas cuyo espacio ocupado en la memoria puede modificarse durante la ejecución del programa.

Las estructuras estáticas son mucho más sencillas de manipular que las dinámicas, y son suficientes para resolver la mayoría de los problemas. Las estructuras dinámicas, de manejo más difícil, permiten aprovechar mejor el espacio en memoria y tienen aplicaciones más específicas.

Además, se pueden mencionar como una clase de estructura de datos diferente las estructuras externas, entendiendo como tales aquéllas que no se almacenan en la memoria principal (RAM) del ordenador, sino en alguna memoria secundaria (típicamente, un disco duro). Las estructuras externas, que también podemos denominar archivos o ficheros, son en realidad estructuras dinámicas almacenadas en memoria secundaria.

En esta parte del libro estudiaremos las estructuras de datos estáticas y su

implementación en lenguaje C, dejando para más adelante las estructuras dinámicas y externas. Nos referiremos principalmente a los arrays, tanto los unidimensionales (vectores) como los bidimensionales (matrices). Los arrays de más dimensiones son extensiones naturales de los anteriores, y también los mencionaremos brevemente. A continuación veremos las estructuras (struct) y las uniones de C, y también hablaremos de las enumeraciones y los tipos definidos por el usuario.

ARRAYS UNIDIMENSIONALES (VECTORES)

Un *array* es una agrupación de muchos datos individuales del mismo tipo bajo el mismo nombre. Cada dato individual de un array es accesible mediante un índice.

(Atención: algunos autores prefieren llamar *tablas* a los arrays y, en latinoamérica, es frecuente usar el anglicismo *arreglo*. Todos esos términos

se refieren a lo mismo)

El caso más simple de array es el array unidimensional, también llamado *vector*. Por ejemplo, un vector de números enteros es una colección de muchos números enteros a los que les adjudicamos un único identificador.

Declaración

La declaración de un vector en C se hace así:

```
tipo_de_datos
nombre_vector[número_de_element
```

Por ejemplo:

```
int serie[5];
```

La variable serie será un vector que contendrá 5 números enteros. Los 5 números reciben el mismo nombre, es decir, serie.

Se puede acceder a cada uno de los números que forman el vector escribiendo a continuación del nombre un número entre corchetes. Ese número se denomina índice. Observa el siguiente ejemplo:

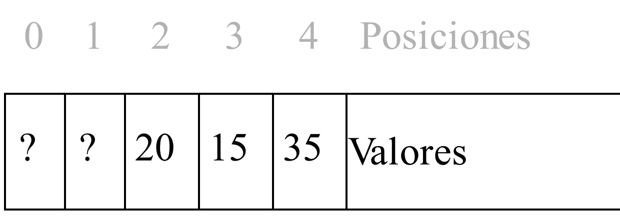
```
int serie[5];
```

```
serie[2] = 20;
serie[3] = 15;
serie[4] = serie[2] +
serie[3];
printf("%i", serie[4]);
```

El vector serie puede almacenar hasta 5 números enteros. En su posición 2 se almacena el número 20, y en su posición 3, el 15. Luego se suman ambos valores, y el resultado se almacena en la posición 4. Finalmente, se imprime en la pantalla el resultado de la suma, es decir, 35.

Es muy útil representar los vectores de forma gráfica para entenderlos mejor. El

vector serie del ejemplo anterior se puede representar así:



Observa algo muy importante: el primer elemento del vector tiene el índice 0, es decir, el primer elemento es serie[0]. Como este vector tiene 5 elementos, el último será serie[4], no serie[5]. Observa también que los elementos 0 y 1 no han sido utilizados y, por lo tanto,

tienen un valor desconocido, exactamente lo mismo que ocurre con cualquier variable de tipo simple que no se inicialice.

C no realiza comprobación de los índices de los arrays, por lo que es perfectamente posible utilizar un índice fuera del rango válido (por ejemplo, serie[7]). Es responsabilidad del programador evitar que esto ocurra, porque los efectos pueden ser desastrosos.

Como es lógico, se pueden construir

vectores cuyos elementos sean de cualquier otro tipo simple, como float o double, con la única restricción de que todos los elementos sean del mismo tipo. Los vectores de caracteres se denominan cadenas de caracteres, y por sus especiales características los estudiaremos en un epígrafe posterior.

También es posible construir vectores cuyos elementos sean de un tipo complejo. Así, podemos tener vectores de vectores o de otros tipos que iremos viendo en a lo largo de este libro.

Operaciones con vectores

Manipulación de elementos individuales

Los vectores en C deben manipularse elemento a elemento. No se pueden modificar todos los elementos a la vez.

Para asignar valores a los elementos de un vector, por lo tanto, el mecanismo es este:

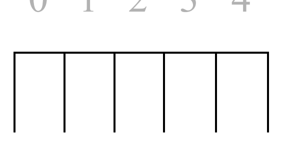
```
int serie[5];
serie[0] = 5;
serie[1] = 3;
```

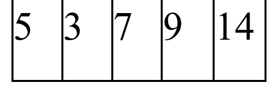
```
serie[2] = 7;
...etc...
```

La inicialización de los valores de un vector también puede hacerse conjuntamente en el momento de declararlo, así:

```
int serie[5] = {5, 3, 7, 9, 
14};
```

El resultado de esta declaración será un vector de 5 elementos de tipo entero a los que se les asigna estos valores:





Cada elemento del vector es, a todos los efectos, una variable que puede usarse independientemente de los demás elementos. Así, por ejemplo, un elemento del vector serie puede usarse en una instrucción de salida igual que cualquier variable simple de tipo int:

```
int serie[5];
serie[0] = 21;
printf("%i", serie[0]);
```

Del mismo modo, pueden usarse elementos de vector en una instrucción

de entrada. Por ejemplo:

```
int serie[5];
scanf("%i", &serie[0]);
serie[1] = serie[0] + 15;
printf("%i", serie[1]);
```

Recorrido de un vector

Una forma habitual de manipular un vector es accediendo secuencialmente a todos sus elementos, uno tras otro. Para ello, se utiliza un bucle con contador, de modo que la variable contador nos sirve como índice para acceder a cada uno de los elementos del vector.

Supongamos, por ejemplo, que tenemos un vector de 10 números enteros declarado como int v[10]; y una variable entera declarada como int i;. Por medio de un bucle, con ligeras modificaciones, podemos realizar todas estas operaciones:

1) Inicializar todos los elementos a un valor cualquiera (por ejemplo, 0):

```
for (i = 0; i <= 9; i++)
{
   v[i] = 0;
}
```

2) Inicializar todos los elementos con

valores introducidos por teclado:

```
for (i = 0; i <= 9; i++)
{
   printf("Escriba el valor
del elemento n° %i: ", i);
   scanf("%i", &v[i]);
}
```

3) Mostrar todos los elementos en la pantalla:

```
for (i = 0; i <= 9; i++)
{
   printf("El elemento n° %i
vale %i\n", i, v[i]);
}
```

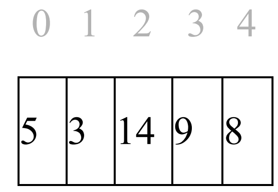
4) Realizar alguna operación que implique a todos los elementos. Por

ejemplo, sumarlos:

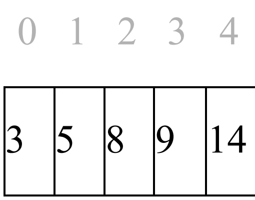
```
suma = 0;
for (i = 0; i <= 9; i++)
{
   suma = suma + v[i];
}
```

Ordenación de vectores

Una de las operaciones más típicas que se realizan con vectores es ordenar sus elementos mediante algún criterio. Por ejemplo, un vector de números enteros puede ordenarse de menor a mayor. Si el vector original es este:



...después de la ordenación nos quedará este otro vector:



Del mismo modo, se pueden ordenar los elementos con cualquier otro criterio: de mayor a menor, primero los pares y luego los impares, o cualquier otro que

nos resulte útil para resolver un problema.

Métodos de ordenación de vectores hay muchos, desde los más simples (e ineficientes) hasta los más elaborados, y constituyen un área de estudio muy interesante dentro de la algorítmica.

En la sección de actividades volveremos sobre este asunto, pero ahora mostraremos tres métodos de ordenación muy populares:

- El método de la burbuja (o de intercambio directo), un

método sencillo de entender pero bastante lento

- El método de selección directa, otro método simple e ineficiente.
- El método rápido o quicksort, un algoritmo elegante y recursivo que ordena vectores con asombrosa rapidez.

Podríamos explicar ahora cómo funciona cada método mediante una larga parrafada, pero probablemente no

se entendería gran cosa y los algoritmos son mucho más informativos por sí mismos. De modo que estudia los tres algoritmos detenidamente para intentar comprenderlos (o, al menos, los dos primeros). Dibuja en un papel un vector desordenado de pocos elementos y haz un traceo (o ejecución "a dedo") de cada función de ordenación para comprender cómo actúa. A estas alturas del libro, deberías ser capaz de entender el funcionamiento del método de la burbuja y el de selección directa. Es posible que el método rápido no puedas

comprenderlo hasta más adelante, ya que utiliza conceptos más avanzados, como la recursividad, y además se trata de un algoritmo que no es trivial en absoluto.

(Nota: LONGITUD_VECTOR es una constante que se supone definida en alguna otra parte del programa)

```
// Ordenación por INTERCAMBIO
DIRECTO (burbuja)
void ordena_vector(int
v[LONGITUD_VECTOR])
{
  int i, j, elem;
  for (i = 1; i <
LONGITUD_VECTOR; i++)
  {
```

```
     for (j = LONGITUD_VECTOR
- 1; j >=i; j--)
     {
        if (v[j-1] > v[j])
        {
          elem = v[j-1];
          v[j-1] = v[j];
          v[j] = elem;
        }
     }
  }
}
// Ordenación por SELECCIÓN
DIRECTA
void ordena_vector(int
v[LONGITUD_VECTOR])
{
  int i, j, minimo,
posicion_minimo;
  for (i = 0; i <
LONGITUD_VECTOR; i++)
```

```
    {
            minimo = v[i];
            posicion_minimo =
i;
            for (j=i; j <
LONGITUD_VECTOR; j++)
            {
                  if (v[j] <
minimo)
                  {
                          minimo
= v[j];
                          posici
= j;
                  }
            }
            v[posicion_minimo] =
v[i];
            v[i] = minimo;
    }
}
```

```
// Ordenación rápida
(QUICKSORT)
// NOTA: en esta
implementación, por
simplicidad, el vector v es
una variable global
void ordena_vector(int iz, int
de)
{
  int i, j, x, w;

  i = iz;
  j = de;
  x = v[(iz+de) / 2];
  do
  {
     while (v[i] < x) i++;
     while (x < v[j]) j--;

     if (i <= j)
     {
```

```
          w = v[i];
          v[i] = v[j];
          v[j] = w;
          i++;
          j--;
       }
     }
     while (i <= j);
     w = v[i];
     v[i] = v[de];
     v[de] = w;

     if (iz < j)
  ordena_vector(iz, j);
     if (i < de)
  ordena_vector(i, de);
  }
```

Búsqueda en vectores

En los vectores, como en todas las estructuras de datos que contienen muchos datos en su interior, también es habitual encontrarse con la operación de búsqueda.

La operación de búsqueda consiste en, dado un vector y dado un dato cualquiera, determinar si el dato está en alguna posición del vector y, si es necesario, averiguar cuál es esa posición.

La operación de búsqueda puede llegar a ser muy lenta (con el método de

búsqueda secuencial, que enseguida veremos), por lo que si en un programa tenemos que realizar búsquedas en vectores grandes repetidas veces, debemos pensar el modo de lograr que las búsquedas sean más rápidas. Por fortuna, existe una forma muy simple de hacer una búsqueda en un vector de manera tremendamente rápida (con el método llamado de búsqueda binaria, que también veremos). Pero esta forma tiene un problema: para que funcione, el vector debe estar previamente ordenado. El proceso de ordenación, como

acabamos de ver, es lento y costoso, pero, a cambio, obtendremos unos tiempos de búsqueda notablemente mejores.

Resumiendo, si necesitamos hacer búsquedas de datos en vectores en algún programa:

- Si las búsquedas se realizan pocas veces, o bien los vectores son pequeños, optaremos por la búsqueda secuencial, que no necesita ordenar previamente el

vector.

- Si las busquedas se realizan muchas veces y los vectores son de gran tamaño, optaremos por la búsqueda binaria, pero antes debemos ordenar el vector con alguno de los métodos que hemos estudiado en la sección anterior.

Búsqueda secuencial

Consiste, simplemente, en recorrer el vector desde el primer elemento hasta el

último. Si encontramos el dato buscado, podemos interrumpir la búsqueda. Si no, continuaremos hasta el final del vector.

Esta es una posible implementación en C:

```
// Búsqueda secuencial
// Buscamos el elemento "dato"
en el vector "v"
// Devolvemos la posición
donde está "dato" o, si no lo
encontramos, -1
int buscar(int
v[LONGITUD_VECTOR], int dato)
{
  int i = 0;
  int x = -1;
```

```
    while ((i < LONGITUD_VECTOR)
  && (x == -1))
    {
       if (v[i] ==
  dato)              // Lo hemos
  encontrado
            x =
  i;                         //
  Anotamos en x la posición
       i++;
    }
    return x;
  }
```

Búsqueda binaria

Para que esta búsqueda funcione, el vector debe estar previamente ordenado, como ya hemos aclarado.

El método consiste en lo siguiente:

- Supongamos que v es el vector y que contiene N elementos. Llamaremos iz a la posición del elemento izquierdo del vector (inicialmente, iz = 0). Llamaremos de a la posición del elemento derecho del vector (inicialmente, de = N-1)
- Tomamos un x igual al punto medio entre iz y de, es decir,

x = (iz/de) / 2

- Miramos el elemento v[x]. Si es el dato que buscábamos, ya hemos terminado. Si no, pueden ocurrir dos cosas:
    - Que v[x] sea mayor que el dato que buscábamos. En ese caso, y dado que el vector está ordenado, continuamos la búsqueda a la izquierda de x, haciendo que de = x.
    - Que v[x] sea menor que el dato que buscábamos. En ese

caso, continuamos la busqueda a la derecha de x, haciendo iz = x.

- Repetimos desde el paso 2 hasta que encontremos el elemento buscado o hasta que iz = de (lo que significará que el elemento no está en el vector)

He aquí una implementación en C:

```
// Búsqueda binaria
// Buscamos el elemento "busc"
en el vector "v", que debe
estar ordenado
```

```
// Devolvemos la posición
donde está “busc” o, si no lo
encontramos, -1
void buscar_binario(int
v[LONGITUD_VECTOR], int busc)
{
   int izq, der, mitad,
encontrado;
   // Iniciamos una búsqueda
binaria
   encontrado = 0;
   izq = 0;
   der = LONGITUD_VECTOR – 1;
   while ((izq < der-1) &&
(encontrado == 0))
   {
        mitad = izq + ((der -
izq) / 2);             //
Calculamos la posición “mitad”
        if (v[mitad] ==
busc)             // Lo hemos
```

```
encontrado !!
            encontrado = 1;
         if (v[mitad] >
busc)              // Seguimos
buscando en la mitad izquierda
            der = mitad;
         if (v[mitad] <
busc)              // Seguimos
buscando en la mitad derecha
            izq = mitad;
      }
      if (encontrado == 1)
         return mitad;
      else
         return -1;
   }
```

El algoritmo de búsqueda es más complejo, como puede verse, pero los tiempos de búsqueda con el método

binario son mucho más pequeños. Para un vector de N elementos, el método secuencial necesita un promedio de N/2 pasos para localizar el elemento buscado, mientras que el método binario tarda una media de log2 N pasos. ¿Qué no parece muy impresionante? Fíjate en estos datos:

Si el vector es pequeño (por ejemplo, N = 10):

- La búsqueda secuencial necesita una media de 5 pasos.

- La búsqueda binaria necesita una media de 3 pasos.

Si el vector es mediano (por ejemplo, N = 100):

- La búsqueda secuencial necesita una media de 50 pasos.
- La búsqueda binaria necesita una media de 6 ó 7 pasos.

Si el vector es grande (por ejemplo, N = 1000), la mejora de tiempo empieza a ser notable:

- La búsqueda secuencial necesita una media de 500 pasos.
- La búsqueda binaria necesita una media de… ¡10 pasos!

Si el vector es muy grande (por ejemplo, N = 100.000), la mejora de tiempo es aún mayor:

- La búsqueda secuencial necesita una media de 50.000 pasos.
- La búsqueda binaria necesita

una media de sólo 16 pasos.

La mejora en el tiempo de búsqueda es, por lo tanto, mayor cuanto mayor es el vector. Por eso dijimos que la búsqueda binaria se emplea cuando los vectores son muy grandes.

Vectores y funciones

Para pasar un vector como argumento a una función, en la llamada a la función se escribe simplemente el nombre del vector, sin índices. Esto sirve para pasar a la función la dirección de memoria

donde se almacena el primer elemento del vector (en un capítulo posterior veremos que, en realidad, el nombre de un array no es otra cosa que un puntero al primer elemento de ese array, es decir, la dirección de memoria de dicho elemento).

Como C guarda todos los elementos de los vectores en posiciones de memoria consecutivas, conociendo la dirección del primer elemento es posible acceder a todas las demás.

El hecho de que a la función se le pase

la dirección del vector y no sus valores provoca un efecto importante: que los arrays siempre se pasan por variable, nunca por valor. Esto incluye a los vectores, que son arrays unidimensionales. Por lo tanto, si algún elemento del vector se modifica en una función, también será modificado en la función desde la que fue pasado.

Como siempre se pasan por variable, no es necesario utilizar el símbolo & delante del parámetro. Por ejemplo, supongamos que serie es un vector de 15

números enteros. Para pasarlo como parámetro a una función llamada funcion1 escribiríamos simplemente esto:

```
int serie[15];
funcion1(serie);
```

En cuanto a la definición de la función, la declaración de un parámetro que en realidad es un vector se puede hacer de tres maneras diferentes:

```
void funcion1 (int
sere[15]);              /* Array
delimitado */
void funcion1 (int
serie[]);              /* Array
```

```
no delimitado */
void funcion1 (int
*serie);              /* Puntero
*/
```

El resultado de las tres declaraciones es, en principio, idéntico, porque todas indican al compilador que se va a recibir la dirección de un vector de números enteros. En la práctica, sin embargo, las dos últimas pueden darnos problemas en algunos compiladores, así que preferiremos la primera declaración (la que utiliza un array delimitado)

Dentro de la función, el vector puede

usarse del mismo modo que en el programa que la llama, es decir, no es preciso utilizar el operador asterisco.

Ejemplo: Un programa que sirve para leer 50 números por teclado, y calcular la suma, la media y la desviación típica de todos los valores. La desviación es una magnitud estadística que se calcula restando cada valor del valor medio, y calculando la media de todas esas diferencias.

Observa el siguiente programa de ejemplo detenidamente, prestando sobre

todo atención al uso de los vectores y a cómo se pasan como parámetros.

Los números de la serie se almacenarán en un vector float de 50 posiciones llamado valores. La introducción de datos en el vector se hace en la función introducir_valores(). No es necesario usar el símbolo & al llamar a la función, porque los vectores siempre se pasan por variable. Por lo tanto, al modificar el vector dentro de la función, también se modificará en el algoritmo principal.

Después, se invoca a 3 funciones que

calculan las tres magnitudes. El vector también se pasa por variable a estas funciones, ya que en C no hay modo de pasar un vector por valor.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
        float valores[50];
        float suma, media,
desviacion;
        introducir_valores(val
        suma =
calcular_suma(valores);
        media =
calcular_media(valores, suma);
        desviacion =
calcular_desviacion(valores,
```

```
media);
        printf("La suma es %f,
la media es %f y la desviación
es %f", suma, media,
desviacion);
        return 0;
}
/* Lee 50 números y los
almacena en el vector N pasado
por variable */
void introducir_valores(float
N[50])
{
        int i;
        for (i=1; i<=49; i++)
        {
                printf("Introd
el valor n° %d: ", i);
                scanf("%f",
&N[i]);
        }
```

```
}
/* Devuelve la suma todos los
elementos del vector N */
float calcular_suma(float
N[50])
{
        int i;
        float suma;
        suma = 0;
        for (i=1; i<=49; i++)
                suma = suma +
N[i];
        return suma;
}
/* Devuelve el valor medio de
los elementos del vector N.
Necesita conocer la suma de
los elementos para calcular la
media */
float calcular_media(float
N[50], float suma)
```

```
{
        int i;
        float media;
        media = suma / 50;
        return media;
}
/* Calcula la desviación
típica de los elementos del
vector N. Necesita conocer la
media para hacer los cálculos
*/
float
calcular_desviacion(float
N[50], float media)
{
        int i;
        float diferencias;
        diferencias = 0;
        for (i=1; i<=49; i++)
                diferencias =
diferencias + abs(N[i] -
```

```
media) ;
        diferencias =
diferencias / 50;
        return diferencias;
}
```

Representación interna de los vectores

En la memoria del ordenador, todos los elementos de los vectores se almacenan en posiciones de memoria consecutivas.

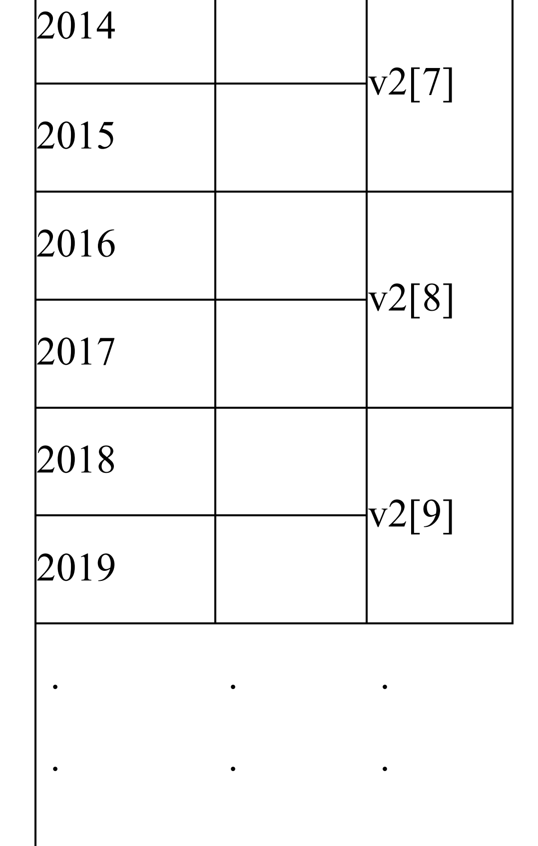
Por ejemplo, si v1 es un vector de 10 números de tipo short int (suponiendo

que cada número de dicho tipo ocupa 1 byte de memoria), el compilador asignará un espacio de memoria al elemento 0. Imaginemos que dicho espacio de memoria se ubica en la dirección 2000. Entonces, el resto de elementos del vector ocuparán la posición 2001, la 2002, la 2003, … hasta la 2009.

En esta tabla vemos una representación de ese fragmento de la memoria del ordenador. Los elementos de v1 ocupan 1 byte y, los de v2, 2 bytes cada uno.

<table>
<tr><th>Dirección</th><th>Vector v1</th><th>Vector v2</th></tr>
<tr><td>2000</td><td>v1[0]</td><td rowspan="2">v2[0]</td></tr>
<tr><td>2001</td><td>v1[1]</td></tr>
<tr><td>2002</td><td>v1[2]</td><td rowspan="2">v2[1]</td></tr>
<tr><td>2003</td><td>v1[3]</td></tr>
<tr><td>2004</td><td>v1[4]</td><td rowspan="2">v2[2]</td></tr>
<tr><td>2005</td><td>v1[5]</td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

<table>
<tr><td>2006</td><td>v1[6]</td><td rowspan="2">v2[3]</td></tr>
<tr><td>2007</td><td>v1[7]</td></tr>
<tr><td>2008</td><td>v1[8]</td><td rowspan="2">v2[4]</td></tr>
<tr><td>2009</td><td>v1[9]</td></tr>
<tr><td>2010</td><td></td><td rowspan="2">v2[5]</td></tr>
<tr><td>2011</td><td></td></tr>
<tr><td>2012</td><td></td><td rowspan="2">v2[6]</td></tr>
<tr><td>2013</td><td></td></tr>
</table>



2014		v2[7]
2015		
2016		v2[8]
2017		
2018		v2[9]
2019		
.	.	.
.	.	.

.   .   .

Por otro lado, si un vector v2 consta de 50 números de tipo int, y suponemos que los datos de este tipo ocupan 2 bytes, si el primer elemento tiene asignada la posición 2000, el siguiente estará en la

posición 2002, el siguiente en la 2004, etc.

¿Qué ocurre si se intenta acceder a un elemento del vector más allá de su límite? Dicho de otro modo, si tenemos un vector de 10 elementos, ¿qué pasa si intentamos utilizar el elemento undécimo? Lógicamente, que estaremos invadiendo el espacio de direcciones que hay más allá del límite del vector: la dirección 2010 y siguientes en el caso del vector v1, y la 2020 y siguientes en el caso del vector v2. Esas direcciones

pertenecerán a otras variables o, lo que es peor, a algún fragmento de código.

Si leemos información de ese espacio de direcciones, lo peor que puede ocurrir es que obtengamos basura. Si escribimos información en ese espacio de direcciones, el efecto es impredecible: puede que alguna otra variable cambie misteriosamente de valor, puede que el programa se detenga en un error de ejecución o, directamente, se “cuelgue”, o, en el peor de los casos, puede que el sistema entero falle y haya

que reiniciar la máquina.

CADENAS

Los vectores cuyos elementos son caracteres se denominan cadenas de caracteres o, simplemente, cadenas. Por lo tanto, una cadena de caracteres se declara así:

```
char
cadena[50];
Cadena de 50 caracteres */
```

La cadenas son sin duda los vectores que más se utilizan y, por ese motivo,

tienen ciertas peculiaridades que comentaremos en este apartado. Todo lo que hemos dicho hasta ahora sobre vectores es aplicable a las cadenas.

Declaración y manipulación de cadenas

Las cadenas pueden manipularse elemento por elemento, como cualquier vector. Por ejemplo:

```
char cadena[50];
cadena[0] = 'H';
cadena[1] = 'o';
```

```
cadena[2] = 'l';
cadena[3] = 'a';
```

Las cadenas deben tener, después de su último carácter válido, un carácter especial llamado nulo. Este carácter marca el final de la cadena. El carácter nulo se simboliza con el código \0. Por lo tanto, en el ejemplo anterior habría que agregar la siguiente línea para que la cadena estuviera completa:

```
cadena[4] = '\0';
```

Todas las cadenas deben terminar en un carácter nulo. De lo contrario, podemos tener problemas al imprimirlas en la

pantalla o al realizar con ellas cualquier otro proceso. En consecuencia, en una cadena definida como la anterior, de 50 caracteres, en realidad sólo tienen cabida 49, ya que siempre hay que reservar una posición para el carácter nulo.

La declaración de una cadena puede ir acompañada de una inicialización mediante una constante. En este caso, la constante debe ir encerrada entre comillas dobles, al tratarse de una cadena y no de caracteres sueltos. Por

ejemplo:

```
char cadena[50] = "Hola";
```

En inicializaciones de este tipo, el compilador se encarga de añadir el carácter nulo.

Por último, señalemos que no es necesario indicar el tamaño de la cadena si se inicializa al mismo tiempo que se declara. Por ejemplo, la declaración anterior puede sustituirse por esta otra:

```
char cadena[] = "Hola";
```

Esto se denomina array de longitud indeterminada. El compilador, al

encontrar una declaración así, crea una cadena del tamaño suficiente para contener todos los caracteres. Esto vale no sólo para las cadenas, sino que también es aplicable a cualquier otro tipo de array que se inicialice al mismo tiempo que se declare.

Funciones para manejo de cadenas

La mayor parte de las veces las cadenas son manipuladas mediante el uso de funciones de librería específicas. En

este apartado comentaremos las más comunes.

Funciones de lectura y escritura

Para leer por teclado una cadena de caracteres se puede utilizar también la función scanf() con la cadena de formato "%s". Como las cadenas son vectores, no es preciso anteponer el símbolo & al nombre de la variable. Sin embargo, es preferible emplear la función gets() por estar específicamente diseñada para la lectura de cadenas. Por ejemplo:

```
char cadena[50];
```

```
printf("Introduzca su nombre
");
gets(cadena);
```

Tanto scanf() como gets() insertan automáticamente el carácter "\0" al final de la cadena.

De manera análoga podemos emplear la función printf() para escribir el contenido de una cadena en la pantalla, pero preferiremos la función puts(), específica de las cadenas. Por ejemplo:

```
char cadena[50] = "Hola,
mundo";
puts(cadena);
```

Funciones de tratamiento de cadenas

Las funciones de librería ANSI C para manejar cadenas suelen empezar por las letras "str" (de "string", que significa "cadena" en inglés) y utilizan el archivo de cabecera string.h. Entre las funciones más habituales encontramos las siguientes:

strcpy()	Copia el contenido de una nulo. Su sintaxis es: **`strcpy(cadena_orige`** El siguiente ejemplo es otr

del "hola, mundo":

```
char cad1[50];
char cad2[50] = "Ho
strcpy(cad1, cad2);
strcpy(cad2, "mundo
printf("%s, %s", ca
```

strlen() Devuelve la longitud de un
caracteres de que consta, s

Por ejemplo, en este fragm
Fíjate que la variable cade
caracteres, pero strlen() só
usando, es decir, los que ha

```
char cadena[50] = "
int longitud;
longitud = strlen(c
```

```
printf("La longitud
```

strcmp() Compara dos cadenas. Dev
mayor que 0 si la primera e
o un valor menor que 0 en

```
strcmp(cadena1, cade
```

Por ejemplo:

```
char cad1[50], cad2
int comparacion;
printf("Introduzca
scanf("%s %s", cad1
comparacion = strcmp
if (comparacion == 0
   printf("Las dos
```

strcat() Concatena dos cadenas. Es
la cadena1, incluyendo el c

```
strcat(cadena1, cade
```

El resultado de este ejempl

```
char cad1[50] = "Hol
char cad2[50] = "mun
strcat(cad1, cad2);
prinft("%s", cad1);
```

Las cadenas y la validación de los datos de entrada

Una de las principales fuentes de error de los programas son los datos de entrada incorrectos. Por ejemplo, si un programa está preparado para leer un

número entero pero el usuario, por error o por mala fe, introduce un carácter, la función scanf() fallará y el programa, probablemente, se detendrá.

El programador tiene la responsabilidad de prevenir estos errores, y hay un modo sencillo de hacerlo: leyendo todos los datos de entrada como cadenas y, luego, convirtiéndolos al tipo de dato adecuado.

Observa el siguiente ejemplo. Sirve para leer un número entero por teclado, pero previniendo los errores

provocados por el usuario que antes mencionábamos. Se utiliza la función atoi(), que convierte una cadena a un número entero, y cuya sintaxis puedes encontrar en el apéndice dedicado a las funciones de uso frecuente de ANSI C.

```
int n;                          //
El número entero que se
pretende leer por teclado
char cad[50];               // La
cadena que se usará para
prevenir errores de lectura
printf("Introduzca un número
entero");
gets(cad);
No se lee un número entero,
sino una cadena
```

```
n = atoi(cad);            // Se
convierte la cadena a entero
```

ARRAYS MULTIDIMENSIONALES

Los arrays unidimensionales o vectores pueden extenderse en arrays de varias dimensiones. El ejemplo más fácil de entender es el del array bidimensional, también llamado matriz.

Arrays bidimiensionales

(matrices o tablas)

Una matriz, tabla o array bidimiensional, como un vector, es una colección de elementos individuales, todos del mismo tipo, agrupados bajo el mismo identificador. La diferencia con el vector es que, en el momento de declararlo y de acceder a cada elemento individual, debemos utilizar dos índices en lugar de uno:

```
int matriz[4][4];
```

Tenemos aquí una variable compleja llamada matriz que no consta de 4

elementos enteros, sino de 16, es decir, 4x4. Podemos representar gráficamente la matriz como una tabla:

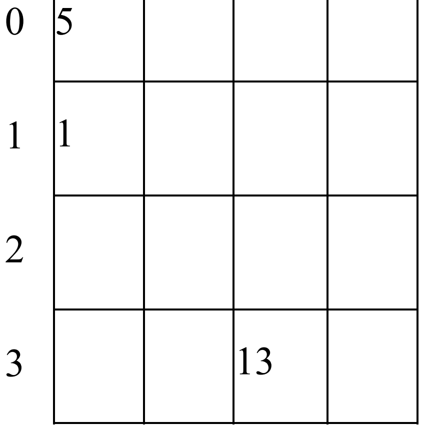
	Columnas			
Filas	0	1	2	3
0				
1				
2				
3				

Cada casilla de la tabla o matriz es identificable mediante una pareja de índices. Normalmente, el primero de los índices se refiere a la fila, y el segundo, a la columna. Por ejemplo, si hacemos estas asignaciones:

```
matriz[0][0] = 5;
matriz[1][0] = 1;
matriz[3][2] = 13;
```

…el estado en el que quedará la matriz será el siguiente:

0	1	2	3



Por descontado, los dos índices de la matriz pueden ser diferentes, obteniéndose tablas que son más anchas que altas o más altas que anchas.

Por lo demás, las matrices se utilizan exactamente igual que los vectores. A

modo de ejemplo, este sería el código para inicializar una matriz de 5x10 enteros con todos sus elementos a 0. Observa cómo se usan los dos bucles anidados para acceder a todos los elementos:

```
int m[5][10];
int i, j;
for (i = 0; i <= 4; i++)
{
    for (j = 0; j <= 9; j++)
    {
        m[i][j] = 0;
    }
}
```

Arrays de múltiples dimensiones

Del mismo modo que a los vectores se les puede añadir un segundo índice, obteniendo las matrices, se puede generalizar esta práctica, dando lugar a arrays multidimensionales. Por ejemplo, el siguiente es un array de cinco dimensiones compuesto de números enteros:

```
int ejemplo[10][10][4][5][7];
```

Estos arrays no se pueden representar gráficamente (aunque con los de tres

dimensiones se puede intentar dibujar un cubo), pero su utilización es idéntica a la de los arrays de una o dos dimensiones.

Estructuras

En los arrays, todos los elementos deben ser del mismo tipo. Pero hay ocasiones en las que debemos agrupar elementos de diversa naturaleza: para eso existen las estructuras.

Una estructura, por tanto, es una agrupación bajo el mismo nombre de

varios datos cuyos tipos pueden ser diferentes.

Declaración

Las estructuras se declaran en la zona habitual de declaración de variables, utilizando esta sintaxis:

```
struct nombre_estructura
{
   tipo1 dato1;
   tipo2 dato2;
   ...
   tipoN datoN;
};
```

Cada dato que forma parte de la

estructura se denomina miembro.

Posteriormente a la definición, se pueden declarar variables cuyo tipo sea la estructura que hayamos definido. Cada una de esas variables contendrá, en realidad, todos los datos miembro de que conste la estructura. Por ejemplo:

```
struct datos_carnet
{
   long int numero;
   char letra;
   char nombre[50];
   char apellidos[100];
};
struct datos_carnet dni;
```

La variable dni que se declara en la última línea no es de un tipo simple, como int o float, sino de un tipo complejo que acabamos de definir, llamado struct datos_carnet. Por lo tanto, una única variable (dni) va a contener varios datos agrupados en su interior (el número del DNI, la letra, el nombre y los apellidos)

Manipulación de estructuras

Una vez que se tiene una variable

compleja definida mediante una estructura surge la pregunta: ¿cómo se puede manipular cada uno de los elementos individuales (miembros) que forman parte de la estructura?

El acceso a los miembros se realiza con el nombre de la variable y el del miembro separados por un punto, así:

```
variable_estructura.miembro;
```

Continuando con el ejemplo anterior, podemos hacer lo siguiente:

```
dni.numero = 503202932;
dni.letra = 'K';
strcpy(dni.nombre, "Manuel");
```

```
strcpy(dni.apellidos, "García
García");
```

Lógicamente, para escribir un miembro en la pantalla, leerlo por teclado o realizar con él cualquier otro proceso, se utiliza la misma sintaxis.

Paso de estructuras a funciones

Al manejar estructuras en un programa modular pueden darse dos situaciones:

- Que queramos pasar una estructura completa como

parámetro a una función

- Que queramos pasar sólo un miembro de una estructura como parámetro a una función

Paso de estructuras completas como parámetros

Las variables basadas en estructuras se pueden pasar como parámetros por valor o por variable, existiendo entre ambos métodos las mismas diferencias que en los tipos de datos simples.

Para pasar, por ejemplo, la variable dni del ejemplo anterior por valor a una función llamada escribir_dni(), procederíamos de este modo:

```
escribir_dni(dni);
```

Y también puede pasarse por variable añadiendo el símbolo "&", de esta otra manera:

```
escribir_dni(&dni);
```

A su vez, la función escribir_dni() debe especificar en su declaración si el argumento se pasa por valor o por variable. El paso por valor se indica

así:

```
void escribir_dni(struct
datos_carnet dni)
```

Mientras que el paso por variable tiene esta forma (usando el símbolo " * "):

```
void escribir_dni(struct
datos_carnet* dni)
```

Dentro de la función, el acceso a los miembros de la estructura es diferente si ésta ha sido pasada por valor o por variable. Así, por ejemplo, el acceso al miembro nombre de la estructura dni, si ésta ha sido pasada por valor, se hace a la manera habitual:

```
printf("%s", dni.nombre);
```

Pero si la estructura dni se ha pasado por variable, se sustituye el punto por la flecha "->":

```
printf("%s", dni->nombre);
```

Paso de miembros de estructuras como parámetros

Los miembros de las estructuras se pueden manipular como cualquier otro dato del mismo tipo que el miembro. Por ejemplo, como dni.numero es de tipo entero largo (long int), puede realizarse

con este miembro cualquier operación que también pueda realizarse con un número entero largo, incluido el paso como parámetro a una función.

Así, para pasar por valor únicamente el miembro dni.numero a una función llamada, por ejemplo, escribir_dni(), haríamos esto:

```
escribir_dni(dni.numero);
```

En la declaración de la función, el parámetro formal debe ser de tipo long int:

```
void escribir_dni(long int
```

```
número)
```

Dentro del cuerpo de la función, la variable número puede usarse como cualquier otra variable de tipo entero.

Si lo que queremos es pasar el miembro dni.numero por variable, no por valor, lo haremos igual que con cualquier dato de tipo entero, es decir, agregando el símbolo & a la llamada:

```
escribir_dni(&dni.numero);
```

Y en la declaración de la función el parámetro debe llevar el símbolo " * ":

```
void escribir_dni(long int
```

```
*numero)
```

En este caso, cada vez que vaya a usarse el parámetro número dentro del código de la función, al estar pasado por variable debe ir precedido del símbolo " * "; por ejemplo:

```
*numero = 5;
```

Un ejemplo de utilización de estructuras

El siguiente programa es un sencillo ejemplo de manejo de estructuras. Se

encarga de almacenar los datos de un alumno en una estructura y luego mostrarlos por la pantalla. Los datos que se almacenan son, simplemente, su número de matrícula, su nombre y su edad, pero se podrían ampliar sin más que añadir otros miembros a la estructura.

La entrada de datos se hace en una función llamada leer_datos(), a la que se pasa como parámetro una variable del tipo de la estructura. Luego se hace una pequeña modificación en la edad del

alumno, para convertirla de años a meses, y se muestran los datos en la pantalla llamando a otra función, escribir_datos().

Presta especial atención a cómo se pasan los parámetros de tipo complejo a las funciones. En la primera función, leer_datos(), se pasa la estructura por variable. En la segunda, escribir_datos(), se pasan los miembros de la estructura (no la estructura completa), y además se hace por valor.

Observa también que la estructura se

define antes de la función main(). Esto la convierte en un tipo de datos global, es decir, utilizable desde cualquier punto del programa. Si la definiéramos dentro de la función main() sólo podría emplearse en esa función.

```
#include <stdio.h>
#include <string.h>
struct
datos_alumno                /*
Definición GLOBAL de la
estructura */
{
        int matricula;
        char nombre[30];
        int edad;
};
```

```
/* Prototipos de las funciones
*/
void leer_datos(struct
datos_alumno *alumno);
void escribir_datos(int matr,
char* nombre, int edad);
int main(void)
{
        struct datos_alumno
alumno;
        leer_datos(&alumno);
        alumno.edad =
alumno.edad * 12;
        escribir_datos(alumno.r
alumno.nombre, alumno.edad);
}
void leer_datos(struct
datos_alumno *alumno)
{
        printf("Introduzca el
n° de matricula :");
```

```
        scanf("%d", &alumno-
>matricula);
        printf("Introduzca el
nombre :");
        gets(alumno->nombre);
        printf("Introduzca la
edad :");
        scanf("%d", &alumno-
>edad);
}
void escribir_datos(int matr,
char* nombre, int edad)
{
        printf("MATRICULA = %d
\n", matr);
        printf("NOMBRE = %s
\n", nombre);
        printf("MESES = %d
\n", edad);
}
```

UNIONES

Las uniones son muy similares a las estructuras: se declaran de manera análoga (cambiando la palabra struct por union) y se utilizan exactamente igual. Por ejemplo:

```
union datos_carnet
{
        long int número;
        char letra;
        char nombre[50];
        char apellidos[100];
};
union datos_carnet
dni;                        /*
Declaración de la variable */
```

La diferencia radica en que todos los miembros de la union comparten el mismo espacio en memoria, de manera que sólo se puede tener almacenado uno de los miembros en cada momento.

El tamaño de la union es igual al del miembro más largo (no hagan chistes con esta frase). Supongamos que, en el ejemplo anterior, la longitud de cada miembro es:

- dni: 4 bytes (32 bits)
- letra: 1 byte (8 bits)

- nombre: 50 bytes
- apellidos: 100 bytes

Por lo tanto, la union ocupa un espacio en memoria de 100 bytes, mientras que si fuera una estructura ocuparía 155 bytes, ya que cada miembro se almacenaría en un espacio de memoria propio.

Al hacer en una unión una asignación como esta:

```
dni.número = 55340394;
```

...estamos asignando el número

55340394 a los primeros 4 bytes de la union. Si posteriormente se hace esta otra asignación:

```
strcpy(dni.nombre, "María");
```

...la cadena "María" ocupará los primeros 5 bytes de la unión y, por lo tanto, se habrá perdido el número almacenado anteriormente.

Al usar uniones, únicamente debemos acceder a los miembros que en ese momento tengan algún valor. El siguiente código, por ejemplo, funciona correctamente y escribe en la pantalla el

texto "María":

```
dni.número = 55340394;
strcpy(dni.nombre, "María");
printf("%s", dni.nombre);
```

En cambio, el siguiente fragmento no funciona bien y escribe en la pantalla un número impredecible, ya que el miembro dni.número ha perdido su valor con la segunda asignación:

```
dni.número = 55340394;
strcpy(dni.nombre, "María");
printf("%d", dni.número);
```

Por lo demás, las uniones se utilizan exactamente igual que las estructuras,

con la ventaja de que ahorran espacio en memoria. Sin embargo, al compartir todos los miembros las mismas posiciones de memoria, la utilidad de las uniones queda reducida a determinados algoritmos en los que esta limitación no representa un problema.

ENUMERACIONES

Una enumeración es un conjunto de constantes enteras. A la enumeración se le asigna un nombre que, a todos los efectos, se comporta como un nuevo tipo

de datos, de manera que las variables de ese tipo son variables enteras que solo pueden contener los valores especificados en la enumeración.

La definición de una enumeración suele hacerse así:

```
enum nombre_enumeración
{constante1 = valor1,
constante2 = valor2, ...,
constanteN = valorN };
```

Por ejemplo:

```
enum dias_semana {LUNES=1,
MARTES=2, MIERCOLES=3,
JUEVES=4, VIERNES=5, SÁBADO=6,
DOMINGO=7 };
```

Las variables que se declaren del tipo dias_semana serán, en realidad, variables enteras, pero sólo podrán recibir los valores del 1 al 7, así:

```
dias_semana dia;
dia = LUNES;
dia = 1;              /* Las dos
asignaciones son equivalentes
*/
```

Si no se especifican los valores en la enumeración, C les asigna automáticamente números enteros a partir de 0. Por ejemplo, en la siguiente definición, la constante LUNES valdrá 0, MARTES, 1, etc:

```
enum dias_semana { LUNES,
MARTES , MIÉRCOLES, JUEVES,
VIERNES, SÁBADO, DOMINGO};
```

Por último, el programador debe tener en cuenta que los identificadores utilizados en una enumeración son constantes enteras y que, por lo tanto, lo siguiente imprime en la pantalla un 2, y no la palabra "MIÉRCOLES":

```
dias_semana dia;
dia = MIERCOLES;
printf("%i", dia);
```

NUEVOS TIPOS DE DATOS

Tipos definidos por el usuario

Se pueden definir nuevos tipos de datos con la palabra reservada typedef:

```
typedef tipo nombre_tipo;
```

Por ejemplo:

```
typedef int entero;
```

A partir de esta declaración, el compilador de C reconocerá el tipo entero, que será exactamente igual al tipo predefinido int.

La definición de tipos es más práctica si

se aplica a tipos complejos. Por ejemplo:

```
typedef struct
{
        int dia;
        int mes;
        int anno;
} t_fecha;
```

Tras esta definición habrá quedado definido un nuevo tipo de datos llamado t_fecha. Por lo tanto, se podrán declarar variables de ese tipo:

```
t_fecha fecha_hoy;
t_fecha fecha_nacimiento;
```

Los identificadores de los tipos deben

cumplir todas las reglas habituales (nada de caracteres especiales ni espacios). Es una buena costumbre que el nombre de un tipo de datos empiece por la letra "t", para diferenciar así los identificadores de tipo de los identificadores de variable.

Tipos supercomplejos

Hasta ahora hemos visto varios tipos de datos simples (entero, real, carácter...) y complejos (arrays, estructuras,

uniones...). Los tipos complejos se refieren a datos compuestos por otros datos como, por ejemplo, un array de números enteros.

Sin embargo, es perfectamente posible que los datos que componen un tipo complejo sean, a su vez, de tipo complejo. Por ejemplo, es posible tener un array de estructuras, o una estructura cuyos miembros son arrays u otras estructuras.

En el siguiente ejemplo podemos ver un array unidimensional (vector) cuyos

elementos son estructuras:

```
/* Array de estructuras */
struct fecha
{
        int dia;
        int mes;
        int anno;
};
struct fecha
lista_de_fechas[100];
```

La variable lista_de_fechas es un vector de 100 elementos. Cada elemento no es un dato de tipo simple, sino una estructura fecha. Para acceder, por ejemplo, al miembro día del elemento nº 3 del array y asignarle el valor 5,

tendríamos que hacer esto:

```
lista_de_fechas[3].día = 5;
```

Otro caso bastante habitual es el de estructuras que tienen como miembros a otras estructuras. Veamos un ejemplo:

```
/* Estructura de estructuras
*/
struct s_fecha
{
        int dia;
        int mes;
        int anno;
};
struct s_hora
{
        int hh;              //
Horas
```

```
        int mm;             //
Minutos
        int ss;             //
Segundos
};
struct calendario
{
        struct s_fecha fecha;
        struct s_hora hora;
}
struct calendario fecha_hoy;
```

La variable fecha_hoy es de tipo struct calendario, que es un tipo que a su vez está compuesto de otras dos estructuras. El acceso a los miembros de fecha_hoy se hará del siguiente modo:

```
fecha_hoy.fecha.dia = 5;
```

```
fecha_hoy.fecha.mes = 12;
fecha_hoy.hora.hh = 23;
```

Estos datos de tipo supercomplejo pueden combinarse de la forma que más convenga al problema que tratamos de resolver.

CUARTA PARTE: FICHEROS

Hasta este momento, todas las operaciones de entrada y salida de datos de nuestros programas se han hecho a través del teclado (entrada) y la pantalla (salida). Estos son los dispositivos de entrada y salida por defecto, pero también se pueden enviar datos hacia un archivo, o recibirlos de él.

Además, todos los datos que hemos manejado, ya sea mediante tipos de datos simples o estructuras complejas, han estado alojados en la memoria principal del ordenador, de manera que al apagar éste, o antes, al terminar el programa, toda esa información se perdía. Como es natural, también es posible almacenar datos en memoria secundaria, es decir, en dispositivos tales como discos duros, discos flexibles, discos ópticos, memorias USB, etc. Estas memorias se caracterizan por ser más lentas que la

memoria principal del ordenador, pero también disponen de más espacio de almacenamiento, y no son volátiles, es decir, no pierden su contenido al desconectar la corriente eléctrica.

Por motivos históricos, estas memorias secundarias agrupan los datos en estructuras que denominamos archivos o ficheros (en inglés, *files*). La traducción correcta en castellano es *archivo*, pero está más extendido el anglicismo *fichero*, por lo que será la palabra que usaremos preferentemente en adelante.

En esta parte del libro veremos cómo podemos crear y manipular los ficheros en memoria secundaria desde nuestros programas en C. No nos limitaremos a mostrar un simple listado de funciones de librería, sino que profundizaremos mucho más, discutiendo las diferencias entre las posibles organizaciones lógicas de ficheros (secuenciales, aleatorios, indexados) y proponiendo implementaciones de cada una de ellas. También caracterizaremos con detalle los ficheros de texto y los ficheros binarios.

LOS ARCHIVOS O FICHEROS

Todos estamos familiarizados con los ficheros. Tenemos toneladas de ellos distribuidos por las carpetas de nuestro disco duro, ¿no es cierto? Pero, ¿alguna vez te has preguntado cómo están organizados por dentro esos ficheros?

Con los ficheros de texto plano es fácil imaginarlo: son largas colecciones de caracteres, almacenados uno tras otro en el disco duro, codificados en ASCII o

algún otro sistema de codificación. Eso es casi exacto. Pero, ¿y los otros ficheros? ¿Y los archivos ejecutables, los archivos de bases de datos, las imágenes, los vídeos? ¿Cómo están construidas sus tripas? ¿Cómo se pueden crear y manipular esos ficheros?

A esas preguntas vamos a tratar de responder en las siguientes páginas. Y, como suele ser habitual, empezaremos por el principio, es decir, por los conceptos más fundamentales: los registros y los campos.

Ficheros, registros y campos

Un archivo o fichero es un conjunto de información relacionada entre sí y estructurada en unidades más pequeñas, llamadas registros.

Cada registro debe contener datos pertenecientes a una misma cosa. Además, cada registro es un estructura de datos, es decir, está compuesto de otros datos más simples, que llamaremos campos.

Un campo es cada uno de los elementos que constituyen un registro. Cada campo se caracteriza por un identificador que lo distingue de los otros campos del registro, y por el tipo de dato que tiene asociado, que, a su vez, puede ser simple (número entero, carácter, lógico, etc) o compuesto (cadena de caracteres, fecha, vector, etc).

Observa el siguiente ejemplo de fichero. Contiene información relacionada entre sí: los datos personales de un conjunto de personas. Toda esa información está

distribuida en registros, que son cada una de las filas de la tabla. Cada registro, por tanto, contiene los datos pertenecientes a una sola persona. Los registros se dividen en campos, que son cada una de las unidades de información que contiene cada registro, es decir, cada una de las columnas de la tabla.

DNI	Nombre	Apellidos	Te
1111	Salvador	Pérez Pérez	23

3333	Margarita	Sánchez Flor	23(
…	…	…	…

(Nota: el DNI que aparece en la tabla se supone que es el número del Documento Nacional de Identidad de países como Argentina, España o Perú; en otros países latinoamericanos, el documento equivalente recibe denominaciones como Cédula de Identidad, Carné de Identidad, Cédula de Ciudadanía, etc)

Si el tipo de dato de un campo es

complejo, el campo puede dividirse en subcampos. Por ejemplo, si un campo contiene una fecha, se puede dividir en tres subcampos que contengan, respectivamente, el día, el mes y el año.

Para diferenciar a un registro de otro es conveniente que alguno de los campos tenga un valor distinto en todos los registros del archivo. Este campo, que identifica unívocamente cada registro, se denomina campo clave o, simplemente, clave. En el ejemplo anterior, el campo clave puede ser DNI, ya que será

diferente para cada una de las personas que forman el archivo.

Registros físicos y registros lógicos

Un registro físico, también llamado bloque, es diferente de los registros que acabamos de ver y que, para diferenciarlos, se denominan registros lógicos.

El registro físico es la cantidad de información que el sistema operativo puede enviar o recibir del soporte de

memoria secundaria en una operación de escritura o lectura. Esta cantidad depende del hardware.

La mayor parte de las veces, cuando un programador dice "registro" se refiere al registro lógico. Entonces, ¿tienen alguna importancia, desde nuestro punto de vista, los registros físicos? Vamos a responder a esa pregunta enseguida.

El registro físico puede ser mayor que el registro lógico, con lo cual, en una sola operación de lectura o escritura, se podrían transferir varios registros

lógicos. También puede ocurrir lo contrario, es decir, que el registro físico sea de menor tamaño que el lógico, lo que haría que para transferir un registro lógico fueran necesarias varias operaciones de lectura o escritura.

Se llama factor de bloqueo al número de registros lógicos contenidos en un registro físico.

Como ejemplo vamos a calcular el factor de bloqueo del archivo del epígrafe anterior. Supongamos que el tamaño del registro físico es de 512

bytes (es decir, en una sola lectura o escritura del dispositivo de almacenamiento se pueden transferir 512 bytes) y el registro lógico ocupa 128 bytes, calculados de esta manera (recuerda que el tamaño exacto que requiere cada tipo simple depende del sistema operativo, del compilador y del hardware de la máquina).

- Campo DNI (10 caracteres) = 10 bytes
- Campo Nombre (30 caracteres) = 30 bytes

- Campo Apellidos (40 caracteres) = 40 bytes
- Campo Teléfono (entero largo) = 8 bytes
- Campo Dirección (40 caracteres) = 40 bytes
- TOTAL = 128 bytes

En estas condiciones, el factor de bloqueo es 4, que es el resultado de dividir 512 (tamaño del registro físico) entre 128 (tamaño del registro lógico). En cada registro físico caben

exactamente 4 registros lógicos, sin que sobre ningún byte, porque la división de 512 entre 128 es exacta, pero puede ocurrir que no sea así.

Por ejemplo, si el registro lógico ocupase 126 bytes en lugar de 128, en cada registro físico cabrían 4 registros lógicos pero sobrarían 8 bytes. Esto tiene una gran importancia desde el punto de vista del rendimiento, ya que cada acceso a la memoria secundaria requiere bastante tiempo y, por tanto, éstos deben reducirse al máximo.

Tipos de registros

(Recuerda: cuando hablemos de "registro" a secas nos estaremos refiriendo al registro lógico, no al físico)

Dependiendo de la longitud de los campos que forman cada registro podemos clasificar éstos en:

A) Registros de longitud fija

Son los que ocupan siempre el mismo espacio a lo largo de todo el archivo (en el ejemplo anterior, 128 bytes). Dentro

de estos registros, podemos encontrar varias posibilidades:

- Igual número de campos por registro e igual longitud de todos los campos
- Igual número de campos por registro y distinta longitud de cada campo, aunque igual en todos los registros
- Igual número de campos por registro y distinta longitud de cada campo, pudiendo ser diferente en cada registro

- Distinto número de campos por registro y distinta longitud de cada campo en cada registro

B) Registros de longitud variable

Aunque es menos habitual, pudiera ocurrir que cada registro del archivo tuviera una longitud propia y diferente del resto. En este tipo de archivos es necesario programar algún mecanismo para averiguar cuál es el principio y el final de cada registro.

Operaciones con archivos

En un archivo se puede realizar operaciones sobre cada registro individual o bien sobre todo el archivo, es decir, sobre todos los registros a la vez.

A) Operaciones con registros individuales

- Inserción (alta): consiste en añadir un registro al fichero. El registro puede añadirse al

final del fichero o entre dos registros que ya existieran previamente.

- Borrado (baja): consiste en eliminar un registro existente.
- Modificación: consiste en cambiar el dato almacenado en uno o varios de los campos del registro
- Consulta: consiste en leer el dato almacenado en uno o varios de los campos del registro.

B) Operaciones sobre el archivo completo

Además de manipular cada componente del archivo (registros y campos), también se pueden llevar a cabo operaciones con la totalidad del archivo, como:

- Creación: La creación del archivo consiste en crear una entrada en el soporte de memoria secundaria y asignarle un nombre para identificar en el futuro a los

datos que contiene.

- Apertura: Antes de trabajar con un archivo es necesario abrirlo, creándose así un canal de comunicación entre el programa y el archivo a través del cuál se pueden leer y escribir datos. Los archivos sólo deben permanecer abiertos el tiempo estrictamente necesario.
- Cierre: Es importante cerrar el canal de comunicación con

el archivo cuando no va a usarse en un futuro inmediato, porque todos los sistemas limitan el número máximo de archivos que pueden estar abiertos simultáneamente. También es importante porque evita un acceso accidental al archivo que pueda deteriorar la información almacenada en él.

- Ordenación: Permite

establecer un orden entre los registros del archivo.

- Copiado: Crea un nuevo archivo con la misma estructura y contenido que el fichero original.
- Concatenación: Consiste en crear un archivo nuevo que contenga los registros de otros dos archivos previamente existentes, de manera que primero aparezcan todos los registros

de un archivo y, a continuación, todos los del otro.

- Mezcla: Parecida a la concatenación, pero el archivo resultante contendrá todos los registros de los dos archivos originales mezclados y ordenados.
- Compactación: Esta operación sólo se realiza sobre archivos en los cuales el borrado de registros se ha

realizado sin eliminar físicamente el registro, sino únicamente marcándolo como borrado para no procesarlo. Después de la compactación, todos los registros marcados como borrados quedan borrados físicamente, con lo que se libera espacio en el dispositivo de almacenamiento.

- Borrado: Es la operación contraria a la creación, ya

que elimina la entrada en el dispositivo de almacenamiento, con lo que se pierde toda la información almacenada en el archivo.

ORGANIZACIÓN DE ARCHIVOS

La organización de los archivos es la forma en que los datos son estructurados y almacenados en el dispositivo de almacenamiento. El tipo de organización se establece durante la fase de creación

del archivo y es invariable durante toda su vida. La organización puede ser secuencial o relativa (o una combinación de ambas), como enseguida veremos.

El tipo de acceso al archivo es el procedimiento que se sigue para situarnos sobre un registro concreto para hacer alguna operación con él. Esto es lo que realmente le interesa al programador: cómo acceder a los registros de archivo. El tipo de acceso está condicionado por el tipo de

organización física del archivo.

A lo largo de todo este apartado estudiaremos los tipos de organización. Después veremos las funciones de C para acceder a archivos y, el resto de este capítulo lo dedicaremos a los tipos de acceso a archivos que se pueden realizar desde C.

Archivos de organización secuencial

La forma mas simple de estructura de

archivo es el archivo secuencial. En este tipo de archivo, los registros se sitúan físicamente en el dispositivo en el orden en el que se van escribiendo, uno tras otro y sin dejar huecos entre sí. El acceso a los registros también debe hacerse en orden, de modo que para acceder al registro N es necesario pasar primero por el registro 1, luego por el 2, luego por el 3, y así hasta llegar al registo N.

Los archivos secuenciales se utilizaban mucho cuando el soporte de

almacenamiento masivo más usual era la cinta magnética. Hoy día siguen siendo utilizados por su simplicidad y porque son suficientemente útiles en muchas ocasiones (por ejemplo, en aplicaciones de proceso de lotes). Pero si el programa necesita acceder a registros individuales y no consecutivos, los archivos secuenciales ofrecen un rendimiento pobre y son preferibles los archivos directos, que luego veremos.

Los archivos secuenciales tienen un indicador de posición (o cursor) que

señala qué registro fue el último que se accedió. Al abrir el archivo, el indicador se sitúa en el primer campo del primer registro. Cada acceso sobre el archivo desplazará el indicador de posición hacia el siguiente registro, hasta que ya no haya más registros que leer.

Cuando un archivo secuencial se abre para escribir datos en él, el indicador de posición se sitúa justo después del último byte del mismo, de manera que los datos sólo se pueden añadir al final.

La organización secuencial cuenta con varias ventajas:

- Es la más sencilla de manejar para el programador.
- Si hay que acceder a un conjunto de registros consecutivos, o a todo el archivo, es el método más rápido.
- No deja espacios entre registro y registro, por lo que se optimiza el uso del espacio en la memoria

secundaria.

Pero también tiene algunos inconvenientes serios:

- Para consultar datos individuales, hay que recorrer todo el archivo desde el principio. Es decir, el acceso a registros individuales es, en general, lento.
- Las operaciones de inserción y eliminación de registros solo pueden hacerse al final

del archivo. Hacerlas con registros intermedios representa mover grandes bloques de información y, por lo tanto, consumir mucho tiempo.

Archivos de organización relativa: hashing

La organización relativa es más compleja que la secuencial.

Consiste en guardar físicamente los

registros en lugares de la memoria secundaria no consecutivos. Pero, entonces, ¿cómo podemos encontrar dónde está cada registro?

La única solución es utilizar un campo clave de entre todos los del registro. Ese campo clave, que suele ser numérico, permite averiguar la dirección física donde está almacenado el registro en la memoria secundaria mediante un algoritmo de transformación. Por eso, la clave suele denominarse dirección de memoria lógica, para distinguirlo de la

dirección de memoria física donde efectivamente se encuentra guardado el registro.

Esta transformación de claves para obtener direcciones físicas se denomina hashing. Más abajo encontrarás un ejemplo muy sencillo de hashing que te ayudará a entender todo esto.

Los archivos relativos son más versátiles que los secuenciales porque permiten acceder a cualquier parte del fichero en cualquier momento, como si fueran arrays. Las operaciones de

lectura y escritura pueden hacerse en cualquier punto del archivo.

Los archivos con organización relativa tienen dos variantes: los archivos directos y los archivos aleatorios (o indirectos). En los siguientes epígrafes estudiaremos cada tipo por separado.

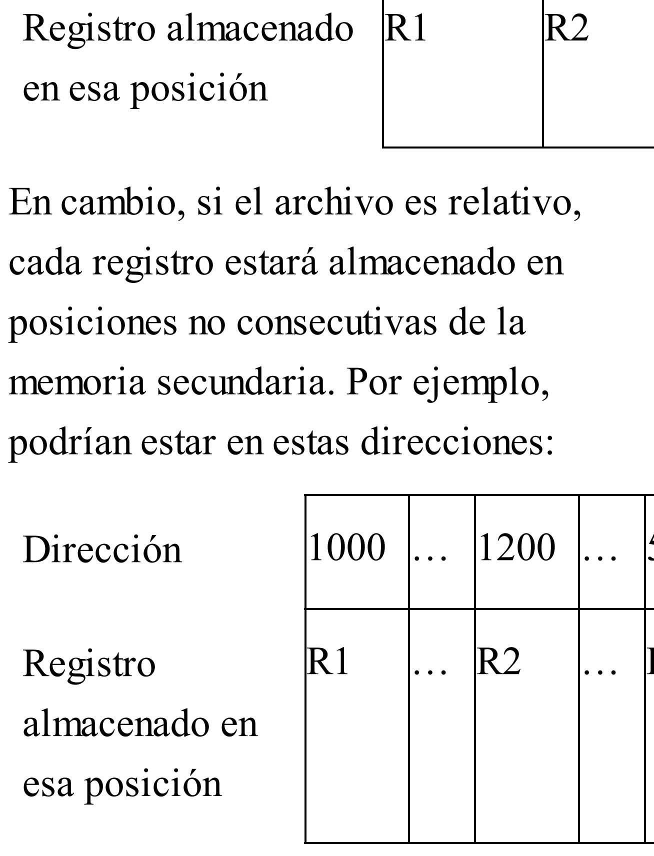
Ejemplo de hashing

Antes de continuar, vamos a tratar de entender bien la técnica de hashing con un sencillo ejemplo.

Supongamos que un archivo almacenado

en una memoria secundaria contiene 5 registros, que llamaremos R1, R2, R3, R4 y R5. En un archivo secuencial, los cinco registros estarán almacenados en posiciones consecutivas de la memoria. Si R1 se guarda, por ejemplo, en la dirección 1000 de la memoria secundaria y cada registro lógico ocupa exactamente un registro físico, tendremos que los registros estarán guardados en estas direcciones:

Dirección	1000	1001



Registro almacenado en esa posición	R1	R2

En cambio, si el archivo es relativo, cada registro estará almacenado en posiciones no consecutivas de la memoria secundaria. Por ejemplo, podrían estar en estas direcciones:

Dirección	1000	…	1200	…	
Registro almacenado en esa posición	R1	…	R2	…	

El problema con este sistema de almacenamiento es cómo localizar los registros en la memoria secundaria. Para eso se utiliza el hashing. Cada registro debe tener un campo clave (que denominaremos R1.clave, R2.clave, etc). El hashing consiste en aplicar una función de transformación a cada clave.

Supongamos que las claves de los registros de este ejemplo son:

```
R1.clave = 500
R2.clave = 600
R3.clave = 2860
R4.clave = 3152
R5.clave = 3159
```

Entonces, la función hash aplicada a este archivo para averiguar la dirección de cada registro ha sido

```
f(clave) = clave x 2
```

Probemos a aplicar la función hash al primer registro (R1):

```
f(R1.clave) = 500 x 2 = 1000
```

Efectivamente, aplicando la función hash a la clave de R1 (500), hemos obtenido su dirección de almacenamiento en memoria secundaria (1000).

Si probamos con otros registros, esta función hash también nos devuelve la

dirección. Por ejemplo, con R3:

```
f(R3.clave) = 2860 x 2 = 5720
```

Si lo compruebas, 5720 es la dirección donde está guardado el registro R3.

Archivos de organización relativa directa

Entre los archivos con organización relativa los más sencillos son los directos.

En ellos, el campo clave de cada

registro debe ser de tipo numérico, e identifica directamente el registro físico donde está almacenado. La función hash, en este caso, es la más simple posible, ya que no transforma la clave:

```
f(clave) = clave
```

En el ejemplo anterior, el registro R1 se almacenaría en la dirección 500, el R2 en la 600, el R3 en la 2860, etc, ya que:

```
f(R1.clave) = clave = 500
f(R2.clave) = clave = 600
f(R3.clave) = clave = 2860
```

El valor de la clave está en relación con la capacidad máxima del dispositivo de

almacenamiento, no pudiendo almacenar registros cuya clave esté por encima de este límite.

En estos archivos no puede haber dos registros con la misma clave, porque ambos ocuparían la misma posición física, solapándose. Esto es lo que se llama una colisión y debe ser evitada.

Las ventajas de los archivos directos son:

- Permite acceder al archivo de dos maneras: directamente (a través de la clave de cada

registro) y secuencialmente.

- Permite realizar operaciones de lectura y escritura simultáneamente.
- Son muy rápidos al tratar registros individuales.

Los inconvenientes principales son:

- El acceso secuencial, del principio al fin del fichero, puede ser muy lento porque podemos encontrarnos con muchos huecos, es decir,

posiciones que no están siendo usadas. Existen técnicas de programación avanzadas para el acceso secuencial eficiente a ficheros directos.

- Relacionado con la anterior, pueden quedar muchos huecos libres en el dispositivo de memoria secundaria, desaprovechándose el espacio.

Archivos de organización relativa aleatoria (o indirecta)

Se denominan así a los archivos relativos que empleen alguna función hash para transformar la clave y conseguir así la dirección física.

La función hash puede ser muy sencilla, como la del ejemplo que vimos en el apartado anterior (que consistía en multiplicar la clave por 2 para obtener la dirección física) o más complicada

(por ejemplo, algo como f(clave) = clave * num_primo + clave, donde "num_primo" es el número primo más cercano que exista a 2n, siendo n el número de bits de la clave), pero el principio es el mismo: transformar la clave para obtener la dirección física.

Dependiendo de la función hash empleada pueden surgir colisiones, es decir, claves que proporcionan la misma dirección física.

Por ejemplo, si la función hash es f(clave) = clave / 2 (división entera),

tendremos que los registros con clave 500 y 501 intentarán ocupar la misma dirección física: la 250. Es responsabilidad del programador evitar estas colisiones y, en caso de que lleguen a producirse, detectarlas y programar algún mecanismo que las resuelva.

Otras funciones hash, como la ya vista f(clave) = clave x 2, no producen colisiones, pero en cambio provocan que muchas direcciones físicas no sean utilizadas, con lo que se desaprovecha

el espacio de almacenamiento.

Por lo tanto, la elección de una función hash adecuada es crucial para el correcto rendimiento y funcionamiento de este tipo de archivos. Existen multitud de funciones hash adaptadas a los más diversos problemas que ofrecen un máximo aprovechamiento del espacio y un mínimo número de colisiones, pero su estudio excede a las posibilidades de este texto.

Las ventajas de los archivos aleatorios son similares a las de los directos, y

entre los inconvenientes podemos quitar el de dejar muchos huecos libres, siempre que, como hemos visto, la función hash elegida sea adecuada.

Archivos de organización indexada

Se denomina archivo de organización indexada (o indizada) a una mezcla entre la organizaciones secuencial y relativa, que pretende aprovechar las ventajas de las dos organizaciones, evitando al

mismo tiempo sus inconvenientes.

Los archivos indexados están divididos en tres zonas o áreas:

1) El área primaria. En esta área se encuentran almacenados los registros del archivo secuencial. Es decir, el área primaria es, en realidad, un archivo secuencial corriente. Los registros deben estar ordenados (normalmente, se hará en orden creciente según sus claves)

El área primaria suele estar segmentada, es decir, dividida en trozos o segmentos.

En cada segmento se almacenan N registros en posiciones de memoria consecutivas. Para acceder a un registro individual, primero hay que acceder a su segmento y, una vez localizado el segmento, buscar secuencialmente el registro concreto.

2) El área de índices. Se trata, en realidad, de un segundo archivo secuencial agregado al primero. Pero es un archivo especial, cuyos registros solo tienen dos campos: uno contiene la clave del último registro de cada segmento, y

otro contiene la dirección física de comienzo de cada segmento.

3) El área de excedentes. Puede ocurrir que los segmentos del área primaria se llenen y no puedan contener algún registro. Esos registros van a parar a un área de excedentes u overflow.

Para acceder a un registro concreto en un archivo indexado, el procedimiento es el siguiente:

- Primero, buscamos secuencialmente en el área de índices la dirección de

comienzo del segmento donde está el registro que queremos buscar.

- Segundo, hacemos un acceso directo al primer registro del segmento.
- Después hacemos un recorrido secuencial dentro del segmento hasta localizar el registro.
- Si el registro no se encuentra, acudimos al área de excedentes y hacemos un

nuevo recorrido secuencial en ella para intentar localizarlo allí.

Observa que los archivos indexados mezclan los accesos secuenciales con los accesos directos.

Ejemplo de archivo indexado

Vamos a mostrar un ejemplo para tratar de entender correctamente esta organización de archivo.

Supongamos un archivo de datos personales de los alumnos que conste de

estos 10 registros:

DNI (clave)	Nombre
1111	Arturo Pérez
1232	Miguel Ruiz
2100	Antonia Camacho
2503	Silvia Ortiz
3330	Sonia del Pino
5362	José Anguita

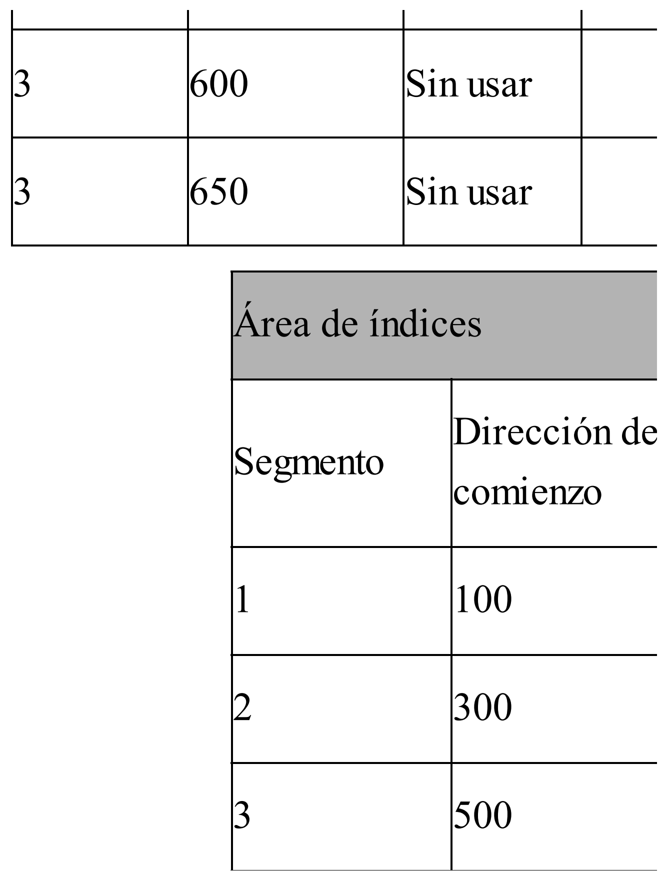
6300	Ana Zamora
6705	Susana Hernández
7020	Rodrigo Sánchez
9000	Natalia Vázquez

Imaginemos que cada segmento tiene 4 registros. Por lo tanto, el archivo se dividirá en 3 segmentos. Si suponemos que cada registro ocupa 50 bytes en memoria secundaria, y que el principio del archivo está en la dirección 100 de dicha memoria, el archivo físico tendrá

este aspecto:

Área primaria			
Segmento	Dirección física	Clave (DNI)	Cont
1	100	1111	Artur
1	150	1232	Migu
1	200	2100	Anto Cama
1	250	2503	Silvi

2	300	3330	Soni Pino
2	350	5362	José
2	400	6300	Ana
2	450	6705	Susa Hern
3	500	7020	Rodr Sánc
3	550	9000	Nata Vázq



3	600	Sin usar	
3	650	Sin usar	

Área de índices	
Segmento	Dirección de comienzo
1	100
2	300
3	500

Observa primero el área primaria: los registros están dispuestos en orden creciente según la clave (que, en este caso, es el campo DNI). A la izquierda aparece la dirección física donde comienza cada registro. Fíjate también en que los registros están agrupados en tres segmentos.

Luego fíjate en el área de índices: contienen una lista de segmentos, guardando la dirección de comienzo del segmento y la clave del último registro de ese segmento.

Para acceder, por ejemplo, al registro cuya clave es 5362, el proceso es el siguiente:

- Buscar en el área de índices secuencialmente, es decir, desde la primera fila, hasta localizar un registro mayor que el que estamos buscando. Eso ocurre en la segunda fila, pues la clave del último registro es 6705. Por lo tanto, sabemos que el registro buscado debe de estar en el

segmento 2.

- Acceder de forma directa a la dirección 300 del área primaria, que es de comienzo del segmento 2. Esa dirección la conocemos gracias a que está guardada en el área de índices.

- Buscar en el área primaria secuencialmente a partir de la dirección 300, hasta localizar el registro buscado, que ocupa la segunda

posición dentro de ese segmento.

Fíjate que han sido necesarios, en total, 4 accesos secuenciales y 1 directo. Si hubiésemos hecho una búsqueda secuencial, hubiéramos necesitado 6 accesos secuenciales desde el principio del archivo. Esto puede no parecer una gran ventaja, pero ahora piensa qué pasaría si el archivo tuviera más segmentos y el registro buscado estuviera muy lejos del principio del archivo. Cuanto mayor es el tamaño del

archivo y más lejos del principio está el registro, más ventajosa resulta la organización indexada frente a la secuencial.

Los archivos en C

Hasta ahora hemos visto las formas de organización de archivos. En este apartado y el siguiente vamos a estudiar las funciones de C para acceder a los archivos.

En principio, quédate con esta idea: el lenguaje C sólo puede manejar archivos

secuenciales y directos. La mayoría de sus funciones sirven para ambos tipos de organización, comportándose de forma ligeramente distinta con una y con otra. Y, luego, existen algunas funciones exclusivamente para archivos secuenciales, y otras para archivos directos, como iremos viendo. Por último, combinando adecuadamente los accesos directos con los secuenciales, se puede lograr en C un acceso indexado, aunque es tarea del programador manejar los índices y todas las complicaciones de este método de

organización.

Clasificación de los archivos en C

Además de los tipos de archivos que ya hemos visto (según su organización: secuenciales y relativos con todas sus variedades), en C podemos hacer otras dos clasificaciones de los archivos:

1) Según la dirección del flujo de datos:

- De entrada: los datos se leen por el programa desde el

archivo.

- De salida: los datos se escriben por el programa hacia el archivo.
- De entrada/salida: los datos pueden se escritos o leídos.

2) Según el tipo de valores permitidos a cada byte:

- De texto: sólo permiten guardar caracteres o, mejor dicho, su código ASCII. Para guardar información

numérica en un archivo de texto es necesario convertirla a caracteres.

- Binarios: guardan información binaria de cualquier tipo.
- Un poco más adelante, cuando ya conozcamos el manejo que de los archivos se puede hacer con C, estudiaremos con más detenimiento las diferencias entre archivos binarios y de

texto.

Flujos

Un flujo (o *stream* en inglés) es una corriente de datos que fluyen desde un origen, llamado productor, y son recibidos por un destinatario, llamado consumidor. Entre el origen y el destino debe existir una conexión de tal naturaleza que permita la transmisión de datos.

En C, para recibir datos desde cualquier dispositivo de entrada (productor) o

para enviar datos a cualquier dispositivo de salida (consumidor), es necesario establecer un canal que permita recibir y enviar esos datos. Este canal es lo que llamamos flujo.

En todos los programas escritos en C existen tres flujos o canales abiertos automáticamente:

- *stdin*: es el flujo de entrada estándar, es decir, el canal de comunicación con el teclado.
- *stdout*: es el flujo de salida estándar, es decir, el canal de

comunicación con la pantalla.

- *stderr*: es el flujo por el que se envían los mensajes de error; como éstos aparecen por defecto en la pantalla, se trata de un segundo canal de comunicación con la pantalla.

Estos flujos no hay que abrirlos, cerrarlos, definirlos ni nada parecido, porque existen de manera automática en todos los programas. Así, cuando invocamos a la función printf(), estamos enviando datos a través del flujo stdout,

del mismo modo que cuando llamamos a scanf() estamos leyendo datos a través del flujo stdin.

Cada vez que utilicemos un archivo en memoria secundaria será necesario crear un flujo nuevo, es decir, un canal a través del cual podamos leer o escribir datos del archivo. En todas las funciones de lectura y escritura deberemos especificar, además de los datos que queremos leer o escribir, el flujo a través del cual deseamos hacerlo.

Archivos y buffers

Para acceder a los archivos, por tanto, es necesario crear flujos nuevos a parte de stdin y stdout. Crear un flujo con un archivo se denomina comúnmente "abrir el archivo". Cuando ya no va a ser necesario escribir ni leer más datos del archivo, el flujo debe destruirse en la operación denominada "cerrar el archivo".

El acceso a los archivos se hace a través de un buffer. Se puede pensar en un buffer como si fuera un array donde se

van almacenando los datos dirigidos al archivo, o los datos que el archivo envía hacia el programa. Esos datos se van colocando en el buffer hasta que éste se llena, y sólo entonces pasan efectivamente a su destinatario. También es posible forzar el vaciado del buffer antes de que se llene invocando a determinadas funciones que luego veremos.

En resumen: cuando se envían datos a través de un flujo, éstos no se escriben inmediatamente en el archivo, sino que

se van acumulando en el buffer, y sólo cuando el buffer está lleno los datos se graban realmente en el archivo. En ese momento el buffer queda vacío y listo para seguir recibiendo datos. Al cerrar el archivo, se terminan de escribir los últimos datos que pudieran quedar en el buffer.

FUNCIONES DE C PARA LA MANIPULACIÓN DE ARCHIVOS

Los prototipos de las funciones de

entrada / salida que vamos a ver en esta sección se encuentran en stdio.h

Apertura

Para usar un archivo desde un programa en C, tanto secuencial como directo, lo primero que hay que hacer es abrirlo. Esto crea un flujo que conecta nuestro programa con el archivo.

La función para abrir archivos es fopen(), que tiene esta sintaxis:

```
FILE *fopen(char
*nombre_archivo, char *modo);
```

El argumento nombre_archivo es el identificador del archivo que se pretende abrir, es decir, su nombre en el dispositivo de memoria secundaria. El argumento modo define qué se va a hacer con el archivo: leer datos de él, escribir datos en su interior, o ambas cosas. Además, el modo también sirve para especificar si el archivo se va a manejar como un archivo secuencial o como un archivo directo. Los valores que puede tomar el argumento se muestran en una tabla más abajo.

La función fopen() devuelve un puntero a archivo. El tipo FILE está definido en stdio.h, por lo que se puede utilizar en cualquier programa que incluya dicha cabecera. El puntero devuelto por fopen() será fundamental para escribir y leer datos del archivo más adelante. Si fopen(), por la razón que sea, no puede abrir el archivo, devolverá un puntero a NULL.

Modo	Significado
	Abre el archivo existente

"r"	secuencial. El archivo deb previamente.
"w"	Crea un archivo nuevo par secuencial. ¡Cuidado! Si e se borrará y se creará uno
"a"	Abre un archivo existente modo secuencial, añadien de los que haya. Si el arch crea.
"r+"	Abre el archivo para lectu directo. El archivo debe e

"w+"	Crea un archivo para lectu directo. Si el archivo ya e crea de nuevo.
"a+"	Abre un archivo existente en modo directo. Si el arc crea.

A todos estos modos se les puede añadir la letra "b" si el archivo es binario, o "t" si es de texto. Por ejemplo: "rb", "w+t", "a+b", etc. Si no se añade "b" ni "t", se supone que el archivo es de texto. Los archivos de texto deben usarse para

almacenar texto ASCII. Los archivos binarios suelen utilizarse para guardar información más compleja, aunque también pueden guardar texto.

Por ejemplo:

```
FILE* archivo;
archivo = fopen("datos.txt",
"r");
if (archivo == NULL)
puts("Error al abrir el
archivo");
```

El archivo "datos.txt" se abre para lectura. No se podrán escribir datos en él, sólo leerlos. La variable puntero archivo será imprescindible para actuar

más adelante sobre el archivo. Fíjate cómo se comprueba si el archivo ha sido abierto comparando el puntero con NULL.

Cierre

Cuando un archivo no va a usarse más, su flujo debe ser cerrado para liberar memoria. Aunque teóricamente todos los archivos abiertos por un programa se cierran automáticamente al finalizar dicho programa, el programador, por precaución, debe ocuparse de hacerlo

dentro del código.

Para cerrar un archivo se usa la función fclose(), con esta sintaxis:

```
int fclose(FILE*
puntero_file);
```

Por ejemplo:

```
FILE *archivo;
archivo = fopen("datos.txt",
"r");
... instrucciones de
manipulación del archivo
"datos.txt" ...
fclose(archivo);
```

Fíjate de fclose() devuelve un número entero. Este número se puede utilizar

para averiguar si ha ocurrido un error al cerrar el archivo, ya que tomará el valor EOF si ha sido así (EOF es otra constante definida en stdio.h)

Lectura y escritura

Las funciones de lectura tienen el mismo comportamiento tanto si el archivo es secuencial como directo: empiezan leyendo desde el primer registro del archivo (si éste está recién abierto), y a partir de ahí van desplazando el punto de lectura hacia el final del archivo.

Las funciones de escritura, sin embargo, tienen un comportamiento diferente:

- Si estamos manejando un archivo en modo secuencial, todas las funciones de escritura que hagamos van a escribir la información al final del archivo. Cada vez que se escribe un registro, el cursor o punto de escritura avanza automáticamente al final del archivo, donde se escribirá el siguiente

registro.

- Si el archivo es directo, la escritura se realizará, por defecto, al principio del archivo (eliminando los registros que existieran), o bien en cualquier otra posición indicada por el programador (ver función fseek() en el siguiente apartado). Cada vez que se escribe un registro, el cursor o punto de escritura no

avanza automáticamente, sino que es el programador el que debe situarlo (nuevamente con la función fseek()) en el sitio adecuado antes de la siguiente lectura o escritura.

Otra diferencia fundamental (y evidente) es que los archivos secuenciales sólo pueden abrirse para lectura o para escritura, de modo que no pueden combinarse ambas operaciones sin antes cerrar el archivo y volver a abrirlo. Los archivos directos, en cambio, pueden

abrirse para lectura/escritura simultánea, compartiendo ambas operaciones el cursor o indicador de posición del archivo.

Por lo demás, y teniendo siempre presentes estas diferencias, las funciones de lectura y escritura son las mismas y se comportan de modo similar con los archivos directos y con los secuenciales. Todo lo que sigue es aplicable, además, tanto a archivos binarios como de texto, aunque luego veremos que algunas funciones se usan

más con un tipo de archivos y otras con el otro tipo.

Leer y escribir caracteres: fgetc() y fputc()

Para escribir un carácter en un archivo de texto se pueden utilizar las funciones putc() o fputc(), que son idénticas:

```
int putc(int carácter, FILE* puntero_a_archivo);
```

Observa que putc() recibe un entero, no un carácter. Esto obedece a razones históricas, pero, en realidad, putc() sólo

se fija en los 8 bits de menos peso del entero, por lo que, a todos los efectos, es como si fuera un carácter.

La función putc() devuelve el código EOF si no ha podido escribir el carácter.

Por ejemplo, de este modo se escribiría el carácter "s" al final del archivo "datos.txt":

```
FILE* archivo;
archivo = fopen("datos.txt",
"a");
if (archivo != NULL) putc('s',
archivo);
```

Para leer un carácter de un archivo de texto se utilizan las funciones getc() o fgetc(), que también son iguales y tienen esta forma:

```
int getc(FILE*
puntero_a_archivo)
```

Observa que getc() devuelve un entero, no un carácter, por las mismas razones que putc(); si lo asignamos a una variable de tipo carácter el resultado será correcto.

Leer y escribir cadenas de caracteres: fgets() y fputs()

Para escribir en un archivo de texto una cadena de caracteres completa se utiliza la función fputs(), que es igual que putc() pero con cadenas:

```
int fputs(char* cadena, FILE* puntero_a_archivo);
```

Del mismo modo, existe una función fgets() para leer cadenas de caracteres de un archivo de texto. En este caso, hay que indicar cuántos caracteres se quieren leer. La función irá leyendo caracteres del archivo hasta que encuentre un fin de línea o hasta que haya leído longitud – 1 caracteres.

Comenzará leyendo desde el primer carácter del archivo y a partir de ahí irá avanzando secuencialmente:

```
char* fgets(char* cadena, int
longitud, FILE*
puntero_a_archivo);
```

Lectura y escritura con formato: fscanf() y fprintf()

También se puede escribir en un archivo de texto como estamos acostumbrados a hacerlo en la pantalla usando printf(), ya que existe un función similar, llamada fprintf(), que envía los datos al archivo

especificado. Su sintaxis es muy parecida, salvo que hay que indicar a qué flujo se desean enviar los datos (de hecho, si escribimos "stdout" en el nombre del flujo, fprintf() funcionará exactamente igual que printf()):

```
int fprintf(FILE*
puntero_a_archivo, char*
cadena_de_formato,
lista_de_parámetros);
```

Por ejemplo, este fragmento de código escribe los caracteres "15 más 5 son 20" en el archivo "datos.txt":

```
FILE* archivo;
int a, b;
```

```
a = 15;
b = 5;
archivo = fopen("datos.txt",
"a");
if (archivo != NULL)
  fprintf(archivo, "%i más %i
son %i", a, b, a+b);
```

También existe una hermana gemela de scanf(); se llama fscanf() y lee los datos de un archivo, en lugar de hacerlo del flujo stdin (es decir, del teclado). Su prototipo es:

```
int fscanf(FILE*
puntero_a_archivo, char*
cadena_de_formato,
lista_de_parámetros);
```

Lectura y escritura de bloques de datos: fread() y fwrite()

También se puede enviar un bloque de memoria completo a un archivo. Para eso utilizaremos la función fwrite(), cuyo prototipo es:

```
int fwrite(void* puntero_a_memoria, int tam_bytes, int num, FILE* puntero_a_archivo);
```

Esta función escribe en el archivo especificado un número (num) de elementos del tamaño indicado en bytes (tam_bytes). Los elementos se cogen de

la memoria principal, a partir de la dirección apuntada por puntero_a_memoria.

Por ejemplo, la siguiente instrucción escribe en el archivo apuntado por el flujo fich 16 números de tipo float. Los números se leen de la memoria a partir de la dirección apuntada por ptr. Observa el uso que se hace de sizeof() para saber cuánto ocupa cada elemento (en este caso, cada número de tipo float):

```
fwrite(ptr, sizeof(float), 16,
fich);
```

La función fwrite() devuelve el número de bytes escritos correctamente, por lo que el programador puede comprobar si se han escrito tantos datos como se pretendía o si ha surgido algún problema.

La función complementaria de fwrite() es fread(), que sirve para leer un bloque de datos de un archivo y colocarlo en la memoria, a partir de determinada dirección apuntada por un puntero. El prototipo es:

```
int fread(void*
puntero_a_memoria, int
```

```
tam_bytes, int num, FILE*
puntero_a_archivo);
```

En este caso, se leen num elementos de tamaño tam_bytes del archivo. Todos los bytes se colocan en la memoria principal, en las direcciones situadas a partir de puntero_a_memoria. La función fread() devuelve el número de bytes leídos correctamente.

Lógicamente, el puntero_a_memoria debe estar apuntando a una zona de memoria que previamente haya sido reservada con malloc() u otra función similar.

Estas dos funciones (fread() y fwrite()) suelen utilizarse con archivos binarios, mientras que el resto de funciones de lectura y escritura (fgets(), fgetc(), fscanf(), etc) es más frecuente usarlas con archivos de texto.

Fin de fichero: feof()

EOF es una constante definida en stdio.h. Se utiliza en el tratamiento de ficheros para localizar el final de los mismos. EOF es un carácter especial no imprimible, cuyo código ASCII es 26, que casi todos los editores de texto

insertan al final de los archivos de texto para marcar el último registro o carácter.

Por lo tanto, si estamos leyendo caracteres secuencialmente de un archivo de texto, podemos ir comparándolos con EOF para comprobar cuándo hemos alcanzado el final del archivo.

Esto no funciona con archivos binarios, porque el valor de EOF puede ser confundido con una parte del último registro del archivo.

Para evitar este problema existe la función feof(), que nos dice si hemos alcanzado el final de un archivo, tanto de texto como binario. Devuelve un 0 (falso) si aún no se ha llegado al final, y otro valor cuando se ha alcanzado. Es muy útil para saber si podemos seguir leyendo caracteres o ya los hemos leído todos. Su prototipo es:

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

Vuelta al principio del archivo: rewind()

Otra función del ANSI C muy útil es rewind(). Sirve para situar el indicador de posición al comienzo del archivo; es como si hubiéramos abierto el archivo en ese momento. Su prototipo es:

```
void rewind(FILE*
puntero_a_archivo);
```

Vaciado del buffer: fflush()

Como ya comentamos, la salida de datos hacia archivos suele hacerse a través de un buffer por motivos de rendimiento. Así, cuando vamos escribiendo datos en un archivo, éstos pueden no escribirse

inmediatamente en el dispositivo de almacenamiento, sino que pemanecen durante un tiempo en un espacio intermedio de memoria llamado buffer, donde se van acumulando. Sólo cuando el buffer está lleno se realiza físicamente la operación de escritura.

Podemos forzar un vaciado del buffer con la función fflush(), que tiene este prototipo:

```
int fflush(FILE*
puntero_a_archivo);
```

Al llamar a fflush(), todos los datos que

estuvieran pendientes de ser escritos en el dispositivo de memoria secundaria se escribirán, y el buffer quedará vacío.

Si el puntero_a_archivo es NULL, se realizará un vaciado de buffer de todos los archivos que hubiera abiertos en ese momento.

La función fflush() devuelve 0 si el vaciado se ha realizado con éxito, o EOF si se produce algún error.

Cuando se cierra un archivo con fclose(), se realiza automáticamente un vaciado del buffer de ese archivo para

no perder los datos que estuvieran aún pendientes de escritura.

Funciones específicas de acceso directo

Aunque, como dijimos al principio, el lenguaje C maneja con las mismas funciones los archivos secuenciales y los directos, dispone de algunas funciones exclusivas para archivos de organización relativa directa. Estas funciones, que, por lo tanto, no tienen

sentido con los archivos secuenciales, son fseek() y ftell().

Las funciones de acceso directo de C no permiten hacer referencia a direcciones de memoria secundaria absolutas, pero sí relativas al comienzo del archivo. Es decir: asignan la dirección 0 al primer byte del archivo, de modo que cada registro tenga una dirección relativa a ese primer byte. Cuando, por ejemplo, enviamos el indicador de posición a la dirección 500, no lo estamos enviando a la dirección 500 de la memoria

secundaria, sino a la dirección 500 desde el comienzo del archivo.

La función fseek() sirve para situarnos directamente en cualquier posición del fichero, de manera que el resto de lecturas se hagan a partir de esa posición. Su prototipo es:

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

El argumento origen debe ser una de estas tres constantes definidas en stdio.h:

- SEEK_SET: principio del fichero
- SEEK_CUR: posición actual
- SEEK_END: final del fichero

El argumento num_bytes especifica en qué posición desde el origen queremos situarnos. Por ejemplo, con esta llamada nos colocamos en el byte número 500 contando desde el principio del archivo:

```
fseek(archivo, 500, SEEK_SET);
```

Y con esta otra nos desplazamos 2 bytes

más allá de la posición actual:

```
fseek(archivo, 2, SEEK_CUR);
```

Esta función devuelve 0 si se ejecuta correctamente o cualquier otro valor si ocurre algún error.

La otra función de acceso directo es ftell(), que devuelve el indicador de posición del archivo, es decir, cuántos bytes hay desde el principio del archivo hasta el lugar donde estamos situados en ese momento. Su prototipo es:

```
long int ftell(FILE*
puntero_a_archivo);
```

Devuelve -1 si se produce un error.

Diferencias entre archivos binarios y de texto

Como se ha dicho anteriormente, en los archivos de texto todos los datos se almacenan en forma de texto ASCII. Esto hace que podamos abrirlos, consultarlos y modificarlos con cualquier editor de texto, mientras que con los binarios no es posible.

En los archivos de texto, y dependiendo

del compilador y del sistema operativo empleado, pueden producirse ciertas transformaciones automáticas de caracteres. En particular, es frecuente que el carácter invisible de fin de línea (representado habitualmente como LF) se convierta en dos caracteres al escribirlo en un archivo (fin de línea – LF – más retorno de carro – CR –). También pueden ocurrir conversiones a la inversa, es decir, durante la lectura del archivo. Esto no sucede con los archivos binarios.

Todas las funciones de E/S sirven para ambos tipos de archivo, pero algunas pueden dar problemas con según qué tipos. Por ejemplo, fseek() no funciona bien con archivos de texto debido a las conversiones automáticas que antes mencionábamos. Desde cierto punto de vista, puede considerarse que un archivo de texto no es más que un archivo secuencial en el que cada registro es un carácter, por lo que tiene sentido que las funciones de acceso directo tengan problemas para tratar este tipo de archivos.

Como normas generales (que nos podemos saltar si la situación lo requiere, ya que C es lo bastante flexible como para permitirlo) propondremos las siguientes:

- Cuando se trate de manipular datos simples usaremos archivos de texto. Esto implica convertir los números a texto ASCII (lo cual es muy fácil de hacer usando fprintf() y fscanf() junto con las cadenas de

formato). Sólo en el caso de que estas conversiones representen un inconveniente grave recurriremos a los archivos binarios.

- Cuando tratemos con estructuras de datos más complejas usaremos archivos binarios, a menos que nos interese abrir esos archivos con un editor de texto, en cuyo caso seguiremos usando archivos de texto.

- Si necesitamos acceso directo, usaremos archivos binarios.
- Las funciones fread() y fwrite() se usarán preferentemente con achivos binarios, y el resto de funciones de lectura y escritura se reservarán para archivos de texto.

Manipulación de directorios

Existe un grupo de funciones de C que sirven para manipulación de directorios (o carpetas, en terminología de Windows). Estas funciones no actúan sobre flujos, sino sobre archivos y directorios directamente, por lo que hay que pasarles el nombre del archivo o del directorio en una cadena de texto.

A este respecto hay que destacar que la barra invertida ("\") que separa los directorios en Windows no puede utilizarse directamente en una cadena, ya que en C la barra invertida se usa para

los caracteres especiales (por ejemplo, el retorno de carro se representa "\n"). Para usar la barra invertida en una constante de cadena debemos usar la secuencia de escape "\\". Por ejemplo, para borrar el archivo C:\PRUEBAS\DATOS.TXT debemos escribir:

```
remove("C:\\PRUEBAS\\DATOS.TXT'
```

Aclarado esto, enumeramos a continuación la funciones de directorio más útiles:

remove()	Borra un archivo del direc

realiza con éxito, u otro va
abierto no podrá borrarse

```
int remove(char* no
```

rename() Cambia el nombre de un ar
realizado u otro valor si oc

```
int remove(char* no
nombre_nuevo);
```

chdir() Cambia el directorio activ
directorio donde está el ar
activo. Todos los archivos
ese directorio, a menos qu

```
int chdir(char* nom
```

La función devuelve 0 si el

valor en caso contrario

mkdir()	Crea un directorio dentro d operación tiene éxito. `int mkdir(char* nom`
rmdir()	Borra un directorio. Para d debe de estar vacío. Devue correctamente. `int rmdir(char* nom`

Además, existen otras funciones para leer el contenido de un directorio (es decir, la lista de archivos que lo componen) y procesar dicho contenido.

Dichas funciones escapan a la extensión de este manual, pero el lector interesado puede buscar información sobre ellas: son opendir(), closedir(), readdir(), etc.

PROCESAMIENTO EN C DE ARCHIVOS SECUENCIALES

En este apartado vamos a estudiar la implementación en C de los algoritmos que habitualmente se utilizan para procesar los archivos secuenciales.

Escritura

Los registros, en un archivo secuencial, se añaden siempre al final. Es necesario abrir el archivo para escritura, ya sea en el modo "w" si queremos borrar lo que contuviera anteriormente, o en el modo "a" si deseamos conservar su información anterior.

Una vez hecho esto, usaremos sucesivas instrucciones de escritura para insertar los registros (si el archivo es binario) o los caracteres (si es de texto). Ten en cuenta que los datos se grabarán en el

archivo exactamente en el mismo orden en el que los escribas.

Las funciones de escritura que se deben usar dependen de la naturaleza del problema y de las preferencias del programador, pero recuerda que, en general, fwrite() suele reservarse para archivos binarios y el resto para archivos de texto.

En el siguiente fragmento de código tienes un ejemplo. Un archivo de texto llamado "ejemplo.txt" se abre para añadir datos al mismo (modo "at").

Luego se escriben en el archivo diez números enteros elegidos al azar. Cada vez que se ejecute el programa, se añadirán otros diez números al azar al final del archivo. Observa cómo se usa fprintf() para enviar el número entero N (seguido de un retorno de carro) al archivo de texto gracias a la cadena de formato. Esta cadena de formato es idéntica a la de la función printf() que tantas veces hemos utilizado.

```
     FILE *fich;
     int i, N;
     fich = fopen("ejemplo.txt",
  "at");
```

```
    if (fich == NULL)
       printf("Error al abrir
el archivo");
    else
    {
       for (i = 0; N < 10; i++)
       {
          N = random(1000)+1;
          fprintf(fich, "%i\n",
N);
       }
       fclose(fich);
    }
```

Lectura

Al abrir un archivo secuencial para lectura (en modo "r"), el indicador de posición se sitúa en el primer byte del

archivo. Cada vez que se lea un dato, el indicador de posición se desplaza automáticamente tantos bytes adelante como se hayan leído. Las lecturas se pueden continuar haciendo hasta que se alcance el final del archivo.

En el siguiente ejemplo, abriremos el archivo del ejemplo anterior y escribiremos en la pantalla todos los números que contenga. Observa como usamos la funcion fscanf() para leer un número e introducirlo directamente en una variable de tipo entero. Si usásemos

otra función de lectura (como, por ejemplo, fgets()), el número sería leído en forma de cadena de caracteres, y luego habría que convertirlo a entero.

Fíjate también en cómo se usa la función feof() para comprobar si se ha alcanzado el final del archivo.

```
    FILE *fich;
    int N;
    fich = fopen("ejemplo.txt",
 "rt");
    if (fich == NULL)
       printf("Error al abrir
 el archivo");
    else
    {
```

```
      while
(!feof(fich))              //
Mientras no se llegue al final
del archivo...
      {
         fscanf(fich, "%i\n",
&N);              // Leemos un
número entero del archivo
         printf("%i\n",
N);                          //
Escribimos el número en la
pantalla
      }
      fclose(fich);
   }
```

Búsqueda

En un archivo secuencial el único

método de búsqueda posible es el secuencial, es decir, que hay que leer todos los registros, partiendo del primero, hasta encontrar el que buscamos.

En el siguiente ejemplo volvemos a utilizar el archivo generado en los ejemplos anteriores para tratar de localizar un número introducido por el usuario. Ese número se guarda en la variable n_busq. Después se van leyendo los números contenidos en el archivo en la variable n_leido,

comparando cada número con el que estamos buscando.

Si el número se encuentra, el programa dice en qué línea del archivo está. Si no se encuentra, se da un mensaje de error. Observa que, cuando el número no se encuentra, es necesario recorrer todo el archivo antes de determinar que el número no está en el mismo.

Si el archivo estuviera ordenado podríamos mejorar el mecanismo de búsqueda, ya que no sería necesario recorrer todo el archivo para determinar

que un elemento no está: bastaría con encontrar un elemento mayor para poder detener la búsqueda en ese instante.

```
    FILE *fich;
    int n_busq, n_leido, linea;
    int encontrado;
    fich = fopen("ejemplo.txt",
  "rt");
    if (fich == NULL)
       printf("Error al abrir
  el archivo");
    else
    {
       printf("¿Qué número
  desea buscar?");
       scanf("%i", &n_busq);
       linea = 0;
       encontrado = 0;
       while (!feof(fich))
```

```
      {
         linea++;
         fscanf(fich, "%i\n",
&n_leido);
         if (n_leido ==
n_busq) {            // ¡Hemos
encontrado el número!
           encontrado = 1;
           printf("He
encontrado ese número en la
línea %i\n", linea);
           break;
         }
      }
      if (encontrado == 0)
         printf("Ese número no
está en el archivo");
      fclose(fich);
   }
```

Borrado

El borrado es una operación problemática. Existen dos formas de hacer el borrado en un archivo secuencial:

1) Crear un segundo archivo secuencial y copiar en él todos los registros del archivo original excepto el que se pretende borrar. Después, se borra el archivo original y se renombra el archivo nuevo con el nombre que tenía el original

Como puedes imaginarte, este método,

aunque funciona, es muy lento, sobre todo si el archivo es largo.

2) Marcar el registro que se pretende borrar como "no válido" y, aunque siga existiendo, ignorarlo a la hora de procesar el archivo. Este segundo método requiere utilizar registros de estructura compleja (no simples archivos de texto como los que estamos viendo), y se hablará de ellos en el siguiente apartado y en las actividades del tema.

En el siguiente fragmento de código se

utiliza el primer método de borrado para eliminar la quinta línea del archivo "ejemplo.txt" usado en los ejemplos anteriores. Se van leyendo números del archivo original y escribiendo en otro archivo llamado "temporal", excepto la quinta línea, que es la que pretendemos borrar. Cuando el proceso acaba, cerramos los dos archivos, borramos "ejemplo.txt" y renombramos el archivo "temporal" para que a partir de ese momento se llame "ejemplo.txt"

```
FILE *f_orig, *f_nuevo;
int N, linea;
```

```
   f_orig =
fopen("ejemplo.txt", "rt");
   f_nuevo = fopen("temporal",
"wt");
   if ((f_orig == NULL) ||
(f_nuevo == NULL))
      printf("Error al abrir
los archivos");
   else
   {
      linea = 0;
      while (!feof(f_orig))
      {
         linea++;
         fscanf(f_orig,
"%i\n", &N);
         if (linea !=
5)                     // La
5ª línea no se escribe
            fprintf(f_nuevo,
"%i\n", N);
```

```
        }
        fclose(f_orig);
        fclose(f_nuevo);
        remove("ejemplo.txt");
        rename("temporal",
  "ejemplo.txt");
    }
```

Modificación

En los archivos secuenciales sólo puede escribirse al final del archivo. Por lo tanto, para modificar un registro hay que actuar de forma similar al primer método de borrado: creando un segundo archivo en el que se copiarán todos los registros exactamente igual que en el

archivo original, excepto el que se pretende cambiar.

Procesamiento de archivos con registros complejos

Hasta ahora todos los ejemplos han tratado con archivos de texto muy simples, en los que sólo había un número entero en cada línea.

Estas técnicas pueden extenderse a los archivos cuyos registros sean más complejos: sólo hay que modificar la

función de lectura o escritura para adaptarla al formato de los datos del archivo.

Por ejemplo, supongamos un archivo en el que, en vez de sencillos números enteros, tengamos almacenada la lista de alumnos del instituto. Cada registro del archivo contendrá el nombre, el número de matrícula y la edad de un alumno/a. Para tratar cada registro definiremos una estructura:

```
struct s_alumno {
  int matricula;
  char nombre[30];
```

```
    int edad;
  };
```

Cada registro del archivo se corresponderá exactamente con una estructura. Así, para añadir un alumno al archivo podemos usar el siguiente algoritmo:

```
     FILE *fich;
     struct s_alumno a;
     fich = fopen("alumnos.dat",
  "wb");
     if ((fich == NULL))
        printf("Error al abrir
  los archivos");
     else
     {
        printf("Introduzca los
```

```
datos del alumno/a que desea
añadir\n");
        printf("Nombre: ");
scanf("%s", a.nombre);
        printf("N° de matrícula:
"); scanf("%i", &a.matricula);
        printf("Edad: ");
scanf("%i", &a.edad);
        fwrite(&a, sizeof(struct
s_alumno),1,fich);
        fclose(fich);
    }
```

Observa que el procedimiento es el mismo que en el caso de sencillos número enteros, salvo que, al tratase de una estructura compleja, es preferible usar archivos binarios y la función

fwrite() en lugar de archivos de texto y la función fprintf(). Pero podría usarse perfectamente fprintf() de este modo (entre otros):

```
fprintf(fich, "%i %s %i ",
a.matricula, a.nombre,
a.edad);
```

Lógicamente, para hacer la lectura de este archivo será necesario usar fread() si se escribió con fwrite(), o fscanf() si se escribió con fprintf().

Los procedimientos de lectura, búsqueda, borrado, etc también son fácilmente extensibles a este tipo de

archivos más complejos.

Ejemplo: archivos secuenciales de texto

El siguiente programa trata de ilustrar cómo se utilizan los archivos de texto con C. Se trata de un programa que se divide en dos funciones. Por un lado, escribir_archivo() sirve para escribir un texto en la pantalla y volcarlo a un archivo llamado "prueba.txt". Todo lo que se va tecleando va apareciendo en la pantalla y, al mismo tiempo, se va

enviando, carácter a carácter, al archivo de disco, hasta que se introduce el carácter "#". Por otro lado, leer_archivo() hace lo contrario: lee todo lo que haya grabado en "prueba.txt" y lo muestra por la pantalla.

Fíjate en cómo se usa feof() para saber cuándo se ha llegado al final del archivo. Además, observa que se han preferido las funciones fgetc() y fputc() en lugar de fscanf() y fprintf(), por ser más adecuadas a la naturaleza de este problema.

```
#include <stdio.h>
int main(void)
{
  int opción;
  puts("¿Qué desea hacer? 1 =
escribir, 2 = leer");
  puts("Teclee 1 ó 2: ");
  scanf("%i", opcion);
  if (opcion == 1)
escribir_archivo();
  if (opcion == 2)
leer_archivo();
  return 0;
}
void escribir_archivo()
{
  FILE* f;
  char car;
  f = fopen("prueba.txt",
"w");
  if (f == NULL)
```

```
    printf("Error al abrir el
archivo\n");
  else
  {
    do
    {
      car =
getchar();            // Lee un
carácter desde el teclado
      fputc(car,
f);           // Escribe el
carácter en el archivo
    }
    while (car != '#');
    fclose(f);
  }
}
void leer_archivo()
{
  FILE* f;
  char car;
```

```
  f = fopen("prueba.txt",
"r");
  if (f == NULL)
    printf("Error al abrir el
archivo\n");
  else
  {
    do
    {
      car =
fgetc(f);          // Lee un
carácter del archivo

 printf("%c",car);          //
Lo muestra en la pantalla
    }
    while
(!feof(f));          //
Repite hasta alcanzar el fin
de fichero
    fclose(f);
```

```
    }
  }
```

Ejemplo: archivos secuenciales binarios

El siguiente ejemplo utiliza archivos binarios para escribir o leer un array de 30 estructuras. En el programa principal se pregunta al usuario qué desea hacer y dependiendo de su respuesta se invoca a una de estos dos funciones:

1) leer_archivo(): Abre el archivo

"alumnos.dat" para lectura y recupera los datos que haya en él, mostrándolos en la pantalla. Observa que es un archivo binario y fíjate sobre todo en el uso de fread():

```
fread(&alumno[i],sizeof(struct
s_alumno),1,archivo);
```

El argumento &alumno[i] es la dirección de memoria donde está guardado el elemento i-ésimo del array. El segundo argumento es sizeof(struct s_alumno), es decir, el tamaño de cada elemento del array. El tercer agumento es 1, porque es el número de elementos que vamos a

escribir. El último argumento es el nombre del flujo.

Fíjate en que esa instrucción se repite NUM_ALUMNOS veces, ya que esa es la cantidad de elementos que tiene el array. Podríamos haber sustituido todo el bucle por una sola instrucción de escritura como esta:

```
fread(alumno,sizeof(struct
s_alumno),NUM_ALUMNOS,archivo);
```

Aquí sólo pasamos la dirección del primer elemento del array y luego le decimos que escriba NUM_ALUMNOS

elementos en lugar de sólo 1.

2) escribir_archivo(): Primero se le pide al usuario que introduzca los datos por teclado y luego se guardan todos esos datos en "alumnos.dat". Observa el uso de la función fwrite(), que es similar al que antes hacíamos de fread().

```
#include <stdio.h>
#define NUM_ALUMNOS 30
struct s_alumno {
  int matricula;
  char nombre[30];
  int edad;
};
void
leer_archivo();
```

Prototipos

```
void escribir_archivo();
int main()
{
  int opcion;
  puts("¿Qué desea hacer? 1 =
escribir, 2 = leer");
  puts("Teclee 1 ó 2: ");
  scanf("%i", &opcion);
  if (opcion == 1)
escribir_archivo();
  if (opcion == 2)
leer_archivo();
  return 0;
}
void leer_archivo()
{
  int i;
  FILE *archivo;
  struct s_alumno
alumno[NUM_ALUMNOS];
```

```
  // Lectura de datos desde el
archivo
  archivo =
fopen("alumnos.dat","rb");
  if (archivo == NULL)
printf("Error al abrir el
archivo");
  else
  {
  for (i=0; i<NUM_ALUMNOS;
i++)
    {

 fread(&alumno[i],sizeof(struct
s_alumno),1,archivo);
    }
    fclose(archivo);
    // Escritura de los datos
en la pantala
    for (i=0; i<NUM_ALUMNOS;
i++)
```

```
    {
       printf("N° matrícula:
%i\n", alumno[i].matricula);
       printf("Nombre: %s\n ",
alumno[i].nombre);
       printf("Edad: %i\n",
alumno[i].edad);
    }
  }
}
void escribir_archivo()
{
  int i;
  FILE *archivo;
  struct s_alumno
alumno[NUM_ALUMNOS];
  // Lectura de datos por
teclado
  for (i=0; i<NUM_ALUMNOS;
i++)
  {
```

```
    printf("Introduzca n° de
matricula :");

 scanf("%d",&alumno[i].matricul
    printf("Introduzca nombre
:");
    gets(alumno[i].nombre);
    printf("Introduzca edad
:");

 scanf("%d",&alumno[i].edad);
   }
   // Grabación del archivo
   archivo =
fopen("alumnos.dat","ab+");
   if (archivo == NULL)
printf("Error al abrir el
archivo");
   else
   {
     for (i=0; i<NUM_ALUMNOS;
```

```
i++)
    {

 fwrite(&alumno[i],sizeof(struc
s_alumno),1,archivo);
    }
  fclose(archivo);
  }
}
```

PROCESAMIENTO EN C

DE ARCHIVOS RELATIVOS

DIRECTOS

Recuerda que en los archivos directos el campo clave coincide con la dirección de memoria secundaria donde se

almacena el registro (repasa el comienzo de esta parte del libro si no sabes de qué estamos hablando)

A la hora de implementar estos archivos en C no usaremos realmente la direcciones absolutas de almacenamiento, sino la posición relativa de cada registro respecto del comienzo del archivo. El primer registro tendrá la posición 0, el segundo la posición 1, el tercero la 2, etc. Como la función fseek() nos permite colocarnos al comienzo de cualquier registro,

podremos usar los archivos directos como si fueran vectores sin necesidad de conocer la dirección física de almacenamiento.

Lectura y escritura

Con los archivos directos se suelen establecer ciertas normas para la creación y manipulación, aunque no son obligatorias:

- Abrir el archivo para lectura y escritura. Esto no es imprescindible: es posible

usar archivos de acceso directo sólo de lectura o de escritura.

- Abrirlo en modo binario, ya que las funciones de acceso directo como fseek() pueden funcionar mal con archivos de texto debido a la conversión automática de caracteres.
- Usar funciones como fread() y fwrite(), que son más apropiadas para los archivos

binarios.

- Usar la función fseek() para situar el puntero de lectura/escritura en el lugar correcto del archivo.

Por ejemplo, supongamos que los registros que queremos guardar y leer de un archivo tienen la siguiente estructura:

```
struct s_registro {
   char Nombre[34];
   int dato;
   int matriz[23];
};
struct s_registro reg;
```

Teniendo en cuenta que el primer registro del archivo es el cero, para hacer una lectura del registro número 6 del archivo usaremos estas dos funciones:

```
fseek(fichero, 5*sizeof(struct
s_registro), SEEK_SET);
fread(&reg, sizeof(struct
s_registro), 1, fichero);
```

Análogamente, para hacer una operación de escritura en esa misma posición usaremos:

```
fseek(fichero, 5*sizeof(struct
s_registro), SEEK_SET);
fwrite(&reg, sizeof(struct
```

```
s_registro), 1, fichero);
```

Recuerda que después de cada operación de lectura o escritura, el indicador de posición del fichero se actualiza automáticamente a la siguiente posición, así que es buena idea hacer siempre un fseek() antes de un fread() o un fwrite() cuando estemos tratando con archivos relativos.

Búsqueda

Si los registros no están ordenados hay que hacer una búsqueda secuencial.

Si el archivo tiene los registros ordenados por algún campo tenemos dos opciones:

- Realizar una búsqueda secuencial mejorada, de las que se detienen en cuanto encuentran un elemento mayor que el que estábamos buscando
- Realizar una búsqueda binaria como la que estudiamos en el tema anterior (en los ejercicios

sobre vectores). Se puede aplicar fácilmente a los archivos directos si identificamos cada registro del archivo con una posición del vector. Es un ejercicio muy interesante que plantearemos entre las actividades del tema.

Cálculo del tamaño de un archivo directo

En ocasiones es útil conocer cuántos registros contiene un archivo. Para calcular el número de registros se puede usar el siguiente procedimiento:

```
long int nRegistros;
long int nBytes;
fseek(fichero, 0, SEEK_END);
// Colocar el cursor al final
del fichero
nBytes = ftell(fichero);
// Tamaño en bytes
nRegistros =
ftell(fich)/sizeof(s_registro);
  // Tamaño en registros
```

Borrado

Borrar registros puede ser complicado, ya que no hay ninguna función de librería estándar que lo haga, de modo que es tarea del programador implementar alguna función para hacerlo.

Como en el caso de los archivos secuenciales, disponemos de dos métodos para hacerlo:

- Copiar todos los registros en otro archivo auxiliar, excepto el que se desea borrar.
- Marcar el registro que se va

a borrar como "no válido".

Como al estudiar los archivos secuenciales nos centramos en el primer método, ahora vamos a referirnos al segundo. Consiste en marcar los registros como borrados o no válidos. Para ello hay que añadir un campo extra en la estructura del registro:

```
struct s_registro {
   char borrado;      // Campo que indica si el registro está borrado (S/N)
   char nombre[34];
   int dato;
   int matriz[23];
};
```

Por ejemplo, si el campo borrado tiene el valor 'N' o ' ', podemos considerar que el registro es válido. Si tiene el valor 'S' o '*', el registro se considerará borrado, aunque realmente seguirá existiendo en el archivo. De este modo, para borrar un registro sólo hay que cambiar el valor de ese campo.

Si se quiere elaborar más este método, se puede mantener un fichero auxiliar con la lista de los registros borrados. Esto tiene un doble propósito:

- Que se pueda diseñar una

función para sustituir a fseek() de modo que se tengan en cuenta los registros marcados.

- Que al insertar nuevos registros, se puedan sobreescribir los anteriormente marcados como borrados, si existe alguno, y así aprovechar el espacio.

Lo normal es implementar una combinación de los dos métodos de

borrado: durante la ejecución normal del programa se borran registros con el método de marcarlos, y cuando se cierra la aplicación, o se detecta que el porcentaje de registros borrados es alto, se compacta el fichero usando el segundo método.

Modificación

La modificación consiste en una sobreescritura de un registro que ya existe. Al principio de este apartado dedicado a los archivos directos

explicábamos como se hacía, al hablar de la lectura y la escritura.

Procesamiento en C de archivos indexados

Lo más ingenioso de los archivos indexados es la zona de índices. Repasa el apartado si no lo recuerdas con exactitud. Los índices pueden guardarse en un archivo independiente o bien en un array de memoria. La utilización de arrays hace que los registros se encuentren más rápidamente, pero

perderíamos la tabla de índices al cerrar el programa.

Lo más habitual es utilizar una combinación de los dos métodos: manener los índices en un archivo independiente y cargarlo en un array de memoria al inicio del programa, para volver a grabarlos cuando hayamos terminado.

En los siguientes ejemplos supondremos que el área de índices se encuentra en un archivo independiente. El archivo de índices se llamará "índice" y, el de

datos, "datos". Qué original.

Búsqueda

Para leer un registro primero hay que localizarlo. El procedimiento es el siguiente:

- Buscamos secuencialmente en el área de índices la dirección de comienzo del segmento donde está el registro que queremos buscar. Usaremos la clave del registro buscado para

localizar el segmento.

- Hacemos un acceso directo al primer registro del segmento.
- Hacemos un recorrido secuencial dentro del segmento hasta localizar el registro.
- Si el registro no se encuentra, acudimos al área de excedentes y hacemos un nuevo recorrido secuencial en ella para intentar

localizarlo allí.

A la hora de implementar todo esto en C es conveniente escribir una función que, recibiendo como parámetro la clave del registro buscado, devuelva su contenido entero.

Supongamos que struct s_registro es la estructura usada para los registros del área primaria y struct s_indice para los registros del área de índices. Una posible implementación sería la siguiente (obsérvala con detenimiento y luego la comentamos):

```
struct s_registro {
   ...                        //
Estructura del registro del
área primaria
};
struct s_indice
{                        //
Estructura del registro del
área de índices
   int
segmento;
N° de segmento
   int
dir_comienzo;             //
Dirección de comienzo
   int
clave_ult_reg;             //
Clave del último registro
};
// Esta función busca un
registro en el área primaria
```

```
// Devuelve los datos en el
parámetro "registro", que está
pasado por variable
void buscar_registro(int
clave, struct s_registro*
registro)
{
   FILE *f_index, *f_datos;
   struct s_indice i;
   struct s_registro r;
   // Abrimos los archivos de
índices y primario
   f_index = fopen("indice",
"rb");
   f_datos = fopen("datos",
"rb");
   if ((f_index == NULL) ||
(f_datos == NULL))  ERROR
   // Buscamos en el área de
índices secuencialmente
   while (!feof(f_index))
```

```
   {
      fread(&i, sizeof(struct
s_indice), 1, f_index);
      if (i.clave_ult_reg >
clave) break;
   }
   fclose(f_index);

   // Ya hemos localizado el
segmento
   // Hacemos un acceso
directo al comienzo del
segmento (i.dir_comienzo)
   fseek(f_datos,
(i.dir_comienzo-
1)*sizeof(struct s_registro),
SEEK_SET);
   // Y ahora recorremos
secuencialmente hasta
encontrar el registro
   while (!feof(f_datos))
```

```
    {
        fread(&registro,
  sizeof(struct s_registro), 1,
  f_datos);
        if (registro->clave ==
  clave) break;
    }
    fclose(f_datos);
  }
```

La estructura para el archivo de índices se corresponde exactamente con la que, según vimos, solía tener la tabla de índices.

La función de búsqueda recibe la clave del registro que se quiere buscar y una estructura donde devolverá los datos del

registro cuando lo encuentre.

Se hace un recorrido secuencial por el archivo de índices hasta encontrar una entrada cuya clave de último registro sea mayor que la clave que buscamos. A partir de esa entrada del índice, podemos saber dónde comienza el segmento del registro buscado. Situamos el indicador de posición en ese lugar con un fseek(), es decir, con un acceso directo, y a partir de ahí comenzamos una búsqueda secuencial que, si todo va bien, debe ser bastante corta.

Esta función es mejorable, porque no controla la posibilidad de que el registro no exista, o de que resida en el área de excedentes.

Otras operaciones sobre archivos indexados

Como vemos, manejar archivos indexados es bastante más complejo que hacerlo con archivos secuenciales, pero si se entiende bien el funcionamiento de las búsquedas no será problema

implementar el resto de operaciones.

Si hubiera que insertar datos nuevos en el archivo, puede hacerse en la zonas vacías del segmento correspondiente (si hay espacio), o puede añadirse un nuevo segmento, siempre que las claves continúen ordenadas. Si hubiera que añadir un registro intermedio, habría que hacerlo en el área de excedentes, que para eso está.

Para eliminar un registro basta con borrarlo del área primaria, quedando el hueco vacío con la posibilidad de ser

ocupado por otro registro en el futuro, siempre que no se desordenen las claves. Los métodos de borrado son los mismos que en el caso de los archivos secuenciales y directos.

Si se hacen muchas inserciones y borrados, es conveniente reordenar el archivo periódicamente, con el fin de compactar los segmentos y recolocar los registros que hayan ido a parar al área de excedentes.

El principal problema de la organización indexada es que el índice

puede llegar a ser muy grande y consumir bastante memoria secundaria. Además, como el índice se recorre de manera secuencial, si es muy largo puede tardarse bastante en llegar a las últimas posiciones. Una solución a este problema es hacer un índice secundario, que sirve para indexar la tabla de índices primaria, que a su vez indexa el área primaria del archivo.

Otra posibilidad para reducir el tamaño del índice es aumentar el tamaño del segmento, es decir, el número de

registros que hay en cada segmento, pero entonces aumentan las búsquedas secuenciales que hay que hacer en el área primaria. Por tanto, hay que buscar un compromiso entre el tamaño del segmento y el tamaño del índice para hacer que el número de búsquedas secuenciales sea el mínimo posible.

QUINTA PARTE: ESTRUCTURAS DE DATOS DINÁMICAS

Las estructuras de datos dinámicas son las que pueden crecer y decrecer en tiempo de ejecución. Un array es una estructura estática porque, al declararlo, hay que indicar cuántos elementos va a tener, y ese número de elementos no cambiará mientras el array exista. Las

estructuras dinámicas son más poderosas porque no tienen esa limitación.

Un gran poder conlleva una gran responsabilidad, le dijo Tío Ben a Spiderman (en realidad, la frase es de Roosevelt, pero eso no viene al caso). Ciertamente, menos limitaciones implican mayor complejidad. Las estructuras dinámicas permiten un manejo más eficaz de la memoria, pero también son mucho más complicadas de operar. La flexibilidad que

proporcionan, sin embargo, es tan grande que raramente podremos prescindir de ellas si se trata de manejar grandes volúmenes de información.

Para construir estructuras dinámicas, C nos proporciona una libertad total, hasta el punto que algunos dirían que C no nos proporciona nada en absoluto. Es como si el lenguaje te ofreciera un terreno en primera línea de playa y las herramientas necesarias para construirte una casa a tu medida. Tendrás que trabajar duro, pero podrás construir lo

que quieras. Otros lenguajes nos proporcionan muchas soluciones prefabricadas con las que montar nuestra casa, pero no es lo mismo que levantarla desde cero. Puede que cueste más esfuerzo, pero el control que tendremos sobre todos y cada uno de los detalles de la construcción es incomparable al de otros lenguajes.

Y, para lograr eso, el lenguaje C cuenta con llave mágica que abre todas las puertas: el puntero.

PUNTEROS

Comprender y usar correctamente los punteros es con seguridad lo más complicado del lenguaje C, pero también se trata de un mecanismo muy poderoso. Tan poderoso que un simple puntero descontrolado (hay quien acertadamente los llama "punteros locos") puede provocar que el programa se cuelgue irremediablemente o incluso que falle todo el sistema.

Todos los programadores con cierta experiencia en C reconocerán que, a

veces, programar con punteros es como quedarse atrapado en un ascensor con un montón de serpientes pitón enloquecidas. Pero, cuando se les coge el tranquillo y se les ata en corto, permiten hacer auténticas virguerías.

Comprendiendo los punteros

Dentro de la memoria del ordenador, cada dato almacenado ocupa una o más celdas contiguas de memoria. El número de celdas de memoria requeridas para

almacenar un dato depende de su tipo. Por ejemplo, un dato de tipo entero puede ocupar 16 bits (es decir, 2 bytes), mientras que un dato de tipo carácter ocupa 8 bits (es decir, 1 byte).

Un puntero no es más que una variable cuyo contenido no es un dato, sino la dirección de memoria donde está almacenado un dato.

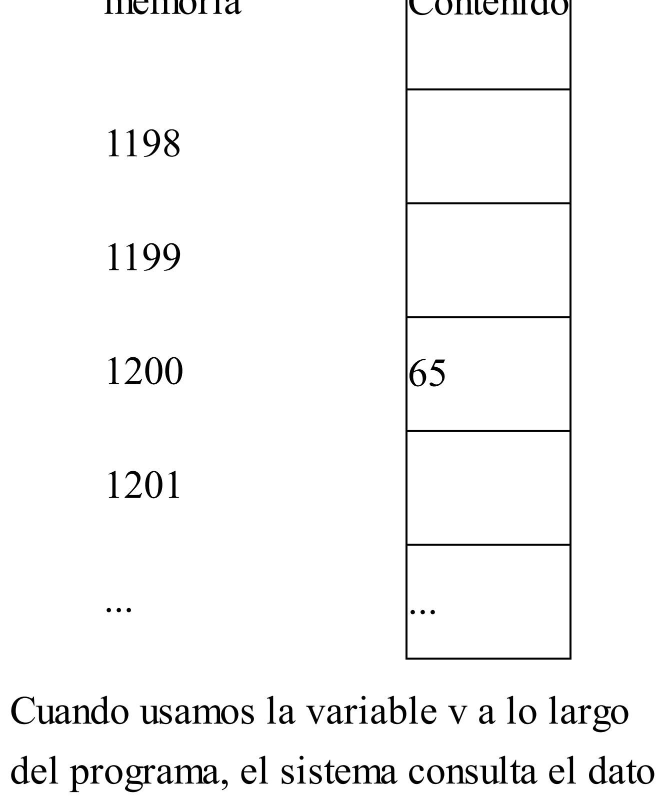
Veámoslo a través de un ejemplo. Imaginemos que v es una variable de tipo carácter y que, por tanto, necesita 1 byte para ser almacenada. La

declaración e inicialización de la variable será como la siguiente:

```
char v;
v = 'A';
```

Al ejecutar este código, el sistema operativo asigna automáticamente una celda de memoria para el dato. Supongamos que la celda asignada tiene la dirección 1200. Al hacer la asignación v = 'A', el sistema almacena en la celda 1200 el valor 65, que es el código ASCII de la letra 'A':

Dirección de



Cuando usamos la variable v a lo largo del programa, el sistema consulta el dato

contenido en la celda de memoria asignada a la variable. Esa celda será siempre la misma a lo largo de la ejecución: la 1200. Por ejemplo, al encontrar esta instrucción:

```
printf("%c", v);
```

.. el compilador acude a la celda 1200 de la memoria, consulta el dato almacenado en ella en ese momento y sustituye la variable v por ese dato.

El programador no tiene modo de saber en qué posición de memoria se almacena cada dato, a menos que utilice punteros.

Los punteros sirven, entonces, para conocer la dirección de memoria donde se almacena el dato, y no el dato en sí.

La dirección ocupada por una variable v se determina escribiendo &v. Por lo tanto, & es un operador unario, llamado operador dirección, que proporciona la dirección de una variable.

La dirección de v se le puede asignar a otra variable mediante esta instrucción:

```
char* p;
p = &v;
```

Resultará que esta nueva variable es un

puntero a v, es decir, una variable cuyo contenido es la dirección de memoria ocupada por la variable v. Representa la dirección de v y no su valor. Por lo tanto, el contenido de p será 1200, mientras que el contenido de v será 65.

El dato almacenado en la celda apuntada por la variable puntero puede ser accedido mediante el operador asterisco aplicado al puntero. Así pues, la expresión *p devuelve el valor 65, que es el contenido de la celda apuntada por p. El operador * es un operador unario,

llamado operador indirección, que opera sólo sobre una variable puntero.

Los operadores monarios & y * son miembros del mismo grupo de precedencia que los otros operadores monarios: -,++,--,!,etc. Hay que recordar que este grupo de operadores tiene mayor precedencia que el de los operadores aritméticos y la asociatividad de los operadores monarios es de derecha a izquierda.

Resumiendo: podemos tener variables "normales" y utilizar el operador & para

conocer su dirección de memoria. O podemos tener variables puntero, que ya son en sí mismas direcciones de memoria, y utilizar el operador * para acceder al dato que contienen. Así pues:

- El operador dirección (&) sólo puede actuar sobre variables que no sean punteros. En el ejemplo anterior, la variable v vale 65 y la expresión &v vale 1200.
- El operador indirección (*)

sólo puede actuar sobre variables que sean punteros. En el ejemplo anterior, la expresión *p vale 65 y la variable p vale 1200.

Las variables puntero pueden apuntar a direcciones donde se almacene cualquier tipo de dato: enteros, flotantes, caracteres, cadenas, arrays, estructuras, etc. Esto es tremendamente útil y proporciona una enorme potencia al lenguaje C, pero también es una fuente inagotable de errores de programación

difíciles de detectar y corregir, como iremos viendo en los siguientes temas

Declaración e inicialización de punteros

Las variables de tipo puntero, como cualquier otra variable, deben ser declaradas antes de ser usadas. Cuando una variable puntero es definida, el nombre de la variable debe ir precedido por un *. El tipo de dato que aparece en la declaración se refiere al tipo de dato

que se almacena en la dirección representada por el puntero, en vez del puntero mismo. Así, una declaración de puntero general es:

```
tipo_dato *puntero;
```

donde puntero es la variable puntero y tipo_dato el tipo de dato apuntado por el puntero.

Por ejemplo:

```
int *numero;
char *letra;
```

La variable numero no contiene un número entero, sino la dirección de

memoria donde se almacenará un número entero. La variable letra tampoco contiene un carácter, sino una dirección de memoria donde se almacenará un carácter.

Cuando un puntero ha sido declarado pero no inicializado, apunta a una dirección de memoria indeterminada. Si tratamos de usarlo en esas condiciones obtendremos resultados impredecibles (y casi siempre desagradables). Antes de usar cualquier puntero hay que asegurarse de que está apuntando a una

dirección válida, es decir, a la dirección de alguna variable del tipo adecuado. Por ejemplo, así:

```
int *numero;
int a;
numero = &a;
```

El puntero numero ahora sí está en condiciones de ser usado, porque está apuntado a la dirección de la variable a, que es de tipo int, como el puntero.

Otra posibilidad es hacer que un puntero apunte a NULL. El identificador NULL es una constante definida en el lenguaje que indica que un puntero no está

apuntando a ninguna dirección válida y que, por lo tanto, no se debe utilizar.

Asignación de punteros

Se puede asignar una variable puntero a otra siempre que ambas apunten al mismo tipo de dato. Al realizar la asignación, ambos punteros quedarán apuntando a la misma dirección de memoria.

Observa este ejemplo y trata de determinar qué resultado se obtiene en

la pantalla (antes de leer la solución que aparece más abajo):

```
int a, b, c;
int *p1, *p2;
a = 5;
p1 = &a;                /* p1
apunta a la dirección de
memoria de la variable a */
p2 = p1;                /* a p2 se
le asigna la misma dirección
que tenga p1 */
b = *p1;
c = *p1 + 5;                /* Suma
5 a lo que contenga la
dirección apuntada por p1 */
printf("%i %i %i %p %p", a, b,
c, p1, p2);
```

En la pantalla se imprimirá “5 5 10”,

que es el contenido de las variables a, b y c al terminar la ejecución de este bloque de instrucciones, y la dirección a la que apuntan p1 y p2, que debe ser la misma. Observa que con printf y la cadena de formato "%p" se puede mostrar la dirección de memoria de cualquier variable.

Aritmética de punteros

Con las variables de tipo puntero sólo se pueden hacer dos operaciones

aritméticas: sumar o restar a un puntero un número entero, y restar dos punteros. Pero el resultado de esas operaciones no es tan trivial como puede parecer. Por ejemplo, si sumamos un 1 a un puntero cuyo valor sea 1200, el resultado puede ser 1201… ¡pero también puede ser 1202 ó 1204! Esto se debe a que el resultado depende del tipo de dato al que apunte el puntero.

Sumar o restar un valor entero a un puntero

Al sumar un número entero a un puntero

se incrementa la dirección de memoria a la que apunta. Ese incremento depende del tamaño del tipo de dato apuntado.

Si tenemos un puntero p y lo incrementamos en una cantidad entera N, la dirección a la que apuntará será:

```
dirección_original + N *
tamaño_del_tipo_de_dato
```

Por ejemplo, imaginemos un puntero p a carácter que se incrementa en 5 unidades, así:

```
char* p;
p = p + 5;
```

Supongamos que p apunta a la dirección de memoria 800. Como cada carácter ocupa 1 byte, al incrementarlo en 5 unidades, p apuntará a la dirección 805.

Veamos ahora que pasa si, por ejemplo, el puntero p apunta a un número entero:

```
int* p;
p = p + 5;
```

Si la dirección inicial de p es también la 800, al incrementarlo en 5 unidades pasará a apuntar a la dirección 810 (suponiendo que cada entero ocupe 2 bytes).

Todo esto también explica qué ocurre cuando se resta un número entero de un puntero, sólo que entonces las direcciones se decrementan en lugar de incrementarse.

A los punteros también se les puede aplicar las operaciones de incremento (++) y decremento (--) de C, debiendo tener el programador en cuenta que, según lo dicho hasta ahora, la dirección apuntada por el puntero se incrementará o decrementará más o menos dependiendo del tipo de dato apuntado.

Por ejemplo, si los datos de tipo int ocupan 2 bytes y el puntero p apunta a la dirección 800, tras la ejecución de este fragmento de código, el puntero p quedará apuntado a la dirección 802:

```
int *p;
p++;
```

Resta de dos punteros

La resta de punteros se usa para saber cuantos elementos del tipo de dato apuntado caben entre dos direcciones diferentes.

Por ejemplo, si tenemos un vector de números reales llamado serie podemos hacer algo así:

```
float serie[15];
int d;
float *p1, *p2;
p1 = &tabla[4];
p2 = &tabla[12];
d = p1 - p2;
```

El puntero p1 apunta al quinto elemento del vector, y el puntero p2, al decimotercero. La restar los dos punteros obtendremos el valor 8, que es el número de elementos de tipo float que pueden almacenarse entre las

direcciones p1 y p2.

Punteros y arrays

Punteros y arrays de una dimensión

Los punteros y los arrays tienen una relación muy estrecha, ya que el nombre de un array es en realidad un puntero al primer elemento de ese array. Si x es un array undimensional, la dirección del primer elemento puede ser expresada como &x[0] o simplemente como x. La dirección del elemento i-ésimo se puede expresar como &x[i] o como (x+i).

(En este caso, la expresión (x+i) no es una operación aritmética convencional, sino una operación con punteros, de cuyas peculiaridades ya hemos hablado en un epígrafe anterior)

Si &x[i] y (x+i) representan la dirección del i-ésimo elemento de x, podemos decir que x[i] y *(x+i) representan el contenido de esa dirección, es decir, el valor del i-ésimo elemento de x. Observa que la forma x[i] es la que hemos estado utilizando hasta ahora para acceder a los elementos de un vector.

Los arrays, por lo tanto, pueden utilizarse con índices o con punteros. Al programador suele resultarle mucho más cómodo utilizar la forma x[i] para acceder al elemento i-ésimo de un array. Sin embargo, hay que tener en cuenta que la forma *(x+i) es mucho más eficiente que x[i], por lo que suele preferirse cuando la velocidad del ejecución es un factor determinante.

Punteros y arrays multidimensionales

Un array multidimensional es en realidad una colección de varios arrays

unidimensionales (vectores). Por tanto, se puede definir un array multidimensional como un puntero a un grupo contiguo de arrays unidimensionales.

El caso más simple de array de varias dimensiones es el bidimiensional. La declaración de un array bidimensional la hemos escrito hasta ahora como:

```
tipo_dato variable
[expresión1][expresión2]
```

Pero también puede expresarse así:

```
tipo_dato (*variable)
[expresión2]
```

Los paréntesis que rodean al puntero deben estar presentes para que la sintaxis sea correcta.

Por ejemplo, supongamos que x es un array bidimensional de enteros con 10 filas y 20 columnas. Podemos declarar x como:

```
int x[10][20];
```

Y también como:

```
int (*x)[20];
```

En la segunda declaración, x se define como un puntero a un grupo de array unidimensionales de 20 elementos

enteros. Así x apunta al primero de los arrays de 20 elementos, que es en realidad la primera fila (fila 0) del array bidimensional original. Del mismo modo (x+1) apunta al segundo array de 20 elementos, y así sucesivamente.

Por ejemplo, el elemento de la columna 2 y la fila 5 puede ser accedido así:

```
x[2][5];
```

Pero también así:

```
*(*(x+2)+5);
```

Esta instrucción parece muy complicada pero es fácil de desentrañar:

- (x+2) es un puntero a la columna 2

- *(x+2) es el objeto de ese puntero y refiere a toda la columna. Como la columna 2 es un array unidimensional, *(x+2) es realmente un puntero al primer elemento de la columna 2.

- (*(x+2)+5) es un puntero al elemento 5 de la columna 2.

- El objeto de este puntero *(*(x+2)+5) refiere al elemento

5 de la columna 2.

Arrays de punteros

Un array multidimensional puede ser expresado como un array de punteros en vez de como un puntero a un grupo contiguo de arrays. En términos generales un array bidimensional puede ser definido como un array unidimensional de punteros escribiendo

```
tipo_dato
*variable[expresión1]
```

...en lugar de la definición habitual, que

sería:

```
tipo_dato variable[expresión1]
[expresión2]
```

Observa que el nombre del array precedido por un asterisco no está cerrado entre paréntesis. Ese asterisco que precede al nombre de la variable establece que el array contendrá punteros.

Por ejemplo, supongamos que x es un array bidimensional de 10 columnas y 25 filas. Se puede definir x como un array unidimensional de punteros

escribiendo:

```
int *x[25];
```

Aquí x[0] apunta al primer elemento de la primera columna, x[1] al primer elemento de la segunda columna, y así sucesivamente. Observa que el número de elementos dentro de cada fila no está especificado explícitamente. Un elemento individual del array, tal com x[2][5] puede ser accedido escribiendo:

```
*(x[2]+5)
```

En esta expresión, x[2] es un puntero al primer elemento en la columna 2, de

modo que (x[2]+5) apunta al elemento 5 de la columna 2. El objeto de este puntero, *(x[2]+5), refiere, por tanto, a x[2][5].

Los arrays de punteros ofrecen un método conveniente para almacenar cadenas. En esta situación cada elemento del array es un puntero que indica dónde empieza cada cadena.

Paso de punteros como parámetros

A menudo los punteros son pasados a las

funciones como argumentos. Esto permite que datos de la porción de programa desde el que se llama a la función sean accedidos por la función, alterados dentro de ella y devueltos de forma alterada. Este uso de los punteros se conoce como paso de parámetros por variable o referencia y lo hemos estado utilizando hasta ahora sin saber muy bien lo que hacíamos.

Cuando los punteros son usados como argumento de una función, es necesario tener cuidado con la declaración y uso

de los parámetros dentro de la función. Los argumentos formales que sean punteros deben ir precedidos por un asterisco. Observa detenidamente el siguiente ejemplo:

```
#include <stdio.h>
void funcion1(int, int);
void funcion2(int*, int*);
int main(void)
{
  int u, v;
  u = 1;
  v = 3;
  funcion1(u,v);
  printf("Después de la
llamada a funcion1:  u=%d
v=%d\n", u, v);
  funcion2(&u,&v);
```

```
  printf("Después de la
llamada a funcion2:  u=%d
v=%d\n", u, v);
}
void funcion1(int u, int
v)
{
   u=0;
   v=0;
}
void funcion2(int *pu, int
*pv)
{
   *pu=0;
   *pv=0;
}
```

La función de nombre funcion1 utiliza paso de parámetros por valor. Cuando es invocada, los valores de las variables

u y v del programa principal son copiados en los parámetros u y v de la función. Al modificar estos parámetros dentro de la función, el valor de u y v en el programa principal no cambia.

En cambio, funcion2 utiliza paso de parámetros por variable (también llamado paso de parámetros por referencia o por dirección). Lo que se pasa a la función no es el valor de las variables u y v, sino su dirección de memoria, es decir, un puntero a las celdas de memoria donde u y v están

almacenadas. Dentro de la función, se utiliza el operador asterisco para acceder al contenido de pu y pv y, en consecuencia, se altera el contenido de las posiciones de memoria apuntadas por pu y pv. El resultado es que las variables u y v del programa principal quedan modificadas.

Por lo tanto, la salida del programa debe ser:

```
Después de la llamada a
funcion1:  u=1 v=3
Después de la llamada a
funcion2:  u=0 v=0
```

Recuerda que la función scanf() requiere que sus argumentos vayan precedidos por &, mientras que printf() no lo necesitaba. Hasta ahora no podíamos comprender por qué, pero ahora podemos dar una razón: scanf() necesita que sus argumentos vayan precedidos del símbolo & porque necesita las direcciones de los datos que van a ser leídos, para poder colocar en esas posiciones de memoria los datos introducidos por teclado. En cambio, printf() no necesita las direcciones, sino únicamente los valores de los datos para

poder mostrarlos en la pantalla.

Al estudiar los arrays y las estructuras ya vimos en detalle cómo se deben pasar como parámetros a las funciones. Recuerda que los arrays siempre se pasan por variable y no es necesario usar el símbolo & en la llamada, ya que el propio nombre del array se refiere, en realidad, a la dirección del primer elemento.

Devolución de punteros

Una función también puede devolver un puntero. Para hacer esto, la declaración de la función debe indicar que devolverá un puntero. Esto se realiza precediendo el nombre de la función con un asterisco. Por ejemplo:

```
double *funcion(argumentos);
```

Cuando esta función sea invocada, devolverá un puntero a un dato de tipo double, y por lo tanto debe ser asignada a una variable de ese tipo. Por ejemplo, así:

```
double *pf;
pf = funcion(argumentos);
```

```
printf("%lf", *pf);
```

Punteros a funciones

Las funciones de C, como todo el código de todos los programas que se ejecutan en el ordenador, también ocupan unas posiciones concretas de la memoria principal. Por lo tanto, es posible disponer de un puntero a una función, es decir, de una variable que contenga la dirección de memoria en la que comienza el código de una función.

Aunque no vamos a usar esta avanzada posibilidad de C, se menciona aquí como información para el lector que desee ampliar sus conocimientos. La declaración de un puntero a función se realiza así:

```
tipo_de_dato (*nombre_puntero)
(lista_de_parámetros);
```

No debe confundirse con la declaración de una función que devuelve un puntero:

```
tipo_de_dato* nombre_función
(lista_de_parámetros);
```

Posteriormente, la dirección de la función puede asignarse al puntero para

luego ser invocada a través del puntero, en lugar de usar una llamada convencional:

```
nombre_puntero =
nombre_función;                /*
Asignación al puntero de la
dirección de la función */
(*nombre_puntero)
(lista_de_parámetros);
Invocación de la función */
```

Punteros a punteros

Un último aspecto (a la vez confuso y potente) de los punteros es la posibilidad de definir punteros que, a su

vez, apunten a otros punteros. Esto no es un trabalenguas, sino que, técnicamente, se denomina *indirección múltiple*. La definición de un puntero a puntero se hace así:

```
tipo_de_dato **nombre_puntero;
```

Por ejemplo, el resultado del siguiente fragmento de código en C debe ser que se imprima el número 15 en la pantalla:

```
int n;
int* p1;
int** p2;
p1 = &n;                /* p1
contiene la dirección de n */
p2 = &p1;               /* p2
```

```
contiene la dirección de p1 */
**p2 = 15;
printf("%i", n);
```

GESTIÓN DINÁMICA DE LA MEMORIA

Según hemos visto hasta ahora, la memoria reservada para cada variable se define en el momento de escribir el código del programa. Por ejemplo, si declaramos una variable de tipo int, ésta tendrá asignados 2 bytes de memoria (aunque esa cantidad puede variar dependiendo del compilador y del

sistema operativo). Entonces, si declaramos un array de 100 números enteros, el array tendrá reservados 200 bytes de memoria.

¿Pero qué ocurre si no sabemos de antemano cuántos elementos puede llegar a tener el array?

Por ejemplo, imaginemos un problema consistente en leer por teclado (u otro dispositivo de entrada) una cantidad indefinida de números para almacenarlos en un array y luego hacer ciertas operaciones con ellos. ¿De qué

tamaño podemos definir el array? ¿De 100 elementos? ¿Y si el usuario introduce 101 elementos?

Podemos pensar, entonces, que será suficiente con definir el array muy grande: de 1000 elementos, o de 5000, o de 10000… pero siempre existe la posibilidad de que el programa no funcione correctamente por desbordamiento del espacio reservado a las variables. Y, por otra parte, si definimos un array de enormes dimensiones y luego la mayoría de sus

posiciones no se utilizan, estaremos desperdiciando los recursos de la máquina.

Para evitar esto existe la asignación dinámica de memoria, que consiste en reservar memoria para las variables en tiempo de ejecución, es decir, mientras el programa está funcionando. Así, es posible "estirar" o "encoger" sobre la marcha el espacio reservado para el array, dependiendo de las necesidades de cada momento, y no limitarse a los 100, 1000 ó 10000 elementos que

definió el programador al escribir el código.

Veremos enseguida que, para manejar la memoria dinámicamente, es imprescindible el uso de punteros. De hecho, este es el mejor fruto que vamos a obtener de ellos.

Reserva dinámica de memoria. Arrays dinámicos.

Utilizaremos el ejemplo de los arrays por ser la estructura de datos más simple

y fácil de entender, pero lo dicho aquí es extensible a otras estructuras de datos diferentes. De hecho, dedicaremos el resto del tema a estudiar otras estructuras de datos dinámicas más complejas.

Ya que un nombre de array es en realidad un puntero a su primer elemento, es posible definir un array como una variable puntero en vez de como un array convencional. Así, estas dos definiciones sirven para un vector de números enteros:

```
int vector1[100];
int* vector2;
```

El vector1 se define del modo convencional de un array. Esto produce la reserva de un bloque fijo de memoria al empezar la ejecución del programa lo suficientemente grande como para almacenar 100 números enteros.

El vector2 se define como puntero a entero. En este caso, no se reserva ninguna cantidad de memoria para almacenar los números enteros.

Si intentamos acceder a los elementos

de los vectores obtendremos resultados diferentes:

```
vector1[5] = 83;
vector2[5] = 27;                    /*
Esto es un error */
```

La primera asignación funcionará correctamente, ya que el quinto elemento del vector1 tiene un espacio de memoria asignado. La segunda asignación producirá un efecto impredecible, ya que vector2 no tiene ningún espacio de memoria asignado y, por lo tanto, el dato 27 se escribirá en una posición de memoria correspondiente a otro dato u

otro programa. La consecuencia puede llegar a ser bastante desagradable.

Se necesita, pues, reservar un fragmento de memoria antes de que los elementos del array sean procesados. Tales tipos de reserva se realizan mediante la función malloc() o alguna de sus variedades. Observa bien su uso en este ejemplo:

```
int *x;
x = (int *) malloc (100 *
sizeof(int));
```

La función malloc() reserva un especio de memoria consistente en 100 veces el

tamaño de un número entero. Fíjate bien en el uso del sizeof(int): se trata de un operador unario que devuelve el tamaño de un tipo de dato cualquiera, tanto simple como complejo.

Suponiendo que sizeof(int) fuera 2 (es decir, que cada número de tipo int ocupase 2 bytes), lo que se le está pidiendo a malloc() es que reserve 100 * 2 bytes, es decir, 200 bytes de memoria.

Además, es necesario usar el molde (int *), ya que malloc() devuelve un puntero

sin tipo (es decir, un puntero a void), así que hay que convertirlo a puntero a entero antes de asignarlo a la variable x, que efectivamente es un puntero a entero.

De esta manera, la variable vector2 pasa a ser lo que podemos denominar un array dinámico, en el sentido de que se comporta como un array y puede usarse como tal, pero su tamaño ha sido definido durante la ejecución del programa (más adelante, en el mismo programa, podemos redefinir el tamaño

del array para acortarlo o alargarlo)

Si la función malloc() falla devolverá un puntero a NULL. Utilizar un puntero a NULL es la forma más segura de estrellar el programa, así que siempre debemos comprobar que el puntero devuelto es correcto. Una vez hecho esto, podemos utilizar x con toda tranquilidad como si fuera un array de 100 números enteros. Por ejemplo:

```
int *x, i;
x = (int *) malloc (100 *
sizeof(int));
if (x == NULL)
  printf("Error en la
```

```
asignación de memoria");
else
{
  printf("Se ha reservado con
éxito espacio para 100
números");
  for (i=0; i<100; i++)
  {
        printf("Introduzca un
número:");
        scanf("%i", &x[i]);
  }
}
```

Liberación de memoria

El programador debe tener dos precauciones básicas a la hora de

manejar la memoria dinámicamente:

- Asignar memoria a un puntero antes de usarlo con malloc() u otra función similar
- Liberar la memoria asignada, cuando ya no va a usarse más, con free() u otra función similar.

Si no se libera la memoria asignada a un puntero, teóricamente no ocurre nada grave, salvo que podemos terminar por agotar la memoria disponible si

reservamos continuamente y nunca liberamos. Es, en cualquier caso, una costumbre muy saludable.

Para liberar la memoria reservada previamente con malloc() u otra función de su misma familia, se utiliza la función free(). Observa su uso en este ejemplo:

```
int *x, i;
x = (int *) malloc (100 *
sizeof(int));
... instrucciones de
manipulación de x ...
free(x);
```

Toda la memoria reservada con malloc() quedará liberada después de hacer

free() y se podrá utilizar para guardar otros datos o programas. El puntero x quedará apuntado a NULL y no debe ser utilizado hasta que se le asigne alguna otra dirección válida.

Funciones básicas para la gestión dinámica de la memoria

Además de malloc() y free() existen otras funciones similares pero con pequeñas diferencias. A continuación

resumimos las más usuales y mostramos un ejemplo de su uso.

Pero antes haremos una advertencia: todas las funciones de reserva de memoria devuelven un puntero a NULL si no tienen éxito. Por lo tanto, deben ir seguidas de un condicional que compruebe si el puntero apunta a NULL antes de utilizarlo: no nos cansaremos de repetir que utilizar un puntero a NULL es una manera segura de estrellar el programa.

calloc()	Reserva un bloque de men

tam bytes y devuelve un pu
sintaxis es:

```
void* calloc(num, t
```

El siguiente ejemplo reser

```
int* p;
p = (int*) calloc(3
```

free() Libera el bloque de memor
previamente había sido res

```
free(puntero);
```

malloc() Reserva un bloque de mem
void al comienzo del mism

```
void* malloc(tam);
```

Por ejemplo, para reservar

caracteres:

```
char* texto;
texto = (char*) mal
```

realloc() Cambia el tamaño de un bl
Dicho bloque ha debido se
otra función similar. El nue
un puntero void al comienz

```
void* realloc(punte
```

En el siguiente ejemplo, se
pero luego se modifica el t
500 caracteres:

```
char* texto;
texto = (char*) mal
```

```
/* Aquí irían las i
puntero texto
   con un tamaño de
texto = (char*) rea
/* A partir de aquí
usarse para
   manejar hasta 50
```

Introducción a las estructuras dinámicas

Las estructuras estáticas tienen una importante limitación: no pueden cambiar de tamaño durante la ejecución. Por ejemplo, los arrays están compuestos por un determinado número

de elementos y ese número se decide durante la codificación del programa, no pudiendo cambiarse en tiempo de ejecución.

En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Esas son las estructuras dinámicas.

C no dispone de estructuras dinámicas predefinidas, por lo que es tarea del programador construirlas basándose en estructuras estáticas y gestión dinámica

de memoria. Además, habrá que programar una colección de funciones que manipulen esas estructuras.

Ya que el programador se toma la molestia de implementar las estructuras y sus funciones, lo más habitual es que se asegure de que todo sea reutilizable, de manera que pueda usarlo en otros programas. A lo largo del tema seguiremos este principio.

Como veremos, para desarrollar las estructuras dinámicas es imprescindible usar punteros y asignación dinámica de

memoria, así que deberías tener bastante claros los dos primeros epígrafes de este tema antes de continuar.

Nodos

El fundamento de las estructuras de datos dinámicas es una estructura estática a la que llamaremos nodo o elemento. Éste incluye los datos con los que trabajará nuestro programa y uno o más punteros al mismo tipo nodo.

Por ejemplo, si la estructura dinámica va a guardar números enteros, el nodo

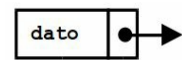
puede tener esta forma:

```
struct s_nodo {
   int dato;
   struct nodo *otro_nodo;
};
```

El campo otro_nodo apuntará a otro objeto del tipo nodo. De este modo, cada nodo puede usarse como un ladrillo para construir estructuras más complejas, y cada uno mantendrá una relación con otro u otros nodos (esto dependerá del tipo de estructura dinámica, como veremos).

A lo largo del tema usaremos una

representación gráfica para mostrar las estructuras de datos dinámicas. El nodo anterior se representará así:



En el rectángulo de la izquierda se representa el dato contenido en el nodo (en nuestro ejemplo, un número entero). En el rectángulo de la derecha se representa el puntero, que apuntará a otro nodo.

Tipos de estructuras dinámicas

Dependiendo del número de punteros

que haya en cada nodo y de las relaciones entre ellos, podemos distinguir varios tipos de estructuras dinámicas. A lo largo del tema veremos sólo las estructuras básicas, pero aquí las vamos a enumerar todas:

- Listas abiertas: cada elemento sólo dispone de un puntero, que apuntará al siguiente elemento de la lista.
- Pilas: son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out: el

último en entrar es el primero en salir). Los elementos se "amontonan" o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila.

- Colas: otro tipo de listas, conocidas como listas FIFO (First In, First Out: El primero en entrar es el primero en salir). Los

elementos se almacenan en una lista, pero sólo pueden añadirse por un extremo y leerse por el otro.

- Listas circulares: o listas cerradas, son parecidas a las listas abiertas, pero el último elemento apunta al primero. De hecho, en las listas circulares no puede hablarse de "primero" ni de "último".
- Listas doblemente enlazadas: cada elemento dispone de

dos punteros, uno apunta al siguiente elemento y el otro al elemento anterior. Al contrario que las listas abiertas, estas listas pueden recorrerse en los dos sentidos.

- Árboles: cada elemento dispone de dos o más punteros, pero las referencias nunca son a elementos anteriores, de modo que la estructura se ramifica y crece

de modo jerárquico.

- Árboles binarios: son árboles donde cada nodo sólo puede apuntar a dos nodos.
- Árboles binarios de búsqueda (ABB): son árboles binarios ordenados, por lo que la búsqueda de información en ellos es menos costosa. Desde cada nodo todos los nodos de una rama serán mayores, según la

norma que se haya seguido para ordenar el árbol, y los de la otra rama serán menores.

- Árboles AVL: son también árboles de búsqueda, pero su estructura está más optimizada para reducir los tiempos de búsqueda.
- Árboles B: son otro tipo de árboles de búsqueda más complejos y optimizados que los anteriores.

- Tablas HASH: son estructuras auxiliares para ordenar listas de gran tamaño.
- Grafos: son árboles no jerarquizados, es decir, en los que cada nodo puede apuntar a nodos de nivel inferior o de nivel superior. De hecho, no se puede hablar de nivel “superior” e “inferior”. Son las estructuras dinámicas más

complejas.

Para terminar con esta introducción, señalar que pueden existir estructuras dinámicas en las que haya nodos de distinto tipo, aunque nosotros no las vamos a estudiar.

Listas abiertas (o listas enlazadas simples)

Qué es una lista abierta y cómo

funciona

La forma más simple, que no la menos potente, de estructura dinámica es la lista abierta. Se trata de una especie de vector dinámico en el que el número de elementos puede crecer y decrecer a voluntad del programador en tiempo de ejecución.

En esta estructura, los nodos se organizan de modo que cada uno apunta al siguiente, y el último no apunta a nada, es decir, el puntero al nodo siguiente vale NULL.

En las listas abiertas existe un nodo especial: el primero. Para manejar la lista es necesario mantener un puntero a ese primer nodo, que llamaremos cabeza de la lista. Mediante ese único puntero-cabeza podemos acceder a toda la lista. Cuando el puntero-cabeza vale NULL, diremos que la lista está vacía.

Podemos representar gráficamente una lista de esta manera:

Esta lista contiene 4 datos. Observa

como cada dato está enlazado con el nodo que contiene el siguiente dato y, además, el puntero primero apunta a la cabeza de la lista, es decir, al primer elemento. Es muy importante no perder nunca el valor de ese puntero, ya que en tal caso sería imposible acceder al primer nodo y, desde él, a todos los demás.

Tipos de datos para implementar listas abiertas

De aquí en adelante supondremos que estamos manejando una lista abierta de

números enteros, pero el lector debe tener en cuenta que el tipo de dato con el que se construye la lista puede ser cualquiera, sin más que modificar la estructura del nodo.

Para construir una lista abierta de números enteros debemos definir los siguientes tipos de datos y variables:

```
struct s_nodo {
   int dato;
   struct nodo *siguiente;
};
typedef struct s_nodo t_nodo;
t_nodo *primero;
```

Observa que la estructura s_nodo

contiene un dato (en nuestro caso, de tipo entero) seguido de un puntero a otro nodo. Después, se define una variable llamada primero, que será el puntero al primer nodo de la lista.

Operaciones con listas abiertas

Con las definiciones anteriores aún no tendremos disponible una lista abierta. Es importante darse cuenta de que el tipo de dato “lista abierta dinámica” no existe en C estándar. Para crearlo, debemos declarar los tipos de datos anteriores y, además, construir funciones

en C que nos sirvan para utilizar esos datos. Entonces sí que tendremos disponible un nuevo tipo de dato para utilizar en nuestros programas y, además, podremos reutilizarlo en todos los programas en los que nos haga falta.

Las operaciones básicas que debemos programar para obtener el nuevo tipo “lista abierta” son:

1. Añadir o insertar elementos.
2. Buscar elementos.

3. Borrar elementos.

Estas son las operaciones fundamentales, pero podemos añadirles otras muchas operaciones secundarias que pueden llegar a sernos muy útiles, como:

1. Contar el número de elementos que hay en la lista.
2. Comprobar si la lista está vacía.
3. Borrar todos los

elementos de la lista.

4. Etc.

En general, procuraremos programar cada operación con una función independiente. Esto facilitará la reusabilidad del código que escribamos. Hay que tener siempre presente que las funciones con listas abiertas, una vez programas, deben poder se reutilizadas en otros programas con el mínimo número de cambios posible.

Con esa idea en mente, vamos a ver a continuación cómo podemos

implementar las operaciones básicas para manejar listas abiertas.

Insertar elementos

Insertar un elemento en una lista vacía

Si una lista está vacía significa que no contiene ningún nodo y, por lo tanto, el puntero primero estará apuntando a NULL. Esto lo representaremos así:

primero ●⟶ NULL

El proceso para insertar un nodo en la

lista vacía consiste en:

- Crear ese nodo reservando memoria para el mismo (con malloc() o una función similar). Tras la creación, dispondremos de un puntero apuntando al nodo (llamaremos nodo a la variable puntero a nodo).
- Hacer que nodo->siguiente apunte a NULL
- Hacer que primero apunte a nodo.

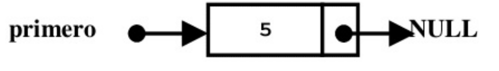
El resultado de la ejecución de estos tres pasos debe ser:

Veamos como se implementa esto en C. Dispondremos de una variable primero, que apunta al primer elemento de la lista, y de una variable nodo, que será el elemento que pretendemos insertar en la lista. El valor del dato de este nodo será 5.

```
t_nodo *primero, *nodo;
primero = NULL;
                        //
Cuando la lista está vacía, su
```

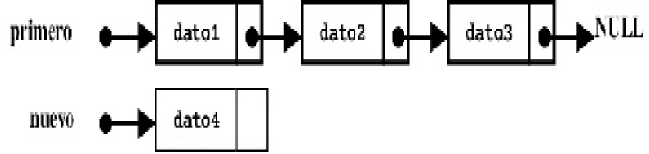
```
primer elemento es NULL
nodo = (t_nodo*)
malloc(sizeof(t_nodo));
            // Nuevo elemento
nodo->dato = 5;
                       // El
dato guardado en el nuevo
elemento es 5
nodo->siguiente = NULL;
                       // El
elemento siguiente a este será
NULL
primero = nodo;
                       // El
primer elemento deja de ser
NULL y pasa a ser "nodo"
```

La lista resultante de la ejecución de este fragmento de código es esta:



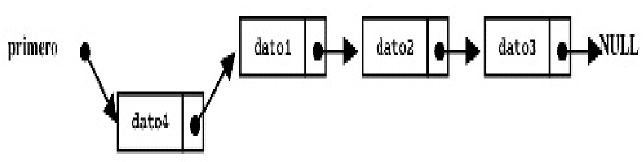
Insertar un elemento en la primera posición de una lista

En este caso dispondremos de una lista no vacía y de un nuevo nodo que queremos insertar al principio de la lista:



Para hacer la inserción, basta con seguir esta secuencia de acciones:

- El puntero primero debe apuntar al nuevo nodo
- El nuevo nodo debe apuntar al que hasta ahora era el primero

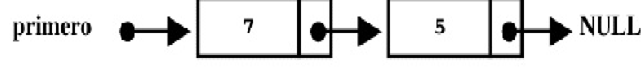


Si lo escribimos en C:

```
t_nodo *nuevo;
nuevo = (t_nodo*)
malloc(sizeof(t_nodo));
           // Nuevo elemento
nuevo->dato = 7;
                      // El
```

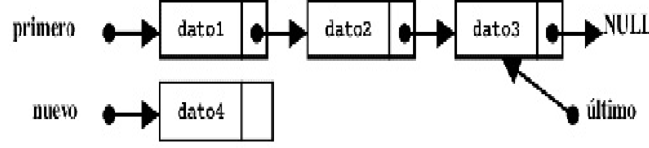
```
nuevo dato guardado en el
nuevo elemento será 7
nuevo->siguiente = primero;
                         // El
elemento siguiente a este será
el que antes era primero
primero = nuevo;
                         // El
nuevo elemento pasa a ser el
primero
```

Si aplicamos este código sobre la lista anterior tendremos este resultado:



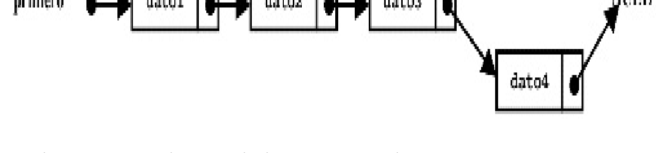
Insertar un elemento en la última posición de una lista

Razonando del mismo modo podemos insertar un nuevo nodo al final de una lista no vacía, sólo que en este caso necesitamos un puntero que nos señale al último elemento de la lista. La forma de conseguir este puntero es muy sencilla: basta con recorrer uno a uno todos los elementos de la lista hasta llegar al último. Podemos reconocer el último porque es el único cuyo elemento siguiente valdrá NULL.



Cuando tengamos todos estos elementos, el proceso de inserción se resume en:

- Hacer que el último elemento deje de apuntar a NULL y pase a apuntar al nuevo nodo.
- Hacer que el nuevo nodo apunte a NULL



Observa detenidamente la implementación en C, prestando atención a cómo se obtiene el puntero al

último elemento de la lista. Recuerda que el último se identifica porque su puntero a su siguiente elemento vale NULL:

```
t_nodo *ultimo, *nuevo;
// Primera parte: buscar el
último nodo de la lista (para
eso, la recorremos desde el
principio)
ultimo = primero;
while (ultimo->siguiente !=
NULL)
     ultimo = ultimo-
>siguiente;
// Segunda parte: crear el
nodo nuevo e insertarlo en la
lista
nuevo = (t_nodo*)
```

```
malloc(sizeof(t_nodo));
            // Creamos nodo
nuevo
nuevo->dato = 18;
            // Le asignamos un
valor al dato
ultimo->siguiente = nuevo;
                       // Lo
enlazamos al (hasta ahora)
último de la lista
nuevo->siguiente = NULL;
                       //
Hacemos que el siguiente del
nodo nuevo sea NULL
```

Si aplicamos este código a la lista de ejemplo del apartado anterior obtendremos esta otra lista:

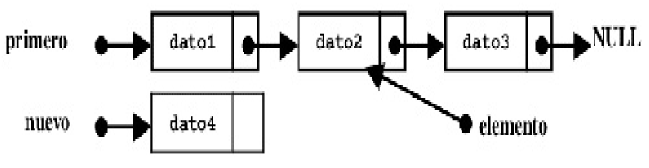
primero → [7 | •] → [5 | •] → [18 | •] → NULL

Insertar un elemento a continuación de un nodo cualquiera de una lista

Para insertar un nodo nuevo en cualquier posición de una lista, es decir, entre otros dos nodos cualesquiera, el procedimiento es similar al anterior, sólo que ahora, en lugar de un puntero al último elemento, necesitaremos disponer de un puntero al nodo exacto a partir del cual pretendemos hacer la inserción.

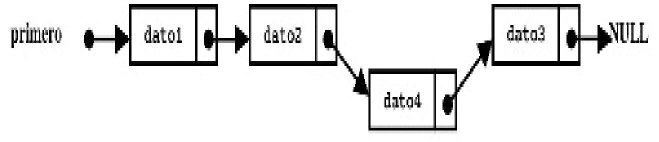
Supongamos que queremos insertar el nuevo nodo entre los elementos 2 y 3 de la lista; entonces necesitaremos un

puntero al elemento 2, así:



Con todos esos elementos, basta con reasignar los punteros para obtener la nueva lista:

- El nodo 2 dejará de apuntar al 3 y pasará a apuntar al nuevo nodo (4)
- El nuevo nodo pasará a apuntar al nodo 3



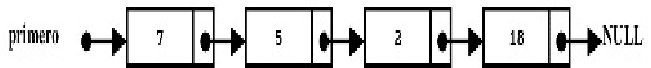
Como hemos hecho en los otros casos, vamos la implementación en C de este tipo de inserción. Supondremos que estamos trabajando con la misma lista que en los ejemplos de los anteriores epígrafes, y que se desea insertar un nuevo nodo entre los datos 5 y 18. Necesitamos obtener un puntero al nodo que contiene el dato 5, y para ello debemos ir mirando los datos contenidos en todos los nodos desde el

primero.

```
t_nodo *elemento, *nuevo;
// Primera parte: buscar el
nodo en el que queremos
insertar el nuevo (contendrá
el dato 5)
elemento = primero;
while ((elemento->dato != 5)
&& (elemento != NULL))
    elemento = elemento-
>siguiente;
// Segunda parte: crear el
nodo nuevo e insertarlo en la
lista
if (elemento != NULL) {
                      //
Comprobamos que hemos
encontrado el punto de
inserción
    nuevo = (t_nodo*)
```

```
malloc(sizeof(t_nodo));
            // Creamos nodo
nuevo
    nuevo->dato = 2;
            // Le asignamos un
valor al dato
    nuevo->siguiente =
elemento->siguiente;
            // Lo enlazamos al
siguiente de la lista
    elemento->siguiente =
nuevo;
            // Hacemos que el
anterior apunte al nodo nuevo
}
```

La lista resultante será esta:



Buscar elementos

Muy a menudo necesitaremos recorrer una lista, ya sea buscando un valor particular o un nodo concreto. De hecho, es algo que ya hemos necesitado hacer en algunos de los algoritmos de inserción que hemos presentado en el epígrafe anterior.

Las listas abiertas sólo pueden recorrerse en un sentido, ya que cada nodo apunta al siguiente, de modo que no se puede obtener un puntero al nodo anterior desde un nodo cualquiera.

Para recorrer una lista procederemos siempre del mismo modo:

- Usaremos un puntero auxiliar (a modo del contador que se usa para recorrer un array)
- El valor inicial del puntero auxiliar será igual al primer elemento de la lista
- Iniciamos un bucle que, al menos, debe tener una condición: que el puntero auxiliar no sea NULL. Cuando el puntero auxiliar

tome el valor NULL significará que hemos llegado al final de la lista.

- Dentro del bucle asignaremos al puntero auxiliar el valor del nodo siguiente al actual.

Por ejemplo, este fragmento de código muestra los valores de los nodos de la lista de los ejemplos anteriores:

```
t_nodo *aux;
aux = primero;
while (aux != NULL)
{
   printf("%d\n", aux->dato);
   aux = aux->siguiente;
```

```
}
```

La condición de salida del bucle puede complicarse si queremos añadir algún criterio de búsqueda, pero SIEMPRE debemos conservar la comparación (aux != NULL) para terminar el bucle en caso de llegar al final de la lista. Si no, el programa fallará.

Por ejemplo, vamos a buscar, en la lista de los ejemplos, el dato 50. Si existe, se mostrará en la pantalla, y, si no, se dará un mensaje de error:

```
t_nodo *aux;
aux = primero;
```

```
while ((aux != NULL) && (aux-
>dato != 50))
{
   aux = aux->siguiente;
}
if (aux->dato == 50)
   printf("El dato 50 está en
la lista");
else
   printf("El dato 50 NO se
encuentra en la lista");
```

Borrar elementos

Eliminar el primer nodo de una lista abierta

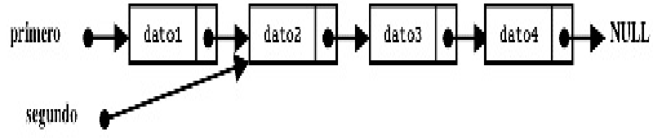
Para eliminar el primer nodo de una

lista usaremos un puntero auxiliar que apunte al segundo, de esta manera:

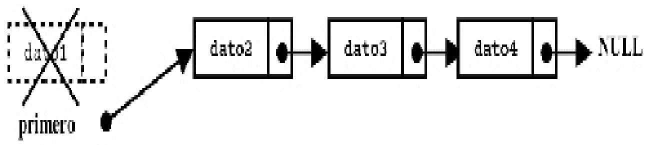
- Hacer que el puntero auxiliar apunte a primero->siguiente (es decir, al segundo nodo)
- Eliminar el elemento primero, liberando la memoria con free() o una función similar
- Reasignar el puntero primero para que pase a apuntar al que antes era el segundo nodo, y que ahora se habrá

convertido en el primero.

Partimos, por tanto, de esta situación:



Y, después del proceso de borrado, debemos obtener este resultado:



Observa que si no guardásemos el puntero al segundo nodo antes de actualizar la lista, después nos resultaría

imposible acceder al nuevo primer elemento, y toda la lista sería inaccesible.

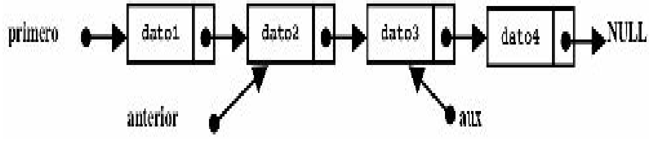
La implementación en C de todo esto podría ser algo así:

```
t_nodo *segundo;
if (primero != NULL) {
// Comprobamos que la lista no
esté vacía
   segundo = primero-
>siguiente;  // Guardamos la
referencia al segundo elemento
   free(primero);
// Eliminamos el primero (para
liberar la memoria)
   primero = segundo;
// El que era segundo se
```

```
convierte en primero
}
```

Eliminar un nodo cualquiera de una lista abierta

En todos los demás casos, eliminar un nodo se hace siempre del mismo modo. Únicamente necesitamos disponer de un puntero al nodo anterior al que queremos eliminar, y un nodo auxiliar que apunte al siguiente, es decir, al que vamos a eliminar:

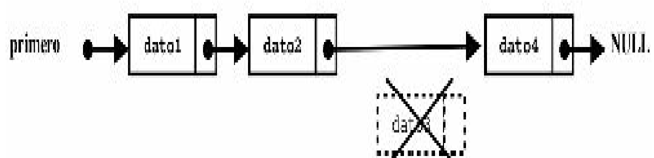


El proceso es muy parecido al del caso anterior:

- Hacemos que el puntero auxiliar apunte al nodo que queremos borrar (anterior -> siguiente)
- Asignamos como nodo siguiente del nodo anterior, el siguiente al que queremos eliminar. Es decir, anterior ->

siguiente = aux -> siguiente.

- Eliminamos el nodo apuntado por aux, liberando la memoria.



Como hacemos siempre, presentamos una implementación de este algoritmo en C. Para ello, supondremos que queremos eliminar el nodo siguiente a aquél que contiene en dato 7:

```
t_nodo *anterior, *aux;
```

```
// Primera parte: buscar el
nodo anterior al que vamos a
borrar (contendrá el dato 7)
anterior = primero;
while ((anterior->dato != 7)
&& (anterior != NULL))
   anterior = anterior-
>siguiente;
// Segunda parte: borrar el
nodo siguiente y reasignar los
punteros
if (anterior != NULL) {
// Comprobamos que hemos
encontrado el punto de
eliminación
   aux = anterior->siguiente;
// aux es el nodo que queremos
eliminar
   anterior->siguiente = aux-
>siguiente; // Reasignamos los
enlaces
```

```
    free(aux);
  // Eliminamos el nodo
  }
```

Eliminar todos los nodos de una lista

Para eliminar una lista completa hay que recorrer todos los nodos e ir liberando la memoria de cada uno, hasta que alcancemos el último nodo (que reconoceremos porque estará apuntando a NULL).

Otra manera de hacerlo es eliminar el primer elemento de la lista repetidamente, según el algoritmo que

hemos visto antes, hasta que el primer elemento sea NULL. Eso significará que la lista se ha quedado vacía.

Ejemplo de implementación en C de las operaciones básicas

A continuación presentamos una posible implementación C de las operaciones básicas sobre listas, para que puedas estudiar en conjunto muchos de los casos particulares que hemos estado

viendo por separado hasta ahora.

Supondremos que ya se ha definido la estructura del nodo (como vimos al principio del epígrafe) y que la lista sirve para almacenar números enteros (para que almacene otro tipo de información basta con cambiar la estructura del nodo)

Implementaremos una función diferente para cada operación sobre la lista, de manera que estas mismas funciones puedan utilizarse en otros programas:

- Función insertar(): servirá

para añadir un dato a la lista. Recibirá como parámetros el puntero al primer elemento de la lista y el dato (número entero en nuestro ejemplo) que se quiere insertar. Insertaremos el dato siempre en la primera posición de la lista, pero esta función se puede modificar para insertar el dato al final o en cualquier otra ubicación (por ejemplo, se puede mantener a lista ordenada insertando el dato

en la posición que le corresponda)

- Función borrar(): servirá para borrar un dato de la lista. Recibirá como parámetros el puntero al primer elemento y el dato que se quiere borrar (un número entero). Buscará en la lista ese dato y, si lo encuentra, lo eliminará. Devolverá 1 si el borrado se ha hecho con éxito, o –1 si ha fallado.

- Función buscar(): servirá para buscar un dato en la lista. Recibirá como parámetros el puntero al primer elemento y la posición del dato que se quiere buscar. Luego recorrerá la lista hasta la posición indicada y devolverá el número almacenado en ella, o bien –1 si esa posición no existe. Fíjate en que esto difiere de la operación “buscar” que

> hemos definido en la parte teórica de este apartado. Allí buscábamos un nodo a través del dato que contenía, y aquí vamos a buscarlo a partir de la posición que ocupa en la lista.

Ten en cuenta que esta es sólo una posible implementación de una lista. Dependiendo de la naturaleza del problema, puede ser necesario modificar las funciones para que actúen de otro modo. Esto no ocurrirá con las

pilas y las colas que veremos en los siguientes apartados, pues son estructuras dinámicas mucho más definidas y, por lo tanto, admiten pocas interpretaciones.

Por último, y antes de pasar a ver el código, observa que, al utilizar funciones para cada operación, tenemos que pasar por variable o referencia el puntero al primer elemento de la lista. Y como el puntero al primer elemento ya es un puntero, hay que pasar como parámetro un puntero a puntero. Eso

plantea algunos problemas sintácticos que debes observar con detalle (en el caso de la función buscar() eso no ocurre porque el parámetro se puede pasar por valor)

```
void insertar(t_nodo
**primero, int v)
{
   t_nodo* nuevo;
   nuevo =
(t_nodo*)malloc(sizeof(t_nodo))
Creamos nodo nuevo
   nuevo->dato =
v;                      // Le
asignamos el dato
   nuevo->siguiente =
*primero;                  //
El primero pasará a ser el
```

```
segundo
   *primero =
nuevo;                     //
Y el nuevo pasará a ser el
primero
}
int borrar(t_nodo **primero,
int v)
{
   t_nodo *anterior, *aux;
   int borrado =
-1;           // Marca de "no
borrado"

   // Primera parte: buscar el
nodo anterior al que vamos a
borrar
   // El que vamos a borrar se
distingue porque contiene el
dato "v"
   anterior = *primero;
```

```
   while (anterior != NULL)
   {
      aux = anterior-
>siguiente;
      if ((aux != NULL) &&
(aux->dato == v))

 break;                          //
aux es el nodo que queremos
eliminar
      anterior = anterior-
>siguiente;
   }
   // Segunda parte: borrar el
nodo siguiente y reasignar los
punteros
   // Comprobamos que hemos
encontrado el nodo que
deseamos eliminar (aux)
   if ((anterior != NULL) &&
(aux != NULL))
```

```
   {
      anterior->siguiente =
aux->siguiente;
           // Reasignamos los
enlaces
      free(aux);
           // Eliminamos el
nodo
      borrado =
1;
           // Marca de
"borrado realizado"
   }
   return borrado;
}
int buscar (t_nodo* primero,
int pos)
{
   int cont, valor;
   t_nodo* nodo;
   nodo =
```

```
primero;              // Nos
situamos en el primer elemento
   cont = 1;             //
Ponemos el contador a su valor
inicial
   while ((cont<pos) && (nodo
!= NULL))  // Repetir hasta
encontrar nodo o terminar
lista
   {
      nodo = nodo-
>siguiente;             //
Pasamos al nodo siguiente
      cont++;            //
Actualizamos el contador de
nodos
   }
   if (cont ==
pos)           // Hemos
encontrado el elemento buscado
       valor = nodo->dato;
```

```
    else                // No hemos
  encontrado el elemento
        valor = -1;
    return valor;
  }
```

Desde el programa principal se usarán estas funciones en el orden adecuado para resolver el problema que se nos haya planteado. Por ejemplo, estas son algunas llamadas válidas:

```
  insertar(primero, 5);
  insertar(primero, n);
  insertar(primero, 2);
  borrar(primero, 5);
  n = buscar(primero, 1);
  ...etc...
```

A las funciones básicas que aquí se presentan, se pueden añadir las operaciones secundarias que ya hemos mencionado (borrar toda la lista, contar el número de elementos, etc). Estas operaciones las realizaremos como ejercicio.

PILAS

Qué son las pilas y cómo funcionan

Una pila es un tipo especial y

simplificado de lista abierta con la que sólo está permitido realizar dos operaciones: insertar y eliminar nodos en uno de los extremos de la lista. Estas operaciones se conocen como push y pop (respectivamente, "empujar" y "tirar"). Además, al leer un dato (pop), el nodo se elimina automáticamente de la lista.

Estas características implican un comportamiento de lista LIFO (Last In First Out), que significa que el último elemento en entrar es el primero en salir.

De hecho, el nombre de "pila" deriva del símil con una pila de platos. En ella sólo es posible hacer dos cosas: añadir platos encima de la pila o coger el plato de la cima de la pila. Cualquier otra manipulación resultaría catastrófica.

La estructura en pila puede parecer un poco extraña, pero en realidad se ajusta como un guante a determinados problemas. Esto, unido a la extrema simplicidad de uso (ya que sólo permite dos operaciones) hace que sea una estructura muy recomendable en ciertas

ocasiones.

La representación interna de una pila es exactamente igual que la de una lista abierta: sólo cambiarán las operaciones que se pueden realizar con ella y que vamos a estudiar enseguida.

Tipos de datos para implementar pilas

En los siguientes apartados supondremos que estamos trabajando con una pila cuyos elementos son números enteros. Para cambiar el tipo

de dato de cada elemento, bastaría con modificar la definición de la estructura s_nodo:

```
struct s_nodo
{
   int dato;
   struct s_nodo *siguiente;
};
typedef struct s_nodo t_nodo;
t_nodo *primero;
```

Fíjate en que es exactamente igual que una lista abierta y, como sucedía con ellas, es fundamental no perder nunca el puntero al primer elemento de la pila, porque es a través de él como podemos

acceder a los demás.

Operaciones con pilas

Las pilas tienen un conjunto de operaciones muy limitado:

- Operación *push*: Añadir un elemento de la cima de la pila.
- Operación *pop*: Leer y eliminar un elemento de la cima de la pila.

Teniendo en cuenta que las inserciones y

borrados en una pila se hacen siempre en un extremo (cima), consideraremos que el primer elemento de la lista es en realidad la cima de la pila.

Push (insertar)

La operación push consiste en insertar un dato en la cima de la pila. Las operaciones con pilas son muy simples: no hay casos especiales, salvo que la pila esté vacía.

Push en una pila vacía

Debemos disponer de un nodo del tipo t_nodo y del puntero primero, que debe apuntar a NULL si la pila está vacía, la operación push será exactamente igual que la inserción en una lista abierta vacía:

- Hacer que nodo->siguiente apunte a NULL
- Hacer que primero apunte a nodo.

Revisa la operación de inserción en una lista abierta vacía para obtener más información.

Push en una pila no vacía

Si la pila ya contiene al menos un nodo, la operación de inserción es igual que la de insertar un elemento al principio de una lista abierta, de modo que puedes repasar aquella operación.

Aquí tienes una posible implementación en C de la operación push en forma de función. Esta implementación contempla las dos posibilidades (inserción en pila vacía y en pila no vacía). La función recibe dos parámetros: un puntero al primer elemento de la pila (cima) y un

número entero, que es el dato que se pretende insertar. Observa que, como el puntero al primer elemento ya es un puntero y hay que pasarlo por variable a la función, se trata en realidad de un doble puntero (**). Fíjate bien en las diferencias sintácticas que eso representa:

```
void push(t_nodo **primero,
int v)
{
   t_nodo *nuevo;
   nuevo =
(t_nodo*)malloc(sizeof(t_nodo));
              // Creamos nodo
nuevo
```

```
    nuevo->dato = v;
            // Insertamos el
dato en el nodo

    nuevo->siguiente =
*primero;                    //
La cima a partir de ahora será
"nuevo"
    *primero = nuevo;
}
```

Pop (extraer y borrar)

La operación pop consiste en leer el dato que hay en la cima de la pila (es decir, el que está apuntado por el puntero primero) y eliminarlo. La

operación de eliminación es exactamente igual que la que vimos referida a listas abiertas (borrado del primer elemento), así que puedes repasarla allí.

Esta es una posible implementación en C de la operación pop en forma de función. La función recibe como parámetro un puntero a la cima de la pila y devuelve el valor del dato que está en la cima, eliminando el nodo que lo contiene:

```
int pop(t_nodo **primero)
{
```

```
   t_nodo *aux;               //
Variable auxiliar para
manipular el nodo
   int v;                     //
Variable auxiliar para
devolver el valor del dato

   aux = *primero;
   if(aux == NULL)    // Si no
hay elementos en la pila
devolvemos algún valor
especial
      return -1;

   *primero = aux->siguiente;
// La pila empezará ahora a
partir del siguiente elemento
   v = aux->dato;
// Este es el dato que ocupaba
la cima hasta ahora
   free(aux);
```

```
// Liberamos la memoria
ocupada la anterior cima
   return v;
// Devolvemos el dato
}
```

COLAS

Qué es una cola y cómo funciona

Una cola es un tipo especial y simplificado de lista abierta. A uno de los extremos de la lista se le denomina cabeza, y al otro, cola. Sólo se pueden insertar nodos en la cola, y sólo se

pueden leer nodos en la cabeza. Además, como sucede con las pilas, la lectura de un dato siempre implica la eliminación del nodo que contiene ese dato.

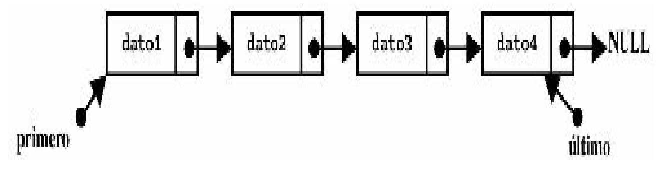
Este tipo de lista es conocido como lista FIFO (First In First Out, es decir, el primer elemento en entrar es el primero en salir). El nombre de "cola" proviene de la analogía con las colas de la vida real; por ejemplo, la cola para pagar en un supermercado. El primero que llega a la cola es el primero que sale de ella,

mientras que los que van llegando después se tienen que ir colocando detrás, y serán atendidos por orden estricto de llegada.

Las colas, como las pilas, son listas abiertas simplificadas que, sin embargo, se adaptan a la perfección a determinados problemas, por lo que, para resolver esos problemas, es preferible usar una cola en lugar de una lista.

Hemos dicho que las colas son listas abiertas simplificadas. Por lo tanto, la

representación interna será exactamente la misma, con la salvedad de que ahora necesitaremos dos punteros: uno al primer nodo (cabeza) y otro al último nodo de la lista (cola).



Tipos de datos para implementar colas

En los siguientes apartados vamos a trabajar con colas de números enteros (como siempre), pero se podría cambiar fácilmente con sólo modificar el tipo del

campo "dato" en la siguiente estructura de nodo:

```
struct s_nodo
{
   int dato;
   struct s_nodo *siguiente;
};
typedef struct s_nodo t_nodo;
t_nodo *primero;
t_nodo *ultimo;
```

Observa que los tipos necesarios con los mismos que en las listas abiertas, con la excepción de que ahora necesitamos dos punteros en lugar de uno: el que apunta al primer elemento (cabeza) y el que apunta al último

(cola).

Operaciones con colas

Las colas, volvemos a repetirlo, son listas abiertas simplificadas. Lo único que cambia respecto de las listas abiertas es el conjunto de operaciones, que en las colas es mucho más reducido (precisamente eso es lo que las hace más simples).

Así, las únicas operaciones permitidas con colas son:

- Insertar: Añade un elemento

al final de la cola.

- Leer: Lee y elimina un elemento del principio de la cola.

Insertar elementos

La inserción de elementos siempre se hace al final de la cola, es decir, a continuación de elemento apuntado por el puntero "último".

Insertar un elemento en una cola vacía

Supondremos que disponemos de un nuevo nodo que vamos a insertar en la cola (con un puntero, que llamaremos "nuevo", apuntando a él) y, por supuesto, los punteros "primero" y "último" que definen la cola.

Si la cola está vacía, ambos deben estar apuntando a NULL.

primero → NULL
último → NULL
nuevo → dato1

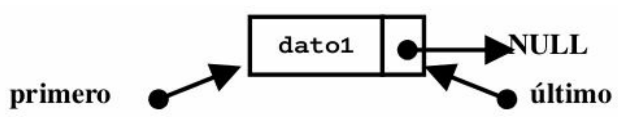
El proceso de inserción consiste en:

1) Hacer que nuevo->siguiente apunte a NULL.

2) Hacer que primero apunte a nuevo.

3) Hacer que último apunte a nuevo.

El estado final de la cola debe ser este:



Una posible implementación en C de este algoritmo de inserción puede ser:

```
t_nodo*
nuevo;
nuevo = (t_nodo*)
malloc(sizeof(t_nodo));     //
Se reserva memoria para el
```

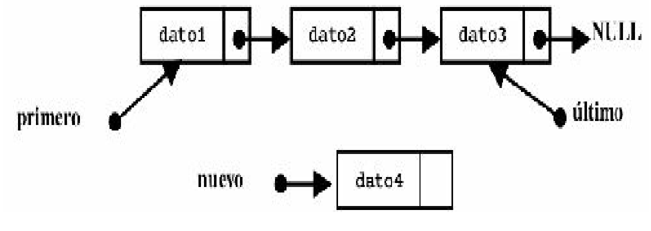
```
nuevo nodo
nuevo->dato =
5;
// Insertamos un dato en el
nuevo nodo
nuevo->siguiente = NULL;
                  // Hacemos
que el nuevo apunte a NULL
primero = nuevo;

// Hacemos que el primero y el
último apunten al nuevo
ultimo = nuevo;
```

El tipo t_nodo y los punteros primero y último han debido ser declarados con anterioridad. Supondremos que la cola almacena números enteros. En este ejemplo, el dato que se inserta en el

nodo nuevo es el número 5.

Insertar un elemento en una cola no vacía

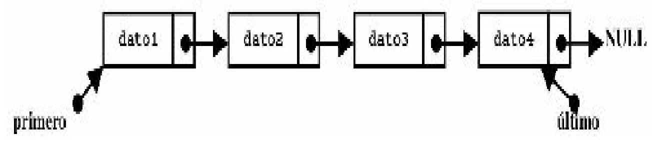
En esta ocasión partiremos de una cola no vacía (en la siguiente figura dispone de 3 elementos), y de un nuevo nodo al que podemos acceder a través de un puntero llamado "nuevo":



Para insertar un nodo en estas condiciones hay que seguir estos pasos:

- Hacer quee nuevo->siguiente apunte a NULL.
- Hacer que ultimo->siguiente apunte a nuevo.

El resultado debe ser esta otra cola, en la que el nuevo elemento se ha insertado al final.



Como hacemos siempre, vamos a

proponer una posible implementación de este algoritmo en C. El dato insertado en este ejemplo será el número 25:

```
t_nodo*
nuevo;
nuevo = (t_nodo*)
malloc(sizeof(t_nodo));
           // Se reserva
memoria para el nuevo nodo
nuevo->dato =
25;                     //
Insertamos un dato en el nuevo
nodo
nuevo->siguiente = NULL;

           // Hacemos que el
nuevo apunte a NULL
ultimo->siguiente =
nuevo;          //
```

```
Enganchamos el nuevo al final
de la cola
ultimo =
nuevo;                       //
A partir de ahora, el nuevo
será el último
```

Leer y eliminar elementos

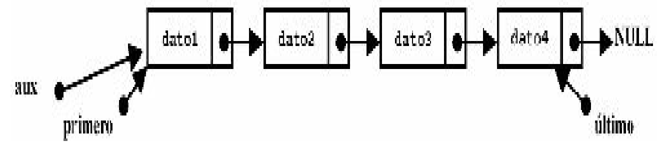
Recuerda que la lectura de un dato de la cola siempre se hace por la cabeza y siempre implica la eliminación automática del nodo.

Distinguiremos dos casos: cuando la cola contiene más de un elemento y

cuando tiene sólo uno. Podríamos añadir un tercero: cuando la cola no tiene elementos, pero, en ese caso, la operación de lectura no tiene sentido.

Leer un elemento en una cola con más de un elemento

Necesitaremos un puntero auxiliar que apunte al primer elemento de la cola (es decir, a la cabeza):

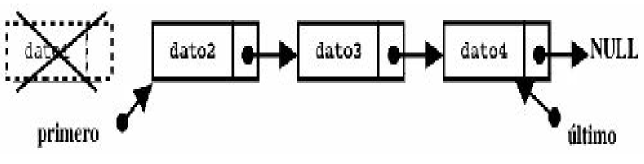


Disponiendo de esos punteros, la operación de lectura se realiza así:

- Hacemos que aux apunte a primero.
- Hacemos que primero apunte al segundo elemento de la cola, es decir, a primero->siguiente.
- Guardamos el dato contenido en aux para devolverlo como valor del elemento
- Eliminamos el nodo apuntado

por aux (mediante la función free() o similar)

El resultado de estas acciones será el siguiente:



En la implementación en C, fíjate como se salva el dato contenido en el nodo antes de eliminarlo, para así poder usarlo en el programa como más convenga:

```
t_nodo*
```

```
aux;
int valor;
aux =
primero;                          /
Hacemos que aux apunte a la
cabeza
primero = primero->siguiente;
          // Hacemos que
primero apunte al segundo
valor = aux->dato;
          // Guardamos en una
variable el dato contenido en
el nodo
free(aux);
          // Eliminamos el
primer nodo
```

Leer un elemento en una cola con un sólo elemento

La forma de proceder es la misma, pero ahora hay que añadir una cosa más: hay que hacer que el puntero "último" pase a apuntar a NULL, ya que la cola se quedará vacía:

Por lo tanto, partimos de esta situación:

Y, después de ejecutar todos los pasos, debemos llegar a esta:

Fíjate en que no es necesario hacer que primero apunte a NULL, sino que basta con hacer que apunte a primero->siguiente, según establece el algoritmo general. Por lo tanto, la implementación en C puede ser esta:

```
t_nodo*
aux;
int valor;
aux =
primero;                          /
Hacemos que aux apunte a la
cabeza
primero = primero->siguiente;
           // Hacemos que
primero apunte al segundo
valor = aux->dato;
```

```
           // Guardamos en una
variable el dato contenido en
el nodo
free(aux);
           // Eliminamos el
primer nodo
if (primero ==
NULL)               // ¡La cola se
ha quedado vacía!
    ultimo =
NULL;               // Hacemos que
el último también apunte a
NULL
```

Ejemplo de implementación en C

A continuación presentamos una posible

implementación en C de las operaciones de inserción y lectura en una cola que tienen en cuenta todos los casos vistos anteriormente. Las operaciones están escritas como funciones para que así puedan ser utilizadas desde cualquier programa.

Supondremos que ya está declarado el tipo t_nodo (como vimos al principio de este epígrafe dedicado a las colas) y que se trata de una cola de números enteros (ya sabes que para hacer una cola con otros datos basta con modificar la

definición de la estructura). Las funciones deben recibir como parámetros los punteros a la cabeza y a la cola. Además, la función de inserción debe recibir el dato que se desea insertar, y la de lectura debe devolverlo en un return.

Como nos ha ocurrido en otras implementaciones anteriores, al convertir cada operación en una función sucede algo que puede confundirte un poco: el puntero al primer elemento (y al último) de la cola debe ser pasado a

la función por variable, es decir, en forma de puntero. Pero como ya es en sí mismo un puntero, el parámetro se convierte en un puntero a puntero (**primero) o doble puntero. Observa con detenimiento las implicaciones que ello tiene en la sintaxis de la función:

```
void insertar(t_nodo
**primero, t_nodo **ultimo,
int v) {
   t_nodo* nuevo;
   nuevo =
(t_nodo*)malloc(sizeof(t_nodo))
Creamos el nuevo nodo
   nuevo->dato =
v;           // Le asignamos
el dato
```

```
   nuevo->siguiente = 
NULL;              // El nuevo 
nodo apuntará a NULL
   if (*ultimo != NULL)
           // Si la cola no 
estaba vacía...
       (*ultimo)->siguiente = 
nuevo;              // 
...enganchamos el nuevo al 
final
   *ultimo = 
nuevo;              // A partir 
de ahora, el nuevo será el 
último
   if (*primero == NULL)
           // Si la cola 
estaba vacía...
       *primero = 
nuevo;              // ...el 
último también será el primero
}
```

```
int leer(t_nodo **primero,
t_nodo **ultimo) {
   t_nodo *aux;             //
Puntero auxiliar
   int v;             // Para
almacenar el valor del dato y
devolverlo

   aux =
*primero;             // El
auxiliar apunta a la cabeza
   if(aux == NULL)
           // La cola está
vacía: devolver valor especial
      return -1;
   *primero = aux-
>siguiente;             // El
primero apunta al segundo
   v = aux->dato;
           // Recoger valor
del primero
```

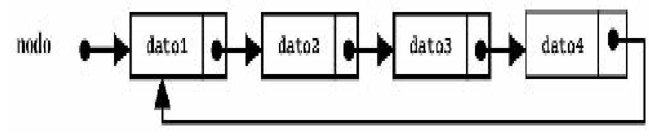
```
    free(aux);              //
Eliminar el nodo primero
    if (*primero==NULL)
            // Si la cola se ha
quedado vacía...
        *ultimo =
NULL;            // ...hacer
que el último también apunte a
NULL
    return v;              //
Devolver el dato que había en
el primero
}
```

OTROS TIPOS DE LISTAS

Listas circulares

Una lista circular es una variedad de

lista abierta en la que el último nodo a punta al primero en lugar de apuntar a NULL.



En estas listas el concepto de "nodo primero" es una convención, porque en realidad no existe: todos los nodos son anteriores a otro y siguientes de otro. No hay principio ni fin de la lista, aunque debemos mantener un puntero a al menos uno de los nodos para poder iniciar desde él las operaciones sobre la lista.

En las listas circulares no hay casos especiales, salvo que la lista este vacía.

Los tipos de datos que se emplean son los mismos que en el caso de las listas abiertas. Así, para construir una lista de números enteros necesitaremos una estructura de este tipo:

```
struct s_nodo
{
   int dato;
   struct s_nodo *siguiente;
};
typedef struct s_nodo t_nodo;
t_nodo* nodo;
```

Fíjate que el puntero a un nodo de la

lista lo hemos llamado "nodo" en lugar de "primero". Esto se debe a que, como hemos dicho, en una lista circular no hay "primero" ni "último". Recuerda que para construir una lista con otros datos que no sean de tipo entero, bastaría con cambiar la definición del campo "dato" en la estructura s_nodo.

En cuanto a las operaciones básicas que se pueden realizar con listas circulares, son las mismas que con listas abiertas, es decir:

- Insertar elementos.

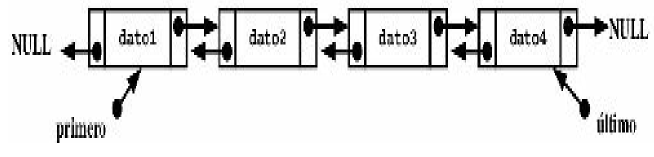
- Buscar elementos.
- Borrar elementos.

A estas operaciones básicas le podemos añadir cuantas operaciones secundarias nos sean necesarias.

Listas doblemente enlazadas

Una lista doblemente enlazada es una variedad de lista abierta en la que cada nodo tiene dos enlaces: uno al nodo siguiente y otro al anterior.

Las listas doblemente enlazadas pueden recorrerse en ambos sentidos (de atrás hacia delante y al revés) a partir de cualquier nodo. Necesitaremos, como en las otras listas, de, como mínimo, un puntero a alguno de los nodos de la lista, para a partir de él poder acceder al resto. Es habitual, sin embargo, mantener dos punteros: uno al primer elemento y otro al último (como en las colas).



El tipo de dato básico para construir los nodos de la lista es diferente al de las listas abiertas, ya que ahora necesitamos dos punteros en cada nodo. Así, para construir, por ejemplo, una lista doblemente enlazada de números enteros necesitaremos esta estructura:

```
struct s_nodo
{
   int dato;
   struct nodo *siguiente;
   struct nodo *anterior;
};
typedef struct s_nodo t_nodo;
t_nodo *primero, *ultimo;
```

El repertorio de operaciones básicas es

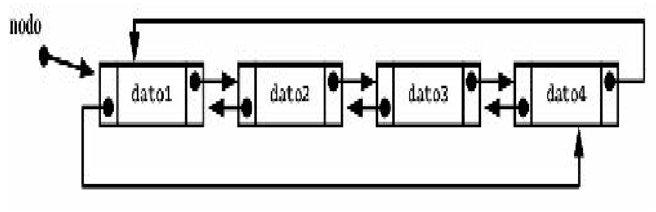
el mismo que en el resto de listas:

- Insertar elementos.
- Buscar elementos.
- Borrar elementos.

Listas circulares doblemente enlazadas

Por último, señalar que es habitual (y bastante útil) combinar la lista doblemente enlazada con la lista circular, obteniendo así listas circulares

doblemente enlazadas, en las que el nodo siguiente al último es el primero, y el anterior del primero es el último. En realidad, el concepto "primero" y "último" se diluye:



Estas son, sin duda, las listas más versátiles, porque permiten recorrer los nodos hacia delante o hacia atrás partiendo de cualquier punto. Como contrapartida, son las más difíciles de

implementar.

Árboles generales

Qué es un árbol y cómo funciona

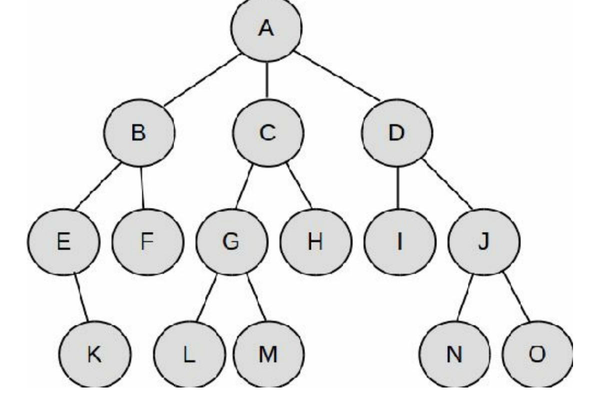
Las estructuras dinámicas que hemos visto hasta ahora (listas, pilas, colas...) son lineales. El procesamiento de estas estructuras es siempre secuencial, es decir, los datos se procesan de uno en uno, ya que cada uno contiene el enlace al siguiente.

Esto es muy útil en ciertos tipos de problemas de naturaleza secuencial, pero es muy lento cuando se tiene una gran cantidad de información, ya que para encontrar un dato puede ser necesario recorrer, uno por uno, todos los elementos de una larguísima lista.

Por eso se han ideado estructuras no líneales, en los que cada elemento no tiene un "elemento siguiente", sino varios. Eso es un árbol: una estructura no lineal en la que cada nodo puede apuntar varios nodos.

También se suele dar una definición recursiva: un árbol es una estructura en la que, de cada nodo, surge otro árbol.

Esto son definiciones simples, pero lo que implican no lo es tanto. El árbol es una estructura muy potente y versátil que permite manejar enormes cantidades de información. Por ejemplo, esto es un árbol:



Para trabajar con árboles hay varios conceptos básicos que pasamos a definir:

- Nodo padre: cualquier nodo que contiene un al menos otro nodo colgando de él. En la figura anterior, el nodo 'A' es

padre de 'B', 'C' y 'D'.

- Nodo hijo: cualquier nodo que tenga nodo padre. Todos los nodos son hijos de otro, excepto el que ocupa el primer puesto del árbol. En la figura anterior, 'L' y 'M' son hijos de 'G'. Las relaciones padre-hijo dentro de un árbol hacen de ésta una estructura jerárquica, en la que unos nodos están por encima de otros.

- Nodo raíz: es el nodo que no tiene padre. Cada árbol tiene un único nodo raíz. En el ejemplo, ese nodo es el 'A'.
- Nodo hoja: son los nodos que no tienen hijos, es decir, los que ocupan la posición más baja en el árbol. En el árbol de la figura hay varios: 'F', 'H', 'I', 'K', 'L', 'M', 'N' y 'O'.
- Nodo intermedio: así denominamos a los nodos que no son ni raíz ni hoja. A

veces también se llaman nodos-rama. En el árbol de la figura, son nodos intermedios 'B', 'C', 'D', 'E', 'G' y 'J'.

Existen otras características que definen a un árbol, y son:

- Orden: es el número máximo de hijos que puede tener cada nodo. Así, existen árboles binarios (de orden 2), ternarios (de orden 3), cuaternarios (de orden 4) o, en general, N-arios (de orden

N). En un árbol binario, cada nodo puede tener 0, 1 ó 2 hijos. En uno ternario, 0, 1, 2 ó 3 hijos, y así sucesivamente.

- Se dice que un árbol está completo cuando cada nodo tiene o todos o ninguno de sus hijos. Es decir, un árbol binario está completo si todos los nodos tienen exactamente 2 hijos (o ninguno, si son nodos-hoja).

Un árbol termario está completo si todos los nodos tienen exactamente 3 hijos (o ninguno), y así sucesivamente

- Grado: el número de hijos que tiene el nodo con más hijos dentro del árbol. En el árbol del ejemplo anterior, el grado es tres, ya que tanto 'A' como 'D' tienen tres hijos, y no existen elementos con más de tres hijos.
- Nivel o Profundidad: se

define para cada elemento del árbol como la distancia a la raíz medida en nodos. El nivel de la raíz es cero, el de sus hijos, uno, el de sus nietos, dos, etc. En el ejemplo anterior, el nodo 'D' tiene nivel 1, el nodo 'G' tiene nivel 2, y el nodo 'N' tiene nivel 3.

- Altura: la altura de un árbol se define como el nivel del nodo de mayor nivel. El

árbol del ejemplo tiene altura 3.

Tipos de datos para implementar árboles

Al usar listas, colas y pilas, debíamos mantener siempre un puntero al primer elemento (y, a veces, también al último) para poder acceder a cualquier elemento de la estructura de datos. Al usar árboles, debemos tener un puntero al nodo raíz, ya que a partir de él se desarrolla el árbol y podemos acceder a cualquier otro nodo.

En adelante, vamos a suponer que estamos usando un árbol de números enteros. Ya sabes que para almacenar otro tipo de dato en el árbol basta con modificar la definición del nodo.

Supongamos que deseamos crear un árbol de orden 3. En ese caso, la definición del nodo sería algo así:

```
struct s_nodo
{
   int dato;
   struct s_nodo *hijo1;
   struct s_nodo *hijo2;
   struct s_nodo *hijo3;
};
typedef struct s_nodo t_nodo;
```

```
  t_nodo *raiz;
```

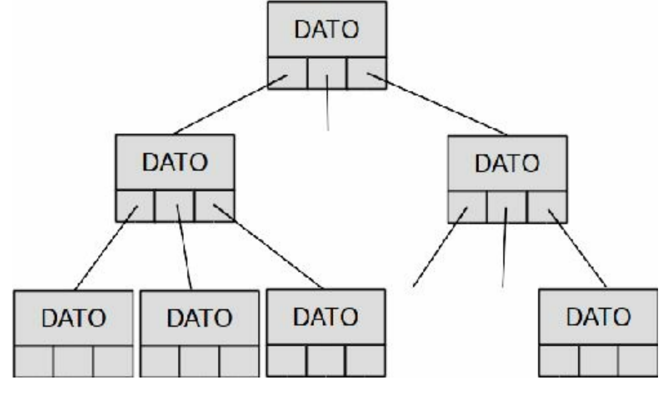
Fíjate que el nodo contiene tres punteros, uno para cada uno de los potenciales hijos, ya que el árbol es de orden 3. Si queremos construir un árbol de orden 4, hay que añadir otro puntero a s_nodo dentro de la estructura. Y si queremos un árbol de orden 2 (binario), hay que quitar uno de los punteros.

Observa la siguiente definición de nodo porque es mucho más general y, por lo tanto, es la que debemos usar. En ella, los punteros a los nodos-hijo se definen

como un array de punteros. Basta con cambiar el rango de ese array para cambiar el orden del árbol. Por ejemplo, para definir un árbol de orden 3:

```
#define ORDEN 3
struct s_nodo
{
   int dato;
   struct s_nodo *hijo[ORDEN];
};
typedef struct s_nodo t_nodo;
t_nodo *raiz;
```

Una posible representación gráfica de un árbol de orden 3 como el que acabamos de definir es:



Operaciones con árboles

Como ocurría con las listas, implementar un "tipo de dato árbol" consiste en declarar las estructuras de datos necesarias (como hemos explicado más arriba) y programar una colección

de funciones para que se puedan realizar operaciones sobre el árbol, tales como insertar elementos, borrarlos o buscarlos.

De nuevo tenemos casi el mismo repertorio de operaciones básicas que disponíamos con las listas. Al fin y al cabo, un árbol no deja de ser una estructura de datos y, por lo tanto, sobre él se pueden hacer las operaciones de:

- Inserción de elementos.
- Búsqueda de elementos.

- Eliminación de elementos.
- Recorrido del árbol completo.

Los algoritmos de inserción, búsqueda y borrado dependen en gran medida del tipo de árbol que estemos implementando. En otros apartados posteriores estudiaremos esas operaciones aplicadas a un tipo de árbol específico. Por ahora, nos centraremos en el modo de recorrer árboles, que es general para cualquier tipo de árbol.

También disponemos, como antes, de

otro conjunto mucho más extenso de operaciones secundarias que, si bien no son imprescindibles, sí que pueden ser muy útiles para determinados problemas. Algunas de esas operaciones pueden ser: contar los elementos que hay en un árbol, comprobar si el árbol está vacío, calcular la profundidad del árbol, etc.

Recorridos por árboles

El modo evidente de moverse a través

de las ramas de un árbol es siguiendo los punteros, del mismo modo en que nos movíamos a través de las listas. Los recorridos dependen en gran medida del tipo y propósito del árbol, pero hay ciertos recorridos que usaremos frecuentemente: los que incluyen todo el árbol.

Supongamos que tenemos un árbol de orden tres que contiene números enteros, y queremos recorrerlo por completo para mostrar por pantalla el contenido de todos los nodos. Partiremos del nodo

raíz:

```
recorrer_arbol(raiz);
```

La función recorrer_arbol() será tan sencilla como invocar recursivamente a la función recorrer_arbol() para cada una de los hijos del nodo actual:

```
void recorrer_arbol(t_nodo*
nodo)
{
   printf("%i", nodo->dato);
   if (nodo == NULL) return;
   recorrer_arbol(nodo-
>hijo[0]);
   recorrer_arbol(nodo-
>rama[1]);
   recorrer_arbol(nodo-
>rama[2]);
```

```
    }
```

Esta función se limita a mostrar por pantalla el dato contenido en el dato y luego continúa el recorrido. Evidentemente, puede modificarse para que haga alguna otra cosa con el dato contenido en el nodo, dependiendo del problema que estemos resolviendo.

Existen tres formas de recorrer un árbol. Las tres utilizan este esquema que acabamos de ver con pequeñas variaciones. En la siguiente tabla se describen y se muestra el código

correspondiente para un árbol de orden 3 (suponiendo que lo único que queremos hacer con cada nodo es mostrarlo por la pantalla):

Recorrido en pre-orden	Recorrido en
Consiste en procesar el nodo actual antes de lanzar el recorrido de los nodos-hijo	Consiste en p nodo actual d haber proces nodo-hijo y a procesar el u hijo (tiene m árbol es bina

<table>
<tr><td></td><td>si cada nodo
hijos)</td></tr>
<tr><td>

```
void
recorrer_arbol(t_nodo*
nodo)
{
   printf("%i", nodo-
>dato);
   if (nodo == NULL)
return;
   recorrer_arbol
             (nodo-
>hijo[0]);
   recorrer_arbol
             (nodo-
>hijo[1]);
   recorrer_arbol
             (nodo-
>hijo[2]);
```

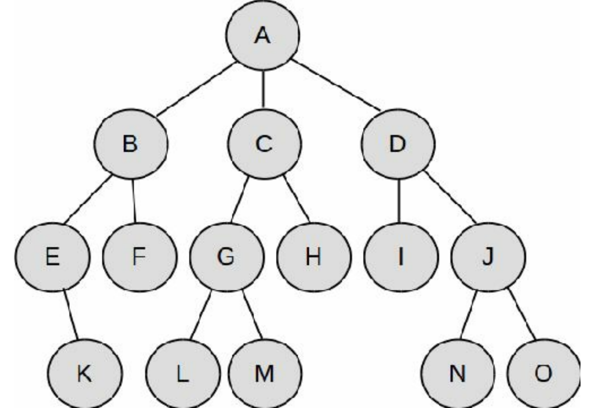
</td><td>

```
void
recorrer_a
nodo)
{
   if (nod
return;
   recorre

>hijo[0]);
   printf(
>dato);
   recorre

>hijo[1]);
   recorre

>hijo[2]);
```

</td></tr>
</table>

}	}

Puede parecer que no hay gran diferencia entre los tipos de recorrido, pero el efecto sobre el orden en el que se procesan los nodos es radical. Veámoslo con un ejemplo. Supongamos que tenemos el siguiente árbol de orden 3:



- El recorrido en pre-orden procesará los nodos en este orden: A B E K F C G L M D H I J N O

- El recorrido en in-orden procesará los nodos en este

orden: K E B F A L G M C H D I N J O

- El recorrido en post-orden procesará los nodos en este orden: K E F B L M G C H I N O J D A

Como ves, el método elegido produce efectos diferentes, por lo que debemos usar el más apropiado para el problema que estemos resolviendo.

ÁRBOLES BINARIOS DE BÚSQUEDA

Qué son los ABB y cómo funcionan

A partir de ahora sólo hablaremos de árboles ordenados, ya que son los que tienen más aplicaciones tienen en el campo de la programación. Pero, ¿qué es un árbol ordenado?

Un árbol ordenado, en general, es aquel a partir del cual se puede obtener una secuencia ordenada siguiendo uno de los recorridos posibles del árbol (in-orden, pre-orden o post-orden). Es fundamental que la secuencia se mantenga ordenada

aunque se añadan o se eliminen nodos, por lo que las funciones de inserción y eliminación de datos son diferentes en árboles ordenados que en los que no lo son.

Existen varios tipos de árboles ordenados:

- Árboles binarios de búsqueda (ABB): son árboles de orden 2 que mantienen una secuencia ordenada si se recorren en in-orden.

- Árboles AVL: son árboles binarios de búsqueda equilibrados, es decir, los niveles de cada rama para cualquier nodo no difieren en más de 1.
- Árboles perfectamente equilibrados: son árboles binarios de búsqueda en los que el número de nodos de cada rama para cualquier nodo no difieren en más de 1. Son por lo tanto árboles AVL

también.

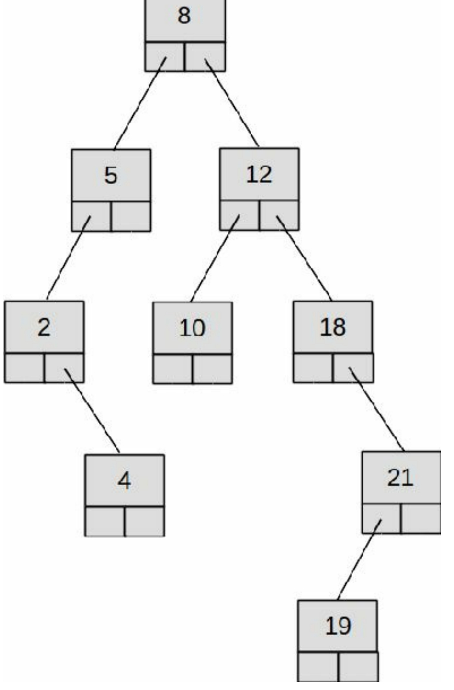
- Árboles 2-3: son árboles de orden 3, que contienen dos claves en cada nodo y que están también equilibrados. También generan secuencias ordenadas al recorrerlos en in-orden.
- Árboles-B: caso general de árboles 2-3, que para un orden M, contienen M-1 claves.

En este apartado nos vamos a centrar en

los árboles binarios de búsqueda o ABB. Se trata de árboles de orden 2 que cumplen una condición: el dato de cada nodo es mayor que el dato de su hijo izquierdo y menor que el de su hijo derecho. Lógicamente, los nodos-hoja no tienen que cumplir esta condición porque no tienen hijos.

Aquí tienes un ejemplo de árbol ABB. Observa que todos los nodos cumplen la condición que acabamos de mencionar y que, para un nodo concreto (por ejemplo, el raiz), todos los nodos que

cuelgan a su derecha son mayores que él, y todos los de la izquierda son menores.



Las definiciones de estructuras de datos y variables necesarias para construir un

árbol binario que contenga números enteros (ya sabes que puedes cambiar los datos contenidos en el árbol modificando el campo "dato" de la estructura) son:

```
struct s_nodo
{
   int dato;
   struct s_nodo *hijo_der;
   struct s_nodo *hijo_izq;
};
typedef struct s_nodo t_nodo;
t_nodo *raiz;
```

El repertorio de operaciones que se pueden realizar sobre un ABB es similar al de cualquier otro árbol. Las

operaciones sobre árboles tienen un denominador común: la recursividad. Las soluciones recursivas, como veremos a continuación al estudiarlas con detenimiento, proporcionan algoritmos simples y elegantes para manejar estructuras complejas como son los árboles.

Buscar un elemento

Partiendo de un nodo, el modo de buscar un elemento se define de forma recursiva (aunque también es posible

una solución iterativa):

- Si el nodo actual está vacío, terminamos la búsqueda: el elemento no está en el árbol.
- Si el dato del nodo actual es igual que el del elemento que buscamos, terminamos la búsqueda con éxito.
- Si el dato del nodo actual es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.

- Si el dato del nodo actual raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

El valor de retorno de una función de búsqueda en un ABB puede ser un puntero al nodo encontrado o NULL si no se ha encontrado. Una posible implementación recursiva en C es:

```
t_nodo* buscar(t_nodo *nodo,
int valor_buscado)
{
   t_nodo *result;
   if (nodo ==
```

```
NULL)
Caso base 1
      result = NULL;
   if (nodo->dato ==
valor_buscado)
Caso base 2
      result = nodo;
   if (nodo->dato >
valor_buscado)
Caso general 1
      result = buscar(nodo-
>hijo_izq, valor_buscado);
   if (nodo->dato <
valor_buscado)
Caso general 2
      result = buscar(nodo-
>hijo_der, valor_buscado);

   return result;
}
```

Para iniciar la búsqueda de un dato cualquiera en el árbol (por ejemplo, para buscar el número 10) haremos una invocación inicial a la función pasándole como parámetro el puntero al nodo raíz, para que comience a buscar desde la raíz, algo así:

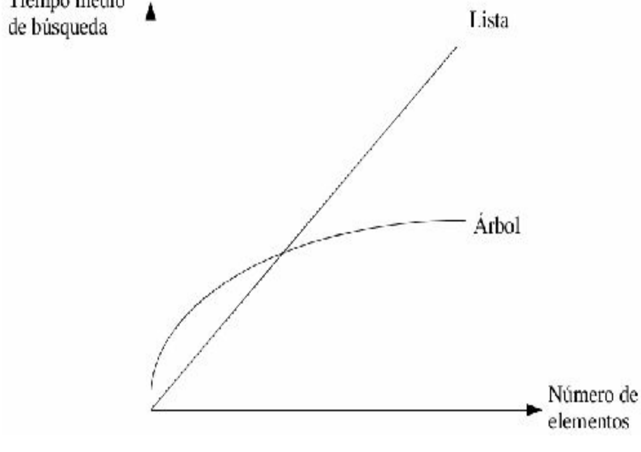
```
t_nodo* nodo;
nodo = buscar(raiz, 10);
if (nodo == NULL)
   printf("Elemento no
encontrado");
else
   ...<procesar elemento>...
```

Otra operación bastante útil puede ser

buscar el nodo-padre de un nodo dado. Se puede implementar fácilmente una función que se encargue de ello modificando un poco la función buscar() que acabamos de presentar. También se puede construir el árbol con un puntero desde cada nodo a su nodo-padre. En este caso, la búsqueda del padre de un nodo dado es instantánea, ya que el propio nodo guardaría un puntero al padre.

La localización de un elemento en un árbol ordenado es una de sus mejores

cualidades. Incluso con cantidades enormes de datos, la búsqueda en un árbol de este tipo es una operación rapidísima comparada con el tiempo de búsqueda en una lista. La siguiente figura muestra gráficamente la diferencia:



Si el número de elementos es muy pequeño, es mejor utilizar una lista porque los tiempos de búsqueda son menores. Pero cuando el número de elementos pasa de un determinado umbral, el tiempo medio de búsqueda es

notablemente menor en el árbol que en la lista. Esta diferencia se va acrecentando conforme crece el número de elementos, ya que, aunque ambos tiempos tienden a infinito (como es lógico), el tiempo de búsqueda en la lista crece linealmente, mientras que el tiempo de búsqueda en el árbol lo hace logarítmicamente.

Insertar un elemento

Para insertar un elemento nos basamos

en el algoritmo de búsqueda. Si el elemento está en el árbol no lo insertaremos. Si no está (cosa que sabremos si alcanzamos un nodo NULL), lo insertaremos en la posición que le corresponda. Para eso es necesario que cada nodo tenga un puntero a su nodo-padre.

El proceso se iniciará en el nodo-raíz e irá recorriendo los nodos. Para cada nodo se realizarán estos pasos:

- Si el nodo actual está vacío, creamos un nodo nuevo (con

malloc() o una función similar) e insertamos en él el dato. Los hijos de este nodo nuevo deberán apuntar a NULL, ya que aún no existen. En el nodo padre, almacenamos un puntero a este nuevo nodo-hijo. Será el hijo izquierdo si el dato es menor que el almacenado en el padre, o el hijo derecho si el dato es mayor.

- Si el dato del nodo actual es

igual que el del elemento que queremos insertar, terminamos sin insertar nada porque el dato ya existe.

- Si el dato del nodo actual es mayor que el elemento que queremos insertar, intentaremos insertarlo en la rama derecha.
- Si el dato del nodo actual es menor que el elemento que queremos insertar, intentaremos insertarlo en la

rama izquierda.

La implementación en C, por tanto, es muy parecida a la de la función buscar(), pero añadiendo el puntero al nodo padre en la llamada a la función:

```
void insertar(t_nodo **nodo,
t_nodo **padre, int valor)
{
   if (*nodo == NULL)    //
Insertamos el dato aquí
   {
      *nodo = (t_nodo*)
malloc(sizeof(t_nodo));    //
Creamos el nodo
      (*nodo)->dato = valor;
// Asignamos el dato
      (*nodo)->hijo_izq =
```

```
NULL;                    // No
tiene hijo izquierdo
      (*nodo)->hijo_der =
NULL;                    //
Tampoco derecho
      if (*padre != NULL)
      {
        if (valor > (*padre)-
>valor)
          (*padre)->nodo_der =
nodo;   // El nuevo nodo será
hijo derecho
        else
          (*padre)->nodo_izq =
nodo;   // El nuevo nodo será
hijo izquierdo
      }
   }
   else               //
Seguimos buscando el lugar
adecuado de inserción
```

```
    {
        if (valor > (*nodo)->dato)     // Intentamos insertar en la rama derecha
            insertar(&(*nodo)->hijo_der, nodo, valor);
        if (valor < (*nodo)->dato)     // Intentamos insertar en la rama izquierda
            insertar(&(*nodo)->hijo_izq, nodo, valor);
        // si valor == nodo_dato no se hace nada (el dato ya está en el árbol)
    }
}
```

Para realizar la inserción de un elemento debemos iniciar el proceso recursivo desde la raíz, pasando NULL como

padre, ya que el nodo raíz no tiene padre, así:

```
insertar(raiz, NULL, 3);             // Inserta el número 3 en el árbol
```

Si observas la función anterior podrás comprobar que también funciona cuando el árbol está vacío, es decir, cuando raiz es NULL.

Borrar un elemento

Diferencia entre borrado y poda

En la operación de borrado debemos

distinguir entre dos casos: el borrado de un nodo hoja y el borrado de un nodo intermedio, del cual cuelgan otros nodos. Para distinguirlos, al segundo tipo de borrado lo llamaremos poda.

El proceso de borrado de un nodo hoja es muy sencillo:

- En el nodo padre, buscar el puntero al nodo que acabamos de eliminar y hacer que apunte a NULL
- Eliminar el nodo, liberando la memoria que ocupaba con

free() u otra función similar.

Si el nodo que se pretende borrar no es una hoja estamos ante una poda del árbol. En ese caso eliminaremos todo el sub-árbol que cuelga del nodo que queremos borrar. Se trata de un procedimiento recursivo en el que debemos aplicar el recorrido post-orden. El resultado será la eliminación del nodo y de todo lo que cuelga de él.

Podemos resumir el procedimiento de poda en estos pasos:

- Lanzar el borrado de todos

los nodos-hijo del nodo que queremos borrar mediante llamadas recursivas a este mismo procedimiento.

- En el nodo padre, buscar el puntero al nodo que acabamos de eliminar y hacer que apunte a NULL
- Eliminar el nodo, liberando la memoria que ocupaba con free() u otra función similar.

Implementación de un algoritmo de

borrado y poda

Para implementar en C un algoritmo de borrado y poda usaremos de nuevo una función recursiva, a la que pasaremos un puntero al nodo que se pretende borrar y un puntero al padre de éste.

La función lanzará el borrado recursivo de los dos hijos antes de borrar el nodo. Si el nodo es una hoja, los dos hijos serán punteros a NULL, y la llamada recursiva a borrar() no hará nada. Si el nodo es intermedio, se producirá un recorrido del árbol en post-orden que

hará que se borren los nodos de abajo a arriba.

Después, el puntero del padre que apuntaba al nodo que vamos a borrar se podrá a NULL y el nodo será borrado liberando la memoria.

```
void borrar(t_nodo **nodo,
t_nodo **padre)
{
   if (*nodo != NULL)     // Si
el nodo es NULL no hay que
borrarlo, claro
   {
      borrar(&(*nodo)-
>hijo_izq, nodo);            //
Borramos hijo izquierdo
      borrar(&(*nodo)-
```

```
>hijo_izq, nodo);               //
Borramos hijo derecho
      if (*padre != NULL)
                                //
Buscamos puntero del padre
      {
        if ((*nodo)->valor >
(*padre)->valor)  // El nodo
era el hijo derecho
          (*padre)->nodo_der =
NULL;

 else
  // El nodo era el hijo
izquierdo
          (*padre)->nodo_izq =
NULL;
      }

 free(*nodo);
Liberamos la memoria
```

```
    }
}
```

Combinando las funciones de búsqueda y borrado se puede eliminar cualquier elemento de un árbol. Por ejemplo, para borrar el nodo que contenga el dato "18" podríamos hacer esto:

```
t_nodo *nodo, *padre;
nodo = buscar(raiz, 18);
if (nodo == NULL)
   printf("El dato 18 no está
en el árbol");
else
{
   padre =
buscar_nodo_padre(raiz, 18);
   borrar(nodo, padre);
```

```
    printf("Nodo borrado");
  }
```

También se puede programar la función de borrado de un nodo intermedio de manera que no se pode toda la rama que cuelga de él, sino que se borre únicamente ese nodo y el resto se reorganicen para que el árbol permanezca ordenado. Como ves, existen muchas variedades de árboles y, dentro de cada una, se puede escoger entre infinidad de implementaciones.

Otras operaciones

Además de las operaciones básicas (inserción, borrado, búsqueda...), sobre un árbol es posible llevar a cabo otras operaciones que se pueden implementar como funciones independientes.

Por ejemplo, comprobar si un árbol está vacío. Un árbol está vacío si su raiz es NULL. Se puede programar una función que se encargue de hacer la comprobación, así:

```
int esta_vacio(t_nodo* raiz)
{
   if (raiz == NULL)
     return 1;
   else
```

```
        return 0;
  }
```

El uso de esta función desde otra parte del programa es muy sencillo. Por ejemplo:

```
  if (esta_vacio(raiz))
     printf("El árbol está
  vacío");
  else
     printf("El árbol no está
  vacío");
```

De un modo similar podríamos escribir funciones sencillas para otras operaciones adicionales que suelen resultar muy útiles. A continuación

mencionamos algunas:

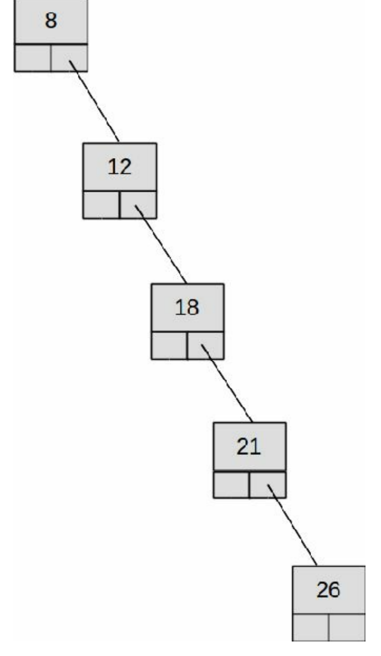
- Comprobar si un árbol está vacío
- Contar el número de nodos de un árbol
- Comprobar si un nodo es hoja, es decir, si sus dos hijos son NULL
- Calcular la altura de un nodo, es decir, su distancia desde la raíz
- Calcular la altura máxima de

un árbol

- Etc.

Árboles degenerados

Los árboles binarios de búsqueda tienen un gran inconveniente. Por ejemplo, supongamos que creamos un ABB a insertando en él una lista de valores ordenada: 8, 12, 18, 21, 26. Difícilmente podremos llamar a la estructura resultante un árbol:



Esto es lo que llamamos un árbol binario de búsqueda degenerado. Si te fijas bien, se trata de una lista cuyo

primer elemento es la raíz. Por lo tanto hemos perdido la ventaja de los árboles sobre las listas (que pueden manejar enormes cantidades de información en un tiempo récord)

La solución a este problema pasa por una operación llamada reequilibrado o rebalanceo del árbol, de manera que se obtenga un árbol bien balanceado. La construcción y manipulación de árboles balanceados (también llamados árboles AVL) es más compleja que la de los árboles binarios convencionales, como

bien te puedes imaginar, y escapa a los propósitos de este texto.

Recursividad

La rescursividad no forma parte de las estructuras de datos dinámicas, sino que es una estrategia más para resolver problemas. Y una muy poderosa. La recursividad es como la dinamita: permite horadar hasta el corazón de una montaña, pero también puede explotarte en las manos.

Como profesor, nunca he sido muy

partidario de introducir la recursividad demasiado pronto en los cursos de programación. En mi experiencia, a la mayor parte de las personas les resulta difícil aplicarla correctamente, y un elevado porcentaje de programadores principiantes tienen la malsana costumbre de intentar hacer que todos sus algoritmos sean recursivos (a veces, hasta involuntariamente).

Por ese motivo, y porque está muy emparentada con los árboles, este puede ser un momento tan bueno como

cualquier otro para hablar de ella. Ten en cuenta que la recursividad no es exclusiva del lenguaje C ni de ningún otro lenguaje, sino que se trata, tan solo, de una estructura algorítmica más, es decir, de otra llave inglesa que añadir a tu caja de herramientas.

Planteamiento de una solución recursiva

La recursión es, al principio, un concepto fácil de entender pero difícil

de aplicar. Una vez asimilado, no obstante, te asombrarás de lo sencillo que resulta resolver determinados problemas de naturaleza recursiva.

Un problema es recursivo cuando su solución se define en términos de sí mismo. Por eso, los problemas recursivos también son siempre iterativos. De hecho, cualquier problema resuelto mediante recursividad puede también resolverse con bucles, pero la afirmación contraria no siempre es cierta. Lo que ocurre es que, para

algunos problemas, la solución con bucles suele ser más larga y farragosa. Así que ahí va la primera cosa que debes saber sobre recursividad: no intentes resolverlo todo de forma recursiva. Hay problemas naturalmente recursivos y hay otros que no lo son.

Para resolver un problema mediante recursividad debemos plantear siempre una condición (si ... entonces) dentro de la función o procedimiento recursivo. La condición debe contemplar dos casos:

- el caso general, que realizará

una llamada recursiva para hallar la solución

- el caso base, en el que no es necesario hacer una llamada recursiva porque la solución sea conocida

El caso base es comparable a la condición de salida de un bucle: si no lo escribimos o está mal planteado, obtendremos una recursión infinita, cuyo resultado suele ser un fallo grave (desbordamiento de pila, agotamiento de la memoria, etc) que puede llegar

incluso a colapsar el sistema.

Ejemplo: Un ejemplo clásico es el del cálculo del factorial de un número entero positivo N. Recuerda que el factorial de un número N (se escribe N!) es:

$$N! = N \times (N - 1) \times (N - 2) \times ... \times 2 \times 1$$

Por ejemplo, el factorial de 6 es:

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

Este problema se presta a una solución iterativa, con un bucle que recorra los números desde 6 hasta 1 (o al revés) y

un acumulador que vaya almacenando las multiplicaciones sucesivas. La solución iterativa, planteada como una función, sería:

```
entero función factorial(N es
entero)
variables
   result, i es entero
inicio
   result = 1
   para i desde result hasta 1
dec 1 hacer
   inicio
      result = result * i
   fin
   devolver (result)
fin
```

Ahora bien, este problema también se puede plantear como un problema recursivo mediante inducción, es decir, definiendo el problema en términos de sí mismo:

- Caso general: el factorial de un número N es N multiplicado por el factorial de N – 1. Es decir, para calcular un factorial (el de N) es necesario calcular otro (el de N – 1). El problema se define “en términos de sí

mismo”. Expresado matemáticamente:

$$N! = N \times (N - 1)!$$

- Caso base: el factorial de 1 es 1

Por ejemplo, el factorial de 6 se puede expresar así:

$$6! = 6 \times 5!$$

Como vemos, en la definición de la solución de 6! aparece otra vez el factorial, solo que aplicado a un número menos (5!). A su vez, el factorial de 5 se

puede calcular como:

$$5! = 5 \times 4!$$

Y así sucesivamente:

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

Finalmente, 1! es el caso base de la recursión, y su solución no necesita del cálculo de otro factorial, sino que podemos decir directamente que es 1. Siempre es necesario alcanzar un caso base en el que la solución se defina

directamente para evitar una recursión infinita.

Transcribiendo esta idea a pseudocódigo tendríamos la versión recursiva de la función factorial (su traducción a C o a cualquier otro lenguaje es trivial):

```
entero función factorial(N es
entero)
variables
   result es entero
inicio
   si (N == 1)
entonces                              /
Caso base
      result = 1
```

```
 si_no
Caso general
      result = N * factorial
(N-1)
   devolver (result)
fin
```

Un par de normas para aplicar correctamente la recursividad

La recursividad mal aplicada provocará rápidamente un desbordamiento de pila o algo peor, y el programa dejará de

funcionar. Para asegurarnos de que una solución recursiva está bien planteada debemos preguntarnos si cumple estas dos normas básicas:

- Primera: debe existir al menos una condición de terminación (caso base) para el cual la solución no es recursiva.
- Segunda: cada llamada recursiva se realiza con un dato cada vez más próximo al caso base.

Estas dos normas por sí solas no garantizan que una solución recursiva sea correcta, pero, a la inversa, sí podemos decir que una solución recursiva que no las cumpla será sin duda incorrecta.

Ventajas e inconvenientes de las soluciones recursivas

Entre las ventajas de las soluciones recursivas podemos destacar las

siguientes:

- Hay problemas intrínsecamente recursivos cuya solución iterativa, aún siendo posible, resulta artificial y enrevesada.
- Aún en el caso de que no sea así, la recursividad es una herramienta de enorme potencia cuyas soluciones resultan más sencillas y elegantes que sus equivalentes iterativas. Esto,

que puede parecer un capricho gratuito, es fundamental en ciertos problemas complejos.

- La recursividad es fundamental algunos ámbitos: por ejemplo, los problemas relacionados con la inteligencia artificial y los sistemas expertos suelen tener soluciones recursivas por naturaleza.
- Hay gente a la que le resulta

más fácil pensar de manera recursiva que iterativa.

Los inconvenientes de la recursividad son:

- Las soluciones recursivas suelen ser menos eficientes que sus equivalentes iterativas, es decir, consumen en general más recursos de la máquina (más tiempo de CPU, más memoria, etc)
- La recursión infinita (por error en la aplicación del

caso base) puede causar problemas por desbordamiento de memoria en sistemas poco fiables.

SEXTA PARTE: ALGUNOS ASPECTOS AVANZADOS DEL LENGUAJE C

En esta parte del libro haremos mención a algunas peculiaridades de C que no hemos tratado en las secciones anteriores. Se puede ser un buen

programador de C sin saber gran cosa de lo que se menciona a continuación, pero, tarde o temprano, querrás conocer algunos secretos arcanos del lenguaje. ¿Estás preparado ahora?

Compilación con archivos múltiples

En los grandes proyectos (y en los no tan grandes) es imposible escribir todo el código fuente en un único archivo. Podemos dar, al menos, dos motivos para ello:

- En un archivo fuente de gran tamaño, es incómodo y mareante para el programador encontrar el fragmento de código de busca (es algo así como soltar sobre la mesa de trabajo unos cuantos miles de lápices de colores y tratar de localizar entre ellos nuestro bolígrafo favorito)
- Cualquier cambio en el código fuente, aunque sea

sólo una coma, implica volver a compilar todo el programa, lo cual, en proyectos voluminosos, puede requerir un tiempo no despreciable (del orden de minutos).

Así pues, nuestros proyectos de programación constarán casi siempre de varios archivos fuente... Y pueden llegar incluso a centenares de ellos.

La compilación conjunta de varios archivos para generar un único

ejecutable depende del compilador, y se habla de ella con más detalle en los Apéndices de este libro (creación de proyectos, en el caso de Dev-CPP; y utilización de la herramienta make, en el caso del gcc)

Pero ocurre que, al usar archivos separados, éstos suelen estar estrechamente relacionados, en el sentido de que desde algunos de ellos se invocarán funciones de otros, e incluso se utilizarán variables o constantes simbólicas definidas en otros. Para que

estas referencias cruzadas funcionen correctamente, se hace imprescindible usar archivos de cabecera.

En general, cuando vayamos a crear un proyecto que contenga varios archivos fuente, seguiremos estas normas:

- Para cada archivo fuente crearemos un archivo de cabecera, con el mismo nombre pero con extensión .h, en el que colocaremos todas las declaraciones globales del archivo

(prototipos de funciones, constantes simbólicas, variables globales, etc)

- En cada archivo fuente haremos un #include de su archivo de cabecera.
- En el resto de archivos fuente, si invocan a alguna función residente en otro archivo, haremos un #include del archivo de cabecera de dicho archivo.

Por ejemplo, supongamos que tenemos

un proyecto consistente en tres archivos fuente, que llamaremos fich1.c (donde está la función main()), fich2.c y fich3.c. Supongamos que desde fich1.c se invocan funciones situadas en fich2.c y fich3.c, y que desde fich2.c se invocan funciones de fich1.c y fich2.c. En cambio, desde fich3.c no se invocará ninguna función de fich1.c, pero sí de fich2.c.

En esta situación, deberíamos crear los siguientes archivos (representados esquemáticamente). Obsérvense las

directivas #include de cada archivo fuente:

<table>
<tr><th>fich1.c</th><th>fich1.h</th></tr>
<tr><td><pre>
#include "fich1.h"
#include "fich2.h"
#include "fich3.h"
int main(void)
{
   ...
   funcion1(); //
Está en fich1.c
   ...
   funcion2(); //
Está en fich2.c
   ...
   funcion3(); //
Está en fich3.c
   ...
</pre></td><td><pre>
// Pro
funcio
void f
...otr
protot
// Otr
defini
#defin
#defin
...etc
</pre></td></tr>
</table>

}
void funcion1()
{
...código de la función...
}	

fich2.c	fich2.h

#include "fich2.h"
#include "fich1.h"
#include "fich3.h"
void funcion2()
{
   ...
   funcion1(); // Está en fich1.c
   ...
   funcion2(); // Está en fich2.c
   ...
	// Pro
	funcio
	void f
	... ot
	...
	// Otr
	defini
	#defin
	#defin
	...etc

`funcion3(); // Está en fich3.c`
`...`
`}`
`...código de otras funciones...`	

fich3.c	fich3.h

`#include "fich3.h"`
`#include "fich2.h"`
`void funcion3()`
`{`
`...`
`funcion2(); // Está en fich2.c`
`...`
`funcion3(); // Está en fich3.c`
`...`
`}`
	`// Pro`
	`funcio`
	`void f`
	`... ot`
	`...`
	`// Otr`
	`defini`
	`#defin`
	`#defin`
	`...etc`

```
...código de otras
funciones...
```

Es posible que surjan, al actuar de este modo, problemas de redefinición de funciones, constantes o variables. Para resolverlo, existen al menos tres mecanismos que veremos en los siguientes epígrafes:

- Utilización de las directivas #ifndef... #endif
- Utilización del modificador "extern" en la definición de variables.

- Definición de espacios con nombre.

Creación de librerías

A menudo, nos sorprendemos a nosotros mismos escribiendo una y otra vez funciones similares en distintos programas. Por ejemplo, cuando necesitamos manipular archivos o estructuras de datos dinámicas, operaciones como insertar, buscar, modificar o eliminar suelen dar lugar a funciones muy parecidas. Lo mismo

ocurre con funciones como esperar(), pulsar_tecla(), borrar_pantalla(), menu() y otros muchos ejemplos.

Sería estupendo poder escribir una sola vez esas funciones y reutilizarlas en todos los programas que las necesitasen. Para eso existen las librerías.

Llamamos librería a un archivo que sólo contiene una colección de funciones, pero no un punto de inicio de programa, es decir, no tiene una función main(). En ocasiones se usa el término “biblioteca” como sinónimo de “librería” aunque,

siendo estrictos, una biblioteca es, en realidad, una colección de librerías.

Las librerías, por esa razón, no pueden compilarse hasta obtener un programa ejecutable, pero sí hasta obtener un programa objeto. Más tarde, ese programa objeto puede enlazarse con otro programa que sí tenga función main() y que, además, haga uso de las funciones de la librería.

Las funciones agrupadas en una librería deberían tener algo en común (es decir, una fuerte cohesión entre sí), y no ser

meras aglomeraciones al azar. El mejor ejemplo de esta idea lo proporcionan las librerías estándar. Así, por ejemplo, stdio contiene todas las funciones de entrada/salida; string, las funciones de cadena; y time, las relativas a la fecha y la hora.

Como vimos en la segunda parte del libro, el enlace puede ser estático o dinámico. Nosotros sólo construiremos librerías estáticas, más sencillas. En todo caso, el enlace es reubicable, esto es, las direcciones de memoria del

código y los datos de la librería no son absolutas, sino relativas, de modo que se pueden enlazar con el resto del código independientemente del estado de la memoria principal en cada momento.

El proceso de creación de una librería no puede ser más sencillo:

- Escribir el código fuente y depurarlo. No debe contener ninguna función main()
- Colocar los prototipos de funciones, así como cualquier

otra definición de ámbito global (constantes, macros, etc.) en un archivo de cabecera. Es una buena costumbre que el archivo de cabecera tenga el mismo nombre que el archivo fuente, pero cambiando la terminación de .c a .h

- Compilar el código fuente indicando al compilador que lo que se pretende conseguir es una librería, no un

programa ejecutable. El modo de hacerlo depende, lógicamente, del compilador.

Por ejemplo, con Dev-C++ u otros derivados, basta con indicar, en las propiedades del proyecto, que deseamos crear una librería estática. Esto se hace en el momento de crear el proyecto o bien posteriormente, buscando la opción "Propiedades del proyecto" entre los menús del IDE.

Por otro lado, desde la línea de comandos del compilador (típicamente,

usando el compilador gcc bajo Linux), hay que usar el comando "ar" para crear la librería. Dicho comando es similar a "tar" y sirve para empaquetar varios archivos en uno, que constituirá la librería. A modo de ejemplo, supongamos que tenemos un archivo llamado prueba.c que contiene el código fuente de nuestra librería. Los pasos para crear la librería desde la línea de comandos serían:

1) Crear el código objeto con gcc:

```
> gcc -c -o prueba.o prueba.c
```

2) Crear la librería con ar a partir del código objeto:

```
> ar rcs libprueba.a prueba.o
```

Las opciones del comando "ar" pueden consultarse en las páginas de manual. Las utilizadas en este ejemplo son las más comunes y significan lo siguiente:

- r: reemplazar el fichero destino si ya existía
- c: crear el fichero destino si no existía
- s: constuir un índice del

contenido (hace que el enlace posterior con el programa principal sea más rápido)

Con esto, nuestra librería estará lista para ser enlazada con cualquier programa y reutilizarla tantas veces como sea necesario. Generalmente, el nombre del archivo binario que contiene la librería es (si ésta es estática) libxxx.a, donde "xxx" es el nombre del archivo fuente original. Esta librería debería ser copiada en el directorio de librerías del compilador, y su archivo de

cabecera, en el directorio de archivos de cabecera. Otra opción es copiar ambos archivos en el directorio de trabajo del programa que vaya a hacer uso de la librería.

Por ejemplo, supongamos que tenemos un programa principal (donde se encuentra la función main()) cuyo nombre de archivo es principal.c, y una librería cuyo nombre es mi_libreria.c, que contiene tres funciones llamadas funcion1(), funcion2() y funcion3(). Debe existir un tercer archivo,

mi_libreria.h, que contenga los prototipos de la funciones y cualquier otra declaración necesaria.

Una visión esquemática de estos tres archivos es:

mi_libreria.c	mi_libreria.h
Debemos indicar al compilador que lo compile como librería. Se generará un archivo llamado libmi_libreria.a	No se compila p es un archivo de cabecera (sólo contiene definic no código ejecu

```
#include
"mi_libreria.h"
void funcion1(int
n)
{
...código de la
función...
}
int
funcion2(void)
{
...código de la
función...
}
float
funcion3(int a)
{
...código de la
función...
}
```

```
// Prototip
funciones
void
funcion1(in
int
funcion2(vo
float
funcion3(in
// Otras
definicione
#define ..
#define ..
const int x
...etc...
```

Espacios con nombre

Los espacios con nombre nos ayudan a evitar problemas con identificadores en grandes proyectos. Nos permite, por ejemplo, que existan variables o funciones con el mismo nombre, declaradas en diferentes ficheros fuente, siempre y cuando se declaren en distintos espacios.

La sintaxis para crear un espacio con

nombre es:

```
namespace [<identificador>] {
...
<declaraciones y definiciones>
...
}
```

Por ejemplo:

```
// Inicio del fichero
"puntos.h"
namespace 2D {
   struct Punto {
      int x;
      int y;
   };
}
namespace 3D {
   struct Punto {
      int x;
```

```
        int y;
        int z;
    };
}// Fin de fichero
```

Este ejemplo crea dos versiones diferentes de la estructura Punto, una para puntos en dos dimensiones, y otro para puntos en tres dimensiones.

Para activar un espacio durante la codificación del programa usaremos esta expresión:

```
using namespace
<identificador>;
Por ejemplo:
#include "puntos.h"
using namespace
```

```
2D;              // Activa el
espacio con nombre 2D
Punto p1;                // Define
la variable p1 de tipo
2D::Punto
...
using namespace
3D;              // Activa el
espacio con nombre 3D
Punto p2;                // Define
la variable p2 de tipo
3D::Punto
...
```

Estando un espacio activado, se pueden usar definiciones propias de otro espacio utilizando esta forma de declaración de variables:

```
<nombre_de_espacio>::
```

```
<identificador>;
```

Por ejemplo:

```
#include "puntos.h"
using namespace
2D;              // Activa el
espacio con nombre 2D
Punto p1;              // Define
la variable p1 de tipo
2D::Punto
Punto 3D::p2;              //
Define la variable p2 de tipo
3D::Punto,
        // aunque el espacio
3D no esté activado
```

EL PREPROCESADOR

El preprocesador o precompilador es un

programa que analiza el fichero fuente antes de la compilación real, realizando las sustituciones de macros y procesando las directivas de compilación. El preprocesador también se encarga de eliminar los comentarios y las líneas adicionales que no contienen código compilable.

(En realidad, el preprocesador no es un programa diferente del compilador, sino que el propio compilador hace varias pasadas sobre el código, dedicando la primera de ellas al preproceso)

Una directiva de compilación es una línea cuyo primer carácter es un #. Puesto que actúa antes de la compilación, la directiva puede usarse para modificar determinados aspectos de la misma.

A continuación describimos algunas de las directivas de compilación de C. Observarás que varias las hemos venido utilizando asiduamente sin preguntarnos realmente qué es lo que hacían.

#include

La directiva "#include", como imaginarás visto, sirve para insertar ficheros externos dentro de nuestro código fuente. Estos ficheros son conocidos como ficheros incluidos, ficheros de cabecera o "headers".

Su sintaxis es:

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

Por ejemplo:

```
#include <stdio.h>
#include “conio.h”
```

El preprocesador elimina la línea "#include" y la sustituye por el fichero especificado.

El código fuente en sí no cambia, pero el compilador "ve" el fichero incluido. El emplazamiento del #include puede influir sobre el ámbito y la duración de cualquiera de los identificadores en el interior del fichero incluido.

La diferencia entre escribir el nombre del fichero entre ángulos (<…>) o comillas (“…”) estriba en la manera de buscar los ficheros que se pretenden

incluir. En el primer caso, el preprocesador buscará en los directorios de “include” definidos en la configuración del compilador. En el segundo, se buscará primero en el directorio actual, es decir, en el que se encuentre el fichero fuente y, si no existe en ese directorio, se trabajará como el primer caso.

Si se proporciona una ruta como parte del nombre de fichero, sólo se buscará es el directorio especificado.

#define

La directiva "#define" sirve para definir macros. Esto suministra un sistema para la sustitución de palabras.

Su sintaxis es:

#define identificador_de_macro <secuencia>

Por ejemplo:

```
#define MAX_ELEMENTOS 100
#define LINUX
```

El preprocesador sustituirá cada ocurrencia del identificador_de_macro

(MAX_ELEMENTOS) en el fichero fuente por la secuencia (100). Cada sustitución se conoce como una expansión de la macro. La secuencia es llamada a menudo cuerpo de la macro.

Si la secuencia no existe (como en el ejemplo LINUX), el identificador_de_macro será eliminado cada vez que aparezca en el fichero fuente. Esto tiene una utilidad importante que enseguida examinaremos (ver directiva #ifdef)

Después de cada expansión individual,

se vuelve a examinar el texto expandido a la búsqueda de nuevas macros, que serán expandidas a su vez. Esto permite la posibilidad de hacer macros anidadas. Ahora bien: si la nueva expansión tiene la forma de una directiva de preprocesador, no será reconocida como tal.

Las macros pueden parametrizarse. Por ejemplo:

```
#define CUADRADO(x)    x * x
#define AREA_CIRCULO(r)
3.1416 * CUADRADO(r)
```

Cuando, durante el programa, se utilice

cualquiera de estas dos macros, será necesario especificar un parámetro entre paréntesis. El preprocesador tomará ese parámetro durante la expansión de la macro y lo colocará en el lugar que le corresponda. Por ejemplo:

```
int r, x = 4;
r = AREA_CIRCULO(x);
```

…se expandirá como:

```
r = 3.1416 * CUADRADO(x);
```

…que a su vez se expandirá como:

```
r = 3.1416 * x * x;
```

Y eso será lo que el compilador reciba.

Usar macros para operaciones sencillas (como las de este último ejemplo) en lugar de funciones tiene la ventaja de una mayor velocidad de ejecución.

Por último, mencionemos que existen otras restricciones a la expansión de macros:

- Las ocurrencias de macros dentro de literales, cadenas, constantes alfanuméricas o comentarios no serán expandidas (por ejemplo: printf(“Aquí la macro

MAX_ELEMENTOS no será expandida”);)

- Una macro no será expandida durante su propia expansión, así #define A A, no será expandida indefinidamente.
- No es necesario añadir un punto y coma para terminar una directiva de preprocesador. Cualquier carácter que se encuentre en una secuencia de macro, incluido el punto y coma,

aparecerá en la expansión de la macro. La secuencia termina en el primer retorno de línea encontrado. Las secuencias de espacios o comentarios en la secuencia se expandirán como un único espacio.

#undef

Sirve para eliminar definiciones de macros previamente definidas. La definición de la macro se olvida y el

identificador queda indefinido.

Su sintaxis es:

#undef identificador_de_macro

Por ejemplo:

```
#undef MAX_ELEMENTOS
```

A partir de ese momento, la macro MAX_ELEMENTOS deja de estar definida.

La definición es una propiedad importante de un identificador. Las directivas condicionales #ifdef e #ifndef (ver más abajo) se basan precisamente

en esta propiedad de los identificadores. Esto ofrece un mecanismo muy potente para controlar muchos aspectos de la compilación.

Después de que una macro quede indefinida puede ser definida de nuevo con #define, usando la misma u otra definición.

Si se intenta definir un identificador de macro que ya esté definido, se producirá un aviso (warning) si la definición no es exactamente la misma. Es preferible usar un mecanismo como este para

detectar macros existentes:

```
#ifndef MAX_ELEMENTOS
  #define MAX_ELEMENTOS 100
```

#endif

De este modo, la línea del #define se ignorará si el símbolo MAX_ELEMENTOS ya está definido.

#ifdef e #ifndef

Estas directivas permiten comprobar si un identificador está o no actualmente definido, es decir, si un #define ha sido previamente procesado para el

identificador y si sigue definido.

La sintaxis es:

```
#ifdef identificador
```

#ifndef identificador

Todo lo que se encuentre entre #ifdef y la directiva #endif será procesado por el compilador si y sólo si el identificador está definido. Por ejemplo:

```
#ifdef MAX_ELEMENTOS
int a[MAX_ELEMENTOS];
printf("Se ha definido la
variable a");
#endif
```

Las dos instrucciones se compilarán e

incluirán en el programa ejecutable únicamente si el identificador MAX_ELEMENTOS ha sido previamente definido con la directiva #define. De este modo, se puede manipular al compilador para que compile un conjunto de instrucciones dependiendo de una condición.

La directiva #ifndef actúa del mismo modo, pero al contrario. Es decir, el código comprendido entre #ifndef y #endif se compilará si y sólo si el identificador no está definido.

Estas dos directivas, combinadas con #endif, son fundamentales cuando se escriben programas con múltiples ficheros fuente, o que se pretenden compilar para diferentes entornos usando librerías no estándar.

Por ejemplo, supongamos que tenemos un programa que puede funcionar en Linux y en Windows. Supongamos que, para el modo consola de Linux hemos decidido usar la librería no estándar ncurses, mientras que para Windows nos hemos decantado por conio, que

tampoco es estándar.

Todo el código del programa compilará en ambos sistemas, excepto las funciones de entrada/salida. Pues bien, podemos escribir un programa que compile perfectamente en ambos usando las directivas de este modo:

```
#ifdef LINUX
…llamadas a las funciones de
E/S de la librería ncurses
#endif
#ifdef WINDOWS
…llamadas a las funciones de
E/S de la librería conio
#endif
```

Previamente, habremos hecho un #define LINUX o un #define WINDOWS, dependiendo del sistema para el que estemos compilando. De este modo, sólo con cambiar una línea (la del #define) podemos compilar nuestro programa en cualquiera de los dos sistemas.

#if, #elif, #else y #endif

Estas directivas permiten hacer una compilación condicional de un conjunto de líneas de código.

La sintaxis es:

```
#if expresión-constante-1
<sección-1>
#elif <expresión-constante-2>
<sección-2>
.
.
.
#elif <expresión-constante-n>
<sección-n>
<#else>
<sección-final>
#endif
```

Todas las directivas condicionales deben completarse dentro del mismo fichero. Sólo se compilarán las líneas que estén dentro de las secciones que

cumplan la condición de la expresión constante correspondiente.

Estas directivas funcionan de modo similar a los operadores condicionales de C. Si el resultado de evaluar la expresión-constante-1, que puede ser una macro, es distinto de cero (true), las líneas representadas por sección-1, ya sean líneas de comandos, macros o incluso nada, serán compiladas. En caso contrario, si el resultado de la evaluación de la expresión-constante-1, es cero (false), la sección-1 será

ignorada: no se expandirán macros ni se compilará.

En el caso de ser distinto de cero, después de que la sección-1 sea preprocesada, el control pasa al #endif correspondiente, con lo que termina la secuencia condicional. En el caso de ser cero, el control pasa al siguiente línea #elif, si existe, donde se evaluará la expresión-constante-2. Si el resultado es distinto de cero, se procesará la sección-2, y después el control pasa al correspondiente #endif. Por el contrario,

si el resultado de la expresión-constante-2 es cero, el control pasa al siguiente #elif, y así sucesivamente, hasta que se encuentre un #else o un #endif. El #else, que es opcional, se usa como una condición alternativa para el caso en que todas la condiciones anteriores resulten falsas. El #endif termina la secuencia condicional.

Cada sección procesada puede contener a su vez directivas condicionales, anidadas hasta cualquier nivel. Hay que tener en cuenta, en ese caso, que cada

#if se corresponde con el #endif más cercano.

El objetivo de una red de este tipo es que sólo una sección, aunque se trate de una sección vacía, sea compilada. Las secciones ignoradas sólo son relevantes para evaluar las condiciones anidadas, es decir, para asociar cada #if con su #endif.

Las expresiones constantes deben poder ser evaluadas como valores enteros.

#error

Esta directiva se suele incluir en sentencias condicionales de preprocesador para detectar condiciones no deseadas durante la compilación. En un funcionamiento normal, estas condiciones serán falsas, pero cuando la condición es verdadera, es preferible que el compilador muestre un mensaje de error y detenga la fase de compilación. Para hacer esto se debe introducir esta directiva en una sentencia condicional que detecte el caso no deseado.

Su sintaxis es:

```
#error mensaje_de_error
```

Esta directiva genera el mensaje:

```
Error: nombre_de_fichero
n°_línea : Error directive:
mensaje_de_error
```

TIPOS DE ALMACENAMIENTO

En C existen ciertos modificadores de variables que se usan durante la definición de las mismas y que afectan al modo en que se almacenan las

variables, así como a su ámbito, es decir, a la zona del programa desde donde las variables son accesibles.

Estos son algunos de ellos.

auto

El modificador auto se usa para definir el ámbito temporal de una variable local. Es el modificador por defecto para las variables locales, y se usa muy raramente.

Por ejemplo:

```
void funcion1(void) {
   auto int n;              //
La variable n será local a
esta función
   ...
}
```

register

Indica al compilador que debe intentar que la variable se almacene en un registro de la CPU, cuando sea posible, con el fin de optimizar su tiempo de acceso.

Los datos declarados con el modificador register tienen siempre un ámbito global.

El compilador puede ignorar la petición de almacenamiento en registro, que sólo debe ser usado cuando una variable vaya a ser usada con mucha frecuencia y el rendimiento del programa sea un factor crucial.

Por ejemplo:

```
register int n;              // La variable n se almacenará en un registro de la CPU (o no)
```

static

Este modificador se usa con el fin de que las variables locales de una función

conserven su valor entre distintas llamadas a la misma.

Por ejemplo:

```
void funcion1(void) {
   static int n;  // La
variable n no perderá su valor
entre dos llamadas a la
función
   ...
}
```

extern

Este modificador se usa para indicar que el almacenamiento y valor de una variable o la definición de una función

están definidos en otro fichero fuente. Las funciones declaradas con extern son visibles por todos los ficheros fuente del programa, salvo que se redefina la función como static.

Por ejemplo:

```
extern int n;              // La
variable n está declarada en
otro fichero, pero la vamos
// a usar también en éste
```

const

Cuando se aplica a una variable, indica que su valor no puede ser modificado.

Cualquier intento de hacerlo durante el programa generará un error. Por eso es imprescindible inicializar las variables constantes cuando se declaran.

En C++ es preferible usar este tipo de constantes en lugar de constantes simbólicas (macros definidas con #define). El motivo es que estas constantes tienen un tipo declarado, y el compilador puede encontrar errores por el uso inapropiado de constantes que no podría encontrar si se usan constantes simbólicas. En C es indiferente.

Por ejemplo:

```
const int n = 3;             //
Esta variable es, en realidad,
una constante
```

ARGUMENTOS EN LA LÍNEA DE COMANDOS

La función main(), que normalmente no tiene argumentos, puede llevar dos argumentos especiales llamados argc y argv. La combinación de ambos permiten pasar al programa órdenes desde la línea de comandos (el símbolo de sistema de Windows, una consola de

Linux, etc.)

Veamos la forma de usarlos a través de un ejemplo que más abajo comentaremos:

```
int main(int argc, char
**argv)
{
   if (argc < 2)
      printf("Ha olvidado
escribir su nombre\n");
   else if (argc > 2)
      printf("Error,
demasiados parámetros\n");
   else
      printf("Hola, %s\n",
argv[1]);
   return 0;
}
```

En este sencillo ejemplo observamos la forma en la que siempre se usan los parámetros de main():

- El primer argumento, argc, es de tipo int y contiene el número de parámetros que se han pasado desde la línea de comandos. El propio nombre del programa cuenta como un parámetro, de modo que si deseamos leer un parámetro adicional desde la línea de comandos, el valor de argc

debería ser 2. Observa que ese valor lo pasa el sistema operativo a nuestro programa cuando lo ejecuta. Nuestro programa sólo puede consultarlo y obrar en consecuencia (típicamente, emitiendo un mensaje de error si el número de argumentos es incorrecto, como en el ejemplo)

- El segundo argumento, argv, es un array de cadenas. La

longitud del array es indefinida, y de hecho tiene tantos elementos como parámetros se hayan pasado en la línea de comandos. El primer elemento, argv[0], siempre contiene el nombre del programa, de manera que los verdaderos parámetros se ubicarán a partir de esa posición, es decir, en argv[1], argv[2], etc.

Suponiendo que el programa anterior se

haya compilado y que el ejecutable se denomine "saludo", obtendríamos los siguientes resultados desde la línea de comandos:

```
> saludo
Ha olvidado escribir su nombre
> saludo Juan
Hola, Juan
> saludo Juan Ana
Error, demasiados parámetros
```

MANIPULACIÓN A NIVEL DE BITS

A diferencia de la mayoría de lenguajes de programación, C incorpora métodos

para acceder a los bits individuales dentro de un byte, lo cual puede ser útil por, al menos, tres razones:

- Se pueden almacenar hasta ocho variables lógicas (booleanas) en solo byte, con el consiguiente ahorro de espacio.
- Ciertos dispositivos codifican las información a nivel de bits, de modo que los programas que se comuniquen con esos

dispositivos deben ser capaces de manejar esos bits

- Asimismo, ciertas rutinas de cifrado y descifrado de información necesitan operar a nivel de bits.

Campos de bits

El método que se utiliza en C para operar con bits está basado en las estructuras (structs). Una estructura compuesta por bits se denomina campo de bits. La forma general de definición

de un campo de bits es:

```
struct s_bits
{
   tipo nombre1: longitud;
   tipo nombre2: longitud;
   ...
   tipo nombre3: longitud;
};
```

Cada campo de bits debe declararse como unsigned int y su longitud puede variar entre 1 y 16. Por ejemplo:

```
struct s_bits
{
   unsigned int mostrar: 1;
   unsigned int rojo: 2;
   unsigned int azul: 2;
   unsigned int verde: 2;
```

```
    unsigned int transparencia:
1;
};
```

Esta estructura define cinco variables. Dos de ellas (“mostrar” y “transparencia”) son de 1 bit, mientras que el resto son de 2 bits. Se puede utilizar esta estructura u otra similar para codificar una imagen RGB de 16 colores, especificando para cada píxel su color, si es o no transparente y si debe o no mostrarse.

Para acceder a cada variable del campo de bits se emplea la misma sintaxis que

en cualquier otra estructura de C, pero teniendo en cuenta que un bit sólo puede tomar dos valores, 0 y 1 (en el caso de los campos de dos bits, serían cuatro los valores posibles: 0, 1, 2 y 3). Por ejemplo:

```
struct s_bits pixel;
…
if (pixel.mostrar == 1)
{
   pixel.transparencia = 0;
}
```

Limitaciones de los campos de bits

Las variables de campos de bits tienen ciertas restricciones:

- No se puede aplicar el operador & sobre una de estas variables para averiguar su dirección de memoria.
- No se pueden construir arrays de campos de bits.
- En algunos entornos, los bits se dispondrán de izquierda a derecha y, en otras, de derecha a izquierda, por lo

que la portabilidad puede verse comprometida.

Operadores a nivel de bits

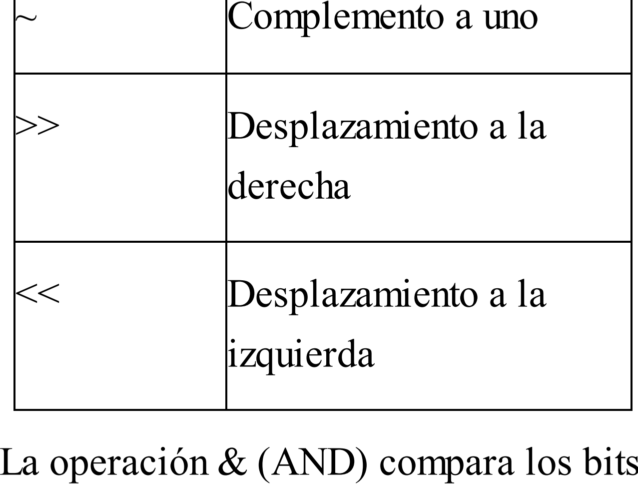
Otro modo de acceder a los bits individuales de un bit sin necesidad de recurrir a los campos de bits es utilizar los operadores específicos de C.

Las operaciones a nivel de bits, directamente heredadas del lenguaje ensamblador, permiten a C comprobar, asignar o desplazar los bits de un byte

con toda libertad, algo que no suele ser posible en otros lenguajes.

Los operadores a nivel de bits de C son:

Operador	Acción
&	Operación AND a nivel de bits
|	Operación OR a nivel de bits
^	Operación XOR (OR exclusivo) a nivel de bits



~	Complemento a uno
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

La operación & (AND) compara los bits uno a uno, dejando a 1 sólo aquéllos que valgan 1 en los dos operandos. Por ejemplo:

```
char c;
c = 255;        // 255 en
binario es 1111 1111
```

```
c = c & 127;   // 128 en
binario es 1000 0000
```

Se realizará la operación AND bit a bit entre 1111 1111 y 1000 0000, resultando el número 1000 0000, es decir, 127. Esto suele ser muy útil para averiguar si un byte tiene determinado bit a 1 o a 0. Por ejemplo, supongamos que tenemos un byte almacenado en la variable c y queremos saber si su bit número 3 (contando desde la derecha, es decir, el tercer bit menos significativo) vale 1 ó 0. Bastaría con hacer la operación AND a nivel de bits entre la variable c y el

número binario 100, es decir, 4 en decimal:

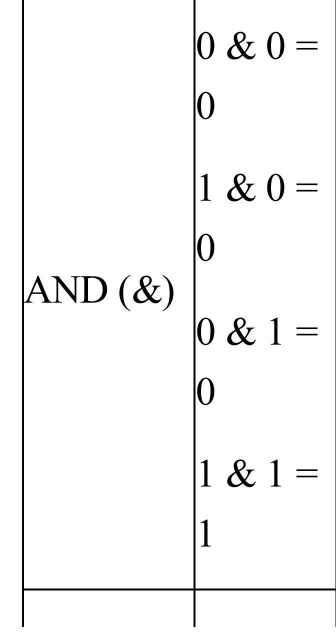
```
c = c & 4;   // 4 en binario
es 100
if (c == 4) printf("El tercer
bit valía 1");
```

Efectivamente, si tras hacer la operación AND el resultado debe ser 4 si el bit buscado valía 1, o 0 si ese bit valía 0.

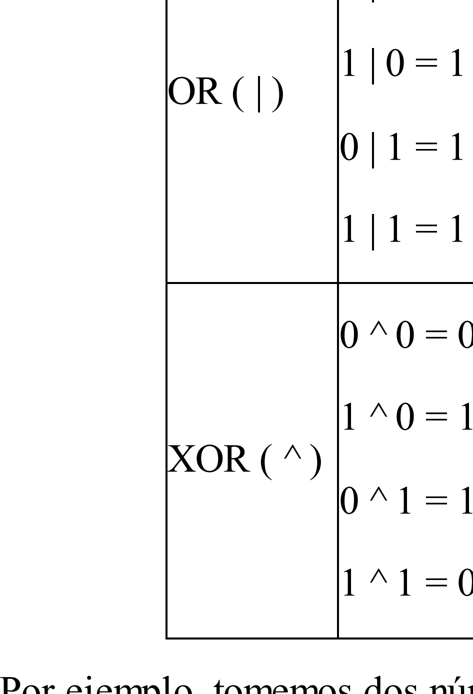
La operación | (OR) es la inversa de AND: pone a 1 todos los bits que valgan 1 en cualquiera de los dos operandos. Mientras que la operación ^ (XOR) pone a 1 los bits que sean diferentes en los

dos operandos.

Podemos resumir el resultado estas tres operaciones en una tabla:



AND (&)	0 & 0 = 0 1 & 0 = 0 0 & 1 = 0 1 & 1 = 1



<table>
<tr><td>OR (|)</td><td>0 | 0 = 0
1 | 0 = 1
0 | 1 = 1
1 | 1 = 1</td></tr>
<tr><td>XOR (^)</td><td>0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
1 ^ 1 = 0</td></tr>
</table>

Por ejemplo, tomemos dos números binarios cualesquiera, como 11110000 y

10101010. El resultado de aplicar las operaciones AND, OR y XOR a los bits uno a uno será:

<table>
<tr><th>AND (&)</th><th>OR (|)</th></tr>
<tr><td>1 1 1 1 0 0 0 0
1 0 1 0 1 0 1 0</td><td>1 1 1 1 0 0 0 0
1 0 1 0 1 0 1 0</td></tr>
<tr><td>1 0 1 0 0 0 0 0</td><td>1 1 1 1 1 0 1 0</td></tr>
</table>

Nótese la diferencia entre los operadores lógicos && y | |, que siempre producen como resultado verdadero o

falso, y los operadores a nivel de bit & y |, que pueden producir cualquier número como resultado del cáculo bit a bit.

Los operadores de desplazamiento (<< y >>) mueven todos los bits de una variable hacia la izquierda o hacia la derecha. La forma habitual de usarlos es:

```
variable >> número de
desplazamientos;
```

Por ejemplo:

```
char c;
c = 128;        // 128 en
```

```
binario es 1000 0000
c = c >> 2;   // Desplazamos
dos bits a la derecha
```

Tras estas instrucciones, c debe valer 0010 0000, es decir, 32.

APÉNDICES

En esta última sección del libro incluimos información útil para el programador pero que no resulta imprescindible, de modo que pueda usarse como manual de consulta rápida cuando sea necesario. Por supuesto, también puedes leerlo si estás recorriendo el libro de principio a fin.

Empezaremos por presentarte las

funciones de uso más frecuente de la biblioteca estándar de ANSI C, y continuaremos por una introducción a dos entornos de compilación de C, uno bajo Windows y otro para GNU/Linux. Terminaremos hablando de dos librerías no estándar, ncurses y SDL. Ncurses permite, en una primera aproximación, añadir colores y un control preciso de la consola (en realidad, mucho más, como manejar capas de texto, pero no profundizaremos tanto). SDL es una potente librería multimedia para añadir gráficos y sonidos a tus programas en C.

Apéndice I: funciones de uso frecuente de ANSI C

La siguiente tabla contiene una lista de algunas funciones de uso frecuente de ANSI C. Para cada función se muestra su prototipo y se explica brevemente cuál es su cometido y cuáles sus datos de entrada y salida. También se indica el archivo de cabecera donde se encuentra el prototipo.

Debe tenerse en cuenta que ANSI C

dispone de muchas otras funciones. Algunas las iremos viendo en los siguientes temas. Por ejemplo, al estudiar las cadenas de caracteres proporcionaremos otra lista de funciones similar a esta, en la que figurarán las funciones de manejo de cadenas.

Pero es que, además, cada compilador de C suele agregar muchas otras funciones que no están en el estándar ANSI y que proporcionan aún más potencia al lenguaje. Ten en cuenta que,

si utilizas funciones no estándar, tu programa perderá portabilidad, es decir, será más difícil compilarlo en otro sistema diferente porque tendrás que modificar el código fuente. En cambio, si tu programa respeta el estándar ANSI, lo podrás hacer funcionar sin cambiar nada (o casi nada) en cualquier otro sistema. Por si esto fuera poco, es posible encontrar en Internet bibliotecas de funciones para casi todo que podemos descargar y utilizar en nuestros programas... con la debida precaución (sobre todo si su procedencia es dudosa;

hay mucho programador chiflado suelto por el mundo)

Para obtener información más extensa sobre estas y otras funciones, el lector puede remitirse a la ayuda en línea de su compilador, a los manuales de referencia existentes en la bibliografía o a las páginas web dedicadas a ello, como c.conclase.net o elrincondelc.com.

Función	Prototipo y cometido
Funciones de entrada/salida	

getchar()	int getchar(void); Devuelve un carácter leido p Es necesario pulsar INTRO.
gets()	char* gets(char* cadena); Lee una cadena de caractere entrada y la sitúa en la posic
printf()	int printf(const char* formato Salida estándar con formato para más detalles.
	int putchar(int carácter);

putchar()	Escribe carácter en la pantal
puts()	int puts(const char* cadena) Escribe cadena en el dispos
scanf()	int scanf(const char* format Entrada estándar con format para más detalles.
Funciones de caracteres	
isalnum()	int isalnum(int carácter); Devuelve 1 si el carácter es

	número), o 0 si no lo es
isalpha()	int isalpha(int carácter); Devuelve 1 si el carácter es mayúscula o minúscula), o (
isdigit()	int isdigit(int carácter); Devuelve 1 si el carácter es
isupper() islower()	int isupper(int carácter); int La primera devuelve 1 si el mayúscula y 0 en otro caso.
Funciones matemáticas	

abs()	int abs(int número); Devuelve el valor absoluto
acos() asin() atan() cos() sin() tan()	double acos(double argumen Todas tienen un prototipo si respectivamente, el arcocos arcotangente, el coseno, el s argumento. Los ángulos se e
	double ceil(double número)

ceil()	Redondea número por exces 4.
exp()	double exp(double potencia) Calcula el exponencial epot
fabs()	double fabs(double número) Devuelve el valor absoluto
floor()	double floor(double número Redondea número por defec 3.

log()	double log(double número); Devuelve el logaritmo natur
log10()	double log10(double númer Devuelve el logaritmo decir
pow()	double log(double base, dou Devuelve la potencia de bas
sqrt()	double sqrt(double número) Devuelve la raiz cuadrada d
Funciones variadas	

atof()	double atof(char* cadena); Convierte la cadena en un nu contiene un número real váli
atoi()	int atoi(char* cadena); Convierte la cadena en un nu contiene un número entero v
atof()	double atof(char* cadena); Convierte la cadena en un nu contiene un número real váli
	char* itoa(int número, int ba

itoa()	Convierte el número en una resultante se determina en ba
rand()	int rand(void); Devuelve un número entero RAND_MAX (RAND_MA) en stdlib.h)
randomize()	void randomize(void); Inicializa el generador de nú necesario invocar esta funci random().
	int random(int máximo);

random()	Devuelve un número al azar

APÉNDICE II: EL COMPILADOR DEV-C++

Herramientas para la programación en C bajo Windows

Existen muchos compiladores de C en entorno Windows, siendo los más populares los de Microsoft (y sus diferentes versiones del compilador

Visual C++) y Embarcadero (antigua Borland). Estos compiladores suelen estar integrados en un IDE (Entorno Integrado de Desarrollo), de manera que bajo el mismo interfaz se puede controlar el editor, el compilador, el enlazador y el depurador, entre otros.

Los compiladores de C de libre distribución (como mingw, djgpp o gcc) suelen ser compiladores independientes, es decir, caracen de IDE. El programador debe encargarse de buscar un editor para escribir su código fuente

y un depurador para corregir errores de ejecución. Esta es la forma clásica de trabajar en entornos Unix/Linux, como luego veremos. Es una forma de trabajo muy recomendable para el aprendiz, porque le permite comprender lo que sucede automáticamente en el IDE cuando pulsa el botón "Run". A veces, las automatizaciones excesivas no son buenas compañeras del aprendizaje.

En la actualidad tienen mucha importancia los IDEs abiertos, multiplataforma y multilenguaje, como

Eclipse o NetBeans. Estos programas no son, hablando con propiedad, compiladores, sino IDEs "vacíos" que, para realizar la compilación, necesitan llamar a un compilador externo. Suelen soportar de forma nativa uno o varios lenguajes (como Java, en el caso de los IDEs mencionados), pero, por medio de plugins, puede añadírsele soporte para muchos otros lenguajes.

Un IDE liviano y muy popular para programar en C bajo Windows es Dev-C++, desarrollado por Bloodshed

Software con licencia GNU y de libre distribución. Se trata de un IDE que trabaja con un compilador de C/C++ llamado mingw (que es una de las versiones para Windows del potentísimo gcc de Linux, muy respetuoso con el estándar ANSI). El IDE incorpora un completo editor de código fuente y un depurador.

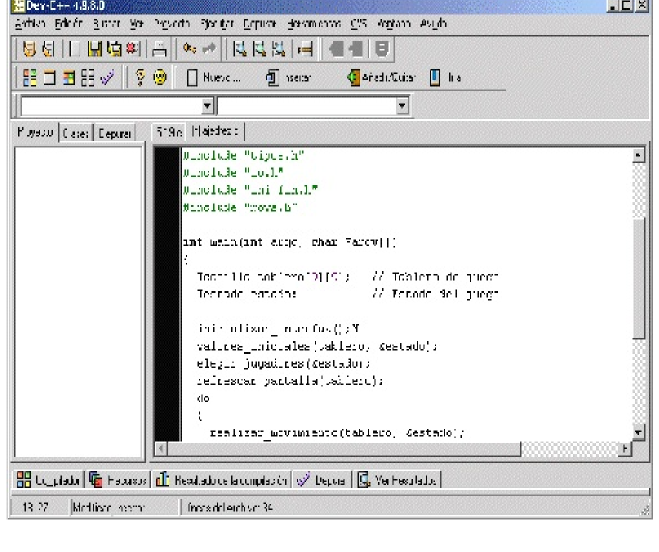
Dev-C++ es muy sencillo de utilizar y de instalar, apenas requiere configuraciones adicionales, ocupa muy poco espacio y compila a gran

velocidad. Todo eso lo hace muy adecuado para principiantes, y también para desarrollar proyectos de tamaño pequeño y mediano. Para programas de más envergadura, probablemente es más adecuado decantarse por otras opciones, como los antes mencionados Eclipse o NetBeans, o compiladores de pago como los de Microsoft o Embarcadero.

A continuación proporcionamos un resumen de las opciones más útiles del IDE de Dev-C++. Todo lo que se explique es fácilmente extensible a otros

IDEs, incluidos los que funcionan bajo Linux.

Dev-C++ puede descargarse gratuitamente de la dirección http://www.bloodshed.net



El IDE de Dev-C++

El Entorno Integrado de Desarrollo (IDE) de Dev-C++ tiene, a primera

vista, es aspecto un editor de texto con una serie de menús adicionales. Efectivamente, el IDE incluye un editor de texto específico para programas en C, pero no se debe olvidar que el editor es sólo una de las aplicaciones integradas dentro del IDE.

Para acceder al compilador y al depurador existe un menú en la parte superior al estilo de cualquier programa habitual. Además, existen otros menús que permiten manejar los programas y proyectos con facilidad y rapidez.

Estos menús son:

- Menú Archivo: Contiene las opciones para abrir y guardar los archivos fuente. Generalmente, los editores de C manejan archivos con las siguientes extensiones:
    - .c: Archivos fuente escritos en C
    - .cpp: Archivos fuente escritos en C++
    - .h: Archivos de cabecera (con prototipos de funciones y

otras definiciones)

  - .hpp: Archivos de cabecera para programas en C++

Desde aquí también se pueden abrir y cerrar proyectos (ver más abajo el Menú "Proyecto")

- Menú Edición: Tiene las opciones típicas para facilitar la edición de programas, incluyendo las utilísimas funciones de Cortar, Copiar y Pegar que seguro que vas a utilizar con

frecuencia. Es muy recomendable que te aprendas los atajos de teclado si aún no los dominas.

- Menú Buscar: Opciones para buscar textos en el programa, reemplazarlos por otros, ir a cierta línea, etc.
- Menú Ver: Tiene opciones para acceder a las distintas ventanas de información del depurador y del compilador.

- Menú Proyecto: Con este menú se pueden manejar aplicaciones distribuidas en varios archivos fuente. A estas aplicaciones se les denomina proyectos. Desde el menú se pueden crear proyectos y agregarles los archivos implicados en el mismo, así como cambiar las propiedades del proyecto.
- Menú Ejecutar: Desde aquí se accede al compilador. La

opción Compilar (Ctrl-F9) produce la compilación y el enlace del código fuente activo en ese momento. Si se producen errores, se muestran en una ventana específica llamada en la parte inferior de la ventana.

- La opción Reconstruir todo recompila todos los archivos que formen parte del proyecto (lo cual puede llevar mucho tiempo si el proyecto es

grande) y los vuelve a enlazar, mientras que la opción Compilar sólo compila el archivo activo y los que tengan dependencias con él

- La opción Compilar y ejecutar (F9) es la más útil y permite ejecutar el programa tras la compilación. Si surgen errores, se muestran (sin ejecutar el programa, obviamente) en la ventana inferior.

- Menú Debug (Depurar): Desde aquí también se accede al depurador, que por su importancia explicaremos en un epígrafe propio más adelante.
- Menú Herramientas: Contiene multitud de opciones de configuración del compilador y del entorno de desarrollo. Las usaremos conforme nos vayan siendo necesarias.

- Menús Ventana y Ayuda: Son similares a los de otras aplicaciones Windows. La mayor crítica que se le puede hacer a este IDE es que el sistema de ayuda en línea es bastante pobre, pero, teniendo un buen manual de referencia de C a mano, o una conexión a Internet, es un detalle de importancia menor.

El Depurador o Debugger

El acceso al depurador desde el IDE es tan sencillo como la invocación del compilador, ya que basta con activar la opción de menú correspondiente, o bien su atajo por teclado. Respecto a esto, debes acostumbrarte a utilizar los atajos de teclado del compilador porque así agilizarás mucho el trabajo con las distintas herramientas.

Manejar el depurador es bastante simple y todas sus opciones están en el menú Depurar. Veamos las opciones más importantes:

- Depurar: inicia la ejecución del programa en "modo de depuración", de manera que se activan el resto de opciones de depuración.
- Puntos de ruptura: Un punto de ruptura es un lugar donde la ejecución debe detenerse para iniciar la ejecución paso a paso. Podemos establecer puntos de ruptura en cualquier lugar del código fuente, y tantos como

queramos. Luego ejecutaremos el programa con normalidad, y éste se detendrá cada vez que encuentre un Punto de Ruptura, pudiendo hacer a partir de ahí la depuración paso a paso.

- Avanzar Paso a Paso: Sirve para ejecutar el programa instrucción a instrucción, comenzando por la primera línea de la función main().

Cada vez se ejecuta una única instrucción, que se va señalando en la pantalla, y luego la ejecución se detiene momentáneamente, permitiéndonos comprobar el estado de la entrada/salida, el contenido de las variables, etc. Pulsando sucesivamente la opción "Siguiente paso", el programa se va ejecutando línea a línea hasta que termine.

- Al llegar a una llamada a una función, podemos optar por introducirnos dentro del código de la misma y ejecutarla también paso a paso (opción "Siguiente Paso"), o bien saltarla para no depurarla (opción "Saltar Paso"). Si optamos por lo primero, cuando la función termina, regresaremos al algoritmo que la invocó y continuaremos la ejecución paso a paso a partir de la

siguiente instrucción.

- Parar: Finaliza la depuración. Esto se puede usar en cualquier momento durante la depuración. Es muy útil si queremos volver a ejecutar el programa con el depurador desde el principio.
- Ir a cursor: A veces, deseamos depurar una parte del programa muy específica y queremos evitar tener que ejecutar paso a paso todo el

programa desde el principio hasta llegar a esa parte. Para esto podemos situar primero el cursor al principio del fragmento que deseamos depurar y luego usar “Ir a cursor”, con lo que conseguiremos que el programa se ejecute hasta la instrucción en la que se encuentre el cursor. Tras esto podemos usar la depuración paso a paso a partir de este punto. “Ir a cursor” puede ser

usado en cualquier momento, incluso aunque la depuración ya haya comenzado.

- Watches (Visualizaciones): Permite mostrar el valor de una variable o una expresión, para así ir comprobando cómo evoluciona tras la ejecución de cada instrucción. Al activar la ventana de Watches, podemos insertar en ella variables y expresiones (o eliminarlas).

También podemos cambiar el valor de una variable en tiempo de ejecución.

¿Y si quiero usar el compilador o el depurador "a mano"?

Hemos dicho ya varias veces que, en nuestra opinión, las automatizaciones que introducen los IDE en los procesos de compilación y enlace son contraproducentes para el aprendiz,

porque no logra comprender cómo funcionan en realidad dichos procesos ni lo que ocurre al pulsar el botón "Run" de su entorno de desarrollo. Eso está muy bien, y puede llegar a ser más productivo, para programadores experimentados, pero el principiante debería aprender antes qué ocurre por debajo.

La buena noticia es que Dev-C++ utiliza el compilador Mingw, una versión para Windows del clásico gcc de Linux. Eso significa que, si hemos instalado Dev-

C++, también hemos instalado gcc en nuestro sistema Windows.

Bastará, por lo tanto, con que salgamos a una consola de símbolo de comandos de Windows y añadamos la ruta hasta el ejecutable de Mingw en el PATH del sistema. La forma exacta de conseguir esto puede diferir según tu versión de Windows, y deberías buscar ayuda si no sabes cómo hacerlo (obviamente, este no es libro sobre Windows).

Luego, en la consola de símbolo de comando, puedes usar exactamente las

mismas instrucciones que usarías bajo un entorno Unix, y que se exponen en el siguiente apartado.

Apéndice III: el compilador de GNU C/C++ (GCC)

Herramientas para la programación en C bajo Linux

Cada día existen más IDEs disponibles para Linux, como Anjuta, K-Developper,

Eclipse, NetBeans, etc. De hecho, gran parte de las distribuciones de Linux actuales incorporan IDEs ya instalados o, al menos, en los repositorios oficiales que no solo sirven para desarrollar programas en C, sino también en otros lenguajes.

Estos IDEs proporcionan un entorno gráfico para manejar el compilador y el depurador sin necesidad de recurrir a la línea de comandos. Su utilización es muy silimar a la de los IDEs para Windows, así que no nos detendremos

en ella. Estudiaremos, en cambio, la forma clásica de proceder en entornos Unix: la compilación desde la línea de comandos. Esta decisión se justifica por dos razones: primera, el alumno/a puede encontrarse aún con muchos sistemas Unix y Linux donde no disponga de ningún IDE; segunda, el IDE lo que hace, en realidad, es realizar llamadas a la herramienta make y al compilador gcc, por lo que es conveniente conocer, aunque sea por encima, la forma de trabajar de ambos comandos.

Todo lo que digamos aquí también puede aplicarse al compilador Mingw para Windows.

Cuando el progamador carece de IDE debe utilizar todas sus herramientas manualmente y por separado. Como es lógico, la herramienta principal es el compilador, pero también necesitaremos, al menos, un editor de texto con el que escribir y modificar el código fuente y un depurador que nos ayude a localizar y corregir errores en tiempo de ejecución.

El compilador gcc

gcc es un compilador rápido, muy flexible y riguroso con el estándar de C ANSI (Mingw es una de sus implementaciones para Windows). Como ejemplo de sus múltiples virtudes, diremos que gcc puede funcionar como compilador cruzado para un gran número de arquitecturas distintas. La forma clásica de utilizar gcc es desde la línea de comandos de un terminal de texto (lo que demuestra que un gran programa no tiene por qué ser un

programa "bonito". Todo depende de a qué tipo de usuario vaya dirigido)

El compilador gcc se encarga de realizar el preproceso del código, la compilación y el enlazado. Dicho de otra manera, nosotros proporcionamos a gcc nuestro código fuente en C, y él nos devuelve un código objeto compilado para nuestro hardware y sistema operativo.

Como curiosidad, mencionar que en realidad gcc no genera código binario alguno, sino código ensamblador. La

fase de ensamblado a código binario la realiza el ensamblador de GNU (gas), y el enlazado de los objetos resultantes, el enlazador de GNU (ld). Este proceso es transparente para el usuario, y a no ser que se lo especifiquemos, gcc realiza el paso desde código en C a un binario ejecutable automáticamente.

Manejo de gcc

Casi siempre, gcc es invocado desde la herramienta make, cuyo funcionamiento se explica más adelante. Pero debemos saber manejar mínimamente gcc para

compilar nuestros programas.

La sintaxis general de gcc en la línea de comandos de Linux es la siguiente:

```
> gcc [opciones] archivo...
```

Vamos a compilar nuestro primer programa con gcc, que, como no podía ser de otra manera, será "hola mundo". Supongamos que el código fuente de "hola mundo" se encuentra almacenado en el archivo holamundo.c. La compilación se realizaría con este comando:

```
> gcc holamundo.c
```

Tras la compilación, aparecerá en el directorio un archivo llamado a.out, que es el archivo ejecutable resultado de la compilación. Si lo ejecutamos el resultado será:

```
> a.out
Hola mundo
```

Errores y warnings

Si el compilador detecta en el código fuente errores en tiempo de compilación, lo comunica al programador del siguiente modo:

```
> gcc holamundo.c
```

```
holamundo.c: In function
'main':
holamundo.c:7: 'a' undeclared
(first use in this function)
holamundo.c:7: (Each
undeclared identifier is
reported only once
holamundo.c:7: for each
function it appears in.)
holamundo.c:7: parse error
before 'return'
```

Como vemos, gcc proporciona el fichero y la línea en la que ha detectado el error. El formato de la salida de error es reconocido por muchos editores, que nos permiten visitar esa posición con atajos de teclado. Obviamente, cuando

gcc detecta algún error de compilación, no se crea archivo ejecutable como resultado.

Las advertencias (warnings) del compilador al analizar el código C, en cambio, no impiden la generación de un ejecutable. Las advertencias se muestran de un modo similar a los errores, indicando el archivo, la línea y la causa de la advertencia.

Opciones comunes

A continuación mostramos algunas de

las opciones más habituales al usar gcc. La colección completa de modificadores se encuentra en su página de manual, man gcc, cuyo manejo se explica un poco más adelante.

Opción	Significado
-help	Indica a gcc que muestre su s
-o <file>	El archivo ejecutable generad este modificador, le especific
-Wall	No omite la detección de nin colección de warnings "poco

-g	Incluye en el binario informa posteriormente.
-O <nivel>	Indica a gcc que utilice optim posibles van desde 0 (no opt Utilizar el optimizador aumer generar ejecutables más rápi cuando generes un ejecutable -g). Las optimizaciones intro
-E	Sólo realiza la fase del prepr enlaza.
-S	Preprocesa y compila, pero r

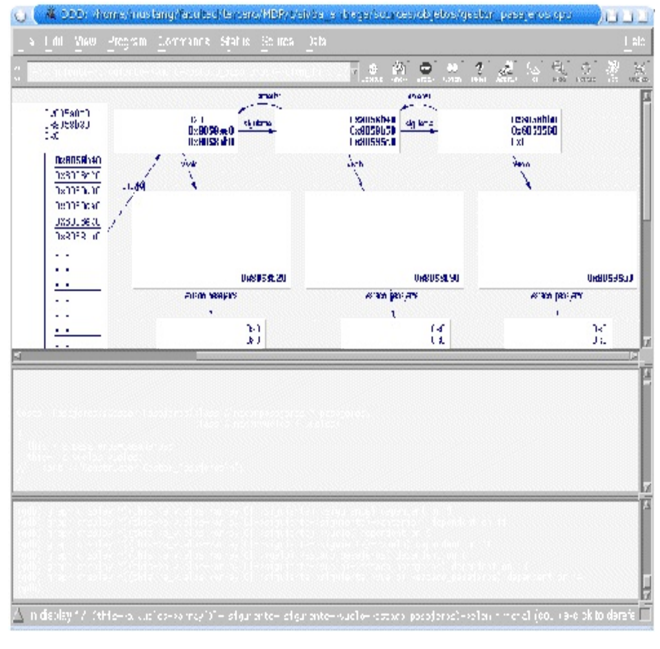
-c	Preprocesa, compila y ensam
-I <dir>	Especifica un directorio adic archivos de cabecera indicad
-L <dir>	Especifica un directorio adic necesarias en el proceso de e
-l<library>	Especifica el nombre de una utilizada en el proceso de enl

Por ejemplo, para compilar el programa "hola mundo" con el nivel máximo de optimización y de manera que el

ejecutable se llame "holamundo", el comando sería:

```
> gcc -O3 -o holamundo
holamundo.c
```

Depuradores: gdb y ddd



Para poder ejecutar un depurador sobre nuestros programas en C, debemos especificar a gcc que incluya

información de depuración en los binarios que genere con la opción –g en la línea de comandos.

gdb es un depurador de línea de comandos, es decir, se ejecuta en una terminar de texto. Tiene todas las funciones típicas de los depuradores (ejecución paso a paso, visualización y modificación de variables, puntos de ruptura, etc), pero es difícil de usar al estar en modo texto.

ddd es un interfaz gráfico para el depurador gdb. La principal ventaja de

ddd es la facilidad para mostrar los contenidos de las variables durante la ejecución de nuestro programa y la posibilidad de ver todo el código fuente del mismo en la ejecución paso a paso.

El manejo de ddd es muy intuitivo y similar a de cualquier otro depurador. En cuanto al depurador gdb, se puede obtener ayuda del mismo en las páginas de manual.

Control de dependencias: la

herramienta make

La mayoría de nuestros proyectos de programación constarán de varios archivos fuente (puede incluso que de centenas de ellos). Cuando modificamos el código fuente de un archivo, luego pasamos a compilarlo (con gcc o cualquier otro compilador). El problema es que puede haber otros archivos que dependan del que acabamos de modificar y que también deben ser recompilados.

La herramienta make nos evita la tarea

de comprobar las dependencias entre ficheros. Para ello se sirve de un fichero (cuyo nombre suele ser Makefile, aunque puede cambiarse) en el que declaramos las dependencias entre ficheros de código fuente y las órdenes necesarias para actualizar cada fichero. Una vez escrito el fichero Makefile, cada vez que cambiemos algún fichero fuente, nos bastará invocar el comando make para que él solito revise todas las dependencias y recompile todos los archivos que sean necesarios.

Para ejecutar la herramienta make basta con escribir en la línea de comandos:

```
> make
```

El fichero makefile

El fichero Makefile más simple está compuesto por "reglas" de este aspecto:

```
objetivo ... : prerrequisitos
...
               comando
               ...
               ...
```

Un objetivo suele ser el nombre de un archivo generado por un programa;

ejemplos de objetivos son los archivos de código objeto. Un objetivo puede ser también el nombre de una acción que debe llevarse a cabo, como "clean", que veremos más adelante en un ejemplo.

Un prerrequisito es un archivo que se usa como entrada para crear un objetivo. Un objetivo con frecuencia depende de varios archivos.

Un comando es una acción que make ejecuta. Una regla puede tener más de un comando, cada uno en su propia línea. Atención: ¡hay que poner un tabulador al

principio de cada línea de comando!

Normalmente un comando está en una regla con prerrequisitos y contribuye a crear un fichero objetivo si alguno de los prerrequisitos cambia. Una regla, por tanto, explica como y cuando reconstruir ciertos archivos que son objetivos de reglas.

A continuación tenemos un ejemplo de un Makefile que describe la manera en la que un fichero ejecutable llamado edit depende de ocho ficheros objeto que a su vez dependen de ocho ficheros de

código fuente C y tres archivos de cabecera. Los ocho ficheros de código fuente C dependen del archivo de cabecera defs.h. Sólo los ficheros de código que definen los comandos de edición dependen de command.h, y sólo los ficheros de operaciones de bajo nivel dependen de buffer.h.

```
edit: main.o kbd.o command.o
display.o insert.o search.o
files.o utils.o
       gcc -o edit main.o
kbd.o command.o display.o
insert.o search.o files.o
utils.o
main.o : main.c defs.h
```

```
        gcc -c main.c
kbd.o : kbd.c defs.h command.h
        gcc -c kbd.c
command.o : command.c defs.h
command.h
        gcc -c command.c
display.c : display.c defs.h
buffer.h
        gcc -c display.c
insert.o : insert.c defs.h
buffer.h
        gcc -c insert.c
search.o : search.c defs.h
buffer.h
        gcc -c search.c
files.o : files.c defs.h
buffer.h command.h
        gcc -c files.c
utils.o : utils.c defs.h
        gcc -c utils.c
clean :
```

```
rm -f edit *.o
```

Para crear el archivo ejecutable edit bastará con escribir en la línea de comandos:

```
> make
```

Para borrar el archivo ejecutable y todos los ficheros objeto del directorio, escribiremos:

```
> make clean
```

En el fichero Makefile del ejemplo, son objetivos el fichero ejecutable edit y los ficheros objeto main.o y kbd.o, entre otros. Son prerrequisitos ficheros como

main.c y defs.h. De hecho, cada fichero .o es tanto objetivo como prerrequisito. Son comandos gcc -c main.c y gcc -c kbd.c

Cuando un objetivo es un archivo, necesita ser recompilado si cualquiera de los prerrequisitos cambia. Además, cualquier prerrequisito que es generado automáticamente debería ser actualizado primero. En el ejemplo, edit depende de cada uno de los ocho ficheros objeto; el archivo objeto main.o depende a su vez del archivo de código fuente main.c y

del archivo de cabecera defs.h, etc.

Un comando de shell sigue a cada línea que contiene un objetivo y prerrequisitos. Estos comandos de shell indican como actualizar el archivo objetivo. Recuerda que hay que poner un tabulador al principio de cada línea de comando para distinguir líneas de comando de otras líneas en el Makefile. La herramienta make no sabe nada sobre cómo funcionan los comandos: depende del programador proporcionar los comandos que actualizarán los

archivos objetivo de manera apropiada.

El objetivo "clean" es especial. No es un fichero, sino el nombre de una acción. Tampoco es un prerrequisito de otra regla ni tiene prerrequisitos. Por tanto, make nunca hará nada con este objetivo a no ser que se le pida específicamente escribiendo make clean en la línea de comandos.

Con esto hemos visto el funcionamiento más esencial de la herramienta make, pero tiene otras muchas funciones.

Páginas de manual

man es el metacomando de Unix/Linux, ya que nos informa sobre el funcionamiento de otros comandos. Pero man también sirve para proporcionar información sobre las funciones de librería y del sistema.

Las páginas de manual se encuentran organizadas en 9 secciones, de las cuales sólo nos interesan en este momento las 3 primeras:

- Programas ejecutables y comandos de la shell

- Llamadas al sistema
- Llamadas a funciones de biblioteca

Antes de comenzar a trabajar con man es recomendable que sepas utilizarlo bien (prueba con el comando man man). Algunas opciones muy útiles son:

Opción	Significado
-a	Muestra de forma consecutiv comando
	Muestra las paginas de manu

-k	lo buscado.

En el momento de buscar información debemos tener en cuenta que algunas funciones se encuentran en varias secciones y, por lo tanto, deberemos indicárselo a man antes de su ejecución. Para especificar la sección sobre la que queremos consultar, lo haremos de la siguiente forma:

```
> man [n° seccion]
[instrucción de C]
```

Por ejemplo, para consultar la página de

manual de la función printf() usaremos este comando:

```
> man 3 printf
```

La buscamos en la sección 3 porque printf() es una función de biblioteca. Sin embargo, es posible que aparezca en otras secciones y podamos así obtener información adicional. Por ejemplo:

```
> man 1 printf
```

Apéndice IV: La librería ncurses, o cómo sacar los

COLORES A NUESTRAS APLICACIONES DE CONSOLA

Qué es Ncurses

Ncurses es una librería de funciones para el manejo de interfaces basadas en texto. Es decir, se trata de un conjunto de funciones, ya programadas, que podemos utilizar en nuestros programas para mejorar su presentación.

Como Ncurses no es una librería

estándar de C, es necesario ordenar al compilador que la enlace con nuestro programa. Esto se hace añadiendo la opción –lncurses al comando gcc. Por ejemplo:

- gcc holamundo.c: compila holamundo.c sin enlazarlo con la librería Ncurses
- gcc –lncurses holamundo.c: compila holamundo.c enlazándolo con Ncurses

Además, debemos hacer un #include <ncurses.h> en el programa que vaya a

utilizar estas funciones.

Ncurses tiene muchísimas funciones, pero nosotros sólo nos referiremos aquí a las más básicas, que nos permitirán añadir color a nuestros textos y controlar libremente la posición del cursor de escritura. Pero Ncurses va mucho más allá, permitiendo la creación de capas de texto superpuestas, menús desplegables y muchas otras cosas en la consola de texto.

Inicialización de

Ncurses

Para utilizar las funciones de Ncurses en nuestro programa, basta con que incluyamos la siguiente llamada:

```
initscr();
```

Esta función crea una ventana de texto. La ventana se llama stdscr (que significa "standard screen", es decir, "pantalla estándar"). A partir de aquí podremos utilizar cualquier función de Ncurses, pues todas actúan sobre esa ventana (se pueden crear varias ventanas sobre stdscr, pero nosotros no

profundizaremos en esa posibilidad).
Por ejemplo, una función que suele ir justo después es:

```
keypad (stdscr, 1);
```

Esto sirve para activar la recepción de teclas especiales (como F1, F2, ESC, etc). Si no llamamos a keypad(), no podremos utilizar ese tipo de teclas en nuestro programa. El segundo parámetro sirve para activar (1) o desactivar (0) la recepción de teclas especiales.

A continuación se enumeran las principales funciones de inicialización

de Ncurses:

initscr()	Inicializa Ncurses y crea la antes que cualquier otra fun `initscr();`
keypad()	Activa / desactiva la recep Intro, etc. Si activar = 1, se desactiva. `keypad(stdscr, activ`

echo()
noecho()
	Activa / desactiva el eco d
	se escriba en el teclado apa
	`echo();`
	`noecho();`

cbreak()
nocbreak()
	Activa / desactiva el envío
	cuando se teclea algo no es
	pulsa "intro". La función cb
	enviado al programa sin ne
	desactiva este comportami
	`cbreak();`
	`nocbreak();`

nodelay()	Activa / desactiva la esper para leer un solo carácter, programa hasta que se puls con el parámetro activar = detenga en getch() aunque desactivarlo, llamaremos a `nodelay(stdscr, act`

endwin()	Finaliza Ncurses. Hay que programa para liberar la m estado inicial.

Escribir y leer

Cuando utilicemos Ncurses debemos olvidarnos de las funciones de entrada/salida estándar, como scanf(), printf(), gets() o puts(). En su lugar usaremos estas otras funciones:

printw()	Para escribir usaremos la f printf() pero sobre una ven

putstr()	putstr(), que es como puts()
getstr() getch()	Para leer disponemos de ge para leer cadenas por tecla número, debemos leerlo co (con las funciones estándar También podemos usar getc
move()	Para colocar el cursor usar la columna x y la fila y de l la fila y luego la columna.
refresh()	Actualiza la pantalla. Es el

	cambios realizados se mue

Colores

Antes de utilizar los colores hay que inicializarlos llamando a la función start_color() sin argumentos, así:

```
if (has_colors())
  start_color();
```

La llamada previa a has_colors() se realiza para asegurarnos de que nuestra consola soporta el uso de colores. Es raro encontrar una consola que no permita colores, pero existen, así que no

está de más hacer la comprobación.

Una vez hecho esto, podemos utilizar los colores básicos definidos en ncurses.h, cuyas constantes son:

```
COLOR_BLACK, COLOR_WHITE,
COLOR_YELLOW, etc.
```

Para utilizar esos colores se deben agrupar en parejas: un color para el texto junto con un color para el fondo. A cada pareja se le asigna un número a través de la función init_pair(), así:

```
init_pair(1, COLOR_YELLOW,
COLOR_BLUE);
```

Esto define a la pareja nº 1 como texto amarillo sobre fondo azul. De este modo podemos definir, por lo general, hasta 64 parejas.

Después, para activar una pareja, haremos esta llamada:

```
attron(COLOR_PAIR(1));
```

Esto activa la pareja de colores nº 1, de manera que todo el texto que aparezca en la pantalla a partir de este momento se verá amarillo con el fondo azul.

La función attron(), además de para activar parejas de colores, sirve para

cambiar otros atributos del texto. Por ejemplo, lo siguiente se utiliza para escribir en negrita:

```
attron(A_BOLD);
```

Puedes obtener más información sobre attron() en las páginas de manual (escribiendo man attron)

Ejemplo de uso de Ncurses

Para terminar esta breve introducción a la librería Ncurses mostraremos un ejemplo ilustrativo del uso de algunas

de las funciones que aquí se han visto.

El siguiente programa utiliza Ncurses para escribir el texto HOLA en color rojo sobre fondo azul y el texto MUNDO en color amarillo sobre fondo verde. El texto HOLA aparece en la línea 11, y MUNDO en la 12. Luego, el programa espera hasta que se pulsa la tecla "flecha arriba", y entonces termina.

```
#include <ncurses.h>
int main(void)
{
  char carácter;
  initscr();           //
Inicializa Ncurses
```

```
  keypad(stdscr, 1); // Activa
teclas especiales (como las
flechas)
  cbreak();          // Para
no tener que pulsar Intro tras
cada carácter
  if (has_colors())
start_color();          //
Inicializa colores
  init_pair(1, COLOR_RED,
COLOR_BLUE);     // Pareja 1 =
Texto rojo, fondo azul
  init_pair(2, COLOR_YELLOW,
COLOR_GREEN); // Pareja 2 =
Texto amarillo, fondo verde
  attron(COLOR_PAIR(1));    //
Activa pareja 1
  move(11, 1);
  printw("HOLA");
  attron(COLOR_PAIR(2));    //
Activa pareja 2
```

```
  move(12, 1);
  printw("MUNDO");
  do
  {
       carácter = getch();  //
Lee un carácter desde el
teclado
  }
  while (carácter != KEY_UP);
  endwin();  // Finaliza
Ncurses
  return 0;
}
```

APÉNDICE V: LA LIBRERÍA SDL, O CÓMO CONSTRUIR

APLICACIONES GRÁFICAS CON C

El siguiente apartado está extraído de mi libro "Ajedrez en C: cómo programar un juego de ajedrez en lenguaje C y que funcione". Allí se hacía una introducción a la librería SDL para dotar de interfaz gráfico al juego que se pretendía desarrollar.

He decidido incluir una adaptación de ese texto en este libro como un apéndice porque la librería SDL es lo

suficientemente potente como para merecer la atención de cualquier interesado en el desarrollo en C.

SDL (iniciales de Single DirectMedia Layer) es una biblioteca libre, con licencia zlib, disponible para múltiples plataformas (entre ellas, Linux y Windows). Puedes bajarte la última versión de http://www.libsdl.org

Esta biblioteca contiene un conjunto muy completo de funciones para manejar gráficos, además de sonidos y distintos dispositivos multimedia (ratón, CD-

ROM, etc). Teniendo en cuenta la complejidad intrínseca a estos dispositivos, la librería es razonablemente sencilla de usar.

Nosotros sólo nos vamos a centrar en la parte de SDL dedicada a los gráficos. Si quieres más información, en la página web reseñada antes encontrarás una completa documentación.

Instalación de SDL

SDL no es una librería C estándar, es decir, no viene "de serie" con el

compilador de C. En realidad, tampoco ncurses lo es, pero su uso está tan extendido en entornos Unix que viene incorporada a las librerías del compilador gcc.

En cambio, la librería SDL debe ser instalada antes de poder utilizarla. A continuación describimos el proceso de instalación en Linux y en Windows

Instalación de SDL en Linux

- Bájate la última versión de la librería de la web de SDL.

Necesitarás el paquete de la librería propiamente dicho (denominado runtime) y el paquete de desarrollo. El paquete runtime tiene un nombre similar a este: SDL-x.x.x-1.i386.rpm, donde "x.x.x" es la versión de la libería e "i386" indica para qué tipo de procesador está compilado. El paquete de desarrollo debe llamarse SDL-devel-x.x.x-i386.rpm o algo similar.

- Instala ambos paquetes en tu sistema. Con el paquete runtime es suficiente para ejecutar programas que usen la librería SDL, pero si además quieres escribir programas nuevos que usen esta librería (y es nuestro caso), también necesitarás el paquete de desarrollo.

Instalación de SDL en Windows

- Bájate la última versión de la librería de la web de SDL.

Necesitarás la librería de vínculos dinámicos (denominada dll) y el paquete de desarrollo. La librería de vínculos dinámicos suele venir comprimida en un archivo cuyo nombre es similar a: SDL-x.x.x-win32.zip, donde "x.x.x" es la versión de la libería. Existirán varios paquetes de desarrollo para varios compiladores. Mi consejo es que bajes el que

está preparado para el compilador de GNU, cuyo nombre es SDL-devel-x.x.x-mingw32.tar o algo similar. También encontrarás paquetes para Visual C++ y otros compiladores.

- Descomprime la librería de vínculos dinámicos. Debes obtener un archivo llamado sdl.dll. Copia este archivo al directorio /windows/system32, o bien

ubícalo en la misma carpeta en la que vaya a estar el programa ejecutable del ajedrez.

- Descomprime el paquete de desarrollo. Encontrarás varios directorios y, dentro de ellos, multitud de archivos. Copia los archivos en los directorios del mismo nombre de tu compilador. Por ejemplo, el copia el directorio "include" del

paquete de desarrollo al directorio "include" de la carpeta donde esté instalado tu compilador. Repite la operación para todos los directorios cuyo nombre coincida.

Compilación y enlace

Al no ser SDL una librería estándar, el enlace entre nuestro programa y las funciones de SDL no se produce

automáticamente. Hay que indicarle al enlazador (o linker) lo que debe hacer.

Compilación y enlace en Linux

Si, por ejemplo, nuestro programa ejecutable se llama "ajedrez" y se construye a partir de 3 programas objeto, llamados "ajedrez.o", "movs.o" e "interfaz.o", debemos modificar la primera parte de nuestro Makefile de este modo:

```
ajedrez: ajedrez.o movs.o
interfaz.o
        gcc -g `sdl-config --
cflags` -o ajedrez ajedrez.o
```

```
movs.o interfaz.o `sdl-config
--libs`
```

Fíjate bien en que las comillas son en realidad acentos graves, es decir, invertidos e inclinados hacia atrás. Debes respetar la sintaxis para que funcione.

Eso es todo lo que tienes que hacer para compilar son SDL. Si te interesa saber POR QUÉ, sigue leyendo. Si no, puedes pasar al siguiente apartado.

En realidad, lo que hay escrito entre esas comillas invertidas son comandos

de SDL que indican la configuración de la librería. Estos comandos los puedes ejecutar desde la consola, obteniendo más o menos esto:

```
$ sdl-config --cflags
-I/usr/local/include -
I/usr/local/include/SDL -
D_REENTRANT
$ sdl-config –libs
-L/usr/local/lib -lSDL -
lpthread
```

Al añadir estos comandos dentro del Makefile, enmarcados entre esas comillas invertidas, obligamos a la herramienta make a ejecutar los

comandos y a sustituir el texto entrecomillado por el resultado del comando. Es decir, sería como si hubiéramos puesto esto en el Makefile:

```
ajedrez: ajedrez.o movs.o
interfaz.o
        gcc -g -
I/usr/local/include -
I/usr/local/include/SDL -
D_REENTRANT -o ajedrez
ajedrez.o movs.o interfaz.o -
L/usr/local/lib -lSDL -
lpthread
```

Pero preferiremos la primera forma porque es más corta y, además, funcionará en todas las situaciones,

mientras que esta segunda depende de dónde y cómo se haya instalado la librería SDL (fíjate que hace referencia a directorios concretos de nuestro sistema)

Compilación y enlace en Windows

Lo siguiente sirve para compilar y enlazar con SDL desde el compilador Dev-C++, que tiene licencia GNU y es gratuito. Con otros compiladores el proceso debe ser similar, aunque es posible que necesites bajar otro paquete de desarrollo adaptado al compilador

concreto.

Para poder compilar y enlazar la libería SDL tienes que abrir las opciones del proyecto (menú "Proyecto") y activar la pestaña "Parámetros". En el cuadro con el título "Linker" escribe lo siguiente:

```
-lmingw32 -lSDLmain –lSDL
```

Si has instalado correctamente la librería SDL, con esto debería bastar. Recuerda que el archivo sdl.dll debe estar en la misma carpeta que el programa ejecutable (o, si no, instalado con las liberías del sistema de

Windows)

Inicialización y terminación de la pantalla gráfica

Una vez instalada la libería y preparado el compilador, podemos usar las funciones de SDL como cualquier otra función estándar de C. Su uso es exactamente igual en Windows y en Linux, por lo que el programa que obtendremos debería compilar sin necesidad de hacerle ningún cambio en

ambos sistemas.

Para usar los gráficos, hay que hacer un #include <SDL/SDL.h> en el archivo fuente, como es natural. Aparece dos veces el nombre "SDL" porque el archivo SDL.h está dentro de una carpeta llamada SDL.

Lo siguiente que hay que hacer es inicializar la pantalla gráfica. Para eso disponemos de dos funciones: SDL_Init() y SDL_SetVideoMode():

SDL_Init(). Debe ser la primera función en invocarse. No se puede usar ninguna

otra función de SDL si antes no se ha llamado a esta. Hay que pasarle un parámetro que indica qué tipo de sistema multimedia queremos manejar (la tarjeta de vídeo, la de sonido, el CD-ROM, etc). En nuestro caso será la tarjeta de vídeo, ya que sólo nos interesa manipular gráficos. La constante para ello es SDL_INIT_VIDEO:

```
SDL_Init(SDL_INIT_VIDEO);
```

La fución SDL_Init() devuelve –1 si ocurre algún error al iniciar el sistema de gráficos. En ese caso, el programa no

podrá continuar, de modo que debemos comprobar el valor devuelto por SDL_Init().

SDL_SetVideoMode(). Esta debe ser la segunda función en invocarse, justo a continuación de SDL_Init(). Sirve para establecer el tipo de pantalla gráfica que queremos. Hay que indicarle el tamaño en píxels, el número de bits de color y los atributos de la pantalla. Por ejemplo:

```
SDL_SetVideoMode(800, 600, 16,
SDL_ANYFORMAT |
SDL_DOUBLEBUFFER);
```

Esto crea una ventana gráfica de

800x600 píxels, con 16 bits de profundidad de color. El último parámetro, SDL_ANYFORMAT, es una constante que indica a SDL que puede seleccionar otra profundidad de color si la elegida no está disponible. Este cuarto parámetro puede tomar otros muchos valores que no vamos a ver, pero sí señalaremos que es conveniente añadir la constante SDL_DOUBLEBUFFER por motivos de rendimiento (ver ejemplo más abajo).

SDL_SetVideoMode() devuelve un

puntero a una estructura llamada SDL_Surface, definida en SDL.h, o NULL si ocurre algún error. Este puntero nos será imprescidible para manejar la pantalla gráfica, así que debes guardarlo en una variable. Esta variable, además, debe ser global si se va a usar en otras partes del programa, contraviniendo una de las buenas prácticas de programación más universales que existen. Sin embargo, si no lo haces así, la variable no funcionará correctamente.

Puedes imaginar que el puntero a

SDL_Surface es como el puntero a FILE que devuelve la función fopen(). Sin ese puntero a FILE no se puede manejar el archivo. Pues bien, sin el puntero a SDL_Surface no podemos manejar la pantalla gráfica. Visto así, la función SDL_SetVideoMode() es parecida a fopen(), solo que aplicada a los gráficos en lugar de a los archivos.

Aquí tienes un ejemplo de inicialización de la pantalla gráfica:

```
#include <SDL/SDL.h>
...
SDL_Surface
```

```
*pantalla;              // Esta
variable debe ser GLOBAL
  ...
  if (SDL_Init(SDL_INIT_VIDEO)
== -1) {
    puts("Error en la
inicialización del sistema de
vídeo\n");
    SDL_Quit();
    exit(-1);
  }
  pantalla =
SDL_SetVideoMode(800, 600, 16,
SDL_ANYFORMAT|SDL_DOUBLEBUF);
  if (pantalla == NULL) {
    puts("Fallo al establecer
el modo de vídeo\n");
    SDL_Quit();
    exit(-1);
  }
  ...
```

```
 SDL_Quit();
Esto se hace al final del
programa
```

Tan importante como inicializar la pantalla gráfica es finalizarla. Ten en cuenta que la pantalla gráfica consume muchos recursos, y éstos deben ser liberados antes de que el programa termine su ejecución. Para eso tenemos la función SDL_Quit(), que se invoca sin argumentos (observa el ejemplo)

Dibujar gráficos en la pantalla

Ya tenemos nuestra pantalla gráfica inicializada y lista para empezar a dibujar en ella. Pero, ¿qué tipo de objetos se pueden dibujar?

Aunque las librerías gráficas permiten al programador pintar píxels individuales en cualquier punto de la pantalla, lo habitual es trabajar con imágenes previamente existentes llamadas sprites. Un sprite es una imagen guardada en un archivo que puede ser cargada por el programa y mostrada en cualquier parte de la pantalla gráfica y tantas veces

como sea necesario.

Por lo tanto, lo primero que necesitas es hacerte con una colección de sprites para tu programa. Si, por ejemplo, suponemos que estamos desarrollando un de ajedrez, necesitaríamos los siguientes:

- Una imagen del tablero.
- Una imagen de cada una de las piezas.
- Opcionalmente, una imagen de fondo para decorar la

pantalla.

Los archivos con las imágenes deben estar en formato BMP. SDL admite otros formatos, pero el BMP es con diferencia el más fácil de manipular, así que es una muy buena idea empezar por las imágenes BMP y luego, cuando ya las manejes bien, dar el salto a otros formatos con compresión.

Para dibujar una imagen en cualquier punto de la pantalla, hay que hacer dos cosas que pasamos a describir con detalle:

- Cargar la imagen en la memoria (procedente de un archivo BMP)
- Mostrar la imagen en la pantalla

Cargar imágenes en la memoria

Sólo es necesario cargar las imágenes una vez. Normalmente, se hará al principio del programa, justo después de la inicialización de SDL. Una vez cargadas en la memoria, podremos utilizarlas tantas veces como las

necesitemos, a menos que liberemos el espacio de memoria que ocupan. La liberación de espacio, por tanto, debería hacerse al final del programa, justo antes de terminar.

Para cargar una imagen BMP se usa la función SDL_LoadBMP(), de esta forma:

```
   SDL_Surface *tablero;
   tablero =
 SDL_LoadBMP("tablero.bmp");
   if (fondo == NULL) {
      printf("Error al cargar
 el archivo tablero.bmp");
      SDL_Quit();
      exit(-1);
```

```
}
```

Observa que SDL_LoadBMP() devuelve un puntero a SDL_Surface. Este puntero será necesario para luego mostrar la imagen en cualquier lugar de la pantalla. La variable "fondo" debe ser global si se va a usar en más de una función (si es local y la pasamos como parámetro a otra función, SDL fallará).

Las imágenes son rectangulares. En muchas ocasiones, necesitamos mostrar una imagen encima de otra. Es el caso de las piezas, que se mostrarán encima

del tablero. Cuando esto ocurre, el color de fondo de la pieza (que decidimos que fuera negro) aparecerá encima del tablero como un desagradable recuadro de color negro. En estas situaciones, hay que avisar a SDL de que, para este sprite en concreto, el color negro va a ser transparente, es decir, no debe ser mostrado. Esto se hace así:

```
SDL_Surface *peon_blanco;
Uint32 color;              //
Para definir el color de
transparencia (donde proceda)
// Cargamos la imagen del
peón blanco
peon_blanco =
```

```
SDL_LoadBMP("peon_bl.bmp");
  if (peon_blanco == NULL) {
     printf("Error al cargar
el archivo peon_bl.bmp");
     SDL_Quit();
     exit(-1);
  }
  // Definimos la
transparencia (color negro =
(0,0,0) )
  color =
SDL_MapRGB(peon_blanco-
>format, 0, 0, 0);
  SDL_SetColorKey(cuadro1,
SDL_SRCCOLORKEY |
SDL_RLEACCEL, color);
```

Las imágenes cargadas en memoria deben ser liberadas antes de finalizar el programa con una llamada a

SDL_FreeSurface(). Por ejemplo, para liberar la memoria ocupada por la imagen "tablero.bmp" que hemos cargado antes usaremos el puntero que obtuvimos al cargarla, así:

```
SDL_FreeSurface(tablero);
```

Mostrar imágenes en la pantalla

Una vez cargada una imagen BMP en la memoria, podemos mostrarla en la pantalla a través del puntero SDL_Surface que obtuvimos al cargarla. Una imagen cargada puede ser mostrada todas las veces que queramos en

cualquier posición de la pantalla.

Por ejemplo, para mostrar la imagen del tablero (que cargamos en un ejemplo del apartado anterior) haríamos lo siguiente (luego comentamos el código)

```
    SDL_Rect rect;
    rect = (SDL_Rect) {10, 10,
  400, 400};
    SDL_BlitSurface(tablero,
  NULL, pantalla, &rect);
    SDL_Flip(pantalla);
```

La variable "rect" es de tipo SDL_Rect, y define un área rectangular de la pantalla. El área rectangular empieza en las coordenadas (10, 10) (esquina

superior izquierda de la pantalla) y mide 400 píxels de ancho y 400 de alto, es decir, termina en (410, 410)

SDL_BlitSurface() es la función que se encarga de mostrar en la pantalla un sprite. La variable "tablero" es de tipo SDL_Surface*, y debe ser la que nos devolvió SDL_LoadBMP() al cargar la imagen del tablero. La variable "pantalla" también es una SDL_Surface*, y debe ser la que nos devolvió SDL_SetVideoMode() al inicializar la pantalla gráfica. Ya

dijimos que los punteros que nos devuelven estas funciones son imprescidibles y que debíamos definirlos como variables globales. La variable "rect" es el área rectangular que acabamos de definir.

Fíjate que "rect" es la que indica en qué lugar de la pantalla va a aparecer el sprite. En este ejemplo, aparecerá en (10,10). Se le han reservado 400x400 píxels para dibujarse, es decir, hasta la posición (410, 410). Si el sprite en más pequeño, no pasará nada (ocupará lo

que mida realmente). Si es más grande, se truncará.

Por último, SDL_Flip() hace que lo que acabamos de dibujar se muestre realmente en la pantalla. Su efecto es parecido al de la función refresh() de ncurses. En realidad, todo lo que dibujamos se escribe en una zona de memoria específica y, al hacer SDL_Flip(), esa zona de memoria se vuelca sobre la memoria de vídeo, apareciendo todo en la pantalla. Esto representa el movimiento de gran

cantidad de información entre distintas zonas de memoria, lo cual es un proceso relativamente lento. Por eso, si vamos a dibujar varios sprites consecutivos, es mejor hacer una sola vez SDL_Flip(), al final, cuando los hayamos dibujado todos. Llamar a SDL_Flip() después de dibujar cada sprite ralentizará notablemente el funcionamiento de nuestro programa.

Control del teclado

Para leer el teclado en una ventana

gráfica creada con SDL no se pueden usar las funciones estándar (como getchar() o gets()), ni mucho menos las de ncurses (como getstr()). SDL solo permite leer los caracteres de uno en uno, y no muestra eco por la pantalla (si queremos eco, tenemos que mostrar los caracteres nosotros mismos después de leerlos)

Por lo demás, la forma de capturar un carácter tecleado es similar a la de ncurses, solo que un poco más complicada. A continuación se muestra

un código de ejemplo:

```
SDL_Event evento;
                 // Para leer
el teclado
// Leer teclado
if (SDL_PollEvent(&evento))
// Comprobar si se ha pulsado
una tecla
{
   if (evento.type ==
SDL_KEYDOWN)     //
Efectivamente, se ha pulsado
una tecla
   {
       switch
(evento.key.keysym.sym)  //
Vamos a mirar qué ecla es
       {
           case SDLK_UP:
...acciones...;
```

```
break;              // Flecha
arriba
            case SDLK_DOWN:
...acciones...;
break;              // Flecha
abajo
            case SDLK_LEFT:
...acciones...;
break;              // Felcha
izquierda
            case SDLK_RIGHT:
...acciones...;
break;              // Flecha
derecha
            case SDLK_RETURN:
...acciones...;
break;              // Intro
            case SDLK_ESCAPE:
...acciones...;
break;              // ESC
            case SDLK_m:
```

```
  ...acciones...;
  break;              // Tecla "m"
  (menú)
          }
      }
  }
```

Existen constantes para cualquiera de las otras teclas del teclado. Todas empiezan por "SDLK_". Por ejemplo, la tecla "a" tendrá el código "SDLK_a".

Definición de colores

Aunque en general trataremos con imágenes ya creadas (como la del

tablero o las de las piezas), es posible que necesites definir algún color para usarlo directamente sobre la pantalla gráfica (por ejemplo, para usar transparencias o para escribir un texto)

En SDL no hay colores predefinidos, como en ncurses. Los colores debemos definirlos nosotros mezclando los colores básicos RGB (rojo, verde y azul)

Hay dos formas de definir un color: con una variable de tipo “SDL_Color” o con una variable de tipo “Uint32”. El uso de

una u otra dependerá de para qué queramos usar ese color:

a) Con una variable de tipo SDL_Color. Se usaría así:

```
SDL_Color color;
color = (SDL_Color) {50, 150,
200, 255};
```

Los cuatro números definen el color. Deben ser números comprendidos entre 0 y 255. El primero es el nivel de rojo (R), el segundo el nivel de verde (G) y el tercero, el nivel de azul (B). El cuarto número es el brillo. El color definido en este ejemplo tiene mucho azul, bastante

verde y poco rojo. El resultado debe ser un azul amarillento.

b) Con una variable de tipo Uint32, que se usaría así:

```
Uint32 color;
color = SDL_MapRGB(pantalla-
>format, 50, 150, 200);
```

En esta ocasión, "pantalla" debe ser un puntero a una imagen SDL_Surface que hayamos cargado previamente. Los tres valores siguientes son los niveles RGB. No hay nivel de brillo, porque éste se toma de la imagen apuntada por "pantalla".

De las dos maneras se pueden definir colores para usarlos posteriormente. Si el color lo necesitamos para una transparencia, recurriremos al segundo método (de hecho, ya vimos un ejemplo de ello al estudiar cómo se cargaban y mostaban las imágenes en SDL; allí usamos el color negro como transparencia). Si el color lo necesitamos para escribir un texto en la pantalla gráfica, usaremos el primer método (como se podrá ver en el siguiente apartado)

Mostrar texto en la pantalla gráfica: la librería SDL_TTF

La librería SDL no permite directamente la escritura de texto en la pantalla gráfica. Esto se debe a que la pantalla gráfica, por definición, no admite caracteres, sino únicamente imágenes.

Por fortuna, a la sombra de SDL se han creado multitud de librerías adicionales que, partiendo de SDL, complementan y mejoran sus prestaciones. Una de ellas es SDL_TTF.

La libería SDL_TTF permite cargar fuentes true type que estén guardadas en archivos ".ttf" y manejarlas como si fueran imágenes BMP en la pantalla gráfica generada por SDL. Necesitamos SDL_TTF, por lo tanto, para escribir los mensajes de usuario y las opciones del menú.

Instalación, compilación y enlace de SDL_TTF

La instalación de la librería SDL_TTF es similar a la de SDL, tanto en Linux como en Windows, de modo que puedes

remitirte al apartado correspondiente para recordar cómo se hacía.

En cuanto a la compilación y enlace, sólo tienes que añadir la opción "-lSDL_ttf" a la línea de compilación del Makefile:

```
gcc -g `opciones de SDL` -o
ajedrez ajedrez.o movs.o...
`más opciones de SDL` -
lSDL_ttf
```

Si estamos compilando en Windows con Dev-C++, agregaremos "-lSDL_ttf" a Opciones del Proyecto / Parámetros / Linker.

Inicialización de SDL_TTF

Igual que SDL, la librería SDL_TTF necesita ser inicializada antes de usarla, y finalizada antes de terminar el programa para liberar los recursos adquiridos.

Como SDL_TTF corre por debajo de SDL, debe ser inicializada después de SDL, y debe ser terminada antes que SDL.

La inicialización de SDL_TTF se hace simplemente así:

```
    if(TTF_Init() == -1) {
```

```
    printf("Fallo al
inicializar SDL_TTF");
    exit(-1);
  }
```

Inmediatamente después podemos cargar una fuente true type de un archivo TTF, así:

```
  TTF_Font* fuente;
  ....
  fuente =
TTF_OpenFont("arial.ttf", 14);
  if(fuente == NULL) {
    printf("Fallo al abrir la
fuente");
    exit(-1);
  }
  TTF_SetFontStyle(fuente,
TTF_STYLE_BOLD);
```

La variable "fuente" es un puntero a TTF_Font. Debe ser una variable global por el mismo motivo que las variables SDL_Surface*. La función TTF_OpenFont() abre el archivo "arial.ttf" y carga el tipo de letra Arial en tamaño 14 para su uso en el programa. Después es conveniente comprobar que el puntero "fuente" contenga un valor válido y no NULL.

Por último, la función TTF_SetFontStyle() puede usarse para determinar el estilo de la fuente.

Tenemos varias posibilidades: TTF_STYLE_BOLD (negrita), TTF_STYLE_ITALIC (cursiva), TTF_STYLE_UNDERLINE (subrayado) y TTF_STYLE_NORMAL. Si queremos combinar varios estilos, podemos separarlos por el operador "|". Por ejemplo, para poner la fuente en negrita y cursiva escribiríamos esto:

```
    TTF_SetFontStyle(fuente,
  TTF_STYLE_BOLD |
  TTF_STYLE_ITALIC);
```

Finalización de SDL_TTF

El proceso de finalización es inverso y complementario al de inicialización. Primero habrá que liberar todas las fuentes cargadas durante la inicialización, y luego hay que terminar el subsistema SDL_TTF.

Para liberar una fuente escribiremos sencillamente:

```
TTF_CloseFont(fuente);
```

La variable "fuente" será de tipo TTF_Font*, y debe coincidir con la que nos devolvió la función TTF_OpenFont(). Esta operación la

repetiremos con cada una de las fuentes que hayamos cargado.

Después finalizaremos SDL_TTF escribiendo:

```
TTF_Quit();
```

Recuerda que esto debe hacerse ANTES de SDL_Quit(), ya que SDL_TTF depende de SDL.

Escribir texto con SDL_TTF

Todo esto lo hacemos con un objetivo: poder escribir texto en la pantalla gráfica y sustituir así todas las funciones

printw() y similares.

Para escribir un texto hay que hacer dos cosas: primero, convertirlo en una imagen; segundo, mostrar la imagen en la pantalla.

La conversión de un texto en una imagen se hace con la función TTF_Render():

```
   SDL_Color color;
   SDL_Surface* txt_img;
   color = (SDL_Color)
{255,100,100,255};
   txt_img =
TTF_RenderText_Blended(fuente,
"Hola mundo", color);
   if(txt_img == NULL) {
         printf("Fallo al
```

```
renderizar el texto");
            exit(-1);
    }
```

Como ves, hay que hacer bastantes cosas para mostrar un texto en la pantalla gráfica, pero todo es acostumbrarse. Primero, hay que definir un color para el texto (cómo se definen los colores es algo que vimos en el epígrafe anterior). En este caso, hemos escogido un rojo brillante.

Después se invoca a TTF_RenderText(), pasándole como parámetros el puntero a la fuente que obtuvimos con

TTF_OpenFont(), el texto que queremos mostrar y el color. La función nos devuelve un puntero de tipo SDL_Surface* que, si recuerdas, es exactamente el mismo que usábamos con las imágenes cargadas desde un archivo BMP.

En realidad, la función TTF_RenderText() tiene tres formas:

- TTF_RenderText_Solid(): realiza una conversión del texto en imagen rápida pero de poca calidad.

- TTF_RenderText_Shaded(): la imagen resultante es de gran calidad pero tiene un recuadro negro alrededor
- TTF_RenderText_Blended(): la imagen resultante es de gran calidad y sin recuadro negro

En general preferiremos el modo "Blended", que es el que proporciona mejores resultados. El modo "Shaded" se puede usar en determinados lugares (si no hay otra imagen debajo del texto).

El modo "Solid" sólo debe usarse si hay que mostrar mucho texto y el modo "Blended" se revela demasiado lento.

Hasta aquí, sólo hemos convertido el texto "Hola mundo" en una imagen, pero aún no la hemos mostrado en la pantalla. Para hacerlo procederemos como con cualquier otra imagen:

```
  // Mostramos el texto como
si fuera una imagen
  rect = (SDL_Rect) { 500,
280, 100, 30 };
  SDL_BlitSurface(txt_img,
NULL, pantalla,
&rect);
  SDL_Flip(scr);
```

Se supone que "rect" es de tipo SDL_Rect y que pantalla es el puntero a SDL_Surface* que nos devolvió SDL_SetVideoMode() al inicializar SDL. Así, el texto "Hola mundo" se mostrará en la posición (500, 280) de la pantalla gráfica, reservándose para él 100 píxels de ancho y 30 de alto.

Una última cosa…

El manual de lenguaje C termina aquí. Si tienes un minuto, te pedimos que vayas a la ficha del libro en Amazon y dejes una opinión honesta. Las opiniones de los lectores son importantísimas para la visibilidad del libro. Como siempre decimos, no nos vamos a hacer millonarios así, pero nos ayudará a seguir desarrollando contenido de interés.

Si quieres recibir noticias sobre nuevas

publicaciones y ofertas especiales, puedes dejarnos tu correo electrónico en http://ensegundapersona.es. No recabamos ningún tipo de información personal. Prometemos no ser muy pesados y, en cualquier caso, podrás darte de baja cuando quieras.

SERIE: PROGRAMACIÓN

AJEDREZ EN C

Cómo programar un juego de ajedrez en lenguaje C

¡y que funcione!

Del diseño en papel
a la inteligencia artificial:
una planificación en 10 fases.

A. M. Vozmediano

ensegundapersona.es

Ajedrez en C: Cómo programar un juego de ajedrez en lenguaje C…

¡y que funcione!

Del diseño en papel a la inteligencia artificial: una planificación en 10 fases.

¿Has deseado alguna vez programar un juego de ajedrez? ¿Te gustaría jugar una partida contra una inteligencia artificial programada por ti mismo? No te vamos a engañar: no se trata de un problema trivial, pero con esta guía puedes conseguirlo.

El ajedrez es un juego hasta cierto punto fácil de transformar en programa de ordenador, ya que sus reglas están muy bien definidas, pero empieza a volverse complicado si queremos dotarlo de un interfaz gráfico y de inteligencia suficiente como para poder echarnos unas partidas contra nuestro propio juego.

En este libro se propone una planificación en 10 fases para que cualquier persona con una cierta experiencia en lenguaje C pueda

acometer la realización completa de un programa para jugar al ajedrez. Al final del proceso, el lector/a dispondrá de un juego de ajedrez plenamente funcional y desarrollado por sí mismo.

Qué incluye este libro:

- Un resumen de las reglas del ajedrez.
- Un plan de trabajo detallado, distribuido en 10 fases, para conseguir culminar con éxito el proyecto.

- Documentación sobre las librerías ncurses y SDL para realizar las versiones en modo texto y modo gráfico del programa.
- Ideas para implementar el control de movimiento de las piezas, del estado de la partida y del tiempo.
- Cómo guardar y recuperar partidas usando la notación algebraica, convirtiendo la aplicación en un PGN

viewer.

- Cómo dotar de inteligencia artificial al juego usando el algoritmo minimax.

Además, en el interior encontrarás las instrucciones para descargarte el código fuente de una implementación del juego de ajedrez completamente gratis, distribuida con Licencia Apache 2.0.

Consigue el libro aquí:

http://amzn.eu/0Q0YNL9

Índice

ANTES DE EMPEZAR

¿A quién va dirigido este libro?

¿Qué encontrarás y qué no encontrarás aquí?

Entonces, ¿este libro no trae ejercicios?

¿Por qué sé que este libro funciona?

Organización del libro

¿Alguna sugerencia?

PRIMERA PARTE: PROGRAMACIÓN

ESTRUCTURADA

Para empezar, los fundamentos

¿Qué es un programa de ordenador?

Codificación de la información

Unidades de medida de información

Estrategias de resolución de problemas

Ingeniería del software

Ciclo de vida clásico

Nadie es perfecto

El papel del programador

Estilos de programación

Programación desestructurada

Programación estructurada

Programación modular

Los datos

Tipos de datos

Operaciones con datos

Constantes y variables

Expresiones

Los algoritmos

Concepto de algoritmo

Notación de algoritmos

La programación estructurada

Teorema de la programación estructurada

Estructura secuencial

Estructuras selectivas

(condicionales)
Estructuras repetitivas (bucles)
Contadores, acumuladores, conmutadores
Programación modular
Descomposición modular: ¡divide y vencerás!
Funciones
Procedimientos
Paso de parámetros
El problema del ámbito
La reutilización de módulos
Algunas reglas de estilo
Partes de un algoritmo

Documentación

Estilo de escritura

SEGUNDA PARTE: EL LENGUAJE C

LOS LENGUAJES DE PROGRAMACIÓN

Lenguajes de alto y bajo nivel

Ensambladores, compiladores e intérpretes

INTRODUCCIÓN AL LENGUAJE C

Características básicas de C

Breve historia de C

Un lenguaje para programadores

Un lenguaje estructurado y modular

Variables, operadores y expresiones en C

- *Generalidades sintácticas de C*
- *Tipos de datos simples*
- *Variables: ámbito y asignación*
- *Constantes*
- *Conversiones de tipo*
- *Operadores y expresiones*

Estructuras de control

- *Condicional simple*
- *Condicional doble*
- *Condicional múltiple*
- *Bucle mientras*
- *Bucle repetir*

Bucle para

FUNCIONES. LA FUNCIÓN MAIN()

Funciones

Procedimientos

Paso de parámetros

La función main()

Prototipos de funciones

Estructura general de un programa en C

ENTRADA Y SALIDA ESTÁNDAR

E/S con formato

E/S simple por consola

FLUJO DE TRABAJO PROGRAMANDO CON LENGUAJE C

Edición del código fuente

Compilación

Enlace (link)

Depuración

Documentación

TERCERA PARTE: ESTRUCTURAS DE DATOS ESTÁTICAS

ARRAYS UNIDIMENSIONALES (VECTORES)

Declaración

Operaciones con vectores

Búsqueda binaria

Vectores y funciones

Representación interna de los

vectores
CADENAS
Declaración y manipulación de cadenas
Funciones para manejo de cadenas
Las cadenas y la validación de los datos de entrada
ARRAYS MULTIDIMENSIONALES
Arrays bidimiensionales (matrices o tablas)
Arrays de múltiples dimensiones
ESTRUCTURAS
Declaración
Manipulación de estructuras

Paso de estructuras a funciones
Un ejemplo de utilización de estructuras
Uniones
Enumeraciones
Nuevos tipos de datos
Tipos definidos por el usuario
Tipos supercomplejos

CUARTA PARTE: FICHEROS

Los archivos o ficheros
Ficheros, registros y campos
Registros físicos y registros lógicos
Tipos de registros

Operaciones con archivos

ORGANIZACIÓN DE ARCHIVOS

Archivos de organización secuencial

Archivos de organización relativa: hashing

Archivos de organización relativa directa

Archivos de organización relativa aleatoria (o indirecta)

Archivos de organización indexada

LOS ARCHIVOS EN C

Clasificación de los archivos en C

Flujos

Archivos y buffers

Funciones de C para la manipulación de archivos

Apertura

Cierre

Lectura y escritura

Funciones específicas de acceso directo

Diferencias entre archivos binarios y de texto

Manipulación de directorios

Procesamiento en C de archivos secuenciales

Escritura

Lectura

Búsqueda
Borrado
Modificación
Procesamiento de archivos con registros complejos
Ejemplo: archivos secuenciales de texto
Ejemplo: archivos secuenciales binarios
PROCESAMIENTO EN C DE ARCHIVOS RELATIVOS DIRECTOS
Lectura y escritura
Búsqueda
Cálculo del tamaño de un archivo

directo

Borrado

Modificación

PROCESAMIENTO EN C DE ARCHIVOS INDEXADOS

Búsqueda

Otras operaciones sobre archivos indexados

QUINTA PARTE: ESTRUCTURAS DE DATOS DINÁMICAS

PUNTEROS

Comprendiendo los punteros

Declaración e inicialización de

punteros

Asignación de punteros

Aritmética de punteros

Punteros y arrays

Arrays de punteros

Paso de punteros como parámetros

Devolución de punteros

Punteros a funciones

Punteros a punteros

GESTIÓN DINÁMICA DE LA MEMORIA

Reserva dinámica de memoria.

Arrays dinámicos.

Liberación de memoria

Funciones básicas para la gestión

dinámica de la memoria
INTRODUCCIÓN A LAS ESTRUCTURAS DINÁMICAS
LISTAS ABIERTAS (O LISTAS ENLAZADAS SIMPLES)
Qué es una lista abierta y cómo funciona
Insertar elementos
Buscar elementos
Borrar elementos
Ejemplo de implementación en C de las operaciones básicas
PILAS
Qué son las pilas y cómo funcionan

Push (insertar)

Pop (extraer y borrar)

COLAS

Qué es una cola y cómo funciona

Insertar elementos

Leer y eliminar elementos

Ejemplo de implementación en C

OTROS TIPOS DE LISTAS

Listas circulares

Listas doblemente enlazadas

Listas circulares doblemente enlazadas

ÁRBOLES GENERALES

Qué es un árbol y cómo funciona

Recorridos por árboles

ÁRBOLES BINARIOS DE BÚSQUEDA

Qué son los ABB y cómo funcionan

Buscar un elemento

Insertar un elemento

Borrar un elemento

Otras operaciones

Árboles degenerados

RECURSIVIDAD

Planteamiento de una solución recursiva

Un par de normas para aplicar correctamente la recursividad

Ventajas e inconvenientes de las

soluciones recursivas

SEXTA PARTE: ALGUNOS ASPECTOS AVANZADOS DEL LENGUAJE C

Compilación con archivos múltiples

Creación de librerías

Espacios con nombre

El preprocesador

#include

#define

#undef

#ifdef e #ifndef

#if, #elif, #else y #endif

#error

Tipos de almacenamiento

auto

register

static

extern

const

Argumentos en la línea de comandos

Manipulación a nivel de bits

Campos de bits

Limitaciones de los campos de bits

Operadores a nivel de bits

APÉNDICES

Apéndice I: funciones de uso frecuente de ANSI C

Apéndice II: el compilador Dev-C++

Herramientas para la programación en C bajo Windows

El IDE de Dev-C++

El Depurador o Debugger

¿Y si quiero usar el compilador o el depurador "a mano"?

Apéndice III: el compilador de GNU C/C++ (gcc)

Herramientas para la programación en C bajo Linux

El compilador gcc

Depuradores: gdb y ddd

Control de dependencias: la herramienta make

Páginas de manual

APÉNDICE IV: LA LIBRERÍA NCURSES, O CÓMO SACAR LOS COLORES A NUESTRAS APLICACIONES DE CONSOLA

Qué es Ncurses

Inicialización de Ncurses

Escribir y leer

Colores

Ejemplo de uso de Ncurses

APÉNDICE V: LA LIBRERÍA SDL, O CÓMO CONSTRUIR APLICACIONES GRÁFICAS CON C

Instalación de SDL
Compilación y enlace
Inicialización y terminación de la pantalla gráfica
Dibujar gráficos en la pantalla
Control del teclado
Definición de colores
Mostrar texto en la pantalla gráfica: la librería SDL_TTF
UNA ÚLTIMA COSA…