

SEGUNDA EDICIÓN

C PARA INGENIERÍA ELECTRÓNICA

Jorge A. Argibay

Mauro Gullino

C para Ingeniería Electrónica

Jorge A. Argibay
Mauro Gullino

Argibay, Jorge

C para ingeniería electrónica / Jorge Argibay ; Mauro Gullino.
2a ed revisada. - Haedo : el autor, 2017. 360 p. ; 25 x 18 cm.

ISBN 978-987-42-3601-2

1. Lenguaje de Programación. 2. Informática. I. Argibay, Jorge

II. Título

CDD 004

2017 © segunda edición por Jorge A. Argibay y Mauro Gullino

2005 © primera edición por Jorge A. Argibay

Reservados todos los derechos.

Quisiera agradecer y destacar a aquellos que conformaron y conforman mi excelente equipo de colaboradores directos, y que de alguna manera han contribuido a la evolución de las clases impartidas y por lo tanto han dado soporte al contenido de este libro. Todos ellos son excelentes profesionales que me han acompañado a lo largo de las asignaturas Informatica I y II, Técnicas Digitales I y las asignaturas de la Tecnicatura Superior en Programación.

En primer lugar menciono al Ing. Ruben Lozano, la Lic. Giannella “Tana” Ligato y al Lic. Hernan Monserrat quienes ya no están trabajando conmigo, aunque siguen en contacto de alguna u otra manera, algunos de ellos en el exterior.

Una mención especial merece la Ing. María Cristina Maidana, amiga personal, compañera de viajes y madrina de una de mis hijas, quien me acompañó en una larguísima trayectoria profesional que comenzó en 1981.

En el presente cuento con la Lic. Artemisa Trigueros, el Ing. Facundo Larosa, la Lic. Vilma Giúdice, la Dra. Rocío Rodriguez y el Lic. Mauro Gullino, coautor de esta edición, sin quienes no sería posible mantener el nivel académico que pretendemos.

Por último quiero mencionar a quienes en la actualidad se desempeñan como auxiliares alumnos, estando en el camino a la graduación en Ingeniería Electrónica, la Srtá. Tamara Martynow y el Sr. Brian E. Ferreyra.

Como siempre, agradezco a mi esposa y mis hijas por la paciencia que me tuvieron y el apoyo que me brindaron durante el trabajo en la primera edición, sabiendo perdonarme por el tiempo que no compartí con ellas.

Gracias Melisa por el dibujo del templo tibetano.

Finalmente agradezco a los alumnos por los elogiosos comentarios realizados sobre la primera edición de este libro, lo que motivó el esfuerzo para la implementación de esta segunda edición.

J. A. A.

A mis padres, por haberme dado la educación.

M. G.

Prólogo a la segunda edición

Esta segunda edición tiene origen en diferentes factores.

Primero, la absoluta vigencia del lenguaje de programación C a lo largo de este tiempo, desde que fuera desarrollado por Dennis Ritchie, entre los años 1969 y 1972. El mismo es base y soporte de lenguajes posteriores.

Luego, la conveniencia de utilizarlo como lenguaje introductorio para el estudio de la Programación.

Además, la capacidad de este lenguaje de permitir accesos al hardware –utilizados en Ingeniería Electrónica– y al mismo tiempo poder comportarse como un lenguaje de alto nivel mediante sus mecanismos de abstracción.

Por último, la utilidad que demostró la primera edición de este libro, manifestada en los elogiosos comentarios de los usuarios. Esto nos lleva a pensar que es una herramienta académica muy útil, aunque precisando algunos retoques.

En esta edición se abandona la orientación hacia el Turbo C de la empresa Borland para utilizar un enfoque que permita compilar los ejemplos en un ambiente multiplataforma moderno, pudiendo utilizar el lector diversos IDEs del mercado, como así también su uso en sistemas operativos Windows y Linux.

En este sentido hemos revisado, modificado e incorporado información a todos y cada uno de los capítulos, incluyendo sus ejemplos ejecutables.

Por último, se revisaron e incorporaron nuevos contenidos que permiten abarcar más temas de las asignaturas Informática II y Programación II. Temas que, por otro lado, ya estaban disponibles para los alumnos en forma de publicaciones de cátedra.

Salvo que se exprese en contrario, siempre tomaremos como referencia el compilador GCC 4.9 y utilizaremos el estándar ANSI C89.

Prólogo a la primera edición

La presente obra surge como compilación ordenada y adaptada de mis publicaciones anteriores en forma de apuntes, los que constituyen los capítulos.

Dichos apuntes, a su vez, son el testimonio escrito de mis propias clases frente a alumnos y contienen el resultado de más de 20 años de experiencia en el dictado de asignaturas como Informática I y II, Computadoras Digitales, Programación, Lenguajes, Técnicas Digitales y Microprocesadores tanto en el ámbito universitario como en empresas privadas.

Este libro se adapta a los temas de la materia Informática I, y cubre parcialmente los de Informática II de la carrera de Ingeniería Electrónica de la Universidad Tecnológica Nacional. Asimismo cubre totalmente los temas de la materia Programación I de la Tecnicatura Superior en Programación dependiente de la UTN FRBA.

El compilador adoptado para este trabajo es el Turbo C de Borland v. 1.01. Si bien es un compilador que lleva muchos años en el mercado, tiene la ventaja de ser de libre distribución, simple y fácil de transportar.

El libro encara la programación en lenguaje C desde la construcción lógica de diagramas estructurados hasta el uso de herramientas en C como punteros lejanos, estructuras, uniones, etc., acompañando el desarrollo teórico con más de 120 ejemplos ejecutables.

Dado que la obra sigue el desarrollo de las clases regulares, los temas están expuestos en orden paulatinamente creciente de dificultad lo que posibilita al lector a seguirlos sin problemas.

Se acompaña la mayoría de los capítulos con grupos de problemas propuestos destinados a aplicar, ejercitarse y afianzar los temas teóricos tratados.

Se adjunta un apéndice que guía al lector en la utilización del entorno integrado de desarrollo de Turbo C.

J. A. A.

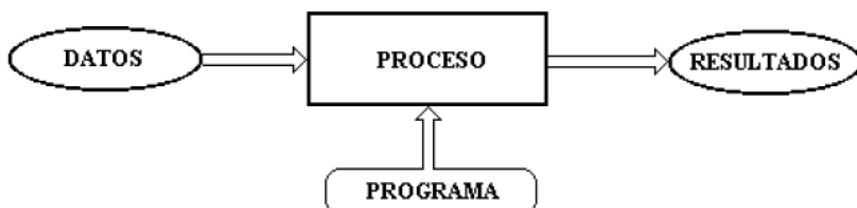
1. DIAGRAMACIÓN

CONSTRUCCIÓN DE PROGRAMAS

Un **programa** es un conjunto de instrucciones u órdenes capaces de ser obedecidas por una computadora de forma tal que, al ejecutarlas, realice una determinada tarea en un tiempo finito.

El objetivo del programa es que la computadora sea capaz de entregar los **resultados correctos** a partir de los datos. Es conveniente tener en cuenta que no siempre los resultados ni los datos serán numéricos. Podemos considerar la simple tarea de perder tiempo como un resultado, si esto fuera lo deseado.

Para esto, la computadora debe ser capaz de seguir las instrucciones de manera precisa y, por lo tanto, éstas no deberán ser ambiguas. La tarea que realiza la computadora obedeciendo al programa recibe el nombre de **proceso**.



Debe comprenderse que la computadora no es capaz de realizar nada por sí sola, y siempre debe seguir la lógica dada por un programa. Por eso decimos que nada está “*sobreentendido*”.

La ventaja es que cada simple acción la realiza con velocidad electrónica. Se dice frecuentemente que “*la computadora es un opa sumamente rápido*”.

Es tarea del **programador** construir la secuencia de instrucciones o **programa** para que se pueda arribar correctamente al resultado deseado. La tarea de construcción de un programa involucra dos pasos fundamentales:

- Diagramación
- Codificación

DIAGRAMACIÓN

Este proceso consiste en la construcción de un **diagrama** (esquema o dibujo) que muestre la secuencia lógica de pasos a seguir para resolver el problema planteado. De lo expresado anteriormente surge que el diagrama contiene la **lógica** del programa.

Se tratará en este capítulo la forma de construir diagramas, el origen y reglas de la diagramación estructurada y la utilización de los Diagramas de Nassi-Schneiderman o **Diagramas de Chapin**.

CODIFICACIÓN

Consiste en expresar el diagrama utilizando la **sintaxis** de algún **lenguaje** de computación, de forma que pueda ser traducido por el programa **compilador** y posteriormente ejecutado por el procesador.

La codificación se desarrollará en capítulos posteriores al estudiar la sintaxis del **lenguaje C**.

PROCESO DE DIAGRAMACIÓN

Según el matemático **George Polya** (fines de los años 40) el proceso de diagramación puede descomponerse en 4 fases:

1. Comprensión del problema
2. Obtención de un algoritmo que lo resuelva
3. Representación del algoritmo en forma de diagrama
4. Evaluación de la corrección del diagrama

1. COMPRENSIÓN DEL PROBLEMA

Este es uno de los obstáculos mas difíciles de sortear que se presentan a programadores noveles. Básicamente consiste en “*saber lo que se quiere*” a fin de encontrar una forma de obtenerlo. También implica superar el difícil primer paso de “*¿cómo arranco?*”.

El tema se complica más aun cuando se debe resolver un problema ajeno, y es el otro el que no sabe lo que quiere, o lo expresa mal o de manera ambigua. Como ser: “*Quiero apretar un botón y que se me solucionen todos los problemas. Para eso tengo una computadora. ¿No?*”.

A fin de organizar la superación de este paso podemos enumerar algunos aspectos relacionados con la comprensión del problema, que pueden orientar al programador. Se deberá tener en claro “*qué quiero, qué tengo y qué me falta*”.

Es decir, determinar claramente los extremos del proceso, que podemos clasificar en:

- I. Objetivos
- II. Datos
 - a. Datos disponibles útiles
 - b. Datos disponibles superfluos
 - c. Datos faltantes

I. OBJETIVOS

Estos surgen generalmente de los **requerimientos** o del enunciado. Es posible que no aparezcan claramente. Si este es el caso, no se debe avanzar hasta clarificarlo.

Esto responde a la pregunta: “*¿Qué quiero?*”

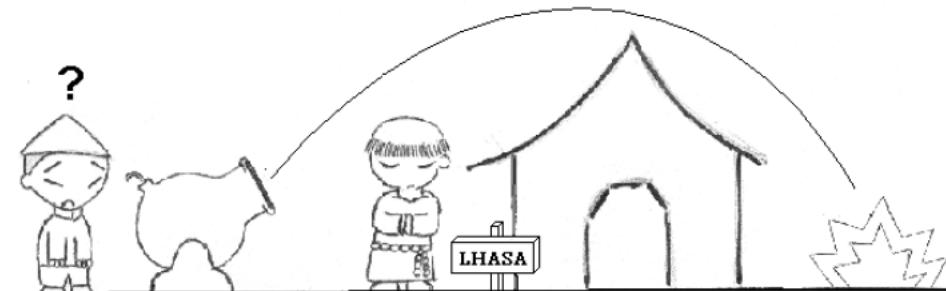
II. DATOS

El enunciado mismo del problema suele incorporar los datos necesarios para su resolución, pero es importante no dejarse engañar por la posible presencia de datos innecesarios o superfluos que pueden llevar a confusión, separando los datos útiles de los inútiles.

Por otra parte, puede ocurrir que no se dispongan de todos los datos necesarios para resolver el problema y sea imprescindible obtenerlos.

Observemos el siguiente caso. El ejercito chino desea bombardear el monasterio budista de Lhasa. Se conoce la distancia entre el cañón y el monasterio, la elevación y la velocidad inicial de la bala. Además no hay viento.

Se aplican las formulas de tiro parabólico, para calcular el ángulo de disparo.



Sin embargo el disparo falla. ¿Por qué?

Ocurre que hay un **dato encubierto** que conduce a un dato faltante o no tenido en cuenta. Lhasa, antigua capital de Tíbet, se encuentra a 4000 metros sobre el

nivel del mar. Allí la densidad del aire es mucho menor y por lo tanto el cálculo de tiro parabólico debe ser corregido, dado que la bala se frena menos y el disparo resulta largo.

2. OBTENCIÓN DE UN ALGORITMO

El siguiente paso es la obtención del **algoritmo** de resolución del problema, ya sea desarrollándolo o utilizando un algoritmo desarrollado por otro.

Pero, ¿qué es un algoritmo? Existen innumerables definiciones de algoritmo. Podemos tomar la siguiente definición:

ALGORITMO: Conjunto finito de pasos definidos, estructurados en el tiempo, formulados en base a un conjunto finito de reglas no ambiguas, que proveen un procedimiento para dar la solución o indicar la falta de ésta a un problema, en un tiempo determinado.

O bien simplificarla un poco:

ALGORITMO: Secuencia finita de pasos ejecutables no ambiguos, que de seguirlos debe terminar en algún momento.

La anterior definición indica que es la **secuencia** de pasos que nos conducirá a la solución del problema, que esta secuencia debe ser finita, y que si la computadora ejecuta estos pasos debe hacerlo en un tiempo también finito.

De manera mucho más simple y casera podemos interpretar al algoritmo como “*la manera de resolver el problema*”.

Este es otro punto crucial en la resolución del problema. El algoritmo de resolución se puede obtener de bibliografía, resoluciones previas, desarrollos teóricos, etc. Pero si el algoritmo no está disponible, entonces es necesario **desarrollarlo**.

Para esto debemos saber cómo resolver el problema que se plantea. “Nadie puede construir un programa de ajedrez si no sabe él mismo jugar al ajedrez.”

Esto no significa que seamos capaces de resolver el problema planteado, sino que simplemente sepamos *cómo* hacerlo.

Probablemente no seríamos capaces de determinar los primeros 10.000 números primos, dado que no nos alcanzarían los días de nuestra vida para hacerlo. Pero sí sabemos como encontrar el primero, el segundo, y todos los demás. Bastará indicarle a alguien muy rápido (la computadora) cómo resolver el problema (algoritmo) mediante una serie de pasos (el programa).

ORIGEN DEL NOMBRE

La denominación *algoritmo* deriva del nombre del matemático árabe Al'Khwarizmi.

Muhammad ibn Musa abu Djafar Al'Khwarizmi, también llamado Al'Khorezmi, nació alrededor del 780 DC (otros citan 800 DC) en Khorezm, al sur del Mar de Aral (hoy Khiva, Uzbekistán), que había sido conquistado 70 años antes por los árabes. Su nombre significa “Mohamed, hijo de Moisés, padre de Jafar, nativo de Khorezmi” (hoy Khiva). Hacia el 820, Al'Khwarizmi fue llamado a Bagdad por el califa abásida Al'Mamun, segundo hijo de Harun al Rashid, por todos conocido gracias a *Las Mil y una Noches*.

Al'Mamun continuó el enriquecimiento de la ciencia árabe y de la Academia de Ciencias creada por su padre, llamada la Casa de la Sabiduría, lo que traería importantes consecuencias en el desarrollo de la ciencia en Europa, principalmente a través de España.

Al'Khwarizmi fue un recopilador de conocimiento de los griegos y de la India, principalmente matemáticas, pero también astronomía (incluyendo el calendario Judío), astrología, geografía e historia. Su trabajo más conocido y usado fueron sus *Tablas Astronómicas*, basadas en la astronomía india. Incluyen algoritmos para calcular fechas y las primeras tablas conocidas de las funciones trigonométricas seno y cotangente. Lo más increíble es que no usó los números negativos (que aún no se conocían), ni el sistema decimal ni fracciones, aunque sí el concepto del cero. Su Aritmética, traducida al latín como *Algoritmi de numero Indorum*, introduce el sistema numérico indio (sólo conocido por los árabes unos 50 años antes) y los algoritmos para calcular con él. Finalmente tenemos el *Álgebra*, una introducción compacta al cálculo, usando reglas para completar y reducir ecuaciones. Además de sistematizar la resolución de ecuaciones cuadráticas, también trata geometría, cálculos comerciales y de herencias.

Rescató de los griegos la rigurosidad y de los indios la simplicidad (en vez de una larga demostración, usar un **diagrama** junto a la palabra “*mira*”). Sus libros son intuitivos y prácticos y su principal contribución fue simplificar las matemáticas a un nivel entendible por no expertos. En particular muestran las ventajas de usar el sistema decimal indio, un atrevimiento para su época, dado lo tradicional de la cultura árabe. La exposición clara de cómo calcular de una manera sistemática a través de algoritmos diseñados para ser usados con algún tipo de dispositivo mecánico similar a un ábaco, más que con lápiz y papel, muestra la intuición y el poder de abstracción de Al'Khwarizmi. Incluso se preocupaba de reducir el número de operaciones necesarias en cada cálculo. Por esta razón, aunque no haya sido él el inventor del primer algoritmo, merece que este concepto esté asociado a su nombre.

TÉCNICAS DE DESARROLLO DE ALGORITMOS

Podemos mencionar dos metodologías básicas para encarar el desarrollo de un algoritmo o la resolución de un problema:

- Top Down
- Bottom Up

TOP DOWN

Esta técnica sigue los lineamientos de Al'Khwarizmi en los que expresa que un problema matemático o de otra índole debe ser descompuesto en **subproblemas** de menor complejidad a fin de encarar cada uno de ellos en forma individual.

Si aún así se presentan subproblemas demasiado complejos, estos deben ser descompuestos nuevamente, lo que lleva a un refinamiento iterativo. El proceso finaliza cuando cada partición es resoluble.

BOTTOM UP

Esta técnica sugiere el proceso inverso. Se toma sólo una fracción del problema y se la expande, probándola en cada paso a fin de verificar la corrección del desarrollo.

Cada nuevo paso le incorpora un grado de complejidad a lo ya realizado.

Ambas técnicas se aplicarán en futuros ejemplos.

3. REPRESENTACIÓN DEL ALGORITMO EN FORMA DE DIAGRAMA

Una vez determinado el algoritmo, es necesario expresarlo en la forma de un diagrama que permita seguir fácilmente su lógica, y posteriormente codificarlo en algún lenguaje de programación.

Existen diagramas de formatos diversos que se aplican a los distintos pasos que pueda presentar un algoritmo. Por lo tanto, para comprender este punto, es necesario conocer los formatos ofrecidos. Esto se presentará en las secciones siguientes.

4. EVALUACIÓN DE LA CORRECCIÓN DEL DIAGRAMA

Una vez elaborado el diagrama es conveniente verificar si resuelve correctamente el problema.

Dicha verificación se realiza mediante una “*prueba de escritorio*”, la que consiste en seguir los pasos del diagrama aplicando conjuntos de datos con resultados conocidos.

Esto no asegura la corrección del diagrama, pero da una presunción de ello.

IMPORTANCIA DE LA DIAGRAMACIÓN

Existen muchos programadores que acostumbran a sentarse directamente frente a la consola y encarar en ese momento el problema, aparentemente, dejando de lado o pasando por alto el proceso de diagramación.

Esto es posible cuando la experiencia del programador y la simplicidad del problema a resolver lo permiten. Pero debe tenerse en cuenta que el proceso de diagramación no se está omitiendo, sino que se está llevando a cabo mentalmente.

Esta es una costumbre un tanto riesgosa dado que puede llevar al programador a quedarse mirando una pantalla en blanco y sin saber qué hacer.

El diagrama tiene además la función de **documentación** de la lógica del programa. Este es entonces un documento que permitirá al lector comprender más fácilmente el programa.

PROGRAMACIÓN ESTRUCTURADA

Existen diversas orientaciones o **paradigmas** de programación como ser programación estructurada, programación orientada a objetos, programación orientada a eventos, programación visual, etc.

Cada una de ellas tiene un área de aplicación específica. Gran parte de los programas aplicables en ingeniería electrónica responden a la necesidad de programación **imperativa** y **estructurada**.

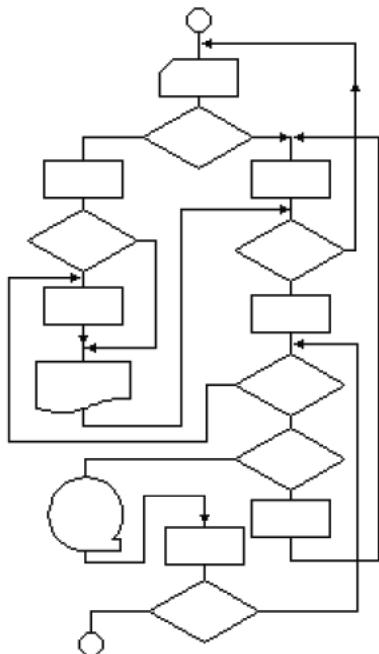
A fin de comprender mejor la necesidad de las **reglas** de dicha programación, es conveniente realizar un breve recorrido por sus orígenes.

ORÍGENES

Durante la década de 1950 las primitivas computadoras eran lentas, presentaban un bajo tiempo entre fallas y sus recursos (memoria) eran escasos.

Forzosamente, los programas diseñados para estas máquinas debían ser relativamente simples dado que no podían tener grandes exigencias sobre el **hardware** existente.

Debido a la simpleza del **software** los programadores podían tomarse ciertas licencias en su construcción que conducían a que la programación fuera artesanal y no se dispusieran de métodos sistemáticos para su desarrollo.



Los **diagramas de flujo** tradicionales disponían de formas o esquemas para diversas aplicaciones que estaban enlazados mediante “*líneas de flujo*” que marcaban el camino a seguir.

La diversidad de formas, así como la presencia indiscriminada de líneas de flujo producía diagramas difíciles de entender y de seguir.

Las líneas de flujo no disponían de ningún tipo de limitación debido al abuso del “*salto incondicional*” o **sentencia goto**.

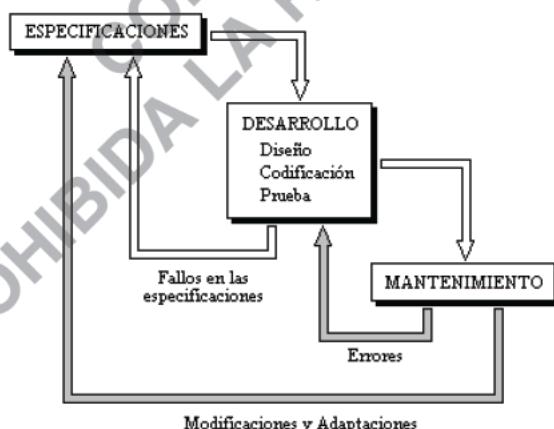
La figura de la izquierda muestra un ejemplo de diagrama tradicional, omitiendo la tarea en cada uno de los bloques.

En él se pueden observar los entrecruzamientos de las líneas de flujo que dan un aspecto caótico al diagrama, y justifican el apodo de “*programas tallarín*” a los programas producidos de esta manera.

A medida que el hardware iba evolucionando, también lo hacían los programas y aumentaba la complejidad del software que pasó, además, a ser un producto genérico de distribución masiva.

CICLO DE VIDA DEL SOFTWARE

En este contexto, hasta el momento de ser finalmente descartado por obsoleto, el software pasó a presentar el ciclo de vida que se muestra a continuación.



Luego de la etapa de desarrollo y aceptación, y una vez puestos en el mercado, los programas eran frecuentemente modificados.

Esto se debía a dos causas:

- **Corrección** de errores detectados tardíamente.
- **Modificación** del software a fin de obtener versiones mejoradas.

En cualquiera de los dos casos, la modificación de los *programas tallarín* se hacía muy problemática. Dada la intrincada vinculación de sus secciones, una modificación producía efectos indeseados en otro lugar del programa.

REGLAS DE LA DIAGRAMACIÓN ESTRUCTURADA

Dada la situación descripta previamente, la construcción artesanal de programas sin ningún tipo de **metodología** de diseño se tornó inaceptable.

El conjunto de reglas y técnicas impuestas para resolver los mencionados problemas recibió el nombre de **Programación Estructurada** aunque se apliquen básicamente al proceso de diagramación.

Estas técnicas aumentan considerablemente la productividad del programa reduciendo el tiempo requerido para escribir, verificar, depurar y mantener.

La programación estructurada utiliza un número limitado de **estructuras de control** que minimizan la complejidad de los problemas y que reducen los errores.

Entre las técnicas incorporadas se encuentra el **diseño descendente** (Top Down), recursos abstractos (la descomposición de acciones complejas en simples) y **modularidad** de diseño.

TEOREMA FUNDAMENTAL DE BOHM Y JACOPINI

Este teorema, también conocido como **Teorema de Dijkstra**, sienta las bases de la diagramación estructurada al limitar la cantidad de estructuras de control a tres.

TEOREMA FUNDAMENTAL: Todo **programa propio** puede ser escrito utilizando solamente tres tipos de estructuras de control

SECUENCIA
SELECCIÓN
ITERACIÓN

Un programa es **propio** si cumple los siguientes requisitos:

- Tiene una única entrada y una única salida.
- Existen caminos desde la entrada hasta la salida que pasan por todas las partes del programa.
- Todas las sentencias son ejecutables.
- No aparecen bucles infinitos.

SECUENCIA: Es aquella estructura de control que garantiza que una acción sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente.

SELECCIÓN: Se utiliza para tomar decisiones lógicas. En ésta se **evalúa** una **condición** y en función de su resultado se **ejecuta** una opción u otra. Las condiciones se especifican usando **expresiones** lógicas.

ITERACION: Se utiliza para repetir una secuencia de instrucciones un número determinado de veces. Se la denomina también **bucle** o lazo.

Estas estructuras o bloques deben a su vez cumplir ciertas reglas:

- Sólo pueden tener una entrada y una salida.
- Pueden contener, a su vez, otros bloques o estructuras de cualquier tipo de manera total.
- No puede haber inclusiones parciales de un bloque en otro.

LENGUAJES ESTRUCTURADOS

Se denominan de esta manera a los lenguajes de computación que poseen **sentencias** capaces de cumplir con las reglas de la programación estructurada. Se desprende de esto que algunos lenguajes no lo son.

Efectivamente, los lenguajes anteriores a la revolución estructurada no prevén sentencias de este tipo. Tal es el caso de Fortran, Cobol y Basic en sus versiones originales. Posteriormente fueron modificados para acceder a los beneficios de la programación estructurada, como es el caso de Cobol Estructurado.

El lenguaje Assembler no es estructurado ni puede serlo debido a su característica de bajo nivel.

Otros lenguajes como Pascal, ADA, C y C++ fueron diseñados originalmente como lenguajes estructurados.

MODULARIDAD

La modularidad de un programa no es un requerimiento de la programación estructurada sino una ventajosa técnica.

Esta técnica consiste en dividir al programa en módulos, cada uno de los cuales ejecuta una única actividad o tarea y se codifican independientemente de otros módulos. Cada uno de estos módulos se analiza, codifica y pone a punto por separado.

Cada programa contiene un módulo llamado **programa principal** que controla todo lo que sucede; se transfiere el control a submódulos de modo que ellos puedan ejecutar sus funciones. Los módulos son independientes en el sentido en

que ningún módulo puede tener acceso directo a cualquier otro excepto el módulo al que llama y sus propios submódulos.

VENTAJAS DE LA PROGRAMACIÓN ESTRUCTURADA

En base a los puntos anteriores podemos enumerar las siguientes ventajas :

- Los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación. La estructura del programa es más clara puesto que las instrucciones están más ligadas o relacionadas entre sí, por lo que es más fácil comprender lo que hace cada función.
- Reducción del esfuerzo en las pruebas. El programa se puede tener listo para producción normal en un tiempo menor que el desarrollo artesanal; por otro lado, el seguimiento de las fallas o depuración (*debugging*) se facilita, debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir más fácilmente.
- Reducción de los costos de mantenimiento.
- Programas más sencillos y más rápidos.
- Aumento en la productividad del programador.
- Se facilita la utilización de las otras técnicas para el mejoramiento de la productividad en programación.
- Los programas quedan mejor documentados internamente.

DIAGRAMAS DE CHAPIN O NASSI-SCHNEIDERMAN

Hasta ahora hemos comentado las reglas de la diagramación estructurada sin establecer su forma gráfica.

Existen diversos diseños siendo el más difundido el de Nassi-Schneiderman (Ike Nassi y Ben Schneiderman), también conocido como Diagramas de Chapin (Ned Chapin).

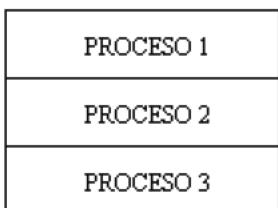
El diagrama de Chapin es un modo gráfico de representar algoritmos en forma de diagrama de flujo, siendo su propósito principal crear la estructura lógica para un programa. En los diagramas de Chapin desaparecen las líneas de flujo de los diagramas tradicionales.

Al estar éstas ausentes se establece que el **flujo** es siempre descendente. Esto hace que sean más fáciles de seguir y además más compactos, debido a la ausencia de líneas de flujo.

Existen programas gráficos dedicados a la construcción de diagramas de Chapin. Inclusive programas generadores de código básico a partir del diagrama.

A continuación veremos los símbolos o formas de Chapin para cada una de las estructuras básicas.

SECUENCIA



Indica que las instrucciones de un programa se ejecutan una después de la otra, en el mismo orden en el cual aparecen en el programa. Se representa gráficamente como una caja después de otra, ambas con una sola entrada y una única salida.

Las cajas denominadas PROCESO 1, PROCESO 2 y PROCESO 3 pueden ser definidas para ejecutar desde una simple instrucción hasta un módulo o programa completo, siempre y cuando éstos también sean programas propios.

SELECCIÓN



También conocida como la estructura **IF-ELSE**, plantea la selección entre dos alternativas con base en el resultado de la evaluación de una **condición booleana**, es decir, cuya respuesta sea *verdadero/falso* o *sí/no*.

En el diagrama de flujo anterior, CONDICION es una condición booleana que se evalúa; PROCESO A es la acción que se ejecuta cuando la evaluación de la condición resulta *verdadera* y PROCESO B es la acción ejecutada cuando resulta *falsa*.

La estructura tiene una sola entrada y una sola salida. Los procesos A y B pueden estar constituidos por cualquier estructura básica o conjunto de estructuras, o bien estar vacíos.

ITERACIÓN

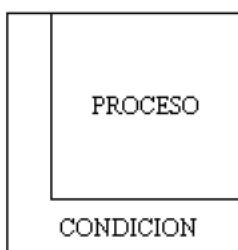
Corresponde a la ejecución repetida de un proceso mientras se cumpla una determinada condición. Existen dos tipos de iteración según la ocasión en que se evalúe la condición.



WHILE

Aquí el PROCESO se ejecuta repetidamente *mientras* que la CONDICION se cumpla o sea cierta. También tiene una sola entrada y una sola salida; igualmente el PROCESO puede ser cualquier estructura básica o conjunto de estructuras.

La característica particular es que se ingresa en el bloque evaluando la condición en primer término y ejecutando el proceso si es que ésta se cumple.



DO-WHILE

Es similar a la anterior pero tiene la particularidad de ejecutar el proceso en primer término, y luego evaluar la condición para determinar si se continúa en el bucle o no.

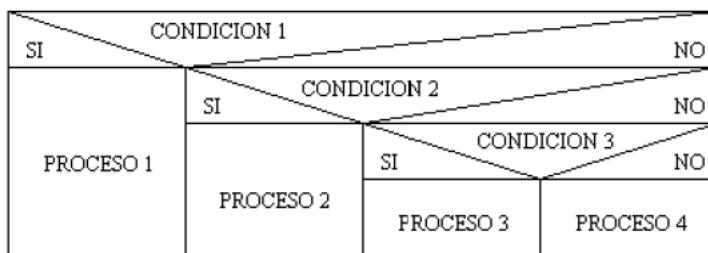
De esta manera la cantidad mínima de veces que se ejecuta el proceso es 1, mientras que en el caso anterior (*while*) el proceso puede no ejecutarse nunca si la condición no se cumple de entrada.

SELECCIONES COMPLEJAS

Las estructuras anteriores son las que prevé el teorema fundamental de Bohm y Jacopini, pero es conveniente presentar una combinación anidada de selecciones conocida como **escalonador** y una selección especial que representa una sentencia presente en la mayoría de los lenguajes, denominada **case**.

ESCALONADOR

Se trata de estructuras de decisión **anidadas**. Esto es, cuando una estructura de selección contiene en alguno de sus bloques de proceso, a su vez, una o más estructuras de selección.



En el caso del escalonador, como se ve en el diagrama, las selecciones anidadas se ubican en la salida por "NO" de la selección anterior.

Si se cumple la **CONDICION 1** se ejecuta el **PROCESO 1**, pero si no, entonces se evalúa la **CONDICION 2**.

Si ésta se cumple se ejecuta el **PROCESO 2**, en caso contrario se evalúa la **CONDICION 3**. De cumplirse se ejecuta el **PROCESO 3** y en caso contrario se ejecuta el **PROCESO 4**.

4 que termina siendo el proceso *default* (por omisión) dado que es aquel que se ejecuta cuando ninguna CONDICION se cumplió.

El escalonador es una estructura de selección múltiple general en el que las condiciones no tienen por qué estar relacionadas entre sí.

CASE

La estructura CASE selecciona un proceso *por igualdad* entre un valor determinado (VALOR CONDICION) y un conjunto de valores distintos VALOR 1, 2, 3.

Si ninguna comparación da positivo se ejecuta el proceso por omisión.

VALOR CONDICION			
VALOR 1	VALOR 2	VALOR 3	DEFAULT
PROCESO 1	PROCESO 2	PROCESO 3	PROCESO DEFAULT

Como se aprecia, la estructura CASE es menos general y más limitada que el escalonador, pero más cómoda cuando se encuentra su aplicación específica. Esta se reduce en la mayor parte de los casos a la implementación de menús.

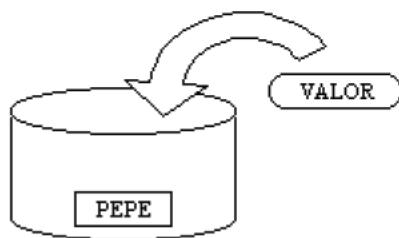
VARIABLES

Antes de presentar algunos ejemplos de diagramas de Chapin que representen algoritmos, es conveniente tener claro el concepto de **variable** y algunas de sus frecuentes aplicaciones.

Podemos considerar que una **variable** es un contenedor identificado con un nombre o etiqueta.

Este contenedor puede albergar **valores** diferentes a lo largo de la ejecución del programa.

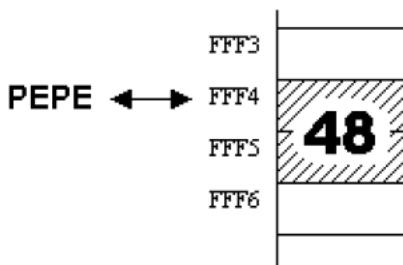
Cuando se hace referencia al nombre de la variable, indirectamente se hace referencia también a su contenido, que es el valor que la variable alberga en ese momento.



En el modelo presentado en la figura PEPE es la variable que contiene algún valor.¹

En la computadora, una variable corresponde a una o más posiciones de memoria que actúan como contenedor.

El nombre de la variable está vinculado a la dirección de memoria donde comienza el espacio reservado a ella.



En la figura de la derecha se observa que la variable PEPE ocupa 2 bytes en memoria, contiene al valor 48 y está vinculada con la posición FFF4H.

CONTADORES, ACUMULADORES Y FLAGS

Como se mencionó en la sección anterior, las variables son contenedores generales de datos. Esto significa que pueden contener datos de distintos tipos con fines diversos. Pero entre estos fines hay aplicaciones particulares que conviene diferenciar ya que se utilizan habitualmente al programar.

CONTADOR

Es una variable entera (es decir, que contiene números enteros) destinada a llevar la cuenta de eventos ocurridos durante el proceso. Esta cuenta puede ser ascendente o descendente. Los contadores generalmente modifican su valor en una unidad cada vez. A esto se denomina **incremento** o **decremento**.

ACUMULADOR

Como su nombre lo indica, esta variable se utiliza para acumular cantidades (tanto positivas como negativas). Contienen cantidades enteras que aumentan o disminuyen en valores también enteros.

FLAG O BANDERA

También llamada testigo o centinela. Generalmente, esta variable contiene uno de dos valores determinados (tomando el significado de *bandera alta/bandera baja, sí/no, verdadero/falso*).

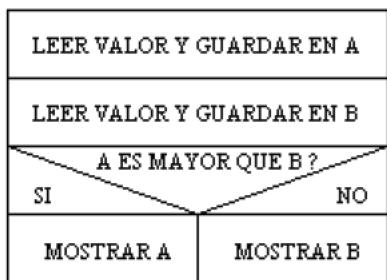
¹ El uso de nombres como PEPE o LOLA en nuestros ejemplos remarca que éstos son nombres de fantasía elegidos por el usuario. En programas reales es conveniente utilizar el mejor nombre que describa los valores que contiene la variable durante la ejecución del programa, por ejemplo: SUELDO, PROMEDIO, SUMA, CANTIDAD, EDAD...

Se la utiliza para determinar si el flujo del programa atravesó determinada sección y posteriormente tomar una decisión en consecuencia.

EJEMPLOS DE DIAGRAMAS DE CHAPIN

A continuación se mostrarán algunos ejemplos de aplicación de las distintas estructuras de los diagramas de Chapin, como así también del uso de variables.

EJEMPLO 1



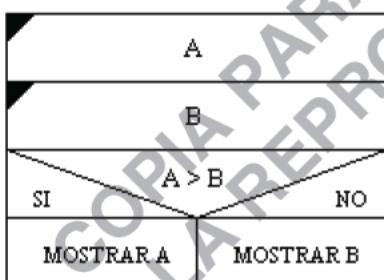
Ingresar 2 valores. Determinar cuál es el mayor.

Este es un problema muy simple que puede ser dividido en dos secciones :

- Ingreso de datos
- Toma de decisión.

Se utilizan dos variables (A y B) para almacenar los datos.

Una vez que los valores están almacenados se decide por comparación cuál es el mayor y se lo muestra en pantalla. Si ambos son iguales no importa cuál se muestra.



Esta variante del diagrama muestra la lectura de datos en forma más compacta.

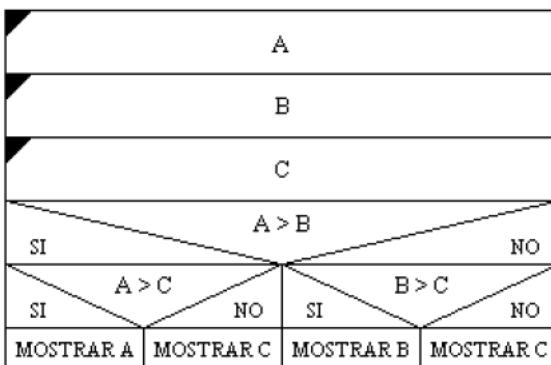
Acostumbramos agregar un triángulo negro en la esquina superior izquierda del bloque para indicar que se está realizando un ingreso de datos.

Esto no altera la filosofía del bloque ya que no es un nuevo tipo de estructura, y permite apreciar rápidamente el ingreso de datos. El interior del bloque sólo indica la variable de almacenamiento en donde se guarda el dato ingresado por el usuario.

También se compactó el texto de la condición. De ahora en más seguiremos este tipo de nomenclatura.

EJEMPLO 2

Ingresar 3 valores. Determinar cuál es el mayor.

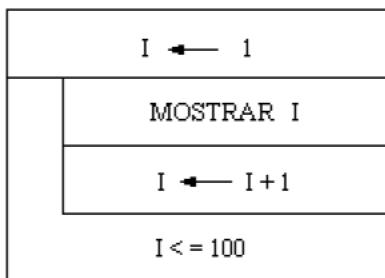


en caso de igualdad es irrelevante cuál variable se selecciona.

Si la cantidad de valores a comparar fuera mayor, el diagrama se complica y debería recurrirse a otro algoritmo.

EJEMPLO 3

Mostrar en pantalla los números de 1 al 100.



Este ejemplo nos permite apreciar una típica secuencia de conteo.

Se utiliza la variable I como contador (usualmente I, J, K representan números enteros) y se la **inicializa** en “1”.

Luego se ingresa en una iteración de tipo **do-while** en la que se muestra el valor que contiene I y luego se lo **incrementa**. La condición “I <= 100” significa “I menor o igual que 100”.

EJEMPLO 4

Ingresar 20 valores. Mostrar su suma.

En este caso es necesario ingresar 20 valores y procesarlos. Nótese que no siempre se necesitan veinte variables para ingresar veinte valores.

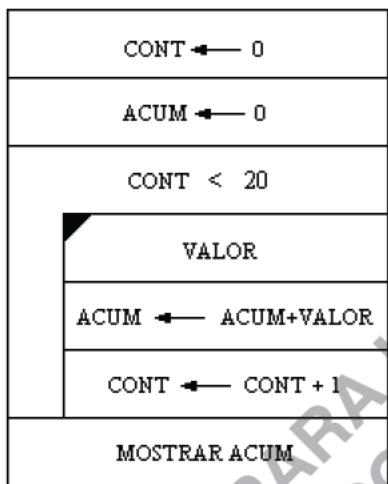
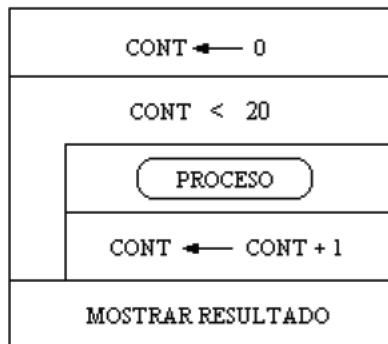
Podemos construir un diagrama preliminar que se encargue de la cuenta de los procesos de datos, esto es “*haremos algo 20 veces*”.

Este ejemplo nos permite aplicar el concepto de selección anidada.

Se ingresan los 3 datos y luego se selecciona el mayor entre A y B.

Si A es mayor se los compara con C en una segunda selección, caso contrario, se comparan B con C.

Igual que en el caso anterior,



Este diagrama será similar al del ejemplo anterior con la diferencia de la utilización, en este caso, de un lazo **while**.

La variable encargada de llevar el registro de la cuenta de procesos se denominará **CONT** y se la inicializará en cero.

Nótese la secuencia: inicialización, ingreso al lazo, proceso (aún no definido) e incremento. Este tipo de estructura será utilizado frecuentemente.

Ahora estamos en condiciones de completar el diagrama, resolviendo el proceso.

Se utilizará una variable llamada **VALOR** para ingresar cada uno de los datos y otra variable que actúa como acumulador denominada **ACUM**.

Obsérvese que tanto **CONT** como **ACUM** fueron **inicializadas** en cero. No así **VALOR**.

La razón de esto es que tanto **CONT** como **ACUM** utilizan su valor inicial.

Esto puede comprobarse en el desarrollo del proceso. Al ingresar al lazo, se lee un valor de teclado y se lo almacena en **VALOR**. Luego se suma este valor al que ya tiene guardado **ACUM** (**ACUM** se carga con **ACUM+VALOR**).

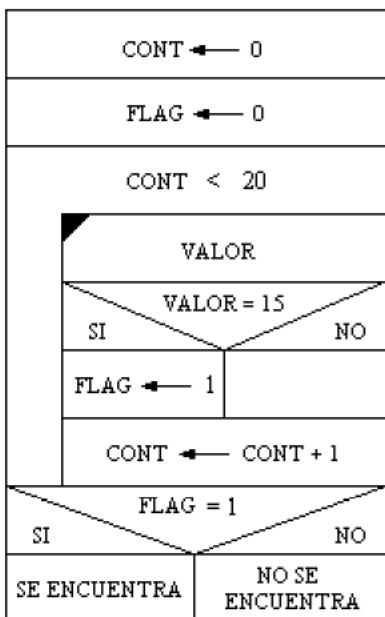
Si **ACUM** empezara conteniendo *cualquier* valor, la suma final sería errónea. Es preciso asegurar que **ACUM** comience valiendo cero.

Sería conveniente en este punto que el lector realice una prueba de escritorio con algunos valores, a fin de comprender claramente el funcionamiento del diagrama. Para esto, anote en un papel los valores que va tomando cada variable, a medida que sigue la ejecución del diagrama.

EJEMPLO 5

Ingresar 20 valores. Indicar si entre ellos se encuentra, o no, el número 15.

En este caso, el diagrama preliminar o primera versión del diagrama de Chapin es similar al del ejemplo anterior.



Solamente cambian el proceso y la presentación de los resultados.

En el presente problema se trata de determinar si el valor 15 está presente entre los datos de entrada.

No importa cuántas veces, solamente se debe determinar si está o no.

Para esta situación se utilizará una variable como *testigo* o *flag*. La denominaremos `FLAG` y su significado será el siguiente: si `FLAG` vale “1” significa que el valor 15 apareció entre los datos, y si vale “0” es que aún no apareció.

Inicialmente, cuando aún no se leyeron datos, es evidente que el 15 no apareció y por lo tanto `FLAG` debe inicializarse en cero.

Por cada dato leído se realiza la comparación con 15 mediante una estructura de selección.

Si resulta igual, el `FLAG` se coloca en “1”. Nótese el bloque vacío en caso de selección por NO.

Una vez finalizado el lazo, solamente puede ocurrir que `FLAG` contenga “1” (significando que al menos un valor fue 15) o contenga “0” (esto sólo ocurrirá si siempre se seleccionó FALSO, debido a que ningún valor fue 15).

Para mostrar el resultado se utiliza otra estructura de selección que pregunta el valor de `FLAG`.

En los dos ejemplos anteriores se presentó un diagrama preliminar o primer paso en la diagramación. Esto es un ejemplo de desarrollo Top Down en el que se creó la estructura general del diagrama dejando para ulteriores desarrollos los bloques internos. Otros ejemplos de esta técnica se presentarán en próximas secciones.

PSEUDOCÓDIGO

El **pseudocódigo** es un lenguaje de especificación de algoritmos diferente a todos los lenguajes de programación y por lo tanto no puede ser ejecutado por la computadora ni reconocido por ningún programa traductor.

El pseudocódigo nació como un lenguaje similar al inglés y era un medio de representar básicamente las estructuras de control de la programación estructurada. Generalmente es de sintaxis similar al lenguaje de programación a utilizar en la codificación final.

Su ventaja es que permite al programador concentrarse en las estructuras de control y despreocuparse de las **reglas de sintaxis** de un lenguaje específico.

Se puede considerar al programa expresado en pseudocódigo como el *borrador* o paso anterior a la codificación final. Este último paso se torna entonces relativamente fácil.

Como ejemplo de programación en pseudocódigo se mostrará el problema del último ejemplo de los diagramas de Chapin.

VARIABLES A USAR (declaracion de variables):

TIPO ENTERO : CONT, FLAG, VALOR

INICIO

```
CARGAR CONT CON 0
CARGAR FLAG CON 0
MIENTRAS CONT < 20 HACER :
    EMPIEZA SECUENCIA
        LEER DATO Y GUARDAR EN VALOR
        SI VALOR = 15 HACER :
            CARGAR FLAG CON 1
        CASO CONTRARIO HACER :
            NADA
        INCREMENTAR CONT
    TERMINA SECUENCIA
    SI FLAG = 1 HACER :
        IMPRIMIR "15 SE ENCUENTRA ENTRE LOS DATOS"
    CASO CONTRARIO HACER :
        IMPRIMIR "15 NO SE ENCUENTRA ENTRE LOS DATOS"
```

FIN

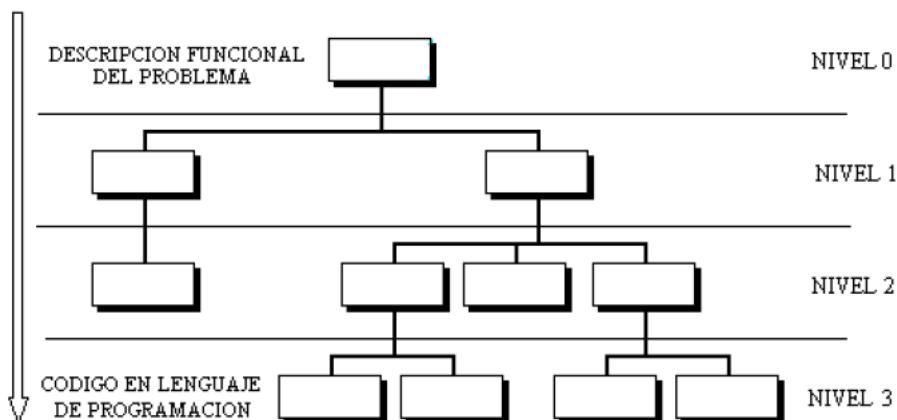
El programa desarrollado precedentemente no se adapta a ningún lenguaje en particular y simplemente trata de mostrar la lógica del programa en un idioma comprensible y simple.

DESARROLLO TOP-DOWN

El concepto de desarrollo Top Down (también conocido como desarrollo Arriba-Abajo, desarrollo Estructurado o desarrollo Descendente) se aplica tanto a la fase de diseño del sistema o aplicación, como a la fase de programación y codificación de los programas.

El desarrollo descendente de programas consiste en construir un programa tratando de descomponerlo en *módulos* (funciones o segmentos), cada uno encargado de realizar un trabajo específico.

Esta organización se va logrando de arriba hacia abajo en forma de árbol y podemos representarla gráficamente como un **diagrama de estructura**.



MODELO DE DIAGRAMA DE ESTRUCTURA

En este modelo se ven las diferentes etapas del desarrollo comenzando con el nivel 0, que representa el “enunciado” o descripción del problema.

METODOLOGÍA DE DISEÑO

- Dividir el problema en subproblemas manejables.
- Cada subproblema se divide a su vez en otros subproblemas.
- El proceso se repite hasta que no se pueda subdividir más.
- La solución de todos los subproblemas constituye la solución global.

Se crea de esta manera una estructura jerárquica de problemas con distintos niveles de refinamiento:

- NIVEL 0: Descripción del problema.
- NIVEL N: Refinamientos sucesivos.

EJEMPLO DE APLICACIÓN

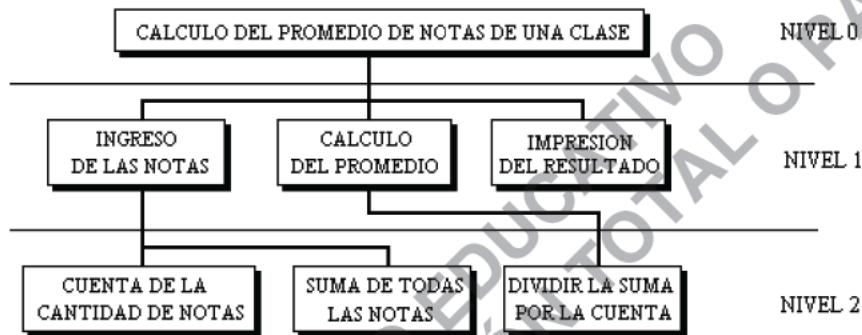
Se desea calcular el promedio de notas de los alumnos de un curso. No se conoce *a priori* la cantidad de alumnos, pero se insertará una nota testigo (podría ser -1) para indicar la finalización del ingreso.

El primer nivel de refinamiento (nivel 0) lo constituye este enunciado. El primer intento de resolución lleva al nivel 1 y se compone de ingreso de datos, cálculo del promedio e impresión del resultado.

Es necesario conocer la suma y la cantidad de notas para el cálculo del promedio. Por lo tanto se agregará al proceso de ingreso la cuenta de los datos y su suma.

La división de estos valores representa un refinamiento del cálculo del promedio.

El desarrollo se muestra en el siguiente esquema:



PROBLEMAS PROPUESTOS

Construir los diagramas de Chapin para los siguientes problemas:

1. Se ingresarán números enteros positivos. Determinar cuáles de éstos son pares. El ingreso finalizará con un número negativo.
2. Se ingresarán 100 números enteros. Sumar los de orden impar (1, 3, 5...) por un lado y los de orden par (2, 4, 6...) por otro. Determinar cuáles proporcionan la mayor suma.
3. Se ingresarán números enteros hasta que se ingrese el 235. Indicar cuántas veces ocurrió el ingreso del número 23.
4. Permitir el ingreso de una clave numérica entera. Finalizar el ingreso solamente cuando la clave ingresada sea 23645.
5. Repetir el problema anterior permitiendo sólo 3 intentos. Luego del tercer intento fallido colocar una advertencia.
6. Determinar si un número entero positivo ingresado por teclado es o no primo (es decir, sólo divisible por sí mismo y por la unidad).
7. Representar los diagramas anteriores en pseudocódigo.

2. FUNDAMENTOS DE C

INTRODUCCIÓN

El lenguaje C fue diseñado en los años sesenta por **Dennis Ritchie** (1941-2011), de los Laboratorios Bell. El propósito era ser el lenguaje del sistema operativo UNIX.

Se trata de un lenguaje desarrollado *por programadores para programadores*. Esto le brinda muchos aspectos deseables, como ser la libertad que se deja al programador. En contrapartida, esta libertad conlleva ciertos riesgos dado que el lenguaje es muy permisivo y puede inducir al programador a cometer errores.

C es un lenguaje de **nivel intermedio** que reúne la potencia de los lenguajes de alto nivel con la versatilidad del bajo nivel. Este aspecto lo hace muy apropiado para aplicaciones de manejo de hardware como las realizadas en Ingeniería Electrónica.

El **compilador** es pequeño y eficiente. El lenguaje tiene, en su definición original, solamente 32 **palabras reservadas**, esto es, el conjunto reducido de palabras con significado específico dentro del lenguaje y reconocidas por el compilador. Esto lo hace relativamente sencillo de aprender y manejar.

La entrada/salida no se considera parte del lenguaje en sí, sino que se suministra a través de **funciones** (subprogramas) de **biblioteca**. La misma política se sigue con cualquier otro tipo complejo de instrucciones.

La gran cantidad de funciones que componen la **biblioteca estándar** suministra la potencia del lenguaje, así como otorga modularidad.

Si bien es un lenguaje antiguo (principios de 1970), está en plena vigencia en aplicaciones de medio y bajo nivel, dando además origen a otros lenguajes basados en él pero orientados a distintos paradigmas como ser C++, Visual C++ y C#. El lenguaje C se utiliza actualmente para desarrollar programas que a su vez se convierten en plataformas para otros lenguajes, como ser sistemas operativos, compiladores, intérpretes, etc. Todas estas son herramientas fundamentales en la informática como la conocemos hoy.

ANSI C

A mediados de los ochenta ya había en el mercado numerosos compiladores C, y muchas aplicaciones escritas originalmente en otros lenguajes, habían sido reescritas en C para aprovechar sus ventajas.

Durante este período muchos fabricantes introdujeron “mejoras” propietarias en el lenguaje, generando de esta forma numerosos dialectos y provocando que un mismo programa no pudiera compilarse en otro compilador. Esto es lo que llamamos **pérdida de la portabilidad**, una característica muy deseable.

El comité estadounidense de estandarización **ANSI** (*American National Standards Institute*) estableció en 1990 especificaciones mínimas para los diversos compiladores creando de esta forma lo que hoy en día llamamos “ANSI C”.

Esta definición tradicional del lenguaje es conocida como C89. En posteriores años se realizaron revisiones y agregados. En la actualidad, las versiones más utilizadas del estándar en la práctica son C99 y C11 (por 2011).

ESTRUCTURA DE UN PROGRAMA C

Todo programa C se basa en **funciones**. Siempre debe estar presente al menos una de ellas que actúa como eje del programa, transformándose en la función principal, y recibiendo de esta manera el nombre de **main**.

La estructura general de un programa C se muestra a continuación :

```
# Comandos al compilador
Prototipo de funciones
Declaración de variables globales
int main ( )
{
    Declaración de variables locales
    Cuerpo del programa
}

< Cuerpo de las funciones propias >
```

Los comandos al compilador o “*comandos al preprocesador*” son directivas a tener en cuenta por el programa compilador en **tiempo de compilación**. Se individualizan por comenzar con el símbolo # (numeral).

Entre estos comandos se encuentran los de inclusión de archivos cabecera (*header*) vinculados a las funciones de biblioteca. Trataremos estos aspectos vinculados con las funciones más adelante.

La **declaración de variables** (tanto **globales** como **locales**) es una indicación que permite al compilador reservar la memoria necesaria para ellas. Recorremos que las variables de nuestro programa no son más que etiquetas para cierta área de la memoria principal (RAM) de la computadora. La declaración consiste en indicar el **nombre de la variable** y su **tipo**.

La estructura del programa principal (**main**) es similar a la de una función y su bloque debe quedar encerrado entre llaves.

Nótese la sangría o **tabulación** utilizada dentro del **bloque**. Esto no es un requerimiento del compilador dado que éste ignora los espacios, sino que es una costumbre que facilita la lectura y comprensión del programa. Su uso hace a un correcto **estilo de programación**.

Es muy importante considerar que el programa es escrito y leído por seres humanos, además de ser compilado y ejecutado. El compilador no necesita (ni puede) comprender el programa: para realizar su tarea sólo necesita que el programador cumpla las reglas de sintaxis del lenguaje. Pero los programadores que deban modificar y comprender el programa sí precisan que el **estilo del código fuente** se adecúe a las **buenas prácticas** para facilitar su comprensión. En síntesis, la correcta tabulación del código fuente hace a una práctica profesional.

PALABRAS RESERVADAS

El estándar ANSI C89 prevé 32 palabras reservadas, es decir, que no pueden ser utilizadas para definiciones propias del usuario.

Las listamos a continuación: auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.

Nótese que, por ejemplo, no es posible definir una variable de nombre auto, ya que es una palabra reservada.

IDENTIFICADORES VÁLIDOS

Un **identificador** (nombre de variable y de función) puede estar compuesto de cualquier combinación de letras (minúsculas y mayúsculas), dígitos y guion bajo. La única restricción es que el primer carácter no puede ser un dígito.

En teoría no se limita la longitud de los identificadores, pero en la práctica los compiladores siempre imponen un límite dada la finitud de los recursos de la computadora. En el compilador Microsoft, por ejemplo, es de 2047 caracteres.

Se diferencia entre mayúsculas y minúsculas. Es decir que una variable cont y una variable Cont serán para el compilador dos variables distintas.

Ejemplos de identificadores válidos: A B2 MAX_VALOR _MINIMO1 Pepe

E inválidos: 4toValor "W" MAY-VALOR MAY VALOR PEPE&LOLA

El lenguaje C es un lenguaje de minúsculas

Todas las palabras reservadas se escriben en minúscula y no es lo mismo un identificador escrito en mayúscula que otro en minúscula.

No respetar lo anterior es una de las fuentes de errores de compilación más frecuentes.

TIPOS DE DATOS

C utiliza cinco palabras reservadas para definir los **tipos de datos** fundamentales. Cada uno de estos puede ir acompañado de una serie de **modificadores**.

Los tipos de datos fundamentales son:

`char int float double void`

Los tipos `char` e `int` corresponden a enteros de longitud 8 y 32 bits respectivamente. Las variables `char` se utilizan frecuentemente para contener el código ASCII de caracteres.

Las variables de tipo `char` NO almacenan caracteres.

Al igual que todas las variables de cualquier tipo, almacenan números binarios.

En el caso particular del tipo `char`, su longitud (8 bits) lo hace adecuado para almacenar números binarios que se correspondan con el código ASCII de representación de algún carácter. De ahí su nombre.

Los tipos `float` y `double` corresponden a números de 32 y 64 bits respectivamente que utilizan la convención de punto flotante estándar IEEE 754 para representar números reales.

El tipo `void` se utiliza para informar que alguna función no retorna nada o indicar algún tipo incierto de datos. No se pueden declarar variables de este tipo.

MODIFICADORES

Los modificadores son opciones que, agregadas al tipo de dato, alteran su tamaño, comportamiento con respecto al signo, tipo de acceso o ubicación en la memoria. Se utilizan de ser necesario al momento de declarar la variable, junto al tipo.

Su listado es el siguiente: `long short signed unsigned register auto extern static const volatile`

Su uso en la declaración de variables es del siguiente modo:

```
modificador tipo nombre_de_variable ;
```

Los modificadores `short` y `long` permiten obtener dos tipos diferentes de longitud de dato. Utilizando `short int` se reservan 2 bytes. En la actualidad, declarar una variable como `int`, `long` ó `long int` genera el mismo resultado: se reservarán 4 bytes de memoria (32 bits).

Las declaraciones `long int` y `long` se interpretan de la misma forma por lo que se suele utilizar la segunda forma omitiendo el “`int`” en la declaración. Nótese que `long` no es un tipo en sí mismo, sino que significa `long int`.

Algo similar ocurre cuando se utilizan los modificadores `signed` y `unsigned` aplicados a enteros (los flotantes siempre son signados). Si se omite el modificador, la variable entera será signada, por lo tanto, es irrelevante el uso de `signed`.

Tanto los modificadores que afectan la longitud del número como las que permiten o eliminan la presencia de valores negativos, tienen incidencia directa en el **rango** de cobertura de la variable.

Recordemos que el rango de representación de números binarios de una longitud de n bits es:

No signado 0 a $2^n - 1$

Signado -2^{n-1} a $2^{n-1} - 1$

De esta manera los diversos tipos con sus modificadores ofrecer las siguientes posibilidades:

`char` entero signado de 8 bits. Puede utilizarse para contener códigos ASCII.

`unsigned char` entero no signado de 8 bits. Puede utilizarse para contener códigos ASCII.

`short int` entero signado de 16 bits.

`unsigned short int` entero no signado de 16 bits.

`long` entero signado de 32 bits.

`unsigned long` entero no signado de 32 bits.

`float` flotante corto.

`double` flotante largo.

`long double` flotante extra largo.

La siguiente tabla representa los tipos descriptos y sus rangos:

Tipo	Longitud (bits)	Memoria (bytes)	Rango de representación
char	8	1	-128 a 127
unsigned char	8	1	0 a 255
short int	16	2	-32768 a 32767
unsigned short int	16	2	0 a 65535
int	32	4	-2147483648 a 2147483647
long int	32	4	-2147483648 a 2147483647
long	32	4	-2147483648 a 2147483647
unsigned long int	32	4	0 a 4294967295
unsigned long	32	4	0 a 4294967295
float	32	4	3.4E-38 a 3.4E+38
double	64	8	1.7E-308 a 1.7E+308
long double	80	10	3.4E-4932 a 1.1E+4932

Existe otro grupo de modificadores que indican la forma en que se almacena y el tipo de acceso a variables de un determinado tipo. Al igual que los anteriores se indican en la declaración, antes del tipo de la variable.

static

El modificador **static** hace que una determinada variable se almacene en una zona de memoria reservada para variables globales (denominada *área estática*), aunque haya sido declarada localmente.

De esta forma adopta el comportamiento global. Este tema será tratado profundamente al estudiar funciones.

auto

Es lo mismo que si no se usara ningún modificador ya que indica que una variable debe ser local, situación que se da por defecto.

volatile

El compilador debe asumir que la variable está relacionada con un dispositivo externo y que puede cambiar de valor en cualquier momento. Esto no significa que su valor cambie espontáneamente y sin motivo, sino que el motivo y ocasión

del cambio no están presentes en el programa. Su comportamiento esta asociado al uso de interrupciones.

register

Es un pedido (más bien una súplica) para que el compilador asigne un **registro del microprocesador** para representar a la variable en lugar de ubicarla en memoria RAM. De esta manera se pretende tener una variable de acceso ultra rápido al no perder tiempo en los accesos a memoria.

extern

La variable se considera declarada en un programa asociado residente en otro archivo. No se le asignará dirección ni espacio de memoria. Se utiliza en los procesos de compilación separada, es decir, cuando el programa está escrito en varios archivos fuente que se compilan por separado y luego se unen para formar el ejecutable final.

const

Provoca que la variable que se está declarando tenga acceso de tipo ROM. Es decir sea una variable de lectura solamente.

Obviamente dejará de comportarse como una variable para transformarse en una **constante**. El único momento en que podrá ser **asignada** (esto es, determinado su valor) es durante su declaración :

```
const tipo identificador = valor ;
```

Ejemplo de declaración de una constante: `const float F1 = 7.5 ;`

FIN DE SENTENCIA

Como se mencionó anteriormente, el compilador ignora los espacios en blanco (salvo su uso como separador de palabras), las tabulaciones y los cambios de línea.

En lo concerniente al compilador, todo el programa podría ser escrito en una única línea. No lo hacemos de esta manera para facilitar su escritura, lectura y su comprensión por parte de seres humanos.

Lo que el compilador necesita para determinar el fin de cada sentencia es el **caracter de fin de sentencia** ; (punto y coma).

Una de las causas mas frecuentes de errores de compilación es la omisión de este carácter. El compilador *cree* que la sentencia actual continúa en la siguiente y detecta un **error de sintaxis**, demasiados **parámetros**, etc.

Por otro lado, la ubicación incorrecta de este carácter puede llevar a errores en **tiempo de ejecución** difíciles de detectar.

OPERADOR DE ASIGNACIÓN

Para asignar valores a las variables se utiliza el **operador de asignación** `=` (signo igual).

Las **asignaciones** se realizan de derecha a izquierda y es imprescindible que el receptor (elemento de la izquierda) sea una variable. De no ser así, el compilador indicará un error de “*left value*” o *valor izquierdo*.

Una variable puede ser asignada con:

- Una constante
- Otra variable
- Una expresión
- El retorno de una función

En los dos últimos casos se resuelve primero la expresión o la función, y *luego* se asigna el valor resultante hacia la izquierda.

Ejemplo:

```
float F , G ;      /* Declaración */  
float H = 4.75 ;    /* Asignación durante la declaración */  
G = 7.5 ;          /* Asignación con constante */  
F = G + 7.23 ;     /* Asignación con expresión */
```

En la última línea se calcula en primer término $G + 7.23$, y su resultado (14.73) se asigna posteriormente a F.

Es importante notar que no se trata de ecuaciones matemáticas. Tomando como ejemplo la última línea, estamos ante una **sentencia** que llamamos **asignación**, cuyo valor derecho es una **expresión** que precisa ser **evaluada**. Luego de realizarse esta evaluación (es decir, la suma) se producirá la copia del valor hacia la izquierda. Es necesario reconocer estas sentencias como una secuencia de acciones precisas, y no como igualdades matemáticas.

COMENTARIOS: Para insertar comentarios en el programa se los debe comenzar con el par `/*` y finalizarlos con el par `*/`

Todo aquello que se encuentre entre estos símbolos será ignorado por el compilador, como si no estuviera escrito en el código.

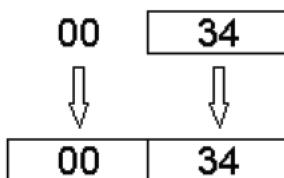
Otra variante es utilizar `//` al inicio del comentario, el cual termina con el cambio de línea. Esta forma proviene del lenguaje C++.

CONVERSIÓN DE TIPOS

¿Qué ocurriría si en operaciones de asignación como las anteriores se involucran variables y valores de tipos diferentes?

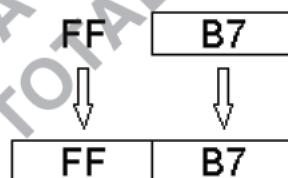
La conversión entre tipos enteros es simple. Los tipos enteros son `char` e `int` con sus modificadores. Si el modificador es `unsigned` todo el número se considera positivo, sin importar el estado de su bit más significativo (*bit de signo*).

En los casos mostrados a continuación se considera que los valores son hexadecimales.

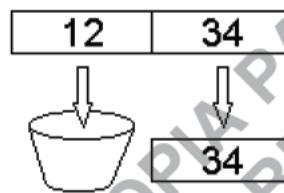


Si se transfiere un valor de menor tamaño a una variable de mayor tamaño, por ejemplo desde un `char` a un `short int`, y el valor es positivo o la variable de origen es no signada, los espacios faltantes en el receptor se completan con ceros.

Pero en caso de ser un valor negativo con variable de origen signada (como ser `B7H`), entonces los espacios incompletos en el receptor se completan con `FFH` para mantener el signo del número.



Nótese que para un número signado de 8 bits `B7H` corresponde al número (-73) decimal, mientras que lo mismo ocurre con el número signado de 16 bits `FFB7H`.



En caso de transferir un valor de mayor tamaño a una variable de menor tamaño, por ejemplo desde un `short int` a un `char`, el excedente simplemente se pierde. Lo más probable es que se trate de un error del programador, pero la filosofía del lenguaje que ya hemos comentado sostiene que "el programador sabe lo que está haciendo", aunque

se ha probado con el tiempo que esto no es cierto. Es importante notar que no habrá advertencias de parte del compilador.

En el caso de conversión entre flotantes, si se parte de un flotante de mayor tamaño hacia otro de menor tamaño, es posible que haya un redondeo de la cifra. Lo opuesto no trae consecuencias.

En el caso de transferir de un flotante a un entero, se perderá la parte fraccionaria y puede llegar a haber un desborde de la parte entera según la magnitud de la misma.

CASTING

El **casting** o *casteo* es una conversión momentánea de tipo. Se utiliza para que una variable o expresión se comporte momentáneamente como si fuera de un tipo diferente.

Frecuentemente el casting es considerado como un operador monario (se aplica a un solo operando o expresión). Su sintaxis es como sigue :

(tipo) expresión

Donde **tipo** es el tipo de dato deseado y **expresión** es la variable o expresión a castear. Ejemplo:

```
float F ;  
F = (float) 7 / 2;
```

El valor 7 se comportará como flotante. Más adelante se verá por qué esta característica es necesaria.

OPERADOR sizeof

sizeof es un operador incorporado en C que se desempeña como un reemplazo que actúa a nivel compilador. Su función es devolver la *cantidad de bytes* que ocupa en la memoria una variable o un tipo de dato.

Su formato es similar al de una función: se coloca entre paréntesis el tipo o variable a evaluar, y el operador retorna el resultado.

Su uso asegura portabilidad de programas en el caso de ejecutarse éstos en procesadores que asignen diferente tamaño a ciertas estructuras de datos. Es decir que utilizaremos **sizeof** cuando necesitemos conocer cuántos bytes ocupa en memoria cierta variable.

En el siguiente ejemplo la variable A se cargará con el valor 4 (tamaño de un **float**, en bytes) y la variable B, con 8 (tamaño de un **double**, en bytes).

```
int A, B;  
float F ;  
A = sizeof(F);  
B = sizeof(double);
```

Dado que el operador **sizeof** se reemplaza en **tiempo de compilación**, las dos últimas líneas del ejemplo presentarán en **tiempo de ejecución** esta forma:

```
A = 4;  
B = 8;
```

OPERADORES ARITMÉTICOS

Operador monario: es un operador que actúa sobre un solo operando.

Operador binario: es un operador que actúa sobre dos operandos para producir un resultado.

La siguiente tabla resume los **operadores aritméticos** de C:

OPERADOR	TIPO	FUNCION
-	MONARIO	SIGNO NEGATIVO
-	BINARIO	RESTA
+	BINARIO	SUMA
*	BINARIO	MULTIPLICACION
/	BINARIO	DIVISION
%	BINARIO	RESTO DE DIVISION ENTERA
++	MONARIO	INCREMENTO
--	MONARIO	DECREMENTO

Los operadores - + * / tienen el mismo significado que en la mayoría de los lenguajes de programación.

El operador % (signo *porcentaje*) no significa cálculo de porcentaje sino *resto de la división entera* o residuo, también llamado *módulo* en programación. Por esta razón no puede ser aplicado a operaciones que involucren flotantes.

Los operadores ++ y -- significan incremento y decremento respectivamente. Esto significa que a la variable involucrada se le sumará o restará la unidad.

Realizar la operación `++A;` es equivalente a `A=A+1;`

Nótese que la variable involucrada cambia su valor. Por esta razón los incrementos y decrementos no pueden ser aplicados a constantes.

Es de destacar también que no se pueden realizar incrementos (ni decrementos) múltiples de la forma `A+++++`, siendo esto un error común en principiantes. El operador que existe en el lenguaje se compone de los dos signos ++.

PRE Y POST INCREMENTOS Y DECREMENTOS

Los operadores de incremento y decremento pueden seguir o preceder al operando. En algunos casos esto es irrelevante pero no así en otros.

Las operaciones `++A`; y `A++`; tienen el mismo efecto que `A=A+1`; debido a que no hay otra acción involucrada en la sentencia. Si así fuera, el incremento se ejecutaría antes o después de las otras acciones, según se trate de un preincremento o un postincremento.

Consideremos los siguientes ejemplos :

//CASO 1: preincremento

```
int A , B ;  
A = 2 ;  
B = ++A ;
```

//CASO 2: postincremento

```
int A , B ;  
A = 2 ;  
B = A++ ;
```

En el caso 1 los valores finales de las variables `A` y `B` son 3 y 3 debido a que en la última sentencia se realiza el (pre)incremento en primer término y la copia de `A` a `B` en segundo término, por lo que el contenido de `A` ya es 3, y ese es el valor que toma `B`.

En el caso 2 los valores finales de `A` y `B` son 3 y 2 respectivamente dado que `A` se (post)incrementa luego de la carga de `B`.

PRECEDENCIA DE OPERADORES ARITMÉTICOS

La **precedencia** es la prioridad que tienen diferentes operaciones cuando se encuentran compartiendo la misma expresión.

Es necesario tener en cuenta la precedencia natural de los operadores a fin de construir las expresiones correctamente. Si fuera necesario alterar la precedencia de operación en una expresión se deben utilizar paréntesis.

PRECEDENCIA	
++	--
- (MONARIO)	
*	/ %
+	-

El cuadro de la izquierda muestra la precedencia de los operadores aritméticos en C.

Esto significa que en una expresión libre de paréntesis, en primera instancia se resolverían los preincrementos, luego el cambio de signo, etc.

Mayor precedencia implica que la operación se hace primero.

Ejercicio: Determine con qué valor se cargaría la variable `F` en cada caso.

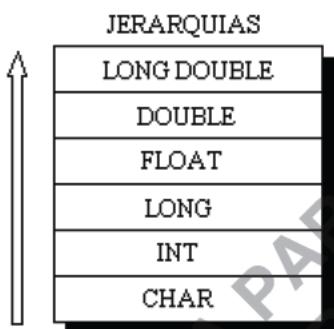
```
int A , B , C , F ;  
A = 3;  
B = 2;  
C = 5;
```

1. $F = B-- + 2 * ++A ;$
2. $F = B + -2 * ++A ;$
3. $F = (B + -C) * ++A ;$
4. $F = C + B \% A ;$
5. $F = (C + B) \% A ;$

Respuestas: 1. (10) 2. (-6) 3. (-12) 4. (7) 5. (1)

JERARQUÍA DE TIPOS

Cuando en una expresión se encuentran valores o variables de diferente tipo, el lenguaje C adopta como tipo del resultado el que corresponde al tipo de mayor jerarquía de la expresión. Es decir que el tipo más jerárquico es el que determina el tipo de la expresión.



Si consideramos la suma entre un valor entero y uno flotante, por ejemplo $(4 + 8.75)$, es esperable que el resultado sea el flotante 12.75 y no el entero 12 .

De esta manera se establece que aquellos tipos de mayor capacidad y/o complejidad tendrán mayor jerarquía y de ser necesario el resultado de una expresión debe ajustarse a ellos.

Es necesario ser cuidadoso con aquellas operaciones que no verifican la *Ley de Cierre o Clausura* con respecto a los tipos con los que están

trabajando, esto es, cuando el tipo del resultado puede ser distinto al tipo de los operandos implicados.

Por ejemplo, una división entre dos números enteros puede dar un resultado no entero, sin embargo el tipo de mayor jerarquía presente en la expresión es entero, y el resultado así lo será en un programa C.

Ejemplo:

```
int A = 18 , B = 5 ;
float F ;
F = A / B ;
```

El valor que toma F es 3.0 . Esto se debe a que el resultado de la división es un entero (3) que es asignado posteriormente a la variable flotante F, habiéndose perdido ya la parte fraccionaria.

La manera de conservar la parte fraccionaria es llevar alguno de los valores enteros a la categoría de flotante, momentáneamente, mediante un *casting*, como se muestra a continuación.

```
int A = 18 , B = 5 ;  
float F ;  
F = (float) A / B ;
```

De esta forma el resultado alojado en F es correctamente 3.6.

OTROS OPERADORES

El lenguaje C posee otros tipos de operadores que serán presentados más adelante, cuando se trate el tema de su aplicación.

Estos operadores pueden ser clasificados en los siguientes grupos:

- Operadores relacionales
- Operadores lógicos
- Operadores a nivel de bits
- Operadores de tipo puntero

ENTRADA/SALIDA DE CONSOLA

Se denomina **consola** al conjunto de pantalla y teclado que constituyen los periféricos tradicionalmente más utilizados para el fin de ingresar y obtener datos de la computadora.

El lenguaje C resuelve las operaciones de entrada y salida mediante **macros y funciones**, y no mediante instrucciones nativas del lenguaje.

Por ahora, adoptaremos un modelo de estas macros y funciones sin entrar en detalle sobre su naturaleza. Estos temas se desarrollarán en detalle en próximas secciones.

Podemos imaginar a una función (y a una macro) como un **subprograma** que toma datos llamados **argumentos**, realiza alguna tarea y retorna un único valor de algún tipo.

En algunos casos, lo importante es el valor retornado y en otros la tarea realizada. Un programa que retorna la raíz cuadrada de un número que se entrega como dato pertenece al primer tipo, mientras que un programa que borre la pantalla y no retorne nada corresponde al segundo tipo.

Tanto las funciones como las macros están asociadas a **archivos cabecera** (*header*) que tienen extensión ".h" y es necesario incluirlos en nuestro programa con el comando del preprocesador `#include`.

FUNCIONES DE SALIDA A PANTALLA

Las funciones más utilizadas son :

- putchar()
- printf()

Ambas están previstas por el estándar ANSI y asociadas a la cabecera stdio.h (*standard input output*).

MACRO PUTCHAR()

Este subprograma es en realidad una macro. Su formato o prototipo es:

```
int putchar (int c) ;
```

Esto significa que recibe como argumento un valor entero (c en este caso) y retorna un valor entero.

El objetivo de este subprograma es colocar en la pantalla (donde se encuentre el cursor actualmente) el carácter cuyo código ASCII se le envía como argumento. El valor retornado es el mismo carácter si tuvo éxito, o bien (-1) si hubo algún error en la ejecución.

Es evidente que lo que interesa de putchar() es la acción que realiza y no el valor retornado por él.

Ejemplo: A continuación varias formas equivalentes de enviar la letra “A” a la pantalla.

```
char LETRA ;
LETRA = 'A';
putchar(LETRA);
putchar('A');
putchar(65);
putchar(0x41);
putchar(0101);
```

Las variantes que se muestran difieren básicamente en el formato del argumento por lo tanto revisaremos estas opciones.

Podemos considerar que un carácter encerrado entre **comillas simples** significa “el código ASCII de...”.

Por lo tanto el programa de ejemplo puede interpretarse como sigue:

- LETRA se carga con el ASCII de la letra “A” y luego se lo entrega a putchar.
- Se entrega a putchar directamente lo que antes se cargó en LETRA. ('A')
- Se entrega a putchar el valor 65 que es el código ASCII decimal de “A”.
- Idem anterior pero en formato hexadecimal (0x...).
- Idem anterior pero en formato octal (0...).

La macro putchar sólo tiene aplicación para el envío de caracteres únicos a pantalla y no resulta cómoda en la mayoría de los casos.

ASCII American Standard Code for Information Interchange

ASCII son las siglas de Código Estándar Americano para Intercambio de Información. Se trata de un código de representación de caracteres, de manera que la información de una computadora sea compatible con la de otra.

Cada combinación del código tiene 8 bits por lo que se pueden representar 256 combinaciones. Entre ellas se encuentran las letras minúsculas y mayúsculas, los números, símbolos y varios caracteres de control para consolas.

EJEMPLO: ENVÍO DE CARACTERES A PANTALLA USANDO PUTCHAR

Este ejemplo servirá para apreciar un programa C completo aunque muy sencillo. Se enviará la palabra “PEPE” a pantalla utilizando putchar().

```
#include <stdio.h>
int main()
{
    putchar('P');
    putchar('E');
    putchar('P');
    putchar('E');
}
```

FUNCIÓN PRINTF()

La función printf() ofrece una impresión en pantalla con formato. Se pueden seleccionar distintos tipos de impresión para los valores, se puede acompañar con texto y posee propiedades de control de impresión.

Su forma o prototipo es el siguiente :

```
int printf (cadena de formato [ , lista de valores] ) ;
```

Los **corchetes** [] en el prototipo indican que la lista de valores a imprimir es opcional y pueden no indicarse, dependiendo del caso.

La **cadena de formato**, que debe escribirse entre comillas dobles, puede contener 3 tipos de elementos:

- Texto
- Caracteres de formato
- Caracteres de control

La cadena de formato es la que indica de qué manera aparecerá la impresión en pantalla. La función de los caracteres de formato es la de indicar cuál será la apariencia de cada uno de los elementos que constituyen la lista de valores.

Esta función retorna un entero con fines de verificación. Este valor representa la cantidad de caracteres que se enviaron a pantalla si todo funcionó correctamente, ó (-1) si ocurrió algún error.

EJEMPLO: HOLA MUNDO

Se mostrará el uso de `printf()` utilizando solamente texto en la cadena de formato, sin lista de valores, para imprimir la frase “Hola Mundo” en pantalla, una tradición en la enseñanza de lenguajes de programación.

Nótense las comillas dobles de la cadena de formato y compárese con el ejemplo anterior.

```
#include <stdio.h>
int main()
{
    printf("Hola Mundo");
}
```

CARACTERES DE FORMATO EN PRINTF

Los **caracteres de formato** tienen la función de indicar, dentro de la cadena de control, qué formato de impresión debe utilizarse para los elementos de la lista de valores. Generalmente hay tantos caracteres de formato en la cadena como valores en la lista de valores.

Los caracteres de formato comienzan con el **carácter de escape %** (signo porcentaje).

Si se desea incluir en la cadena de formato, como texto, al carácter “%”, se le debe anteponer otro carácter “%” a fin de evitar que se lo tome erróneamente por un carácter de formato.

TIPO	VALOR	FORMATO
%d	ENTERO	DECIMAL CON SIGNO
%u	ENTERO	DECIMAL SIN SIGNO
%o	ENTERO	OCTAL
%x	ENTERO	HEXADECIMAL CON MINUSCULAS
%X	ENTERO	HEXADECIMAL CON MAYUSCULAS
%ld	LONG	DECIMAL LONG CON SIGNO
%c	CARACTER	CARACTER
%f	REAL	[-]dddd.dddd
%e	REAL	[-]d. dddd e[+ / -]ddd
%E	REAL	[-]d. dddd E[+ / -]ddd
%g	REAL	%f o %e SEGUN VALOR
%G	REAL	%f o %E SEGUN VALOR
%p	PUNTERO	DIRECCION EN HEXADECIMAL
%s	PUNTERO	STRING
%%	NINGUNO	CARACTER %

CARACTERES DE FORMATO

Los formatos octal y hexadecimal representan de esa forma el contenido binario de la variable. Debido a esto carecen de signo, es decir, se los presenta siempre como positivos.

Existe una serie de modificadores de formato que se colocan entre el "%" y el carácter que completa el formato, y que alteran la forma de impresión.

MODIFICADORES DE FORMATO

- N Donde N es un número. Cantidad mínima de espacios asignada.
- N Justificación a la izquierda.
- .N Cantidad de dígitos luego del punto decimal.
- + Precede la posición con el signo (- o +)
- # Precede un número octal con "o" y uno hexadecimal con "OX".
- 1 Indica un número **long**. Debe acompañar a d, u, o, x ó X.
- 0. Precede la posición con ceros.

Por ejemplo, **%5d** indica que el valor entero ocupará un mínimo de 5 espacios, mientras que **%-5d** indica que el valor entero ocupará como mínimo 5 espacios

pero comenzando a ocuparlos desde la izquierda. Esto sirve para mostrar en pantalla números tabulados en forma de tabla.

CARACTERES DE CONTROL

Los **caracteres de control**, al ser enviados a la salida estándar (por defecto, nuestra pantalla), ejecutan una determinada acción.

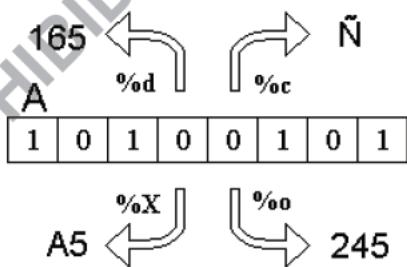
A fin de reconocerlos comienzan con el **carácter de escape** \ (contrabarra o barra invertida).

En caso de ser necesario enviar como cadena de texto para mostrar en pantalla los caracteres contrabarra, comilla simple o doble, se deberá anteponerles el carácter de escape (\a la contrabarra).

TIPO	EFFECTO
\a	ALERTA SONORA
\b	ESPACIO ATRAS
\f	SALTO DE PAGINA
\n	SALTO DE LINEA
\r	RETORNO DE CARRO
\t	TABULACION HORIZONTAL
\v	TABULACION VERTICAL
\\\	CONTRABARRA
'	COMILLA SIMPLE
"	COMILLA DOBLE
\o	CONSTANTE OCTAL
\x	CONSTANTE HEXADECIMAL

CARACTERES DE CONTROL

EJEMPLO: FORMATOS NUMÉRICOS



Se utilizará la función printf() para mostrar el contenido de la variable de tipo unsigned char A. El esquema de la izquierda muestra el verdadero contenido (binario) de dicha variable y las diferentes interpretaciones que se obtienen mediante los caracteres de formato.

```
#include <stdio.h>
int main()
{
    unsigned char A ;
    A = 'Ñ' ;
    printf("\n\n %10c %10d %10o %10X", A, A, A, A);
}
```

EJEMPLO: FORMATOS DE IMPRESIÓN

En este ejemplo se realizarán tres operaciones sobre números enteros almacenados en las variables A y B: una división entera, el residuo de dicha división y una división en flotante.

Se utilizará la función printf() incluyendo texto, caracteres de control para cambio de línea, tabulación horizontal y caracteres de formato.

```
#include <stdio.h>
int main()
{
    int A , B ;
    A = 130 ;
    B = 7 ;

    printf("\nDivisión entera \t %d / %d = %d", A, B, A/B );
    printf("\nResto de División entera \t %d %% %d = %d", A, B, A%B );
    printf("\nDivisión real \t\t %d / %d = %.2f", A, B, (float) A/B );
}
```

La impresión en pantalla arrojó :

División entera	$130 / 7 = 18$
Resto de división entera	$130 \% 7 = 4$
División real	$130 / 7 = 18.57$

Nótense los siguientes aspectos:

- El uso del doble carácter **%%** a fin de obtener la impresión de un **%**
- El uso del modificador **.2** dentro del formato flotante a fin de obtener 2 decimales.
- La inclusión de expresiones a calcular dentro de la lista de valores.
- El uso del casting en la tercera impresión para obtener la división flotante.

FUNCIONES DE ENTRADA DE TECLADO

Existen diversas funciones que realizan el ingreso de datos desde teclado. Nos limitaremos en esta publicación a `getchar()` y `scanf()`.

MACRO GETCHAR()

La macro `getchar()` está definida en la cabecera `stdio.h` mencionada anteriormente.

Su función es obtener caracteres de la entrada estándar, asociada por defecto al teclado. Esta entrada, llamada `stdin` (*standard input*), es un buffer de entrada de líneas, es decir, almacena secuencia de bytes que deben finalizar con un <Enter>.

El efecto visible de esta situación es que el programa se detiene a la espera del ingreso de un carácter desde teclado, pero éste recién es tomado por la macro cuando hay un <Enter> en el buffer.

Su formato o prototipo es:

```
int getchar(void);
```

En condiciones normales el entero retornado tiene sus bytes más significativos en cero y su byte menos significativo con el ASCII del carácter ingresado, por lo que es correcto tomar este valor en una variable de tipo `char` o `unsigned char`.

EJEMPLO: USO DE GETCHAR

Se realizará un programa que permita el ingreso de un carácter por teclado y muestre en pantalla su código ASCII.

```
#include <stdio.h>
int main()
{
    unsigned char A ;
    printf("Carácter = ") ;
    A = getchar() ;
    printf("ASCII = %d", A );
}
```

FUNCIÓN SCANF()

La función `scanf()` está también asociada a la cabecera `stdio.h` y su propósito es el ingreso de valores de diferente formato desde la entrada estándar.

Al igual que la macro anterior, requiere la presencia de <Enter> en el buffer. Los tipos de los valores a ingresar deben coincidir con los expresados en la cadena de formato.

La forma de la función es:

```
int scanf(cadena de formato, lista de direcciones de memoria);
```

La tarea de `scanf()` es tomar del buffer de teclado los valores y colocarlos en las **direcciones de memoria** indicadas en la lista de direcciones.

Hasta ahora hemos declarado, asignado y manejado variables, pero nunca nos preguntamos en qué lugar de la memoria estaban alojadas. No hemos tenido necesidad de conocer dichas **direcciones**. De hecho, esa es una gran tarea que resuelve por nosotros el compilador, esto es, la conversión de los nombres de variables a direcciones de memoria y viceversa. Esta es la razón por la cual trabajamos en C y no en un lenguaje de más bajo nivel como Assembler: para no tener que lidiar con estas direcciones directamente.

El problema de desconocer las direcciones de las variables se resuelve a través del operador `&` (signo *ampersand*).

Este es un operador monario de tipo **puntero** que retorna la dirección de la variable `cada` como operando. Trabajaremos con punteros más adelante.

De esta manera, la forma de leer un valor entero del teclado y colocarlo dentro del área de memoria que corresponde a la variable A sería:

```
scanf("%d", &A);
```

Podemos leer al signo `&` como “*la dirección de*”, de manera que lo que estamos enviando a `scanf` es “*la dirección de la variable A*” (y no su contenido).

EJEMPLO: USO DE SCANF, SUPERFICIE DEL TRIÁNGULO

Se ingresarán desde teclado los valores flotantes correspondientes a la base y la altura de un triángulo mediante `scanf()`.

Posteriormente se calculará la superficie y se informará el resultado en pantalla.

```
#include <stdio.h>
int main()
{
    float BASE , ALTURA , SUPERFICIE ;

    printf("BASE = ") ;
    scanf("%f" , &BASE);
    printf("ALTURA = ") ;
    scanf("%f" , &ALTURA);

    SUPERFICIE = BASE * ALTURA / 2 ;
    printf("SUPERFICIE = %.2f", SUPERFICIE );
}
```

En el ejemplo anterior se utilizó un `scanf()` por cada valor a ingresar. Es posible ingresar varios valores en un mismo `scanf()` utilizando **separadores**.

El separador utilizado más frecuentemente es el “blanco”. Debe destacarse que C considera “blancos” al espacio en blanco generado con la barra espaciadora (uno o más), el espacio de tabulación (tecla TAB) y el cambio de línea. Por ejemplo, la sentencia:

```
scanf("%d %d %d", &A, &B, &C);
```

Admite como formatos de entrada válidos:

```
45 <Enter>  
56 <Enter>  
74 <Enter>
```

```
45 56 74 <Enter>
```

Si no se respetan los separadores acordados, el ingreso se aborta a partir de la ocurrencia de la primer anomalía.

De ingresar lo siguiente (obsérvese la barra ubicada incorrectamente) :

```
45 / 56 74 <Enter>
```

Los valores asignados serían :

A = 45

B = *valor anterior*

C = *valor anterior*

Las variables B y C mantienen su valor anterior debido a que se abortó la carga por incumplirse el formato solicitado. Sin embargo, el valor asignado a la variable A es correcto.

EJEMPLO: USO DE SCNF CON SEPARADORES

Se ingresará la hora con formato “*hora:minutos:segundos*” utilizando `scanf()`. El separador utilizado es : (signo *dos puntos*).

El programa informará la cantidad de segundos transcurridos desde la hora 00:00:00.

```
#include <stdio.h>  
int main()  
{  
    int H , M , S , TIEMPO ;  
    printf("Ingrese la hora con formato HH:MM:SS \n") ;  
    scanf("%d:%d:%d", &H, &M, &S) ;  
  
    TIEMPO = H * 3600 + M * 60 + S ;  
    printf("\n\nEl tiempo fue %ld segundos", TIEMPO);  
}
```

Obsérvese que `scanf` no puede utilizarse para enviar caracteres a la pantalla. Para realizar las indicaciones al usuario se debe utilizar previamente `printf`. Es una confusión muy habitual en los principiantes equivocarse en las cadenas de formato de este modo.

EJEMPLO: VERIFICACION DE FALLA DE SEPARADORES

El valor returnedo por `scanf()` es la cantidad de valores que se transfirieron correctamente a las variables, es decir, a sus posiciones en memoria.

Si se esperan tres valores de ingreso, el valor returnedo debe ser 3. Si no lo es, significa que debería repetirse el ingreso hasta que el usuario haga coincidir los separadores ingresados con los esperados por `scanf()`.

En este ejemplo se realiza un ingreso similar al del ejemplo anterior pero se informa el valor returnedo por `scanf()`.

Se sugiere intentar diversos ingresos con el fin de verificar las fallas.

```
#include <stdio.h>
int main()
{
    int H , M , S , NUM ;
    printf("Ingrese la hora con formato HH:MM:SS \n") ;
    NUM = scanf("%d:%d:%d", &H, &M, &S);

    printf("\n\nH = %d\nM = %d\nS = %d", H , M , S);

    printf("\n\nValores ingresados correctamente = %d", NUM);
}
```

Para el ingreso: 16:45-36

Arrojó como resultado :

H = 16
M = 45
S = 4200688

Valores ingresados correctamente = 2

El valor de la variable `S` es claramente un **valor basura**, que se encontraba en la memoria en el lugar asignado a la variable de nuestro programa. Esta posición de memoria no fue modificada por `scanf` dado que abortó la operación por el formato incorrecto. Por esta razón encontramos un valor extraño.

PROBLEMAS PROPUESTOS

1. Permitir el ingreso del radio (flotante) e imprimir en pantalla la longitud de la circunferencia y la superficie del círculo correspondiente.
2. Ingresar 3 valores enteros y calcular su promedio.
3. Realizar un programa que permita el ingreso de las diagonales de un rombo y muestre el valor de su superficie.
4. Realizar un programa que permita el ingreso de un valor de temperatura y muestre los valores equivalentes en las tres escalas (Celsius, Farenheit y Kelvin). Dado que no se sabe en qué escala se ingresó el valor, deberán contemplarse los 3 casos.
5. Permitir el ingreso de la superficie de un círculo y determinar su diámetro utilizando la función `sqrt()`, que retorna la raíz cuadrada de un flotante y está asociada a la cabecera `math.h`.
6. Permitir el ingreso de 5 letras que componen una palabra. Al finalizar mostrar en pantalla la palabra formada por los códigos ASCII de cada letra a los que se le sumó el valor 5.
7. Realice un programa que dibuje en pantalla un triángulo retángulo, utilizando asteriscos, de esta forma:

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

Permitir que el usuario ingrese el tamaño de los catetos (ingresar un sólo número, dado que son iguales).

8. Repita el programa anterior pero dibujando el triángulo al revés:

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

9. Realice un programa para ingresar el total de una factura (`float`) y el porcentaje a descontar (otro `float`).

Mostrar en pantalla el precio final. Por ejemplo, para una factura de \$120 y un descuento de 8.8% el precio final es \$109.44

3. CONTROL DE FLUJO

SECUENCIA NATURAL

Los programas que hemos considerado hasta ahora presentan una secuencia de seguimiento lineal y descendente. No nos referimos al seguimiento del diagrama de Chapin, que siempre es descendente, sino al seguimiento de **instrucciones** en memoria que sigue el procesador. Recordemos que al ejecutar un programa, el sistema operativo carga las instrucciones que lo componen en la memoria RAM y, una vez hecho esto, se comienza su ejecución desde la primera de ellas.

El microprocesador sigue la secuencia de instrucciones de un programa auxiliándose con el **registro Contador de Programa** (PC), también llamado *Program Counter* o Registro de Instrucción.

Este registro almacena la dirección de memoria donde está alojada la **próxima** instrucción que debe ser ejecutada. Esto significa que cuando se está ejecutando la instrucción actual, dicho registro, ya “apunta” a la siguiente instrucción.

El proceso es el siguiente: en el momento de cargarse en el procesador y analizarse la instrucción actual (y antes de ejecutarla) se incrementa el contenido del PC en la cantidad de bytes correspondientes al largo de dicha instrucción (no todas las instrucciones de máquina tienen la misma longitud). De esta manera el PC queda apuntando a la siguiente instrucción, ubicada en la posición de memoria L bytes mayor que la de la instrucción actual, siendo L la longitud de esta instrucción.

Nótese que se está presuponiendo que las instrucciones se ubicarán en memoria RAM una a continuación de la otra.

De esta manera lo *entiende* el procesador. Este seguimiento, por lo tanto, constituye la **secuencia natural** de instrucciones en un programa. Toda alteración de este seguimiento constituye una ruptura de la secuencia natural que se denomina “*salto*”.

Los saltos se realizan cargando una determinada dirección en el registro Contador de Programa y pueden ser de dos tipos:

- Saltos incondicionales
- Saltos condicionales

SALTOS INCONDICIONALES

Esta es una instrucción que simplemente carga un nuevo valor en el Contador de Programa, con lo que se altera la secuencia natural debido a que la instrucción a ejecutar no será la que esté a continuación en memoria sino otra en alguna otra dirección.

La manera de implementarlo en C es mediante una sentencia `goto` (*ir a*) y un rótulo que indique a qué lugar del programa se realiza el salto.

La programación estructurada desaconseja fuertemente el uso de `goto`. Recorremos que este paradigma es superador de la metodología anterior, que desembocaba en los programas “tallarín” debido, justamente, a la cantidad de saltos en el diagrama. Sin embargo, si el beneficio de su uso supera sus inconvenientes no hay problema en hacerlo.

Salvo algún ejemplo de aplicación, no se usará `goto` en esta publicación.

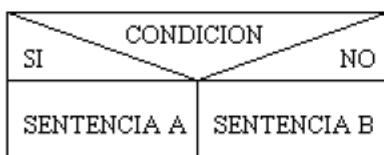
SALTOS CONDICIONALES

Son similares al anterior con la diferencia de que el salto se produce en caso de cumplirse una condición. En caso contrario se continúa con la secuencia natural.

Las rupturas de secuencia tanto condicionales como incondicionales permiten implementar en los programas los bloques de toma de **decisión** e **iteración**. El lenguaje estructurado se encarga de respetar las reglas de la estructuración en estos bloques básicos que, como se vio, utilizan a nivel de lenguaje de máquina, instrucciones no estructuradas. En otras palabras, el lenguaje estructurado (C) es compilado y convertido en una secuencia de instrucciones no estructuradas (lenguaje de máquina).

Dado que en los bloques de toma de decisión e iteración se pueden seguir caminos alternativos, a las sentencias involucradas se las denomina **sentencias de control de flujo**.

TOMA DE DECISIÓN



La toma de decisión se resuelve con la sentencia **if-else**

Su formato es :

```
if (condición) sentencia A ;  
[ else sentencia B ; ]
```

Donde los corchetes indican que las líneas asociadas son opcionales. Recordemos que las sentencias pueden ser simples, nulas o compuestas. Las sentencias compuestas se construyen encerrándolas entre **llaves** { }.

Se recomienda respetar las tabulaciones a fin de poner de manifiesto cuáles sentencias están incluidas dentro de cada bloque.

El diagrama anterior adoptaría en C la forma:

```
if ( condición )
    Sentencia_A ;
else
    Sentencia_B ;
```

Nótese la ausencia de llaves, ya que ambos bloques tienen una sola sentencia. En caso contrario es obligatorio el agrupamiento con llaves.

EJEMPLOS

A continuación se presentarán algunos ejemplos de pseudocodificación de diagramas que representen tomas de decisión no anidadas.

1. Como se mencionó anteriormente, la presencia de la sección correspondiente al `else` es opcional. En este caso, si se cumple la condición se ejecuta la sentencia A, y en caso contrario no se realiza ninguna acción.



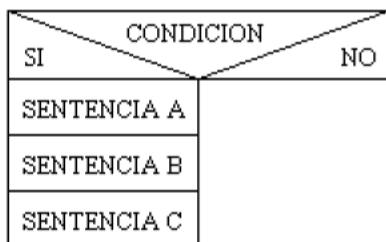
```
if (condición)
    Sentencia_A ;
```

Otra variante del caso anterior sería:

```
if (condición)
    Sentencia_A ;
else ;
```

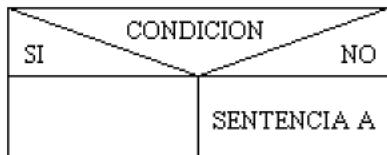
Nótese que la presencia del punto y coma detrás del `else` cierra su bloque, indicando una **sentencia nula**.

2. En este caso se representa un bloque de sentencias múltiples en el “sí”. La estructura de pseudocódigo difiere de la anterior en la presencia de las llaves.



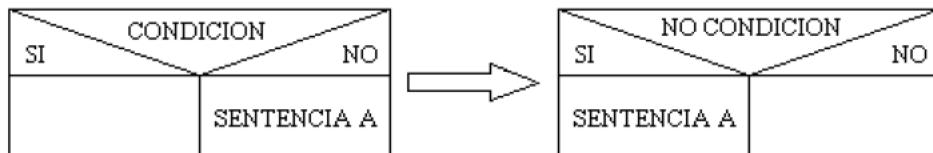
```
if (condición) {
    Sentencia_A ;
    Sentencia_B ;
    Sentencia_C ;
}
```

3. Una estructura poco frecuente es aquella que posee sentencias en el bloque del “no” y no las posee en el del “sí”. El diagrama y la pseudocodificación correspondiente se muestran a continuación:



```
if (condición) ;
else
    Sentencia_A ;
```

Esta estructura puede alterarse **negando** la condición, es decir, utilizando la condición inversa (*negación lógica*). Esta situación se ilustra a continuación:



4. La estructura general de sentencias compuestas en ambos bloques y su codificación en pseudocódigo se muestran a continuación:

CONDICION	
SI	NO
SENTENCIA A	SENTENCIA E
SENTENCIA B	
SENTENCIA C	SENTENCIA F

```

if (condición) {
    Sentencia_A ;
    Sentencia_B ;
    Sentencia_C ;
}
else {
    Sentencia_E ;
    Sentencia_F ;
}

```

CONDICIONES

Las condiciones evaluadas tanto en las tomas de decisión como en las iteraciones, son expresiones que deberán arrojar como resultado “verdadero” o “falso”. Debido a esto, reciben la denominación de **condiciones Booleanas**.²

Es posible que la expresión utilizada en la condición, ya sea por error o intencionalmente, no tenga “la forma” de una condición booleana. De todas maneras el procesador evaluará dicha expresión en términos lógicos, obteniendo como resultado *verdadero* o *falso*.

Generalmente, las expresiones booleanas se valen de los **operadores relacionales** para su construcción.

OPERADORES RELACIONALES

Los operadores relacionales son aquellos que permiten determinar la relación entre dos variables o valores.

OPERADOR	FUNCION
>	MAYOR QUE
>=	MAYOR O IGUAL QUE
<	MENOR QUE
<=	MENOR O IGUAL QUE
==	IGUAL QUE
!=	DISTINTO A

² George Boole (1815-1864), matemático inglés, desarrolló un álgebra proposicional cuya expresión más simple maneja sólo estados: verdadero y falso.

Todos ellos son **operadores binarios**. Por lo tanto, no es posible realizar una comparación entre más de 2 valores utilizando solamente operadores relacionales.

El resultado de una operación relacional es un valor booleano, es decir, *verdadero* o *falso*.

Si se asigna a una variable entera el resultado de una operación relacional, los valores asignados serán “0” o “1”, siendo equivalentes a *falso* y *verdadero* respectivamente en el lenguaje C.

Ejemplo:

```
int F ;  
F = 4 > 2 ;           /* F toma el valor 1 */  
F = 4 == 2 ;          /* F toma el valor 0 */
```

Los operadores relacionales no representan la única forma de obtener valores booleanos. El lenguaje C considera que todo valor numérico distinto de cero es VERDADERO, mientras que cero representa FALSO.

De esta manera, toda vez que se debe tomar una decisión lógica se puede evaluar una expresión numérica o bien una relacional.

CONDICIÓN COMPUESTA

Cuando una única expresión booleana esta formada por más de una condición, se dice que es una **condición compuesta**.

Para construirla se recurre a la utilización de los **operadores lógicos**.

OPERADORES LÓGICOS

Los operadores lógicos se aplican sobre operandos que representan expresiones o **estados lógicos**, y arrojan como resultado un **valor lógico**.

OPERADOR	TIPO	FUNCION
!	MONARIO	INVERSOR LOGICO
&&	BINARIO	AND
	BINARIO	OR

INVERSOR LÓGICO

El inversor lógico es un operador monario que invierte el resultado de la evaluación de una expresión o valor lógico. Se escribe como un signo de exclamación: !

Si A es una proposición lógica, el siguiente cuadro (llamado **tabla de verdad**) resume la acción del inversor:

A	!A
FALSO	VERDADERO
VERDADERO	FALSO

Inversor Lógico

Es justamente este operador el que se utiliza para *negar* una condición.

OPERADOR AND LÓGICO

Es un operador binario que vincula dos expresiones o valores lógicos. Para que su resultado sea verdadero, es necesario que *ambos* operandos o expresiones lógicas sean verdaderas de manera simultánea, como lo muestra su tabla de verdad.

El operador se construye con dos signos *ampersand*: &&

A	B	A && B
FALSO	FALSO	FALSO
FALSO	VERDADERO	FALSO
VERDADERO	FALSO	FALSO
VERDADERO	VERDADERO	VERDADERO

Operador AND lógico

OPERADOR OR LÓGICO

Este operador binario se representa sintácticamente con el doble *pipe* ó tubería (||, código ASCII 124). En este caso, para que su resultado sea verdadero, es necesario que *al menos uno* de sus operandos o expresiones lógicas sean verdaderas, como lo muestra la tabla de verdad que sigue:

A	B	$A \parallel B$
FALSO	FALSO	FALSO
FALSO	VERDADERO	VERDADERO
VERDADERO	FALSO	VERDADERO
VERDADERO	VERDADERO	VERDADERO

Operador OR lógico

DECISIONES ANIDADAS

Los bloques de una toma de decisión pueden contener estructuras o combinaciones de estructuras. Por lo tanto también pueden contener otras tomas de decisión. A este caso se lo denomina **decisiones anidadas**.

Una de estas situaciones lo representa el “escalonador” que se tratará más adelante.

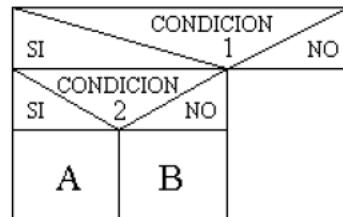
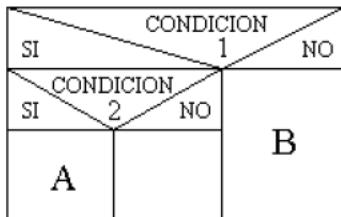
Es necesario ser muy cuidadoso con las aperturas y cierres de bloques (con las llaves { }) de cada toma de decisión a fin de evitar errores.

Veamos el caso del siguiente pseudocódigo :

```
if(condición 1)
    if (condición 2)
        Sentencia_A ;
else
    Sentencia_B ;
```

Se observa que la `Sentencia_A` se ejecuta en caso de cumplirse la condición 1 **y también** la condición 2. Pero, ¿cuándo se ejecuta la `Sentencia_B`?

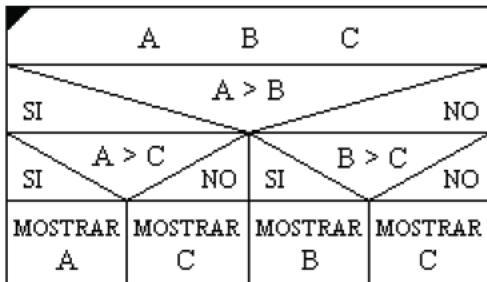
Expresado de otra manera, ¿a cuál de los siguientes diagramas de Chapin corresponde el pseudocódigo anterior?



EJEMPLO: EL MAYOR DE 3 NÚMEROS

Veremos en este ejemplo algunas formas de resolver la determinación del mayor entre tres números enteros utilizando las herramientas recientemente adquiridas.

Se ingresarán tres números enteros y se pide indicar cuál de ellos es el mayor.



VERSIÓN I

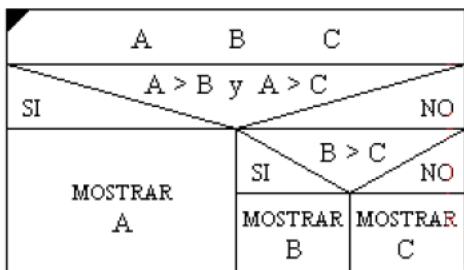
Se ingresarán los tres valores enteros al inicio del programa.

Luego se determinará si el primero de ellos es mayor que el segundo, a fin de descartar uno de los dos.

Esto abre dos caminos. Siguiendo cualquiera de los dos se realiza la comparación con el tercer valor.

```
#include <stdio.h>

int main ()
{
    int A , B , C ;
    printf("Ingrese tres números: ");
    scanf("%d %d %d", &A, &B, &C) ;
    if ( A>B )
        if ( A>C )
            printf ("\nEl mayor es %d" , A ) ;
        else
            printf ("\nEl mayor es %d" , C ) ;
    else
        if ( B>C )
            printf ("\nEl mayor es %d" , B ) ;
        else
            printf ("\nEl mayor es %d" , C ) ;
}
```



VERSIÓN II

Este es un caso similar al anterior excepto que el primer if presenta una condición compuesta que permite determinar en una sola “pregunta” si la variable A contiene el mayor valor.

En caso contrario sólo se debe discernir entre B y C con otra toma de decisión.

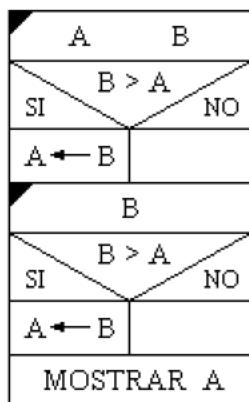
```

#include <stdio.h>

int main ()
{
    int A , B , C ;
    printf("Ingrese tres números: ");
    scanf("%d %d %d", &A, &B, &C ) ;

    if ( A>B && A>C )
        printf ("\nEl mayor es %d" , A ) ;
    else
        if ( B>C )
            printf ("\nEl mayor es %d" , B ) ;
        else
            printf ("\nEl mayor es %d" , C ) ;
}

```



VERSIÓN III

En esta versión se utiliza la variable A para almacenar el mayor valor parcial. Se ahorra el uso de la variable C.

Nótese que se compara entre los valores de A y B. Si ésta última variable contiene un valor mayor que A, se transfiere a la variable A.

Como sea, en esta última queda almacenado siempre el mayor valor hasta el momento.

El nuevo valor de teclado se ingresa nuevamente en la variable B y se repite el proceso.

```

#include <stdio.h>

int main ()
{
    int A , B ;
    printf("Ingrese tres números: ");
    scanf("%d %d", &A, &B) ;

    if ( B>A )
        A = B ;

    scanf("%d", &B) ;

    if ( B>A )
        A = B ;
    printf ("\nEl mayor es %d" , A ) ;
}

```

ESCALONADOR

Como se mencionó en el apartado de diagramación, el **escalonador** es un caso de tomas de decisión anidadas y sucesivas en la que cada nuevo nivel de decisión se ubica en la salida por “no” del nivel anterior.

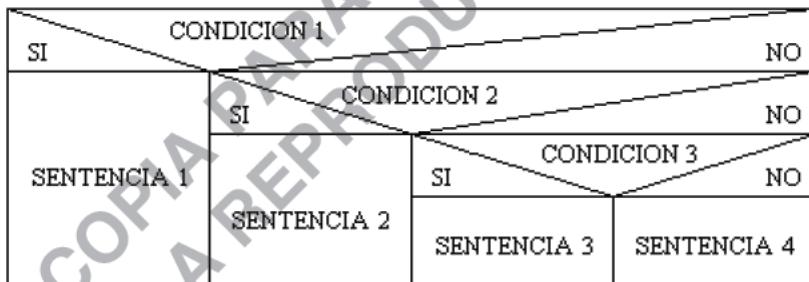


Diagrama esquemático de un escalonador

Su pseudocódigo se representa como :

```

if (condición 1)
    Sentencia_1 ;
else if (condición 2)
    Sentencia_2 ;
else if (condición 3)
    Sentencia_3 ;
else
    Sentencia_4 ;

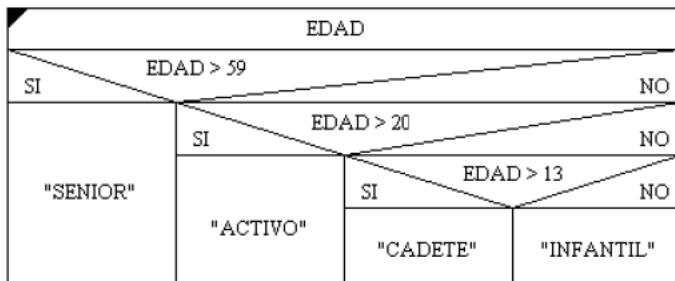
```

Recordemos que cada sentencia puede ser nula, simple o compuesta. El escalonador se aplica en el siguiente ejemplo.

EJEMPLO: USO DEL ESCALONADOR

Se desea clasificar a los socios de un club por edades de la siguiente manera: Infantil (menor de 14 años), Cadete (entre 14 y 20 años), Activo (entre 21 y 59 años) y Senior (60 o más años).

Se pide ingresar la edad e imprimir la categoría correspondiente.



```
#include <stdio.h>

int main ()
{
    int EDAD ;
    printf ("\n EDAD DEL SOCIO =  " ) ;
    scanf( "%d", &EDAD ) ;

    if ( EDAD > 59 )
        printf ("\n SOCIO SENIOR " ) ;
    else
        if ( EDAD > 20 )
            printf ("\n SOCIO ACTIVO " ) ;
        else
            if ( EDAD > 13 )
                printf ("\n SOCIO CADETE " ) ;
            else
                printf ("\n SOCIO INFANTIL " ) ;
}
```

Se pueden omitir las llaves porque cada bloque solo tiene una sentencia. Más allá de esto, puede ser beneficioso colocar las llaves siempre, para evitar problemas a futuro: si agregamos una sentencia a un bloque, y olvidamos agregar las llaves ¡en realidad estaremos colocando la sentencia *fuera* del bloque!

SELECTOR SWITCH CASE

Existe un caso particular del escalonador que es el selector `switch-case`. En éste se compara el valor de una variable contra una serie de constantes a fin de determinar el flujo a seguir.

El `switch-case` es un escalonador con las siguientes restricciones :

- Las condiciones se resuelven *por igualdad* solamente.
- La variable a comparar debe ser de un tipo enumerable (int, char, etc.)
- Los valores a comparar con esta variable deben ser *constantes* del programa.

Estas pautas restringen seriamente el rango de aplicación del `switch-case`, limitándolo prácticamente a implementación de menús.

Su diagrama y pseudocódigo son:

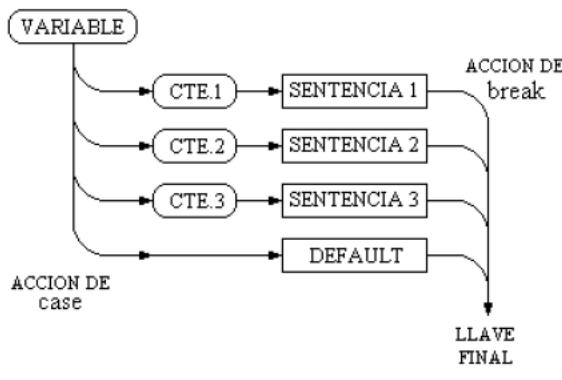
VALOR CONDICION			
VALOR 1	VALOR 2	VALOR 3	DEFAULT
SENTENCIA 1	SENTENCIA 2	SENTENCIA 3	SENTENCIA DEFAULT

```
switch ( VARIABLE_DE_SELECCION ) {  
    case CONSTANTE 1 : SENTENCIA 1 ;  
        break ;  
    case CONSTANTE 2 : SENTENCIA 2 ;  
        break ;  
    case CONSTANTE 3 : SENTENCIA 3 ;  
        break ;  
    default : SENTENCIA_POR_DEFAULT ;  
}  
}
```

SENTENCIA BREAK

La sentencia `break` provoca un salto al final del bloque de mayor anidamiento. Es el equivalente a un `goto` al siguiente cierre de llaves.

Dicha sentencia, de naturaleza no estructurada, permite *estructurar* el `switch-case`. Obsérvese en la figura que se tiene una sola entrada y una sola salida del conjunto.



La sentencia `case` permite la comparación con cada constante y el desvío del flujo cuando corresponda.

Si se ejecuta el flujo de sentencia 1, de todas maneras se ejecutarán las restantes sentencias.

La función de `break` es evitar esta situación provocando un salto a la salida.

EJEMPLO: IMPLEMENTACIÓN DE UN MENÚ

Se implementará un menú de 3 opciones y una `default`. Las acciones a seguir serán simplemente informar la elección realizada, a modo de ejemplo.

```

#include <stdio.h>

int main ()
{
    int SEL ;
    printf ("\n\t\t\t MENU \n\n" ) ;
    printf ("1. OPCION 1 " ) ;
    printf ("2. OPCION 2 " ) ;
    printf ("3. OPCION 3 " ) ;
    printf ("\n\nIngrese su opcion : " ) ;
    scanf( "%d", &SEL ) ;

    switch ( SEL ) {
        case 1 : printf ("\n Ud. seleccionó OPCION 1 " ) ;
                   break ;
        case 2 : printf ("\n Ud. seleccionó OPCION 2 " ) ;
                   break ;
        case 3 : printf ("\n Ud. seleccionó OPCION 3 " ) ;
                   break ;
        default: printf ("\n Ud. seleccionó otra cosa " ) ;
                  break ;
    }
}

```

ITERACIONES

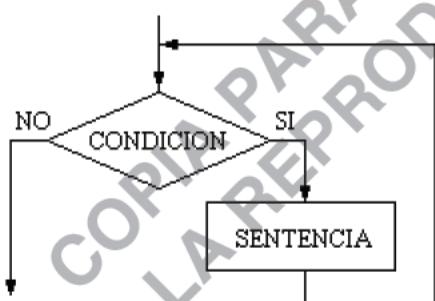
Como se vio anteriormente, una **iteración** consiste en la ejecución repetida de una secuencia de sentencias *mientras* se cumpla una determinada condición. Existen dos tipos de iteración según la ocasión en que se evalúe la condición.

- `while`
- `do-while`

¡Cuidado! Una vez que se cumplió la condición de permanencia en el lazo, ésta debe modificarse dentro del mismo lazo (es decir, dejar de cumplirse), dado que en caso contrario, la condición siempre se cumplirá y el lazo será infinito. El programa se “colgará”.

Mediante el uso de interrupciones de hardware y variables globales se puede modificar la condición del lazo desde fuera del mismo, constituyéndose esto como una excepción de la advertencia anterior.

LAZO WHILE



que muestra el esquema de la izquierda.

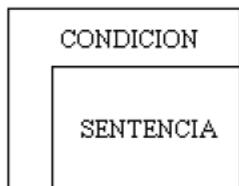
Nótese que la sentencia puede no ejecutarse nunca si la condición no se cumple en primera instancia.

A continuación se muestra el diagrama y la sintaxis correspondiente.

En este lazo se realiza primamente la evaluación de la condición.

Si ésta se cumple se ejecuta la sentencia (que puede ser simple, nula o compuesta), y luego vuelve a evaluarse la condición.

Por lo tanto, “mientras (while) la condición se cumpla” se ejecutará la sentencia, como lo muestra el es-

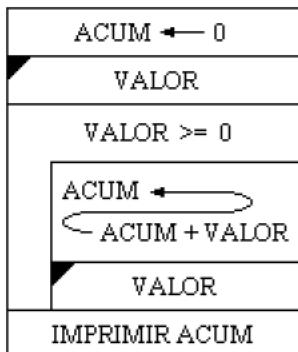


```
while (condición)
    Sentencia ;
```

EJEMPLO: UTILIZACIÓN DEL LAZO WHILE

Se ingresarán y sumarán valores enteros positivos hasta el ingreso de un valor negativo. Este valor no forma parte de los datos a sumar.

Este es el caso del uso de un elemento **centinela** que nos permite saber cuándo finaliza la secuencia de datos. Nótese que el elemento centinela no debe confundirse con los datos válidos: no forma parte de éstos. Es decir que es un no-dato.



Se utilizará una variable “acumulador” ACUM para almacenar la suma.

Esta variable debe ser inicializada en cero.

La variable VALOR se empleará para tomar cada uno de los números ingresantes.

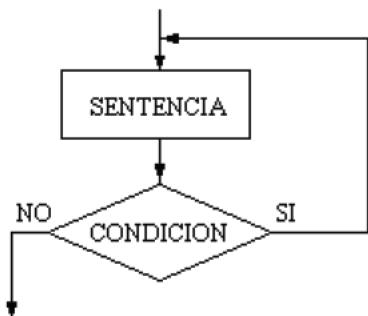
Una estructura frecuente del bucle while es ingresar un dato, verificar su validez en la condición, procesarlo y volver a ingresar un valor dentro del bucle. Esta estructura se puede apreciar en el diagrama de Chapin.

```
#include <stdio.h>

int main ()
{
    int ACUM = 0 , VALOR ;
    printf ("\nIngrese valores enteros. \n" ) ;
    printf ("\nFinaliza con un negativo\n\n" ) ;

    scanf( "%d" , &VALOR ) ;
    while ( VALOR >= 0 ) {
        ACUM = ACUM + VALOR ;
        scanf( "%d" , &VALOR ) ;
    }
    printf ("\nLa suma de los valores es %d\n\n" , ACUM ) ;
}
```

LAZO DO-WHILE



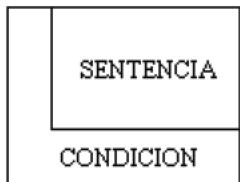
En el lazo `do-while` se ejecuta la sentencia (nula, simple o compuesta) en primer término.

A continuación se evalúa la condición. Si ésta se cumple, se ejecuta nuevamente la sentencia y se repite el ciclo.

La condición, al igual que en el bucle `while`, sigue siendo "condición de permanencia" dentro del bucle.

En este caso, la mínima cantidad de veces que se ejecutará la sentencia es uno, dado que la evaluación de la condición se da al final, contrario al bucle `while`.

A continuación se muestra el diagrama y la sintaxis correspondiente.

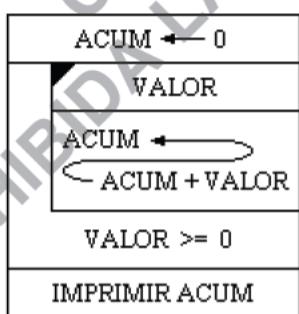


```
do {  
    Sentencia ;  
} while (condición) ;
```

EJEMPLO: UTILIZACIÓN DEL LAZO DO-WHILE

Se planteará un problema similar al anterior, pero con una diferencia que hace más adecuado el uso de `do-while`.

En este caso, se pide ingresar valores enteros e imprimir su suma. El ingreso finalizará con un valor negativo, pero a diferencia del caso anterior, dicho valor pertenece a la secuencia de datos y por lo tanto debe incorporarse a la suma.



Obsérvese el diagrama de Chapin de la izquierda comparándolo con el del ejemplo anterior.

Este es más compacto y ahorra el ingreso del dato previo al bucle.

Esto se debe a que al menos se ingresará un (1) valor válido (característica de `do-while`).

```

#include <stdio.h>

int main ()
{
    int ACUM = 0 , VALOR ;
    printf ("\nIngrese valores enteros. \n" ) ;
    printf ("\nFinaliza con un negativo\n\n" ) ;

    do {
        scanf ("%d", &VALOR ) ;
        ACUM = ACUM + VALOR ;
    } while ( VALOR >= 0 ) ;

    printf ("\nLa suma de los valores es %d\n\n",ACUM ) ;
}

```

CONDICIÓN EJECUTIVA

Hasta ahora hemos utilizado condiciones “pasivas” cuya única función fue ser evaluadas.

Es posible incluir en la condición sentencias ejecutables y llamados a funciones que cumplan una tarea ejecutiva, y además su resultado sea evaluado para dar respuesta a la condición. Esto se denomina **efecto secundario** o *side effect*, ya que la evaluación de la expresión genera ejecución de código y posibles modificaciones en valores de variables. Si una expresión genera efectos secundarios, la cantidad de veces que sea evaluada generará cambios en los resultados o en la funcionalidad ejecutada.

EJEMPLO: CONDICIÓN EJECUTIVA EN UN LAZO

Se leerán caracteres mediante `getchar()` hasta el ingreso de una “Q”. Nótese que el `while` presenta una sentencia nula.

```

#include <stdio.h>

int main ()
{
    printf ("\nPara salir ingrese una 'Q'\n\n" ) ;
    while ( getchar() != 'Q' ) ;

    printf ( " \n\n\n Fin del programa \n" ) ;
}

```

EJEMPLOS DE CONDICIONES

- `A == 2`

Si el contenido de la variable `A` es igual a 2, la condición es verdadera, caso contrario, es falsa. La variable no se modifica.

- `A = 2`

La variable `A` se carga con el valor 2 (asignación). La condición siempre es verdadera, porque el valor 2 es “verdadero” (el único falso es 0). La expresión “vale” el valor asignado. Hay efecto secundario, porque la variable `A` cambia de valor.

- `A == getchar()`

Si el contenido de la variable `A` es igual al ASCII del carácter ingresado (pero no almacenado) de teclado, la condición es verdadera, caso contrario, es falsa.

- `A = getchar()`

La variable `A` se carga con el valor ASCII carácter ingresado de teclado. La condición es verdadera, si este valor ASCII no es cero, caso contrario, es falsa, dado que la asignación “vale” el valor asignado. Hay efecto secundario porque la variable cambia de valor.

- `(A = getchar()) != 'X'`

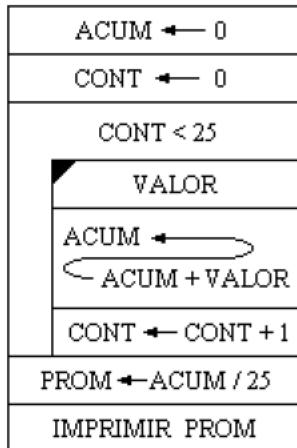
La variable `A` se carga con el valor ASCII ingresado de teclado. La condición es verdadera, si este valor ASCII es distinto del ASCII de la “X”, caso contrario, es falsa. Hay efecto secundario.

LAZO FOR

En la mayoría de los lenguajes se implementa otro tipo de lazo, denominado lazo `for`. Este bucle es en realidad un caso particular del lazo `while`, como veremos a continuación mediante un ejemplo.

EJEMPLO: ITERACIÓN CON NÚMERO FIJO DE VUELTAS (WHILE)

Se calculará el promedio de 25 valores enteros ingresados por teclado.



En este caso, la cantidad de datos es fija y conocida previamente.

Es necesario controlar la cantidad de iteraciones que realiza el lazo `while`. Con este fin se utiliza la variable `CONT`, la que se transforma en variable de control del bucle.

Por cada vuelta se lee un `valor` y se lo acumula en la variable `ACUM`.

El contador de vueltas se incrementa en cada una de ellas.

Es costumbre nombrar a las variables enteras de uso general con las letras `I`, `J`, `K`, etc. Por tal razón utilizaremos `I` en reemplazo

de `CONT`. Nótese que se utilizó un casting para el cálculo del promedio.

```
#include <stdio.h>

int main ()
{
    int ACUM , I , VALOR ;
    float PROM ;
    ACUM = 0 ;
    I = 0 ;

    while ( I < 25 ) {
        scanf ( "%d" , &VALOR ) ;
        ACUM = ACUM + VALOR ;
        I = I + 1 ;
    }

    PROM = (float) ACUM / 25 ;
    printf ( " \n\n\n El promedio es %.2f " , PROM ) ;
}
```



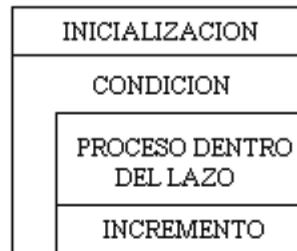
Si restringimos el diagrama de Chapin del ejemplo a la sección del lazo y su control, este se reduce al diagrama que se presenta a la izquierda.

En él se aprecian cuatro secciones: Inicialización de la variable de control, condición de mantenimiento, incremento del contador y operaciones dentro del lazo.

Salvo el proceso que se realiza dentro del lazo, que es particular en cada caso, los otros tres componentes presentan un comportamiento estándar.

Se pueden reconocer entonces las secciones:

- Inicialización
- Condición
- Incrementos



Dado lo habitual de esta situación, el lenguaje C, como muchos otros lenguajes, la resume mediante un lazo llamado `for`, cuya sintaxis es:

```
for ( Inicialización ; Condición ; Incremento )  
    Sentencia ;
```

Como se ve, dentro de los paréntesis se presentan los tres campos separados por punto y coma.

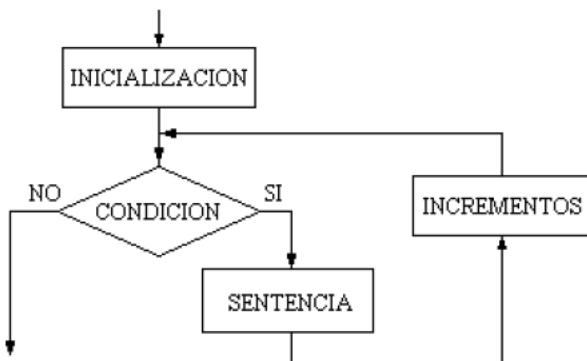
Veamos algunas consideraciones :

- El separador de campos es el *punto y coma* (;).
- Los campos de inicialización e incrementos pueden ser múltiples. Es decir, contener una o varias sentencias. Si es así, estas deben ir separadas por *coma* (,).
- La condición puede ser compuesta.
- La sentencia del `for`, es decir el cuerpo del bucle, puede ser nula, simple o compuesta.
- Los campos pueden estar vacíos, pero los separadores (;) deben estar presentes.
- Salvo en la condición, no es obligatorio que los campos realicen estrictamente acciones de “inicialización” e “incremento”, sino que esto constituye sólo un nombre para el campo dado que la mayor parte de las veces es para lo que se utilizará.

Con respecto a este último punto, es importante conocer la verdadera naturaleza de los campos :

- **Inicialización:** Se ejecuta *una sola vez* al inicio del lazo `for`.
- **Condición:** Se evalúa al inicio de *cada vuelta* del lazo, incluyendo la primera, por lo que la cantidad mínima de vueltas es cero (no podía ser de otra forma dado que `for` es un caso particular del lazo `while`).
- **Incremento:** Se ejecuta *al final* de cada vuelta, justo antes de realizar nuevamente la evaluación de la condición.

El siguiente diagrama ilustra la descripción anterior.



Secuencia del lazo for

EJEMPLO: ITERACIÓN CON NÚMERO FIJO DE VUELTAS (FOR)

Se presentará el mismo caso del ejemplo anterior, pero esta vez resuelto con un lazo for. Se aprovecha este ejemplo para introducir las siguientes modificaciones:

- Se utiliza el comando del **preprocesador #define** para definir la constante **N** con el valor 25.
- Se utiliza el postincremento **I++** en lugar de **I=I+1**

```

#include <stdio.h>
#define N 25

int main ()
{
    int ACUM , I , VALOR ;
    float PROM ;
    ACUM = 0 ;

    for ( I = 0 ; I < N ; I++ ) {
        scanf ( "%d" , &VALOR ) ;
        ACUM = ACUM + VALOR ;
    }

    PROM = (float) ACUM / N ;
    printf ( "\n\n\n El promedio es %.2f " , PROM ) ;
}

```

Dado que el bucle `for` es un caso particular del `while`, no está prevista una estructura propia en los diagramas de Chapin. Por lo tanto, cada usuario lo puede representar como mejor le convenga.

Dado que el lazo for es una derivación del while cualquiera de ellos puede reemplazar al otro.

Generalmente, el lazo for se utiliza cuando se conoce previamente el número de iteraciones del lazo, mientras que el while se aplica en caso que el número sea incierto. Esto es meramente una cuestión de estilo y tradición entre los programadores.

EJEMPLO: USO NO TRADICIONAL DE LAZO FOR

Este ejemplo pretende ilustrar el uso no tradicional de los campos del lazo `for` asignándoles acciones diferentes de las de inicialización e incremento.

Se mostrará en pantalla una cuenta creciente. Se adicionaron sentencias `printf` a los campos de inicialización e incremento.

Se separaron los tres campos del `for` a fin de visualizarlos mejor. Obsérvense los siguientes aspectos :

- El campo de *inicialización* es múltiple.
- Una de sus partes se utiliza para imprimir “Ingresando”.
- El campo de *incrementos* contiene la función `printf()`.
- Esta muestra el valor contenido en `I` en la pantalla.
- El contenido de `I` se incrementa luego de mostrarse en pantalla por ser un *postincremento*.

```
#include <stdio.h>

int main ()
{
    int I ;

    for (printf("Ingresando") , I=0 ;      /* Inicializacion */
          I < 5 ;                      /* Condicion        */
          printf("\n%d", I++ ) ) ;      /* Incrementos     */

}
```

En pantalla se observa:

```
Ingresando
0
1
2
3
4
```

SENTENCIA CONTINUE

Es una sentencia no estructurada que se aplica en los lazos **for**, **while** y **do-while**.

Su efecto es conducir el flujo del programa directamente al campo de incrementos o, lo que es lo mismo, al final del bloque del bucle más interno que lo contiene. Podemos decir que produce que el lazo pase “a la próxima vuelta”.

A continuación, un ejemplo sencillo para demostrar el funcionamiento de **continue**:

```
#include <stdio.h>
int main()
{
    int i;
    for(i=0; i<10; i++) {
        if(i==2 || i==6) continue;      // "saltea" el 2 y el 6
        printf("\n %d", i);
    }
}
```

SENTENCIA GOTO

La sentencia **goto** representa el salto incondicional. Por supuesto, se trata de una sentencia no estructurada.

Provoca el salto incondicional al lugar del programa donde se encuentra el rótulo asociado.

Su sintaxis es:

```
goto ROTULO;
//sentencias que serán omitidas
ROTULO:
//sentencias donde "cae" el salto
```

EJEMPLO: APLICACIÓN DE GOTO

Analice el programa y determine los valores que aparecerán en pantalla.

```

#include <stdio.h>
int main ()
{
    int I , J ;
    char LETRA ;

    for ( I=0 ; I<10 ; I++ ) {
        printf ( " \nI = %d  Ingrese una letra ", I );
        LETRA = getchar(); getchar();
        if (LETRA=='Q') {
            J = 5 ;
            goto LAZO_J;
        }
    }
    printf ( "\n\n\nFin del lazo I \n\n\n" );
    getchar() ;
    for ( J=0 ; J<10 ; J++ ) {
LAZO_J :
        printf ( " \n\nJ vale %d ", J );
    }

    printf ( " \n\n\n Fin del Programa " );
}

```

Nótese el uso de un `getchar` doble cuando se realiza la lectura del carácter que el usuario ingresó por teclado. La razón de esto es que `getchar` nos retornará el código ASCII de lo ingresado por teclado, pero lo hará recién cuando presionemos la tecla Enter. Hecho esto, `getchar` no retira del buffer de entrada de teclado la ocurrencia del Enter (que también tiene su código ASCII), por lo que el segundo `getchar` se precisa para “eliminar” ese Enter del buffer. Con otras palabras, `getchar` sólo retira del buffer un carácter por vez, y lo hace cuando ocurre un Enter.

Puede comprobar el funcionamiento diferente del programa si se omite el segundo `getchar`. Se observará que el programa “se saltea” un ingreso, por la presencia de este Enter residual.

Si bien el `goto` es una sentencia no-estructurada, algunas veces permite una reducción en la complejidad del programa.

Por lo tanto, se aconseja no usarla a menos que la ventaja de su aplicación sea evitente.

DESARROLLO TOP DOWN

El desarrollo top-down es una técnica de desarrollo de software orientada al código (a diferencia de otras técnicas orientadas a datos).

Esta técnica, desarrollada por John Wellum, contempla diversos niveles de rutinas anidadas de forma tal que los niveles superiores (*top*) comanden el comportamiento en gran escala y los niveles inferiores (*down*) se encarguen de los detalles.

La programación top-down refiere a una técnica de desarrollo en la que un programa se construye comenzando por la descripción de mayor nivel (generalmente el enunciado o las especificaciones) y procede dividiendo ésta en secciones cada vez más simples, hasta que el nivel de cada una de estas partes pueda ser resuelto directamente. A esto le llamamos **refinamiento**.

A continuación se mostrará la aplicación de esta técnica mediante un ejemplo.

EJEMPLO: DESARROLLO TOP DOWN

En esta aplicación se pretende mostrar en pantalla los números enteros primos menores que 10.000 sin importar el efecto de “scrolling” de pantalla.

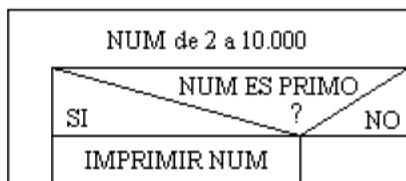
El desarrollo top-down conduce a la construcción de módulos o **funciones** que se evitarán en este caso por no haberse visto aún esta característica del lenguaje.

NIVEL I

IMPRIMIR LOS NUMEROS
PRIMOS MENORES QUE 10.000

Como se mencionó anteriormente, el primer nivel de desarrollo corresponde a las especificaciones.

NIVEL II



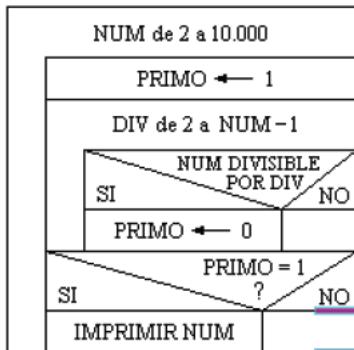
El segundo nivel contempla la primera **depuración** del nivel anterior.

En ésta se realiza un barrido mediante un lazo **for** de todos los números enteros desde 2 (primer primo) hasta 10.000 utilizando la variable **NUM**. Toda vez que **NUM** sea primo se muestra su valor en pantalla.

El problema se reduce ahora a determinar si cada valor de **NUM** es primo o no. Es decir, el enunciado se redujo a “*dado un número entero, determinar si es primo*”.

Recordando que los números primos son *aquellos enteros positivos divisibles solamente por la unidad y por sí mismos*, podemos realizar para cada uno de ellos la división con valores comprendidos entre 2 y **NUM-1**. Si no se encuentra un divisor exacto, el número es primo.

NIVEL III



Se utilizará la variable DIV para verificar la divisibilidad de NUM.

En principio se supondrá que el número es primo hasta que no se demuestre lo contrario, al encontrar un divisor exacto.

Esta suposición se realiza asignando a la bandera PRIMO el valor 1.

Si se encuentra un divisor exacto, la bandera pasará a 0. Esta bandera contiene la información que permite determinar si el número es primo o no.

A fin de completar el diagrama, solo resta determinar cuándo un número es divisible por otro.

Esto ocurre cuando el **resto** de la división entera de uno por el otro arroja el valor cero.

Se utiliza el operador % para la determinación del resto entre NUM y DIV.

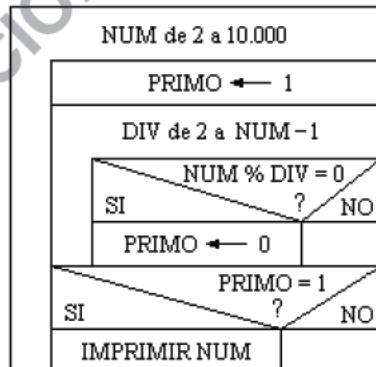
A continuación se muestra el código del programa:

```

#include <stdio.h>
int main()
{
    int NUM, PRIMO, DIV;

    for ( NUM=2 ; NUM < 10000 ; NUM++ ) {
        PRIMO = 1 ;
        for( DIV=2 ; DIV<NUM ; DIV++ )
            if ( ! ( NUM % DIV ) )
                PRIMO = 0 ;
        if(PRIMO)
            printf( "%8d" , NUM ) ;
    }
}
  
```

NIVEL IV



Al ejecutar el programa anterior se aprecia el elevado tiempo de ejecución. Si cambiamos el valor tope 10.000 por 100.000 se ve que la resolución se torna muy lenta: alrededor de 18 segundos en una computadora actual.

Nótese además que la determinación de la **primalidad** de un número se hace más lenta a medida que el número crece, dado que cada vez hay más posibles divisores que “testear”.

Es posible realizar algunas mejoras a fin de disminuir el tiempo de ejecución.

PRIMOS: DEPURACIÓN 1

En la solución anterior se generan todos los números enteros de 2 a 10.000 y se verifica si son primos.

Pero se sabe que el único número **primo par** es dos, por lo que se está perdiendo tiempo inútilmente al generar y verificar el resto de los números pares.

Modificación en el programa: mostrar el número 2 y luego generar solamente los números impares a partir de 3.

PRIMOS: DEPURACIÓN 2

En el lazo de determinación de número primo, si se encuentra un divisor exacto se coloca la bandera **PRIMO** en 0 pero se continúa evaluando al resto de los posibles divisores.

Si se encontró un divisor exacto, el número ya no es primo y debe salirse del lazo.

Modificación en el programa: utilizar una condición compuesta en el lazo de determinación de número primo de forma tal que continúe si **DIV** no alcanzó el valor de **NUM** y *además* no se haya detectado aún ningún divisor.

PRIMOS: DEPURACIÓN 3

Cuando se divide un número entero por un divisor exacto, se obtiene como resultado otro número entero, que también es divisor exacto del primero.

Por ejemplo :

$$15 / 3 = 5 \quad \text{y} \quad 15 / 5 = 3 \quad \text{dado que} \quad 5 \times 3 = 15$$

En el algoritmo de búsqueda de divisores exactos planteado en el programa, es imposible encontrar el divisor 5 sin antes encontrar el divisor 3.

Tomemos el caso del número 36. Sus pares de divisores son :

$$(2, 18) \quad (3, 12) \quad (4, 9) \quad (6, 6) \quad (9, 4) \quad (12, 3) \quad (18, 2)$$

Vemos que los pares de la izquierda se repiten invertidos a la derecha. El límite está dado por el par (6, 6), siendo 6 la **raíz cuadrada** del valor a dividir.

Esto nos permite afirmar que: si no encontramos un valor de DIV que sea divisor exacto de NUM hasta la raíz cuadrada de éste, ya no lo encontraremos. Podemos utilizar esta propiedad para acortar el lazo de determinación de número primo.

Modificación en el programa: restringir el límite de DIV a la raíz cuadrada de NUM (inclusive) utilizando la función `sqrt()` asociada con la cabecera `math.h`.

De esta forma la **condición de mantenimiento** en el lazo queda:
`(DIV <= sqrt(NUM)) && PRIMO`

El tiempo para el máximo de 100.000 se redujo a 100 milisegundos (!).

El programa depurado se muestra a continuación :

```
#include <stdio.h>
#include <math.h>
#define MAX 10000

int main()
{
    int NUM , PRIMO , DIV ;
    printf( "%8d", 2 ) ; //mostramos el 2

    for ( NUM=3 ; NUM<MAX ; NUM+=2 ) {
        PRIMO = 1 ;
        for( DIV=2 ; (DIV<=sqrt(NUM)) && PRIMO ; DIV++ )
            if(!(NUM % DIV))
                PRIMO = 0 ;
        if(PRIMO)
            printf( "%8d", NUM ) ;
    }
}
```

CONSIDERACIONES SOBRE LA PROGRAMACIÓN TOP DOWN

La programación top-down complica la prueba del programa dado que no existe nada ejecutable durante los pasos intermedios del desarrollo. Nótese que en el ejemplo anterior, el programa hizo su aparición al final del proceso.

Otra desventaja es que todas las decisiones tomadas desde el inicio del proyecto dependen directa o indirectamente de las especificaciones del nivel más alto. La consecuencia de esto es que si estas especificaciones varían en el tiempo, es probable que la mayor parte del programa deba ser reescrito.

La programación top-down tiende a generar módulos muy específicos de la aplicación en desarrollo, y por lo tanto, difícilmente reutilizables.

De todas formas, la programación top-down es una herramienta muy útil, aplicable a la mayoría de los casos que se tratarán en este libro.

DESARROLLO BOTTOM UP

La programación **bottom-up** o ascendente es el caso opuesto a la programación top-down.

Bottom-up hace referencia a un estilo de programación donde la aplicación se construye comenzando con los módulos más primitivos o simples del lenguaje de programación, y agregándole gradualmente niveles de complejidad hasta que la aplicación completa esté terminada.

Una gran ventaja que presenta el desarrollo bottom-up (no así el top-down) es que el código puede ser probado en cada paso, con muy pocas líneas adicionales.

El siguiente ejemplo ilustra los pasos sucesivos de un desarrollo bottom-up.

EJEMPLO: DESARROLLO BOTTOM UP

El programa a desarrollar pretende verificar si la cantidad de números primos en diversos rangos numéricos se mantiene más o menos constante. Para ello se quiere calcular la densidad promedio de números primos en sucesivos rangos de 1000 números.

Un enunciado de esta naturaleza podría parecer muy complejo inicialmente, por lo que nos plantearemos uno mucho más simple: ingresar dos números enteros por teclado y determinar si el primero es divisible por el segundo.

```
#include <stdio.h>
int main ()
{
    int NUM , DIV ;
    scanf("%d %d", &NUM, &DIV);
    if ( ! (NUM % DIV))
        printf ( "\n\n%d es divisible por %d", NUM , DIV );
    else
        printf ( "\n\n%d no es divisible por %d", NUM , DIV );
}
```

Este programa es muy simple y puede ser testeado a fin de lograr su **aceptación**. A partir de este programa ya probado se agregará un grado de dificultad. Si el programa correspondiente a la segunda versión no funciona, siempre podemos volver a la versión anterior que representa un “lugar seguro”, y recomenzar el diseño desde allí.

Basandonos en la determinación de la divisibilidad, se tratará de determinar si un entero ingresado por teclado es primo o no.

Dado que este programa fue desarrollado en el ejemplo anterior, se lo tomará con las depuraciones realizadas.

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

int main()
{
    int N , PRIMO , DIV;
    printf("Ingrese un entero: ");
    scanf("%d" , &N ) ;
    PRIMO = 1 ;
    for(DIV=2 ; (DIV<=sqrt(N)) && PRIMO ; DIV++ )
        if(!(N % DIV))
            PRIMO = 0;

    if(PRIMO)
        printf("\n\n%u es primo" , N );
    else
        printf("\n\n%u NO es primo" , N );
}
```

Habiendo realizado la aceptación del programa que determina si un número es primo o no, se construirá basándose en éste un programa que cuente la cantidad de números primos en un determinado rango.

Para este caso se seleccionó el rango 1000 – 2000.

```
#include <stdio.h>
#include <math.h>
int main()
{
    int N , PRIMO, DIV , INICIO , FIN , CONT = 0 ;
    INICIO = 1000;
    FIN     = 1999;

    for ( N=INICIO ; N<=FIN ; N++ ) {
        PRIMO = 1;
        for( DIV=2 ; (DIV<=sqrt(N)) && PRIMO ; DIV++ )
            if(!(N % DIV))
                PRIMO = 0;
        if(PRIMO)
            CONT++;
    }
    printf("\n\n\nHay %u primos entre %u y %u.",CONT,INICIO,FIN);
}
```

El siguiente paso es automatizar la selección de rangos consecutivos y mostrar la cantidad de primos en cada uno de ellos.

Enunciado: *Determinar la cantidad de números primos por rango de intervalo 1000, entre 1000 y 30000.*

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

#define INCREM 1000

int main()
{
    int N , PRIMO , DIV , INICIO , FIN , CONT ;

    for( INICIO=1000 ; INICIO<30000 ; INICIO+=INCREM ) {
        CONT = 0 ;
        FIN = INICIO+INCREM;
        for ( N=INICIO ; N<FIN ; N++ ) {
            PRIMO = 1;
            for( DIV=2 ; (DIV<=sqrt(N)) && PRIMO ; DIV++ )
                if(!(N % DIV))
                    PRIMO = 0;
            if(PRIMO)
                CONT++;
        }
        printf("\n\n\n %10u %10u %10u.", \
               INICIO , INICIO+INCREM , CONT );
    }
}
```

El último paso es tomar las cantidades obtenidas en el programa anterior y calcular su promedio. Esto indicará la densidad de primos por rango.

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

#define INCREM 1000

int main()
{
    int N, PRIMO, DIV, INICIO, CONT, ACUM=0, CONTAD = 0 ;
    for( INICIO=1000 ; INICIO<20000 ; INICIO+=INCREM ) {
        CONT = 0 ;
        for ( N=INICIO ; N<INICIO+INCREM ; N++ ) {
            PRIMO = 1 ;
            for( DIV=2 ; (DIV<=sqrt(N)) && PRIMO ; DIV++ )
```

```

        if(!(N % DIV))
            PRIMO = 0 ;
        if(PRIMO)
            CONT++ ;
    }
    ACUM += CONT ;
    CONTAD++ ;
}
printf("\n\n El promedio de primos es .2f", (float)ACUM/CONTAD);
}

```

CONSIDERACIONES SOBRE LA PROGRAMACIÓN BOTTOM UP

Como se mencionó anteriormente, el desarrollo bottom-up permite el testeo de todos los pasos del desarrollo. Las secciones de programa escritas con la mecánica bottom-up tienden a ser más generales, y por lo tanto reutilizables, que las escritas como top-down.

Nótese que en el ejemplo anterior cada paso fue desarrollado como si fuera el último. En las especificaciones de cada uno de estos pasos no se hizo mención alguna al paso siguiente. Es decir, fue una aplicación independiente que podía ser utilizada para muchos y diversos enunciados posibles del siguiente paso.

Esta particularidad permite postergar la decisión final concerniente al exacto funcionamiento de la aplicación. La posibilidad de demorar dicha decisión final hace al desarrollo más inmune a los cambios de especificaciones durante el proyecto.

Nótese que esto no ocurre con el desarrollo top-down dado que el primer paso de diseño ya lo constituye el enunciado y los pasos posteriores ya lo tienen incorporado. Un cambio de especificaciones en este caso puede significar comenzar el proyecto desde el inicio.

PROBLEMAS PROPUESTOS

1. Se ingresarán números enteros positivos. Determinar cuántos de estos son pares. El ingreso finalizará con un número negativo.
2. Se ingresarán 100 números enteros. Sumar los de orden impar 1,3,5,... por un lado y los de orden par 2,4,6,... por otro. Determinar cuáles proporcionan la mayor suma.
3. Se ingresarán números enteros hasta que se ingrese el 235. Indicar cuántas veces ocurrió el ingreso del número 23.
4. Permitir el ingreso de una clave numérica entera. Finalizar el ingreso solamente cuando la clave ingresada sea 23645.
5. Repetir el problema anterior permitiendo sólo 3 intentos. Luego del tercer intento fallido colocar una advertencia.
6. Determinar si un número entero positivo ingresado por teclado es o no un número perfecto. Los números perfectos son aquellos cuyo valor es igual a la suma de todos sus divisores exactos con excepción del mismo número, por ejemplo $6 = 1+2+3$
7. Mostrar los primeros 5 números perfectos.
8. Simular el tiro de una moneda para 1, 10, 100, 1000 y 10000 intentos. Mostrar el porcentaje de ocurrencia de cara y seca.

Para realizar este programa necesitará obtener números aleatorios, que permitan simular el arroje de la moneda. En la biblioteca estándar de C existe la función `rand()`, que devuelve un número pseudoaleatorio. Este número se obtiene por un algoritmo específico, que devuelve números que *parecen* no seguir ninguna serie, por lo que se pueden utilizar como pseudoaleatorios. Además, se precisa inicializar la secuencia partiendo de un valor que cambie cada vez (el inicio de la secuencia se denomina “semilla” ó “seed”) u obtendremos siempre la misma secuencia. Aquí un ejemplo:

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

int main()
{
    int i;
    srand(time(0)); //inicialización de la semilla con el reloj actual
    for(i=0; i<20;i++)
        printf("%8d", (rand() % 9)+1); //aleatorio de 1 al 9 inclusive
}
```

9. Determinar si hay enteros de 3 cifras cuyo valor sea igual al producto de cada una de las mismas.
10. Ingresar 10 valores por teclado. Indicar cuál fue el mayor y en que posición entró.
11. Ingresar 10 valores por teclado. Indicar si esta secuencia es creciente (todo valor es mayor que el anterior).
12. Ingresar por teclado un entero decimal positivo. Mostrarlo en pantalla en binario.
13. Ingresar por teclado un número binario. Mostrar en pantalla su valor decimal.
14. Ingresar los datos de 10 alumnos consistentes en legajo y 8 notas. Indicar cuál es el alumno de mejor promedio.
15. Ingresar los datos de los alumnos de un curso. Estos datos consisten en nota (int) y sexo (char: M/F). Indicar si el mejor promedio pertenece a los alumnos o a las alumnas. Utilizar switch para la selección.

4. FUNCIONES

Para comprender el fundamento de las **funciones**, los beneficios de su uso y los problemas que deben ser resueltos para su utilización, debemos remontarnos a sus orígenes en el bajo nivel.

Paralelamente podremos comprender la diferencia entre éstas y las macros, apreciando las ventajas y desventajas de cada una.

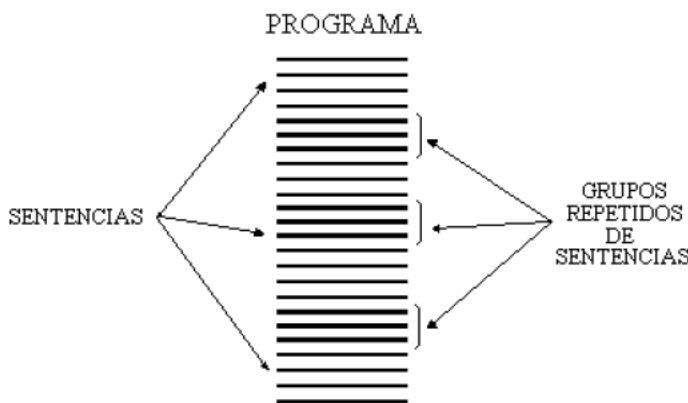
MACROS

No se desarrollará en este capítulo el modo de implementar **macros** en el lenguaje C, pues no es éste el tema que nos ocupa. Simplemente se presentará el concepto a fin de poder compararlo con las subrutinas o funciones, dejando su implementación para capítulos posteriores.

Podemos considerar un programa compuesto por una secuencia de sentencias o instrucciones, en el cual se necesite repetidamente realizar una determinada tarea, y no se disponga de una sentencia específica que la lleve a cabo.

Se deberá implementar, por lo tanto, una sección del programa, cuya ejecución solucione el problema mencionado.

En la siguiente figura se muestra esquemáticamente, con líneas destacadas, la sección del programa que se repite de esta manera:



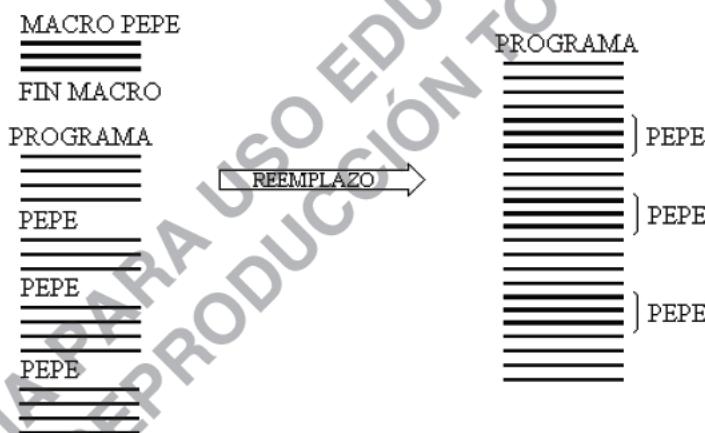
Sería deseable disponer de una instrucción que realizara la tarea antes mencionada. Una especie de instrucción gigante que involucrara a las otras que componen el grupo de repetición. Sería ésta una **macro-instrucción**.

El primer lenguaje que permitió la utilización de este recurso fue el Macro-Assembler.

La macro-instrucción (o simplemente **macro**), debe ser definida previamente e identificada con un rótulo. Dentro del programa se la invoca sencillamente por el rótulo antes mencionado.

Antes del proceso de traducción de Assembler a **código de máquina** se produce el reemplazo del rótulo de la macro por el grupo de instrucciones que la componen. De esta manera el programa traducido no presentará ninguna diferencia con el programa original en el cual no se utilizó la macro. Por lo tanto, ésta ofrece ventajas aplicables solamente al programa fuente.

La siguiente figura ilustra el proceso de utilización de una macro:



Las ventajas que podemos enunciar al utilizar macros son:

- El programa fuente es más corto, siempre que la macro se utilice más de una vez.
- El programa fuente es más fácil de comprender, pues una vez que la macro ha sido entendida, no es necesario repetir su análisis.
- Se puede disponer de una biblioteca de macros.
- Al desarrollarse, depurarse y aceptarse las macros en forma independiente del programa que las utiliza, ofrecen modularidad al proceso.

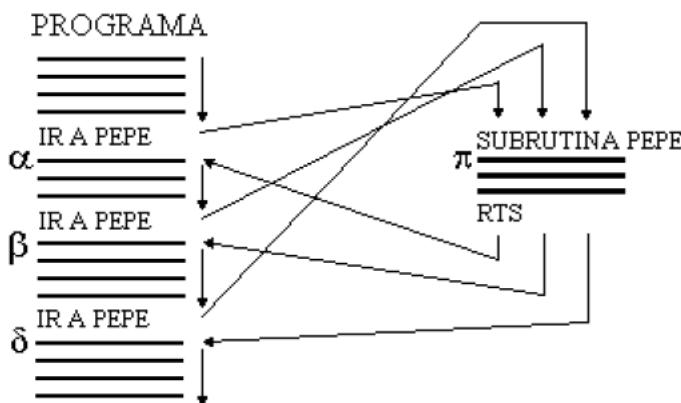
SUBRUTINAS

También llamadas **funciones** en el lenguaje C, y **procedimientos** en el lenguaje Pascal. Son programas relativamente independientes del programa invocante (el cual puede ser el programa principal, el sistema operativo u otra función).

A diferencia de las macros, no se realiza ningún tipo de reemplazo *antes* de la traducción (**compilación**). Las funciones son **invocadas** cada vez que sea necesario su uso *durante la ejecución* del programa, produciendo una ruptura de la secuencia natural del programa. Luego de su ejecución se continúa el programa invocante en la sentencia siguiente a la de llamada a la función.

Decimos que la secuencia de ejecución pasará del **llamante** (quien invoca) a la función. Cuando ésta **retorne** (es decir, finalice), la línea de ejecución volverá al llamante.

Este proceso se muestra en siguiente figura:



Obsérvese que luego de la ejecución de la subrutina, se continúa en diferentes direcciones (marcadas en la figura con las letras griegas alfa, beta y gamma), según desde dónde haya sido realizada la invocación o llamada.

Para realizar el retorno se utiliza una sentencia especial en Assembler denominada **RTS** (*retorno de subrutina*). El compilador de C colocará esta instrucción cuando traduzca nuestro código fuente en el código que ejecutará el microprocesador.

Podemos mencionar como ventaja del uso de funciones, que resultan programas objeto y ejecutables más cortos que en el caso de las macros.

También en este caso se puede disponer de **bibliotecas** de funciones, propias o adquiridas y se asegura la modularidad del proceso. De hecho, como hemos dicho, el lenguaje C tiene muy pocas palabras reservadas propias del lenguaje. En cambio, es muy rico en bibliotecas de macros y funciones, tanto por contar con

una **biblioteca estándar** que todo compilador estándar debe proveer como por la cantidad de bibliotecas de terceros que se encuentran disponibles.

Como desventaja frente a las macros o a una programación lineal que repita las instrucciones literalmente, podemos mencionar la gran cantidad de rupturas de secuencia (saltos) que se deben realizar como resultado de las llamadas o invocaciones. También será necesaria generalmente la transferencia de datos entre el llamante y la función, con la consecuente utilización de memoria en el momento de ejecutar el programa.

Como se puede ver una vez más, la optimización de los **recursos** principales del sistema (velocidad en el caso de las macros, y ahorro de memoria en el de las funciones) está en contraposición.

A fin de manejar las funciones es necesario enfrentar dos problemas:

1. Salvaguarda de la dirección de retorno
2. Transferencia de argumentos

SALVAGUARDA DE LA DIRECCIÓN DE RETORNO

Como se vió en la figura anterior, al finalizar la función el Contador de Programa (PC, *Program Counter*) debe pasar a contener la dirección de memoria en donde se encuentra la instrucción siguiente a la que realizó la llamada a la función. Esto tiene como objetivo retornar al programa invocante en el lugar desde donde se partió para ejecutar la función.

Si recordamos que el *Program Counter* es aquel registro del procesador que contiene en todo momento la dirección de la siguiente instrucción a ser ejecutada, bastará con guardar su contenido en el momento anterior a la ruptura del flujo hacia la función. Esto es: *antes de cargar el PC con la dirección pi, se deberá guardar su contenido (sea éste alfa, beta o gamma) en algún lado*.

Esta salvaguarda podría realizarse en un registro especial (es decir, dentro del mismo microprocesador), del cual se recuperaría la dirección de retorno en el momento oportuno. Pero no es ésta la solución adecuada.

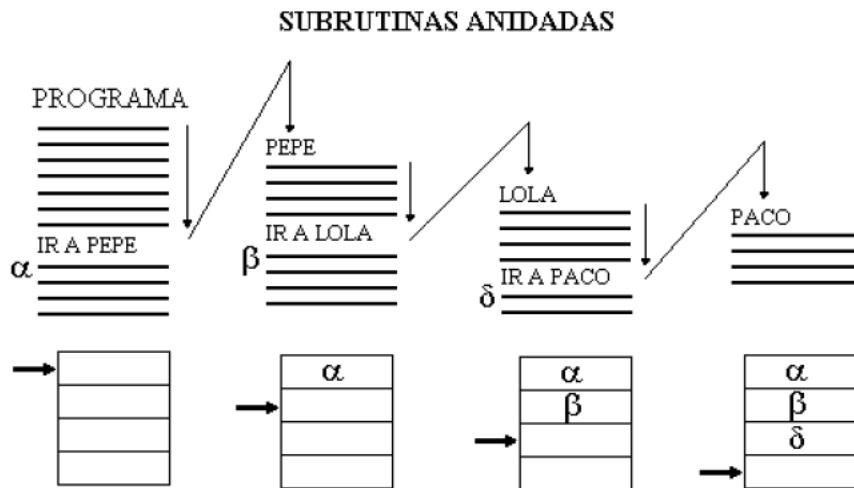
La función es, en definitiva, un programa y nada impide que desde ella se quiera invocar otra función, y desde esta última se invoque otra, y así sucesivamente. Estamos en el caso de **funciones o subrutinas anidadas**.

Como el registro de salvaguarda propuesto sólo puede albergar una dirección de retorno, ésta sería la última y se perdería el contacto con las anteriores, lo cual llevaría a la pérdida del control del programa dado que no sabríamos a dónde retornar cuando termine la función llamada. Por lo tanto, un registro para conservar la dirección a donde debe retornarse no es la solución.

Este problema podría solucionarse utilizando un **vector** (que estudiaremos en el capítulo siguiente). Para conceptualizarlo con una idea resumida, se trata de una *lista* en la cual se irán guardando las sucesivas direcciones de retorno. Debería

llevarse el control de cuál es la posición vacante del vector, en un registro que conserve el **subíndice** de dicha posición vacante. El subíndice es un número que identifica a cada “hueco” de esta lista o vector. Técnicamente éste control se realiza con una variable denominada **puntero**, algo que trataremos en un posterior capítulo.

La siguiente figura muestra un sistema de funciones o subrutinas anidadas en el que figura la progresión en que se va “llenando” el mencionado vector, como así también la indicación de cuál es la próxima posición vacante (puntero que dibujamos como una flecha):



Obsérvese que el vector se va escribiendo en forma descendente, y se actualiza automáticamente la posición del “puntero” (dibujado como una flecha) a la ubicación vacante. Es decir, al guardar una **dirección de retorno** se realizan dos tareas: la escritura de dicha dirección en la posición vacante, y el incremento automático del puntero para señalar al próximo lugar vacante.

Cuando se realizan los **retornos**, la operación será inversa: se decrementará el puntero a la posición anterior y luego se leerá la dirección de retorno de la posición del vector indicada por el puntero. A partir de ese momento, dicha posición del vector se considerará vacante.

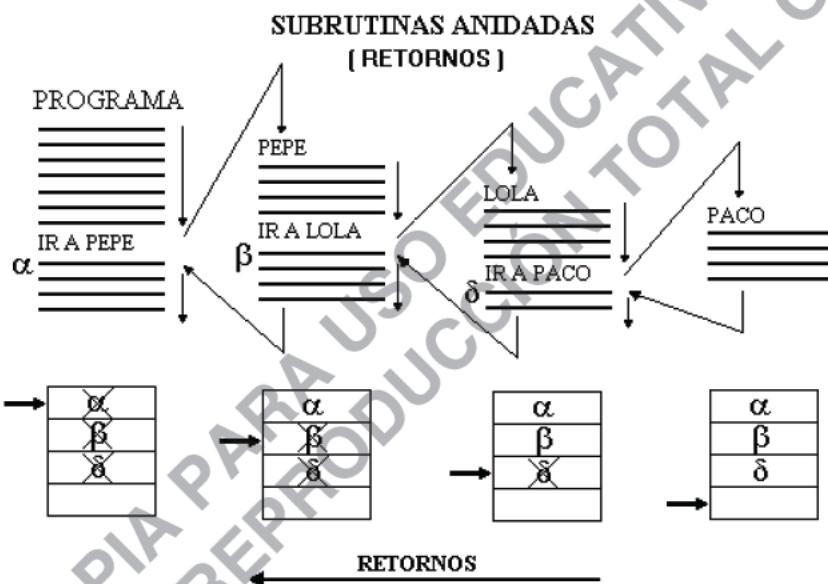
Se observa que el sentido de **lectura** es exactamente opuesto al de **escritura**. Podemos decir que el último elemento en ser escrito, será el primero en ser leído.

La memoria que se utiliza de esta manera es denominada **LIFO** (*last in, first out*), es decir, el **último** en entrar es el **primero** en salir. Habitualmente a este tipo de memoria se la denomina **pila** (en inglés, *stack*), por la similitud a una pila de objetos, de manera que cuando tomamos uno del **tope** de la pila éste será el **último** objeto que habíamos dejado allí. Es decir que el orden de recuperación es inverso al del almacenamiento. Por esto, al puntero que marca la posición vacante lo llamamos “**puntero de pila**” o **stack pointer** (SP).

Es necesario notar que el comportamiento LIFO de la pila es estricto, lo que significa que debe cumplirse siempre. No se pueden acceder a datos intermedios de la pila, ni darle un acceso *random* (acceso arbitrario a cualquier posición). El único dato accesible es el del **tope de pila**.

Por otro lado debe considerarse que la lectura de la pila es destructiva, por cuanto un dato leído se considera extraído de la pila y no se puede volver a leer. Si bien no existe destrucción física hasta que el dato sea sobrescrito, estará inaccesible en forma lógica.

La siguiente figura muestra el procedimiento de retorno de las subrutinas anidadas y la lectura de la pila en forma inversa, en la cual se marcarán los datos leídos como eliminados lógicamente.



Es entonces a través de una **pila**, ubicada en la **memoria principal** de la computadora (y que forma parte de la memoria que utiliza nuestro programa cuando está en ejecución) que se implementa la salvaguarda de la dirección de retorno. Esto posibilita el anidamiento de funciones, es decir, que dentro de una función se pueda llamar a otras, sin limitaciones.

TRANSFERENCIA DE ARGUMENTOS

Podemos afirmar que todo programa comanda un proceso que toma datos y devuelve resultados, pudiendo ser tanto los datos y los resultados de muy diversa índole (incluso rulos).



En el caso de una función, ésta tomará datos del programa invocante y le devolverá a él los resultados.

Una función en lenguaje C puede retornar un resultado como máximo. En caso de no retornar ninguno, se representa esta situación mediante el tipo de dato **void**.

En caso de ser necesario un retorno de mayor cantidad de valores será necesario utilizar técnicas más complejas.

Los datos que recibe una función desde el programa invocante reciben la denominación de **argumentos**.

Las variables de la función destinadas a recibir los argumentos se llaman **parámetros formales**.



Existen dos formas básicas de transferir argumentos a una función:

- Mediante un área común de memoria.
- Mediante áreas individuales de memoria.

ÁREA COMÚN DE MEMORIA

Existe un área de memoria accesible por todas las funciones del programa. Es posible dejar en ella los argumentos que debe tomar la función, como así también los resultados que ella arroja.

Las variables que se alojan en este sector se denominan **variables globales**.

ÁREAS INDIVIDUALES DE MEMORIA

Se pueden determinar áreas de memoria correspondientes a cada función en particular, a las que sólo puedan acceder las funciones relacionadas con ellas, siendo dichas áreas invisibles para las demás funciones (incluso para el programa invocante).

Las variables que se alojan en estas áreas se llaman **variables locales**.

Aquellas variables locales destinadas a recibir los argumentos del programa invocante se denominan los **parámetros formales**.

VARIABLES GLOBALES

Son aquellas que se declaran fuera de toda función (fuera también de la función `main`, que también es una función aunque no lo hayamos precisado).

Estas variables residen en un área común de memoria denominada **área estática**, dentro del bloque de memoria que ocupan los datos de nuestro programa cuando se ejecuta (que se denomina **segmento de datos**).

Su validez, también denominada **ámbito**, *scope* o visibilidad, se extiende a todas las funciones, con una única excepción que veremos más adelante. Es decir, son *visibles* desde todos lados (salvando un caso especial).

Debido a que todas las funciones tienen derecho a “ver” (y usar) las variables globales, éstas deberán permanecer en el segmento de datos hasta el final del proceso. Por lo tanto su tiempo de vida coincide con el del proceso total y ocupan memoria durante ese lapso. Recordemos que, sintéticamente, un **proceso** es un programa en ejecución, según los términos que se utilizan normalmente en los sistemas operativos.

VARIABLES LOCALES

Son aquellas variables que se **declaran** dentro de una determinada función (ésta puede ser el `main`).

El área en donde residen las variables locales es la pila, en el sector destinado a la función en que fueron creadas. Es decir que la pila del programa contiene no sólo las direcciones de retorno, como hemos visto, sino que también almacena las variables locales de las funciones.

Por residir en la pila es que las variables locales no pueden ser accesibles desde otras funciones, dado que éstas no ven otros sectores de memoria más que su propia área de pila y el área estática (donde están las variables globales). De esta forma, podríamos declarar variables locales con el mismo nombre en funciones distintas, y todas ellas serían variables diferentes.

Una vez que se abandona definitivamente una función para retornar al llamante, el sector de pila de esta función queda destruido como se vió anteriormente (la flecha retrocede). Por lo tanto podemos decir que una variable local ocupa memoria mientras la función en la que fue declarada esté **vigente**.

Estar **vigente** no significa que esté **activa**. Cuando una función llama a otra deja de estar activa para estarlo la invocada, pero en algún momento se retornará a la invocante, por lo que aún está vigente y sus variables locales existen.

PARÁMETROS FORMALES

Son variables locales destinadas a recibir los argumentos que transfiere el programa llamante. Fuera de esto, se comportan exactamente igual que las demás variables locales. En otras palabras, decimos que los parámetros formales son **variables automáticas**: por el mero hecho de declararlos obtenemos variables con esos nombres. Más adelante exploraremos el modo en que se declaran.

En el siguiente cuadro se resumen las características de los distintos tipos de variables que pueden existir en un programa:

VARIABLES	VALOR INICIAL	HÁBITAT	TIEMPO DE VIDA	VISIBILIDAD
GLOBALES	CERO	ÁREA ESTÁTICA	TODO EL PROCESO	TODAS LAS FUNCIONES (1 EXCEPCIÓN)
LOCALES	VALOR ALEATORIO	ÁREA DE LA PILA ASIGNADA A LA FUNCIÓN	VIGENCIA DE LA FUNCIÓN DONDE FUERON CREADAS	SOLO EN LA FUNCIÓN EN QUE FUERON DECLARADAS
PARÁMETROS FORMALES	ARGUMENTO TRANSFERIDO			

CONSIDERACIONES ADICIONALES

La utilización de variables globales para realizar transferencia de datos a las funciones puede ser un procedimiento cómodo, especialmente cuando las funciones deben retornar más de un resultado, pero crea una fuerte dependencia entre la función y el programa invocante debido a que es necesario conocer exactamente el nombre de las variables destinadas a realizar el intercambio, y el proceso no funcionará en otros contextos. Si esto sucede, decimos que hay un **acomplamiento fuerte** entre la función y las variables globales. El diseño del software debe tratar de reducir el acoplamiento lo más posible para incrementar la posibilidad de **reuso**.

El acoplamiento le resta **portabilidad** a las funciones y al programa. Recorremos que una función es portable si puede funcionar sin cambios con cualquier programa invocante.

Por otra parte, las variables globales, por ser accesibles desde cualquier lado, están expuestas a ser modificadas por error. Por ejemplo, dos funciones que no tengan relación entre sí, y utilicen variables globales con el mismo nombre, podrán interferir erróneamente entre ellas. Esto conduce a errores de muy difícil detección.

FUNCIONES EN EL LENGUAJE C

El siguiente es el formato general de declaración de una función:

```
[tipo del retorno] Nombre_de_función (declaración de parámetros)
{
    Declaración de variables locales
    Cuerpo de la función
}
```

El **tipo del retorno** indica de qué tipo es el dato que devolverá la función. Si ésta devuelve un valor entero (`int`), no será necesario indicarlo, pues éste es el *default* que toma el compilador³. Por esta razón figura el término entre corchetes, pero cabe indicar que si el tipo retornado por la función es otro, su especificación es obligatoria.

Se recomienda incluir siempre la especificación de tipo retornado por una función, aún cuando este tipo sea `int` y su inclusión no sea estrictamente necesaria por tratarse del default.

Puede notarse la similitud de este formato con el del `main`, en donde sólo cambia el nombre de la función por “`main`”. No es de extrañar esta situación dado que el programa principal `main` es una función más (que también puede recibir argumentos de su programa invocante, el sistema operativo o algún “proceso padre”, como estudiaremos más adelante).

RETORNO DE LA FUNCIÓN

La función finaliza su ejecución cuando llega a su llave final, o bien cuando encuentra la sentencia `return`. Cabe aclarar que `return` no es una función sino una sentencia o instrucción, y por ello no lleva paréntesis.

La sentencia `return` tiene dos aplicaciones. Permite salir de la función en cualquier lugar. Esto es, se pueden colocar mas de un `return` en una función. Cualquiera de ellos que se ejecute provocará el retorno al programa invocante. Esta modalidad atenta contra la regla de estructuración que propone que cada bloque (en este caso la función) sólo tenga un punto de entrada y uno de salida, pero en algunos casos es sumamente utilizada.

La otra aplicación de `return` es la transferencia del resultado de la función al programa llamante. Esto nos permite **retornar un valor**.

³ Esto es así en el estándar C89 que utilizamos en este libro. Posteriores versiones han eliminado este valor por defecto y han hecho obligatoria la declaración explícita del tipo de retorno, aunque fuera `int`.

El formato es: `return valor_devuelto;`

A continuación se desarrollarán varios ejemplos donde se puede observar lo expuesto anteriormente y de los que se pueden extraer nuevas conclusiones.

EJEMPLO: VARIABLES LOCALES HOMÓNIMAS

```
#include <stdio.h>
int main ( void )
{
    int A ; // variable local del main
    A = 2 ;
    funcion(); // Se invoca a la función llamada "funcion"
    printf ("\n main %d", A);
}

funcion ( void )
{
    int A ; // variable local de la función
    A = 3 ;
    printf ("\n funcion %d", A);
}
```

La ejecución produjo el siguiente resultado:

```
función 3
main 2
```

Este resultado demuestra que las dos variables llamadas `A` son dos variables diferentes, una local del `main` y la otra local de la función, caso contrario, la carga del `3` en la función habría modificado la del `main`, y no fue así. Además se comprueba que la variable local se *destruye* al retornar de la función.

Es importante notar que el **llamado** o invocación de una función se realiza a través de su nombre y un juego de paréntesis. Es importante definir adecuadamente los nombres de las funciones, ya que luego es muy difícil cambiarlo, puesto que no basta con modificar su declaración sino que habrá que corregir cada una de las invocaciones.

EJEMPLO: VISIBILIDAD DE LAS VARIABLES LOCALES

En este caso se usará el mismo programa principal del ejemplo anterior, pero se suprimirá la declaración de la variable local de la función.

```
funcion ( void )
{
    A = 3; // Error de compilación !!
    printf ("\n funcion %d", A);
}
```

Esto ocasiona un **error de compilación** en la línea indicada debido a que la variable `A` no es reconocida. Esto es así porque no está declarada dentro del **ámbito** de la función.

Esto confirma que la variable `A` que se declaró en el `main` no es válida dentro de la función.

VARIABLES LOCALES A UN BLOQUE

Es posible declarar una variable dentro de un **bloque** (sección de código encerrada entre llaves) al principio del mismo, aunque este bloque se encuentre en la mitad de una función o programa.

La variable así declarada se comporta como **local al bloque**, es decir, limita su visibilidad a éste, y su tiempo de vida se reduce a la vigencia del bloque.

Sin embargo, el bloque no es una función y mantiene, por lo tanto, la visibilidad de las variables declaradas en la función donde él se encuentra (excepto si alguna es homónima con otra declarada local al bloque).

Esta particularidad es fundamental en la construcción de **macros**, dado que éstas pueden necesitar utilizar variables propias, y como la macro es en definitiva una incrustación en el código, las variables declaradas en ellas quedarían fuera de lugar, dando origen a errores de compilación.

Este problema se soluciona construyendo la macro dentro de un bloque, y haciendo que las variables declaradas sean locales a él.

EJEMPLO: VARIABLES LOCALES A UN BLOQUE

```
#include <stdio.h>
int main ( void )
{
    int A , B ;
    A = 2 ;
    B = 20 ;
    { /* Inicio del bloque */
        int A ;
        A = 5 ;
```

```

        printf ("A del bloque es %d \n", A);
        printf ("B en el bloque es %d \n", B);
        B++ ;
    }      /* Fin del bloque */
    printf ("%d %d" , A , B );
}

```

En este ejemplo se declaran variables `A` y `B` locales al programa principal, y posteriormente se declara otra variable `A`, local a un bloque.

La ejecución del programa da como resultado:

```

A del bloque es    5
B en el bloque es 20
2   21

```

Este comportamiento indica que la variable `B` fue válida en todo el programa, mientras que la variable `A`, declarada en `main()`, no fue visible en el bloque.

La variable `A` declarada en el bloque solamente tuvo validez en él. Esta variable se guardó en un área de la pila dedicada al bloque, y se destruyó al salir de él.

Por esta razón la impresión fuera del bloque arrojó el valor 2 y no 5 que fue el último valor cargado.

EJEMPLO: VARIABLES LOCALES Y GLOBALES CONFLICTIVAS

Hemos dicho que una variable global es válida en todas las funciones, mientras que las variables locales sólo lo son dentro de la función donde fueron declaradas. ¿Qué ocurriría si en un programa existiera una variable local y una global con el *mismo* nombre?

Es de suponer que sólo una de ellas sería válida dentro de la función, pero ¿cuál? Podemos averiguarlo con el siguiente programa:

```

#include <stdio.h>
int A;          /* Declaración de A como variable global */

int main (void)
{
    printf("Valor inicial de A = %d", A );
    A = 2 ;
    funcion();    /* Llamado a la función */
    printf("\nValor de A en el main es %d", A );
}

```

```
funcion (void)
{
    int A ;      /* Declaración de A como variable local */
    A = 3 ;
    printf ("\nEl valor de la A local es %d", A );
}
```

Resultado en pantalla:

```
Valor inicial de A = 0
El valor de la A local es 3
Valor de A en el main es 2
```

Puede verse que en el programa principal `main`, donde no se declaran variables, es reconocida y modificada la variable global `A`. También se comprueba que su valor inicial es cero.

En cambio, dentro de la función, la variable que fue reconocida fue la local. Esto determina una excepción en el **alcance** de las variables globales, como se había mencionado oportunamente.

¿Podemos predecir cuál habría sido la impresión en caso de omitirse la declaración de `A` como variable local de la función?

En vista del anterior ejemplo, podemos reformular la expresión para el alcance de las variables globales como sigue:

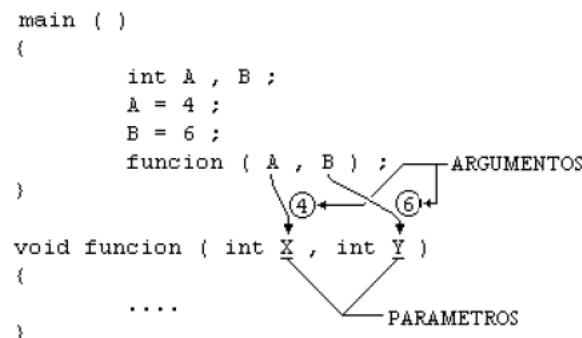
Las variables globales son válidas (visibles y modificables) en todas las funciones, excepto en aquellas donde se declaró una variable local con el mismo nombre de la global.

ARGUMENTOS Y PARÁMETROS

Recordemos que el programa invocante es capaz de transferir datos (que reciben el nombre de **argumentos**) a las funciones, utilizando áreas individuales de memoria. Dicho de otro modo, pueden hacer una transferencia de valores a algunas variables locales de la función.

Estas variables locales deben recibir un tratamiento formal específico y reciben el nombre de **parámetros**. Este tratamiento formal exige que se respete el orden de transferencia, y el tipo de dato transferido debe coincidir con el recibido. Por otro lado veremos que lo que se transfiere es el **valor** contenido en las variables.

El siguiente esquema ilustra la diferencia entre argumentos y parámetros. En él se muestra la transferencia que se hace desde el programa principal a una función, de los contenidos de las variables A y B. Estos contenidos son recibidos en los parámetros formales X e Y.



EJEMPLO: FUNCIÓN SUMA

En los siguientes ejemplos se muestra una simple función de suma que recibe dos argumentos enteros y retorna la suma de ambos. Obsérvese la manera en que el programa principal recibe el valor retornado por la función.

```

#include <stdio.h>
int main ( )
{
    int A , B , C;
    A = 5;
    B = 9;
    /* C capturará el valor retornado por la función SUMAR */
    C = SUMAR(A , B);
    printf ( "\n %d + %d = %d " , A , B , C );
}

int SUMAR ( int X , int Y )
{
    int Z;
    Z = X + Y;
    return Z;
}
  
```

Impresión :

5 + 9 = 14

Podemos ver en este programa que los **nombres** de los parámetros de la función no tienen por qué corresponder a los de las variables transferidas.

En el programa y la función se utilizaron variables redundantes a fin de presentarlo más claramente, pero el mismo efecto se obtendría del siguiente programa, más compacto.

EJEMPLO: OTRA FUNCIÓN SUMA

```
#include <stdio.h>
int main ()
{
    int A = 5 , B = 9 ;
    printf ("\n %d + %d = %d ", A , B , SUMAR(A, B));
}

int SUMAR (int X , int Y)
{
    return X + Y;
}
```

Nótese que `return` está devolviendo una expresión, la cual se evalúa en primera instancia (se calcula) y luego se transfiere el valor a la función.

En `main` el valor retornado no es capturado en una variable (mediante el operador de asignación, como en el programa anterior) sino que es utilizado directamente como argumento para enviar a la función `printf()`.

DESAPAREAMIENTO DE TIPOS EN TRANSFERENCIA DE ARGUMENTOS

Es necesario ser muy cuidadoso en la correspondencia de los **tipos** de argumentos transferidos y los de los parámetros formales que los reciben.

Cuando se transfieren contenidos con el operador de asignación directamente de una variable numérica de algún tipo, hacia otra de otro tipo, es de esperar cierta coherencia en los valores transferidos. Esto ya ha sido analizado en un capítulo previo.

Esto no ocurre en la transferencia de argumentos a función, en los cuales la copia no es directa. Al invocarse la función, el programa invocante “deja” los valores (argumentos) en la pila con el formato que les corresponde.

La función “toma” estos valores de la pila para colocarlos en los parámetros formales. Esto lo realiza con el formato de los parámetros, desconociendo totalmente el formato original.

Claro está que si los formatos coinciden, como es de esperar, no se producirá error alguno. Pero si esto, por alguna razón no es así, no se produce ninguna conversión de formato y los valores capturados resultarán erróneos sin mediar advertencia alguna del compilador. Recordemos que la filosofía de diseño del lenguaje C es que “el programador sabe lo que está haciendo”.

FUNCIONES ANIDADAS

El lenguaje C posee muy pocas palabras reservadas o instrucciones directas. Su potencia está dada por la gran cantidad de funciones que posee en sus bibliotecas, a las que se suman las que se pueden ir agregando como aporte del usuario y de terceros.

Cada tarea que se emprenda en C involucrará seguramente funciones. Pero estas tareas tendrán frecuentemente su solución en forma de otras funciones que contengan a aquellas.

El caso de funciones que contienen llamados a otras funciones se denomina **anidamiento**, y se llama a éstas **funciones anidadas**. La profundidad con que ingresamos llamando a una función dentro de otra, recibe el nombre de **nivel de anidamiento**.

De esta manera, las funciones más complejas estarán sustentadas en otras más simples. Un programa se puede construir partiendo de la estructura más compleja, dejando el detalle de las funciones más simples para una segunda instancia (se denomina a esta técnica desarrollo top-down), o bien, construyendo en primer término las funciones más simples y, basándose en éstas, llegar a las más complejas (técnica de desarrollo bottom-up).

EJEMPLO: DESARROLLO MODULAR BOTTOM-UP

En este ejemplo se mostrará un desarrollo utilizando la técnica bottom-up (de lo particular se llegará a lo total) planteando enunciados parciales que conducirán a funciones que serán utilizadas en enunciados más complejos, dando como resultado un sistema de funciones anidadas.

Enunciado 1: Realizar una función que reciba un entero de una cifra y retorne su cubo.

```
int CUBO ( int A )
{
    return A * A * A ;
}
```

Enunciado 2: Realizar una función que reciba 3 enteros de una cifra y retorne la suma del cubo de los tres, utilizando la función anterior.

```

int SUM_CUBO ( int X , int Y , int Z )
{
    int W;
    W = CUBO (X) + CUBO (Y) + CUBO (Z);
    return W;
}

```

Enunciado 3: Realizar una función que reciba un entero de 3 cifras y devuelva un 1 en caso que se cumpla que *dicho número es igual a la suma del cubo de sus cifras individuales*, y retorne un 0 en caso contrario. Utilizar para esto la función del enunciado anterior.

```

int CUBOS ( int N )
{
    int A , B , C ;

    // Separación en las tres cifras
    A = N / 100 ;
    C = N % 10 ;
    B = ( N / 10 ) % 10 ;

    if ( N == SUM_CUBO (A, B, C) ) return 1;
    else return 0;
}

```

Enunciado Final: Determinar cuáles son los números de tres cifras (que no comiencen con cero) tales que *la suma del cubo de cada una de sus cifras individuales sea igual al número propuesto*.

Es decir aquellos que cumplen: $ABC = A^3 + B^3 + C^3$

Siendo ABC un número de tres cifras (y no un producto). Existen cuatro de estos números, quedando para el lector determinar cuáles son. Se da a continuación el programa completo :

```

#include <stdio.h>
int main()
{
    int J ;
    for ( J = 100 ; J < 1000 ; J++ )
        if ( CUBOS (J) )
            printf ("%d \n" , J ) ;
}

```

```

int CUBOS ( int N )
{
    int A , B , C ;
    /* Separación en las tres cifras */
    A = N / 100 ;
    C = N % 10 ;
    B = ( N / 10 ) % 10 ;
    if ( N == SUM_CUBO(A, B, C) ) return 1 ;
    else return 0 ;
}

int SUM_CUBO ( int X , int Y , int Z )
{
    int W;
    W = CUBO (X) + CUBO (Y) + CUBO (Z) ;
    return W ;
}

int CUBO ( int A )
{
    return A * A * A ;
}

```

FUNCIONES QUE DEVUELVEN VALORES NO ENTEROS

Como se mencionó previamente, el compilador C supone que la función devolverá un valor de tipo **entero**. Si esto no es así, será necesario realizar la indicación del tipo devuelto en dos lugares:

- En la declaración de la función.
- Antes de invocarla.

La indicación del tipo devuelto en la declaración de la función se muestra en el siguiente ejemplo:

```

float superficie ( float radio )
{
    float sup ;
    sup = 3.14 * radio * radio ;
    return sup ;
}

```

También es necesario hacerle saber al compilador que la función retornará un valor no entero, antes de realizar una llamada desde el programa invocante.

De no ser así, el compilador supondrá un valor devuelto entero, asignado a una variable (o utilizado como una variable) de otro tipo, resultando de esto un *type mismatch* o error de **desapareamiento** de tipos.

Esto puede ser solucionado de dos formas :

- Definiendo la función antes que la función invocante.
- Utilizando un especificador explícito de tipo devuelto.

EJEMPLO: FUNCIÓN QUE RETORNA UN FLOTANTE I

En este ejemplo se recibe el radio de un círculo en formato flotante y se utiliza una función que calcula su superficie, retornando este valor en formato también flotante.

La función se define en este caso *antes* del `main`, utilizando la primera de las soluciones propuestas anteriormente.

```
#include <stdio.h>
float superficie (float radio)
{
    float sup;
    sup = 3.14 * radio * radio; //pi por radio al cuadrado
    return sup;
}

int main ( )
{
    float radio ;
    printf ( "Ingrese el radio: " );
    scanf ( "%f" , &radio ) ;
    printf ( "\n\n Radio = %4.2f Superficie = %4.2f" , \
             radio , superficie(radio) ) ;
}
```

De todas formas, por una cuestión de orden y facilidad de lectura del código fuente, preferimos definir las funciones *después* del `main()`. Para poder realizar esto con funciones que devuelven valores no enteros debemos recurrir a la especificación o declaración explícita de tipo devuelto, utilizando los *prototipos*.

PROTOTIPO DE UNA FUNCIÓN

El **prototipo** de una función es una declaración que se realiza para que el compilador conozca el tipo de retorno y el tipo de los argumentos de una función que será declarada concretamente más adelante (es decir, su cuerpo aparece más adelante en el código fuente).

La sintaxis del prototipo es como sigue:

```
tipo_devuelto nombre_funcion (tipo, tipo, ... , tipo);
```

Si posteriormente no concuerda el prototipo con la utilización de la función o con su definición, se producirá un error de compilación, lo cual, lejos de ser perjudicial, es deseable, pues de esta forma podemos notarlo y corregirlo antes de que se produzca en tiempo de ejecución donde su detección y corrección es mucho más problemática y costosa.

Enumerando sus ventajas tenemos:

- Ayuda a atrapar fallos antes de que se produzcan.
- No permite llamadas a función con argumentos erróneos.
- No es un error no incluirlo, pero al poner la declaración explícita de tipo devuelto como obligatoria, no es un perjuicio incluir el prototipo.

EJEMPLO: FUNCIÓN QUE RETORNA UN FLOTANTE II

Se repetirá el ejemplo anterior utilizando esta vez el prototipo.

```
#include <stdio.h>
float superficie ( float ) ;           /*  Prototipo  */

int main ( )
{
    float radio ;
    printf ( "Ingrese el radio " ) ;
    scanf ( "%f" , &radio ) ;
    printf ( "\n\n Radio = %4.2f    Superficie = %4.2f " , \
              radio , superficie(radio) ) ;
}

/* Declaración de la función */
float superficie ( float radio )
{
    float sup ;
    sup = 3.14 * radio * radio ;
    return sup ;
}
```

FUNCIONES QUE RETORNAN VOID

El tipo de dato **void**, pese a ser un tipo de dato “nulo”o “vacío”, es decir que no se pueden declarar variables con él, es un tipo de dato no entero. Las funciones que devuelven este tipo de dato, deben recibir el mismo tratamiento que cualquier otra que retorne un **no entero**.

En nuestros ejemplos, y además habitualmente, el programa **main()** toma sus datos de teclado y envía sus resultados a pantalla o impresora, es decir, no tiene argumentos ni retorna valores al programa invocante.

Lo correcto parece ser entonces declarar la función como `void main()`. Sin embargo, el compilador puede configurarse para emitir una advertencia (*warning*) si el tipo de retorno declarado de `main` no es `int`. Esto es así porque el estándar POSIX de sistemas operativos indica que `main` debe retornar un entero, que es utilizado por el sistema operativo (que sería el llamante de `main`) para tomar decisiones posteriores.

De la misma forma, si declaramos nuestra función principal como `int main()` el compilador puede emitir un *warning* en caso de alcanzar el final de la función y no encontrar ningún `return` que, efectivamente, retorne un entero. Por esta razón es normal que se utilice este esquema cuando se trabaja con conformidad estricta de los estándares (tanto C como POSIX):

```
int main() {  
    //sentencias del programa  
    // . . .  
  
    return 0;    //0 indica que el programa finalizó normalmente  
}
```

Creemos que es necesaria esta aclaración porque distintos IDEs configurados con distintos compiladores pueden dar *warnings* y errores diferentes, de acuerdo a cuál sea la configuración puntual del compilador.

PROTOTIPOS Y ARCHIVOS CABECERA

Cuando se utilizan funciones de la biblioteca estándar de C es necesario “incluir” determinados **archivos cabecera** (*header*) mediante la instrucción al preprocesador llamada `#include`, seguida del nombre del archivo cabecera, los cuales tienen la extensión `.h`. Esto ya ha sido observado en nuestros programas de ejemplo.

En dichos archivos de cabecera se encuentran los prototipos de las funciones de la biblioteca. Por esta razón, si se utiliza la función relacionada, será necesario incluir la cabecera correspondiente. De esta manera el compilador no encontrará llamados a funciones desconocidas, como hemos analizado en el apartado sobre prototipos, en cuanto a la necesidad de su existencia cuando declaramos nuestras funciones luego de `main`. Estamos ante la misma situación.

Cabe mencionar que algunos compiladores incluyen automáticamente los archivos de cabecera de la biblioteca estándar de C, por lo que es posible que el código compile exitosamente aún si omitimos los `#include` correspondientes. Esta situación no es recomendada, ya que afecta la portabilidad del código, que puede no compilar en otro compilador configurado para exigir estrictamente las cabeceras. Si utilizamos funciones desarrolladas por terceros, es decir no de la biblioteca estándar, la inclusión de los prototipos será obligatoria y lo más probable es que los prototipos se nos provean agrupados dentro de un archivo

cabecera que indicaremos con `#include` de la misma forma (aunque en lugar de los signos <> se usan comillas dobles para la ruta y nombre del archivo).

La instrucción al preprocesador `#include` sencillamente introduce el código fuente del archivo cabecera especificado en el lugar donde se lo coloca, de manera que es como si los prototipos estuvieran escritos allí cuando el compilador empieza a realizar su tarea. Con otras palabras es como “copiar-pegar” del archivo cabecera completo. Recordemos que el preprocesador se ejecuta antes de iniciar la compilación.

TRANSFERENCIA POR VALOR Y POR REFERENCIA

Hasta ahora hemos utilizado transferencias de argumentos donde el valor transferido *se copia* en un parámetro formal (variable local) de la función. Por lo tanto, en un determinado instante existirán dos variables conteniendo el mismo valor: una, la variable de origen, y la otra, el parámetro de la función. Se ve de esta manera que lo que se transfiere (realizando una copia) es el **valor** de la variable del programa invocante. De igual forma se puede realizar la transferencia de constantes.

A esta modalidad se le denomina **transferencia o pasaje por valor** del argumento que toma la función.

Es necesario destacar que dentro de la función se trabaja sobre la variable local de la función, es decir, no sobre el “original”, el cual, al producirse el retorno de la función, no queda modificado.

Otra forma de realizar el traspaso del argumento lo constituye la **transferencia o pasaje por referencia**.

Esto consiste en transferir, no el valor del dato en sí, sino una “pista” o referencia que permita ubicar al dato, una vez dentro de la función invocada. La referencia transferida es la **dirección de memoria** del dato.

Es fundamental notar en este punto que no existe *copia* del valor del dato propiamente dicho, y que cualquier modificación que se realice se está haciendo sobre el original ya que trabajamos con una referencia directa a él.

Por otro lado, si el dato a transferir es muy voluminoso, como podría serlo una **estructura** (que estudiaremos en un capítulo posterior), se ahorra memoria al no duplicarlo, y se ahorra asimismo el tiempo que demanda esta transferencia dado que lo único que se copia es la dirección en donde se encuentra el dato.

Es necesario recibir la dirección transferida en una variable de tipo **puntero**, tema que se desarrollará en otro capítulo.

LA TRANSFERENCIA POR REFERENCIA NO EXISTE

Cuando se realiza una transferencia por referencia, lo que se está transfiriendo es una *dirección*, la cual es un tipo de dato como cualquier otro de C.

Esta dirección se está tomando (copiando) en una variable local (parámetro formal) de la función capaz de recibirla. Estas variables son de tipo puntero (variables que contienen direcciones de memoria), por lo tanto, lo que en realidad se está realizando es una transferencia **por valor** del argumento “dirección”.

Puede resultar cómodo o conveniente realizar una abstracción mental de tal situación y considerar que el argumento transferido es en realidad el que ocupa la dirección en cuestión y la transferencia se realiza **por referencia**.

Como esto no modifica el efecto final del procedimiento, el usuario puede adoptar el modo de interpretarlo que le resulte más apropiado.

El lenguaje C simula la transferencia “por referencia” de argumentos realizando una transferencia “por valor” de la dirección de éstos.

RECURSIVIDAD

La **recursividad** es una poderosa herramienta de programación, que en muchos casos permite resolver problemas de manera sencilla y natural, cuando utilizar herramientas iterativas genera soluciones mucho más complejas.

La recursividad consiste en permitir que una función *se invoque a sí misma*.

Podemos decir que una función es **recursiva**, si en alguna parte de su código hay una invocación (directa o indirecta) a sí misma. O dicho más simplemente, una función es recursiva si se llama a sí misma.

Se completará la exposición de este tema, a modo de cierre, con un ejemplo que muestre una función recursiva.

EJEMPLO: CÁLCULO RECURSIVO DEL FACTORIAL

Tal vez uno de los ejemplos más claros y simples de una función recursiva es la que calcula el factorial de un número entero.

La aplicación del método recursivo es inmediata debido a que la definición misma del factorial también es recursiva.

Recordemos que el factorial de un número entero N, representado como $N!$, se define como:

$$\begin{aligned}N! &= N \times (N-1)! \\0! &= 1\end{aligned}$$

```
/* Calculo del factorial de N en forma recursiva */
#include <stdio.h>
int factorial (int);

int main ( )
{
    int N ;
    scanf ( "%d" , &N ) ;
    printf ( "\n %d ! = %d " , N , factorial(N) );
}

int factorial(int N)
{
    int F = 1 ;
    if ( N != 0 ) F = N * factorial(N-1);
    return F ;
}
```

Se sugiere al lector realizar el seguimiento de este programa para valores pequeños de N, a fin de comprender la naturaleza de la invocación recursiva.

PROBLEMAS PROPUESTOS

1. En el siguiente programa, se puede comprobar que la variable `A` local al bloque es diferente de la variable `A` local al `main`.

```
#include <stdio.h>
int main ()
{
    int A , i ;
    for ( i = 1 ; i < 4 ; i++ )
    {
        int A ;
        A++ ;
        printf ("\n A del bloque es %d " , A ) ;
    }
    printf ("\n %d " , A ) ;
}
```

Los valores iniciales carecen de sentido pues son variables locales no inicializadas, pero es notable que la variable `A` del bloque respete el incremento.

Se proponen dos explicaciones para este hecho :

- La variable `A` solamente se declara una vez dentro del bloque y en las restantes iteraciones dicha declaración se ignora.
- El funcionamiento dinámico de la pila hace que siempre ocupe el mismo lugar de memoria, en este caso manteniendo el valor allí guardado.

¿Qué opina el lector?

2. En el siguiente programa se está cometiendo un error al transferir un entero a la función y recibirla en un parámetro de tipo `char`.

```
#include <stdio.h>

funcion ( char A )
{
    printf ("\nFuncion %d " , A ) ;
}

int main( )
{
    int A ;
    A = 1857 ;
    funcion ( A ) ;
    printf ("\nMain %d " , A ) ;
}
```

Predecir los valores que se mostrarán en pantalla y verificarlos posteriormente ejecutando el programa.

3. Se desea realizar un programa que utilice una función que calcule y retorne el valor de X^Y (X elevado a la potencia Y), siendo X e Y enteros positivos, y por lo tanto, dando un resultado también entero.

Debido al rápido crecimiento de este valor será necesario retornar el valor en un unsigned long.

4. Construir una función que reciba un entero positivo y retorne 1 si éste es primo y 0 en caso contrario.
5. Utilizar la función del problema anterior para construir otra función que reciba dos enteros positivos y retorne la cantidad de números primos que se encuentran en el rango determinado por ellos.
6. Utilizar la función del problema anterior para mostrar las cantidades de números primos que hay en los rangos [1000-2000] , [2000,3000] y [3000,4000].
7. Construir una función que reciba un entero positivo y retorne 1 si éste es perfecto y 0 si no lo es. Utilizarla para mostrar los primeros 5 números perfectos.

Número perfecto: aquel entero cuyo valor es igual a la suma de sus divisores exactos, excluyendo al número mismo. Por ejemplo: $6 = 3 + 2 + 1$

8. Construir una función que reciba dos números enteros positivos y retorne un 1 si estos constituyen un par de números amigos, y 0 si no lo son. Utilizarla para mostrar algún par de números amigos.
Números amigos: dos números enteros son amigos si cada uno de ellos es igual a la suma de los divisores exactos del otro, exceptuando al número mismo.
9. Construir una función que lea un carácter de teclado y lo escriba en pantalla colocando el carácter cuyo código ASCII es el siguiente al recibido.

5. VECTORES

Frecuentemente se presenta la necesidad de almacenar una cierta cantidad de datos que presentan la misma característica.

Hasta el momento la solución al problema del almacenamiento de información fue dado por el uso de variables. Pero éstas no pueden solucionar una amplia gama de situaciones.

Consideremos el siguiente problema: *Ingresar por teclado 10 valores enteros y mostrar cuántos de éstos superan el promedio.*

Ya en este enunciado tan simple tenemos una dificultad: para contar los valores que superan el promedio, es necesario conocer dicho promedio. Para ello es necesario calcularlo. Y para esto debemos ingresar la totalidad de los valores (!).

Podemos solicitarle al operador que reingrese los valores una vez calculado el promedio, pero seguramente no se mostrará muy complacido, y claramente no representa la solución al problema.

Otra variante es asignar una variable a cada valor ingresado, pero esto ocasionará la repetición del código de lectura de valores en el programa, tantas veces como datos ingresen. Ahora imaginemos que en lugar de 10 valores fueran 1000... Claramente no es ésta la solución.

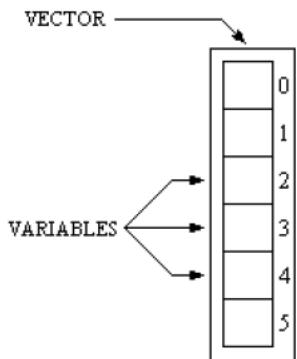
Se necesita una forma de almacenar esta serie de valores de manera repetitiva y eficaz. Esta forma de almacenamiento debe ser resuelta en una iteración y no mediante una secuencia de lecturas.

La solución la aporta una estructura de datos llamada *array*, frecuentemente traducida al castellano como **arreglo** o **vector**. Utilizaremos esta última.

Un vector es un conjunto de variables del mismo tipo referenciadas por un nombre común e individualizadas mediante un subíndice numérico.

Según la definición el vector es un conjunto o colección de “contenedores” o variables. Toda esta estructura tendrá un nombre que la identifique.

Esto significa que cada una de las variables se llamará de la misma forma.



Su nombre es el mismo que el del vector, pero se diferencia de las otras variables mediante una segunda identificación, en este caso numérica.

La figura de la izquierda nos muestra una representación gráfica de un vector.

Vemos que el vector está representado por la estructura completa mientras que cada una de las casillas numeradas es una variable.

DECLARACIÓN DE VECTORES

`Tipo Nombre [Tamaño] ;`

Donde:

- **Tipo** Tipo de dato que será el tipo de las variables
- **Nombre** Nombre del vector
- **Tamaño** Cantidad de variables que contendrá el vector.

Ejemplos de declaración de vectores:

```
int VEC [6]; // un vector de 6 enteros
float V [4]; // un vector de 4 flotantes
```

El tamaño del vector debe ser una constante del programa, es decir que el compilador debe encontrar una constante dentro de los corchetes, y no una variable, para poder realizar la compilación.

Las variables que componen el vector se refieren con el nombre del vector y un **subíndice** numérico entre corchetes (`[]`). Estas variables del vector se manejan de forma análoga a cualquier otra variable de C.

Ejemplos de declaración, lectura y escritura de vectores:

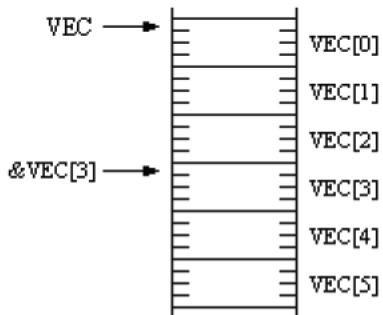
```
int VEC[6]; //declaración
VEC[2] = 4; //asignación a uno de los int del vector
```

```
VEC[5] = 3 + VEC[2]; //asignación de otro int con una expresión  
scanf("%d", &VEC[3]); //uso con scanf
```

Nótese que `VEC[3]` es una variable entera y se requiere el operador `&` (*dirección de*) en la función `scanf()` a fin de obtener su dirección.

EL VECTOR EN LA MEMORIA

El vector se dispone en la memoria como una secuencia de variables del tipo indicado en la declaración.



El nombre del vector en realidad representa su **dirección de inicio**, como se muestra en la figura de la izquierda.

Considerando el vector de enteros que se declaró anteriormente, se ve que cada variable está formada por 4 bytes.

Cada vez que se accede a una de las variables del vector, es necesario para el procesador conocer su **dirección**.

Dicha dirección se obtiene de la siguiente forma:

$$\&VEC[I] = VEC + I * \text{sizeof(TIPO)}$$

Donde:

- `I` representa el subíndice de la variable que se intenta acceder
- `TIPO` es el tipo de dato de los elementos del vector
- `sizeof()` indica cuántos bytes ocupa en memoria un tipo

Si en el caso del ejemplo anterior la variable a acceder fuera `VEC[3]`, para encontrarla habría que desplazarse a la dirección `VEC+12`, es decir 12 bytes a partir de la dirección indicada por `VEC`. Con otras palabras, en 12 bytes entran tres `int` ($3 \text{ int} * 4 \text{ bytes}$), por lo que `VEC[3]` debe acceder al cuarto `int` de la secuencia, que se encuentra 12 bytes más adelante que el inicio del vector. Recordemos que `VEC` (sin corchetes) es para el compilador de C la *dirección de inicio del vector*.

Este cálculo que realiza el procesador recibe la denominación de **aritmética de subíndices**.

Para que esta aritmética pueda funcionar correctamente deben cumplirse ciertas condiciones en el vector:

- Las variables del vector deben ubicarse en memoria en forma consecutiva (sin espacios intermedios).
- Al **primer elemento** del vector le corresponde el subíndice **0**.
- Los elementos del vector son del mismo tipo (y por lo tanto ocupan igual cantidad de bytes).
- El vector *crece* hacia posiciones superiores de memoria.

SUBÍNDICE NUMÉRICO

La verdadera ventaja que presentan los vectores reside en el subíndice numérico. Este puede ser manejado aritméticamente de manera de automatizar la manipulación del vector.

Cuando se hace referencia a una variable del vector se debe expresar el nombre del vector que la identifica y el subíndice que la individualiza. Pero el subíndice puede ser representado por una variable entera, y ésta a su vez puede tomar diferentes valores. En términos generales, dentro del corchete se puede utilizar cualquier **expresión**. El programa calculará el valor de la expresión y con esto realizará la aritmética de la dirección para encontrar esa posición del vector.

De esta manera, la misma sintaxis de código puede hacer referencia a distintas variables del vector. Por ejemplo en el vector de los ejemplos anteriores, **VEC[I]** representará distintos elementos del vector según el valor que asuma la variable entera **I**.

EJEMPLO: MANEJO BÁSICO DE UN VECTOR

Se declarará un vector de 10 enteros, se ingresarán los valores por teclado y posteriormente se los mostrará en pantalla.

Nótese que los valores se ingresan y quedan almacenados, todos. A partir de este hecho se puede realizar cualquier tarea con ellos.

```
#include <stdio.h>
#define N 10

int main()
{
    int VEC[N] , I ;
    for( I=0 ; I<N ; I++ ) {
        printf( "\n VEC[%d] = " , I ) ;
        scanf ( "%d" , &VEC[I] ) ;
    }

    printf("\n\n");
    for( I=0 ; I<N ; I++ )
        printf ( "%7d" , VEC[I] ) ;
}
```

Obsérvese que el **tamaño** en la declaración del vector es una **constante**.

Este requerimiento no es caprichoso. Se podría pensar que una variable declarada y asignada previamente a la declaración del vector (es decir, antes en el código fuente: más arriba) podría permitir obtener un vector del tamaño indicado por el usuario.

Sin embargo esto no es así, debido a que la asignación de la variable ocurre en *tiempo de ejecución* mientras que el tamaño del vector debe ser conocido en *tiempo de compilación*, es decir, mucho antes (en tiempo, no *geográficamente* como se ve en el código). Es muy importante notar que en el código fuente quedan mezcladas “instrucciones” que son para el preprocesador (los `#define`), para el compilador (las declaraciones) y además sentencias (la mayor parte de las líneas) que se ejecutarán una vez traducido al programa final por parte del microprocesador. Esas tres fases ocurren en sucesivos momentos.

En estándares posteriores a **ANSI C89** se permite la declaración de un vector con una variable en su tamaño, pero sólo para ciertos casos puntuales. En términos generales, y siguiendo nuestra regla de presentar el lenguaje para su aprendizaje en su versión ANSI C89 decimos que, estrictamente, *el tamaño del vector debe ser una constante en tiempo de compilación*.

```
int A;
scanf("%d", &A);
float VEC[A]; //estándar ANSI C89: no!
```

ASIGNACION DE VALORES EN LA DECLARACION

Puede resultar cómodo y conveniente en algunos casos, asignar valores a las variables del vector en el momento de la declaración.

Si se asigna la totalidad de las variables es posible, incluso, omitir el tamaño del vector. Dicho tamaño será el adecuado para contener los valores.

Sintaxis:

```
tipo VEC[] = {valor, valor, ... , valor};
```

Ejemplo:

```
int VEC[] = {5, 7, 9, 1, 3, 4};
```

COPIA DE VECTORES

Puede ocurrir que, por algún requerimiento del programa, se necesite igualar los contenidos de dos vectores de similares características.

Resultaría natural intentar realizar una asignación directa a través del operador de asignación como se muestra a continuación:

```
int VEC1[5];
int VEC2[] = {1,2,3,4,5};
VEC1=VEC2;      /* ERROR!! */
```

Sin embargo, esta última línea de código genera un error de compilación :

“Lvalue required”

Esta frase significa que se necesita una **variable** a la izquierda (*left value*) de la asignación, y el vector no lo es.

El vector en sí mismo no es una variable y por lo tanto no puede ser asignado directamente.

El **nombre del vector** representa la dirección de su comienzo, pero no es una variable. Podemos considerarlo como un **puntero constante**, o una constante de tipo puntero. Por tratarse de una constante es que no puede aparecer a la izquierda del operador de asignación, ya que una constante no puede ser modificada, naturalmente.

La copia de vectores deberá realizarse variable por variable. Pero ésto no representa ninguna complicación en el código, como se ve a continuación:

```
int VEC1[5] , I ;
int VEC2[] = {1,2,3,4,5};

for ( I=0 ; I<5 ; I++ )
    VEC1[I] = VEC2[I];      /* correcto */
```

EJEMPLO: VALORES SUPERIORES AL PROMEDIO

Ya estamos en condiciones de resolver el problema planteado en un principio: “*Ingresar por teclado 10 valores enteros y mostrar cuántos de estos valores superan el promedio*”.

Si bien este programa podría ser más corto y con utilización de menor cantidad de variables, se prefirió separar los diferentes bucles y aplicaciones en pos de una mayor claridad.

```

#include <stdio.h>
#define N 10
int main()
{
    int VEC[N],I;
    int ACUM = 0 , CONT = 0 ;
    float PROM ;

    /* Ingreso de valores */
    for ( I=0 ; I<N ; I++) {
        printf ( "\nVEC[%d] = " , I ) ;
        scanf ( "%d" , &VEC[I] ) ;
    }

    /* Cálculo del promedio */
    for ( I=0 ; I<N ; I++ )
        ACUM = ACUM + VEC[I] ;
    PROM = (float)ACUM / N ;

    printf ( "\n\n" ) ;

    for( I=0 ; I<N ; I++ ) {
        printf ( "%7d" , VEC[I] ) ;
        if ( VEC[I] > PROM )
            CONT++ ;
    }
    printf ( "\n\nHay %d valores mayores que %4.2f.", CONT, PROM ) ;
}

```

NO COMPROBACIÓN DE CONTORNOS

Cuando se accede a un elemento de un vector, se realiza el cálculo de su dirección según la aritmética de subíndices ya explicada. El valor de la posición del elemento dentro del vector (I en la fórmula) no es verificado, por lo que puede exceder los límites declarados para el vector.

El lenguaje C otorga máxima libertad al programador en este sentido. Es su responsabilidad no exceder dichos límites, bajo riesgo de leer valores sin sentido o escribir en posiciones de memoria no reservadas con riesgo de sobrescritura.

Los sistemas operativos modernos protegen la memoria de los distintos procesos, de manera que no pueda leerse o escribirse fuera de él. Esto hace que el sistema sea más confiable y que un único proceso no pueda “colgar” toda la máquina. El siguiente ejemplo permite verificar que, luego de cierta frontera, el sistema operativo no permite el acceso y procede a “matar” a nuestro proceso:

```
#include <stdio.h>
int main() {
    int vec[5];
    int pos = 4000; //hasta qué valor de pos se puede leer?
    printf("leyendo posición %d: %d", pos, vec[pos]); //leo por fuera
}
```

En nuestro ensayo, con un valor de `pos` de `100000` se indica la finalización anormal del proceso por una lectura ilegal. Esto es 400 mil bytes más allá del origen del vector. El compilador no informa ningún error sino que el problema surge en tiempo de ejecución.

PASAJE DE VECTORES A FUNCIONES

Cuando se transfiere un argumento a una función, el parámetro formal que recibe el argumento debe ser del mismo tipo. Su declaración en la función es una copia (con excepción del nombre, que *puede* ser otro) de la declaración del argumento.

Con el mismo criterio, la transferencia de un vector a una función debe seguir el formato que se muestra :

```
int VEC[N] ;           /* Declaración del vector */
.....
funcion(VEC);         /* Llamada a la función */
.....
/* Declaración de la función */
funcion (int V[N])
{ .....
```

El siguiente ejemplo nos permitirá comprobar el comportamiento del vector ante la transferencia propuesta.

EJEMPLO: TRANSFERENCIA DE VECTORES A FUNCIONES

En este ejemplo se declara un vector y una variable entera y se les asignan valores. Posteriormente se los transfiere a una función en la que se modifican dichos valores.

Por último, los contenidos del vector y de la variable se verifican en el programa principal, luego de la ejecución de la función.

```

#include <stdio.h>
#define N 6

/* Prototipo de la función */
void funcion (int[N], int);

int main()
{
    int VEC[N] , I , A = 3 ;

    for ( I=0 ; I<N ; I++ )
        VEC[I] = I ;

    funcion ( VEC , A );

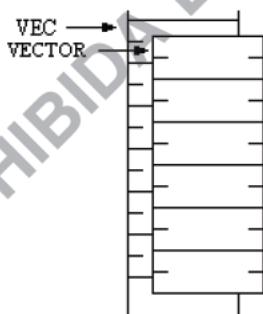
    printf("\nVALOR DE A = %d \n\n\n", A);
    printf("VALORES DEL VECTOR \n\n");
    for( I=0 ; I<N ; I++ )
        printf ( "%7d" , VEC[I] );
}

void funcion (int VECTOR[N] , int A )
{
    int I;
    for ( I=0 ; I<N ; I++ )
        VECTOR[I] = VECTOR[I] * 2;
    A = 8;
}

```

El resultado de la ejecución del programa muestra que el valor de la variable `A` permaneció inalterado (como era de esperar en una **transferencia por valor**), pero el vector original sí vio modificados sus valores dentro de la función.

Esto indica que dentro de la función la variable `A` que se utilizó era una copia de la original, pero no ocurrió lo mismo con el vector.



Dentro de la función se trabajó sobre el vector original, aunque tenía un nombre diferente.

Podemos considerar un modelo en el que dentro de la función se trabaja sobre un “vector superpuesto” como el que muestra la figura de la izquierda.

Dado que ambos vectores están superpuestos, comparten la misma memoria y cualquier modificación que se haga sobre el vector local a la función (`VECTOR`), también se hará sobre el original (`VEC`).

Esta interpretación es solamente un *modelo* que se ampliará más adelante.

VECTORES SIN TAMAÑO

Si consideramos el modelo anterior vemos que el tamaño utilizado al declarar el parámetro formal podría omitirse, declarando un vector sin tamaño.

Dado que ambos vectores están “superpuestos” y no hay comprobación de contornos, el vector VECTOR podrá extenderse en forma segura hasta N valores dado que sus lugares ya fueron reservados al declarar el vector original en el lenguaje.

En el siguiente ejemplo vemos un vector sin tamaño como parámetro formal.

EJEMPLO: TAMAÑO DE LOS VECTORES

En este ejemplo se investiga el tamaño del vector original y del vector “superpuesto” dentro de la función utilizando el operador `sizeof`, que retorna la cantidad de bytes que ocupa en memoria un tipo de dato o una variable.

```
#include <stdio.h>
#define N 6

/* Prototipo */
void funcion (int [] ) ;

int main()
{
    int VEC[N] ;
    printf("\n\nTamaño de VEC = %d", sizeof(VEC) );

    funcion (VEC) ;
}

void funcion (int VECTOR[])
{
    printf("\n\nTamaño de VECTOR = %d", sizeof(VECTOR) );
}
```

El resultado de la ejecución arroja:

```
Tamaño de VEC      =  24
Tamaño de VECTOR   =  4
```

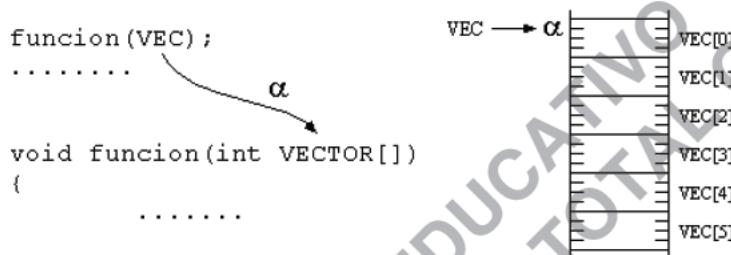
Al mismo resultado se llega si se declara el parámetro formal con tamaño N.

El tamaño de 24 bytes para el vector original VEC coincide con la dimensión de memoria necesaria para albergar 6 variables enteras (4 bytes cada una).

Pero para VECTOR se obtiene un resultado sorprendente: 4 bytes.

En base al modelo presentado en párrafos anteriores, sería de esperar un tamaño igual al de VEC o bien un tamaño cero por haber hecho una declaración con corchetes "vacíos".

Pero analizando un poco más cuidadosamente la transferencia que se está realizando, vemos que el argumento transferido es VEC, y éste en realidad representa la **dirección de inicio** del vector original, como se muestra en la siguiente figura, en donde utilizamos la letra alfa para esta dirección.



El parámetro formal en la función debe ser el adecuado para contener tal argumento, es decir, un **puntero** (esto es, una variable destinada a contener una dirección de memoria).

En una computadora de 32 bits las direcciones de memoria ocupan, justamente, 4 bytes. Si ejecutamos el mismo ejemplo en una máquina con arquitectura de 64 bits el resultado será distinto. Es decir que la cantidad de bytes que ocupa una dirección depende de la **arquitectura física** del hardware (puntualmente, del microprocesador utilizado).

Si se intentara declarar un vector sin tamaño como variable local de una función, el compilador informaría:

```
int VEC[];      ERROR: Array size missing in 'VEC'
```

Con otras palabras, los corchetes vacíos sólo pueden aparecer en la **declaración** de la función, indicando que se recibirá una dirección de memoria donde comienza un vector (del tipo declarado). No pueden aparecer corchetes vacíos en el cuerpo de la función, es decir, en su código.

El vector sin tamaño es solamente un artificio para recibir un vector (por referencia) en una función. Se trata en realidad de una forma de declarar un puntero como parámetro formal, sin conocer aún el tema "punteros".

ACCESO Y BÚSQUEDA EN VECTORES

El **acceso** a un vector consiste en, dada la posición dentro del vector, obtener el contenido. Como ya se vio, es una tarea muy simple que se soporta en el lenguaje directamente, mediante la aritmética de subíndices utilizando el operador [].

Podemos considerar que en un vector se dispone de **acceso directo** (*random*) dado que con el valor del índice se calcula la dirección en memoria, a la cual se accede en forma directa.

La **búsqueda** en un vector es el proceso opuesto. Dado un valor supuestamente contenido en el vector, se debe determinar en qué posición dentro del vector se encuentra, o bien si no se encuentra.

Como veremos más adelante, los procesos de búsqueda y acceso a vectores están frecuentemente ligados.

BÚSQUEDA SECUENCIAL

Es el más simple de los procesos de búsqueda. Consiste en recorrer secuencialmente el vector y comparar cada uno de sus elementos con el valor buscado.

En caso de encontrarse dicho valor se informa la posición en la que se realizó el hallazgo.

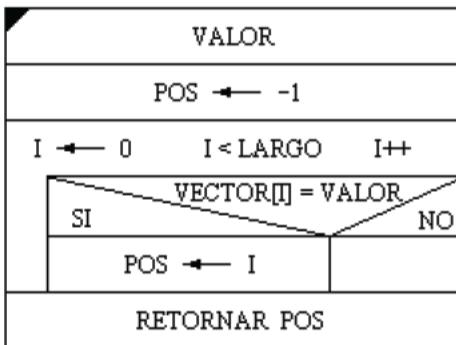
Existen varios criterios de búsqueda secuencial que pueden arrojar diferentes resultados en caso de que el elemento buscado se encuentre más de una vez dentro del vector:

- Recorrer la totalidad del vector guardando siempre la posición del elemento encontrado. En este caso, se informará la posición del *último* elemento encontrado.
- Recorrer el vector solamente hasta encontrar el elemento buscado. En esta ocasión se informa la posición del *primer* elemento encontrado.

EJEMPLO: BÚSQUEDA SECUENCIAL 1

Se construirá una función que reciba un vector de enteros, su tamaño y un valor a buscar en el vector. Nótese que decir que la función “recibe un vector” es una simplificación que nos permite ahorrar espacio: ya hemos visto que en realidad recibe la *dirección* en la que comienza el vector en memoria.

La función retornará la posición del último valor encontrado, si éste se encuentra en el vector, ó -1 si no se encuentra.



```
/* Retorna el último valor encontrado */

int BUSCAR1(int VECTOR[], int L, int VALOR)
{
    int I , POS = -1 ;
    for ( I=0 ; I<L ; I++ )
        if ( VECTOR[I] == VALOR )
            POS = I ;
    return POS ;
}
```

EJEMPLO: BUSQUEDA SECUENCIAL 2

Este es un caso similar al anterior pero se retorna el primer valor encontrado.

Se utiliza el valor -1 inicialmente cargado en POS como *flag* de detección del hallazgo. Mientras POS valga -1 significa que no se encontró aún el valor buscado.

Obsérvese que la condición de mantenimiento en el lazo es compuesta: para seguir iterando se requiere que no se haya llegado al final del vector *y además*, que no se haya encontrado el valor buscado.

```
/* Retorna el primer valor encontrado */
int BUSCAR2 (int VECTOR[] , int L , int VALOR )
{
    int I , POS = -1 ;
    for ( I=0 ; (I<L) && (POS<0) ; I++ )
        if ( VECTOR[I] == VALOR )
            POS = I ;
    return POS ;
}
```

En la siguiente variante (no estructurada) se aprovecha el retorno de la función para realizar una salida inmediata en caso de encontrarse el valor buscado:

```

int BUSCAR3 ( int VECTOR[] , int L , int VALOR )
{
    int I ;
    for ( I=0 ; I<L ; I++ )
        if ( VECTOR[I] == VALOR )
            return I ; //no estructurado, pero más claro
    return -1 ;
}

```

BÚSQUEDA BINARIA

La **búsqueda binaria** (también llamada *búsqueda dicotómica*) consiste en realizar particiones sobre un **vector ordenado** de manera tal que el rango de búsqueda se vaya acotando entre extremos determinados.

A fin de optimizar el procedimiento conviene realizar las particiones por la mitad, aunque no es estrictamente necesario.

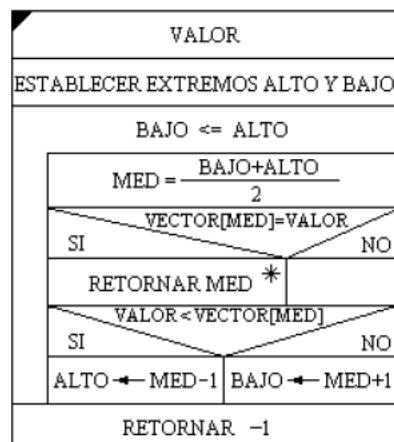
El fundamento de este método de búsqueda es el siguiente :

- Se dispone de las posiciones extremas del rango de búsqueda (ALTO y BAJO).
- Se verifica si el valor buscado se encuentra justo en la mitad de este rango.
- Si no es así, el valor buscado estará por encima o por debajo del punto mitad. Esto permite descartar la parte del vector que no contiene al valor y establecer el nuevo rango por ajuste de uno de los extremos.
- El proceso se repite hasta que se encuentre el valor buscado, o bien, los extremos se entrecrucen, lo que significa que el valor buscado no se encuentra en el vector.

EJEMPLO: BÚSQUEDA BINARIA ITERATIVA

Se construirá una función con la misma funcionalidad de los ejemplos anteriores.

Recuérdese que la búsqueda binaria debe efectuarse sobre un *vector ordenado*.



* SALE DE LA FUNCION

```

/* Busqueda Binaria */
int BUSBIN ( int VECTOR[] , int L , int VALOR)
{
    int BAJO , ALTO , MED ;
    BAJO = 0 ;

```

```

ALTO = L - 1 ;

while ( BAJO <= ALTO ) {
    MED = (BAJO+ALTO)/2 ;

    if ( VECTOR[MED] == VALOR )
        return MED ; //lo encontramos

    if ( VALOR < VECTOR[MED] )
        ALTO = MED - 1 ;
    else
        BAJO = MED + 1 ;
}
return -1 ;
}

```

El algoritmo de búsqueda binaria propone un método similar a cuando buscamos una palabra en un diccionario. El diccionario está ordenado alfabéticamente, lo que nos permite aplicar este método.

Abrimos el diccionario y decidimos si la palabra buscada está *hacia la derecha* o *hacia la izquierda* del lugar en donde estamos. Entonces descartamos una gran cantidad de páginas donde sabemos que no estará la palabra buscada, y continuamos refinando la búsqueda hasta encontrarla, con sucesivas particiones.

BÚSQUEDA DEL MÍNIMO

La **búsqueda del mínimo** consiste en determinar la posición dentro de un vector desordenado, del valor mínimo.

Cuando se encara este tipo de problemas, es conveniente plantearse cómo se los resolvería en el papel, recordando que el microprocesador sólo puede comparar entre dos valores al mismo tiempo.

Será necesario *recorrer* el vector comparando cada uno de sus elementos “recorriendo” cual es el valor mínimo y su posición hasta ese momento. Para ello se necesitarán una o dos **variables auxiliares**.

EJEMPLO: BÚSQUEDA DEL MÍNIMO

Se construirá una función que reciba un vector de enteros y su longitud, y retorne la posición dentro del vector del elemento de valor mínimo.

En caso de haber dos o más valores mínimos iguales se retorna la posición del primero de ellos.

La variable `POSMIN` contendrá la posición del *mínimo transitorio*, considerando inicialmente que el menor valor es el primero (hasta que se demuestre lo contrario encontrando otro aún menor).

```
/* Retorna la posición del valor mínimo */
int MINIMO (int VECTOR[], int L )
{
    int I , POSMIN = 0 ;
    for ( I=0 ; I<L ; I++ )
        if ( VECTOR[I] < VECTOR[POSMIN] )
            POSMIN = I ;
    return POSMIN ;
}
```

EJEMPLO: INTERCAMBIO ENTRE EL MÍNIMO Y EL PRIMERO

Este ejemplo es una extensión del anterior y prepara el terreno para el próximo tema. Se trata de intercambiar el elemento de valor mínimo con el primero, de manera de dejar el menor valor en el primer lugar.

Para realizar el intercambio (*swapping*) será necesaria una variable auxiliar del mismo tipo de las variables a intercambiar.

```
int CAMBIO (int VECTOR[], int L )
{
    int I , AUX , POSMIN = 0 ;
    for ( I=1 ; I<L ; I++ )
        if ( VECTOR[I] < VECTOR[POSMIN] )
            POSMIN = I ;

    /* swapping */
    AUX          = VECTOR[0] ;
    VECTOR[0]     = VECTOR[POSMIN] ;
    VECTOR[POSMIN] = AUX ;
}
```

Una variante del código anterior consiste en parametrizar la posición inicial. Nótese en la siguiente función la presencia de la variable `PRIM` cargada con el valor `0`.

Es evidente que, dado que `PRIM` vale cero durante toda la función, nada cambia con respecto a la anterior. Sin embargo, deja la función lista para el siguiente paso.

```

int CAMBIO (int VECTOR[], int L )
{
    int I , POSMIN ;
    int AUX , PRIM = 0 ;
    POSMIN = PRIM ;

    for ( I=PRIM+1 ; I<L ; I++ )
        if ( VECTOR[I] < VECTOR[POSMIN] )
            POSMIN = I ;

    /* swapping */
    AUX          = VECTOR[PRIM] ;
    VECTOR[PRIM] = VECTOR[POSMIN] ;
    VECTOR[POSMIN] = AUX ;
}

```

ORDENAMIENTO DE VECTORES

Ordenar un vector significa posicionar sus elementos de acuerdo a una distribución que cumpla un patrón determinado, por ejemplo de menor a mayor, alfabéticamente, etc.

El **ordenamiento** es una tarea fundamental en informática debido a que los datos ordenados permiten un manejo mucho más eficiente y además es muy común que los datos se deban presentar al usuario ordenados de alguna manera.

Podemos mencionar diferentes **algoritmos de ordenamiento** con *performance* variada. Algunos de ellos son:

Métodos simples

- Intercambio
- Método del Pivote
- Burbuja
- Burbuja Mejorado
- Agitación
- Selección
- Inserción

Métodos avanzados

- Shell sort
- Heap sort
- Quick sort

ORDENAMIENTO POR SELECCIÓN

Es tal vez el método de ordenamiento más intuitivo, similar al utilizado con un mazo de cartas.

Supongamos que tenemos todos los elementos del vector representados por cartas de un mismo palo, que ubicamos en una hilera desordenada sobre la mesa.

Ahora *seleccionamos* la menor de ellas y la sacamos de este grupo colocándola como primera carta en una nueva hilera.

Repetimos esta operación con la *segunda menor* carta del grupo original y la ubicamos a continuación de la que tenemos en la nueva hilera.

Si repetimos los pasos anteriores, el grupo original irá disminuyendo mientras que la nueva hilera crecerá ordenada hasta contener todas las cartas.

Aplicar este procedimiento a un vector equivale a crear un vector auxiliar que represente a la "nueva hilera".

9	2
6	6
2	9
4	4
8	8

2	2
6	6
9	9
4	4
8	8

A fin de no duplicar la memoria utilizada se puede construir el nuevo vector sobre el original, simplemente *intercambiando* las posiciones de los elementos seleccionados, con los de su futura ubicación.

La figura de la izquierda muestra dicho intercambio para el primer elemento (¿resulta conocido?).

De esta forma se tiene en el primer lugar al menor de todos los elementos. En la figura de la izquierda se muestra grisada la casilla del elemento *ya ubicado*.

Al repetir el procedimiento al remanente del vector, se detecta que el menor valor es 4. Al intercambiarlo queda ubicado en su orden definitivo.

Se ve que las componentes grisadas del vector van quedando en orden.

Recuérdese que la nueva búsqueda del valor mínimo debe realizarse sobre el vector remanente y no sobre el total.

A continuación se ilustra la finalización del procedimiento. Obsérvese que una vez ubicados en orden $N-1$ elementos, forzosamente el elemento restante también está ordenado.

2	2	2	2
4	4	4	4
9	6	6	6
6	9	8	8
8	8	9	9

2	2	2	2
4	4	4	4
6	6	6	6
9	8	8	8
8	9	9	9

2
4
6
8
9

VECTOR ORDENADO

EJEMPLO: ORDENAMIENTO POR SELECCIÓN

```
void SELECCION (int VECTOR[] , int L )
{
    int I , POSMIN ;
    int AUX , PRIM ;
    for ( PRIM = 0 ; PRIM < L-1 ; PRIM++ ) { // L-1 veces
        POSMIN = PRIM ;

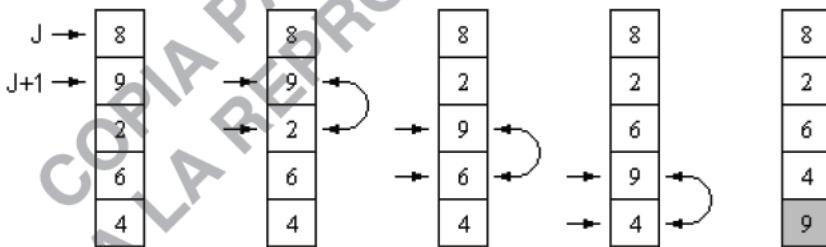
        for ( I=PRIM+1 ; I<L ; I++ ) //búsqueda del mínimo
            if ( VECTOR[I] < VECTOR[POSMIN] )
                POSMIN = I ;

        /* swapping */
        AUX          = VECTOR[PRIM] ;
        VECTOR[PRIM] = VECTOR[POSMIN] ;
        VECTOR[POSMIN] = AUX ;
    }
}
```

MÉTODO DEL BURBUJEOS

El **método de la burbuja** o simplemente “burbujeo” es otro representante de la familia de métodos de intercambio.

Consiste en recorrer el vector *comparando pares de valores consecutivos*. Es decir, se compara el elemento de orden J contra el de orden J+1. Si se encuentran fuera del orden que se desea establecer, se los intercambia.

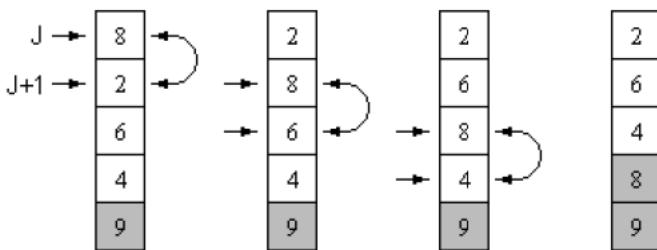


En la figura precedente se muestra la secuencia de un “barrido” en el vector mostrando los casos en que se intercambiaron los lugares.

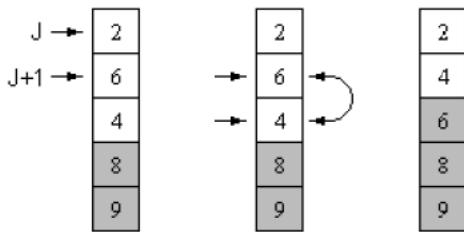
Nótese que el mayor elemento (9) rápidamente ganó su lugar. “Cayó” al fondo como una moneda en una copa de champagne mientras que los elementos menores “suben” a sus posiciones lentamente, como las *burbujas* en esa misma copa. De aquí el nombre del método.

Luego del primer barrido, el mayor elemento quedó ubicado (9). Se puede repetir el proceso anterior “acortando”el vector, omitiendo el último elemento.

El segundo barrido se muestra a continuación.



En este caso fue necesaria una comparación menos debido al elemento ya ubicado. Si no se “acorta” el vector el ordenamiento se produce igual, pero de manera más ineficiente debido a comparaciones inútiles.

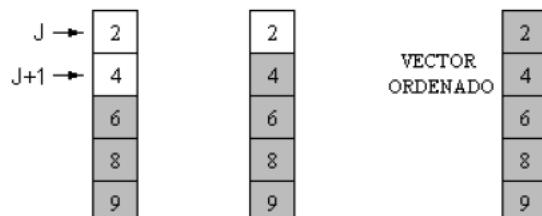


El tercer barrido muestra cómo se ubica el tercer elemento (6).

Cuando se hayan producido $N-1$ barridos estarán ubicados en su lugar $N-1$ elementos, pero entonces el elemento restante también ocupará su lugar.

Se puede apreciar que será necesaria la implementación de un lazo que permita realizar $N-1$ barridos, siendo N la cantidad de elementos a ordenar.

También se deberá implementar un lazo que realice cada barrido a fin de realizar la comparación entre el elemento de orden J y el de orden $J+1$, con el eventual intercambio.



EJEMPLO: ORDENAMIENTO POR BURBUJEOS

A continuación se muestra la función que implementa el método descrito anteriormente.

Obsérvese que el **lazo externo**, controlado por la variable I ejecuta $L-1$ iteraciones, es decir, las necesarias para ordenar en su totalidad el vector. El **lazo interno**, controlado por la variable J , se utiliza para recorrer el vector y comparar

los pares. El bucle interno va decrementando la cantidad de iteraciones cada vez, debido a la presencia de la variable *I* en el valor límite (*J < L-I-1*). Esto es así porque, como ya analizamos, el vector se va ordenando “desde abajo” con este método.

```
/* Ordena el vector por el método del burbujeo */
void BURBUJE0 (int VECTOR[], int L )
{
    int I , J ;
    int AUX ;
    for ( I=0 ; I < L-1 ; I++ )
        for ( J=0 ; J <L-I-1 ; J++ )
            if ( VECTOR[J] > VECTOR[J+1] ) { //están al revés?

                /* swapping */
                AUX      = VECTOR[J] ;
                VECTOR[J] = VECTOR[J+1] ;
                VECTOR[J+1] = AUX ;
            }
}
```

VECTORES APAREADOS

Frecuentemente los problemas informáticos presentan la necesidad de manejar conjuntos de datos de diferente naturaleza, pero con una determinada relación entre ellos.

Los vectores tienen la limitación de contener datos del mismo tipo, y por lo tanto, para manejar datos de naturaleza diferente, se deberá contar con un grupo de vectores, al menos tantos de ellos como tipos diferentes se manejen.

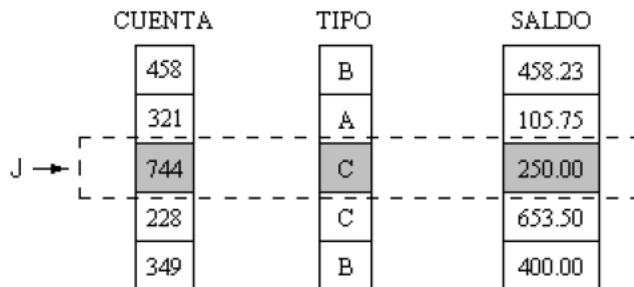
Supongamos que se desea manejar las cuentas de un banco, y los datos básicos fueran:

- Número de CUENTA Tipo: entero
- TIPO de Cuenta Tipo: carácter
- SALDO de la Cuenta Tipo: flotante

Para contener los datos será necesario declarar 3 vectores del tamaño adecuado para contener todas las cuentas del banco. En este caso, como el banco es más bien modesto, supondremos sólo 5 cuentas.

```
int CUENTA[5];
char TIPO [5];
float SALDO[5];
```

La información contenida en cada uno de los vectores estará relacionada de modo que a la cuenta de la posición J del vector CUENTA[] le corresponda el tipo de cuenta del vector TIPO[] y el saldo de cuenta del vector SALDO[].



Esta relación debe cumplirse para todo J en los vectores y marca la correspondencia o **apareamiento (match)** de los mismos.

Cuando se realice el manejo de los vectores debe tomarse la precaución de nunca alterar dicho apareamiento.

ACCESO Y BÚSQUEDA EN VECTORES APAREADOS

Los procesos de búsqueda y acceso están íntimamente ligados en los sistemas de vectores apareados. Cada uno de los vectores que forman el sistema recibe el nombre de **campo**.

Es muy frecuente conocer el dato correspondiente a uno de los vectores y desear obtener los datos de los otros que se corresponden con el primero.

Esto implica un proceso de **búsqueda** del dato conocido en un vector, a fin de obtener su **posición**, y con ella producir el **acceso** en los otros vectores para obtener sus datos.

EJEMPLO: VECTORES APAREADOS

Siguiendo con el ejemplo de las cuentas bancarias presentado en esta sección, se mostrará cómo construir, mostrar y acceder a los vectores apareados.

Se declara un sistema de 10 cuentas y se cargan los vectores correspondientes en el momento de la declaración. Esto representa una simplificación muy grande del software real, a los efectos de concentrarnos en el apareamiento.

Se presenta en pantalla el sistema de 3 vectores cargados y se pide el ingreso de un número de cuenta, que se buscará dentro del vector CUENTA[].

La búsqueda propiamente dicha la realiza la función BUSCAR() que es similar a la presentada en el ejemplo de búsqueda secuencial.

La función ACCESO() muestra en pantalla el contenido de los 3 campos habiendo obtenido su posición de la función BUSCAR().

Si ésta retornó -1 se informa que el número de cuenta buscado no está contenido en el vector.

```
#include <stdio.h>
#define N 10

void MOSTRAR (int[], char[], float[], int);
void ACCESO ( int[], char[], float[], int, int);
int BUSCAR (int[], int, int);

int main()
{
    int CUENTA[N] = {458,321,744,228,349,201,198,705,789,227};
    char TIPO[N] = {'B','A','C','C','B','A','C','A','B','C'};
    float SALDO[N] = {458.32,105.00,750.00,250.00,633.50,
                      129.00,732.40,900.00,498.99,867.66 };
    int C ;

    MOSTRAR (CUENTA, TIPO, SALDO, N);
    printf("\n\n\nNumero de Cuenta a Consultar = ");
    scanf("%d", &C );
    ACCESO (CUENTA,TIPO,SALDO,N,C);
}

void MOSTRAR (int CUENTA[], char TIPO[], float SALDO[], int L )
{
    int I ;
    printf("\n\t\t CUENTA \t\t TIPO \t\t SALDO\n\n");
    for ( I=0 ; I<L ; I++ )
        printf("\n%25d%14c%20.2f", CUENTA[I],TIPO[I],SALDO[I]);
}

/* Muestra los campos resultantes de la búsqueda */
void ACCESO (int CUENTA[], char TIPO[], float SALDO[], int L, int C)
{
    int POS ;
    POS = BUSCAR (CUENTA,L,C);
    if ( POS < 0 )
        printf ("\n\n\nNUMERO DE CUENTA INEXISTENTE\n\n");
    else {
        printf("\n\n\n");
        printf("%25d%14c%20.2f", CUENTA[POS],TIPO[POS],SALDO[POS]);
    }
}
```

```

/* Retorna la posicion de la cuenta buscada o -1 */
int BUSCAR (int CUENTA[] , int L , int NUM )
{
    int I ;
    for ( I=0 ; I<L ; I++ )
        if ( CUENTA[I] == NUM )
            return I ;
    return -1 ;
}

```

ORDENAMIENTO CON ARRASTRE EN VECTORES APAREADOS

Durante el proceso de ordenamiento en un sistema de vectores apareados, es necesario tener cuidado de no *romper el apareamiento* de los vectores involucrados.

Cuando procedemos a ordenar un sistema de vectores apareados se selecciona uno de los campos como determinante del ordenamiento, al que se llama *campo clave de ordenamiento*.

El vector del campo clave debe quedar ordenado según los requerimientos impuestos, pero el resto de los vectores debe respetar el apareamiento, por lo que ante cualquier movimiento (*swapping*) en el campo clave, debe producirse el mismo movimiento (*arrastre*) en el resto de los vectores.

EJEMPLO: ORDENAMIENTO EN VECTORES APAREADOS

En base al problema del ejemplo anterior, se ordenará el sistema en orden creciente de números de cuenta. Esto es, los números más chicos primero y los más grandes después (*ascendente*).

Solamente se presentará la función de ordenamiento (que utiliza el método de la burbuja), debido a que la estructura del programa es la misma que la del ejemplo anterior. Esta función puede ser insertada directamente (con su correspondiente prototipo) en el mencionado programa.

Obsérvese que es necesario declarar una variable auxiliar para el intercambio *por cada tipo diferente* de los vectores involucrados.

En el ordenamiento se muestra el intercambio (*swapping*) del campo clave y también el intercambio por arrastre del resto de los vectores.

Consideremos el esfuerzo que representaría el proceso de arrastre en ordenamientos de sistemas de mayor cantidad de campos, por ejemplo, si en el caso anterior se agregaran los datos de los últimos 10 movimientos de cada cuenta.

Afortunadamente, esta complejidad desaparece al utilizar **estructuras** (**struct**) en las que el arrastre se produce automáticamente al formar los campos un bloque único. Estructuras es un tema que abarcaremos posteriormente.

```

/* Ordena los vectores en base al campo clave CUENTA */
/* Los vectores TIPO y SALDO son arrastrados           */

void ORDENAR (int CUENTA[], char TIPO[], float SALDO[], int L )
{
    int I,J ;
    int AUXINT   ;
    char AUXCHAR ;
    float AUXFLOAT ;

    for ( I=0 ; I<L-1 ; I++ )
        for ( J=0 ; J<L-I-1 ; J++ )
            if ( CUENTA[J] > CUENTA[J+1] ) {

                /* Swapping */
                AUXINT      = CUENTA[J] ;
                CUENTA[J]   = CUENTA[J+1] ;
                CUENTA[J+1] = AUXINT ;

                /* Arrastre de TIPO */
                AUXCHAR     = TIPO[J] ;
                TIPO[J]     = TIPO[J+1] ;
                TIPO[J+1]   = AUXCHAR ;

                /* Arrastre de SALDO */
                AUXFLOAT   = SALDO[J] ;
                SALDO[J]   = SALDO[J+1] ;
                SALDO[J+1] = AUXFLOAT ;
            }
}

```

EJEMPLO: ORDENAMIENTO MULTIPLE EN VECTORES APAREADOS

Puede ocurrir que no exista un campo clave de ordenamiento único, sino que haya uno principal y otro (o más) secundario.

En este ejemplo se plantea el caso de ordenar el sistema de los ejemplos anteriores según el campo **TIPO** de cuenta (orden creciente), pero ante igualdad en este campo se debe ordenar por número de **CUENTA**. Esto es similar a lo que ocurre en el padrón electoral: las personas están ordenadas por apellido, y aquellos con apellido igual lo están además por nombre.

La resolución es sencilla si se considera una **condición compuesta** en la toma de decisión del intercambio, es decir, cuando decidimos si los elementos del par que estamos comparando están o no desordenados (para proceder a intercambiarlos). Dicha condición se puede plantear del siguiente modo : “*Se debe realizar el intercambio si entre el elemento de orden J y el de orden J+1 se da que: los códigos están fuera de lugar, o bien si los códigos son iguales y además los números de cuenta están fuera de lugar*”

```

/* Ordena los vectores en base al campo clave TIPO */
/* Cuando hay igualdad en TIPO se ordena según CUENTA */

void ORDENAR (int CUENTA[],char TIPO[],float SALDO[],int L )
{
    int I,J, AUXINT ;
    char AUXCHAR ;
    float AUXFLOAT ;

    for ( I=0 ; I < L-1 ; I++ )
        for ( J=0 ; J < L-I-1 ; J++ )
            if ( (TIPO[J] > TIPO[J+1]) || \
                ( (TIPO[J] == TIPO[J+1]) && \
                  (CUENTA[J] > CUENTA[J+1]) )
            )
        {
            /* Swapping */
            AUXINT      = CUENTA[J] ;
            CUENTA[J]   = CUENTA[J+1] ;
            CUENTA[J+1] = AUXINT ;

            AUXCHAR     = TIPO[J] ;
            TIPO[J]     = TIPO[J+1] ;
            TIPO[J+1]   = AUXCHAR ;

            AUXFLOAT   = SALDO[J] ;
            SALDO[J]   = SALDO[J+1] ;
            SALDO[J+1] = AUXFLOAT ;
        }
}

```

Es común pensar que esto también puede resolverse con dos burbujas, una a continuación de la otra. No sólo tiene peor rendimiento, ya que se ejecutarán todos los bucles dos veces, sino que podemos no arribar al resultado correcto, dependiendo de cuál es el criterio de ordenamiento que nos pidan, puesto que la segunda burbuja desordena lo ordenado por la primera.

MATRICES

Las **matrices** son vectores multidimensionales. La **dimensión** queda determinada por la cantidad de subíndices que se declaran.

Si bien no hay límite (más que el impuesto por la cantidad de memoria disponible en la máquina) para la dimensión de la matriz, las más comunes a utilizar son 2 y 3.

La forma de declaración es como sigue:

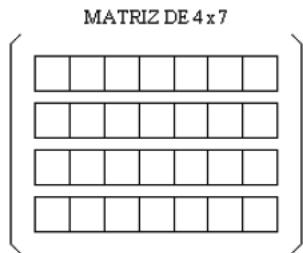
```
tipo NOM_MATRIZ [tamaño][tamaño]...[tamaño];
```

TAMAÑO DE LOS VECTORES MULTIDIMENSIONALES

Se debe tener cuidado con el tamaño de las matrices dado que crecen enormemente. Por ejemplo, una matriz declarada como `int MAT[16][16][16][16]` ocupa en memoria 262144 bytes ($4 \times 16 \times 16 \times 16$), según podemos obtener con `sizeof(MAT)`.

MATRIZ BIDIMENSIONAL

Es el tipo más frecuente de matrices. Podemos imaginar que el primer subíndice representa las *filas* y el segundo las *columnas*.

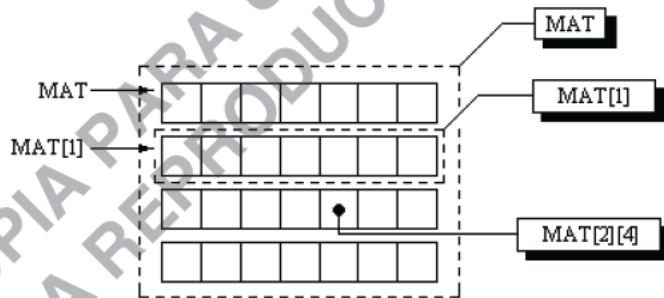


Puede resultar también conveniente imaginar esta matriz como un *vector de vectores*.

Si declaramos una matriz como:
`int MAT[4][7] ;`

Puede ser visualizada como un conjunto de 4 elementos. Cada uno de ellos constituido por un vector de 7 enteros.

Es necesario tener en claro cómo se referencia cada elemento de la matriz. La siguiente figura ilustra el significado e interpretación de algunas expresiones:



Podemos interpretar que `MAT` es la matriz completa pero además sabemos que es la dirección donde comienza esta estructura de datos. Esta dirección coincide con la del primer elemento de la matriz.

El elemento `MAT[1]` de la matriz es el segundo vector de la figura (por supuesto, las filas o vectores internos comienzan a numerarse a partir de cero), pero por ser un vector también representa la dirección de inicio, que coincide con la dirección de su primer elemento.

El elemento `MAT[2][4]` es un `int`. Es el quinto elemento del tercer vector de la matriz (recordemos que el inicio de los subíndices es siempre cero).

Podemos ver en la figura anterior que el elemento `MAT[2][4]` está ubicado en la fila 2, columna 4, de la matriz.

PASAJE DE MATRICES A FUNCIONES

Cuando se transfiere un vector a una función se está transfiriendo en realidad su dirección inicial. Hemos dicho que esto se denomina **pasaje por referencia**. El parámetro formal en la función representa un vector sin tamaño, de elementos del tipo coincidente con los del vector transferido. Este tipo debe quedar absolutamente establecido debido a la aritmética de subíndices.

Si consideramos a una matriz de orden N como un vector de elementos que a su vez son matrices de orden $N-1$, debe quedar establecido en la recepción de la función el tipo exacto de estas matrices de orden $N-1$.

Ejemplo:

```
tipo MAT[A][B][C]...[Y][Z] ; //declaración de la matriz
.....
FUNCION (MAT) ; //llamada a función con pasaje de la matriz
.....
void FUNCION (tipo M[] [B][C]...[Y][Z]) //declaración de la función
{
    .....
}
```

En este ejemplo se declaró una matriz multidimensional `A...Z`. Posteriormente se transfirió a la función llamada `FUNCION()` la dirección `MAT`.

Esta dirección se tomó en el parámetro formal `M`, pero fue necesario establecer los tamaños de los vectores de orden `B...Z` (nótese que sólo los corchetes correspondientes a `A` están vacíos).

Al recibir una matriz en una función todos los tamaños, menos uno, deben estar definidos.

EJEMPLO: PASAJE DE MATRICES A FUNCIONES

En este ejemplo se muestra el pasaje de una matriz de 7 filas por 10 columnas a dos funciones. Recordemos que no se transferirá una *copia* de la matriz sino su dirección de inicio, algo por demás beneficioso para ahorrar memoria.

En la primera función se cargan las variables de la matriz con enteros aleatorios menores que 100, y en la segunda se la muestra en pantalla.

Obsérvese el doble lazo para barrido de filas y columnas y el modo en que se obtienen números pseudoaleatorios utilizando funciones de la biblioteca estándar.

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

#define FILAS      7
#define COLUMNAS 10

void CARGAR ( int [][COLUMNAS] , int ) ;
void MOSTRAR ( int [][COLUMNAS] , int ) ;

int main()
{
    int MATRIZ[FILAS][COLUMNAS] ;

    CARGAR ( MATRIZ , FILAS ) ;
    MOSTRAR ( MATRIZ , FILAS ) ;
}

void CARGAR ( int MAT[][COLUMNAS] , int N )
{
    int I, J ;

    srand(time(0)); //inicializamos semilla

    for ( I=0 ; I<N ; I++ )
        for ( J=0 ; J < COLUMNAS ; J++ )
            MAT[I][J] = rand() % 100; //entre 0 y 99
}

void MOSTRAR ( int MAT[][COLUMNAS] , int N )
{
    int I , J ;
    printf ( "\n\n\n" ) ;
    for ( I=0 ; I<N ; I++ ) {
        printf ( "\n\n\t" ) ;
        for( J=0 ; J < COLUMNAS ; J++ )
            printf ( "%6d" , MAT[I][J] ) ;
    }
}
```

Se verifica que múltiples ejecuciones generan distintos valores en la matriz, debido a la generación pseudoaleatoria que se obtiene con el par de funciones estándar `srand` y `rand`.

EJEMPLO: SISTEMA DE 3 ECUACIONES CON 3 INCÓGNITAS

Se resolverá en este ejemplo, a modo de aplicación de matrices, un sistema de 3 ecuaciones lineales con 3 incógnitas de la forma:

$$\begin{array}{l} a_{11} X + a_{12} Y + a_{13} Z = I_1 \\ a_{21} X + a_{22} Y + a_{23} Z = I_2 \\ a_{31} X + a_{32} Y + a_{33} Z = I_3 \end{array}$$

Donde, a los efectos del programa, cada a_{ij} es un entero con signo.

La solución del sistema de ecuaciones se obtiene mediante el cálculo de las expresiones que siguen:

$$X = \frac{\begin{vmatrix} I_1 & a_{12} & a_{13} \\ I_2 & a_{22} & a_{23} \\ I_3 & a_{32} & a_{33} \end{vmatrix}}{\Delta}, \quad Y = \frac{\begin{vmatrix} a_{11} & I_1 & a_{13} \\ a_{21} & I_2 & a_{23} \\ a_{31} & I_3 & a_{33} \end{vmatrix}}{\Delta}, \quad Z = \frac{\begin{vmatrix} a_{11} & a_{12} & I_1 \\ a_{21} & a_{22} & I_2 \\ a_{31} & a_{32} & I_3 \end{vmatrix}}{\Delta}$$

Donde el valor *delta* queda determinado de la siguiente forma:

$$\Delta = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

Los determinantes se resuelven en el programa mediante la aplicación de la regla de Laplace. La función DET() recibe un determinante de 3×3 y retorna su valor.

La matriz auxiliar DETER se utiliza para construir la matriz adecuada a cada caso y transferirla a la función DET().

La matriz MATRIZ de 3×4 es la que contiene los valores del sistema de ecuaciones, mientras que el vector de flotantes D[] albergará los resultados.

```
#include <stdio.h>

int DET(int [[3]]);
void CARGAR(int [[4]]);
void MOSTRAR(int [[4]]);
```

```

int main()
{
    int MATRIZ[3][4] , DETER[3][3] ;
    float D[3] ;
    int DELTA , I , J , Z ;

    CARGAR(MATRIZ);
    MOSTRAR (MATRIZ);

    /* Calcula el determinante DELTA */
    for(I=0;I<3;I++)
        for(J=0;J<3;J++)
            DETER[I][J]=MATRIZ[I][J] ;
    DELTA = DET(DETER) ;

    /* Obtiene las 3 soluciones */
    for (Z=0;Z<3;Z++) {
        for(I=0;I<3;I++)
            for(J=0;J<3;J++)
                DETER[I][J] = MATRIZ[I][J] ;
        for(I=0;I<3;I++)
            DETER[I][Z] = MATRIZ[I][3] ;

        D[Z] = (float) DET(DETER) / DELTA ;
    }

    printf("\n\n\n\n\tSOLUCION\n\n");
    printf("\tX = %8.2f\n\tY = %8.2f\n\tZ = %8.2f " ,
           D[0],D[1],D[2]) ;
}

/* Lee el sistema de ecuaciones */
void CARGAR (int MAT[][4] )
{
    int I,J ;

    for ( I=0 ; I<3 ; I++ ) {
        for ( J=0 ; J<3 ; J++ ) {
            printf("%d %d ", I+1, J+1);
            scanf("%d",&MAT[I][J]) ;
        }
        printf("Termino independiente %d ", I+1);
        scanf("%d", &MAT[I][3]) ;
    }
}

```

```

/* Presenta en pantalla el sistema de ecuaciones */
void MOSTRAR (int MAT[][][4] )
{
    int I , J ;
    printf("\n");
    for ( I=0 ; I<3 ; I++ ) {
        printf("\n\n\t");
        printf("%6d X +%6d Y +%6d Z=%6d", \
               MAT[I][0], MAT[I][1], MAT[I][2], MAT[I][3] ) ;
    }
}

```

```

/* Resuelve un determinante de 3 x 3 */
int DET ( int M[][3] )
{
    int RES,R0,R1,R2 ;
    R0 = M[0][0] * (M[1][1] * M[2][2] - M[1][2] * M[2][1]) ;
    R1 = M[1][0] * (M[0][1] * M[2][2] - M[0][2] * M[2][1]) ;
    R2 = M[2][0] * (M[0][1] * M[1][2] - M[0][2] * M[1][1]) ;
    RES = R0 - R1 + R2 ;
    return RES ;
}

```

La impresión en pantalla para un sistema de ecuaciones de prueba resultó como se muestra:

$$\begin{array}{rclcl}
 3 & X & + & 5 & Y & + & 7 & Z & = & 2 \\
 4 & X & + & 7 & Y & + & 1 & Z & = & 8 \\
 3 & X & + & 5 & Y & + & 3 & Z & = & 3
 \end{array}$$

SOLUCION

$$\begin{array}{lcl}
 X & = & -15.00 \\
 Y & = & 9.75 \\
 Z & = & -0.25
 \end{array}$$

PROBLEMAS PROPUESTOS

1. Ingrese 20 valores enteros y muéstrelos en pantalla en una línea, en el orden de ingreso, y en la línea siguiente, en orden inverso al de ingreso.
2. Ingrese 20 valores enteros y posteriormente un valor entero adicional al que llamaremos DIV. Indique cuantos componentes del vector son divisibles por DIV.
3. Realizar una función que reciba un vector de enteros y su longitud, y retorne el promedio de sus valores.
4. Crear una función que reciba un vector de enteros y su longitud, y cargue sus variables con valores aleatorios comprendidos entre 0 y 99 (utilizar la función rand).
5. Realizar una función que simule arrojar un dado (retorna valores enteros aleatorios comprendidos entre 1 y 6).
Utilizando dicha función realice el experimento de arrojar 1000 veces el dado, almacenando las cantidades de aparición de cada cara en un vector.
Imprima la cantidad de veces que apareció cada cara y su porcentaje sobre el total.
6. Realizar una función que reciba un vector de enteros y su longitud y retorne 1 si el vector esté ordenado en forma creciente, y 0 en caso contrario.
7. Implementar una función que reciba dos vectores de enteros del mismo tamaño y su longitud, y retorne 1 si ambos vectores son iguales (idénticos contenidos en idénticas posiciones) y 0 en caso contrario.
8. Tomando como base el ejemplo de vectores apareados con cuentas bancarias, muestre en pantalla todos los datos de cuentas de tipo "A".
9. Tomando como base el mismo ejemplo de vectores apareados con cuentas bancarias, muestre en pantalla los datos de la cuenta de mayor saldo.

10. Tomando como base el mismo ejemplo de vectores apareados con cuentas bancarias, muestre en pantalla todos los datos de las cuentas cuyo saldo supera los \$500, ordenados en forma decreciente de saldo (los más grandes primero).

11. En un torneo de búsqueda submarina compiten 15 buceadores (numerados del 1 al 15).

Se pide mostrar la tabla de posiciones actualizada (y ordenada) con cada captura hecha por un competidor. El dato que se proporciona cada vez es simplemente el número de buceador que logró una captura. El ingreso de datos (y la competencia) finaliza con el ingreso de un número de competidor negativo.

12. Se ingresarán (en un orden cualquiera) los datos de 16 equipos de fútbol, compuestos por código del equipo (`int`) y cantidad de puntos (`int`). Mostrar la tabla de posiciones (ordenada) y a continuación el *fixture* de la primera fecha de *play offs*, es decir, los partidos entre el primero y el último, el segundo y el anteúltimo, etc.

13. Generar y mostrar una matriz de 8×8 enteros cuyos elementos valgan 0, excepto 2 de ellos que valdrán 1 y estarán ubicados en posiciones aleatorias de la matriz.

Mostrar dicha matriz en pantalla.

14. Crear una función que reciba la matriz del problema anterior y retorne 1 si los elementos distintos de 0 comparten la misma fila, columna o diagonal. En caso contrario, retornar 0.

6. STRINGS

Una string o “cadena” es un vector de caracteres (char) cuya secuencia finaliza con el carácter nulo, esto es, un byte con todos sus bits en cero.

Las **strings** son la forma que se dispone en C para procesar y manejar nombres, frases y texto en general, es decir, información alfabética.

Si bien los strings son **vectores de char**, puede ser conveniente considerarlos como un único bloque de información que es manejado a través de funciones específicas.

El acceso al string se produce a través de su dirección de inicio, es decir, la dirección de su primer elemento (el primer char de la string) que está representada por el nombre del vector o string.

El carácter nulo ('\0'), utilizado para detectar la finalización del string, es aquel cuyo código ASCII es 00H, es decir que es un byte cuyos bits están todos en estado bajo o cero. Por lo tanto, su evaluación lógica es *falso*. Solemos llamar **cero terminal** a este carácter que marca el fin de la string.

Las funciones de la biblioteca que manejan strings esperan encontrar, por convención, el carácter nulo para marcar su finalización.

En caso de no encontrarlo como corresponde, el resultado de la función será incorrecto o incierto.

ASIGNACION DIRECTA DE STRINGS

Dado que las strings son en sentido estricto vectores, será necesario reservar el espacio de memoria adecuado para contenerlas.

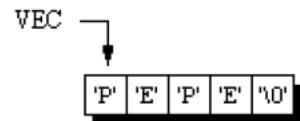
Consideremos una string que contenga la palabra “PEPE”.

Sabemos que en memoria esta string ocupará 5 bytes y por lo tanto podemos hacer la reserva correspondiente:

```
char VEC[5];
```

Ahora podemos asignar los contenidos:

```
VEC[0] = 'P';
VEC[1] = 'E';
VEC[2] = 'P';
VEC[3] = 'E';
VEC[4] = '\0'; //cero terminal
```

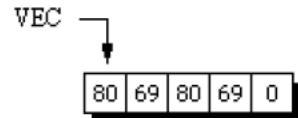


La figura de la derecha muestra una imagen de la disposición en memoria de los caracteres asignados. Esta imagen es solamente una representación de la disposición de los caracteres en memoria a partir del inicio del vector y no es fidedigna en cuanto a los datos que realmente se almacenan en memoria.

Sabemos que los caracteres y símbolos colocados entre comillas simples (') se transforman al compilar en el código ASCII que le corresponde a dicho carácter.

De esta manera se podría haber asignado al vector como sigue, sabiendo que los códigos ASCII de las letras P y E son 80 y 69 (decimal), respectivamente:

```
VEC[0] = 80;
VEC[1] = 69;
VEC[2] = 80;
VEC[3] = 69;
VEC[4] = 0 ;
```



La imagen de la derecha muestra ahora una disposición más real de lo que ocurre en memoria.

Dado que los vectores pueden ser asignados en el momento de su declaración, se podrían haber logrado los mismos resultados de la siguiente forma :

```
char VEC[] = {80, 69, 80, 69, 0};
```

o lo que es idéntico:

```
char VEC[] = {'P', 'E', 'P', 'E', '\0'};
```

FORMATO TIPO STRING

Toda vez que en un programa C se coloca texto entre comillas dobles (" ") se trata como una string o cadena.

Esto ya se comprobó en la utilización de las funciones `printf()` y `scanf()` en las que existe una **cadena de formato** encerrada entre comillas.

Lo que produce la aparición de las comillas dobles en el momento de compilación es que la secuencia de caracteres (más un cero terminal) se almacene en el segmento de datos del programa, y en el lugar donde se colocó la string el compilador coloca la dirección de memoria en donde comienza la string (esto es, un puntero).

Podemos utilizar el formato de string, es decir las comillas dobles, para realizar diversas operaciones. En este caso particular realizaremos una asignación directa de una string a un vector durante su declaración:

```
char VEC[] = "PEPE";
```

El efecto de esta operación es idéntico al descripto anteriormente. Esto significa que el tamaño del vector es de 5 bytes dado que se incluirá al carácter nulo.

La asignación directa de strings a vectores es una operación que solamente se puede realizar en el momento de su declaración.

Para realizarla en otro momento del programa es necesario utilizar una función específica que presentaremos en este mismo capítulo, llamada `strcpy`.

En el siguiente ejemplo se realizará la asignación directa de una string a un vector y posteriormente se verificará su contenido.

Debe recordarse que el tamaño del vector declarado y asignado de esta forma, se ajusta exactamente para contener a la string asignada más el cero terminal.

EJEMPLO: COMPROBACIÓN DE ASIGNACIÓN DE STRINGS

En este ejemplo se asigna la palabra “PEPE” al vector y posteriormente se lo recorre verificando la existencia del carácter nulo.

```
#include <stdio.h>
int main()
{
    char I , VEC[] = "PEPE";
    for (I=0; I<5; I++)
        printf("\n%d\t%c", VEC[I], VEC[I]);
}
```

La ejecución del programa arrojó como resultado:

```
80      P
69      E
80      P
69      E
0
```

Obsérvese que al carácter nulo (0) no le corresponde ningún símbolo.

No debe confundirse al carácter nulo con el espacio generado por la barra espaciadora, cuyo código ASCII es 20.

FUNCIONES PARA LECTURA DE STRINGS

Son funciones destinadas a tomar una cadena de caracteres ingresada desde teclado y almacenarla en algún lugar de la memoria, cuya ubicación se entrega a la función como argumento.

Se disponen de dos funciones para este propósito: `scanf()` y `gets()`

Ambas están asociadas a la cabecera `stdio.h`

FUNCIÓN SCANF

La función `scanf()` aplicada a strings requiere la utilización del carácter de formato `%s` y la entrega, como argumento, de la dirección inicial a partir de la cual se almacenará la string en memoria.

El formato y uso más habitual es como sigue:

```
char palabra[20] ;
scanf("%s", palabra);
```

Nótese que se reservaron 20 bytes para contener el string, y que `palabra` es el nombre del vector (y por lo tanto, por definición, la dirección de memoria donde éste comienza). Por esta razón no se debe utilizar el operador `&` como lo hemos hecho repetidamente hasta aquí.

La función `scanf`, en este caso, tomará uno a uno los caracteres que ingresan de teclado y los almacena a partir de la posición de memoria indicada. Esta operación la repite hasta el ingreso de un **carácter blanco**.

En ese momento reemplaza al carácter blanco por el carácter nulo, el cual es almacenado en memoria y finaliza la operación.

El lenguaje C considera “caracteres blancos” al espacio de tabulación (tecla TAB), al espacio de la barra espaciadora y al cambio de línea (Enter).

Esta situación puede dar lugar a resultados imprevistos, debido a la presencia de espacios en frases o nombres que se quieran ingresar en una única string.

Consideremos la siguiente situación: *Se desean almacenar en 3 vectores los nombres “Pepe Luis”, “Lola” y “Paco”*. Se utilizará para ello el siguiente código:

```
char V1[20], V2[20], V3[20];
scanf ("%s", V1);
scanf ("%s", V2);
scanf ("%s", V3);
```

Al ejecutarlo se cortará el ingreso sin permitir la escritura de “Paco”, habiendo quedado almacenados en cada uno de los vectores las palabras “Pepe”, “Luis” y “Lola”.

Esto se debe a la presencia del espacio de separación entre “Pepe” y “Luis”.

FUNCTION GETS

La función `gets()` tiene un comportamiento similar a la anterior mientras que presenta algunas diferencias.

En primer término, es una función específica para manejo de strings, como su nombre lo indica (*get string*) y por lo tanto, es de comportamiento más eficiente respecto de `scanf`, que está preparada para otro tipo de tareas (el ingreso con formato).

La segunda diferencia radica en su funcionamiento: la función `gets()` requiere que se le entregue como argumento una posición de memoria. Tomará los caracteres que ingresen desde teclado y los irá almacenando a partir de dicha posición de memoria, hasta el ingreso de un **cambio de línea** (Enter). Completa su operación almacenando el carácter nulo a fin de “cerrar” la string.

Si repitiéramos el experimento anterior utilizando la función `gets()` en lugar de `scanf()`, el resultado sería el esperado.

```
char V1[20], V2[20], V3[20];
gets(V1);
gets(V2);
gets(V3);
```

Ni la función `scanf`, ni `gets` realizan comprobación de contorno de los vectores de almacenamiento, por lo tanto, deberá preverse la reserva de memoria suficiente para contener el texto que será ingresado desde teclado a los efectos de evitar desbordamientos de memoria.

Existen funciones específicas que se utilizan en aplicaciones reales para prevenir estas situaciones que derivan en problemas de seguridad. Véase `sscanf`.

FUNCIONES DE IMPRESIÓN DE STRINGS

Las funciones de impresión de strings tienen como objetivo enviar los caracteres provenientes de un área de memoria, cuya dirección inicial se da como argumento, a la pantalla.

Las más utilizadas son `printf()` y `puts()`.

FUNCIÓN PUTS()

Esta es la función inversa a `gets()`. A partir de la posición de memoria entregada como argumento, la función `puts()` envía cada uno de los caracteres a la pantalla a partir de la posición donde se encuentre el cursor.

Cuando detecta el carácter nulo de finalización envía un **cambio de línea** a la pantalla. Esto significa que por cada uso de `puts()` se producirá automáticamente un cambio de línea.

A modo de ejemplo, el siguiente código:

```
char VEC[] = "Hola, yo soy Pepe." ;
puts(VEC);
puts("Y yo soy Lola.");
```

Da como resultado :

Hola, yo soy Pepe.
Y yo soy Lola.

Obsérvese que en el segundo caso se entregó a `gets()` una frase entre comillas dobles. Puede interpretarse, por comodidad, que la orden es mandar esa frase a la pantalla. Sin embargo, lo que realmente se le está entregando es una dirección de memoria en donde está almacenada la verdadera frase como secuencia de

bytes, como hemos descripto, en el segmento de datos de nuestro programa en ejecución.

Cuando se entrega un string literal entre comillas, éste en realidad ocupa algún lugar de la memoria, y lo que se entrega es su dirección de inicio.

FUNCIÓN PRINTF()

La función `printf()` ya es conocida por el lector, pero aquí se mostrará en detalle su aplicación a las strings.

En forma similar al caso anterior, la función `printf()` con el carácter de formato `%s`, requiere que se le entregue como argumento la dirección de memoria de comienzo de la string, siendo válida la consideración de la nota precedente.

Existen diferencias con la función `puts()`. En primer lugar, `printf()` no envía automáticamente el cambio de línea a la pantalla. Si se desea utilizarlo, habrá que indicarlo de forma explícita con `\n`.

Por otro lado, la incorporación de caracteres de formato a la función `printf()` permite tener algunas alternativas adicionales que trataremos a continuación.

A modo de ejemplo lograremos la misma impresión en pantalla del ejemplo del punto anterior, pero utilizando `printf()`:

```
char VEC[] = "Hola, yo soy Pepe.";
printf("%s\n", VEC);
printf("%s", "Y yo soy Lola.");
```

Nótese la presencia del cambio de línea explícito. Una variante de la última línea podría haber sido:

```
printf("Y yo soy Lola.");
```

La posibilidad de utilización de caracteres de formato (`%`) nos permite lograr tabulaciones con justificación a derecha o a izquierda.

La forma general `%-nx` indica:

- % Inicio de un carácter de formato. A continuación se indica la forma de impresión.
- El signo “menos” indica justificación izquierda. Su ausencia, justificación derecha.
- n Cantidad mínima de espacios que ocupará el elemento a imprimir.
- x Formato del elemento a imprimir.

Para ilustrar lo expuesto veamos los siguientes códigos y las impresiones resultantes.

Caso 1:

```
printf("1%12s%12s\n", "Pepe Luis", "Gonzalez");
printf("2%12s%12s\n", "Lola", "Miranda");
```

```
1 Pepe Luis    Gonzalez
2      Lola     Miranda
```

Caso 2 :

```
printf("1 %12-s %12-s\n", "Pepe Luis", "Gonzalez");
printf("2 %12-s %12-s\n", "Lola", "Miranda");
```

```
1 Pepe Luis    Gonzalez
2 Lola         Miranda
```

FUNCIONES DE TRATAMIENTO DE STRINGS

Existe una gran cantidad de funciones de manejo de strings, pero de todas ellas las más utilizadas son:

- `strcmp()`
- `strlen()`
- `strcpy()`

Todas estas funciones están vinculadas con la cabecera `string.h`, por lo que será siempre necesaria su inclusión.

FUNCIÓN STRCMP

Esta función recibe como argumentos dos strings y retorna un valor entero que indica el resultado de la **comparación** entre ambos.

Su prototipo es:

```
int strcmp(const char *str1 , const char *str2)
```

Esto significa que recibe las direcciones de inicio de dos strings, direcciones que serán constantes dentro de la función y por lo tanto ésta no las modificará.

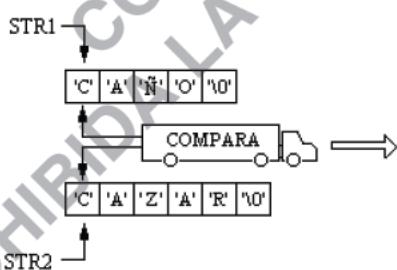
La función retorna un valor entero que responde al siguiente patrón de comparación:

Valor retornado > 0	→	STR1 > STR2
Valor retornado = 0	→	STR1 = STR2
Valor retornado < 0	→	STR1 < STR2

Regla mnemotécnica: A fin de recordar las relaciones anteriores suponga que la función strcmp() simplemente resta STR1 - STR2 como si fueran números.

¿Pero qué significa que una string sea *menor* que otra? Podemos considerar en primera instancia que si una string es menor que otra, irá ubicado primero en un ordenamiento alfabético. Pero esto no es tan simple.

El mecanismo según el que trabaja strcmp() es el de ir comparando los caracteres de cada string desde sus posiciones inferiores (en el dibujo, desde la izquierda) hasta encontrar algún par desequilibrante o bien el carácter nulo.



El valor del nulo es menor que el de cualquier carácter o símbolo, ya que su valor numérico es exactamente cero.

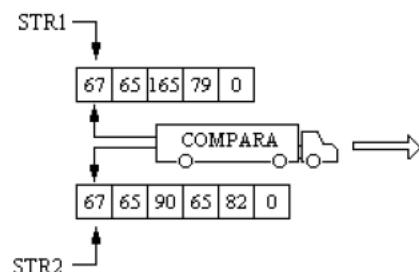
En el ejemplo de la figura se aprecia que la palabra "CAÑO" antecede a "CAZAR" en un ordenamiento alfabético. Es decir, "CAÑO" es *menor* que "CAZAR".

Sin embargo la comparación de caracteres que realiza `strcmp()` es a nivel de codificación ASCII.

Según la tabla ASCII, los códigos correspondientes a “Ñ” y “Z” son 165 y 90, respectivamente.

De lo anterior surge que la palabra “CAÑO” es *mayor* que “CAZAR” (erróneamente).

Del mismo modo debe considerarse que los códigos ASCII de las letras **minúsculas** son mayores numéricamente que los de sus correspondientes **mayúsculas**. No es lo mismo entonces comparar palabras con mayúsculas y minúsculas intercaladas.



Si se desea realizar la comparación con independencia de mayúsculas y minúsculas se puede utilizar la función `stricmp()` o bien la macro `strcmpi()`.

EJEMPLO: COMPARACIÓN DE STRINGS

En este ejemplo se permite el ingreso de strings por teclado hasta que uno de ellos sea igual a la palabra “FIN”.

Se considera que ninguna palabra ingresada por teclado superará los 20 caracteres.

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

#define MAX 20

int main()
{
    char string[MAX] ;
    printf("Ingrese palabras. Para finalizar : FIN \n\n") ;

    gets(string) ;
    while (strcmp(string,"FIN"))
        gets(string);

    printf ( "\n\nPrograma terminado \n" ) ;
}
```

FUNCIÓN STRLEN

La función `strlen` (*string length*) recibe como argumento la dirección de inicio de un string y retorna su **longitud**, considerada como la cantidad de caracteres que lo forman *sin contar el carácter nulo*.

Su prototipo es:

```
size_t strlen(const char * S)
```

El tipo retornado `size_t` es una macro definida como `unsigned int`, es decir es un entero no signado, y el lenguaje C lo utiliza para indicar medidas o cantidades que no pueden ser negativas.

FUNCIÓN STRCPY

La función `strcpy` (*string copy*) recibe como argumentos dos direcciones de memoria y copia a partir de la primera los caracteres que están ubicados a partir de la segunda, hasta encontrar el carácter de finalización nulo (el cual también es copiado).

Su prototipo es :

```
char* strcpy (char *DEST, const char *ORIG)
```

Nótese que la dirección `ORIG` figura en el prototipo como constante indicando que no será modificada dentro de la función.

El valor retornado es la dirección del string destino.

Es evidente que el string destino debe haber sido dimensionado adecuadamente para contener al string de origen.

EJEMPLO: LARGO DE STRINGS

Este es un ejemplo derivado del anterior.

Se ingresarán strings hasta que una de ellas sea “FIN”. Al igual que en el caso anterior, se supone que ninguno mide mas de 20 caracteres incluyendo al nulo.

Se debe informar cual es la string ingresada más larga.

```
#include <stdio.h>
#include <string.h>
#define MAX 20
```

```

int main()
{
    char string[MAX] , maslargo[MAX] ;

    printf("Ingrese palabras. Para finalizar : FIN \n\n");

    gets(string) ;           /* Primera lectura */
    strcpy(maslargo, string); /* El mas largo por ahora */

    while ( strcmp(string,"FIN") ) {
        if ( strlen(string) > strlen(maslargo) )
            strcpy (maslargo,string);
        gets ( string ) ;      /* Nueva lectura */
    }

    printf ( "\n\nMas largo : %s \n" , maslargo ) ;
}

```

ERRORES FRECUENTES CON STRINGS

Es muy frecuente (casi inevitable) que el programador que se inicia en el manejo de strings cometía errores relacionados con éstos.

El origen de estos errores es considerar que un string es una variable (y manejarlo como tal) cuando en realidad en C es una estructura de datos de tipo vector.

Las strings deben ser manipuladas a través de las funciones específicas antes descriptas a fin de evitar errores.

A continuación se mostrarán algunos ejemplos de estos errores y los resultados que producen.

EJEMPLO ERRÓNEO: ASIGNACIÓN

Un error frecuente es intentar asignar el contenido a una string en forma similar a como se asigna una variable numérica.

```

char STR[10];
STR = "PEPE";      /* ERROR!!! */

```

En este caso **STR** es el nombre de un vector y por lo tanto una constante. La línea de código anterior intenta asignar una dirección a una constante generando un

error de compilación que indica que se necesita una variable a la izquierda del operador de asignación. Esta situación ya la hemos mencionado en el capítulo sobre vectores como un error general.

Lo correcto hubiera sido emplear:

```
strcpy (STR, "PEPE"); /* correcto */
```

EJEMPLO ERRÓNEO: COPIA

En la copia de strings se puede producir un error similar al del caso anterior si se intenta de este modo:

```
char STR1[10], STR2[] = "PEPE";
STR1 = STR2;      /* ERROR!!! */
```

Se produce un error de compilación idéntico al del caso anterior debido a las mismas razones. No se puede asignar una constante, salvo en el momento de la declaración, como se hizo en la primera línea de código.

Nuevamente lo correcto es recurrir a la función strcpy().

```
strcpy (STR1, STR2); /* correcto */
```

Los errores mencionados no se producen si el destino de la copia o de la asignación es una variable de tipo puntero, como veremos más adelante.

EJEMPLO ERRÓNEO: COMPARACIÓN

En este caso se pretende determinar si el contenido de ambas strings es igual o distinto, mediante una comparación.

```
char STR1[] = "PEPE";
char STR2[] = "PEPE";

if (STR1 == STR2)
    printf("SON IGUALES");
else
    printf("SON DISTINTOS");
```

La situación es más compleja que en los casos anteriores debido a que una comparación directa de las strings, como se muestra en el código precedente, no genera un error de compilación. Esto se debe a que es una operación "*legal*" para el compilador.

Efectivamente, el compilador entiende que se están comparando los valores de las constantes STR1 y STR2. Estos valores corresponden a las **direcciones** de inicio de ambos vectores y son, evidentemente, distintos.

El resultado de este código es *siempre* la impresión "SON DISTINTOS" cuando lo que se esperaba era otra cosa, dando lugar a un error (conceptual) en tiempo de ejecución. El error es conceptual porque el programa es válido, pero no hace lo que nosotros suponemos.

La implementación correcta es mediante el empleo de la función `strcmp()` como se muestra a continuación:

```
char STR1[] = "PEPE";
char STR2[] = "PEPE";
if ( ! strcmp(STR1,STR2) )
    printf("SON IGUALES");
else
    printf("SON DISTINTOS");
```

Obsérvese que se está preguntando dentro de la condición si el resultado de la función `strcmp()` es cero, lo cual indica *igualdad* de los contenidos de STR1 y STR2.

VECTOR DE STRINGS

Hasta ahora hemos considerado los contenedores para un único nombre o frase. Frecuentemente se necesitará manejar listas de nombres o frases, y será necesario disponer de un *conjunto* manejable de este tipo de contenedores.

La estructura de datos adecuada para ello es un **vector** de los mencionados contenedores, cada uno de los cuales es a su vez un vector de char (string).

Este es el caso, entonces, de un **vector de strings** o "vector de vectores de caracteres". El elemento básico de esta estructura es el carácter.

La estructura definida de esta forma es una **matriz bidimensional** de caracteres o matriz de char.

Creemos más conveniente, en este caso, imaginar a toda esta estructura como un **vector de vectores**, aunque conviene siempre tener en claro el significado de cada término de las definiciones empleadas.

Consideraremos el caso en que se desean almacenar 5 nombres de ciudades de la antigüedad que no excederán los 18 bytes cada uno (incluyendo el carácter nulo).

Para cada nombre será necesario disponer de un vector de 18 caracteres, y a su vez, se deberán reservar 5 de estos vectores.

Llamaremos CIUDAD a la matriz de caracteres (o vector de strings), definida de la siguiente forma:

```
char CIUDAD[5][18];
```

De esta forma, cada CIUDAD[I] representará un nombre y CIUDAD[I][J] será una letra o carácter.

Podemos asignar 5 nombres de ciudades antiguas recurriendo a la función strcpy() de la siguiente manera:

```
strcpy(CIUDAD[0], "ALEJANDRIA");
strcpy(CIUDAD[1], "CONSTANTINOPLA");
strcpy(CIUDAD[2], "CARTAGO");
strcpy(CIUDAD[3], "ESPARTA");
strcpy(CIUDAD[4], "BABILONIA");
```

Otra forma de asignar estos nombres es en el momento de la declaración del vector, como se muestra a continuación:

```
char CIUDAD[][18] = {"ALEJANDRIA", "CONSTANTINOPLA", \
                      "CARTAGO", "ESPARTA", "BABILONIA"};
```

La siguiente figura muestra la disposición de los caracteres dentro de cada string. Las casillas sombreadas representan posiciones de memoria que no fueron asignadas y contienen “basura”.



A continuación se acompañan varios ejemplos de manejo de strings en diversas situaciones con el objetivo de familiarizarse con su utilización.

EJEMPLO: VECTOR DE STRINGS

Ingresar 10 strings que no superen los 20 caracteres cada una (incluyendo el carácter nulo). Al finalizar el ingreso se debe mostrar la lista en pantalla, en orden inverso al de ingreso.

```
#include <stdio.h>
#define MAX 20
#define NUM 10

int main()
{
    char MAT [NUM] [MAX] ;
    int I ;
    printf("Ingrese 10 palabras. \n\n") ;

    /* INGRESO */
    for ( I=0 ; I<NUM ; I++ ) {
        printf ("Palabra %3d : " , I+1 ) ;
        gets ( MAT[I] ) ;
    }

    /* IMPRESION */
    printf("\n\nVector invertido \n") ;
    for ( I = NUM-1 ; I>=0 ; I-- )
        printf ("\n%s" , MAT[I] ) ;

    printf ( "\n\nFin del programa" ) ;
}
```

EJEMPLO: BÚSQUEDA EN VECTOR DE STRINGS

Se ingresarán 10 strings en un vector. Posteriormente se ingresará una string adicional con el objeto de buscarla dentro del vector.

Se debe indicar si este string se encuentra dentro del vector o no, y en caso afirmativo, en qué posición.

Se limita la longitud de las strings a 20 caracteres incluyendo al carácter nulo.

Se utiliza el procedimiento tradicional de búsqueda secuencial.

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

#define MAX 20
#define NUM 10
```

```

int main()
{
    char MAT[NUM][MAX] , palabra[MAX];
    int I , POS = -1 ;
    printf("Ingrese 10 palabras. \n\n") ;

    /* INGRESO */
    for ( I=0 ; I<NUM ; I++ ) {
        printf ("Palabra %d : " , I+1 ) ;
        gets ( MAT[I] ) ;
    }

    printf ("\n\nIngrese palabra a buscar : ");
    gets(palabra) ;

    /* BUSQUEDA */
    for( I=0 ; (I<NUM) && (POS== -1) ; I++ )
        if ( ! strcmp ( MAT[I] , palabra ) )
            POS = I ;

    /* IMPRESION */
    printf("\n\nVector de busqueda \n") ;
    for ( I = 0 ; I<NUM ; I++ )
        printf ("\n%10d\t%s" , I , MAT[I] ) ;
    printf("\n\n");

    if ( POS == -1 )
        printf("%s no se encontró", palabra);
    else
        printf("%s está en la posición %d", palabra , POS);

    printf (" \n\nFin del programa" ) ;
}

```

EJEMPLO: ORDENAMIENTO EN VECTOR DE STRINGS

El objetivo de este programa es ordenar alfabéticamente una lista de strings que se ingresarán de forma similar a la del ejemplo anterior.

El método de ordenamiento utilizado es el de burbujeo. Debe prestarse especial atención a la etapa de ordenamiento, en la cual el contenedor de **intercambio** es una string de las mismas características de las que integran el vector.

Naturalmente, la comparación se realiza utilizando la función `strcmp()` y el intercambio con la función `strcpy()`.

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

```

#define MAX 20
#define NUM 10

int main()
{
    char MAT [NUM] [MAX] , AUX [MAX] ;
    int I , J ;
    printf("Ingrese 10 palabras. \n\n") ;

    /* INGRESO */
    for ( I=0 ; I<NUM ; I++ ) {
        printf ("Palabra %3d : " , I+1 ) ;
        gets ( MAT[I] ) ;
    }

    /* ORDENAMIENTO */
    for( I=0 ; I<NUM-1 ; I++ )
        for ( J=0 ; J<NUM-I-1 ; J++ )
            if ( strcmp(MAT[J] , MAT[J+1]) > 0 ) {
                strcpy ( AUX , MAT[J] ) ;
                strcpy ( MAT[J] , MAT[J+1] ) ;
                strcpy ( MAT[J+1] , AUX ) ;
            }

    /* IMPRESION */
    printf("\n\nVector ordenado \n") ;
    for ( I = 0 ; I<NUM ; I++ )
        printf ("\n%s" , MAT[I] ) ;

    printf ( "\n\nFin del programa" ) ;
}

```

EJEMPLO: ORDENAMIENTO CON ARRASTRE, VECTORES APAREADOS DE STRINGS

En este caso se manejarán dos vectores apareados correspondientes a una nómina de alumnos y sus correspondientes notas.

Se pretende obtener el listado ordenado en forma decreciente de notas.

Nótese que el *vector clave de ordenamiento* es el de notas. El vector de nombres será “arrastrado” por aquel en todos los intercambios. Para esto se utilizará la función `strcpy()` y un vector auxiliar, como en el ejemplo anterior.

```

#include <stdio.h>
#include <string.h>
#define MAX 20
#define NUM 10

```

```

int main()
{
    char MAT [NUM] [MAX] , AUX[MAX] ;
    int NOTA[NUM] , NOTAUX , I , J ;
    printf("Ingrese los datos de 10 alumnos. \n\n") ;

    /* INGRESO */
    for ( I=0 ; I<NUM ; I++ ) {
        printf ("\nAlumno %3d : " , I+1 ) ;
        fflush(stdin); /* limpia el buffer de teclado */
        gets ( MAT[I] ) ;
        printf ("Nota : " ) ;
        scanf( "%d" , &NOTA[I] ) ;
    }

    /* ORDENAMIENTO */
    for( I=0 ; I<NUM-1 ; I++ )
        for ( J=0 ; J<NUM-I-1 ; J++ )
            if ( NOTA[J] < NOTA[J+1] ) {

                NOTAUX      = NOTA[J] ;
                NOTA[J]     = NOTA[J+1] ;
                NOTA[J+1] = NOTAUX ;

                /* ARRASTRE */
                strcpy ( AUX , MAT[J] ) ;
                strcpy ( MAT[J] , MAT[J+1] ) ;
                strcpy ( MAT[J+1] , AUX ) ;
            }

    /* IMPRESION */
    printf("\n\nNOMINA DE ALUMNOS \n") ;
    for ( I = 0 ; I<NUM ; I++ )
        printf ("\n%30s%10d", MAT[I] , NOTA[I] ) ;
    printf ( "\n\nFin del programa" ) ;
}

```

La presencia de la llamada a la función `fflush(stdin)` se debe a que `scanf` deja en el buffer de lectura de teclado una ocurrencia del carácter Enter, que luego es encontrado por `gets` en la próxima vuelta del bucle, lo que provoca un ingreso incorrecto. Esta función indica el descarte (*flush*) del contenido de este buffer de teclado, lo que corrige la situación.

PROBLEMAS PROPUESTOS

1. Realizar un programa que permita comprobar la diferencia de comportamiento entre las funciones `gets()` y `scanf()`.
2. Construya una función de comparación de strings similar a `strcmp()` que contemple la existencia de las letras “ñ” y “Ñ”.
3. El programa que sigue permite el ingreso de 2 strings por teclado y luego los muestra en pantalla.

```
#include <stdio.h>
int main()
{
    char VEC[2][5] , I;
    for(I=0;I<2;I++)
        gets(VEC[I]);
    printf("\n\n\n");
    for(I=0;I<2;I++)
        puts(VEC[I]);
}
```

Explique la diferencia de comportamiento cuando los datos cargados son “PEPE” y “LOLA”, y cuando son “CARLOS” y “MAGNO”.

4. Repetir el problema del ejemplo “búsqueda en vector de strings”, utilizando el método de búsqueda binaria en lugar de la búsqueda secuencial.
5. Ingresar la nómina de 10 alumnos (es decir, sus nombres) y sus respectivos promedios. Determinar quién es el alumno de mejor promedio (suponerlo único).

7. ESTRUCTURAS

Una estructura es una colección de variables, no necesariamente del mismo tipo, referenciadas bajo un nombre común (el de la variable de tipo estructura) e individualizadas por nombres particulares.

Muchas veces se dispone de información en forma de datos de diferente naturaleza, todos ellos relacionados entre sí. Debido a esta relación, se desea almacenar dichos datos *agrupados*, y no *dispersos* en distintas variables independientes.

Los vectores no ofrecen una solución al problema debido a su limitación en cuanto a la necesidad de que el **tipo** de las variables debe ser el mismo.

Se recurre entonces a una estructura de datos denominada comúnmente **registro** o *record*, en inglés. Esta estructura es similar a una ficha en la que se dispone de particiones para almacenar información diversa. Cada una de estas particiones recibe el nombre de **campo** o **miembro**.

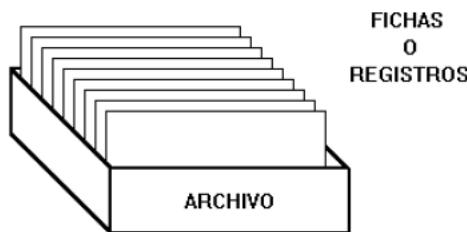
La palabra *registro* tiene diversas acepciones que conviene aclarar. Un registro puede considerarse como un conjunto de dispositivos electrónicos capaces de almacenar un bit cada uno, y por lo tanto, en su conjunto, podrán albergar una **palabra** de una longitud determinada. Este sentido del vocablo *registro* como un dispositivo de almacenamiento (memoria) es al que aludimos cuando nos referimos a los *registros del microprocesador*. Estos se denominan en inglés como *register* (y no *record*).

En cambio, la acepción de *registro* como estructura de datos, que es la que tratamos en este caso, recibe la denominación de *record* y así se la considera en algunos lenguajes de programación diseñados para enseñanza como, por ejemplo, el lenguaje **Fascal**.

Un ejemplo de un *registro* de este tipo es cada una de las fichas que lleva un médico de sus pacientes. En cada una de ellas está almacenada información de distinta índole como por ejemplo: número de historia clínica, nombre y apellido del paciente, edad, obra social, etc.

Todos estos datos están relacionados entre sí, pues corresponden a un mismo paciente físico.

Estas fichas se agrupan generalmente, para formar una estructura mayor denominada **archivo**.



En lenguaje C, estos registros reciben el nombre de **estructuras**, y son **tipos de datos no estándar**. Es decir, el lenguaje no los trae incorporados, sino que el usuario debe crearlos según su necesidad.

DECLARACIÓN DE ESTRUCTURAS

```
struct nombre_del_tipo_de_estructura {
    tipo_de_dato    nombre_de_campo ;
    tipo_de_dato    nombre_de_campo ;
    . . . . . .
    tipo_de_dato    nombre_de_campo ;
} nombre_de_variable ;
```

- El lenguaje C utiliza la palabra reservada `struct` para indicar variables de tipo estructura.
- En la sintaxis de declaración, `tipo_de_dato` es cualquier tipo reconocido por C, incluyendo tipos definidos por el usuario previamente, como ser otras estructuras.
- El `nombre_de_campo` es un nombre elegido por el usuario y servirá para identificar al mencionado campo dentro de la variable estructura.
- El `nombre_del_tipo_de_estructura` sirve para identificar al nuevo tipo de dato creado (dado que obviamente se pueden crear muchos tipos de estructuras diferentes).
- El `nombre_de_variable` es un nombre seleccionado por el usuario y será la referencia común de todos los campos que contiene la estructura, creando una variable para ella.

En la declaración mostrada precedentemente, tanto el `nombre_del_tipo_de_estructura` como el `nombre_de_variable`, pueden omitirse pero no simultáneamente.

Esto da lugar a algunas variantes de declaración que se mostrarán utilizando un ejemplo a continuación:

```
struct tipo_nuevo {  
    int campo_A ;  
    float campo_B ;  
} VAR1 , VAR2 ;
```

Este es el formato más general, presentado anteriormente, que declara dos variables de tipo estructura VAR1 y VAR2. Sólo tiene sentido indicar el nombre del nuevo tipo de dato (**tipo_nuevo** en este ejemplo) si es que se lo va a utilizar en algún otro lugar del programa, por ejemplo, para declarar más variables de este nuevo tipo.

De no ser así se puede omitir, con lo que se obtiene el siguiente formato :

```
struct {  
    int campo_A ;  
    float campo_B ;  
} VAR1 , VAR2 ;
```

Una tercera variante es la de crear el nuevo tipo de dato sin declarar variable alguna en ese momento. Como el tipo de dato ya está declarado, pueden declararse las variables posteriormente con este nuevo tipo.

Esto tiene la ventaja de poder crear un formato de estructura que tenga visibilidad **global**, mientras que las variables declaradas de este tipo pueden ser **locales**.

```
struct tipo_nuevo { /* Declaración del tipo */  
    int campo_A ;  
    float campo_B ;  
};  
.....  
/* Declaración de variables con ese nuevo tipo */  
struct tipo_nuevo VAR1 , VAR2 ;
```

Es importante notar la diferencia entre **declarar un tipo** y **declarar una variable**. La declaración de tipo no ocupa lugar en la memoria. La declaración de variables sí, dependiendo de cuánto espacio se necesite para almacenar la información de ese tipo. Si bien le decimos *declaración* a ambas situaciones, declarar un tipo y declarar una variable generan situaciones diferentes en el momento de la compilación.

ACCESO A LOS CAMPOS DE UNA ESTRUCTURA

Para poder acceder a las variables que integran la estructura se utiliza el **operador miembro . (punto)**. Este operador indica que estamos referenciando a un campo *miembro* de una variable, siendo ésta de tipo estructura.

La sintaxis será:

```
variable_estructura.nombre_de_campo
```

Ejemplo:

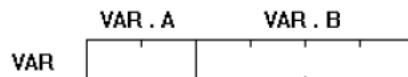
```
struct PARES {  
    int    A ;  
    float  B ;  
}    VAR ;
```

Tendremos aquí que:

- **VAR.A** es una variable de tipo **int**.
- **VAR.B** es una variable de tipo **float**.
- **VAR** es una variable de tipo **struct PARES**.

Los tipos creados cuando se declaran estructuras siempre contienen la palabra **struct** como parte de su nombre. Decimos que la variable **VAR** no es de tipo **PARES** sino de tipo **struct PARES**.

ESQUEMA DE LA ESTRUCTURA



COMPARACIÓN ENTRE ESTRUCTURAS Y VECTORES

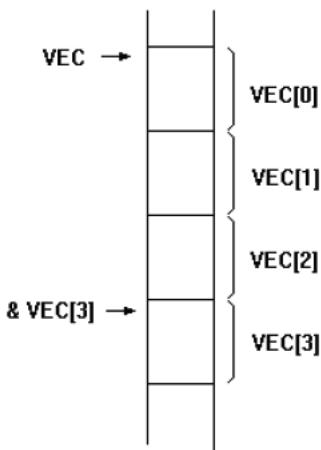
Un vector es una colección de variables del mismo tipo referenciadas por un nombre común, y diferenciadas por un subíndice numérico.

Esta definición es de alguna manera parecida a la de las estructuras, pero conviene resaltar algunas diferencias fundamentales.

El **subíndice** numérico de los vectores permite darles un tratamiento algebraico que no se puede realizar con los campos de las estructuras.

Los vectores requieren que las variables que los integran sean del mismo tipo. Esta es una limitación de los vectores y tiene su razón de ser en la **aritmética de subíndices**.

Veamos esto con un ejemplo:



Declaramos un vector llamado VEC compuesto por 4 variables enteras.

```
int VEC[4];
```

La dirección de la componente *i*-ésima del vector se calcula como la dirección inicial del vector mas *i* veces el tamaño en bytes del tipo de dato del vector.

Si tuviésemos componentes del vector de distinto tipo (y seguramente de distinto tamaño) el cálculo anterior arrojaría direcciones erróneas.

Por esta razón, para calcular la dirección de la variable VEC[3] tendríamos:

```
&VEC[3] = VEC + 3 * sizeof(int)
```

El **nombre del vector** está asociado con su **dirección** de inicio. Podemos decir que el nombre es equivalente a esta dirección.

Es el compilador el que asigna la dirección del vector, la que no variará a lo largo de todo el proceso. Estamos hablando entonces de una **constante**. Para ser más específicos, podemos decir que el nombre del vector es una constante de tipo **puntero** (contiene una dirección), o si se prefiere, un **puntero constante**.

Esto tiene sus implicancias: no se pueden copiar vectores directamente, como ya hemos visto.

Las **estructuras** en cambio, son variables en sí mismas y por lo tanto, pueden ser copiadas unas a otras. Esto es una gran ventaja.

Dos estructuras del mismo tipo pueden ser copiadas transfiriendo uno por uno los contenidos de sus campos, o bien transfiriendo el bloque completo de la estructura (lo cual es más cómodo).

EJEMPLO: COPIA DE ESTRUCTURAS

```
#include <stdio.h>
struct par_de_valores {           /* Declaración de tipo */
    int A ;
    float B ;      } ;

int main ( )
{
    struct par_de_valores primera , segunda ;

    primera.A = 4 ;           /* Asignación de valores */
    primera.B = 7.5 ;

    segunda = primera ;        /* Copia de estructuras */
    printf ( "\n\n %d %f " , segunda.A , segunda.B ) ;
}
```

Se verifica que se muestran en pantalla los valores de los campos, como si estos se hubieran copiado individualmente.

Si las estructuras son de tipo diferente también pueden ser copiadas empleando un casting, pero se generará seguramente una asignación errónea.

NO PORTABILIDAD DEL TAMAÑO DE LAS ESTRUCTURAS

Podemos suponer en una primera aproximación que la cantidad de bytes de memoria que ocupa una estructura, es la suma de lo que ocupa cada campo individualmente.

Esto no siempre es así, dependiendo de la computadora y del compilador que se trate. Las variables se “alinean” en memoria para maximizar las lecturas con una sola instrucción y, de esta forma, hacerlo más rápidamente. Estas optimizaciones dependen de la arquitectura del hardware y de las decisiones que tome el compilador al compilar.

Para asegurar la **portabilidad** del tamaño de una estructura será necesario utilizar la función `sizeof()` aplicada al tipo de estructura en cuestión, o bien a la variable de tipo estructura. El próximo ejemplo muestra el efecto de la alineación de las variables y la necesidad del `sizeof`.

```
#include <stdio.h>

int main()
{
    struct {
        int A ;
        char B ;
        int C ; } prueba ;

    printf("int %d \n", sizeof(int) ) ;
    printf("char %d \n", sizeof(char) ) ;
    printf("prueba %d \n", sizeof(prueba) ) ;
}
```

La ejecución arrojó correctamente 4 bytes para `int` y 1 para `char`, pero sorprendentemente 12 para la estructura.

No se debe considerar al tamaño de una estructura como la suma del tamaño de cada uno de sus campos.

COPIA DE VECTORES UTILIZANDO ESTRUCTURAS

Como sabemos, no es posible realizar la copia directa de vectores. Sin embargo, es posible implementar un “engaño” mediante estructuras para llevar a cabo dicha copia.

Consideremos la siguiente situación: si dos estructuras tienen un campo único consistente en un vector, estos podrán ser copiados directamente al realizarse la copia de las estructuras (lo cual es “legal”), como se muestra en el siguiente ejemplo.

EJEMPLO: COPIA DE VECTORES MEDIANTE ESTRUCTURAS

En este ejemplo se declara el tipo de estructura `VECTOR` que contiene un campo vector de 4 enteros, denominado `VEC`.

Se declaran dos variables del tipo de `struct VECTOR`, denominadas `VEC1` y `VEC2`.

No confundir a estas últimas con vectores, dado que son estructuras.

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

```

struct VECTOR {
    int VEC[4] ;
};

int main ()
{
    struct VECTOR VEC1 , VEC2 ;
    int I ;

    for (I=0 ; I<4 ; I++ )
        VEC1.VEC[I] = 2 * I ;

    for (I=0 ; I<4 ; I++ ) /* Primera impresion */
        printf("%10d", VEC2.VEC[I] );

    VEC2 = VEC1 ;           /* Copia de las estructuras */

    printf("\n\n");
    for (I=0 ; I<4 ; I++ ) /* Segunda impresion */
        printf("%10d", VEC2.VEC[I] );
}

```

En el programa se realiza la inicialización de cada elemento del vector de la estructura `VEC1` con el doble del valor de su subíndice.

Posteriormente se imprime el contenido del vector contenido en la estructura `VEC2`, antes y después de la copia directa desde `VEC1`.

En el primer caso, se muestran valores aleatorios (basura presente en la memoria). En el segundo, los valores asignados al vector de la estructura `VEC1`.

PASAJE DE ESTRUCTURAS A FUNCIONES

Considerando que las estructuras son variables como las demás, se las puede transferir como argumento a las funciones, y ser retornadas de ellas, del mismo modo.

Estamos diciendo entonces que la transferencia de estructuras a funciones responde a un **pasaje por valor**.

Es necesario, sin embargo, contemplar dos aspectos. Dado que las estructuras suelen ser voluminosas es preciso considerar lo siguiente:

- **Uso de memoria de pila:** Dado que la transferencia implica una copia del dato a un parámetro formal de la función, mientras dicha función esté activa, se estará ocupando el doble de memoria. Recordemos que las

variables locales viven en la pila y que el pasaje por valor implica la transferencia de una copia.

- **Tiempo de transferencia:** Por el mismo motivo, como es necesario realizar la copia de numerosos bytes, esta operación demandará un tiempo de transferencia que, en algunos casos, puede ser considerable.

Debido a los puntos expuestos anteriormente, puede ser conveniente considerar transferir las estructuras voluminosas **por referencia**. De todas formas, este punto se desarrollará más adelante cuando se introduzcan los **punteros**.

EJEMPLO: ESTRUCTURA COMO ARGUMENTO DE UNA FUNCIÓN

En este ejemplo muy simple, se mostrará como se transfiere una estructura *por valor* a una función, redeclarando el tipo de la estructura como tipo local de la función.

```
#include <stdio.h>

int main (void)
{
    struct {
        int    ENT ;
        char   CAR ;
        float  FLO ;
    } ESTR ;

    ESTR.ENT = 25 ;
    ESTR.CAR = 'A' ;
    ESTR.FLO = 7.34567 ;

    imprimir (ESTR) ; //llamada a la función
}

void imprimir ( struct {
    int    A ;
    char   C ;
    float  F ;
} PARAM )
{
    printf ( "%10d%10c%20f", PARAM.A, PARAM.C, PARAM.F);
}
```

El objetivo del programa es crear una estructura con diversos campos, asignarlos, e invocar a la función de impresión a fin de que ésta los muestre en pantalla.

Es importante notar que en la declaración del tipo de la estructura, no se asignó nombre de tipo alguno. Fue necesario redeclarar la estructura en la función, a fin de establecer un parámetro formal de ese tipo, para recibir el argumento.

De todas formas, si se hubiera asignado un **nombre de tipo** en la declaración del tipo estructura, realizada en el programa principal, no hubiera sido reconocida en la función, dado que la declaración hubiera sido *local del main*.

Es necesario además tener suma precaución en la redeclaración del tipo estructura dentro de la función. Recordemos que los argumentos transferidos a una función se “depositan” en la pila, de donde son “tomados” por los parámetros formales. No hay ningún tipo de comprobación de correspondencia de dichos argumentos con los mencionados parámetros. Es decir, éstos “se toman como vienen”. Si hay una discrepancia en la transferencia, en cuanto al orden o tipos de los argumentos, esto redundará en errores *en tiempo de ejecución*.

EJEMPLO: TRANSFERENCIA INCORRECTA

```
#include <stdio.h>
int main (void)
{
    struct {
        int    ENT ;
        char   CAR ;
        float  FLO ;
    } ESTR ;

    ESTR.ENT = 25 ;
    ESTR.CAR = 'A' ;
    ESTR.FLO = 7.34567 ;

    imprimir ( ESTR ) ;
    printf ( "\n\n%10d%10c%20f", ESTR.ENT, ESTR.CAR, ESTR.FLO);
}

void imprimir ( struct {
    float F ;
    char  C ;
    int   A ; } PARAM )
{
    printf ( "%10d%10c%20f", PARAM.A, PARAM.C, PARAM.F);
}
```

En este ejemplo se muestra el efecto de transferir incorrectamente los argumentos de los campos de una estructura. Obsérvese que el *orden* de los campos en la redeclaración de la estructura dentro de la función fue alterado.

Las dos impresiones de datos dan valores diferentes en cada caso. No hay advertencias del compilador ya que “el programador sabe lo que hace”.

TIPOS GLOBALES Y VARIABLES LOCALES

A fin de minimizar el riesgo del caso comentado anteriormente, como así también agregarle simplicidad y claridad al programa, se recomienda realizar la declaración del nuevo tipo de dato en forma global. Para esto, será necesario asignar un **nombre de tipo**.

El hecho de utilizar un **tipo global**, trae como beneficio que este tipo de dato es reconocido en todas las funciones. Declarar de esta forma un tipo, no trae aparejada ninguna reserva de memoria. No está vinculado de ninguna manera a **variables globales** por sí mismo.

Lo único global es el tipo de dato. Las variables se declararán en forma local utilizando para ello el nuevo tipo de dato, ya reconocido por el compilador.

EJEMPLO: TIPOS GLOBALES Y VARIABLES LOCALES

```
#include <stdio.h>

struct NUEVA {           /* Declaración de tipo global */
    int    ENT ;
    char   CAR ;
    float  FLO ; } ;

int main (void)
{
    struct NUEVA ESTR ; /* Variable local con un tipo global */

    ESTR.ENT = 25 ;
    ESTR.CAR = 'A' ;
    ESTR.FLO = 7.34567 ;

    imprimir ( ESTR ) ;
}

void imprimir ( struct NUEVA PARAM ) /* Se reconoce el tipo global */
{
    clrscr ( ) ;
    printf ( "%10d%10c%20f", PARAM.ENT, PARAM.CAR, PARAM.FLO ) ;
}
```

En este caso se ha utilizado el mismo programa de los ejemplos anteriores a fin de facilitar la comparación. Se crea en forma global el tipo de estructura NUEVO

que posteriormente es reconocido dentro del programa principal y de la función `imprimir()`.

Nótese que dentro de la función es necesario utilizar los mismos nombres de campo que fueron declarados al crear el tipo `struct NUEVA`.

Cuando se trabaja con estructuras que serán transferidas a funciones se recomienda declarar en forma global los tipos de datos de dichas estructuras, de manera que tales tipos sean válidos y utilizables en todas las funciones.

ESTRUCTURAS RETORNADAS DE FUNCIONES

El **retorno** de estructuras desde funciones se realiza como el de cualquier otro tipo de variable, con las consideraciones de tiempo y uso de memoria realizados anteriormente.

Se mostrará el **pasaje y retorno** de estructuras a funciones mediante el siguiente programa de ejemplo.

EJEMPLO: RETORNO DE ESTRUCTURAS DESDE FUNCIONES

En este ejemplo sencillo se pide el ingreso de los campos de una estructura de tipo `struct FECHA`, cuyo tipo de dato fue declarado globalmente.

Se transfiere esta estructura a la función `corregir()`, la cual verifica si se trata del 29 de febrero y en ese caso, transforma la fecha en el 1 de marzo.

Corregida o no dicha fecha, la mencionada función la retorna al programa invocante donde se la almacena en la misma variable estructura `HOY`, que fue utilizada para transferir hacia la función.

Nótese que tal vez resulte más conveniente, en este caso, realizar una transferencia por referencia de la estructura, pero ese tema será desarrollado con posterioridad.

Asimismo nótese la utilización del ampersand & en la lectura de `scanf()` para cada uno de los campos de la estructura. Esto es así debido a que el nombre de cada campo de una estructura es una variable y no una dirección.

```
#include <stdio.h>

struct FECHA {
    int DIA ;
    int MES ;
    int ANIO ; } ;
```

```

/* Prototipo de la función */
struct FECHA corregir ( struct FECHA ) ;

int main (void)
{
    struct FECHA HOY ;
    printf ( "Ingrese la fecha con formato DD-MM-AA    " ) ;
    scanf ("%d-%d-%d" , &HOY.DIA , &HOY.MES , &HOY.ANIO ) ;

    HOY = corregir (HOY) ;

    printf ( "\n\nFECHA CORRECTA : " ) ;
    printf ( "%02d-%02d-%02d" , HOY.DIA , HOY.MES , HOY.ANIO ) ;
}

struct FECHA corregir ( struct FECHA DIA )
{
    if ( DIA.DIA == 29 && DIA.MES == 2 ) {
        DIA.DIA = 3 ;
        DIA.MES = 1 ;
    }
    return DIA ;
}

```

VECTORES APAREADOS VS. VECTORES DE ESTRUCTURAS

Numerosas veces hemos tenido que manejar bases de información relacionada de distintos tipos y hemos recurrido a los **vectores apareados**.

Es decir, distintos vectores de variables de igual o distinto tipo, pero que mantienen una relación de correspondencia lógica entre los elementos del mismo subíndice.

En estos casos, un ordenamiento utilizando a uno de los vectores como campo clave implica el *arrastre* de los otros vectores en el proceso de *swapping*, a fin de mantener la mencionada relación de correspondencia.

Esta situación equivale a realizar tantos *swappings* como vectores apareados se tenga, y al uso (en el peor de los casos) de otras tantas variables auxiliares de intercambio.

La utilización de estructuras hace que sea innecesario el uso de vectores apareados y podemos considerarlos una aplicación obsoleta.

Se mostrará a continuación la resolución de un problema típico de ordenamiento de vectores aparentados, pero en este caso, utilizando un vector de estructuras. Se sugiere al lector la resolución previa del mismo problema, sin utilizar estructuras, a fin de comparar posteriormente ambos métodos.

EJEMPLO: ORDENAMIENTO DE VECTOR DE ESTRUCTURAS

Esta estructura de datos es denominada frecuentemente “*Archivo en RAM*” debido a que es una **colección** de estructuras o registros, como se discutió anteriormente, y su lugar de residencia es la memoria RAM de la computadora.

El problema que se presentará a continuación tiene el siguiente enunciado:

Se ingresarán los datos correspondientes a 20 alumnos. Estos datos consisten en: nombre (string de 20 caracteres máximo incluyendo el nulo), sexo (char) y promedio (float).

Se pide permitir el ingreso de los datos y posteriormente mostrarlos en pantalla ordenados alfabéticamente.

El programa se modularizará utilizando 3 funciones. Una de éstas, la función `ordenar()`, es la que interesa fundamentalmente en este ejemplo.

Como se ve a continuación, el programa principal es muy simple. En él se aplican algunos conceptos descriptos anteriormente.

```
#include <stdio.h>
#define N 20

/* Declaración de tipo */
struct ALUMNO {
    char NOMBRE[20] ;
    char SEXO ;
    float PROMEDIO ;
};

/* Prototipos */
void ingresar ( struct ALUMNO [ ] , int ) ;
void ordenar ( struct ALUMNO [ ] , int ) ;
void imprimir ( struct ALUMNO [ ] , int ) ;

int main (void)
{
    struct ALUMNO DATO[N] ;

    ingresar ( DATO , N ) ;
    ordenar ( DATO , N ) ;
    imprimir ( DATO , N ) ;
}
```

Recuérdese que se está transfiriendo el vector de estructuras a las funciones por referencia (como todo vector), y por lo tanto, dentro de las mencionadas funciones se está trabajando (y modificando) el vector original que reside en la pila del llamante (el `main` en este caso).

```
void ingresar ( struct ALUMNO V[ ] , int NUM )
{
    int I ;
    float F ;
    for(I=0 ; I<NUM ; I++) {
        printf("Ingrese el nombre del alumno : ");
        fflush(stdin);
        gets(V[I].NOMBRE);

        printf("Ingrese el sexo M/F : ");
        V[I].SEXO = getchar();

        printf("\nIngrese el promedio del alumno : ");
        scanf("%f", &F ) ;
        V[I].PROMEDIO = F ;
    }
}
```

La función `ordenar()` utiliza la variable auxiliar `AUX` de tipo `struct ALUMNO` a fin de realizar el *swapping* del ordenamiento.

Para éste se utilizó el método del burbujeo. El campo clave de ordenamiento es el nombre del alumno (string) y por lo tanto debe ser comparado mediante la función `strcmp()`.

```
void ordenar ( struct ALUMNO V[ ] , int NUM )
{
    struct ALUMNO AUX ;
    int I , J ;
    for ( I=0 ; I<NUM-1 ;I++ )
        for ( J=0 ; J<NUM-I-1 ; J++ )

            if ( strcmp( V[J].NOMBRE , V[J+1].NOMBRE ) > 0 )
            {
                AUX      = V[J]      ;
                V[J]    = V[J+1] ;
                V[J+1] = AUX      ;
            }
}
```

```

void imprimir ( struct ALUMNO V[] , int NUM )
{
    int I ;
    for ( I=0 ; I<NUM ; I++ )
        printf ( "\n%20s%10c%12.2f" , V[I].NOMBRE , \
                  V[I].SEXO , V[I].PROMEDIO) ;
}

```

ESTRUCTURAS ANIDADAS

Como se mencionó en párrafos anteriores, una estructura puede contener variables de diferente tipo, reconocidos por C.

Vimos en el ejemplo anterior, un vector de estructuras que a su vez contenían vectores de caracteres (strings).

Puede darse el caso de estructuras en las que uno o más de sus campos sean otras estructuras (definidas previamente). Estamos entonces en el caso de **estructuras anidadas**.

A fin de aclarar el manejo de las estructuras anidadas plantearemos el siguiente ejemplo: Se declaran los tipos de estructura SECUNDARIA y PRIMARIA, como así también, la variable VAR.

```

struct SECUNDARIA {
    int A ;
    int B ;
    char C ;
} ;

struct PRIMARIA {
    float A ;
    struct SECUNDARIA X ;
} ;

struct PRIMARIA VAR ;

```

Ante la pregunta ¿qué es cada una de estas “cosas”? Podemos contestar:

- X Es un campo de la estructura de tipo PRIMARIA.
- A Es otro campo de la estructura de tipo PRIMARIA.
- VAR Es una variable de tipo struct PRIMARIA.
- VAR.A Es una variable de tipo float.
- VAR.X Es una variable de tipo struct SECUNDARIA.
- VAR.B No existe.
- VAR.X.A Es una variable de tipo int.
- VAR.X.C Es una variable de tipo char.

Podemos apreciar que se utiliza sucesivamente al **operador miembro** (el punto) para adentrarse en niveles más profundos del anidamiento.

De esta manera, dado que VAR.X es una estructura, VAR.X.C es entonces un campo de dicha estructura. Con otras palabras, a la izquierda del operador miembro sólo puede haber el nombre de una variable de tipo estructura.

EJEMPLO: ESTRUCTURAS ANIDADAS

Se planteará en este caso un programa similar al del ejemplo de ordenamiento anterior. En esta ocasión se ingresarán los datos de 20 alumnos compuestos por nombre, promedio y fecha de nacimiento. Esta fecha estará constituida por una estructura de tipo struct FECHA (vista en un ejemplo anterior).

Se desea mostrar los datos de los alumnos en orden creciente de edad. Como consideración se entiende que todos han nacido en el siglo XX.

```
#include <stdio.h>
#define N 20

struct FECHA {
    int DIA ;
    int MES ;
    int ANIO ; } ;

struct ALUMNO {
    char NOMBRE[20] ;
    struct FECHA NACIM ;
    float PROMEDIO ; } ;

/* Prototipos */
void ingresar ( struct ALUMNO [ ] , int ) ;
void ordenar ( struct ALUMNO [ ] , int ) ;
void imprimir ( struct ALUMNO [ ] , int ) ;
unsigned int dias ( struct FECHA ) ;

int main (void)
{
    struct ALUMNO DATO[N] ;

    ingresar ( DATO , N ) ;
    ordenar ( DATO , N ) ;
    imprimir ( DATO , N ) ;
}

void ingresar ( struct ALUMNO V[ ] , int NUM )
{
    int I ;
    float F ;
```

```

for ( I=0 ; I<NUM ; I++ ) {
    printf ( "Ingrese el nombre del alumno : " ) ;

    fflush ( stdin ) ;
    gets ( V[I].NOMBRE ) ;

    printf ( "Ingrese la fecha con formato DD-MM-AA " ) ;
    scanf ("%d-%d-%d" , &V[I].NACIM.DIA , &V[I].NACIM.MES , \
           &V[I].NACIM.ANIO ) ;

    printf ( "Ingrese el promedio del alumno : " ) ;
    scanf ( "%f" , &F ) ;
    V[I].PROMEDIO = F ;
}

void ordenar ( struct ALUMNO V[ ] , int NUM )
{
    struct ALUMNO AUX ;
    int I , J ;
    for ( I=0 ; I<NUM-1 ; I++ )
        for ( J=0 ; J<NUM-I-1 ; J++ )
            if ( dias(V[J].NACIM) < dias(V[J+1].NACIM) ) {
                AUX      = V[J] ;
                V[J]    = V[J+1] ;
                V[J+1] = AUX ;
            }
}

void imprimir ( struct ALUMNO V[ ] , int NUM )
{
    int I ;
    for ( I=0 ; I<NUM ; I++ )
        printf ( "\n%20s%10d%4d%4d%12.2f" , V[I].NOMBRE,\n
                  V[I].NACIM.DIA,V[I].NACIM.MES ,V[I].NACIM.ANIO ,\n
                  V[I].PROMEDIO ) ;
}

unsigned int dias ( struct FECHA date )
{
    long X ;
    X = 365 * date.ANIO + 31 * date.MES + date.DIA ;
    return X ;
}

```

Se acompañan las funciones a fin de apreciar las diferencias de tratamiento con el anterior ejemplo.

Se agrega la función `dias()` destinada a calcular la cantidad de días transcurridos desde el inicio del siglo XX hasta la fecha de nacimiento de cada individuo,

considerando años de 365 días y meses de 31 días. Cuanto mayor sea el valor retornado por `dias()`, menor es la edad del individuo.

Esta información se gestiona a los fines del ordenamiento, como se puede apreciar en la función `ordenar()`.

EJEMPLO INTEGRAL CON VECTORES DE ESTRUCTURAS

A continuación se presenta un ejemplo completo del uso de vectores de estructuras. Se trata de manejar un vector de estructuras que contienen a su vez vectores. Uno de estos vectores tiene contenidos numéricos, que determinan el ordenamiento del “macro-vector” de estructuras.

Este ejercicio tiene su origen en un examen final para el cual se precisaba demostrar conocimiento sobre los conceptos desarrollados hasta este punto.

EJEMPLO: WEST POINT

De un instituto militar egresan por cada promoción 200 soldados comandos, los cuales han sido calificados en 12 diferentes disciplinas de especialización (por ejemplo: explosivos, sabotaje, inteligencia, etc.). Dichas disciplinas se codifican con números del 1 al 12.

Estos soldados son novatos y están disponibles para ser asignados a su primera misión. Una vez cumplida ésta, no vuelven a integrar la lista de novatos, por lo que dicha lista se irá reduciendo paulatinamente.

El Estado Mayor solicitará grupos de estos comandos para misiones, indicando (por su código) cual es la especialidad que se requiere en la ocasión. Se debe asignar para el grupo de soldados pedido, a aquellos disponibles que tengan mejor calificación en la asignatura indicada. O sea, a los mejores en ese área.

Se requiere construir un programa que permita el ingreso de los datos de los soldados formados por el legajo, el nombre y las 12 notas de las especialidades.

Luego de esto, se debe permitir el ingreso de los requerimientos de los grupos, consistentes en cantidad de soldados y código de especialidad.

Se mostrará en pantalla el listado de los soldados asignados, con el puntaje que tienen en la especialidad pedida. Si la cantidad de soldados disponibles no alcanzara a satisfacer el número solicitado, se informará y se asignarán los que sean posibles.

El programa finaliza cuando la lista de soldados novatos haya quedado vacía.

WEST POINT: DESARROLLO

Se incorpora a la estructura `SOLDADO` un campo `flag` llamado `DISPONIBLE` que indica si el soldado en cuestión ya fue asignado anteriormente.

La variable VACANTES indica cuántos comandos quedan aún en la lista de novatos. El programa finaliza cuando el contenido de VACANTES es igual a cero.

El código de disciplina se ingresa en la variable CODIGO, la cual se decremente para ajustarla a la posición correspondiente dentro del vector de notas.

La función ordenar recibe CODIGO como argumento. Esto permite efectuar el ordenamiento, realizando una selección aritmética del campo clave de ordenamiento.

Obsérvese que el campo clave utilizado es FICHA[J].NOTA[CODIGO], dado que CODIGO es una variable numérica, es posible automatizar la selección del ordenamiento.

El método de ordenamiento utilizado es el conocido burbujeo sin flag.

```
/* WEST POINT */

#include <stdio.h>
#define MAX      200
#define MAX_MAT 12

struct SOLDADO {
    int LEGAJO ;
    char NOM[20] ;
    int NOTA[MAX_MAT] ;
    char DISPONIBLE ;
} ;

int main ( )
{
    struct SOLDADO DATO[MAX] ;
    int I , CODIGO , CANTIDAD , VACANTES = MAX ;

    /* INGRESO */
    INGRESO ( DATO , MAX ) ;

    while ( VACANTES ) {

        /* PEDIDOS */
        printf ( "\n\nIngrese codigo de asignatura preferencial " );
        scanf ( "%d" , &CODIGO ) ;
        CODIGO-- ; /* Ajuste a la posicion del vector */
        printf ( "\n\nIngrese la cantidad de comandos solicitada " );
        scanf ( "%d" , &CANTIDAD ) ;

        /* ORDENAMIENTO */
        ORDENA ( DATO , MAX , CODIGO ) ;
```

```

/* ASIGNACION */
if ( VACANTES < CANTIDAD ) {
    printf ( "NO ALCANZAN" ) ;
    VACANTES = 0 ;
}
else
    VACANTES -= CANTIDAD ;

for ( I=0 ; (I<MAX) && CANTIDAD ; I++ )
    if ( DATO[I].DISPONIBLE ) {
        DATO[I].DISPONIBLE = 0 ;
        CANTIDAD-- ;
        printf ( "\n%d\t%20s\t%d" , DATO[I].LEGAJO , \
DATO[I].NOM , DATO[I].NOTA[CODIGO] );
    }
}

INGRESO ( struct SOLDADO FICHA[ ] , int N )
{
    int I , J ;

    for ( I = 0 ; I<N ; I++ ) {
        printf("Legajo : ");
        scanf ( "%d" , &FICHA[I].LEGAJO ) ;
        fflush ( stdin ) ;
        printf("Nombre : ");
        gets ( FICHA[I].NOM ) ;
        printf("\nNotas : ");

        for ( J=0 ; J<MAX_MAT ; J++ )
            scanf ( "%d" , &FICHA[I].NOTA[J] ) ;
        FICHA[I].DISPONIBLE = 1 ;
    }
}

ORDENA ( struct SOLDADO FICHA[ ] , int N , int CODIGO )
{
    int I , J ;
    struct SOLDADO AUX ;
    for ( I=0 ; I<N-1 ; I++ )
        for ( J=0 ; J<N-I-1 ; J++ )
            if ( FICHA[J].NOTA[CODIGO] < FICHA[J+1].NOTA[CODIGO] ) {
                AUX      = FICHA[J] ;

```

```
    FICHA[J] = FICHA[J+1] ;  
    FICHA[J+1] = AUX ;  
}  
}
```

Se recomienda al lector tratar de resolver este problema sin utilizar estructuras, a fin de comprobar las dificultades que se plantean, ya sea por los arrastres involucrados en los vectores apareados, o por el uso de una matriz de notas.

PROBLEMAS PROPUESTOS

1. Los datos de los 130 integrantes de las divisiones inferiores de un club se encuentran almacenados en un vector de estructuras de la siguiente manera:

Nombre (string de 20 caracteres), sexo (char), fecha de nacimiento (estructura de tipo fecha: int dia, int mes, int año).

Se pide mostrar los nombres discriminados por sexo y categoría sabiendo que Juveniles incorpora hasta los nacidos en 2005, Cadetes hasta 2007, e Infantiles hasta 2009.

2. Se desea organizar un torneo de tenis doble mixto con 16 parejas. Los datos de los participantes son:

- Nombre (string de 20 caracteres)
- Sexo (char)
- Handicap (int)

Se pide:

- Estructurar los datos y permitir su ingreso desde teclado utilizando para ello un solo vector de estructuras.
- Indicar cuáles son las parejas formadas considerando que el hombre de mayor handicap debe jugar con la mujer de menor handicap y viceversa, y así sucesivamente, a fin de obtener un torneo parejo.
- Indicar los nombres de la pareja cuya suma de handicaps es la máxima (considerarla única).

3. Se desea llevar una estadística de los 80 alumnos de una carrera cuyos datos son:

- Nombre (string de 20)
- Sexo (char)
- Nota de 20 materias codificadas de 1 a 20 (int)
- Fecha de nacimiento (estructura de tipo fecha)

Se pide:

- Estructurar los datos y permitir el ingreso de los mismos desde teclado.
- Indicar cuál es el promedio de notas de los varones y de las mujeres (se asegura que hay por lo menos 10 de cada uno).
- Mostrar un listado de los datos ordenados en forma decreciente de una materia cuyo código se ingresa desde teclado.

- d. Permitir el ingreso de un nombre de alumno e indicar cuál es su promedio.
4. Se desea llevar el control de los datos de un curso de 12 alumnos. Los datos consisten en:
- Nombre (string de 20)
 - Nota de 10 materias (enteros de 0 a 10)
- Las materias son: Matemática, Física, Química, Economía, Algebra, Informática, Legislación, Tecnologías, Redacción y Proyecto.

Se pide:

- a. Permitir el ingreso por teclado de los nombres de los alumnos.
- b. Completar las notas con números enteros aleatorios comprendidos entre 0 y 10 inclusive.
- c. Mostrar en pantalla la totalidad de los datos, encolumnados correctamente y encabezados por los nombres de las materias, pero utilizando solamente las 3 primeras letras de cada uno de ellos (Mat, Fis, Qui, etc.)
- d. Permitir el ingreso del nombre de una materia e imprimir el listado de los alumnos que deben rendir examen recuperatorio de ésta por tener calificación menor que 4.

El programa debe resolverse siguiendo una lógica que considere que la cantidad de materias podría ser mucho mayor.

Se sugiere utilizar un vector auxiliar que contenga sus nombres.

5. Construir una función que reciba dos vectores de estructuras. El primero de ellos esta compuesto por 12 estructuras BASEDAT que contiene los datos de una lista de precios.

El segundo es un vector compuesto por estructuras AUMENTO. Este último finaliza con un nombre de proveedor llamado “FIN”.

Se pide:

- a. Actualizar la lista de precios aplicando los aumentos porcentuales indicados en el segundo vector a los artículos de cada proveedor del primer vector.
- b. Imprimir la lista de precios completa ordenada por proveedor y dentro de estos en forma creciente de número de artículo.

```
struct BASEDAT {
    int ART; // Número de artículo
    char DESC[50]; // Descripción del artículo
    char PROV[20]; // Nombre del proveedor
    float PRECIO; // Precio del artículo
};

struct AUMENTO {
    char PROV[20]; // Nombre del proveedor
    float AUM; // Aumento porcentual a aplicar
};
```

PROHIBIDA LA REPRODUCCIÓN TOTAL O PARCIAL

8. ENUMERACIONES, UNIONES, CAMPOS y OPERADORES DE BIT

ENUMERACIONES

Las **enumeraciones** son otro tipo de datos definido por el usuario.

Una enumeración es un conjunto de constantes enteras con identificador, definidas exhaustivamente por extensión dentro de su propio conjunto.

Esto significa que mediante una enumeración se define un conjunto de nombres que representan valores enteros invariantes. En el momento de la definición es necesario indicar todos y cada uno de los elementos del conjunto que se está definiendo.

SINTAXIS DE DECLARACIÓN

La declaración de una enumeración es similar a la de una estructura, excepto por el hecho de que todos los campos son enteros y constantes.

Es posible declarar el tipo de dato de la enumeración asignándole un *nombre de tipo*, y/o también declarar variables de ese tipo, que podrán contener valores pertenecientes al conjunto, y no otros.

Para la declaración se utiliza la palabra reservada `enum`, como se muestra a continuación:

```
enum nombre_de_tipo { constantes de enumeración } variable ;
```

Ejemplo:

```
enum GRIEGAS { ALFA , BETA , GAMMA } LETRA ;
```

Luego de esta declaración, GRIEGAS es un tipo de dato que incluye a las constantes enteras ALFA, BETA y GAMMA, mientras que LETRA es una variable entera que puede ser asignada con aquellos valores.

En forma similar al caso de la declaración de estructuras, puede omitirse, o bien el nombre del nuevo tipo, o bien el nombre de variable (pero no ambos).

Posteriormente se pueden declarar variables de la forma:

```
enum GRIEGAS OTRA_LETRA ;
```

y luego realizar asignaciones de la forma:

```
OTRA_LETRA = GAMMA ;
OTRA_LETRA = BETA + 1 ;
OTRA_LETRA = ALFA + BETA ;
```

VALORES DE LAS CONSTANTES

Las constantes de enumeración sólo pueden ser asignadas en el momento de la declaración.

Los valores asignados automáticamente comienzan con cero y se van incrementando, según la enumeración de las constantes. De esta manera, en el ejemplo anterior, ALFA vale 0, BETA vale 1 y GAMMA vale 2.

Esta asignación puede ser modificada por el usuario con las siguientes consideraciones:

- El usuario asigna el valor en el momento de la declaración.
- Las constantes posteriormente mencionadas toman sus valores a partir del definido.
- Varias constantes pueden tener el mismo valor.

Veamos estos aspectos mediante un ejemplo.

EJEMPLO 1: ENUMERACIONES

El objetivo de este ejemplo es simplemente observar los valores de las constantes de enumeración.

```
#include <stdio.h>

enum GRIEGAS { ALFA , BETA = 20 , GAMMA , EPSILON = 21 , PI } ;

int main ( )
{
```

```

enum GRIEGAS LETRA ;
LETRA = 2 * GAMMA + PI ;

printf ( "%4d%4d%4d%4d%4d" , ALFA , BETA , GAMMA ,
EPSILON , PI , LETRA ) ;
}

```

Los valores obtenidos son:

```
0 20 21 21 22 64
```

USO DE LAS ENUMERACIONES

La aplicación de las enumeraciones en un programa se reduce simplemente a brindarle mayor claridad y comprensión, pero no reporta una gran ventaja operativa.

En este aspecto, pueden ser utilizadas en los campos de comparación de un `switch` o como valor subíndice en un vector.

UNIONES

Una unión es una variable de tipo estructura en la que todos sus campos comienzan en la misma dirección de memoria.

Esto significa que la **unión**, como la **estructura**, es una colección de variables de igual o distinto tipo, referenciadas por un nombre común, e individualizadas por nombres particulares (nombres de campo o miembros).

Pero a diferencia de la estructura, los campos de la unión comparten el área de memoria de almacenamiento.

SINTAXIS DE DECLARACIÓN

La sintaxis de declaración es similar a la de las estructuras, con excepción del reemplazo de la palabra reservada `struct` por la palabra reservada `union`, como se muestra a continuación:

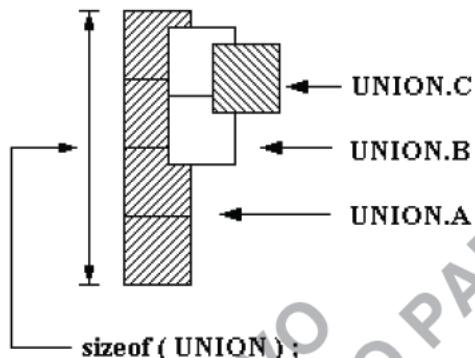
```

union nom_tipo {
    tipo nom_campo ;
    tipo nom_campo ;
    .....
} nom_variable ;

```

A continuación se observa la disposición de los campos de la unión declarada como se muestra.

```
union {  
    float A;  
    short int B;  
    char C;  
} UNION ;
```



Se observa en la figura la superposición de los campos, y el tamaño total resultante de la unión.

El tamaño de la unión es igual al tamaño del mayor de sus campos.

Es evidente que, dada la superposición de los campos, las sucesivas escrituras en estos modificarán los contenidos de los campos afectados por la superposición. De esta manera, es impensable un aprovechamiento de la unión.

Sin embargo, podemos mencionar dos aplicaciones bien diferentes para las uniones:

- Ahorro de memoria: cuando no se da la superposición de la información.
- Conversiones de formato: se aprovecha la superposición de campos.

AHORRO DE MEMORIA

Existen casos en los que se conoce que no habrá superposición de información y por lo tanto, se puede utilizar uno de los campos de la unión sin riesgo de ver afectada dicha información.

La pregunta que surge entonces es: ¿para qué utilizar entonces una unión, y no solamente el campo que contendrá la información?

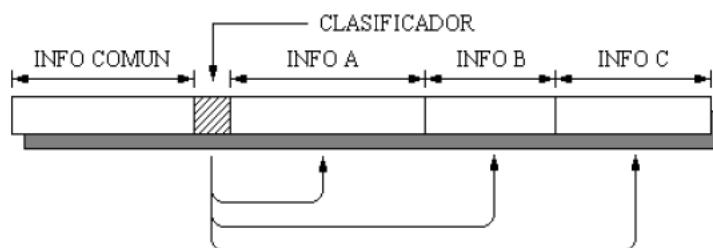
La respuesta es que tal vez no se conoce *a priori* (en tiempo de programación, cuando se hace el código) cuál será el campo utilizado, dado que este puede ir cambiando caso por caso.

Planteemos el caso de querer armar un único “*archivo en RAM*” con la información de una gran cantidad (N) de elementos clasificados en tres grupos.

Supongamos que la información concerniente a cada grupo es de estructura diferente, es decir, a los elementos del grupo A les corresponde información estructuralmente distinta a los de los grupos B y C.

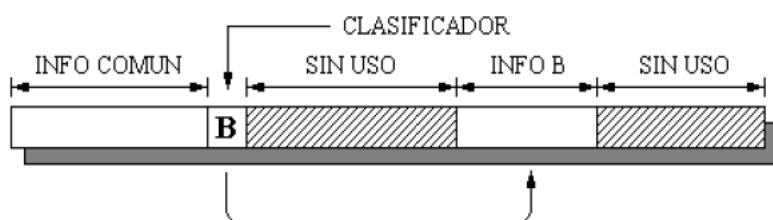
Llamemos **INFO A**, **INFO B** e **INFO C** a estas estructuras de información propias de cada grupo, e **INFO COMUN** a la información estructuralmente compartida por ellos.

Se podría construir una estructura de las siguientes características, y con ellas armar un vector de estructuras (*archivo en RAM*).



La función del clasificador es la de indicar cuál de los tres campos (**INFO A**, **INFO B** o **INFO C**) es el que contiene la información.

Una estructura como la anterior quedaría de la siguiente forma dentro del vector de estructuras:



Se aprecia que dos de los tres campos de información particular quedan sin uso, pero ocupando memoria. Esta situación se repite para las N estructuras que conforman el vector.

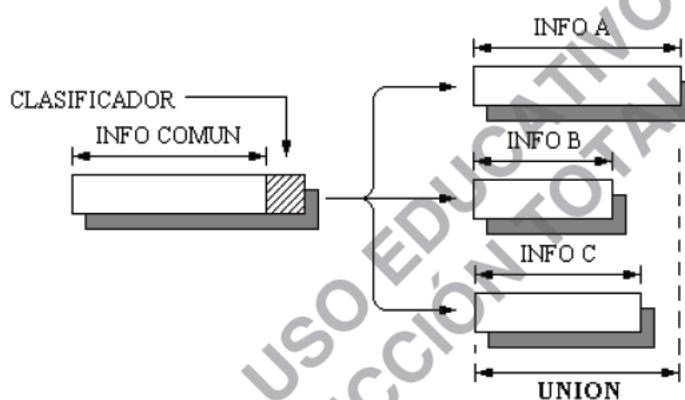
Por lo tanto, si bien de esta forma se resuelve el problema, se lo hace de una manera muy poco eficiente.

Una solución alternativa sería la de construir 3 archivos diferentes, uno para cada grupo de información. Pero esto no contempla la necesidad de un archivo único, como se solicitó inicialmente.

USO DE LA UNIÓN

Dado a que sólo uno de los 3 campos de información será utilizado, se puede construir una unión con ellos. Esta unión será a su vez un campo de la estructura general.

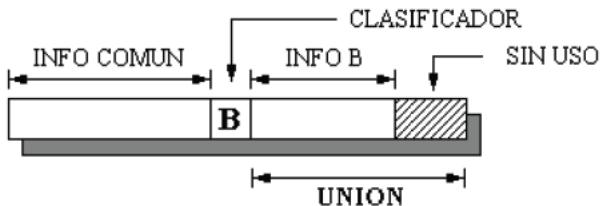
La situación descripta se muestra en la figura:



De esta forma, cualquiera sea el grupo al que pertenece el elemento cuya información se almacena, no se ocupará más memoria para sus datos particulares que la necesaria para la unión. Esto equivale al tamaño del mayor de sus campos.

Probablemente habrá un "desperdicio" de memoria no utilizada, pero el tamaño de ésta dependerá de la relación de tamaño de los campos de la unión y de su frecuencia de aparición.

De todos modos, el aprovechamiento de memoria es muy superior al caso en que no se empleó la unión, como se aprecia en el siguiente esquema.



Compárese esta disposición con la obtenida en el caso del uso de una estructura única.

EJEMPLO: USO DE LA UNION PARA AHORRO DE MEMORIA

Se presentará a continuación un ejemplo completo de manejo de uniones (cuyos campos son estructuras), contenidas a su vez en otras estructuras que integran un vector.

El ejemplo se enriquece con el uso de enumeraciones a fin de clarificar el programa. El programa se ha modularizado mediante múltiples funciones para aumentar su claridad.

Enunciado:

Se desea llevar el control de los elementos de una biblioteca que tiene 3000 ítems. Estos ítems pertenecen a 3 categorías: CDs, libros y revistas.

Estas categorías o grupos tienen datos que son comunes a todos los elementos y otros que son particulares de cada grupo.

Los datos comunes a todos ellos son :

- Código de elemento (int)
- Número de socio al que fue prestado ó 0 si no está prestado (int)

Los datos particulares de cada categoría son :

- CDs:
 - Nombre del CD (string de 20 caracteres)
 - Ubicación en la biblioteca (string de 10)
 - Soporte de ejecución [W(Windows), L(Linux), etc.] (char)
- Libros:
 - Nombre del libro (string de 30)
 - Nombre del autor (string de 20)
 - Nombre de la editorial (string de 20)
- Revistas:
 - Nombre de la revista (string de 20)
 - Volumen (int)
 - Número (int)

Se pide realizar un programa que permita el ingreso de los datos y luego los muestre en pantalla agrupados por categoría, utilizando funciones para cada caso.

```
#include <stdio.h>
#define N 3000
```

```

/* Declaración de tipos de datos */

struct CDS {
    char NOMBRE[20];
    char UBIC[10];
    char SOPORTE; } ;

struct LIBROS {
    char NOMBRE[30];
    char AUTOR[20];
    char EDITORIAL[20]; } ;

struct REVISTAS {
    char NOMBRE[20];
    int VOL;
    int NUM; } ;

union TODOS {
    struct CDS CD ;
    struct LIBROS LIBRO ;
    struct REVISTAS REVISTA ; } ;

enum TIPOS { CD , LIBRO , REVISTA } ;

struct DATOS {
    int CODIGO ;
    int PRESTADO ;
    enum TIPOS TIPO ;
    union TODOS ITEM ; } ;

/* Prototipos */
void INGRESO (struct DATOS [] , int ) ;
struct CDS LEECD ( void ) ;
struct LIBROS LEELIBRO ( void ) ;
struct REVISTAS LEEREVISTA ( void ) ;
void IMPRESION (struct DATOS [] , int ) ;
void PRINTCD ( struct CDS ) ;
void PRINTLIBRO ( struct LIBROS ) ;
void PRINTREVISTA ( struct REVISTAS ) ;

int main()
{
    struct DATOS DATO[N] ;
    INGRESO(DATO,N) ;
    IMPRESION(DATO,N) ;
}

```

Se comienza definiendo los tipos de datos “desde adentro hacia afuera”. Los primeros tipos definidos son las tres estructuras internas CDS, LIBROS y REVISTAS.

Nótese que los nombres de tipos se colocaron en plural, mientras que los nombres de campos equivalentes se colocaron en singular.

Posteriormente se define el tipo `union TODOS`, cuyos campos son las tres estructuras definidas anteriormente.

El siguiente paso es la definición de la enumeración `TIPOS`, cuyos componentes son las constantes `CD`, `LIBRO` y `REVISTA`. Estos nombres no interfieren con los nombres de campo utilizados en las definiciones anteriores.

Por último se define la estructura `DATOS`, que está compuesta por los campos generales `CODIGO` y `PRESTADO`, por el *clasificador* `TIPO` (obsérvese que es de tipo `enum TIPOS`) y por la unión `TODOS` cuyo nombre de campo es `ITEM`.

El programa principal es extremadamente simple, y se reduce a la declaración del vector de estructuras y a la invocación de las funciones `INGRESO()` e `IMPRESION()`.

```
void INGRESO (struct DATOS D[] , int CANT)
{
    int I ;
    for ( I=0 ; I<CANT ; I++ ) {
        printf ( "\nIngrese el codigo\n" ) ;
        scanf ( "%d" , &D[I].CODIGO ) ;
        printf ( "\nIngrese el tipo de dato\n" ) ;
        printf ( "0:CD    1:LIBRO   2:REVISTA\n" ) ;
        scanf ( "%d" , &D[I].TIPO ) ;

        D[I].PRESTADO = 0 ;

        switch ( D[I].TIPO ) {
            case CD : D[I].ITEM.CD      = LEECD ( );      break;
            case LIBRO : D[I].ITEM.LIBRO = LEELIBRO ( );    break;
            case REVISTA : D[I].ITEM.REVISTA = LEEREVISTA ( ); break;
        }
    }
}
```

La función `INGRESO()` lee los datos comunes a todos los elementos, como así también el identificador de grupo. Este último se selecciona de un menú, y se lee en la variable `D[I].TIPO`.

Con esta variable se ingresa en un *switch* y, según cual haya sido la elección, se llama a una de tres funciones: `LEEC(())`, `LEELIBRO()` o `LEEREVISTA()`.

Éstas son funciones dedicadas a ingresar las secciones particulares de cada elemento.

Obsérvese que en los `case` se utilizaron las constantes de enumeración, y que `D[I].TIPO` es a su vez una variable de tipo `enum TIPOS`.

```
/* LEECD                                     */
/* Permite el ingreso de los datos particulares de tipo CD  */
struct CDS LEECD ( )
{
    struct CDS C ;

    printf ( "\nIngrese el nombre\n" ) ;
    fflush ( stdin ) ;
    gets ( C.NOMBRE ) ;

    printf ( "\nIngrese la ubicacion\n" ) ;
    gets ( C.UBIC ) ;

    printf ( "\nIngrese el soporte\n" ) ;
    C.SOPORTE = getchar();

    return C ;
}

/* LEELIBRO                                    */
/* Permite el ingreso de los datos particulares de tipo LIBRO */
struct LIBROS LEELIBRO ( )
{
    struct LIBROS C ;

    printf ( "\nIngrese el nombre\n" ) ;
    fflush ( stdin ) ;
    gets ( C.NOMBRE ) ;

    printf ( "\nIngrese el nombre del autor\n" ) ;
    gets ( C.AUTOR ) ;

    printf ( "\nIngrese el nombre de la editorial\n" ) ;
    gets ( C.EDITORIAL ) ;

    return C ;
}

/* LEEREVISTA                                   */
/* Permite el ingreso de los datos particulares de tipo REVISTA */
struct REVISTAS LEEREVISTA ( )
{
    struct REVISTAS C ;

    printf ( "\nIngrese el nombre\n" ) ;
    fflush ( stdin ) ;
    gets ( C.NOMBRE ) ;
```

```

printf ( "\nIngrese el numero de volumen\n" ) ;
scanf ( "%d" , &C.VOL ) ;

printf ( "\nIngrese el numero de la revista\n" ) ;
scanf ( "%d" , &C.NUM ) ;

return C ;
}

```

Las tres funciones anteriores se encargan del ingreso de los datos particulares de cada grupo.

Estas funciones no reciben argumentos, generan la estructura correspondiente, cargan sus campos de datos y la retornan a la función invocante INGRESO(). Ésta a su vez toma este valor retornado para asignarlo al correspondiente campo de la unión.

```

void IMPRESION (struct DATOS D[ ] , int CANT)
{
    int I , J ;

    for ( J=CD ; J<=REVISTA ; J++ ) {
        printf ( "\n\nCATEGORIA = " ) ;

        switch ( J ) {
            case CD      : printf ( "CD\n" ) ;      break ;
            case LIBRO   : printf ( "LIBRO\n" ) ;    break ;
            case REVISTA : printf ( "REVISTA\n" ) ;  break ;
        }

        for ( I=0 ; I<CANT ; I++ )
            if ( D[I].TIPO == J ) {
                printf ( "\n%6d%6d" , D[I].CODIGO , D[I].PRESTADO ) ;

                switch ( D[I].TIPO ) {
                    case CD      : PRINTCD( D[I].ITEM.CD ) ;
                        break ;
                    case LIBRO   : PRINTLIBRO( D[I].ITEM.LIBRO ) ;
                        break ;
                    case REVISTA : PRINTREVISTA( D[I].ITEM.REVISTA ) ;
                        break ;
                } /* switch */
            } /* if */
        } /* for ( J ) */
    }
}

```

La función IMPRESION() realiza un doble barrido del vector de datos. El primero de ellos lo hace para determinar las diferentes categorías. Nótese el uso de las constantes de enumeración en el `for` de la variable J.

El segundo barrido cubre la totalidad del vector.

En definitiva, se barre el vector completo 3 veces a fin de realizar las impresiones selectivas por categorías.

Se invocan a las funciones PRINTCD(), PRINTLIBRO() y PRINTREVISTA() a fin de mostrar en pantalla los datos particulares de cada grupo. Estas funciones se muestran a continuación.

```
void PRINTCD ( struct CDS C )
{
    printf("%20s%15s%5c", C.NOMBRE, C.UBIC, C.SOPORTE);
}

void PRINTLIBRO ( struct LIBROS C )
{
    printf("%20s%20s%20s", C.NOMBRE, C.AUTOR, C.EDITORIAL);
}

void PRINTREVISTA ( struct REVISTAS C )
{
    printf("%30s%10d%10d", C.NOMBRE, C.VOL, C.NUM);
```

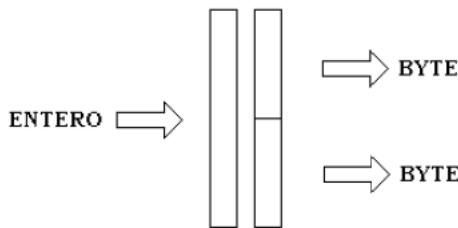
CONVERSIÓN DE FORMATO

En el caso anterior se aprovecha la situación en la que no se producirá superposición de información en los distintos campos de la unión.

Otra aplicación es la de aprovechar esa posible superposición para escribir en un campo y leer en otro campo superpuesto con el anterior.

Está claro que se obtendrá una o más fracciones del dato escrito. De esta manera se puede ingresar información en un formato y obtenerla en otro formato diferente del que se ingresó. Esta situación se aclarará mediante un ejemplo.

EJEMPLO: CAMBIO DE FORMATO

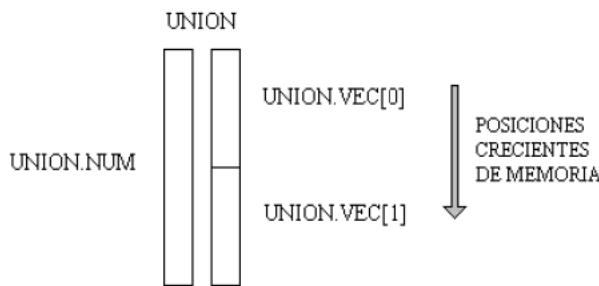


En este ejemplo se fraccionará un número entero (formado por dos bytes) en sus dos bytes consecutivos, con el objeto de inspeccionar la manera en que se almacena en la memoria.

Para ello se implementará una unión que permita el ingreso de una variable entera y la lectura de dos bytes alojados en forma consecutiva en la memoria, como se muestra en la figura.

Para lograr esto no se debe superponer un entero con dos `char`, dado que sólo se logrará superponer 3 veces el primer byte en memoria.

La solución es la superposición de un entero con un vector de dos caracteres. De esta manera si llamamos `UNION` a la variable de tipo unión, `NUM` al campo entero y `VEC` al vector de caracteres, tendremos la disposición en memoria que se muestra en este diagrama:



```
#include <stdio.h>
int main ( )
{
    union {
        unsigned char VEC[2] ;
        unsigned short int NUM ;
    } UNION ;

    UNION.NUM = 0X5789 ;
    printf("%4X%4X%4X" , UNION.NUM , UNION.VEC[0] , UNION.VEC[1] );
}
```

El resultado de la ejecución del programa es: 5789 89 57

Se comprueba que el byte menos significativo se almacena en posiciones inferiores de memoria. A este modo de almacenamiento se lo denomina *little endian*.

CAMPOS DE BIT

Los campos de bit son estructuras con campos enteros (int o char), cuyas longitudes en bits pueden ser seleccionadas en forma individual en un rango comprendido entre 1 y el máximo del tipo.

Cada uno de estos campos debe ser una variable de tipo entero (int o char).

Estos tipos de campos pueden estar afectados por el modificador `unsigned`.

Es necesario especificar la longitud, en bits, de cada campo en la declaración del tipo de dato.

Sintaxis de declaración:

```
struct nom_tipo {  
    tipo_entero nom_campo : longitud;  
    tipo_entero nom_campo : longitud;  
    . . . . .  
    tipo_entero nom_campo : longitud;  
} nombre_de_variable;
```

Ejemplo:

```
struct campo_bit {  
    char A : 3;  
    unsigned int B : 2;  
    int C : 4; } BITS;
```

Es evidente que cualquier variable en memoria debe ocupar un número entero de bytes. Es decir, una estructura de campos de bit ocupará una cantidad de bytes en memoria, de tal manera que los bits asignados igualen o superen la suma de las longitudes de cada uno de los campos.

Sorprende saber que la variable `BITS` así declarada ocupa 8 bytes en memoria, según se puede investigar mediante `sizeof` en el compilador GCC 4.9.

Dado que los campos de bit son estructuras, valen las consideraciones realizadas para éstas en cuanto a las variantes de declaración, siendo una de las más utilizadas la siguiente :

```

struct nom_tipo {
    tipo_entero nom_campo : longitud ;
    . . . . .
    tipo_entero nom_campo : longitud ;
} ;

. . . . .

struct nom_tipo nom_variable ;

```

Esto se debe a que raramente se utilizan las estructuras de campos de bit aisladas, sino que integran estructuras (o uniones) mayores, y por ello es necesario especificar un identificador de tipo de dato.

Por esta razón, es fundamental conocer claramente la ubicación relativa de cada uno de los campos en la memoria.

El primer paso es determinar la ubicación de los campos dentro de un byte.

Al declarar la estructura:

```

struct BITS {
    char A : 1 ;
    char B : 3 ;
    char C : 4 ;
} ;

```

¿Dónde se ubicará el campo A? ¿En el bit más significativo, en el menos significativo o en otro lado?

Se puede realizar la comprobación con un sencillo programa que utilice la superposición de esta estructura con una variable `char`, para ver el resultado de diversas asignaciones de los campos de bit. Para ello nos valdremos de una `union`.

EJEMPLO: UBICACIÓN DE LOS CAMPOS DE BIT

En el presente ejemplo se pretende investigar los valores obtenidos al asignar los campos de una estructura de campos de bit, con la finalidad de precisar la ubicación de dichos campos.

```

#include <stdio.h>

int main ()
{

```

```

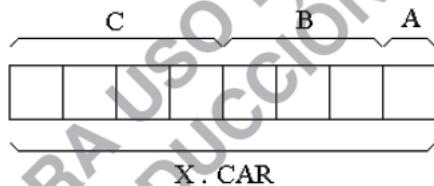
struct BITS {
    char A : 1 ;
    char B : 3 ;
    char C : 5 ; } ;

union {
    struct BITS NUM;
    unsigned char CAR; } X ;

X.NUM.A = 1 ;
X.NUM.B = 0 ;
X.NUM.C = 0 ;
printf ( "%X" , X.CAR ) ;
}

```

En este programa se asigna un 1 al campo A, y ceros a los otros campos (se pueden asignar diversas combinaciones de valores para completar la comprobación). Al observar el valor de la variable X.CAR se deduce que la asignación de campos se realizó como lo muestra la figura, siendo el de la izquierda el bit más significativo:



EJEMPLO: UBICACIÓN DE LOS CAMPOS EN DOS BYTES

Podemos extender la investigación para determinar la ubicación de los campos de bit en los casos en que estén involucrados más de un byte, y algún campo deba ser compartido por éstos.

En el siguiente ejemplo se utiliza una unión entre una estructura de campos de bit y un vector de dos caracteres. Como se sabe el elemento cero del vector se ubica en la posición de memoria anterior a la del elemento uno.

```

#include <stdio.h>
int main ()
{
    struct BITS {
        char A : 1 ;
        char B : 4 ;
        char C : 5 ;
        char D : 6 ; } ;

```

```

union {
    struct BITS NUM ;
    unsigned char CAR[3] ; } X ;

X.NUM.A = 1 ;
X.NUM.B = 1 ;
X.NUM.C = 1 ;
X.NUM.D = 1 ;
printf ( "%X %X %X" , X.CAR[0] , X.CAR[1] , X.CAR[2] ) ;
printf(" %d", sizeof(struct BITS));
}

```

El resultado de la ejecución arrojó 3 1 1 3, lo que quiere decir que el compilador está distribuyendo los 16 bits en 3 bytes. Se invita al lector a probar distintas combinaciones para deducir cómo se están distribuyendo los bits en memoria. Los campos A y B quedan en el primer byte, C en el segundo y D en el tercero. Insistimos en que este comportamiento corresponde al compilador GCC en su versión 4.9.

EJEMPLO: DETERMINACIÓN DE LA UBICACIÓN DE LOS BITS

Se plantea aquí un ejercicio para que resuelva el lector. Determinar cómo se distribuyen en memoria los campos de la struct BITS, sabiendo que el resultado de la ejecución del programa arroja 10272 en pantalla.

```

#include <stdio.h>
int main()
{
    struct BITS {
        char A : 6 ;
        char B : 5 ;
        char C : 2 ; } ;

    union {
        struct BITS NUM ;
        unsigned short VALOR ; } X ;

    X.VALOR = 0;
    X.NUM.A = 32 ;
    X.NUM.B = 8 ;
    X.NUM.C = 1 ;

    printf ( "%d" , X.VALOR); //muestra 10272
}

```

CAMPOS NO DENOMINADOS

Es necesario tener precaución con los campos innecesarios. Podría darse el caso en que no se necesite realizar ningún acceso a un campo intermedio.

En esta situación, no sería necesario asignarle un nombre al campo, quedando la declaración como se muestra a continuación:

```
struct BITS {  
    int A : 6 ;  
    int : 5 ;  
    char C : 2 ;  
    char D : 3 ; } ;
```

El campo innombrado se habría incluido para mantener la ubicación de los campos restantes.

PROBLEMA PROPUESTO

Mostrar en binario los campos signo, exponente y mantisa correspondientes a un número de punto flotante (float).

OPERADORES DE BIT

Hasta ahora las operaciones realizadas sobre valores los involucraban en su conjunto. Por ejemplo, en el caso de una suma de dos enteros, las operación de suma entre dos bits del mismo orden de cada uno de estos valores, no es independiente de las operaciones de suma de los bits menos significativos, ni deja de afectar a las operaciones de los más significativos.

Esto significa una interacción entre ellos llevada a cabo mediante los acarreos parciales. En conclusión, podemos considerar que la operación de suma (así como muchas otras) se realiza sobre la totalidad del (o de los) operando(s).

En esta sección veremos operaciones que se aplican en forma individual *bit a bit*, sobre uno o dos operandos.

Esto significa que si se tienen dos operandos A y B , sobre los que se aplica una operación α , se realizarán tantas operaciones independientes $A_i \alpha B_i$ como bits contengan los operandos.

INVERSIÓN (COMPLEMENTO A 1)

La operación de inversión bit a bit, llamada también “negación bit a bit” o “complemento a uno”, es una operación monaria (se aplica sobre un solo operando).

El operador que la realiza en C es la *virgulilla*: \sim

El efecto de la operación **complemento a uno**, o simplemente **complemento**, es el de invertir en forma individual, cada uno de los bits que forman el operando. Esto es, cada uno se transformará en cero y viceversa.

Se puede describir esta operación mediante su tabla de verdad, la cual describe totalmente el comportamiento de un operador lógico.

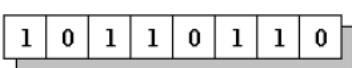
La tabla de verdad es una tabla en la que se muestran los estados de las variables de salida, en función de todas las variantes (combinaciones) posibles de las variables de entrada.

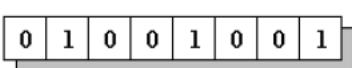
A	$\sim A$
0	1
1	0

En este caso, la tabla de verdad es sumamente sencilla dado que el operador complemento es monario.

Si bien la operación se realiza bit a bit, esto se aplica a todos los bits que conforman el dato o variable operando.

A continuación se muestra el efecto de la inversión sobre una variable de varios bits.

A A binary number consisting of 8 bits: 1, 0, 1, 1, 0, 1, 1, 0.

$\sim A$ The result of applying the complement operation to A. It is a binary number consisting of 8 bits: 0, 1, 0, 0, 1, 0, 0, 1.

En esta figura se aprecia el efecto de aplicar el complemento a una variable A.

Se observa que luego de la operación complemento los bits correspondientes quedaron invertidos.

EJEMPLO: RESTA EN COMPLEMENTO A 2

En este ejemplo se implementará una resta por el método del **complemento a 2**. Recordemos que, de acuerdo con este método, se le suma al *minuendo* el complemento a uno del *sustraendo*, más uno.

```
#include <stdio.h>

int main ( )
{
    int A , B , RESTA ;
    printf ( "Ingrese los valores de los operandos: " ) ;
    scanf ( "%d %d" , &A , &B ) ;
```

```

    RESTA = A + ~B + 1 ;      /* Suma el complemento a dos */
    printf ( "\n\n%d - %d = %d" , A , B , RESTA ) ;
}

```

DIFERENCIA CON EL INVERSOR LÓGICO

Es necesario aclarar la diferencia que existe entre el **inversor complemento** (\sim) y el **inversor lógico** (!). Este último invierte el estado lógico del operando en su totalidad.

Todo número en C tiene dos interpretaciones, una es la de su valor numérico y la otra es la de su valor lógico.

Con respecto a este último, se interpreta que todo número diferente de cero representa un estado lógico verdadero (representado por 1), mientras que si su valor numérico es cero, se interpreta que su significado lógico es falso (representado por 0).

De esta forma, tomando $A = B6_{16}$

$$\begin{array}{l} \sim A \rightarrow 49_{16} \\ !A \rightarrow 0 \\ !0 \rightarrow 1 \end{array}$$

OPERACIÓN OR

A continuación veremos tres operaciones lógicas binarias. A fin de interpretar mejor su significado, es necesario recordar que el 1 en realidad significa *verdadero*, mientras que el 0 tiene el significado de *falso*.

Estas operaciones son binarias (se aplican a dos operandos), y reciben frecuentemente la denominación de **funciones lógicas**.

La tabla de verdad que las represente tendrá entonces 4 valores correspondientes a las cuatro combinaciones posibles de sus entradas.

La primer operación lógica corresponde a la función OR, y se la representa con el símbolo: | (pipe o tubería), correspondiente al código ASCII 124.

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

En esta operación, para que el resultado sea verdadero es necesario que *alguno* de los dos operandos sea verdadero.

Esta situación se muestra en la tabla de verdad. En ella puede observarse que, para que la salida sea 1, es necesario que *alguna* de las variables A ó B sean 1, ó bien *ambas* lo sean.

La operación OR recibe también la denominación de **suma lógica**. Podemos decir entonces que, en términos lógicos:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

Surge además, de observar la tabla de verdad que :

$$A + 0 = A$$

$$A + 1 = 1$$

De aquí que podemos decir:

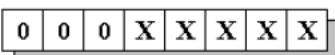
El cero es el valor neutro de la suma lógica (OR). Es decir que cero más un valor, repite ese valor; mientras que el 1 fuerza un 1 como resultado, independientemente del valor del otro operando.

APLICACIÓN DE SUMA LÓGICA

En muchas ocasiones de acceso al hardware, es necesario escribir determinados bits de algún registro, *sin modificar el resto de los bits*.

Debido a que la escritura del registro en cuestión se hace con la totalidad de su byte, será necesario conformar de alguna manera el byte a escribir, de forma tal que no modifique algunos bits, mientras que *setea* otros.

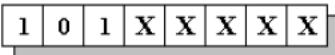
Simplifiquemos el problema diciendo que se desea escribir 101 en los 3 bits más significativos del registro identificado como A (bits que por simplicidad supondremos en cero), mientras que los restantes 5 bits deben permanecer intactos.

A 

Estos 5 bits que deben permanecer intactos tienen un contenido desconocido que se representa en la figura por medio de letras X.

B 

Se coloca en una variable auxiliar B el valor $A0_H$ como se muestra en la figura.

A | B 

Posteriormente se realiza la operación OR entre los dos valores.

Solamente resta almacenar este resultado en el registro A y queda resuelto el problema.

OPERACIÓN AND

La siguiente operación lógica corresponde a la función AND y se la representa con el símbolo: & (ampersand).

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

En esta operación, para que el resultado sea verdadero, es necesario que *ambos* operandos sean verdaderos.

Esta situación se muestra en la tabla de verdad. En ella puede observarse que para que la salida sea 1 es necesario que *las dos* variables A y B sean 1 de manera simultánea.

La operación OR recibe también la denominación de **producto lógico**.

Podemos decir entonces que en términos lógicos:

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

Surge además, de observar la tabla de verdad que:

$$A \cdot 0 = 0$$

$$A \cdot 1 = A$$

De aquí que podemos decir:

El uno es el valor neutro del producto lógico AND. Es decir que uno por un valor, repite ese valor; mientras que el 0 fuerza un 0 como resultado, independientemente del valor del otro operando.

APLICACIÓN DE PRODUCTO LÓGICO

En algunas ocasiones nos encontraremos con la necesidad de investigar o aislar determinados bits de un número o de un registro, de manera que el resto de los bits no interfiera.

La forma de lograr el objetivo anterior es **enmascarar** dicho valor. El proceso de enmascaramiento consiste en aplicar una **máscara** o filtro que permita “pasar” a los bits seleccionados, mientras que “detiene” (pone en cero) al resto de los bits.

Consideremos el siguiente ejemplo:

Supongamos que tenemos una variable llamada A que contiene un byte del que se desean conservar los bits 3, 5 y 6, mientras que se pretenden descartar los restantes.

A	<table border="1"><tr><td>X</td><td>Y</td><td>Y</td><td>X</td><td>Y</td><td>X</td><td>X</td><td>X</td></tr></table>	X	Y	Y	X	Y	X	X	X
X	Y	Y	X	Y	X	X	X		
M	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	0	1	0	0	0
0	1	1	0	1	0	0	0		
A & M	<table border="1"><tr><td>0</td><td>Y</td><td>Y</td><td>0</td><td>Y</td><td>0</td><td>0</td><td>0</td></tr></table>	0	Y	Y	0	Y	0	0	0
0	Y	Y	0	Y	0	0	0		

En la figura se representan los bits “descartables” como X, mientras que los que se desean conservar están representados por Y.

Se construye una máscara M con *unos* en las posiciones “deseables” y *ceros* en las “descartables”.

Posteriormente se realiza una operación AND entre ellas y se obtiene el resultado deseado, como podemos ver en la figura.

La fracción de código en lenguaje C que lo realiza se muestra a continuación:

```
M = 0x68 ;  
A = A & M ;
```

EJEMPLO: USO DE LA MÁSCARA

En este ejemplo se determinará la cantidad de unos que están presentes en una variable entera, utilizando una máscara para investigar cada uno de ellos.

La máscara llamada MASK se inicializa con el valor 1. Esto significa que su bit menos significativo está en uno, mientras que el resto está en cero.

Se realiza la operación AND con el operando a investigar (almacenado en la variable A) y si el resultado de esta operación es verdadero, significa que el bit a investigar vale uno, y por lo tanto se incrementa el contador CONT.

A continuación es necesario desplazar el uno de la máscara una posición hacia la izquierda, a fin de investigar el siguiente bit.

La forma de realizarlo, en este ejemplo, es la de multiplicar la máscara por 2.

Recuérdese que multiplicar un número por su base equivale a desplazarlo un lugar a la izquierda.

El conteo finaliza cuando la máscara vale cero. Esto se produce como resultado de un overflow en ella, debido a las sucesivas multiplicaciones por 2.

```
#include <stdio.h>  
  
int main ( )  
{  
    int A , MASK , CONT = 0 ;
```

```

printf ( "Ingrese el valor a investigar    " ) ;
scanf ( "%d" , &A ) ;
for ( MASK = 1 ; MASK ; MASK *= 2 )
    if ( A & MASK )
        CONT++ ;

printf ( "\n\nLa cantidad de unos es %d" , CONT ) ;
}

```

EJEMPLO: SETEO PARCIAL DE UN REGISTRO

Este es un problema frecuente cuando se accede a hardware. Supongamos que se necesita colocar los 3 bits menos significativos de un determinado puerto (en el ejemplo se seleccionó el port 0x200) en 101 sin afectar en modo alguno los bits restantes.

A diferencia de un ejemplo anterior, se desconoce el estado de los 3 bits menos significativos, es decir, no se puede suponerlos en cero.

El acceso al port se realiza en Linux utilizando las funciones `inb()` y `outb()`. Se advierte al lector que el siguiente ejemplo no funcionará sin el agregado de otras llamadas a funciones específicas del trabajo con puertos, además de necesitar ciertas condiciones de ejecución especiales. Sólo se muestra aquí el código para exemplificar el uso de máscaras ya que la utilización de puertos queda fuera del alcance de este capítulo.



```

#include <stdio.h>

#define PORT 0x200

int main ( )
{
    unsigned char A ;
    A = inb (PORT);          (1)
    A = A & 0XF8 ;           (2)
    A = A | 0X05 ;           (3)
    outb (A, PORT);
}

```

La figura muestra claramente la evolución del contenido de la variable `A` a lo largo del programa, como así también el formato de las máscaras, para poner en cero (2) y en uno (3) los bits que nos interesan.

OPERACIÓN XOR

El término **XOR** es la contracción de *exclusive OR*, mal llamada comúnmente “**OR exclusiva**”.

Es una operación **OR** en la que se *excluye* la combinación 11. Se trata entonces, más precisamente, de una operación **OR excluyente**.

Su símbolo de representación en C es el circunflejo (*caret* en inglés): ^

A	B	$A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

Se puede apreciar de su tabla de verdad que esta coincide con la de la operación **OR** excepto en la última combinación, en la que la salida vale cero.

Surge además, de observar la tabla de verdad que :

$$A \wedge 0 = A \quad A \wedge 1 = \sim A$$

APLICACIONES DE XOR

La operación **XOR** tiene diversas aplicaciones, entre ellas:

- Detección de variables de entrada diferentes.
- Detección de cantidad impar de unos en entradas consecutivas.
- Inversor controlado.

Esta última ha sido mostrada anteriormente. Si una de las variables es cero, la salida es igual a la otra variable, pero en caso de ser uno, la salida toma la forma de la otra variable complementada.

EJEMPLO: DETERMINACIÓN DE PARIDAD MEDIANTE XOR

Un caso habitual en transmisión de datos es acompañar a los bits de información con un bit adicional que establezca la **paridad** (o imparidad) en la *cantidad de unos* que se transmite, de manera que esta cantidad sea constante (siempre par o siempre impar).

Si en recepción no se concuerda con la paridad acordada, se supone que hubo al menos un error durante la transmisión.

Ya sea en la etapa de transmisión, como en la de recepción, será útil poder determinar la paridad en cantidad de unos de un determinado número.

Plantearemos ahora el problema de determinar si la cantidad de unos de un entero es par o impar, con las herramientas disponibles hasta el momento.

```

#include <stdio.h>

int main ( )
{
    unsigned int A , PARIDAD = 0 ;

    printf ( "Ingrese un entero: " ) ;
    scanf ( "%d" , &A ) ;

    while ( A ) {
        PARIDAD = PARIDAD ^ A ;
        A = A / 2 ;
    }

    if ( PARIDAD & 1 )
        printf ( "\nLa cantidad de unos es IMPAR. " ) ;
    else
        printf ( "\nLa cantidad de unos es PAR. " ) ;
}

```

El número a analizar se va desplazando hacia la derecha mediante sucesivas divisiones por 2. Se comprueba solamente el último bit.

DIFERENCIA CON LOS OPERADORES LÓGICOS

Los **operadores de bit** que corresponden a funciones lógicas que hemos visto hasta ahora (`&` y `|`) tienen una similitud gráfica con los **operadores relacionales lógicos** (`&&` y `||`).

Los operadores relacionales operan sobre variables y expresiones **booleanas**, es decir, resultados de proposiciones que se evalúan por *verdadero* o *falso*, y todos los valores en C son susceptibles de ser evaluados de esa manera.

De esta forma, y a modo de ejemplo, podemos mostrar que:

0x55	<code>&</code>	0xF0	\rightarrow	0X50
0x55	<code>&&</code>	0xF0	\rightarrow	1
0x55	<code> </code>	0xF0	\rightarrow	0XF5
0x55	<code> </code>	0xF0	\rightarrow	1

DESPLAZAMIENTOS

Las operaciones de desplazamiento implican el *corrimiento* hacia la derecha o hacia la izquierda de la totalidad de los bits de un número, una cantidad *N* de lugares.

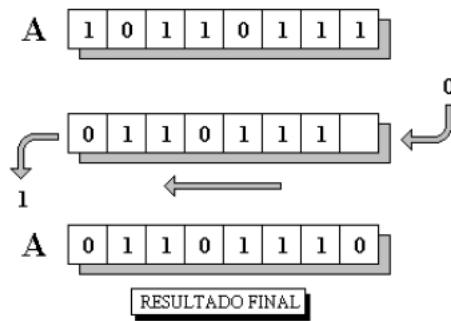
Los operadores de desplazamiento en C son: << y >> (formados con dos signos)
La sintaxis de utilización es:

A = A << N; A = A >> N;

Esto significa que el contenido de la variable A sufrirá un desplazamiento a la izquierda, en el primer caso, o a la derecha en el segundo, una cantidad N de lugares.

DESPLAZAMIENTO HACIA LA IZQUIERDA

Recordemos que el desplazamiento hacia la izquierda está relacionado con la operación de multiplicación por la base de numeración del dato.



Como se puede apreciar en la figura, la operación:

A = A << 1;

Produce un corrimiento hacia la izquierda de todos los bits. El bit vacante se completa con cero, mientras que el bit extremo izquierdo se pierde.

EJEMPLO: MULTIPLICACIÓN

En este ejemplo se muestra cómo realizar una multiplicación rápida de dos números positivos en binario, sin utilizar en el programa los operadores de multiplicación (*) ni resta (-).

Se utiliza el algoritmo de multiplicación más habitual, que consiste en *sucesivos desplazamientos* del dividendo a medida que lo requiere el divisor.

No se puede utilizar la definición de multiplicador, equivalente a sumar el multiplicando “multiplicador veces”, debido a la imposición del enunciado de no utilizar restas ni decrementos (que serían restas encubiertas).

```
#include <stdio.h>

int main ( )
{
    int MULTIPLICANDO , MULTIPLICADOR , ACUM = 0 ;
    printf ( "Ingrese dos valores positivos a multiplicar: " ) ;
```

```

scanf ( "%d %d" , &MULTIPLICANDO , &MULTIPLICADOR ) ;
printf ( "\n\n%d x %d = " , MULTIPLICANDO , MULTIPLICADOR ) ;

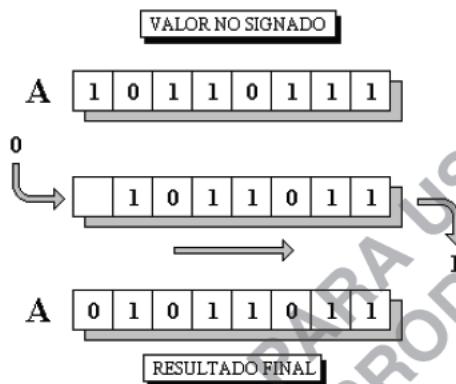
while ( MULTIPLICADOR ) {
    if ( MULTIPLICADOR & 1 )
        ACUM += MULTIPLICANDO ;
    MULTIPLICANDO <<= 1 ;
    MULTIPLICADOR >>= 1 ;
}

printf ( "%d" , ACUM ) ;
}

```

DESPLAZAMIENTO HACIA LA DERECHA

En el caso de desplazamiento hacia la derecha es necesario diferenciar entre el caso en que el valor a ser desplazado es un número signado, de cuando no lo es.



En el caso de ser un número **no signado**, el procedimiento es similar al del caso de desplazamiento a la izquierda.

La posición vacante se completa con cero.

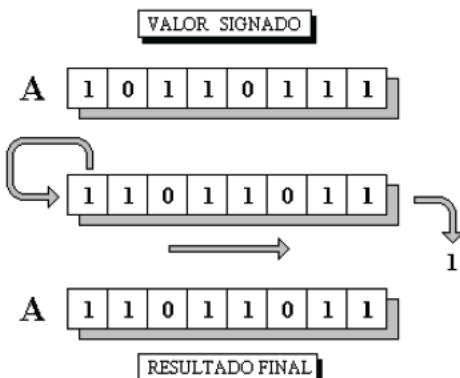
Obsérvese que esto equivale a dividir al número por 2^N donde N representa la cantidad de lugares a desplazar.

Los bits “perdidos” en esta operación representan el resto.

La situación es diferente en el caso de un número entero **signado**. Dado que este desplazamiento equivale a una multiplicación, es necesario mantener el signo original del número.

En el caso en que el valor a desplazar sea signado, la diferencia se encuentra en que en la posición vacante se reinserta el mismo bit que se encontraba allí.

De esta manera se asegura que no se altere el signo del número total, debido al desplazamiento.



Es necesario aclarar que en el caso de los números negativos, cuando existe resto en esta operación, el resultado (entero), estará ajustado al valor menor (o sea al más negativo).

Esto puede comprobarse fácilmente con el valor (-1) equivalente a FF_H. Al desplazarlo en estas condiciones hacia la derecha un lugar, el resultado sigue siendo FFH ,es decir (-1), y no “0” debido al resultado esperado en la división de (-0.5) .

EJEMPLO: DESPLAZAMIENTO A DERECHA

Se planteará un ejemplo que permita comprobar lo aseverado anteriormente en el caso del desplazamiento a la derecha, como así también el ajuste del redondeo en el caso de los números negativos.

A fin de facilitar la visualización se muestran los valores tanto en decimal como en hexadecimal.

```
#include <stdio.h>
int main ()
{
    unsigned char NOSIGNADO ;
    char      SIGNADO      ;

    NOSIGNADO = SIGNADO = 0X89 ;
    printf ( "Numeros originales \n%10X%10X", NOSIGNADO, SIGNADO ) ;
    printf ( "\n%10d%10d" , NOSIGNADO , SIGNADO ) ;

    NOSIGNADO = NOSIGNADO >> 2 ;
    SIGNADO   = SIGNADO   >> 2 ;

    printf ( "\n\nResultado final \n%10X%10X", NOSIGNADO, SIGNADO ) ;
    printf ( "\n%10d%10d" , NOSIGNADO , SIGNADO ) ;
}
```

Se plantearán a continuación dos ejemplos que probablemente ya hayan sido resueltos por el lector de alguna manera más laboriosa o ingeniosa, pero que nos permiten enriquecer y mostrar el manejo de las herramientas recientemente estudiadas y, en todo caso, seleccionar el método más conveniente para futuras aplicaciones.

EJEMPLO: CONVERSIÓN DE UN NÚMERO DECIMAL A BINARIO

Se desea ingresar por teclado un número decimal y mostrarlo en pantalla en binario. Se considera por simplicidad que el número es un entero de 16 bits.

El algoritmo utilizado es el de desplazar 16 veces el número hacia la izquierda y verificar el estado del bit mas significativo mediante la máscara 0x8000.

Obsérvese la utilización reducida del desplazamiento NUMDEC <= 1 ; que es equivalente a NUMDEC = NUMDEC << 1 ;

```
#include <stdio.h>
int main ( )
{
    unsigned short int NUMDEC , I ;

    printf ( "Ingrese un numero decimal : " ) ;
    scanf ( "%d" , &NUMDEC ) ;
    printf ( " \n\nNumero binario : " ) ;

    for ( I = 0 ; I<16 ; I++ ) {
        if ( NUMDEC & 0X8000 )
            putchar ( '1' ) ;
        else
            putchar ( '0' ) ;

        NUMDEC <<= 1 ;
    }
}
```

EJEMPLO: CONVERSIÓN DE UN NÚMERO BINARIO A DECIMAL

Se desea ingresar por teclado un número binario sin signo y mostrar su valor decimal en pantalla. Por simplicidad se supone que no se producirá desborde de dicho número.

```
#include <stdio.h>
int main ()
{
    unsigned int NUMDEC = 0 ;
    unsigned char BIN ;

    printf ( "Ingrese un numero binario : " ) ;
    BIN = getchar ( ) ;
    while ( (BIN == '1') || (BIN == '0') ) {
        NUMDEC <<=1 ;
        NUMDEC += BIN - '0' ;
        BIN = getchar ( ) ;
    }

    printf ( " \n\nNumero decimal : %d" , NUMDEC ) ;
}
```

PROBLEMAS PROPUESTOS

1. Realizar un programa que reciba dos números enteros por teclado y muestre su diferencia sin utilizar el operador binario de resta.
2. En C no se dispone del operador **XOR** a nivel lógico. Construya una expresión lógica que reemplace a:
`(COND1 ^^ COND2)`
3. En el formato de punto flotante el bit más significativo es el signo y los siguientes 8 representan el exponente. Construya las funciones:

`float incexp(float) float decexp(float)`

que incrementen y decrementen el exponente de un flotante respectivamente. Construya un programa que permita cotejar el resultado.

4. Se desean realizar lecturas de un dispositivo conectado al port 400_{16} . Se utilizará para ello la función `inb(0x400)` que retorna el byte transmitido.
Este byte está conformado de la siguiente manera :

b_7	Validación (1. Dato válido 0. Dato inválido)
$b_6 - b_4$	Dato 1
$b_3 - b_0$	Dato 2

Se realizarán 1000 mediciones válidas (se deben descartar las mediciones con $b_7 = 0$) a intervalos de 1 segundo, tras lo cual se mostrará el promedio de cada uno de los datos que las conforman.

Considerar estos datos como no signados.

- a. Resolver el problema utilizando uniones y campos de bit.
- b. Resolver el problema utilizando operadores de bit.

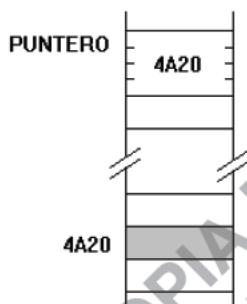
9. PUNTEROS

Los **punteros** constituyen una de las herramientas mas versátiles y poderosas del lenguaje de programación C. Es necesario que el programador tenga una comprensión acabada de su manejo.

Debe tenerse en cuenta que, a través de los punteros, el programador posee un gran flexibilidad para la implementación de algoritmos, pero también recae en él una gran responsabilidad, debido a que un manejo imprudente puede derivar en resultados catastróficos.

Un puntero es una variable destinada a contener una dirección de memoria.

Esta posición de memoria frecuentemente corresponde a la ubicación de otra variable, se dice entonces que el puntero “apunta” a la otra variable.



En este esquema suponemos que la variable PUNTERO contiene el número $4A20_H$. Decimos, entonces, que la dirección $4A20_H$ (y lo que hubiera en ella) *es apuntada* por el puntero.

TAMAÑO DE LOS PUNTEROS

El tamaño que ocupan los punteros en memoria depende de la **arquitectura** subyacente del microprocesador y del sistema operativo. En una máquina de 32 bits los punteros ocuparán 4 bytes, ya que deben tener el tamaño suficiente para almacenar direcciones de 32 bits.

Con otras palabras, el tamaño del puntero no depende del *dato apuntado* por él, sino de la arquitectura de la computadora. Para el caso de la figura, no importa qué dato tenemos en la dirección $4A20_H$. Tanto si fuera un char, el comienzo de un int o el comienzo de una struct muy grande, en cualquier caso el puntero tendrá la cantidad de bytes necesarios para contener una dirección de memoria según la arquitectura de la máquina en donde se compile.

Otra forma de verlo es que, para una misma computadora, todos los punteros ocupan el mismo tamaño, independientemente de a qué apunten.

USOS DE LOS PUNTEROS

A continuación se enumeran algunos usos frecuentes de los punteros y las ventajas que proporcionan:

- Permiten el acceso a cualquier posición de la memoria, para ser leída o escrita (en los casos en que esto sea posible).
- Permiten la transferencia de argumentos a las funciones **por referencia**. Hasta este punto hemos visto la transferencia de argumentos que las funciones tomaban en sus parámetros formales, **por valor**. En este caso, la función *copia* el valor transferido en una variable local (el parámetro formal) duplicando la memoria utilizada. Sólo vimos transferencia por referencia en el caso de los vectores; justamente porque lo que se estaba transfiriendo a la función era el nombre (del vector) que, como vimos, contiene la *dirección* de su inicio, es decir es un *puntero* (pero constante).
- Soportan el uso de la **asignación dinámica de memoria**. Cuando es necesario solicitar memoria que no fue reservada al inicio del programa, se utilizan rutinas de asignación dinámica. Estas rutinas nos informan del lugar de la memoria otorgada a través de un puntero, como veremos en un capítulo posterior.
- Son el soporte para la implementación de los enlaces que utilizan algunas **estructuras avanzadas de datos**, tales como las **listas enlazadas**, doblemente enlazadas, **pilas, colas y árboles**.
- Operan más eficientemente en los arrays que el modo de operación mediante subíndices.

DECLARACIÓN DE TIPO PUNTERO

La sintaxis de declaración de un puntero es:

```
tipo * nom_puntero ;
```

El nombre del puntero es el identificador de la variable, el “*” (asterisco) es el **operador de indirección** que nos indica que la variable es de tipo puntero. El tipo es el tipo de variable *apuntada por el puntero*, denominado **tipo base**.

El tipo base es sumamente importante pues tiene incidencia en el comportamiento del puntero frente a la aritmética de punteros.

Dos punteros que apuntan a tipos base diferentes deberían ser considerados de tipos diferentes aunque los dos, como sabemos, contienen direcciones de memoria. Inclusive en el caso de asignación de un puntero a otro puede ser necesario realizar un casting a fin de permitir la transferencia del dato.

Ejemplo de declaración:

```
float x , *p ;
```

En esta declaración múltiple se está indicando que `x` es una variable `float` y `p` es un puntero a `float`.

OPERADORES PUNTERO

Son dos: `&` y `*`. Ambos son operadores monarios, es decir que actúan sobre un solo operando.

El operador `&` es un operador monario que devuelve la dirección de memoria de su operando. Puede considerarse que equivale a “dirección de...”.

A través de él podemos relacionar a los punteros con las variables a ser apuntadas por ellos.

Ejemplo:

```
int A, *p ;  
p = &A ;
```

A partir de esta última sentencia, `p` se carga con la *dirección de A*, es decir: “*p apunta a A*”.

Cuando se declara un puntero, éste contiene un valor desconocido (basura electrónica) y por lo tanto “no apunta a nada”. Es lo que se llama un **puntero descontrolado**. Es necesario, por lo tanto, inicializarlo con el valor adecuado.

Omitir la inicialización es uno de los errores mas frecuentes que se producen con el uso de los punteros.

Cuando se declara una variable y el puntero que la ha de apuntar, se realiza el proceso mental de relacionarlas, lo que no ocurrirá hasta que se lo realice explícitamente a través de una asignación con la dirección de la variable.

El operador `*` es relativamente complementario del anterior. Retorna el valor de la variable cuya dirección es la que sigue. Podemos interpretarlo como: “el contenido de lo apuntado por...”.

Ejemplo:

```
int A, *p ;  
p = &A ;
```

A partir de este momento, `*p` es el contenido de lo apuntado por `p`, como `p` apunta a `A`, será entonces el contenido de `A`, o sea que:

`*p` es equivalente a `A`

Recordemos que en la declaración `int A, *p` se está expresando que la variable `A` corresponde a un entero y la variable `*p` también corresponderá a entero.

ASIGNACIÓN DE PUNTEROS

Un puntero puede ser asignado de 3 maneras:

- A través de otro puntero.
- Con la dirección de una variable.
- Directamente con la dirección que deba contener.

La asignación de un puntero a través de otro puntero, se realiza en forma similar a la de cualquier otro tipo de variable, copiándola.

Ejemplo:

```
int *p , *q , A ;  
q = &A ;  
p = q ;
```

Luego de esta última sentencia, los dos punteros apuntan a `A`, y por lo tanto `*p` es equivalente a `*q`.

El caso de asignación con la dirección de una variable ya fue mencionado. Se utiliza al operador `&`.

La asignación directa de una dirección se realiza entregándole un valor entero. Lo habitual es expresar las direcciones de memoria en hexadecimal.

Por ejemplo: `p = 0x0B75 ;`

Con lo cual `p` contendrá la dirección de 16 bits `0B75H`.

CASTEO EN LA ASIGNACIÓN DE PUNTEROS

Sin embargo, si bien la asignación anterior está clara y no presenta dudas, ambos tipos en la asignación son diferentes.

Mientras que `p` es un *puntero a entero*, `0x0B75` es un *entero*. Este *desapareamiento de tipos* puede dar lugar a un *warning* (advertencia) o a un error, en el momento de compilación, dependiendo del compilador actuante y cómo esté configurado.

Esta situación se resuelve mediante un casting en el momento de la asignación, como se muestra a continuación :

```
puntero = ( tipo_base * ) valor entero ;
```

Ejemplo:

```
int *p ;  
P = (int *) 0x0B75 ;
```

COMPARACIÓN DE PUNTEROS

Los punteros soportan todos los **operadores relacionales** que actúan sobre las otras variables: `==` `!=` `<` `<=` `>` `>=`

Los punteros pueden ser comparados de tres formas posibles:

- Con otros punteros.
- Con direcciones de variables expresadas mediante el operador `&`.
- Con direcciones dadas directamente.

Ejemplo: Sean `p` y `q` punteros a entero, y `A` una variable entera.

- `p < q` es una condición booleana en la que la dirección apuntada por `p` tiene menor valor numérico que la apuntada por `q`.
- `p == &A` es una condición booleana en la que `p` apunta a `A`.
- `p != 0x0020` condición booleana en la que la dirección contenida por `p` es diferente de `20H`.

ARITMÉTICA DE PUNTEROS

SUMA Y RESTA DE ENTEROS

A un puntero puede sumársele o restársele un entero. En el caso particular que este entero sea la unidad, tendremos las operaciones de incremento y decremento.

Ejemplo:

```
p = p + 3 ;  
p = p - 3 ;  
p++ ;  
p-- ;
```

Tiene especial importancia en estos casos el **tipo base** apuntado.

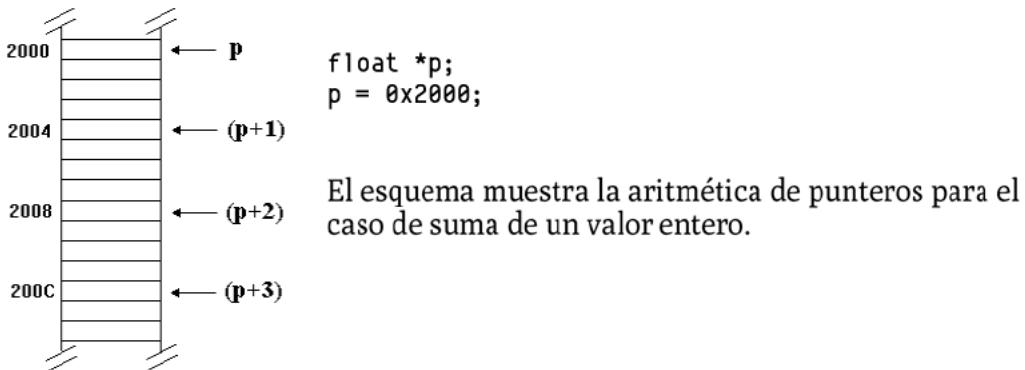
La unidad de incremento o decremento de punteros no es el byte, sino la cantidad de bytes determinados por el tamaño del tipo base.

La característica enunciada permite el correcto desempeño de los punteros en el manejo de vectores, como se verá más adelante.

Ejemplo:

```
float *p = 0X2000 ;  
p++ ;           // p contendrá 2004H;  
p = p + 2 ;     // p contendrá 200CH;
```

Ocurre de este modo porque una variable float ocupa 4 bytes en memoria.



DIFERENCIA DE PUNTEROS

Es posible realizar la diferencia de punteros siempre y cuando dicha diferencia sea asignada a un entero.

El significado del resultado es la *distancia* entre las direcciones apuntadas por los dos punteros, expresadas en *cantidad de tipos base*.

La distancia obtenida por diferencia de punteros no está expresada en bytes, sino en cantidades enteras de grupos de bytes, equivalentes al tamaño del tipo base apuntado por los punteros. Éstos deberán tener el mismo tipo base a fin de evitar errores.

EJEMPLO: DIFERENCIA DE PUNTEROS

Se declaran dos punteros a flotante, se los asigna y se muestra en pantalla la diferencia entre ellos a fin de constatar el funcionamiento de la aritmética de punteros.

```
#include <stdio.h>
int main()
{
    float *p , *q ;
    int A ;
    p = (float *) 0x2000 ;
    q = (float *) 0x200A ;
    A = q - p ;
    printf("\n\n %d  %p  %p", A , q , p );
}
```

La ejecución de este programa dio como resultado los valores 2000_H para p y $200A_H$ para q , como era de esperar. El valor final de A fue 2.

Esto surge de contar la cantidad entera de floats que separan las direcciones apuntadas por p y q . Con otras palabras, entre $2C00_H$ y $200A_H$ entran 2 float.

El carácter de formato de impresión %p corresponde a la impresión del contenido de un puntero, en formato hexadecimal.

Es un formato muy útil ya que, como vimos, distintas computadoras tienen punteros de distinta cantidad de bytes. Esto hace a un programa portable.

OPERACIONES NO PERMITIDAS

Las siguientes operaciones no pueden realizarse con variables tipo puntero:

- Suma de punteros
- Multiplicación de punteros
- División de punteros

- Suma a un puntero de un float o un double
- Operaciones de desplazamiento sobre punteros
- Enmascaramiento de punteros (operaciones lógicas)

TIPOS DE DATOS APUNTADOS POR PUNTEROS

Los **tipos base** apuntados por punteros pueden ser todos los provistos de manera standard por el lenguaje C: `int`, `char`, `long`, `float` y `double`.

También puede apuntar a un *tipo no especificado* de dato que se indica con `void`, o inclusive a otro puntero.

Los punteros también pueden apuntar a tipos de datos creados por el usuario como cadenas, estructuras y uniones.

No pueden apuntar a campos de bit pues no pueden contener direcciones no enteras.

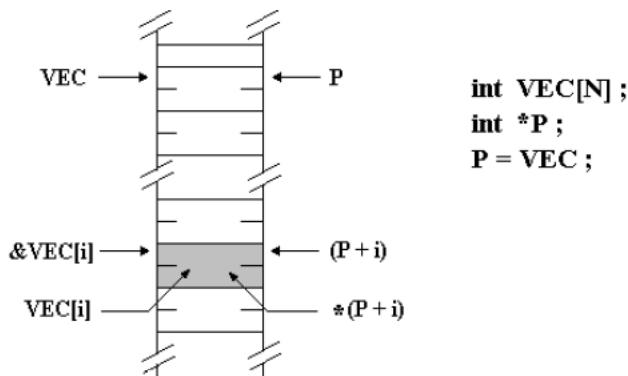
RELACIÓN ENTRE PUNTEROS Y VECTORES

Existe una estrecha relación entre punteros y vectores.

Veremos a continuación el posible manejo de un vector mediante la tradicional aritmética de subíndices y también a través de un puntero, comparando ambos casos.

Declaremos un vector de enteros y también un puntero a entero, haciendo que apunte al inicio del vector.

Esta situación se ilustra en la siguiente figura:



En las condiciones establecidas `VEC` contiene la dirección de inicio del vector, por lo que:

`VEC` es equivalente a `&VEC[0]`

Pero como P fue asignado con la dirección contenida por VEC , resulta que P también apunta a $\text{VEC}[0]$, es decir:

VEC es equivalente a P

Cuando mencionamos $\text{VEC}[i]$ estamos haciendo referencia a la variable entera que ocupa el i -ésimo lugar en el vector.

$(P+i)$ apunta, según la aritmética de punteros, al elemento $\text{VEC}[i]$ mencionado precedentemente, y por lo tanto:

$P + i$ es equivalente a $\&\text{VEC}[i]$

Además:

$*(P+i)$ es equivalente a $\text{VEC}[i]$

Lo mismo ocurriría si incrementáramos i veces el puntero P , y luego hacemos mención a $*P$.

Esto sugiere que se pueden utilizar punteros directamente para manejar vectores, en lugar de la tradicional utilización de las variables subindexadas con corchetes. De hecho, la utilización de punteros con este propósito es más eficiente.

EJEMPLO: MANEJO DE UN VECTOR MEDIANTE PUNTEROS

Se ingresará una lista de 30 valores enteros. Se almacenarán en un vector y luego se mostrará el valor máximo y su posición dentro del vector.

A fin de enriquecer el ejemplo, se presentan dos maneras de utilizar el puntero p , aún a costa de declarar variables innecesarias. Queda a cargo del lector, a modo de ejercicio, la optimización de este programa.

En el ingreso de datos se utiliza un desplazamiento del puntero p , mientras que en la búsqueda del máximo el puntero permanece constante y se utiliza la variable de barrido i .

No se almacena el máximo valor, sino que lo que se mantiene es un puntero al máximo.

```
#include <stdio.h>
#define N 30

int main()
{
    int VEC[N] , *p , *max , i ;
```

```

/*      Ingreso de datos      */
for( p = VEC ; p < VEC+N ; p++ )
    scanf("%d" , p);

/*  Calculo del maximo   */
for ( max = p = VEC , i = 0 ; i < N ; i++ )
    if( *(p+i) > *max )
        max = p+i ;

/*  Impresion de resultados */
printf("\n\n    Max = %d" , *max );
printf(" \n    Pos = %d" , max-VEC );

}

```

Obsérvese que en la lectura mediante `scanf` se omitió el `&` correspondiente a la variable. Esto es porque `scanf` requiere la dirección donde se debe almacenar el dato ingresado, y ésta es proporcionada directamente por el puntero.

PUNTEROS Y SUBÍNDICES

La utilización de subíndices con los punteros tiene el mismo significado que al aplicárselos al nombre de un vector.

Por ejemplo, al expresar:

```

int VEC[N] , *P ;
P = VEC ;

```

Resultan términos equivalentes para referirse al elemento i -ésimo del vector:

VEC[i]	*(VEC+i)	*(P+i)	P[i]
--------	----------	--------	------

EJEMPLO 3: PUNTEROS SUBINDICADOS

Se presenta como ejemplo un simple ingreso e impresión de datos almacenados en un vector.

El ingreso de datos se realiza de manera análoga a la del ejemplo anterior, mientras que la impresión utiliza al puntero subindicado, de la manera que habitualmente se utiliza con el nombre del vector.

```

#include <stdio.h>
#define N 10

```

```
int main()
{
    int VEC[N] , *p , i ;
    /*      Ingreso de datos      */
    for( p = VEC ; p < VEC+N ; p++ )
        scanf("%d" , p ) ;

    /*      Impresion de datos   */
    for ( p = VEC , i = 0 ; i < N ; i++ )
        printf("\n    Valor = %d" , p[i] ) ;
}
```

PUNTEROS A STRING

Puesto que una *string* es una cadena (**vector**) de caracteres, es válido pensar que en este caso son aplicables también los conceptos expuestos anteriormente para vectores.

La forma de declarar un puntero a string es idéntica a la de declarar un puntero a carácter.

Sintaxis:

```
char * nom_string ;
```

donde *nom_string* es el identificador del puntero a carácter.

Ejemplo:

```
char * frase = "Primera frase" ;
printf ("%s", frase );
```

Esto tiene como resultado la impresión de “*Primera frase*” en la pantalla.

Debe prestarse especial atención a que el solo hecho de declarar el puntero a carácter *no reserva memoria* alguna para la string. En el ejemplo, la reserva de memoria ocurrió al haber asignado la cadena de caracteres en el mismo momento de la declaración.

De otra forma, habrá que reservar memoria primero y luego apuntar el puntero al inicio del string.

Ejemplo:

```
char * frase ;
scanf ("%s", frase );      /* Incorrecto!!! */
```

Esto es incorrecto, dado a que la cadena de caracteres leída desde teclado se almacenará a partir de la dirección de memoria apuntada por `frase`, y éste es un **puntero descontrolado ya que no fue inicializado**.

Lo correcto es producir una reserva de memoria de algún modo, y luego apuntar el puntero `frase` a ella:

```
char palabra[10] , * frase ;
frase = palabra ;
scanf ("%s", frase ) ; /* Correcto */
printf ("%s", frase ) ;
```

Esto último podría haberse realizado sin utilizar el puntero, y por lo tanto no se observa ventaja alguna. Sin embargo, podemos apreciar la utilización de los punteros en un caso donde la asignación del vector al puntero está implícita, como en una transferencia de argumentos a función.

```
char palabra[10] ;
leer (palabra) ;
imprimir (palabra) ;
.....
void leer ( char * frase )
{ scanf ("%s" , frase ) ; }
.....
void imprimir ( char * frase )
{ printf ("%s" , frase ) ; }
```

Las **funciones de string**, reciben un puntero al inicio de la string. La memoria ya fue reservada anteriormente. La longitud del string depende de la ubicación del carácter nulo de terminación ('\0').

De esta manera el string se transfiere **por referencia** (es decir que se transfiere **por valor** una dirección de memoria, que se copia del puntero).

EJEMPLO: LONGITUD DE UN STRING I

Función que calcula y devuelve la longitud de una string.

```
int longitud ( char * palabra )
{
    int i = 0 ;
    while ( *palabra ++ ) i++ ;
    return i ;
}
```

Recordar que el carácter nulo tiene el significado lógico *falso*.

EJEMPLO 5: LONGITUD DE UN STRING II

Se realiza un programa que cumple la misma tarea que la función anterior, utilizando algunas variantes con fines didácticos.

Recordemos que la asignación directa de un string, en el momento de la declaración, reserva memoria y agrega automáticamente el carácter nulo. Por esta razón podemos asignar dicho string a un vector sin tamaño, como se muestra en el ejemplo.

```
#include <stdio.h>
int main ()
{
    char str[] = "Hola Mundo" ;
    char *p = str ;
    while ( *p++ ) ;
    printf ("La longitud es %d", p - str );
}
```

PUNTEROS A ESTRUCTURA

Siendo la estructura un tipo de dato (definido por el usuario), puede ser apuntado por un puntero.

Declaración:

```
struct tipo_struct * nom_punt ;
```

Donde `tipo_struct` es un tipo de dato estructura definido previamente, y `nom_punt` es el identificador del puntero.

Puede considerarse que `struct tipo_struct *` es el tipo de la variable (que se lee: *puntero a estructura tipo_struct*), y `nom_punt` es una variable de ese tipo.

Ejemplo:

```
struct fecha {
    int dia ;
    int mes ;
    int anio ; }; /* Estructura de tipo fecha */
struct fecha hoy ; /* hoy es una variable estructura fecha */
struct fecha * P ; /* P es puntero a estructura fecha */
```

Esta sección de programa crea un **tipo de dato** estructura con el nombre de **fecha**. Luego declara una variable de tipo **struct fecha** (llamada **hoy**), y finalmente un puntero a **struct fecha** llamado **P**.

Cabe aclarar que **P** no apunta a **hoy**, sino que aún es un **puntero descontrolado**. Para hacer que **P** apunte a **hoy** será necesario cargarlo con la dirección de la estructura:

```
P = &hoy ;
```

Obsérvese la presencia del operador **&**.

Es necesario no confundir una estructura con un vector. En este último, el nombre representa la **dirección de inicio** del vector y, por lo tanto, puede asignarse directamente a un puntero. En el caso de la estructura, el nombre es el identificador de la variable y representa el **valor** de la variable (no a su dirección). No puede asignarse una estructura directamente a un puntero, sino que es necesario el operador **&** para indicar su dirección.

ACCESO A LOS CAMPOS DE LA ESTRUCTURA

Utilizando el ejemplo anterior, podemos decir que ***P** representa una variable de tipo **struct fecha**, y como **P** apunta a **hoy**, podemos afirmar que :

***P** es equivalente a **hoy**

Como fue visto anteriormente, la referencia al campo **mes** de la estructura **hoy**, se expresa como:

hoy.mes

Pero ***P** equivale a **hoy**, por lo tanto podríamos expresar el campo **mes** de la estructura **hoy**, referenciándola a través del puntero **P**, de la siguiente forma:

(*P).mes

El paréntesis es necesario pues el operador miembro tiene **precedencia** de operación frente al operador de indirección *****, y si no fuera utilizado, el significado de la expresión sería "*lo apuntado por P.mes*". Esto no tiene sentido porque **mes** no es un puntero, y por otro lado provocaría un error en tiempo de compilación dado que al no ser **P** una variable estructura que tenga un campo llamado **mes**, la expresión **P.mes** no será reconocida.

El lenguaje C ofrece otra manera más simple y sintética de referenciar campos de estructuras a través de punteros, y es utilizando el símbolo **->**, formado por un **-** seguido de **>**, para formar una flecha (operador flecha), a la que le damos el significado "... que apunta a..." .

De esta manera podemos decir que:

$P \rightarrow \text{mes}$ equivale a $(*P).\text{mes}$

y ambas son equivalentes a `hoy.mes`.

EJEMPLO: PUNTEROS A ESTRUCTURA

Se ingresará una lista de datos de los 20 alumnos de un determinado curso, y se la mostrará en pantalla ordenada en forma decreciente de notas. Los datos consisten en nombre y nota de cada alumno.

```
#include <stdio.h>
#define N 20
int main()
{
    struct {
        char nombre[10] ;
        int nota; } VEC[N] , aux , *P ;
    int i , j ;

    /* Ingreso de datos */
    for( P = VEC ; P < VEC+N ; P++ ) {
        fflush ( stdin ) ;
        gets( P->nombre );
        scanf("%d" , &( P->nota ) ); }

    /* Ordenamiento */
    for(P = VEC , i = 0 ; i < N-1 ; i++ )
        for ( j = 0 ; j < N-i-1 ; j++ )
            if ( (P+j)->nota < (P+j+1)->nota ) {
                aux      = *(P+j) ;
                *(P+j)   = *(P+j+1);
                *(P+j+1) = aux ;
            }

    /* Impresión de los datos */
    printf("\n\n\n");
    for ( i = 0 ; i < N ; i++ )
        printf("\n\t%12s\t%d" , (P+i)->nombre , (P+i)->nota );
}
```

Se utilizó el método de ordenamiento de burbujeo sin flag por considerarlo uno de los más difundidos, y seguramente conocido por el lector. Se debe prestar atención en el ejemplo precedente a los distintos usos dados al puntero, según los temas expuestos precedentemente.

PASAJE DE ESTRUCTURAS A FUNCIONES (POR REFERENCIA)

Al pasar los argumentos **por valor** a las funciones , el valor de la variable se copia en una variable local de la función a la que llamamos *parámetro formal*.

Si la variable a transferir es voluminosa, como bien puede serlo una de tipo estructura, demandará una gran cantidad de memoria adicional a fin de “duplicarla” e insumirá un tiempo correspondiente al traspaso de cada uno de los contenidos de los campos.

Debe considerarse además que los cambios realizados sobre la estructura, dentro de la función, en realidad se producen sobre la variable local, sin afectar para nada los contenidos de la variable original. Si fuera necesario que ésta se modificara habrá que retornar los cambios producidos al finalizar la ejecución de la función, con la consiguiente pérdida de tiempo.

Esta situación se pone de manifiesto en el siguiente ejemplo.

Se supone que en el programa principal se manejan fechas representadas por estructuras de tipo **fecha** según fuera definida en ejemplos anteriores.

Se plantea la necesidad de evitar el manejo de la fecha “*29 de Febrero*” , reemplazándola por el “*1 de Marzo*” .

La labor de corrección de fecha se dejará a la función **corregir()** .

Nótese que la corrección debe trascender la función alcanzando la variable original.

En este caso, se opta por retornar de la función la estructura corregida, con las desventajas que ya se explicaron.

```
hoy = corregir(hoy) ;           /* Llamada desde el programa */
.....
/* Cuerpo de la función */
struct fecha corregir ( struct fecha dia_hoy )
{
    if( (dia_hoy.dia == 29) && (dia_hoy.mes == 2) ) {
        dia_hoy.dia = 1 ;
        dia_hoy.mes = 3 ; }
    return dia_hoy ;
}
```

Una manera de evitar lo anteriormente mencionado es utilizar variables globales, con las ventajas y desventajas que su uso implica.

Otro medio es usar el pasaje de argumentos a funciones **por referencia**. Los punteros nos proporcionan dicho medio.

Lo que se hace en este caso es transferir a la función la **dirección** de la variable en cuestión. Ésta se almacena en la función, en un puntero, y luego todas las referencias a la variable original se hacen a través del puntero.

Se tomará el mismo caso del ejemplo anterior, utilizando transferencia de la estructura por referencia.

Obsérvese que se transfiere la **dirección** de la estructura (`&hoy`), y ésta se recibe en la función, en el puntero `P`. Nótese además, al comparar con el ejemplo anterior, que no se retorna nada al programa principal (no existe ningún `return`).

```
corregir(&hoy) ;           /* Llamada desde el programa */
.....
/* Cuerpo de la función */
void corregir ( struct fecha *P )
{
    if( (P->dia == 29) && (P->mes==2) ) {
        P->dia = 1 ;
        P->mes = 3 ;
    }
}
```

El uso de punteros permite simular una transferencia de argumentos por referencia en C, cuando en realidad lo que se está haciendo es una transferencia de punteros por valor.

PUNTEROS A PUNTEROS

Dado que un puntero apunta a cualquier tipo de variable, éste puede ser otro puntero. Tenemos el caso entonces, de puntero a puntero.

Esto constituye una forma de indirección doble, que puede ampliarse a indirección múltiple.

Sintaxis de declaración:

```
tipo ** nom_puntero ;
```

Esto significa que `nom_puntero` es un *puntero que apunta a un puntero a tipo*.

Si declaramos `int ** P;` resultará que `P` es del tipo `int **`, que se lee: *puntero a puntero a entero*.

Mientras que `*P` será un *puntero a entero*, y `**P` resultará un entero.

EJEMPLO: OBSERVANDO PUNTEROS A PUNTERO

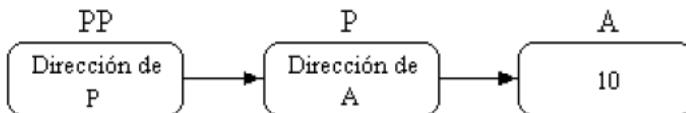
En este ejemplo se investigan los contenidos y los accesos efectuados a través de un *puntero a puntero a entero*.

```
#include <stdio.h>
int main()
{
    int A , *P , **PP ;
    A = 10 ;
    P = &A ;
    PP = &P ;
    printf ( "%4d%4d%4d", A , *P , **PP ) ;
}
```

La ejecución del programa da como resultado 10 10 10.

Obsérvese en el ejemplo la necesidad de asignar ambos punteros (esto es, hacer que P apunte a A, y que PP apunte a P), para establecer la conexión entre PP y A.

Podemos representar esta situación mediante el siguiente esquema:



Se observa claramente que existe una *doble indirección*, y que fue necesario establecer los vínculos entre PP y P, y entre P y A.

EJEMPLO: TRANSFERENCIA POR REFERENCIA DE UN PUNTERO

Se considerará un vector de enteros (VEC) con un máximo de N miembros, y un puntero a entero (P_ultimo) destinado a guardar la primera posición vacante en el vector.

El mencionado vector se podría utilizar para implementar, por ejemplo, una cola o una pila.

Se utilizará una función (agregar) a fin de agregar miembros al final del vector. A esta función se le transferirán el inicio del vector y la información del puntero al último elemento.

```
.....  
int VEC[N] , *P_ultimo , A ;      /* Declaración de variables */  
P_ultimo = VEC ;                  /* Inicialización del puntero */  
.....
```

```
agregar ( VEC , P_ultimo , A ) ; /* Llamado a la función */
.....
/* Cuerpo de la función */
agregar ( int * VEC , int * P_ultimo , int nuevo )
{
    if ( P_ultimo == VEC + N ) {
        printf ("No hay lugar");
        return ;
    }
    *P_ultimo = nuevo ;
    P_ultimo ++ ;
}
```

En la función se recibe el valor del puntero a la posición vacante, se verifica que dicha posición pertenezca al vector, y de ser así, se ubica allí el dato entero transferido. Luego se incrementa el puntero en 4 bytes (la cantidad de bytes que ocupa un `int`), a fin de apuntarlo a la nueva posición vacante del vector y se retorna al programa principal a fin de esperar otro dato.

Sin embargo, en este ejemplo se produce un error de funcionamiento. No importa la cantidad de datos ingresados, nunca se llena el vector.

¿Puede el lector determinar a qué se debe esta anomalía y sugerir una solución sin leer previamente el siguiente párrafo?

Ocurre que el puntero a la posición vacante `P_ultimo` se transfirió **por valor**, asignándose a un parámetro formal (variable local), en este caso, del mismo nombre.

La actualización de este puntero no tiene incidencia en su homónimo del programa principal y por lo tanto no se ocuparán nunca nuevas posiciones del vector, sino que se sobreescibirá siempre la primera.

Una forma de solucionar el problema es realizar la transferencia a la función del puntero a la posición vacante **por referencia**. Para ello, no se deberá transferir el valor del puntero `P_ultimo` sino su **dirección**, y en la función deberá contenerse este dato en un **puntero a puntero**.

Al realizar la transferencia por referencia los cambios realizados sobre el elemento transferido ocurren verdaderamente sobre el original.

Veamos las modificaciones realizadas al ejemplo para lograr esto:

```

.....
int VEC[N] , *P_ultimo , A ;      /* Declaración de variables */
P_ultimo = VEC ;                  /* Inicialización del puntero */
.....
agregar ( VEC , &P_ultimo , A ) ; /* Llamado a la función */
.....
/* Cuerpo de la función */
agregar ( int * VEC , int ** P , int nuevo )
{
    if ( *P == VEC + N ) {
        printf ("No hay lugar");
        return ;
    }
    **P = nuevo ;
    (*P) ++ ;
}

```

Compárese esta solución con la presentada anteriormente a fin de determinar las diferencias. Se utiliza una doble indirección que puede ser engorrosa de comprender cuando se comienza a estudiar punteros.

VECTORES DE PUNTEROS

Los punteros pueden estructurarse en vectores como cualquier otra variable. La manera de declarar un vector de N punteros al tipo de variable *tipo* es:

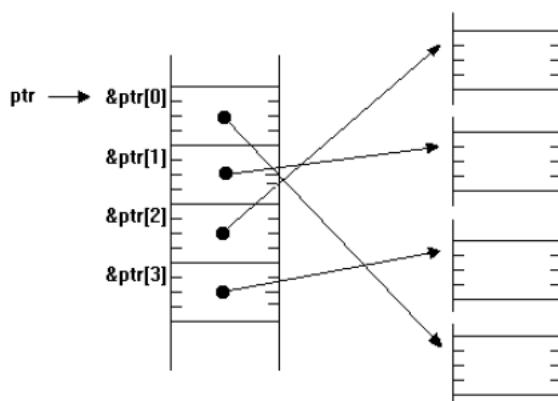
```
tipo * PTR[N];
```

Donde *PTR* es el nombre del vector, y *PTR[i]* es un puntero genérico presente en el vector.

Si declaramos el vector de punteros a entero *ptr* como vector de 4 elementos tendremos un esquema similar al siguiente:

```
int * ptr[4] ;
```

Obsérvese que cada uno de los punteros del vector es un puntero descontrolado, es decir, apunta a cualquier lado. Será necesario, por lo tanto, asignarlos con los valores adecuados.



Es necesario recordar que `ptr` no es un puntero a entero, sino que es un *puntero constante* a punteros a entero. Cada `ptr[i]` es un puntero a entero. Y cada `*ptr[i]` es un entero.

EJEMPLO: VECTOR DE PUNTEROS

Se ingresará una lista de datos de los 20 alumnos de un determinado curso, y se la mostrará en pantalla ordenada en forma decreciente de notas. Los datos consisten en nombre y nota de cada alumno.

Para resolverlo se declarará un vector de estructuras, a fin de reservar memoria para los datos. Se creará un vector de punteros a estructura, y se apuntará cada uno de ellos a la estructura correspondiente del vector de estructuras.

Cada referencia a las estructuras del vector de estructuras o a sus campos, durante el proceso de ordenamiento o de impresión, se realizará a través del vector de punteros.

Durante el ordenamiento se realizará el *swapping* sobre el vector de punteros, utilizando para ello un puntero auxiliar.

```
#include <stdio.h>
#define N 20
int main()
{
    struct {
        char nombre[10] ;
        int nota; } VEC[N], *PUNT[N], *PAUX ;
    int i , k ;

    /* Ingreso de datos */
    for( PAUX = VEC ; PAUX < VEC+N ; PAUX++ ) {
        fflush(stdin);
        gets( PAUX->nombre );
        scanf("%d",&( PAUX->nota ) );
    }

    /* Inicialización del vector de punteros */
    for( i = 0 ; i < N ; i++ )
        *(PUNT+i) = VEC+i ;

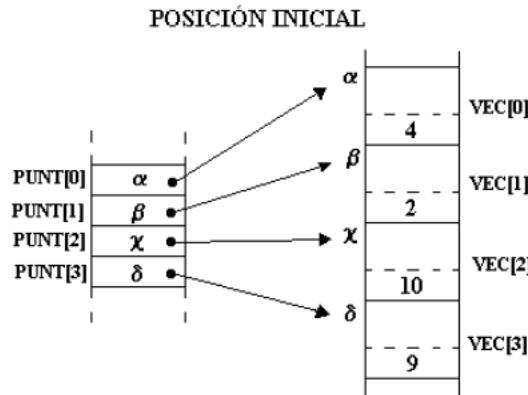
    /* Ordenamiento */
    for( i = 0 ; i < N-1 ; i++ )
        for ( k = 0 ; k < N-i-1 ; k++ )
            if ( PUNT[k]->nota < PUNT[k+1]->nota ) {

                PAUX      = PUNT[k] ;
                PUNT[k]   = PUNT[k+1];
                PUNT[k+1] = PAUX      ;
            }
}
```

```

/* Impresion de datos */
printf("\n\n\n");
for ( i = 0 ; i < N ; i++ )
    printf("\n\t%12s\t%d",PUNT[i]->nombre ,PUNT[i]->nota );
}

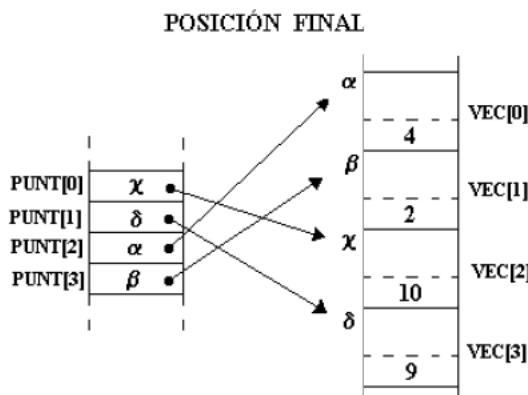
```



La figura anterior muestra, en un ejemplo de 4 elementos, la disposición inicial con cada puntero del vector PUNT apuntando a la estructura correspondiente del vector VEC.

Se muestra cada estructura de VEC, con el campo nombre vacío, y el campo nota con valores de prueba.

Luego del ordenamiento, sólo se habrán modificado los contenidos de los punteros del vector PUNT. No así los contenidos de las estructuras del vector VEC, como lo muestra la figura siguiente.



Se puede observar que al mostrar los contenidos de las estructuras de VEC a través de los punteros de PUNT, la impresión quedará ordenada en forma decreciente de notas como fue pedido.

Cuando se utilizan estructuras, es frecuente que éstas estén formadas por varios campos, por lo que pueden llegar a ser muy voluminosas.

Una copia de estructuras de este tipo demandará un cierto tiempo (mayor cuanto mayor sea el tamaño de la estructura). Este problema ya se discutió al transferir estructuras a funciones por valor.

Mucho más tiempo insumirá un intercambio (*swapping*) en un ordenamiento, que una simple copia. De hecho, 3 veces el tiempo de una copia.

Se ve entonces que el método de este último ejemplo es superior al anterior, pues en él no se produce movimiento de datos en las estructuras sino que éste se realiza solamente en el vector de punteros (mucho más pequeños).

Considerando un caso más extremo, si el vector de estructuras se encontrara almacenado en memoria ROM, sería literalmente imposible realizar *swappings* (ni ningún tipo de movimiento) sobre él, y no hubiera sido posible realizar el ordenamiento de otra forma más que del presente modo. Por supuesto, el vector de punteros debe estar ubicado en memoria de lectura/escritura.

Otro caso de **puntero a puntero** se presenta cuando se tiene un vector de punteros en el que cada uno de ellos apunta a otro vector.

Un vector de punteros se puede utilizar en forma parecida a un vector bidimensional, pese a no ser lo mismo.

La ventaja del vector de punteros es que el acceso a un elemento se efectúa mediante una indirección a través del puntero, en lugar de hacerlo mediante una multiplicación y una suma. Además, los vectores apuntados por los punteros pueden ser de longitud variable.

EJEMPLO: MANEJO DE UNA MATRIZ UTILIZANDO PUNTEROS

Se creará una matriz de enteros, y luego se realizará la impresión utilizando la modalidad subindicada en primera instancia y a continuación, el modo de desplazamiento con un puntero.

```
#include <stdio.h>
int main()
{
    int matriz[ ][5] = { { 10 , 12 , 14 , 16 , 18 },
                        { 20 , 22 , 24 , 26 , 28 },
                        { 30 , 32 , 34 , 36 , 38 },
                        { 40 , 42 , 44 , 46 , 48 } };
    int i,k;

    /* Usando arrays */
    for( i = 0 ; i < 4 ; i++ )
        printf("\n\n\t\t\t");
}
```

```

        for( k = 0 ; k < 5 ; k++ )
            printf("%5d", matriz[i][k] );
    }

/* Usando punteros */

printf("\n\n\n\n");
for( i = 0 ; i < 4 ; i++ ) {
    printf("\n\n\t\t\t");
    for( k = 0 ; k < 5 ; k++ )
        printf("%5d", * (* ( matriz + i ) + k ) );
}

printf("\n\n");
printf("matriz = %p matriz+3 = %p\n", matriz , matriz+3 );
printf("*matriz = %p *matriz+3 = %p\n", *matriz ,*matriz+3);
printf("**matriz = %d *((matriz+3)+2)=%d", \
                **matriz , *((matriz+3)+2) );
}

```

Recordemos que la matriz puede ser considerada como un **vector de vectores**. De esta forma podemos decir que:

- matriz[i][k]** es un entero
- matriz[i]** es un puntero a entero (vector de enteros)
- matriz** es un puntero a punteros a entero (vector de vectores de enteros)

Desde el punto de vista de los **punteros**, podemos afirmar que:

- matriz** es un puntero a punteros a entero
- *matriz** es un puntero a entero
- **matriz** es un entero

La ejecución del programa del ejemplo anterior dio como resultado dos imágenes idénticas. Del pedido de información sobre lo apuntado por los punteros se obtuvo:

matriz = 0028FEB8 matriz + 3 = 0028FEF4

Esto equivale a decir que **matriz** apunta a la dirección del elemento **matriz[0][0]**, y **(matriz+3)** apunta 15 elementos enteros mas adelante (al inicio de la fila 3).

La distancia entre las dos posiciones es 3C bytes, que expresado en forma decimal es 60 bytes. Esto es equivalente a 15 enteros de 4 bytes cada uno.

Esto significa, según las reglas de la aritmética de punteros, que `matriz` apunta a un tipo de dato de 20 bytes (5 enteros), es decir a un vector de 5 enteros.

La imagen en pantalla mostró que:

```
*matriz = 0028FEB8           *matriz + 3 = 0028FEC4
```

Dado que $28FEC4 - 28FEB8$ es igual C bytes (o sea, 3 enteros), se desprende que `*matriz` representa un puntero a entero.

También se obtuvo :

```
** matriz = 10           *(*( matriz + 3 ) + 2 ) = 44
```

Podemos afirmar entonces que:

$\ast(\ast(\text{matriz}+i)+k)$ es equivalente a `matriz[i][k]`

PUNTEROS A FUNCIÓN

Si bien una **función** no es una **variable** que pueda ser apuntada por un puntero, está asociada a la dirección de memoria donde está ubicado el comienzo de su código.

Cuando el compilador traduce el **código fuente** de la función al **código objeto** y lo ubica en algún lugar de la memoria, relaciona el nombre de la función con la posición de memoria donde está ubicado su inicio.

El acceso a una función se produce a través de su nombre. Dicho de otro modo, el nombre de la función es la clave para acceder a su código.

Podemos considerar entonces que el **nombre de la función** es un puntero (constante) que apunta al inicio de la función.

Si bien esto es similar a lo que ocurre con los nombres de los vectores, en cuanto a que ellos apuntan a la dirección de inicio, existe una diferencia conceptual importante:

Mientras que el nombre de un vector apunta al área de datos, el nombre de una función apunta al área de código.

Sintaxis de declaración de un **puntero a función**:

```
tipo ( * nom_punt ) ();
```

donde **tipo** es el tipo de dato *devuelto* por la función, y **nom_punt** es el identificador de la variable puntero.

Es opcional colocar entre los paréntesis el o los tipos de los argumentos, en forma análoga a lo que sucede en el prototipo de las funciones. Utilizando dicha opción, la declaración quedaría :

```
tipo (* nom_punt) (tipo, tipo, ... , tipo) ;
```

La asignación al puntero de la **dirección de la función** se realiza igualando el identificador del puntero al nombre de la función, sin paréntesis ni argumentos:

```
nom_punt = nom_funcion ;
```

La invocación de la función a través del puntero se realiza expresando *lo apuntado por el puntero* entre paréntesis, seguido por un juego de paréntesis que contenga los argumentos de la función (c que esté vacío si no tuviera).

De esta forma resulta equivalente:

```
nom_funcion(A,B)    que    (* nom_punt)(A,B)
```

No se aprecia de esta manera ninguna ventaja en el uso de punteros a función, pero si consideramos que podemos formar un **vector de punteros a función** y luego seleccionarlos numéricamente, vemos que se puede sustituir una selección a través de **switch**, por una selección de funciones a través de un vector de punteros utilizando el **subíndice** numérico.

Otra aplicación es la de *enviar funciones como argumento a funciones* (obviamente por referencia) utilizando un puntero a función como parámetro formal.

EJEMPLO: SELECCIÓN DE FUNCIÓN A TRAVÉS DE PUNTEROS

Se considerarán 2 funciones, **sumar** y **restar**, y un vector de 2 punteros a función **p[]**. La asignación de dicho puntero tendrá lugar simultáneamente con su declaración.

La selección de la función se realizará a través del subíndice del vector de punteros.

```
#include <stdio.h>
int sumar ( int , int ) ;
int restar ( int , int ) ;

int main()
{
```

```

int a , b , i ;
int ( *p[] )( ) = { sumar , restar } ;

scanf ("%d %d" , &a , &b ) ; /* Ingreso de datos */
do {
    printf ("\n\nIngrese opción. 0- Sumar 1- Restar\n");
    scanf ("%d" , &i ) ;
} while ( ( i != 0 ) && ( i != 1 ) ) ;
printf ("\n\n Resultado = %d " , (*p[i])(a,b) ) ;
}

int sumar ( int x , int y )
{ return x + y ; }

int restar ( int x , int y )
{ return x - y ; }

```

EJEMPLO: PASAJE DE FUNCIONES A FUNCIONES (POR REFERENCIA)

Se mostrará en este ejemplo cómo transferir una función como argumento a otra función utilizando un **puntero a función** como parámetro.

En este caso, se calculará el valor de $X^2 + Y^2$, o bien de $X^3 + Y^3$ según se seleccione. Las funciones de cálculo del cuadrado o del cubo se transferirán a la función de suma como argumentos, a través de un puntero a función que forma parte de un vector de 2 punteros.

```

#include <stdio.h>

/* Prototipos */
int suma(int,int,int(*)(int));
int cuadrado(int);
int cubo(int);

int main()
{
    int a , b , seleccion ;
    int ( *p[2] )( int ) ;
    p[0] = cuadrado ;
    p[1] = cubo ;
    printf ("Suma de cuadrados : 0      Suma de cubos : 1 \n\n") ;

    /* Menú */
    scanf ( "%d" , &seleccion ) ;
    printf ( "\n\n Ingrese dos valores " ) ;

```

```

/* Ingreso de datos */
scanf ( "%d %d", &a , &b ) ;
printf("\n\n\n Resultado = %d", suma( a, b, p[seleccion] ) ) ;
}

/* Funciones */
int cuadrado ( int x )
{ return x * x ; }

int cubo ( int x )
{ return x * x * x ; }

int suma( int x , int y , int ( *ptr ) ( int ) )
{ return (*ptr)(x) + (*ptr)(y) ; }

```

EJEMPLO: VECTOR DE PUNTEROS A FUNCIÓN

Se ingresarán a través del port 44H las señales provenientes de 8 alarmas. Cada una de ellas corresponderá a una rutina de acción que se deberá ejecutar en el caso de estar activada la alarma correspondiente.

Una manera de efectuar la selección sería a través de una instrucción `switch` con 8 opciones, en las que se invoque la rutina de cada alarma para cada uno de los `case`.

En cambio en el ejemplo se utilizará un vector de 8 punteros a función. Cada uno de ellos apuntará a una de las rutinas de acción de las alarmas y la selección del puntero se hará en forma numérica. No se acompañará el código correspondiente a las funciones, pues no es relevante para el ejemplo. Tampoco se incluye la totalidad de las acciones necesarias para trabajar con puertos.

Nota: se incorpora un lazo para que la ejecución del programa no tenga fin.

```

#include <stdio.h>

void alarma1 ();
void alarma2 ();
void alarma3 ();
void alarma4 ();
void alarma5 ();
void alarma6 ();
void alarma7 ();
void alarma8 ();

int main()
{
    void (*pf[])() = { alarma1,alarma2,alarma3,alarma4,alarma5,\n
                      alarma6,alarma7,alarma8 } ;
    unsigned char dato , mascara , i ;

```

```
while ( 1 ) {          /* Bucle infinito */
    dato = inb(0x44) ; /* Ingreso de información */
    mascara = 1 ;        /* Filtro para cada alarma */
    for( i = 0 ; i < 8 ; i++ , mascara <<=1 )
        if ( dato & mascara )
            (*pf[i])() ;
}
```

PROHIBIDA LA REPRODUCCIÓN TOTAL O PARCIAL

PROBLEMAS PROPUESTOS

1. Realice un programa que esté compuesto por diversas funciones (propias y de librerías). Indique en qué dirección del segmento de código comienza cada una, incluyendo el `main`.
2. Realice un programa que utilice variables globales y locales. Determine en qué lugar de la memoria se ubican. ¿Qué conclusión obtiene?
3. Declare varias variables locales. Estas se ubican en el área de memoria llamada “pila”. Determine en qué sentido crece la pila.
4. Ante la siguiente fracción de programa:

```
int A, B, C , *P, *Q, *R ;  
P = &A; R = &C;  
printf("%p %p %p", P, Q, R);
```

Explique los resultados.

5. Ante la siguiente fracción de programa:

```
int A, B, C , *P, *Q, *R ;  
P = &A; Q = &B; R = &C;  
printf("%p %p %p", P, Q, R);
```

Explique los resultados. Explique la diferencia con los resultados del programa anterior. ¿Qué conclusiones obtiene?

10. ARGUMENTOS DEL MAIN

El `main` es una función de características similares a las demás. Es la función principal y no puede estar ausente en nuestro programa pero es, en definitiva, una función.

Como tal es capaz de recibir argumentos que se alojan, como toda función, en sus parámetros formales.

Recibir argumentos y retornar resultados no significa realizar ingresos desde teclado e impresiones en pantalla, sino recibir y entregar estos valores al programa **invocante o llamante**, que es aquel que gestionó el llamado a nuestro programa.

La mayor parte de las veces el programa invocante del nuestro es el sistema operativo. En otros casos, el programa invocante es otro proceso, que recibe el nombre de *proceso padre*, para el cual nuestro programa representa su *proceso hijo*.

En esta sección se desarrollará el caso en que el sistema operativo es el programa invocante. Para poder enviar argumentos al `main` necesitamos hacer uso de la **línea de comandos**, a través de una **consola** de texto.

TIPO DE LOS ARGUMENTOS

`main` es capaz de recibir 3 argumentos de tipo definido y en un orden preestablecido. A continuación se muestra el formato de los parámetros formales que reciben dichos argumentos.

```
int main( int argc , char *argv[] , char *envp[] )
```

Siendo éstos:

- `argc` Cantidad de argumentos ingresados por línea de comandos.
- `argv` Vector de punteros a los argumentos ingresados por línea de comandos.
- `envp` Vector de punteros a las variables de entorno.

Los nombres de los parámetros formales son nombres que se han utilizado tradicionalmente a lo largo del tiempo, pero podrían ser reemplazados por otros más acordes al gusto del lector sin problemas.

Los nombres tradicionales representan (en inglés) la función de cada argumento de la siguiente manera: `argc` (*argument count*) , `argv` (*arguments vector*) y `envp` (*environment pointers vector*).

Dado que tanto `argv` como `envp` son **vectores de punteros a char**, pueden ser representados como lo que realmente son, **punteros a punteros a carácter**:

```
int main( int argc , char **argv , char **envp )
```

El orden de los parámetros debe ser respetado. Esto indica que no puede utilizarse el segundo parámetro sin declarar el primero, ni utilizarse el tercero sin declarar los dos anteriores.

ARGUMENTOS POR LÍNEA DE COMANDOS

Cuando se invoca al programa desde la línea de comandos, se puede acompañar el nombre de éste con una serie de strings que constituyen los argumentos transferidos. Por ejemplo:

```
C:\pruebas> prueba.exe hola mundo
```

Los argumentos se ingresan separados por blancos (espacios) y finalizan al hacerse Enter, lo que provocará la ejecución del programa.

Si se desea incorporar un argumento que contenga blancos, será necesario encerrarlo entre comillas.

El nombre del programa ejecutable constituye el argumento cero.

Los argumentos de línea de comandos se copian en la pila donde quedan a disposición del programa. Dichos argumentos residen en la pila como una secuencia de strings de diferente longitud, siendo el primero de ellos el nombre del programa ejecutable.

Esta secuencia de strings es acompañada por un vector de sendos punteros que almacenan las direcciones donde aquellos comienzan. Por lo tanto, el vector de punteros constituye la clave para acceder a los argumentos.

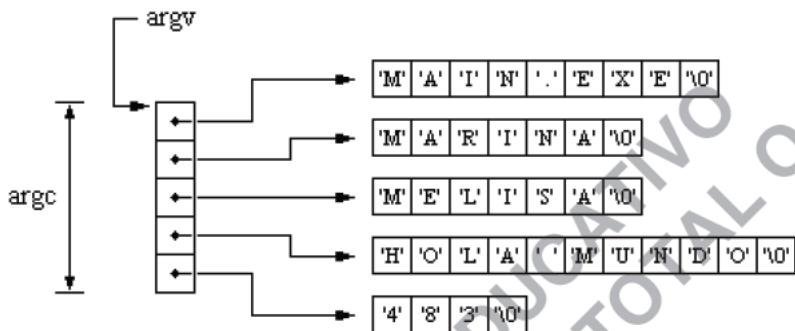
La función `main` recibe como argumento la dirección del mencionado vector de punteros. Esta información es insuficiente para el manejo del vector. Es necesario conocer también su longitud. Ésta no se puede prever dado que la cantidad de argumentos de la línea de comandos se define recién en tiempos de ejecución.

La situación se soluciona debido a que el parámetro entero `argc` contiene la longitud del vector `argv`.

De esta manera, los dos primeros argumentos de `main` están relacionados y se utilizan en conjunto.

En la siguiente figura se muestra la disposición esquemática del vector y los strings, para la siguiente invocación del programa `MAIN.EXE` desde la carpeta de trabajo:

C:\pruebas> MAIN MARINA MELISA "HOLA MUNDO" 483



Obsérvese que el último argumento no se almacenó como el número 483, sino como el string "483".

EJEMPLO: LISTANDO LOS ARGUMENTOS

En este ejemplo se mostrará en pantalla la lista de argumentos ingresados, valiéndose de `argc` para manejar el barrido del vector `argv`.

Este programa no debe ejecutarse desde nuestro IDE como todos los ejemplos y ejercicios anteriores. Debe compilarse y enlazarse para construir el archivo ejecutable por medio del IDE, y luego la ejecución debe realizarse desde el sistema operativo. En Windows, es preciso abrir una línea de comandos mediante el programa `cmd.exe` que se ejecuta a través del menú Inicio. Esto nos permitirá ejecutar el programa, a la vez que le envíamos argumentos.

```
#include <stdio.h>
int main ( int argc , char *argv[] )
{
    int I ;
    printf("La cantidad de argumentos es : %d \n\n" , argc );
    for ( I=0 ; I<argc ; I++ )
        puts ( argv[I] ) ;
}
```

EJEMPLO: MANIPULANDO LOS ARGUMENTOS

Este programa mostrará en pantalla el segundo argumento de línea de comandos tantas veces como lo indique el primero.

Una posible invocación sería: `MAIN 10 hola`

De acuerdo con lo enunciado debe imprimirse la palabra “hola” 10 veces.

Un problema que se debe resolver es el manejo numérico del primer argumento que, como hemos visto, siempre quedan representados como string. Esta situación queda resuelta mediante el auxilio de la función `atoi()`, que recibe una string como argumento y retorna el entero correspondiente. La cabecera relacionada es `stdlib.h`.

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc , char *argv[] )
{
    int I ;
    for ( I=0 ; I < atoi(argv[1]) ; I++ )
        puts ( argv[2] ) ;
}
```

VARIABLES DE ENTORNO

El sistema operativo trabaja con una serie de parámetros seleccionables que determinan parte de su comportamiento. Estas selecciones se almacenan en contenedores denominados **variables de entorno**. En Windows es posible verlas (y editarlas) accediendo a Equipo, botón derecho “Propiedades”, Configuración avanzada del sistema, Opciones avanzadas, Variables de Entorno.

Es posible observar estas variables de entorno desde un programa C.

El parámetro `envp` representa un vector de punteros, donde cada uno de ellos apunta a una copia de variable de entorno.

No se dispone en este caso de un parámetro que indique la longitud de dicho vector, sino que éste presenta un elemento “testigo” que marca su finalización.

El vector de punteros finaliza con el puntero `NULL`. Este valor está definido como dirección cero y representa el valor lógico *falso*. Es decir que es una dirección de memoria con todos sus bits en cero.

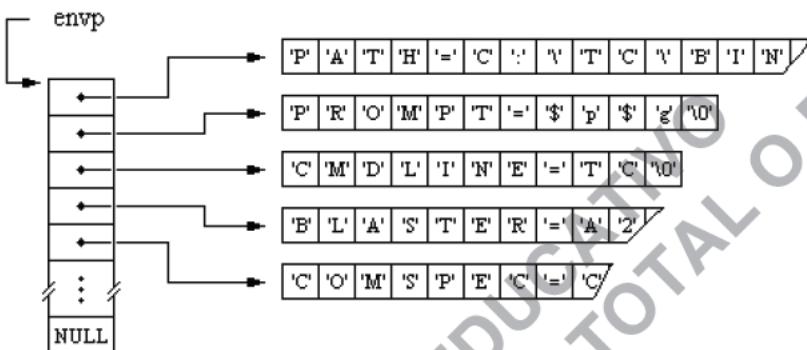
La lista de variables de entorno disponible de esta manera *es una copia* de la original, y por lo tanto su modificación directa no tiene efecto alguno sobre el comportamiento del sistema operativo.

Si se desea modificar las verdaderas variables se deberá interactuar con el sistema operativo utilizando las **llamadas a sistema** apropiadas, algo que excede el foco de este capítulo.

Tanto el vector de punteros como la copia de las variables de entorno se ubican en el **área estática** del segmento de datos y no en la **pila**, como en el caso anterior.

A continuación se ilustra la disposición esquemática del vector `envp` y la lista de strings que representan las variables de entorno.

Nótese el puntero `NULL` al final del vector.



EJEMPLO: VARIABLES DE ENTORNO

Este ejemplo que puede ejecutarse desde el IDE, debido a que no hay argumentos en línea de comandos, permite recorrer el vector de punteros `envp` y mostrar las variables de entorno del sistema.

```
#include <stdio.h>
int main ( int argc , char **argv , char **envp )
{
    int I = 0 ;
    while ( *(envp+I) ) {
        puts ( *(envp+I) ) ;
        I++ ;
    }
}
```

A modo ilustrativo se muestra una porción de un resultado:

```
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\Pepe\AppData\Roaming
CommonProgramFiles=C:\Program Files (x86)\Common Files
COMPUTERNAME=PEPE-PC
ComSpec=C:\Windows\system32\cmd.exe
HOMEDRIVE=C:
HOMEPATH=\Users\Pepe
LOCALAPPDATA=C:\Users\Pepe\AppData\Local
```

PROBLEMAS PROPUESTOS

1. Construir un programa que se denomine **POTENCIA.EXE** y que se invoque desde línea de comandos con dos argumentos enteros X e Y, y muestre en pantalla el valor de X elevado a la Y.
2. Construir un programa que determine si “**PATH**” se encuentra entre las variables de entorno. Realizar una comparación para que no importen las diferencias en mayúsculas y minúsculas.
3. Construir un programa que se denomine **GRADOS.EXE** y que se invoque desde línea de comandos con dos argumentos. El primero representa un valor de temperatura y tiene formato flotante. El segundo representa el sistema en que está expresada dicha temperatura y puede ser **C** (Celsius), **F** (Farenheit) o **K** (Kelvin).

GRADOS 57.25 F

Debe mostrarse en pantalla el equivalente de la temperatura en los tres sistemas.

6. Realizar un programa denominado **PROMEDIO.EXE** que reciba valores enteros por línea de comandos e imprima el promedio de éstos.

Evitar la división por cero y notar que la cantidad de valores es arbitraria.

PROMEDIO 56 34 789 23

7. Realizar un programa que reciba nombres como argumentos por línea de comandos y los muestre en pantalla ordenados alfabéticamente.

Realizar el ordenamiento sin mover ni copiar los strings. Utilizar para ello el vector de punteros.

Excluir del ordenamiento el nombre del programa ejecutable.

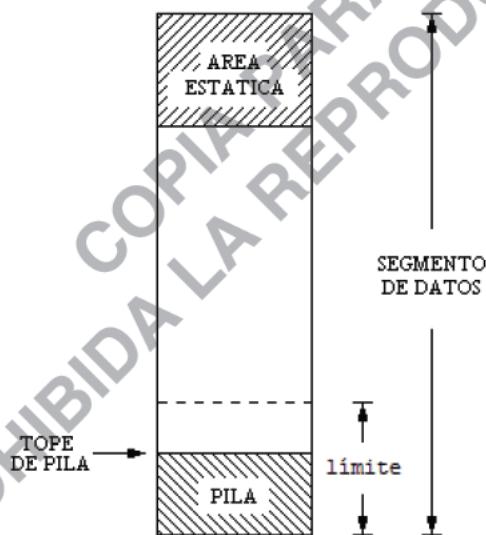
11. GESTIÓN DINÁMICA DE MEMORIA

El funcionamiento de nuestros programas requiere usualmente del uso de cierto monto de memoria RAM.

Esta memoria se gestiona en forma de variables locales y globales de diferente tipo, ya sean estáNDAR de C, o bien de tipos creados por el usuario como ser vectores, estructuras, uniones, etc.

Las variables y constantes **globales** (inicializadas o no), como así también las variables locales declaradas como estáticas (mediante el uso del modificador `static`), se alojan en un área del segmento de datos, de tamaño indeterminado, a la que llamamos **área estática**.

Las variables **locales** y parámetros formales de las funciones (incluyendo a `main`) se ubican en la **pila** (*stack*), la cual tiene un tamaño que oscila entre algunos cientos de kilobytes hasta varios megabytes, dependiendo del sistema operativo en cuestión.



La disposición del área estática y la pila, *desde el punto de vista de nuestro programa*, es la que muestra la figura. Estas direcciones serán en realidad **direcciones virtuales** que son luego traducidas a direcciones reales por el microprocesador y el sistema operativo. En otras palabras, nuestros punteros contienen direcciones que no son las reales.

El área estática se ubica a partir de las posiciones inferiores del segmento de datos, mientras que la pila lo hace en las posiciones superiores.

Nótese que la pila crece hacia posiciones inferiores de memoria.

Además, se aprecia una amplia zona de memoria comprendida entre los bloques descriptos.

El uso de la memoria considerada en los párrafos anteriores requiere que el programador conozca, en **tiempo de diseño** del software, la cantidad de memoria que requerirá en cada caso a fin de declarar las variables.

Frecuentemente la cantidad de memoria requerida recién se conoce en **tiempos de ejecución**, cuando ya es demasiado tarde para hacer cualquier tipo de declaración de variables a fin de establecer una reserva de memoria.

Tal es el caso de la siguiente situación de ejemplo: “Almacenar los datos de las personas que se presenten ante una determinada solicitud de empleo. Los datos consisten en nombre, edad y número telefónico. Los interesados se presentarán en el horario de 8:00 a 18:00 hs.”

Recién a las 18:00 se conocerá el número de datos a almacenar. Cualquier tamaño de vector de estructuras destinado a tal fin será solamente tentativo, tratándose de cubrir las necesidades probables, en exceso, con el menor desperdicio de memoria posible.

En todo caso, se corre el riesgo de reservar memoria insuficiente, o de manera ineficiente.

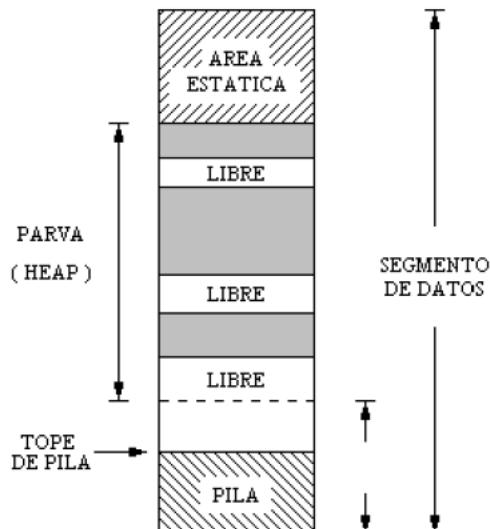
ASIGNACIÓN DINÁMICA DE MEMORIA

Una forma de solucionar este problema es recurrir a una segunda forma de gestión de memoria, denominada **gestión dinámica de memoria**.

Esta forma de gestión consiste en asignar memoria en tiempos de ejecución, según ésta se solicite, del área de memoria libre ubicada entre la pila y la zona estática, liberando dicha memoria cuando se deja de utilizar.

El área de memoria utilizada para esta asignación recibe el nombre de **parva (heap)**.

Dado que en la parva se asignan bloques de memoria que posteriormente se liberan al finalizar su uso, luego de un tiempo, ésta estará constituida por espacios de memoria en uso y espacios libres, de diferente tamaño.



En la figura anterior se representa la parva constituida por *espacios en uso* (grises) y *espacios libres*, ubicada entre el área estática y la pila.

Cuando la parva está inicialmente libre, los bloques solicitados se asignan ocupando espacios de memoria de direcciones crecientes.

Esto significa que el crecimiento de la parva es contrapuesto al de la pila.

Una ventaja de esta forma de asignación es que la memoria que se deja de utilizar puede ser liberada por el programa, quedando disponible para nuevas asignaciones dentro del mismo.

La posibilidad de este tipo de gestión permite la implementación de estructuras enlazadas de datos tales como **listas** y **árboles**.

*Cuando ya no se utilizan los bloques de memoria gestionados dinámicamente es necesario liberarlos, a fin de que estén disponibles para nuevos pedidos. El uso de memoria sin liberación se denomina **memory leak** y es un problema de muy difícil resolución.*

FUNCIONES DE GESTIÓN DINÁMICA DE MEMORIA

El estándar ANSI define cuatro funciones de gestión dinámica de memoria.

- `malloc()`
- `calloc()`
- `realloc()`
- `free()`

De las cuales `malloc()` y `free()` cubren la necesidades más frecuentes.

Estas funciones están asociadas según el estándar ANSI a la cabecera `stdlib.h`.

MALLOC()

Mediante la función `malloc()` se solicita la reserva de un bloque de memoria de un determinado tamaño. Su prototipo es:

```
void * malloc(size_t tamaño)
```

La función `malloc()` requiere que se le informe el tamaño *en bytes* del bloque libre que se desea obtener.

El tipo definido como macro `size_t` se utiliza en la biblioteca estándar para representar **cantidades enteras positivas**, como ser un tamaño en bytes o una determinada cantidad de bloques.

El resultado entregado por `malloc` es la dirección inicial de un bloque de memoria libre si la gestión tuvo éxito, o el puntero `NULL` si no se pudo asignar memoria.

Dado que `malloc` “no sabe” para qué se está gestionando el bloque de memoria, retorna la dirección de éste con formato `void *`. Probablemente sea necesario *castear* este tipo a fin de asignarlo al puntero correspondiente, dependiendo de la configuración del compilador.

Es necesario siempre comprobar que la dirección retornada por `malloc()` sea una dirección válida. Esto significa que sea diferente de `NULL`.

Ignorar tal situación conlleva a resultados desastrosos.

PORATILIDAD DE LOS TAMAÑOS

Es importante asegurar la **portabilidad** de los tamaños gestionados dinámicamente en la memoria a fin de no exceder los límites de la memoria asignada.

Esta portabilidad se logra utilizando el operador `sizeof()` para establecer los tamaños.

Un error frecuente en este sentido es suponer que el tamaño de una estructura es igual a la suma del tamaño de sus campos.

Exceder los límites de la memoria asignada dinámicamente puede derivar en la pérdida de la totalidad de la información almacenada en la parva.

A continuación se muestra la gestión de memoria para una estructura, considerando todos los puntos anteriores :

```
struct DATO {  
    char NOMBRE[20];  
    char SEXO ;  
    int EDAD ;  
} *P ;
```

```
if ( !(P = (struct DATO *) malloc( sizeof(struct DATO)))) {  
    printf("NO HAY SUFICIENTE MEMORIA");  
    exit(1);  
}
```

Nótese que se evalúa la condición de *falso* del valor asignado al puntero P. Esto es debido a que la evaluación lógica de NULL es *falso*, puesto que su valor numérico es cero.

En caso de detectarse tal situación, se la informa y se aborta el proceso.

CALLOC()

La función `calloc()` es similar a `malloc()` en cuanto a la gestión de la memoria. La diferencia estriba en que `calloc()` permite expresar el tamaño de memoria solicitado, mediante una cierta cantidad de bloques de una determinada magnitud.

Su prototipo se muestra a continuación:

```
void * calloc(size_t numero, size_t tamaño)
```

El tamaño de la memoria asignada será igual a `numero * tamaño` por lo que se podría haber utilizado en su lugar `malloc(numero * tamaño)` con idénticos resultados.

Se muestra a continuación la gestión de memoria para un vector de 10 flotantes utilizando `calloc()`.

```
#define N 10  
float *P ;  
if ( !(P = (float *) calloc( N , sizeof(float) ))) {  
    printf( " NO HAY SUFICIENTE MEMORIA ");  
    exit(1);  
}
```

REALLOC()

Esta función permite reasignar el tamaño asignado a un bloque. Puede ocurrir que el tamaño de un bloque solicitado anteriormente ya no sea el adecuado, por ser demasiado chico o por ser demasiado grande.

Su prototipo es:

```
void * realloc ( void * , size_t nuevo_tamaño )
```

La función `realloc()` requiere que se le entregue la dirección del bloque a reasignar y el nuevo tamaño deseado.

Puede ocurrir que `realloc()` no pueda redimensionar el bloque en el lugar asignado anteriormente. De ser así, gestiona un nuevo bloque sobre el cual copia la información del anterior y retorna su dirección.

En caso de no poder reasignar dicho bloque, `realloc()` retorna el puntero `NULL`.

```
int *P ;  
#define N 10  
P = (int *) malloc( N ) ;  
· · · · · · · · · · · ·  
P = (int *) realloc( P , 3 * N ) ;
```

FREE()

La función `free()` libera un bloque de memoria gestionado anteriormente a fin de poder asignarlo nuevamente.

Su prototipo es:

```
void free ( void * )
```

La función `free()` requiere que se le entregue la dirección de un bloque de memoria gestionado por alguna de las funciones anteriormente descriptas.

El argumento entregado a la función `free()` debe provenir de una asignación dinámica anterior.

Si no fuera así, se corre el riesgo de destruir el sistema de asignación dinámica de memoria con la consiguiente pérdida de información residente en la parrva.

PROBLEMAS PROPUESTOS

1. Desarrolle un programa que lea un número de teclado, que indica a su vez cuántos números enteros ingresará el usuario a continuación. Crear el vector para almacenar el tamaño exacto de los datos y leer los enteros que serán guardados en el vector "dinámico". Informar el promedio de los datos ingresados.
2. Utilizando la siguiente definición de estructura, realice un programa que solicite memoria para 5 alumnos.

```
struct alumno {  
    int legajo;  
    char sexo;  
    char nombre[30];  
    float promedio;  
};
```

El usuario ingresará por teclado los datos que se cargan en las estructuras. Mostrar el nombre de los alumnos uno debajo del otro. A continuación mostrar cuál es el alumno con mejor promedio.

12. FORMATO DE PUNTO FLOTANTE IEEE 754

NÚMEROS DE PUNTO FIJO

Las variables y datos de tipo entero (llamados de **punto fijo**) son enormemente utilizadas, pero no son aplicables a todos los casos.

En general, la cantidad de combinaciones que permite un número de n dígitos expresados en una base b es igual a b^n .

Si consideramos que una de estas combinaciones debe ser aplicada al número cero, el rango de un número genérico X no signado es:

$$0 \leq X \leq b^n - 1$$

En el caso de los números binarios la base es igual a 2, por lo que la expresión anterior resulta:

$$0 \leq X \leq 2^n - 1$$

De esta forma, si consideramos una variable de tipo `int` cuya longitud es 32 bits, podremos representar números no signados (`unsigned int`) comprendidos en el rango:

$$0 \leq X \leq 2^{32} - 1$$

Esta relación establece que el mayor número representado es:

$$2^{32} - 1 = 4.294.967.295$$

Aproximadamente igual a 4.29×10^9 .

Si bien este valor es grande, puede ser insuficiente para representar determinadas cantidades como ser:

- Carga eléctrica del electrón 1.6×10^{-19} coulombs
- Distancia a la estrella Rigel Orión 8.51×10^{15} kilómetros

En el primer caso, la cantidad expresada es demasiado pequeña para ser representada por números enteros. Esto constituye una condición de **underflow**.

En el segundo caso, la cantidad supera el valor máximo representado. Esta condición se denomina "desborde" u **overflow**.

Si se representan los números enteros en la recta numérica, se puede apreciar que la separación entre los mismos es uniforme. Cualquier intento de representación entre dos números enteros consecutivos, derivará en una situación de *underflow*, mientras que al excederse el mayor valor representado se occasionará un *overflow*.

En la figura se muestra esta situación para el caso de una variable de tipo `unsigned char`.



PRECISIÓN MULTIPLE

Una posibilidad de resolver, o al menos mejorar, situaciones de *overflow* consiste en recurrir al método de **precisión múltiple**.

Este consiste en utilizar k números de punto fijo, de manera de obtener un número de longitud kn bits que permita extender el alcance de representación a $2^{kn} - 1$.

El caso en que k vale 2 se denomina de **doble precisión**. El costo a pagar en este caso es la utilización del doble de circuitos, o bien, una demora del doble de tiempo en el procesamiento.

Si consideramos el caso en que $n=32$ y $k=2$, tendremos que el máximo valor representado es $2^{2 \cdot 32} - 1$, es decir $2^{64} - 1$.

Este valor equivale a 1.84×10^{19} , el cual alcanza para representar la distancia en kilómetros a la estrella Rigel de la constelación de Orión (valor mencionado anteriormente). Sin embargo, el rango anterior puede no ser suficiente para otras representaciones.

Nótese además que para cantidades tan grandes, no es importante generalmente una gran precisión en cuanto a la determinación del número. De hecho, el mencionado valor de 1.84×10^{19} , es solamente una **aproximación** al verdadero valor.

Por otra parte, el sistema de precisión múltiple no resuelve el problema planteado por el *underflow*.

NOTACIÓN CIENTÍFICA

Es una representación numérica que consiste en introducir la **notación exponencial** con respecto a un número denominado *base*, de manera de facilitar el manejo de las cantidades.

En el sistema decimal, la base utilizada es 10.

NOTACIÓN CIENTÍFICA NORMALIZADA

En el sistema decimal, cualquier número real puede expresarse mediante la denominada **notación científica normalizada**.

La **normalización** consiste en la determinación de la ubicación del punto decimal del número.

Para expresar un número en notación científica normalizada multiplicamos o dividimos por 10 tantas veces como sea necesario para que todos los dígitos menos uno, aparezcan a la derecha del punto decimal y de modo que el dígito ubicado a la izquierda del punto decimal no sea cero.

Luego se compensa el paso anterior multiplicando al número así obtenido por 10 elevado a un exponente que realice el ajuste. Este exponente puede ser positivo o negativo.

A continuación se muestran unos ejemplos:

- 526.3235 equivale a 5.263235×10^2
- 0.0000564 equivale a 5.64×10^{-5}
- -25200000 equivale a -2.52×10^7

En general, en el sistema de numeración decimal, un número real distinto de cero puede representarse como:

$$\pm R \times 10^E$$

Donde:

- R es un número real ajustado como se mencionó anteriormente, de tal manera que $1 \leq |R| < 10$
- E es un exponente entero positivo, cero o negativo.

FORMATO DE PUNTO FLOTANTE

El formato de **punto flotante** es la forma de implementar la **notación científica** en la computadora.

Debemos recordar que en ésta los números tienen una **longitud fija** determinada por su arquitectura (cantidad de bits). Por lo tanto, la representación de los valores R y E mencionados anteriormente también tendrán esa limitación.

Dado que el procesador trabaja con números binarios, la base b en este caso será 2. Como la base es entonces un valor conocido, no estará representado en el número mismo, por lo que los bits se utilizarán para representar al signo, a R y al exponente E.

Por supuesto, los valores de R y E también estarán codificados en binario.

Veremos a continuación el formato característico de los flotantes de 32 bits, tal como se utilizan en el lenguaje C.

FORMATO DE PUNTO FLOTANTE DE 4 BYTES (IEEE 754)

En este formato se asigna un bit al signo del número, 8 bits al exponente y los restantes 23 bits a la mantisa.

El esquema de dicho formato se muestra a continuación. El byte mas significativo se representa a la izquierda.



Si bien el exponente (E) puede tener distintas codificaciones numéricicas, podemos afirmar que es un número entero con signo.

La mantisa (M) es considerada frecuentemente como una fracción, de tal forma que sus valores están acotados de la siguiente forma :

$$-1 < M < 1$$

Con estas consideraciones, el rango del número esta comprendido aproximadamente en el intervalo: $\pm 2^{\pm 128}$

Para representar este rango en punto fijo sería necesario un número de 128 bits de longitud.

NORMALIZACION DE LA MANTISA

Un número binario puede tener, según lo expuesto, diversas representaciones. Consideremos el número 10011.1011 .

Éste puede representarse, entre otras formas, como :

- 100111011×2^{-4}
- 100.111011×2^2
- 1.00111011×2^4
- 0.100111011×2^5

Es conveniente adoptar un formato estándar de representación. Las formas más usuales son:

- El bit más significativo (*msb*) de la mantisa se ubica **inmediatamente a la izquierda** del punto decimal (1.00111011×2^4)
- El bit más significativo (*msb*) de la mantisa se ubica **inmediatamente a la derecha** del punto decimal (0.100111011×2^5)

BIT OCULTO

En caso de seleccionar la primera de las opciones, se sabe que todos los números, con excepción del cero, comenzarán con “1” a la izquierda del punto decimal. Por esta razón se lo omite ganando entonces un bit adicional para representar a la mantisa.

Este bit implícito recibe el nombre de **bit oculto**.

POLARIZACION DEL EXPONENTE

La representación del cero admite múltiples exponentes debido a que en la expresión 0.00×2^E cualquier valor de E permite que la expresión valga cero.

Si el valor a representar es muy pequeño, es decir, muy cercano a cero pero no cero, entonces el exponente debe ser un valor negativo grande. Para el número mas chico distinto de cero, el exponente debe ser $-E_{\max}$.

El valor cero, por estar muy cercano al anterior, también debería tener como exponente al valor $-E_{\max}$.

De esta forma la representación del cero quedaría: $0.0 \times 2^{-E_{\max}}$

Por otra parte, es deseable que la representación del cero en punto flotante coincida con la misma representación en punto fijo.

La forma de compatibilizar los dos puntos anteriores es asignar la combinación 000...0 al valor $-E_{max}$.

Esto equivale a adoptar para la representación del exponente un código en exceso E_{max} ($X - E_{max}$), en el que cada representación de exponente se obtiene de sumarle E_{max} al valor del verdadero exponente.

De esta manera el valor $-E_{max}$ se representa por 0 y el exponente 0 se representa por $+E_{max}$.

NORMA IEEE 754

A partir de 1985 se ha adoptado como estándar un formato para representación de números en punto flotante, auspiciado por el *Institute of Electrical and Electronics Engineers* (IEEE, pronunciado: "i triple e"), conocido como **Norma IEEE 754**.

Según esta norma, se adoptan los siguientes criterios:

- El formato es el mostrado anteriormente para flotantes de 32 bits.
- La mantisa se representa en la convención de signo-magnitud.
- Se asigna el bit más significativo para representar al signo de la mantisa.
- Se asignan 8 bits para el exponente.
- Se asignan 23 bits para la mantisa.
- La mantisa ubica el *msb* justo a la izquierda del punto decimal.
- El *msb* (bit más significativo) adopta la forma de bit oculto.
- El exponente es un número de 8 bits signado en complemento a 2.
- El exponente se representa en el código "Exceso 127" ($X-127$).
- El exponente original -128 (equivalente a 1111111 en $X-127$) se reserva.

Un **número flotante** representado en esta norma se puede calcular según la expresión:

$$N = (-1)^S \cdot 2^{E-127} \cdot M$$

EXONENTE RESERVADO

El exponente más negativo (-128) se representa en $X-127$ como -1, es decir:

$$E = 11111111$$

Este valor se reserva para denotar números "infinitos" que se producen con ciertos tipos de desborde hacia infinito, como así también para la representación de ciertas combinaciones inválidas de bits, consideradas como "no numéricas".

Estas combinaciones se denotan con la expresión NaN (*Not a Number*).

EJEMPLOS

A continuación veremos algunos ejemplos de números en punto flotante expresados como signo, exponente y mantisa, en la forma en que se almacenan en memoria, junto a sus correspondientes valores:

0 00000000 0000000000000000000000000000	+0
1 00000000 0000000000000000000000000000	-0
1 01111111 0000000000000000000000000000	-1
0 01111111 1111111111111111111111111111	$2 - 2^{23} \cong 2$
0 10000001 1110000000000000000000000000	+7.5
0 11111111 1011101010001011101011	NaN
0 11111111 0001110101100101001100	Error (INF)

EJEMPLO: GENERAR Y OBSERVAR FORMATOS DE PUNTO FLOTANTE

Este programa permite cargar los 4 campos `unsigned char` de un vector superpuesto a un flotante mediante una unión.

Posteriormente se muestra el flotante así generado.

En la segunda parte, se pide el ingreso por teclado de un flotante y se muestran posteriormente, los cuatro bytes que lo forman en formato hexadecimal.

```
#include <stdio.h>
int main()
{
    union {
        unsigned char VEC[4] ;
        float          F      ;   } X ;

    /* Primera Parte */
    X.VEC[3] = 0X45 ;
    X.VEC[2] = 0XEF ;
    X.VEC[1] = 0X9C ;
    X.VEC[0] = 0X58 ;
    printf("Para los valores cargados\n");
    printf("\n\nFlotante = %f " , X.F ) ;

    /* Segunda Parte */
    printf ("\n\nAhora ingrese un flotante\n\n");
    scanf ( "%f" , &X.F ) ;
    printf("\n%02X %02X %02X %02X", X.VEC[3], \
           X.VEC[2], X.VEC[1], X.VEC[0]);
    printf("\n\n Fin del programa");
}
```

Nótese que para ciertas combinaciones, la impresión del flotante es directamente “[INF” o “NaN”.

EJEMPLO DE CÁLCULO

En esta sección se muestra cómo realizar el cálculo de un número flotante para expresarlo en el formato IEEE 754.

Consideremos el número: 7667.54321

Transformemos a binario su parte entera:

$$7667 \text{ })_{10} = 1110111110011 \text{ })_2$$

y luego su parte decimal:

$$.54321 \text{ })_{10} = .100010110000111111 \dots \text{ })_2$$

por lo que el número expresado en binario toma la forma:

$$1110111110011.100010110000111111 \dots$$

Ahora podemos colocar el punto decimal inmediatamente a la derecha del bit más significativo, ajustando el valor por medio de 2^E .

$$1.110111110011100010110000111111 \dots \times 2^{12}$$

Extrayendo el bit oculto, queda entonces la mantisa como:

$$M = 110111110011100010110000111111 \dots$$

Pero esta mantisa excede los 23 bits de longitud. Es necesario, por lo tanto, realizar un “recorte” en el mencionado número, quedando la mantisa como :

MANTISA	1 1 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1 1 0 0 0
---------	---

POLARIZACIÓN DEL EXPONENTE

Como se vio, el exponente vale 12. Lo que en formato binario de 8 bits se representa como: 00001100

Pero es necesario transformar este número a la codificación X-127, por lo tanto le sumamos 127:

$$00001100 + 01111111 = 10001011$$

EXPONENTE	1 0 0 0 1 0 1 1
-----------	-------------------------------

Dado que el número es positivo, el signo será cero, por lo que finalmente tenemos lo siguiente:

0	1 0 0 0 1 0 1 1	1 1 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1 1 0 0 0
---	-------------------------------	---

Se puede chequear el resultado de este análisis mediante el ejemplo, observando que la impresión de las componentes del flotante dan:

$$7667.54321 \rightarrow 45_{\text{H}} \text{ EF}_{\text{H}} \text{ 9C}_{\text{H}} \text{ 58}_{\text{H}}$$

Sin embargo, la impresión como flotante del mencionado número da como resultado:

$$45_{\text{H}} \text{ EF}_{\text{H}} \text{ 9C}_{\text{H}} \text{ 58}_{\text{H}} \rightarrow 7667.542969$$

Esto puede comprobarse mediante un programa más sencillo aun:

```
float F ;  
F = 7667.54321 ;  
printf("\n\n7667.54321 = %f " , F ) ;
```

El resultado de la ejecución es:

$$7667.54321 = 7667.542969$$

Esto es debido a que la mantisa no pudo contener *todos los bits* que generó nuestro cálculo de la parte fraccionaria del número original.

Es decir, la **precisión** del número de punto flotante no es infinita y está determinada por la mantisa.

PRECISIÓN DE LOS NÚMEROS EN PUNTO FLOTANTE

Como se mencionó anteriormente la precisión de estos números depende de la mantisa. Ésta tiene 23 bits de longitud. Considerando el bit oculto, se disponen de 24 bits para representar la parte real del número.

Se obtiene, entonces, el valor máximo de $2^{24} \approx 10^7$.

Esto significa que números decimales con más de 7 cifras (ya sean enteras o fraccionarias) sufrirán una aproximación *por recorte* al ser representados en el formato de punto flotante descripto.

Los números de punto flotante aumentan su rango de representación a costa de una pérdida de precisión en algunos casos.

REPRESENTACIÓN EN LA RECTA NUMÉRICA

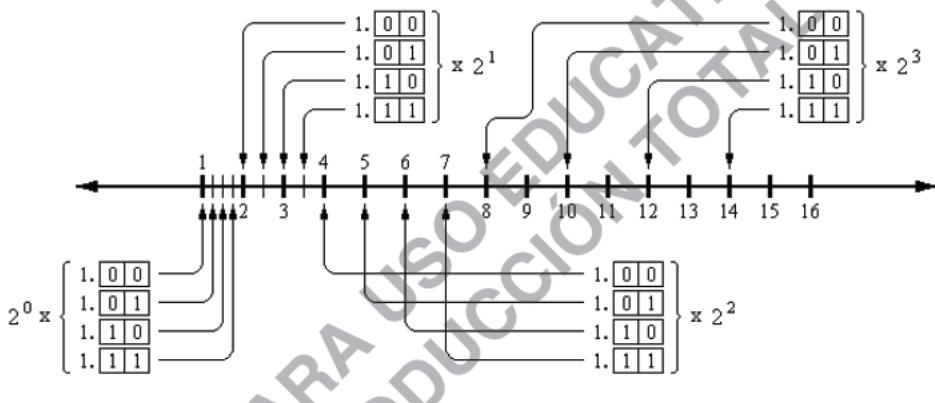
Es importante no perder de vista el hecho de que los números de punto flotante de 32 bits, si bien tienen un amplio rango numérico, no pueden representar más de 2^{32} valores distintos.

Es decir que no pueden representar más números que un `long int`.

Lo que cambia, en el caso de los números de punto flotante, es la distribución de estas representaciones.

Consideremos un formato de punto flotante cuya mantisa esté compuesta por dos bits más el bit oculto.

La representación de los números de 1 a 15 se muestra en la siguiente figura:



Se está representando el rango $[2^0, 2^4]$.

Se pueden apreciar del diagrama las siguientes características:

- Para cada valor de exponente existen 2^M representaciones (siendo M la cantidad de bits de la mantisa).
- Para cada valor de exponente, la densidad de representación se mantiene constante.
- La densidad de representación cambia si cambia el exponente.

La cantidad de **combinaciones** de la mantisa es constante, pero el rango de representación se duplica en longitud por cada incremento del exponente. Por lo tanto la densidad se reduce a la mitad y la separación entre valores pasa al doble.

Nótese que para $E = 0$ la separación entre representaciones consecutivas es de $\frac{1}{2}$, mientras que para $E = 2$ dicha separación vale 2.

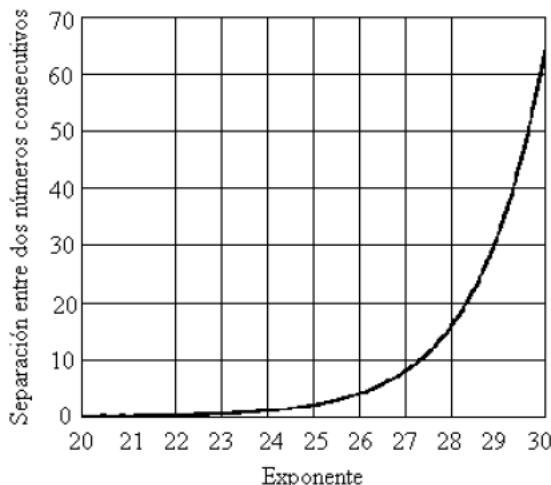
En el caso de los números de punto flotante de 4 bytes con mantisa de 23 bits, se tendrán 2^{23} combinaciones por cada valor de exponente.

El próximo gráfico muestra la separación entre valores consecutivos en función del exponente, en el rango $E = [20, 30]$, para una mantisa de 24 bits.

Considerando que M es el número de bits de la mantisa, en el intervalo correspondiente a $E = 0$ es posible representar 2^M números igualmente espaciados y separados por una distancia $1/2^M$.

La distancia D en el intervalo $[2^E, 2^{E+1}]$ es igual a $(2^{E+1} - 2^E)$.

$$D = 2^{E+1} - 2^E = 2^E \times 2^1 - 2^E = 2^E(2^1 - 1) = 2^E$$



Por lo tanto, en cualquier intervalo de E , de extensión 2^E , hay 2^M números equiespaciados, por lo que la separación entre valores consecutivos δ para cada intervalo es $2^E / 2^M$, o lo que es lo mismo, $\delta = 2^{(E-M)}$.

$$\text{Separación entre valores consecutivos} = 2^{(E-M)}$$

Por ejemplo, entre 2^{27} y 2^{28} hay (para $M = 24$) 2^{24} números, por lo que el espacio entre dos números sucesivos es de $2^{(27-24)} = 2^3 = 8$ valores.

Puede cotejarse este valor con el obtenido del gráfico anterior.

ERROR ABSOLUTO Y RELATIVO EN PUNTO FIJO

En los números de **punto fijo**, la separación entre valores consecutivos es constante. En caso de intentar representar un número ubicado entre dos consecutivos de ellos, el **error** estará acotado a dicha separación.

En el caso de los enteros, considerando un redondeo por truncamiento, el **error máximo** será menor que 1.

Si N es el número a representar y N_R es el verdadero valor representado, el **error absoluto** ϵ cometido se puede expresar como:

$$\epsilon = |N - N_R|$$

Y dado que δ es la distancia entre dos representaciones consecutivas, el **error absoluto** estará acotado a:

$$\epsilon < |\delta|$$

El **error relativo** al valor a representar es muy grande para números pequeños, pero disminuye a medida que el valor a representar crece.

$$\epsilon_R = |N - N_R| / N \quad \epsilon_R < |\delta / N|$$

Dado que δ es constante, el valor de ϵ_R aumenta en forma inversa al valor a representar.

ERROR ABSOLUTO Y RELATIVO EN PUNTO FLOTANTE

En el caso de **punto flotante**, como se vio, la separación entre números consecutivos aumenta con el exponente del número: $\delta = 2^{(E-M)}$

Donde M es la cantidad constante de bits que forman la mantisa.

Si consideramos (como ya se vio), que el redondeo de los valores se produce por truncamiento de los bits excedentes de la mantisa, entonces:

$$|N - N_R| < \delta \quad \text{y por lo tanto} \quad \epsilon < \delta$$

Esto significa que el **error absoluto** aumenta exponencialmente con el valor del número representado.

El error relativo ϵ_R puede representarse como:

$$\epsilon_R = |N - N_R| / N \quad \epsilon_R < |\delta / N|$$

Donde el número a representar N vale $N = 1.M \times 2^E$. Reemplazando en la expresión anterior:

$$\epsilon_R < |\delta / 1.M \times 2^E|$$

Dado que $1.M$ es un valor comprendido entre 1 y 2 ($1 \leq 1.M < 2$), podemos simplificar la expresión anterior, quedando:

$$\epsilon_R < |\delta / 2^E|$$

Reemplazando $\delta = 2^{(E-M)}$:

$$\epsilon_R < |2^{(E-M)} / 2^E|$$

De donde surge:

$$\epsilon_R < 2^{(-M)}$$

Esto significa que el **error relativo** se mantiene aproximadamente **constante**, pero siempre acotado al valor dado por la expresión anterior, determinada por la longitud de la mantisa.

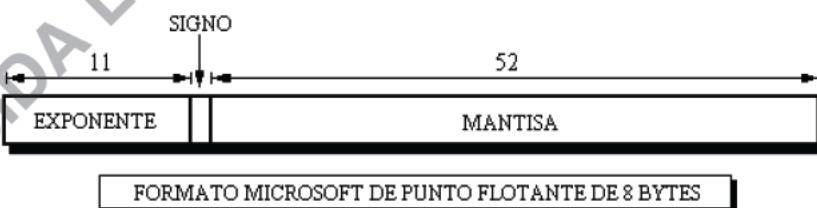
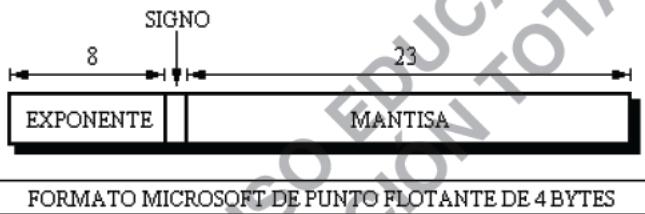
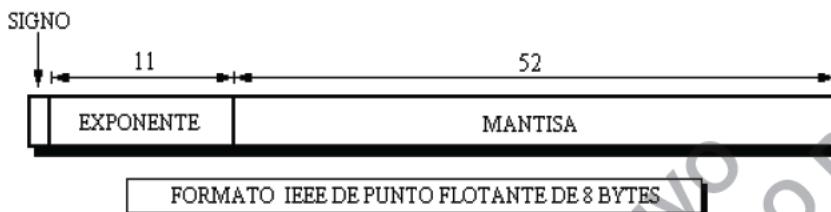
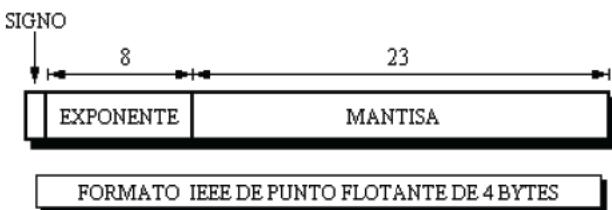
Por lo tanto, si se desea reducir el error relativo ϵ_R será necesario aumentar la cantidad de bits M de la mantisa.

DISTINTOS FORMATOS DE PUNTO FLOTANTE

El formato que hemos estudiado no es el único. Existen otros formatos en los que se busca aumentar la precisión aumentando la cantidad de bits de la mantisa y extender el rango aumentando la cantidad de bits del exponente.

En todos los casos, la mantisa se presenta en convención de signo-magnitud con normalización de bit oculto. Este último no se muestra, salvo en el caso del formato IEEE de 10 bytes.

A continuación se muestran diversos formatos:



13. COMANDOS DEL PREPROCESADOR

El lenguaje C permite incluir en el programa fuente comandos destinados al compilador.

Estos comandos deben ser tenidos en cuenta, y llevados a cabo, antes del proceso de compilación. Se trata, por lo tanto, de un proceso *previo* a la compilación, es decir, de un *pre-proceso*.

Estos comandos están incorporados al **programa fuente**, aunque no son sentencias del lenguaje. No estarán presentes en el **programa objeto** ni en el **ejecutable** pues no son sentencias ejecutables.

Cuando el compilador inicia su tarea, busca en primer término estos comandos. Para reconocerlos, todos ellos comienzan con el símbolo numeral (#).

El estándar ANSI define los siguientes comandos del preprocesador:

- #define
- #undef
- #include
- #if
- #else
- #elif
- #endif
- #ifdef
- #ifndef
- #error
- #line
- #pragma

#DEFINE

El #define es una orden de reemplazo por la cual el compilador busca en el programa fuente una etiqueta y la sustituye por una cadena de caracteres, todas las veces que dicha etiqueta sea encontrada.

Existe una excepción a lo expuesto anteriormente: si la etiqueta se encuentra entre comillas, o formando parte de una expresión encerrada por comillas, el compilador supone que se desea que la etiqueta aparezca literalmente como está y no realiza el reemplazo.

Normalmente la etiqueta recibe el nombre de **macro**.

El formato de este comando es:

```
#define etiqueta cadena_de_caracteres
```

La separación entre la etiqueta y la cadena es el espacio (uno o varios). El final de la cadena está determinado por el cambio de línea.

Si una línea no fuera suficiente para contener la cadena, se puede realizar un cambio de línea utilizando el carácter de escape *contrabarra* (\) precediendo al Enter. De esta forma el compilador lo ignorará, teniendo efecto el salto de línea solamente en la presentación del programa fuente.

A continuación se muestra un ejemplo de lo expuesto:

```
#define nombre_de_macro cadena demasiado larga para que \
pueda ser ubicada en una sola linea.
```

USOS DE #DEFINE

Podemos clasificar las aplicaciones del `#define` de la siguiente manera :

- Reemplazo de expresiones
- Definición de constantes
- Declaración de *macros*

REEMPLAZO DE EXPRESIONES

Si bien el `#define` es en sí un reemplazo, podemos atribuirle en la definición de constantes y en la declaración de macros un significado más elaborado que analizaremos posteriormente.

La aplicación del `#define` en el reemplazo de expresiones aporta la ventaja de un programa fuente más ordenado y más sencillo de leer, al estar los títulos y otras expresiones definidos aparte.

EJEMPLO: REEMPLAZO DE ETIQUETA

Se muestra aquí el funcionamiento del `#define` en el reemplazo de la etiqueta TITULO.

```
#include <stdio.h>
#define TITULO "Este es el Titulo"
int main ( )
{
    printf ( "TITULO" ) ;
    printf ( "\n \n" ) ;
    printf ( TITULO ) ;
}
```

La ejecución de este programa da como resultado la impresión de:

TITULO

Este es el Titulo

Esto muestra que la palabra TITULO entre comillas dobles fue ignorada para los reemplazos, pero no así cuando *no* estaba encerrada por comillas.

EJEMPLO: REEMPLAZOS CON FORMATO

En el siguiente ejemplo, los reemplazos son más elaborados, incluyendo expresiones de formato como tabulación y cambios de línea.

La ejecución dará como resultado una impresión centrada del primer título y el subtítulo algunos renglones más abajo, sobre el margen izquierdo.

EJEMPLO: REEMPLAZO INCLUYENDO FUNCIONES

Este ejemplo es una variante del anterior. En este caso el reemplazo se realiza con la invocación de la función `printf()` completa. El resultado de la ejecución es idéntico al del ejemplo anterior. No podía ser de otra manera dado que los programas objeto y ejecutable también son iguales, pues luego de los reemplazos en el programa fuente antes de la compilación no hay diferencia entre ellos.

En este ejemplo se finaliza el `nombre_de_macro` con un ; . Esto no sería necesario si al *punto y coma* lo colocamos dentro de la cadena de reemplazo.

Como veremos posteriormente, este ejemplo deberíamos clasificarlo en la categoría de **macros**.

```
#include <stdio.h>

#define TITULO1 printf("\n\n\n\t\t\t ORGANIZACION DE EMPRESAS\n\n")
#define TITULO2 printf("PRINCIPALES CLIENTES\n\n")
```

```
int main ( )
{
    TITULO1 ;
    TITULO2 ;
}
```

DEFINICIÓN DE CONSTANTES

Una **constante** es un valor que permanece inalterado durante toda la ejecución del programa.

Cuando colocamos un valor numérico (o de otra índole) formando parte del texto del programa fuente, le estamos dando a ese valor status de constante, dado que no es posible cambiarlo.

Por ejemplo:

```
CIRCULO = 3.1416 * DIAMETRO ;
```

En este caso, 3.1416 es un valor constante, también llamado **valor literal**.

Recordemos que ningún valor constante puede estar a la izquierda de una **asignación**. Dicho de otra forma, no podrá haber una constante a la izquierda de un signo = (no confundir con el operador de **comparación** ==) pues si no el proceso de compilación arrojará un error de *left value*.

Podemos tener gran cantidad de estos valores en nuestro programa. En el caso del 3.1416 se intuye fácilmente que estamos haciendo referencia al valor de π . Pero en el caso de otros valores podría no estar tan claro su significado, dificultando la comprensión del programa fuente.

Puede darse el caso de que una **constante del programa** deba ser modificada (obviamente, no durante la ejecución del programa) en alguna ocasión.

Será necesario ubicar todos y cada uno de los valores en el programa fuente, realizar el cambio y compilar nuevamente.

Esto trae aparejado algunas fuentes de error. En primer término, podemos omitir erróneamente el reemplazo en alguno de los valores. Por otra parte, durante el proceso de reemplazo, podríamos cambiar un valor numérico idéntico al que buscamos pero que no tenga relación con éste. En este caso estamos modificando, por error, un valor que debía permanecer inalterado.

Podemos poner como ejemplo un programa que incluya el IVA en gran cantidad de líneas, y en algún momento el Ministerio de Economía decrete que su valor cambie.

Al realizar la actualización del programa, podríamos omitir el cambio del número en algún lugar y realizar este cambio donde no corresponda.

Es conveniente en estos casos realizar una **parametrización** de estos valores referenciándolos por un identificador, y que dicho identificador sea el que los represente dentro del programa.

De esta manera, si se debe realizar una actualización del programa como la antes expuesta, será muy fácil hacerla modificando la asignación del valor al identificador una sola vez en todo el programa. Por otra parte, así estamos evitando los riesgos de algún reemplazo erróneo.

La utilización de **identificadores** clarifica el programa fuente y facilita su comprensión. Es recomendable seleccionar el nombre de estos identificadores de manera que se los relacione fácilmente con los valores que representan.

Es evidente que podríamos utilizar una variable para parametrizar valores en un programa, con la intención de no modificar su contenido y de esta manera darle el estatus de constante.

Este recurso se muestra a continuación:

```
float CIRCULO , DIAMETRO , PI ;
PI = 3.1416 ;
.....
CIRCULO = PI * DIAMETRO ;
```

Sin embargo, si por error se modificara el contenido de dicha variable durante la ejecución del programa, esto no constituiría ninguna violación para el compilador y el error no sería detectado. Constituiría entonces un **error en tiempo de ejecución**.

Estos errores son los más difíciles de detectar y corregir y pueden pasar desapercibidos (lo que es aún peor) durante años.

Para evitar el riesgo anterior, el lenguaje C nos permite utilizar el modificador **const** en la declaración de la variable.

Una variable declarada con el modificador **const** no podrá ser asignada durante el programa, salvo en el momento de la declaración (nuestra única oportunidad), y por lo tanto se comportará como una **constante**. Cualquier intento de modificación posterior generará un error de compilación.

El ejemplo anterior quedaría como sigue:

```
float CIRCULO , DIAMETRO ;
const float PI = 3.1416 ; /* Constante */
.....
CIRCULO = PI * DIAMETRO ;
.....
PI = 3.14 ; /* Esta línea generará un error */
```

USO DE #DEFINE EN DECLARACIÓN DE CONSTANTES

Una tercera forma de declarar constantes es utilizando `#define`. El ejemplo anterior quedaría de la siguiente manera :

```
#define PI 3.1416
.....
float CIRCULO , DIAMETRO ;
.....
CIRCULO = PI * DIAMETRO ;
.....
PI = 3.14 ; /* Esta linea dará error */
```

Recordemos que en este caso, el identificador `PI` no se encontrará en los programas objeto y ejecutable sino que en su lugar estará el valor `3.1416`. Este reemplazo se produjo *antes* del proceso de compilación.

También se debe notar que utilizando `#define` no se declara una variable en la memoria y, por lo tanto, se ahorra dicha memoria el área de datos. Esto es, no se crea la variable-constante `PI` del anterior ejemplo (con `const`) y por lo tanto no se ocupan los bytes correspondientes a ella.

La línea de asignación `PI = 3.14 ;` generará un error de compilación pues cuando esto ocurra, lo que el compilador encontrará allí será:

```
3.1416 = 3.14 ;
```

Es decir, tenemos un valor constante a la izquierda de una asignación. Esta situación generará un evidente error de compilación.

MACROS

Para comprender el concepto de **macro** debemos remontarnos a sus orígenes en el lenguaje Assembler.

En este lenguaje las sentencias son muy rudimentarias y poco poderosas pues reflejan órdenes directas al microprocesador. De esta manera los programas son muy laboriosos de construir, aunque se obtienen los mejores resultados en cuanto a velocidad de ejecución y aprovechamiento de los recursos del sistema.

Si se necesitara una aplicación no representada por las instrucciones del procesador habría que construirla mediante programación.

Si esa aplicación se repitiera a lo largo del programa, las instrucciones que la representan estarían repetidas a lo largo de éste como se muestra en el siguiente esquema:



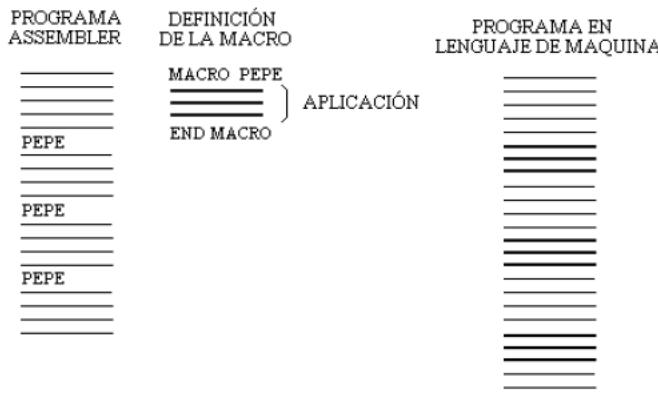
Una forma de encarar esta situación es mediante la utilización de **subrutinas** (concepto equivalente al de **funciones** del lenguaje C), con la consiguiente problemática de utilización de la pila para la salvaguarda de la dirección de retorno y transferencia de argumentos.

Una variante a la solución del problema la dio la posibilidad de definir *aparte* a estos grupos de instrucciones que representan la aplicación, referenciándolos dentro del programa mediante un identificador asignado.

Antes del proceso de traducción de Assembler al lenguaje de máquina (proceso conocido como **ensamblado**) se reemplazaban los identificadores por los grupos de instrucciones representados por ellos. De esta manera el programa traducido al lenguaje de máquina quedaba exactamente igual que en el caso de no utilizar este recurso. La ventaja se observa únicamente en el programa fuente, el cual resulta más corto y más comprensible.

La presencia del identificador dentro del programa fuente se ve como una instrucción grande (porque involucra varias instrucciones en lenguaje de máquina) o **macro-instrucción**, de donde deriva el término **macro**.

La situación está representada en el esquema:



Una ventaja adicional en la utilización de macros es la posibilidad de crear **bibliotecas de macros**. Si estas bibliotecas de macros se incorporan al lenguaje Assembler como propias de él, entonces algunas instrucciones del lenguaje se corresponderán con más de una instrucción en lenguaje de máquina y el nivel del lenguaje en cuestión se elevará, desembocando posteriormente en los lenguajes de alto nivel.

USO DE #DEFINE PARA DECLARAR MACROS

Como vimos anteriormente, `#define` produce un reemplazo dentro del programa fuente.

Dado que una macro constituye en sí misma un programa, será necesario transferirle argumentos en forma parecida a como se lo hace con las funciones.

El lenguaje C prevé esta situación y permite al compilador tener un criterio amplio en el proceso de identificación del nombre de macro, y realizar un “ajuste” al producir el reemplazo.

El agregado de paréntesis a continuación del nombre de macro alerta sobre esta situación.

Como ejemplo crearemos una macro destinada a mostrar en pantalla un valor entero proveniente de una variable.

```
#define imprimir(X) printf("El valor es %d", X)
```

Dentro del programa podremos invocar a esta macro con variables diferentes de X e inclusive con expresiones constantes o que involucren operadores, como se muestra a continuación:

```
int A, B ;  
A = 2 ; B = 4 ;  
imprimir ( A ) ;  
imprimir ( B ) ;  
imprimir ( 5 ) ;  
imprimir ( A + B - 3 ) ;
```

El compilador detecta el formato de “*imprimir (alguna cosa)*” y produce el reemplazo determinado por el `#define` considerando que en lugar de X deberá colocar *alguna cosa*.

De este modo, inmediatamente antes del proceso de compilación, y después del reemplazo, las líneas anteriores tendrán la forma:

```
int A, B ;  
A = 2 ; B = 4 ;  
printf ("El valor es %d", A ) ;  
printf ("El valor es %d", B ) ;
```

```

printf ("El valor es %d" , 5 ) ;
printf ("El valor es %d" , A + B - 3 ) ;

```

Dado que la macro representa un reemplazo **literal** de la expresión, donde los argumentos también se “transfieren” textualmente, es necesario tomar algunas precauciones, teniendo en cuenta situaciones particulares.

A continuación se muestran algunos ejemplos que ilustran estos llamados de atención:

EJEMPLO: MACRO DEFECTUOSA I

```

#include <stdio.h>
#define cuad(X) X * X
int main ( )
{
    int Z ;
    Z = 5 ;
    printf ( "\n%d" , cuad ( 2 ) ) ;
    printf ( "\n%d" , cuad ( Z ) ) ;
    printf ( "\n%d" , cuad ( Z + 3 ) ) ;
}

```

En este ejemplo se define la macro `cuad()` como `X*X`. Esperamos entonces que los valores en pantalla sean los cuadrados de los valores ingresados.

Es decir, esperamos ver:

4 25 64

Sin embargo los valores mostrados son:

4 25 23

Tenemos una discrepancia en `cuad (Z + 3)`. ¿Cómo puede ser?

Esperamos que se calcule $(Z + 3)^2$, o sea $(5 + 2)^2$, dando como resultado $8^2 = 64$. Sin embargo el resultado fue 23.

El razonamiento anterior es aplicable a las **funciones**, en las que si se coloca una expresión como argumento, primero se evalúa dicha expresión y posteriormente se transfiere el valor resultante de la misma a la función.

En el caso de la **macro**, lo que se realiza es un reemplazo textual, por lo que la expresión `cuad (Z + 3)` se reemplazará por `Z + 3 * Z + 3`.

La evaluación de la expresión anterior arroja como resultado:

$$5 + 3 * 5 + 3 = 5 + 15 + 3 = 23$$

Es necesario tener presente esta situación al definir la macro, colocando los paréntesis que aseguren la evaluación correcta:

```
#define cuad(X) (X) * (X)
```

Obsérvese que en la declaración de la macro, el paréntesis de apertura debe ir exactamente a continuación del nombre de macro. Si se deja un espacio, éste será considerado como separador entre el nombre de macro y la cadena, generando posteriormente reemplazos erróneos.

Esta situación no es tan estricta al buscar el identificador de la macro en el programa fuente, dado que reconoce la etiqueta aún con espacios delante del paréntesis de apertura, como se muestra en los ejemplos.

EJEMPLO: MACRO DEFECTUOSA II

Este ejemplo es similar al anterior. En este caso se tuvo la precaución de colocar paréntesis individuales a cada una de las variables de la cadena de reemplazo.

```
#include <stdio.h>
#define lineal(M,X,N) ( M ) * ( X ) + ( N )

int main ( )
{
    int A , B , Z ;
    A = 2 ; Z = 3 ; B = 5 ;
    printf ( " \n%d " , lineal ( 2 , 3 , 4 ) ) ;
    printf ( " \n%d " , lineal ( A , Z , B ) ) ;
    printf ( " \n%d " , 2 * lineal ( A , Z , B ) ) ;
}
```

Sin embargo, uno de los valores resultantes difiere del esperado.

Se esperaban los valores: 10 11 22

Se observaron los valores: 10 11 17

Observemos la expresión `2 * lineal (A , Z , B)` luego del reemplazo. Esta adoptará la siguiente forma: `2 * (A) * (Z) + (B)` es decir:

$$2 * 2 * 3 + 5 = 17$$

La correcta definición de la macro hubiera sido:

```
#define lineal(M,X,N) ( (M) * (X) + (N) )
```

EJEMPLO: MACRO CON OPERADOR TERNARIO (?)

En este ejemplo se implementa una macro que retorna el valor mínimo entre dos valores recibidos utilizando el operador ternario ? .

```
#include <stdio.h>
#define MIN(A,B) ( ( A ) ) < ( ( B ) ) ? ( A ) : ( B )
int main()
{
    int X , Y ;
    printf ( " Ingrese un par de valores enteros \n\n" ) ;
    scanf( "%d %d" , &X , &Y ) ;
    printf ( " \n\n El valor minimo es %d " , MIN( X , Y ) ) ;
}
```

EJEMPLO: MACRO DE IMPRESIÓN DE UN VECTOR

La macro definida en este ejemplo muestra en pantalla un vector de long int valores con formato entero. Obsérvese la contrabarra \' que indica el cambio de línea precediendo al Enter.

La utilización de la palabra reservada long no trae aparejado ningún inconveniente pues no estará presente en el programa cuando se realice la compilación, dado que en su lugar estará el número 10 .

```
#include <stdio.h>

#define MAX 10
#define impri(V,long) { int I ; for ( I=0 ; I<long ; I++ ) \
                    printf ( " \n\t%d " , V[I] ) ; }

int main ( )
{
    int VEC[MAX] , I ;
    for ( I=0 ; I<MAX ; I++ ) VEC[I] = I ;
    impri ( VEC , MAX ) ;
}
```

EJEMPLO: ORDENAMIENTO DE UN VECTOR USANDO MACROS

Ampliando el ejemplo anterior se incluyen en éste dos macros adicionales destinadas a realizar el ingreso del vector por teclado, y un ordenamiento en orden creciente utilizando el método de burbujeo.

Nótese que las variables declaradas en cada macro son locales a los bloques de cada una de ellas. Por esta razón no interfieren con el programa ni sus variables, y las macros pueden insertarse en cualquier lugar.

El programa resultante en el que se ingresa un vector de MAX valores y se lo imprime ordenado, resulta extremadamente sencillo y corto, debido a que su complejidad queda absorbida por las macros.

Esto sugiere la posibilidad de construir una **biblioteca de macros** propia a fin de utilizarlas en forma similar a las funciones del C.

De hecho, muchas de las funciones que empleamos, proporcionadas por C, son en realidad macros.

```
#include <stdio.h>
#define MAX 10
#define imprimir(V,long) { int I ; for ( I=0 ; I<long ; I++ ) \
                           printf (" \n \t %d ", V[I] ) ; }

#define leer(V,long)      { int I ; for( I=0 ; I<long ; I++) { \
                           printf("Ingrese el elemento %d = ",I); \
                           scanf("%d" , &V[I] ) ; } }

#define ordenar(V,long)   { int I , J , AUX ; \
                           for ( I=0 ; I<long-1 ; I++ ) \
                           for ( J=0 ; J<long-I-1 ; J++ ) \
                           if ( V[J] > V[J+1] ) { \
                               AUX      = V[J] ; \
                               V[J]    = V[J+1] ; \
                               V[J+1] = AUX ; } }

int main ( )
{
    int VEC[MAX] ;
    leer ( VEC , MAX ) ;
    ordenar ( VEC , MAX ) ;
    imprimir ( VEC , MAX ) ;
}
```

COMPARACIÓN ENTRE MACROS Y FUNCIONES

La utilización de las macros y las funciones es similar. Como se mencionó anteriormente, muchas de las funciones provistas por C que hemos utilizado, son en realidad macros.

Sin embargo, el comportamiento de las macros y las funciones es diferente.

En el caso de las macros se produce un reemplazo cada vez que éstas son invocadas, por lo que están presentes en su totalidad en el programa ejecutable. Si una macro es invocada n veces, ocupará en el programa compilado n veces su tamaño.

En cambio, una función sólo ocupa el lugar donde está declarada (es decir una vez su tamaño), y se realiza un *salto* en el programa cada vez que es invocada.

Esto significa que la utilización de funciones conduce a programas compilados más reducidos que en el caso de la utilización de macros. Tanto más cuanto mayor sea la cantidad de invocaciones.

Por otra parte, el acceso a las funciones trae aparejado los problemas de salvaguardia de la dirección de retorno y transferencia de argumentos. Para poder implementar funciones anidadas se hace necesario el acceso a una memoria LIFO, que se resuelve mediante la utilización de una pila (*stack*).

Cada salto a función representa accesos a la pila para guardar la dirección de retorno, crear y cargar los parámetros formales, crear y utilizar las variables locales, y finalmente recuperar la dirección de retorno para regresar al programa invocante.

Considerando que el acceso a la pila es indirecto pues se realiza a través del contenido del *stack pointer*, podemos decir que el proceso será mucho más lento que si la ejecución fuera lineal.

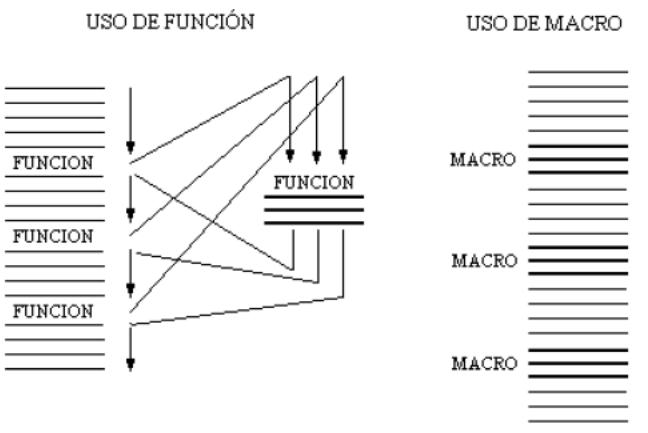
Este es el caso cuando se utilizan macros. No hay necesidad de rupturas en la secuencia del programa (saltos) pues las macros fueron insertadas en él.

Podemos afirmar entonces que la utilización de macros conduce a programas de ejecución mas rápida que en aquellos en que se utilicen funciones.

Se seleccionará, de ser posible, la utilización de macros o funciones, teniendo en cuenta las ventajas y desventajas anteriores.

El siguiente esquema muestra lo descripto anteriormente. Compárese la longitud relativa de ambos programas y nótese la cantidad de saltos en un caso, contra la linealidad de ejecución en el otro.

PROGRAMAS EN LENGUAJE DE MAQUINA



#UNDEF

Los *nombres_de_macro* definidos por `#define`, además de ser buscados y reemplazados por la cadena correspondiente, adoptan el estatus de *nombre_de_macro definida*.

El reemplazo de un `#define` por su cadena, tiene lugar a partir de la ubicación del mismo, y es posible cambiar la cadena de cambio con otro `#define`. Esto tiene efecto a partir de la ubicación del nuevo `#define`.

Asimismo, es posible devolverle el status de *nombre_de_macro no definida* a una macro, utilizando el comando `#undef` para *desdefinirla*. Esto tendrá efecto a partir de la ubicación del `#undef`.

Esta situación se mostrará con el siguiente ejemplo.

EJEMPLO: USO DE #UNDEF

```
#include <stdio.h>
#define MACRO 10
int main ( )
{
    printf ( " \t \t %d " , MACRO ) ;
#define MACRO 20
    printf ( " \t \t %d " , MACRO ) ;
#undef MACRO
    printf ( " \t \t %d " , MACRO ) ; /* ERROR */
}
```

Al intentar compilar el programa anterior surge un error debido a “*símbolo MACRO no definido en función main*” en la línea que se marca.

Esto significa que a partir del comando `#undef MACRO`, dicho *nombre_de_macro* adoptó el status de *no definido*.

Al eliminar dicha línea el programa se compila con un *warning* que indica que “*la redefinición de MACRO no es idéntica*”, es decir que MACRO fue redefinida con otro valor.

La ejecución del programa compilado, como era de esperar, arrojó como resultado:

10	20
----	----

#INCLUDE

El comando `#include` hace que el compilador agregue al programa otro archivo fuente y lo compile. En otras palabras, es como si lo copiara y lo pegara.

El archivo a agregar debe ser de **texto**, a fin de ser reconocido posteriormente por el compilador.

El nombre del archivo a incluir debe estar encerrado por comillas dobles ("") o por signos mayor/menor (< >).

Sintaxis del comando:

```
#include <nombre_de_archivo>
#include "nombre_de_archivo"
```

Las buenas prácticas indican que los archivos a incluir deben tener la extensión .**h**. Esta extensión deriva de la palabra *header* o **cabecera**. Nos referimos a ellos indistintamente como *cabeceras*, *archivos cabecera* ó *headers*.

Si se utilizan signos < > para la invocación del archivo a incluir, el compilador lo buscará en la carpeta que tenga configurada para las *cabeceras* y en otras especificados en la línea de comandos. No lo buscará en el directorio actual.

Si el archivo a incluir se encuentra en cualquier otro lado, se deberá utilizar la variante de comillas dobles. En este caso se busca en primera instancia en el directorio actual, y si allí no se encuentra se busca en los directorios definidos en el *path* (variable de entorno del sistema operativo).

Si se utiliza una ruta de acceso completa (*path*) para el archivo, sólo se buscará en ese directorio.

Ejemplos de inclusión:

#include <stdio.h>	Busca en los directorios default
#include "vectores.h"	Busca en directorio actual
#include "C:\HEADERS\head1.h"	Busca en C:\HEADERS

Los archivos cabecera pueden incluir comandos al preprocesador, definiciones de macros, constantes, declaraciones de tipos de datos, e inclusive partes de programa o funciones destinadas a ser transformadas en código por el compilador.

CABECERAS PROPIAS

Los **#include** nos dan la posibilidad de incorporar fácilmente texto preexistente a nuestros programas. Recordemos que la opción “*copiar y pegar*” no siempre estuvo disponible.

De esta manera podemos construir cabeceras que contengan macros y funciones creadas por nosotros y para nuestras necesidades, es decir, generar **bibliotecas** de macros y bibliotecas de funciones (no compiladas).

Hay una importante diferencia en estos casos que analizaremos a continuación.

BIBLIOTECAS DE FUNCIONES FUENTE

Podemos construir archivos de texto compuestos por el **cuerpo** de funciones listas para ser compiladas.

Incluso se pueden compilar en forma aislada para detectar posibles errores de compilación, y efectuar la depuración correspondiente, a fin de tener la certeza de que ellas no generarán errores de compilación cuando este proceso se realice sobre todo el programa.

La inclusión del archivo de funciones podrá hacerse al final o al principio del programa, siempre fuera de toda función (incluso del `main`).

En caso de realizar la inclusión al final del programa, será necesario incluir al principio los **prototipos**. En caso de realizarse la inclusión al principio, esto no será necesario.

Al realizar una biblioteca tanto de funciones como de macros, debe tenerse en cuenta que probablemente no todas ellas serán utilizadas por el programa.

Veremos qué efecto tiene esta situación en el caso de biblioteca de funciones, mediante dos ejemplos.

EJEMPLO: CABECERA PROPIA

Se creó un archivo llamado `FUNCION1.H` que contiene el código fuente de una función de impresión de un vector de enteros.

```
/* FUNCION1.H */
void imprimir ( int V[ ] , int L )
{
    int I ;
    for ( I=0 ; I<L ; I++ )
        printf ( " \n\t %d " , V[I] ) ;
}
```

El programa que incluye al anterior se denomina `INCLUDE1.C` y generará un `INCLUDE1.EXE`.

La inclusión se realizó al principio del programa por lo que no se utilizaron los prototipos.

```
/* INCLUDE1.C */
#include <stdio.h>
#define MAX 10
#include "FUNCION1.H"
```

```

int main ( )
{
    int VEC[MAX] , I ;
    for ( I=0 ; I<MAX ; I++ ) VEC[I] = I ;
    imprimir ( VEC , MAX ) ;
}

```

EJEMPLO: CABECERA PROPIA CON FUNCIONES MÚLTIPLES

En este caso, se creó un archivo de texto con el código fuente de tres funciones. La descripta anteriormente, una función de lectura y otra de ordenamiento de un vector de enteros. Este archivo se denomina FUNCION2.H .

```

/* FUNCION2.H */
void imprimir ( int V[ ] , int L )
{
    int I ;
    for ( I=0 ; I<L ; I++ )
        printf ( " \n\t %d " , V[I] ) ;
}

void leer ( int V[ ] , int L )
{
    int I ;
    for ( I=0 ; I<L ; I++ ) {
        printf ( "Ingrese el elemento %d = " , I ) ;
        scanf("%d" , &V[I] ) ;
    }
}

void ordenar ( int V[ ] , int L )
{
    int I , J , AUX ;
    for ( I=0 ; I<L-1 ; I++ )
        for ( J=0 ; J<L-I-1 ; J++ )
            if ( V[J] > V[J+1] ) {
                AUX      = V[J] ;
                V [J]   = V[J+1] ;
                V [J+1] = AUX ;
            }
}

```

El programa es similar al del ejemplo anterior, excepto que incluye a la cabecera FUNCION2.H en lugar de FUNCION1.H .

Está claro que el programa no utiliza dos de las tres funciones de FUNCION2.H.

El ejecutable generado se denominará INCLUDE2.EXE .

```

/* INCLUDE2.C */
#include <stdio.h>
#define MAX 10
#include "FUNCION2.H"

int main ( )
{
    int VEC[MAX] , I ;
    for ( I=0 ; I<MAX ; I++ ) VEC[I] = I ;
    imprimir ( VEC , MAX ) ;
}

```

Al observar los programas objeto y ejecutable que generan los fuentes INCLUDE1.C e INCLUDE2.C, podemos observar que hay una diferencia de tamaño apreciable para dos programas que están haciendo lo mismo.

INCLUDE1.O	841 bytes
INCLUDE2.O	1400 bytes
INCLUDE1.EXE	12077 bytes
INCLUDE2.EXE	15916 bytes

Esta diferencia en los archivos objeto se debe a que en el archivo FUNCIONES2.H incluido en INCLUDE2.C, se compilan también las dos funciones no utilizadas.

La diferencia es aún mayor en los ejecutables debido a la incorporación de funciones de C invocadas por las funciones no utilizadas.

Como conclusión podemos señalar que, al utilizar una biblioteca de funciones extensa, se desperdiciará memoria generando programas innecesariamente largos debido a las funciones incorporadas pero no usadas.

BIBLIOTECAS DE MACROS

Podemos repetir el razonamiento empleado para la utilización de bibliotecas de funciones fuente, diciendo que se pueden construir colecciones de macros propias en archivos de texto que luego pueden ser adosados a algún programa usando la directiva #include.

En este caso, la inclusión debe realizarse necesariamente al principio del programa, en algún punto anterior a la invocación de las macro, a fin de que puedan realizarse los reemplazos ordenados por los #define.

Podemos graficar esta situación con un par de ejemplos similares a los anteriores pero utilizando bibliotecas de macros.

EJEMPLO: CABECERAS QUE CONTIENEN MACROS

Se creó una cabecera llamada `MACRO1.H` que dispone de una macro que cumple la misma función que la comentada en `FUNCION1.H`.

```
/* MACRO1.H */
#define imprimir(V,long) { int I ; for ( I=0 ; I<long ; I++ ) \
                           printf ( " \n\t %d " , V[I] ) ; }
```

El programa es similar a los vistos en los dos ejemplos anteriores, con la diferencia de que se incluye la cabecera `MACRO1.H`.

El archivo fuente se llama `INCLUDE3.C` y generará posteriormente los archivos `INCLUDE3.O` e `INCLUDE3.EXE`.

Por similitud con los programas anteriores se respetó la ubicación del `#include`, pero éste podría haberse ubicado en cualquier lugar anterior a la línea que contiene la invocación a `imprimir(VEC, MAX)`;

```
/* INCLUDE3.C      */
#include <stdio.h>
#define MAX 10
#include "MACRO1.H"

int main ( )
{
    int VEC[MAX] , I ;
    for ( I=0 ; I<MAX ; I++ ) VEC[I] = I ;
    imprimir ( VEC , MAX ) ;
}
```

EJEMPLO: BIBLIOTECA PROPIA DE MACROS

En este caso se creó un archivo llamado `MACRO2.H` que contiene tres macros que cumplen las mismas tareas que las funciones contenidas por `FUNCION2.H`.

```
/*      MACRO2.H      */

#define imprimir(V,long) { int I ; for ( I=0 ; I<long ; I++ ) \
                           printf ( " \n\t %d " , V[I] ) ; }

#define leer(V,long) { int I ; for( I=0 ; I<long ; I++ ){ \
                           printf ( "Ingrese el elemento %d =",I );\
                           scanf ( "%d" , &V[I] ) ; } }

#define ordenar(V,long) { int I , J , AUX ; \
                           for ( I=0 ; I<long-1 ; I++ ) \
                           for ( J=0 ; J<long-I-1 ; J++ ) \
```

```

        if ( V[J] > V[J+1] ) { \
            AUX      = V[J] ; \
            V[J]    = V[J+1] ; \
            V[J+1] = AUX      ; } \

```

El programa INCLUDE4.C es similar a INCLUDE3.C excepto que incluye a MACRO2.H en lugar de MACRO1.H.

```

/*      INCLUDE4.C      */
#include <stdio.h>
#define MAX 10
#include "MACRO2.H"

int main ( )
{
    int VEC[MAX] , I ;
    for ( I=0 ; I<MAX ; I++ ) VEC[I] = I ;
    imprimir ( VEC , MAX ) ;
}

```

Al observar los programas objeto y ejecutable que generan los fuentes INCLUDE3.C e INCLUDE4.C, podemos observar que, a diferencia del caso de bibliotecas de funciones, no existe diferencia de tamaño entre ellos.

INCLUDE3.O	676 bytes
INCLUDE4.O	676 bytes
INCLUDE3.EXE	11933 bytes
INCLUDE4.EXE	11933 bytes

Esto es debido a que las macros son *reemplazos*, y en caso de no detectarse en el programa el *nombre_de_macro* correspondiente, no habrá reemplazo y por lo tanto no habrá código, pues el texto de la macro (la cadena) es una indicación para el **preprocesador** y no será incorporada al programa.

Concluimos entonces que podemos construir grandes bibliotecas de macros propias sin correr el riesgo de generar programas innecesariamente voluminosos como en el caso de utilizar bibliotecas de funciones fuentes.

COMPILACIÓN CONDICIONAL

Las herramientas de **compilación condicional** permiten definir uno o más bloques del programa fuente que se compilarán (o no) según el cumplimiento (o no) de una condición.

El cumplimiento de la condición es evaluado *antes* del proceso de compilación, y por lo tanto, esta condición no puede incluir variables. Sólo puede contener constantes previamente definidas, o bien se puede decidir por la definición o no de una macro.

Los bloques seleccionados también pueden contener comandos al compilador que serán tenidos en cuenta o no según el resultado de la condición.

La compilación condicional tiene aplicación cuando se construye un programa general del cual se extraerán variantes a medida de diferentes usuarios, o bien en procesos de *debugging*.

Los comandos involucrados en la compilación condicional son:

- `#if`
- `#else`
- `#elif`
- `#ifdef`
- `#ifndef`
- `#endif`

Los comandos `#if` y `#else` funcionan en forma similar a las instrucciones `if-else` de C.

El comando `#endif` sirve para delimitar el bloque condicional, es decir, indicar el límite final del bloque de compilación condicional.

El comando `#elif` se utiliza para concatenar un nuevo `#if` a un `#else` y obtener de esta forma la estructura del *escalonador*.

Se ilustraran los usos de estos comandos mediante los ejemplos siguientes.

EJEMPLO: PREGUNTA PARA EL LECTOR

Observe el siguiente programa e indique si puede presentarse algún problema y, en ese caso, cuál sería la causa.

```
#include <stdio.h>
int main(void)
{
    int A ;
    A = 1000 ;
#ifndef A > 500
    printf ( " VALOR GRANDE " ) ;
#endif
}
```

El problema que aquí se presenta es que la condición está basada en el contenido de una variable y esto genera un error de compilación. En el mensaje de error se indicará que es necesaria una expresión constante en el `#if`.

Esto es lógico dado que es necesario evaluar la condición *antes* de compilar, y la variable `A` solo tiene significado *después* de este proceso.

EJEMPLO: SELECCIÓN ENTRE DOS BLOQUES DE COMPILACIÓN

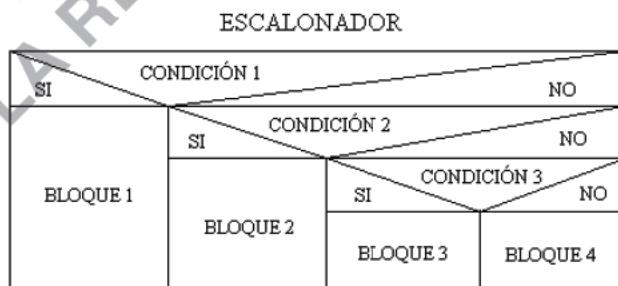
Este ejemplo muestra una selección entre dos bloques a ser compilados, basada en la evaluación del valor asignado a `MAX` mediante un `#define`. Obsérvese que en este caso se está evaluando una **expresión constante**.

```
#include <stdio.h>
#define MAX 2500

int main()
{
    int VEC[MAX] ;
#if MAX > 1000
    printf ( " CANTIDAD GRANDE DE ELEMENTOS " ) ;
#else
    printf ( " CANTIDAD NORMAL DE ELEMENTOS " ) ;
#endif
}
```

EJEMPLO: SELECCIÓN ENTRE MÚLTIPLES BLOQUES DE COMPILACIÓN

Se observará en este caso la estructura del **escalonador** implementado con los comandos `#if`, `#elif` y `#else`.



El formato que adopta el escalonador implementado con comandos al compilador tiene la siguiente forma:

```

#if    CONDICIÓN 1
      sentencias ;
#elif CONDICIÓN 2
      sentencias ;
#elif CONDICIÓN 3
      sentencias ;
#else
      sentencias ;
#endif

```

En este ejemplo se selecciona un país de la lista y se compila el bloque que le corresponde. Nótese que la comparación es numérica, pues primero se asigna un valor a cada macro y luego se compara según dicho valor.

```

#include <stdio.h>
#define ARGENTINA 1
#define URUGUAY   2
#define BRASIL    3
#define PAIS      URUGUAY

int main()
{
#if    PAIS == ARGENTINA
    printf ( " El país seleccionado es ARGENTINA " ) ;
#elif  PAIS == URUGUAY
    printf ( " El país seleccionado es URUGUAY " ) ;
#elif  PAIS == BRASIL
    printf ( " El país seleccionado es BRASIL " ) ;
#else
    printf ( " No se selecciono ningún país valido " ) ;
#endif
}

```

Está claro que en el ejemplo cada bloque es extremadamente simple, pero en un proyecto real éstos serían largos y complejos.

Podría haberse realizado la selección mediante un `switch` o un escalonador en tiempo de ejecución, pero en ese caso se hubiera compilado innecesariamente una gran porción del programa fuente.

Recuérdese que en este caso no se realiza una selección del país en el momento de ejecutar el programa, sino que se tiene un programa general (aplicable a varios países) pero luego se lo transforma en un programa a medida de cada país realizando la selección *antes* de compilarlo.

EJEMPLO: COMPILACION CONDICIONAL CON FINES DE DEPURACION

```
#include <stdio.h>

/* Dejando o eliminando esta linea se seleccionará */
/* si se desea tener impresiones de seguimiento o no */
#define DEPURANDO

int main()
{
    int I , N , SUM = 0 ;

#ifndef DEPURANDO
    int AUX = 0 ;
#endif

    printf ("Ingrese un valor para calculo de sumatoria \n") ;
    scanf ( "%d" , &N ) ;
    for ( I=1 ; I<N ; I++ ) {
        SUM += I ;

#ifndef DEPURANDO
        printf ( " \n %d " , SUM ) ;
        AUX++ ;
#endif

    }

    printf ( " \n La SUMATORIA de todos los valores " ) ;
    printf ( " naturales menores que %d vale %d " , N , SUM ) ;

#ifndef DEPURANDO
    printf ( " \n Calculado en %d vueltas. " , AUX ) ;
#endif
}
```

El programa mostrado selecciona los bloques evaluando si la macro DEPURANDO está definida o no.

El comando del compilador utilizado es `#ifdef`. Si la macro fue definida (*if defined*), se compilarán una serie de líneas destinadas a realizar un seguimiento de la ejecución del programa.

Una vez que el programa ha sido analizado y funciona correctamente, bastará con eliminar la línea en la que se define la macro, o bien *desdefinirla* antes de las evaluaciones con el comando `#undef DEPURANDO`, y compilar nuevamente.

Como resultado tendremos un programa objeto y luego un ejecutable que ignoran que en el programa fuente existen las líneas de depuración y los comandos relacionados.

EJEMPLO: USO DE #IFNDEF

En este ejemplo se plantea el caso de una posible pero no segura definición de macro. Obviamente tal definición no estaría en el propio programa pues si así fuera tendríamos la certeza de que la macro fue o no definida.

Pero podría estar definida (o no) dentro de un programa de texto incluido en el nuestro. Si así fuera podemos desear que se conserve el valor de tal definición, pero si no, asignar una definición propia.

En este caso, si el nombre de un archivo de trabajo ha sido definido en la cabecera CONSTANT.DEF, lo mantendremos. Si no fue definido (cosa que detectaremos con el comando `#ifndef = if not defined`), definiremos en nuestro programa a DATOS.DAT como archivo de trabajo.

```
#include <stdio.h>
#include "CONSTANT.DEF"

int main()
{
#ifndef ARCHIVO
    #define ARCHIVO "DATOS.DAT"
#endif

    printf (" El archivo de trabajo es %s \n\n " , ARCHIVO ) ;
}
```

El archivo cabecera CONSTANT.DEF debe encontrarse en el directorio actual.

Se puede compilar y ejecutar el programa con el archivo cabecera tal como está, y posteriormente volverlo a compilar y ejecutar con el archivo cabecera modificado, anulándole la línea donde se define la macro ARCHIVO con la expresión DEFINIDO.TXT, observando en ambos casos el nombre del archivo de trabajo seleccionado en pantalla.

```
/* CONSTANT.DEF */
/* Archivo de definición de constantes */

#define PI 3.1416
#define ARCHIVO "DEFINIDO.TXT"
#define VALOR_MAX 200

/* Final del archivo */
```

14. ARCHIVOS DE DISCO

Cuando se menciona la palabra **archivo**, inmediatamente se presenta la imagen de un programa o documento, guardado en un disco, pendrive o similar.

Desde este punto de vista, podríamos definir archivo como un conjunto de datos, almacenados con un nombre común (el nombre del archivo) en algún soporte magnético, óptico o de otro tipo.

Esta es una imagen parcial del término archivo, tal vez la más difundida y familiar.

Cuando trabajamos con diferentes estructuras de datos, podemos encontrar que hay información de diferente tipo, pero relacionada con un mismo ítem general.

Por ejemplo, en la ficha médica de un paciente, coexisten datos como *Juan Pérez* (string), 34 (int), *cardiopatía congénita* (string), OSECAC (string), etc.

Estos datos, que corresponden al nombre (string), edad (int), historia clínica (string), obra social (string), etc., tienen en común que pertenecen a Juan Pérez.

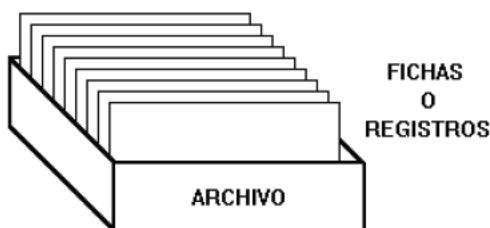
La estructura de datos antes mencionada, es decir, un conjunto de datos relacionados, no necesariamente del mismo tipo, recibe frecuentemente el apelativo de **registro**, en inglés, *record* (no confundir con *register*).

Vemos que este tipo de dato es el que conocemos en C como **estructura**.

NOMBRE	EDAD	HISTORIA CLINICA	OBRA SOCIAL

REGISTRO

Como se mencionó anteriormente, este conjunto corresponde a una ficha del paciente. Con una gran cantidad de ellos, se tendrá un conjunto de fichas o registros, agrupados en un **fichero** o **archivo** (en inglés, *file*).



Desde este punto de vista, un archivo se puede interpretar como un *conjunto de registros*, constituyendo cada uno de estos conjuntos de información, de igual o diferente tipo, relacionada de alguna manera.

Una aplicación ideal para este tipo de estructuras es la planilla de cálculo. A continuación se muestra un ejemplo en el que se maneja el archivo de pacientes descripto anteriormente.

A1	B	C	D	E
1	NOMBRE	EDAD	HISTORIA CLINICA	OBRA SOCIAL
2	Perez, Juan	34	Ver legajo 4356	Dasuten
3	Lopez, Javier	25	Ver legajo 4585	Osplad
5	Mendez, Luisa	52	Ver legajo 3689	Osde Binario
6	Morán, Blanca	43	Ver legajo 4647	Medicus
7	Alvarez, María	62	Ver legajo 3950	Ostel
8	Gonzalez, Carlos	47	Ver legajo 4564	Osplad
9	Rojas, Mónica	41	Ver legajo 4351	Dasuten
10	Díaz, Bruno	38	Ver legajo 4463	Medicus
11	Simpson, Homero	49	Ver legajo 4336	Pami
12				
13				

Otra forma de implementar este tipo de datos es utilizar una estructura en C, y luego agrupar varias de ellas en un vector. Dicho de otro modo, constituye un **vector de estructuras**, como el que se declara a continuación :

```
struct {  
    char NOMBRE [20] ;  
    int EDAD ;  
    char HISTORIA[60] ;  
    char OBRA_SOCIAL[20] ;  
} VECTOR [ N ] ;
```

Dado que el vector así declarado residirá en el segmento de datos de la memoria ocupada por nuestro programa, se lo denomina frecuentemente **Archivo en RAM**.

ANSI C nos ofrece otra perspectiva para un archivo. En este caso, se considera que un archivo es cualquier dispositivo físico capaz de mandar o recibir una secuencia de bytes a la computadora. De esta manera, pueden ser considerados archivos un CD, el disco rígido, un pendrive, modem, pantalla, teclado, etc.

Obsérvese que el **almacenamiento físico** de los datos no es un atributo excluyente de los archivos. Es decir, los archivos no solamente almacenan datos. Esto se aplica especialmente al teclado, la pantalla, la impresora, etc.

Como sabemos, el lenguaje C dispone de muy pocas instrucciones propias, y ninguna de ellas está destinada al manejo de archivos. Esta situación se subsana mediante una abundante biblioteca de funciones.

Existen tres maneras de manejar los archivos, a saber:

- **Archivos de tipo ANSI:** utilizan un buffer intermedio para almacenar temporalmente la información que se transfiere hacia o desde el archivo. Por esta razón se los suele denominar *archivos con buffer* o “*buffereados*”.
- **Archivos de tipo UNIX:** en este caso la información se transfiere al archivo sin agruparse previamente en un buffer. Por eso reciben el nombre de *archivos sin buffer*. Este es el modo implementado para los primeros compiladores de C que trabajaban bajo UNIX. No es tan eficiente como el anterior.
- **Accesos de bajo nivel:** Consiste en acceder directamente al hardware mediante funciones específicas que lo manejen. Como veremos a continuación, de ser este el caso, se deberá tratar cada archivo según su propio caso particular.

Los archivos de tipo ANSI, al tener un almacenamiento intermedio, optimizan el acceso al hardware, realizando el mismo con bloques completos de información. Focalizaremos nuestra atención en este tipo de archivos.

La cabecera que contiene los prototipos es `stdio.h` .

FLUJOS

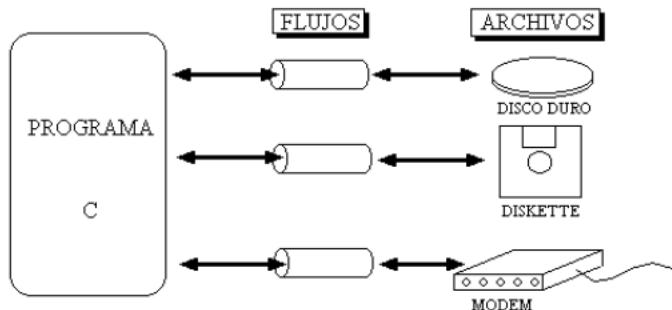
Dado que se tiene una gran diversidad de dispositivos físicos diferentes bajo la denominación de archivo, será necesario implementar un acceso apropiado para cada uno, dependiendo de las características particulares de su hardware.

Esto haría muy difícil la tarea del programador, dado que deberá interiorizarse del funcionamiento de cada dispositivo.

El lenguaje C proporciona, en colaboración con el sistema operativo, una **interfaz lógica** entre el programa y el dispositivo de manera de tratar a todas estas interfaces de la misma manera. Es decir, el tratamiento será independiente del dispositivo al que se accede.

Esta interfaz recibe el nombre de **flujo** o *stream*. De hecho se trata de eso, un flujo o corriente de datos fluyendo entre nuestro programa y el dispositivo. Se deja entonces el apelativo de *archivo* al dispositivo físico.

Esta situación se muestra en el siguiente esquema:



Los flujos proporcionan un nivel de **abstracción** entre el programador y el dispositivo físico al que se accede. Obsérvese en el dibujo anterior que, desde el programa, todos los dispositivos se ven igual y se les pueden aplicar las mismas funciones.

Según este esquema, entonces, se accede a los archivos a través de los flujos. Cada archivo tiene un flujo asociado a él. Por lo tanto, habrá tantos flujos como archivos abiertos haya en un momento dado. Es necesario entonces individualizarlos de alguna manera.

A continuación se resumen los pasos necesarios para trabajar con un archivo, según lo planteado en los párrafos precedentes.

- Creación de un flujo
- Asignación de un identificador
- Ligadura entre el flujo y el archivo

Estos tres pasos se unifican en una operación llamada **apertura de archivo**.

La dissociación entre el archivo y el flujo, como así también la liberación del buffer, se producen en el proceso de **cierre de archivo**.

TIPOS DE FLUJOS

Podemos diferenciar entre dos tipos diferentes de flujo:

- Flujos de texto
- Flujos binarios

Los **flujos de texto** son secuencias de caracteres en las que cada byte es interpretado como un carácter ASCII.

En lenguaje C, '\n' representa el **cambio de línea**, pero bajo Windows, históricamente esto mismo es representado por el conjunto **retorno de carro (CR)** y **cambio de línea (LF)**. La misma situación en C sería '\r' '\n'.

Para resolver esta diferencia se realiza una transformación en los caracteres que se envían al archivo, y una *antitransformación* cuando se los recupera.

Por esa razón puede no coincidir la cantidad de bytes enviados a un archivo en modo texto con la cantidad de bytes almacenados (en Windows).

Los **flujos binarios** son secuencias de bytes en los que no se realiza transformación alguna. Lo almacenado en un archivo binario es exactamente lo que se envía. Es necesario conocer lo que se envió a fin de recuperarlo correctamente. Por ejemplo, si se envió un `float`, se habrán guardado 4 bytes, que al recuperarlos pueden interpretarse como un `float`, o como un `long`, o dos `short int`, etc.

Los flujos binarios son adecuados para almacenamiento de números, dado que este se realiza más eficientemente. Por ejemplo, almacenar el número 31265, demanda 2 bytes equivalentes a un entero corto de 16 bits, mientras que en modo texto necesitaríamos 5 bytes correspondientes a los códigos ASCII de cada dígito, más eventualmente un carácter nulo de finalización de string.

Por otro lado, los archivos binarios no son legibles por procesadores de texto, ya que estos programas se limitan a convertir secuencias de bytes en los caracteres ASCII correspondientes. Existen programas para visualizar y editar cómodamente archivos binarios.

FLUJOS ESTÁNDAR

Cada vez que se requiera la creación de un flujo será necesario realizar el proceso explícito de apertura del archivo correspondiente.

Sin embargo, al iniciarse un programa, hay determinados flujos que se crean automáticamente por el sistema operativo, sin mediar petición alguna. Los conocemos como **flujos estándar**.

Estos flujos y sus dispositivos asociados por defecto son:

- `stdin` *standard input* (teclado)
- `stdout` *standard output* (pantalla)
- `stderr` *standard error* (pantalla)

Es necesario aclarar que según cómo se ejecute nuestro programa, estos flujos pueden haber sido *redirigidos* hacia otro archivo. Tanto las líneas de comandos de Windows como de Linux son capaces de permitir esto último. Por ejemplo, la ejecución de un programa del siguiente modo enviará la salida a un archivo, y no a pantalla:

```
programa.exe > salida.txt
```

Se invita al lector a comprobar esto último mediante el uso de `printf`.

ARCHIVOS EN ANSI C

ANSI C dispone de una serie de funciones destinadas al manejo de archivos. Todas ellas están relacionadas con el archivo cabecera `stdio.h` por lo que éste deberá ser incluido en el programa.

Las describiremos a medida que tratemos cada etapa en el manejo de un archivo a través de un flujo.

APERTURA DE ARCHIVO

El proceso de apertura del archivo implica la creación de un flujo, la asignación de una etiqueta al mismo a fin de reconocerlo, y la conexión de dicho flujo con el archivo físico.

Esta tarea la realiza la función `fopen()`, cuyo prototipo se da a continuación :

```
FILE * fopen ( const char * nom_arch , const char * modo ) ;
```

En el prototipo se observa que `fopen()` recibe una string que representa el nombre del archivo a abrir, y otra string que representa el **modo de apertura**.

El hecho de que estos parámetros estén afectados por el modificador `const` solamente significa que no serán modificados dentro de la función.

Supondremos de aquí en más que el archivo, es un **archivo de disco**. De esta forma, `nom_arch` representará el nombre de dicho archivo. Dentro de este string, se podrá colocar el *path* completo, de ser necesario, para acceder al archivo mencionado.

Por otro lado, la función `fopen()` retorna un puntero a `FILE`, que es una **estructura** declarada dentro de `stdio.h` . El uso de `typedef` posibilita que el nombre no contenga la palabra `struct`, pero no por ello lo retornado deja de ser un puntero a `struct`.

DETALLE DEL PROCESO DE APERTURA DE UN ARCHIVO

Los pasos descriptos anteriormente forman un modelo simplificado de lo que realmente ocurre. Los veremos ahora en más detalle.

a. Creación de un flujo: Consiste en crear en memoria, una estructura de control de tipo `FILE`, como así también un buffer destinado al almacenamiento temporal de los datos en tránsito (recordemos que se trata de *archivos con buffer*).

Se relaciona la estructura `FILE` con el buffer mediante un puntero que pertenece a la primera y apunta al inicio del último.

b. Identificación del flujo: La función `fopen()` retorna la dirección donde está ubicada la estructura de control `FILE`. Deberemos asignar dicha dirección a un

puntero a FILE (que crearemos para tal fin). A partir de ese momento, el identificador del flujo será el nombre del puntero.

c. **Nexo con el archivo físico:** Esto se realiza almacenando en la estructura FILE un *handler* c identificador, que permite el acceso a una *tabla de archivos abiertos* que maneja el sistema operativo. En esta tabla se encuentran los *FCB* (*File Control Blocks*) o *bloques de control de archivos* que utiliza el sistema operativo para manejar los mismos.

MODOS DE APERTURA DE ARCHIVOS

Como vimos anteriormente los flujos pueden ser **binarios** o de **texto**. Esto se selecciona en el proceso de apertura del archivo.

Asimismo se puede optar por abrir archivos para escritura, lectura, añadir datos o combinaciones de ellos.

Los caracteres asociados a estos modos son seis (**w** **r** **a** **+b** **t**).

Si no se indica lo contrario, los archivos se abren en modo texto, por lo que el agregado de la t es superfluo.

- **w** Apertura (creación) de un archivo de texto para escritura. Si existe se lo destruye y luego se lo abre.
- **wt** Idem anterior.
- **wb** Apertura (creación) de un archivo binario para escritura. Si existe se lo destruye y luego se lo abre.

- **r** Apertura de un archivo de texto para lectura. Si no existe se genera un error y falla.
- **rt** Idem anterior.
- **rb** Apertura de un archivo de texto para lectura. Si no existe se genera un error y falla.

- **a** Apertura de un archivo de texto en modo “*append*”. Lo que se escriba se añadirá al final del archivo. Si el archivo no existe se lo crea.
- **at** Idem anterior.
- **ab** Apertura de un archivo binario en modo “*append*”. Lo que se escriba se añadirá al final del archivo. Si el archivo no existe se lo crea.

- **r+** Apertura de un archivo de texto para actualizar. Se permite la lectura y escritura. Si el archivo no existe se genera un error y falla.
- **r+t** Idem anterior.
- **r+b** Apertura de un archivo binario para actualizar. Se permite la lectura y escritura. Si el archivo no existe se genera un error y falla.

- **w+** Creación de un archivo de texto en modo lectura-escritura. Si existe se lo destruye y posteriormente se lo crea.
- **w+t** Idem anterior.
- **w+b** Creación de un archivo binario en modo lectura-escritura. Si existe se lo destruye y posteriormente se lo crea.

- **a+** Apertura de un archivo de texto para lectura y añadidura. Si el archivo no existe se lo crea.
- **a+t** Idem anterior.
- **a+b** Apertura de un archivo binario para lectura y añadidura. Si el archivo no existe se lo crea.

Es necesario tener mucho cuidado con los modos que involucran a **w** porque al utilizarlos con un archivo existente *este resulta destruido*. Si esto se hiciera por error, acarrearía una pérdida de información.

MECÁNICA DE APERTURA DE UN ARCHIVO

La apertura de un archivo puede fracasar debido a diversos motivos, como ser:

- Apertura para lectura de un archivo inexistente.
- Apertura para escritura en una carpeta protegida contra la misma.
- Apertura para escritura sin tener los permisos de usuario.
- Apertura para escritura de un archivo con atributo *read only*.
- Apertura para escritura en un disco lleno.
- Etc.

En estos casos la función **fopen()** nos permite saber la ocurrencia del error, retornando un puntero **NULL**. La dirección **00H** equivalente a **NULL** no es una dirección válida.

Podría decirse que ninguna función “*en su sano juicio*” retornará una dirección **00H** como válida, por lo que se la reserva para indicación de error.

Es fundamental realizar la comprobación del valor returnedo por **fopen()**, debido a que si no se pudo abrir el archivo, y tomamos la dirección retornada como buena, estaremos trabajando erróneamente sobre una estructura **FILE** que no existe, y apunta a un buffer que tampoco existe. La desreferencia del puntero nulo provocará seguramente la finalización anormal de nuestro programa.

La siguiente es una disposición habitual de comprobación, con su correspondiente aviso de error y aborto del programa en curso:

```
FILE * fp ;
if ( ( fp = fopen("PRUEBA.TXT" , "w") ) == NULL ) {
    printf ("\n Error en apertura del archivo. Programa abortado.");
    exit (1) ;
}
```

Podría colocarse también, en la comprobación, la evaluación de lo retornado por verdadero o falso, de la siguiente forma:

```
if (!( fp = fopen("PRUEBA.TXT" , "w")) ) { /* Notar la negación */
```

A partir de este momento, toda referencia al flujo se hará a través de fp.

CIERRE DE UN ARCHIVO

El proceso de cierre de un archivo consiste en la destrucción del flujo y el cierre del archivo a nivel sistema operativo.

Durante el proceso de cierre del archivo se envía al mismo el contenido remanente de información que se encuentre aun en el buffer.

Es importante llevar a cabo esta operación debido a que, si se produce una terminación anormal del programa, se perdería la información que esté en el buffer y que aún no se transfirió al archivo.

Posteriormente se libera la memoria ocupada por el buffer, dado que éste se encuentra en la parrilla. Es decir que en el proceso de apertura se la gestiona como memoria dinámica.

Por último se realiza un cierre formal del archivo a nivel del sistema operativo.

La función utilizada para llevar a cabo este proceso es `fclose()`, cuyo prototipo vemos a continuación:

```
int fclose ( FILE * nom_puntero ) ;
```

La función `fclose()` retorna un 0 si el archivo se cerró correctamente, y retorna un EOF (*End Of File*) o *caracter de fin de archivo*, si se produjo algún error.

EOF es una macro definida en `stdio.h` como -1, es decir que se trata de un número con todos sus bits en 1.

```
#define EOF (-1)
```

Posibles errores que podrían producirse al cerrar un archivo son: haber retirado el pendrive donde se alojaba el archivo, disco lleno, etc.

MODOS DE ACCESO A LOS ARCHIVOS

Podemos reconocer dos modos de acceder a la información de los archivos:

- Modo secuencial
- Modo directo o *random*

Modo secuencial: En este caso la información se accede *byte por byte*, uno a continuación del otro. Para acceder a un determinado byte, es necesario recorrer todos los anteriores en secuencia. Si bien es posible acceder a nivel de bloques de varios bytes simultáneamente, esto no cambia el concepto de acceso secuencial, dado que los mencionados bloques también se acceden en secuencia.

Modo directo: El concepto de acceso directo es el de indicar la dirección (o posición) de la información dentro del archivo, y acceder a ella sin necesidad de tener acceso a la información que se encontrara antes.

Para comprender estos modos encontramos muy útil e ilustrativo el concepto de **ventana**.

MODELO DE VENTANA

Imaginemos que el archivo es una sucesión muy larga de bytes, colocados uno a continuación del otro, en compartimentos que simulan una larga fila de cajas del mismo tamaño (1 byte).

Consideremos que el interior de estas cajas solo es visible a través de una **ventana** que tiene la facultad de desplazarse por esta larga hilera. Toda lectura y/o escritura del archivo deberá realizarse a través de la ventana.

Cuando se lee o escribe algo a través de la ventana, esta se *desplaza automáticamente* hacia adelante (hacia el final del archivo) una cantidad de bytes igual a los transferidos hacia o desde el archivo. Este desplazamiento es automático e inevitable.

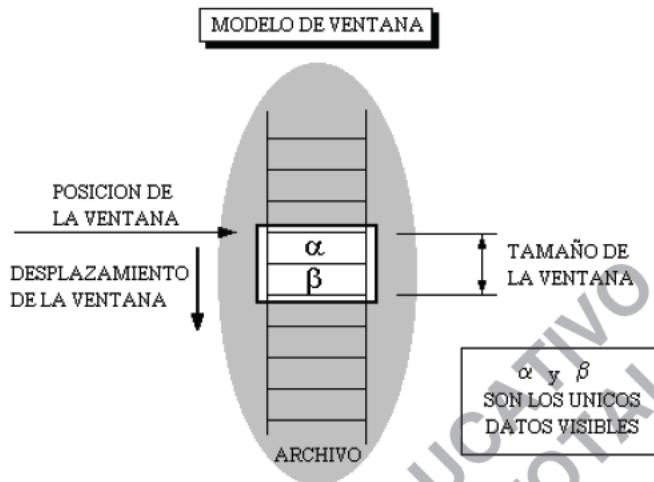
No solo no es necesario producir el desplazamiento, dado que este se produce solo, sino que no se lo puede evitar.

Leer o escribir un byte, un entero, un flotante o un bloque de n bytes, significa modificar momentáneamente el **tamaño** de la ventana.

Tener un acceso directo a una posición dentro del archivo significa mover el **inicio** de la ventana al lugar deseado, y posteriormente realizar la transferencia de datos a través de ella. Una vez ocurrido esto, la ventana se moverá hacia adelante, una cantidad de bytes igual a los transferidos.

Esto significa que la naturaleza intrínseca del manejo de archivos es **secuencial**. Aún los archivos de acceso directo son secuenciales.

El acceso directo se logra a costa de reposicionar la ventana cada vez que se desea acceder a una posición determinada dentro del archivo.



En la figura se muestra una ventana de dos bytes de longitud. Los dos únicos datos accesibles para lectura a través de ella son α y β . En el caso de escribir sobre el archivo, las posiciones sobreescritas serían las de α y β .

Luego de realizada alguna de estas operaciones (lectura o escritura), la ventana se desplazaría hasta que su tope coincida con la casilla del dato siguiente a β . Es decir, avanzará dos bytes.

ESCRITURA DE UN BYTE EN EL ARCHIVO

Esta es la más simple de las operaciones de escritura. Nos permitirá crear nuestros primeros archivos. La operación de escritura de un byte o un carácter en el archivo, a través de un flujo, tendrá efectos diferentes si se trata de un flujo de texto o un flujo binario.

Por lo demás, la operatoria es indistinta, se trate de uno o del otro.

La función `fputc()` y la macro `putc()` nos permiten realizar esta tarea.

Sus prototipos son:

```
int putc ( int carácter , FILE * puntero ) ;  
int fputc ( int carácter , FILE * puntero ) ;
```

Ambas se comportan de manera similar, excepto por la diferencia intrínseca macro/función. De hecho, el estándar no especifica que deban serlo.

Los argumentos a ser transferidos son el byte o carácter a ser enviado al archivo, y el puntero que identifica al flujo.

Obsérvese que el carácter a enviar al archivo se define como `int`. Esto ocurre por razones históricas, sin embargo, en caso de transferirse como argumento un entero, solo el byte menos significativo será considerado.

El valor returnedo por `putc()` o por `fputc()` es el carácter o byte enviado, si la operación tuvo éxito, o bien `EOF` si es que hubo algún error.

PRIMER EJEMPLO: ESCRITURA

Se creará un archivo de texto llamado `TEXTO.TXT`, abierto en el modo escritura de texto (`w`) al que se le enviarán caracteres utilizando la macro `putc()`. Estos caracteres provienen del teclado y finalizarán cuando se ingrese un carácter `$`.

```
/* Crea un archivo de texto llamado TEXTO.TXT */
/* El ingreso de caracteres finaliza con '$' */

#include <stdio.h>
int main ( )
{
    FILE * fp ;
    char car ;

    /* Crea un archivo de texto para escritura */
    if ( ( fp = fopen ( "TEXTO.TXT" , "w" ) ) == NULL ) {
        printf ( " No se puede abrir el archivo.\n\n " ) ;
        return 1;
    }

    printf ( " El ingreso finaliza con '$' \n \n " ) ;

    car = getchar ( ) ;      /* Lee carácter de teclado */
    while ( car != '$' ) {
        putc ( car , fp ) ; /* Manda carácter a disco */
        car = getchar ( ) ; /* Lee carácter de teclado */

    }
    fclose ( fp ) ;
}
```

El archivo `TEXTO.TXT` puede ser leído mediante cualquier editor dado que es texto puro en ASCII. O bien, construyendo nuestro propio programa de lectura, como veremos a continuación.

LECTURA DE UN BYTE DESDE UN ARCHIVO

Disponemos de funciones similares a las anteriores para realizar la lectura de un carácter o un byte desde el archivo.

Como en el caso anterior, disponemos de una función, `fgetc()`, y de una macro, `getc()` que se comportan de igual forma desde el punto de vista funcional. A continuación tenemos sus prototipos.

```
int getc ( FILE * puntero ) ;  
int fgetc ( FILE * puntero ) ;
```

Ambas reciben como argumento el puntero al flujo, y retornan un byte. Si bien el prototipo indica que el valor returned es un entero, sus bytes altos en condiciones normales siempre contienen cero, por lo que es válido considerar que se retorna un `char`, salvo en caso de detectar el fin del archivo.

Cuando se ha llegado al final del archivo, `getc()` y `fgetc()` devuelven un valor `EOF`. Es preciso recordar que `EOF` es un entero. En este caso `getc()` o `fgetc()` retornan el valor `FFFFFFFH`.

Si capturamos este valor returned en una variable de tipo `char`, ésta lo tomará como `FFH`. Una comparación de igualdad entre esa variable y `EOF` dará positivo, pero también lo dará cualquier dato `FFH` que se encuentre en el archivo.

De todas formas la detección con una variable `char` funcionará para archivos escritos y leídos en modo texto porque ningún carácter se representa con `FFH` en ASCII, pero puede traer problemas en el caso de archivos binarios, como se verá más adelante.

EJEMPLO: LECTURA

En este ejemplo se leerá el archivo generado en el ejemplo anterior, es decir, `TEXTO.TXT`. La mecánica utilizada es la de leer carácter por carácter desde el flujo y mostrarlos en pantalla, hasta detectar la lectura del `EOF`.

```
/* Abre el archivo de texto "TEXTO.TXT" y lo muestra en pantalla */  
/* Aplicación : uso de la macro getc() y detección de EOF */  
  
#include <stdio.h>  
  
int main ( )  
{  
    FILE *fp ;  
    char car ;
```

```

if ( ( fp = fopen ( "TEXTO.TXT" , "r" ) ) == NULL ) {
    printf ( " No se puede abrir el archivo. \n\n " ) ;
    return 1;
}

car = getc ( fp ) ; /* Lee un caracter del archivo */

while ( car != EOF ) {
    putchar ( car ) ;
    car = getc ( fp ) ;
}

fclose ( fp ) ;
printf ( "\n\n Fin del archivo. \n " ) ;
}

```

USO DE LOS FLUJOS ESTÁNDAR STDIN Y STDOUT

Dado que `stdin` y `stdout` son flujos similares a los de cualquier archivo de disco, podríamos tomar caracteres de teclado y enviarlos a la pantalla mediante el uso de las herramientas vistas recientemente, a fin de verificar su comportamiento.

Puede el lector verificar el efecto al reemplazar `getc(stdin)` y `putc(stdout)`, por `getchar()` y `putchar()`. Estas últimas son casos particulares de las primeras.

De hecho, son macros definidas en `stdio.h` en base a las primeras:

```

#define getchar()  getc(stdin)

#define putchar(c)  putc((c), stdout)

```

EJEMPLO CON FLUJOS ESTÁNDAR

Se verificará en este ejemplo, el ingreso desde teclado y salida a pantalla utilizando los flujos estándar. Se leerán caracteres desde el flujo `stdin` y se enviarán al flujo `stdout` hasta que se detecte el ingreso de un carácter `$`.

```

/* Uso de los flujos standard stdin y stdout */
/* El programa termina al ingresar '$' */

#include <stdio.h>
int main ( )
{
    char C = 0 ;
    printf ( " Para terminar ingrese '$'. \n\n " ) ;

```

```
while ( C != '$' ) {  
    C = getc ( stdin ) ;  
    putc ( C , stdout ) ;  
}  
}
```

Al ejecutar el programa el programa con la siguiente entrada, se observa :

```
Ensayo para stdin y stdout. $ Pero la frase sigue. <Enter>  
Ensayo para stdin y stdout. $
```

La primera línea aparece debido al eco producido en la escritura, pero no es tomada por `getc(stdin)` hasta que se detecta el `<Enter>` en el buffer. A partir de ese momento, se produce la lectura de caracteres y su consecuente escritura, hasta el signo `$`, dando lugar a la segunda línea y produciendo al corte del bucle.

El comportamiento de `getc(stdin)` es similar al que muestra la función `getchar()`. Podemos considerar que esta última representa un caso particular de ingreso de caracteres, referido al flujo de entrada estándar.

DETERMINACIÓN DEL FINAL DE ARCHIVO

En el programa anterior de escritura de un archivo en modo de texto, vimos que se enviaban caracteres al flujo relacionado al archivo, utilizando la macro `putc()`, o bien la función `fputc()`.

Estas funciones envían los mencionados caracteres al buffer en donde se van almacenando temporalmente hasta que dicho buffer se llena. En ese momento se transfiere la totalidad del contenido del buffer al archivo. Este procedimiento es eficiente ya que evita que se acceda al dispositivo físico carácter por carácter.

Otra manera de obligar dicha transferencia es realizar el cierre del archivo.

Al leer, la función o macro nos señala el fin del archivo retornando `EOF`. Este método funciona correctamente para flujos de texto, dado que podemos suponer que ningún carácter que se ingrese coincidirá con el byte menos significativo de del fin de archivo `EOF` (-1). Esto corresponde al carácter de ASCII 255.

Sin embargo, no podemos decir lo mismo de los archivos binarios, donde los datos son sucesiones de bytes y no de caracteres.

Consideraremos el caso en que se almacenan en disco los datos provenientes de una placa de adquisición de datos o cualquier otro dispositivo similar.

El valor `0xFF` es uno más entre todos los datos posibles que provienen de la placa, y como en flujos binarios no hay reemplazo de datos, el falso `EOF` estará en alguna parte del archivo.

Si se intenta determinar el fin de archivo mediante la detección del EOF, se tendrá una finalización anticipada errónea.

EJEMPLO: ESCRITURA DE ARCHIVO BINARIO

Se generará el archivo BINARIO.DAT, abierto en modo de escritura binaria.

Los datos enviados a este flujo binario se obtendrán de un vector, en el que está presente el dato EOF.

```
/* Genera el archivo binario "BINARIO.DAT", con los datos del      */
/* vector VEC, en el que se ha introducido un EOF para engañar      */
/* a una posterior lectura en modo texto                          */

#include <stdio.h>
int main( )
{
    FILE *fp ;
    char I , VEC[ ] = { 65 , 66 , 67 , EOF , 68 , 69 , 70 } ;

    if ( ( fp = fopen ( "BINARIO.DAT" , "wb" ) ) == NULL ) {
        printf ( " No se puede abrir el archivo. \n\n " ) ;
        return 1 ;
    }

    for ( I = 0 ; I < 7 ; I++ )
        putc ( VEC[I] , fp ) ;

    fclose ( fp ) ;
}
```

Si el archivo así generado se lee mediante el programa anterior (cambiando el nombre del archivo de datos, por supuesto), lo que se verá es:

ABC

Esto corresponde a los caracteres ASCII de los valores 65, 66 y 67 del archivo, verificándose que no se avanzó más allá del falso EOF.

Es decir, se logró engañar la detección del fin de archivo.

Si la apertura del archivo se realiza en forma binaria, pero se detecta igualmente el fin de archivo mediante EOF, también resulta engañado el programa por el fin de archivo falso. Esto puede comprobarse en el siguiente ejemplo.

EJEMPLO: LECTURA DE ARCHIVO BINARIO

Este ejemplo es similar al anterior, con la diferencia de que el modo de apertura es lectura binaria. De este modo se espera una sucesión de bytes que se mostrará en pantalla en formato hexadecimal.

La ejecución del programa genera:

41 42 43

que corresponde a los caracteres del archivo hasta el falso EOF.

```
/* Lee datos de un archivo binario hasta EOF */

#include <stdio.h>
int main( )
{
    FILE *fp ;
    char car ;

    if ( ( fp = fopen ( "BINARIO.DAT" , "rb" ) ) == NULL ) {
        printf ( " No se puede abrir el archivo. \n\n " ) ;
        return 1;
    }

    car = getc ( fp ) ; /* Lee un caracter */
    while ( car != EOF ) { /* Detección de fin de archivo: EOF */
        printf ( " %02X " , car ) ;
        car = getc ( fp ) ;
    }

    fclose ( fp ) ;
    return 0 ;
}
```

REEMPLAZO DE CARACTERES EN FLUJOS DE TEXTO

Puede utilizarse el programa precedente para realizar una lectura binaria de un archivo **TEXTO.TXT**, el cual debe cargarse con algún carácter de cambio de línea para observar el reemplazo efectuado en los flujos binarios. Esto puede hacerse con cualquier editor de texto.

Como actividad se propone escribir el archivo **TEXTO.TXT** con la secuencia :

ABC<Enter>CDE\$

Posteriormente, lea este archivo con el programa del ejemplo anterior y verifique que la impresión arroja:

41 42 43 0D 0A 44 45 46

Se puede observar el reemplazo del *retorno de carro* por el par *cambio de línea/retorno de carro*. Es decir, <Enter> equivale al par hexadecimal 0D 0A (dos bytes).

MACRO FEOF

Para solucionar el problema de la detección errónea de fin de archivo, podemos utilizar la macro estándar `feof()`.

Esta macro recibe como argumento el puntero al flujo y retorna un valor verdadero (*distinto de cero*) si ya se alcanzó el final del archivo, y un valor falso (*cero*) si aún no se lo alcanzó.

Su prototipo es el siguiente:

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

Esta macro es indicada para utilizar en lecturas de archivos binarios, aunque también funciona con archivos de texto.

EJEMPLO: FIN DE ARCHIVO CON FEOF

En este ejemplo podemos ver que se soluciona el problema presentado precedentemente. Al leer el archivo BINARIO.DAT que contiene un falso EOF, se ve que éste es ignorado, y el valor mostrado para él es FF, el equivalente al EOF (-1).

```
/* Lee datos del archivo binario "BINARIO.DAT" mientras */
/* ocurra que el valor retornado por feof() es falso      */

#include <stdio.h>
int main ( )
{
    FILE *fp ;
    unsigned char car ;

    if ( ( fp = fopen ( "BINARIO.DAT" , "rb" ) ) == NULL ) {
        printf ( " No se puede abrir el archivo. \n\n " ) ;
        return 1 ;
    }

    car = getc ( fp ) ; /* Lee un caracter */

    while ( !feof ( fp ) ) { /* Sigue mientras feof(fp) == 0 */
        printf ( " %X" , car ) ;
        car = getc ( fp ) ;
    }

    fclose ( fp ) ;
}
```

Los valores leídos en el archivo BINARIO.DAT fueron:

41 42 43 FF 44 45 46

Se observa correctamente la presencia del FF correspondiente al EOF.

FUNCIONAMIENTO DE FEOF

La detección realizada por `feof()` se basa en el testeo de uno de los *flags* de estado de la estructura de control `FILE` asociada al flujo.

Si se investiga dentro del archivo cabecera `stdio.h`, se puede encontrar la declaración de la macro `feof()`.

```
#define feof(F) ((F)->_flag & _IEOF)
```

Dado que `F` representa el puntero a la estructura `FILE` asociada al flujo, `f->_flag` será un campo de dicha estructura.

`_IEOF` es una macro definida en el mismo archivo cabecera como:

```
#define _IEOF 0x0010 /* EOF reached on read */
```

Obviamente `_IEOF` actúa como una máscara para acceder al bit indicador de fin de archivo dentro de un registro de estado de la estructura de control `FILE` asociada al archivo. Si esa bandera está en 1, `feof()` responde que se llegó al fin de archivo.

Cabe aclarar que `feof()` responde afirmativamente cuando ya se superó el fin de archivo, y no cuando se lo está por alcanzar. Por esto se utiliza una primera lectura fuera del bucle.

VALORES MÁS ALLÁ DEL FINAL DEL ARCHIVO

Los datos enviados al disco se guardan físicamente en cantidades de bytes de información fijas, preferentemente coincidentes con la longitud del buffer. Esta cantidad se denomina **unidad de asignación** y se determina a nivel sistema operativo y la configuración del sistema de archivos o *filesystem*. En sistemas modernos el tamaño suele ser 4 Kbytes.

¿Qué ocurre cuando mediante un `fclose()` forzamos a enviar el contenido incompleto del buffer? ¿Con qué se rellena la unidad de asignación en el disco?

Esto se puede dilucidar realizando una lectura más allá del final del archivo. La detección del fin de archivo es un punto fundamental para el manejo de los mismos, a fin de no considerar información inexistente. Pero no estamos obligados a respetarla.

EJEMPLO: LECTURA MÁS ALLÁ DEL FIN DE ARCHIVO

En el siguiente ejemplo observaremos BINARIO.DAT mas allá de su fin.

```
/* Lee 40 datos de un archivo binario y los muestra en pantalla */
/* Lee mas allá del fin en un archivo de menos de 40 bytes          */

#include <stdio.h>
int main ( )
{
    FILE *fp ;
    unsigned char car , I ;

    if ( ( fp = fopen ( "BINARIO.DAT" , "rb" ) ) == NULL ) {
        printf ( " No se puede abrir el archivo. \n\n " ) ;
        return 1 ;
    }

    car = getc ( fp ) ;

    for( I=0 ; I < 40 ; I++ ) {
        printf ( " %02X" , car ) ;
        car = getc ( fp ) ;
    }
    fclose ( fp ) ;
}
```

El resultado de la ejecución fue que luego del final se encontró una secuencia de FF. Lo que indicaría que al almacenar un archivo en disco, la unidad de asignación se completa con caracteres EOF.

También puede interpretarse como que `getc()` o `fgetc()` retornan -1 cuando se lee más allá del fin del archivo. El lector podrá sacar sus conclusiones mas adelante.

Esto no afecta la operatoria normal que venimos mostrando. El primer caso podría ser considerado como una “medida de seguridad” adicional, en caso de que fallara la detección del primer EOF.

De todas formas, en condiciones normales, es irrelevante con qué caracteres se completa la unidad de asignación, fuera del archivo en cuestión.

EOF RETORNADO POR GETC Y FGETC

Los valores retornados por `getc()` y `fgetc()` son enteros en los cuales, la mayor parte de las veces, el byte más significativo es cero. De esta manera, es correcto tomar ese valor en una variable de tipo `char`.

Lo correcto sin embargo, sería tomar el valor retornado en una variable entera a fin de realizar una comparación completa con EOF.

Pero, ¿cómo pueden retornar un entero si *investigan* byte por byte del buffer? Observemos la definición de la macro `getc()` hallada en `stdio.h`.

DEFINICIÓN DE GETC

Como hemos mencionado, el estándar no especifica si `getc` y `fgetc` deben ser macros o funciones, por lo que esta decisión queda en manos del fabricante de compilador. En el caso de GCC 4.9 encontramos que `getc` es una función `inline`.⁴

```
inline int getc (FILE* F)
{
    return (-- F->_cnt >= 0)
        ? (int) (unsigned char) *F->_ptr++
        : _filbuf (F);
}
```

A fin de definir el comportamiento de `getc()` se evalúa el contenido de la variable de la estructura `FILE` asociada al flujo, `F->_cnt`, siendo `F` el parámetro puntero al flujo.

`_cnt` (de *count*) como veremos más adelante es un campo de la estructura `FILE`, que contiene la cantidad de bytes que *restan ser leídos* en el buffer para llegar al final del mismo, o al final del archivo.

Si este valor es *mayor que cero*, se retorna el `unsigned char` apuntado por `F->_ptr`. Siendo `_ptr` (de *pointer*) un campo de la estructura `FILE` que indica cuál es la dirección del byte que debe ser leído (o escrito) a continuación.

De aquí se observa que se retorna solamente un byte (`unsigned char`) que a su vez se castea a `int`.

Pero en caso de detectarse que el campo `_cnt` es igual a cero, debido a que se alcanzó el final del buffer, o bien se finalizó el archivo, se invoca a la función `_filbuf`.

Es ésta la responsable de verificar entonces si es necesario cargar un nuevo bloque en el buffer o retornar el valor -1 de EOF.

Nótese la presencia del **operador ternario** (`? :`), que es una característica de sintaxis de C para representar abreviadamente una decisión `if-else`.

⁴ Recordemos que el modificador `inline` solicita al compilador que el cuerpo de la función se reemplace en donde se encuentra el llamado, en lugar de realizar una verdadera llamada a función. La declaración de `getc` que se muestra se ha simplificado un poco, con fines educativos.

ENGAÑANDO A GETC

Dado que getc utiliza el campo _cnt para su invocación a _filbuf, la que en definitiva es la que informa la finalización del archivo, observemos qué ocurre al modificar el contenido del campo _cnt para “hacer creer” a getc que restan mayor cantidad de bytes de los que realmente hay esperando en el buffer.

EJEMPLO: MODIFICACIÓN DE ESTRUCTURA FILE

Es una modificación del programa de lectura en la que se declara como entera la variable car, y se agrega la lectura del campo _cnt de la estructura FILE.

Posteriormente se modifica el valor de dicho campo. Si esta línea es suprimida, el programa funciona correctamente. Se sugiere correrlo sin esta línea, en primera instancia, para luego habilitarla y observar la diferencia.

```
/* Abre el archivo de texto "TEXTO.DAT" y lo muestra en pantalla */
/* Muestra el valor del campo _cnt y lo modifica */

#include <stdio.h>
int main ( )
{
    FILE *fp ;
    int car ;

    if ( ( fp = fopen ( "BINARIO.DAT" , "r" ) ) == NULL ) {
        printf ( " No se puede abrir el archivo. \n\n " ) ;
        return 1;
    }

    car = getc ( fp ) ; /* Lee un caracter del archivo */

    printf ( "_cnt = %d \n", fp->_cnt ) ;
    printf ( "Caracteres a ser leídos (luego del primero) \n " ) ;

    /* Esta línea debe suprimirse para observar */
    /* el funcionamiento correcto del programa */
    fp->_cnt = 15 ;      /* Modificación del campo _cnt */

    while ( car != EOF ) {
        printf("%X ", car ) ;
        car = getc ( fp ) ;
    }

    fclose ( fp ) ;
    printf ( " \n\n Fin del archivo. \n " ) ;
}
```

Se sugiere cargar al archivo BINARIO.DAT con una secuencia que incluya al par FF_H FF_H entre los valores, a fin de observar que al operar sobre la variable entera car, el programa no fue engañado por el falso EOF.

En cambio, sí fue engañado al modificarle el valor del campo _cnt, mostrando una serie de valores que se encuentran en el buffer, pero que no pertenecen al archivo.

Observe los valores resultantes de la ejecución y plantéese las siguientes preguntas:

- ¿Lee realmente EOF la función fgetc ?
- ¿Dónde está el carácter EOF de fin de archivo al final de la secuencia?
- ¿Existirá realmente?
- ¿Cuál es el tamaño del archivo según el sistema operativo?

DETECCIÓN DE ERRORES

La macro ferror() comprueba si hubo errores en la operatoria de un archivo relacionado al flujo que se le indica. Está incorporada al estándard ANSI C, y su definición se encuentra en la cabecera stdio.h .

Su prototipo es:

```
int ferror ( FILE * fp ) ;
```

ferror requiere que se le pase como argumento el puntero al flujo, y retorna *cero* si no se produjo ningún error, o un valor *distinto de cero* si lo hubo.

La definición de esta macro en stdio.h es como sigue :

```
#define ferror(F) ( (F)->_flag & _IOERR )
```

Mientras que la macro _IOERR se define como:

```
#define _IOERR 0x0020 /* I/O error from system */
```

Vemos entonces que se trata simplemente de retornar el valor contenido en el bit de errores perteneciente al campo _flag de la estructura FILE asociada al flujo en cuestión. Esto es similar a la operatoria de fclose por lo que no se entrará en más detalles.

Este indicador permanece activo luego de producirse el error, hasta que se ejecuta una llamada a rewind() o se cierra el archivo (fclose). Todo esto relativo al flujo en cuestión.

En el próximo ejemplo práctico se utilizará esta característica de detección de errores al manejar archivos, además de trabajar con dos flujos a la vez.

COPIA DE ARCHIVOS

Se realizará la copia de un archivo origen a otro de destino. Los nombres de ambos archivos se ingresarán como argumentos de la función `main`, por lo que este programa debe ser ejecutado desde la línea de comandos. El lector podrá modificarlo para realizar un ingreso por teclado si así lo desea.

```
/* Copia un archivo en otro con el formato "copiar origen destino" */
/* La copia la realiza en formato binario */
/* Realiza chequeo de errores */

#include <stdio.h>

int main ( int argc , char *argv [ ] )
{
    FILE *entrada , *salida ;
    char car ;
    int ERROR = 0 ;

    if ( argc != 3 ) {
        printf ( " Falta nombre de archivo. \n\n " ) ;
        printf ( " El formato es: copiar origen destino \n\n " ) ;
        return 1 ;
    }

    if ( ( entrada = fopen ( argv[1] , "rb" ) ) == NULL ) {
        printf ( "Error apertura archivo origen. \n\n " ) ;
        return 1 ;
    }

    if ( ( salida = fopen ( argv[2] , "wb" ) ) == NULL ) {
        printf ( "Error apertura archivo destino. \n\n " ) ;
        return 1 ;
    }

    car = getc ( entrada ) ;
    while ( !feof ( entrada ) && !ERROR ) {
        putc ( car , salida ) ;
        car = getc ( entrada ) ;
        ERROR |= ferror ( entrada ) ;
        ERROR |= ferror ( salida ) ;
    }

    ERROR |= fclose ( entrada ) ;
    ERROR |= fclose ( salida ) ;
    if ( !ERROR ) printf ( "\n1 Archivo copiado " ) ;
    else printf ( "\n1 Error al copiar " ) ;
}
```

En este programa se chequea el correcto ingreso de los argumentos de la función `main`, luego se abren los respectivos archivos, detectando posibles errores.

La copia se realiza en forma binaria abriendo el archivo origen para lectura binaria, y el archivo destino para escritura binaria. Por cada transferencia se verifican errores. En caso de producirse, se aborta el lazo `while`.

Por ultimo, se cierran ambos archivos, verificando además que no se produzcan errores en este paso.

Para ello se utiliza el *flag* `ERROR`, el cual será distinto de cero si algo anduvo mal debido a la utilización de las operaciones `OR`.

TRANSFERENCIA DE STRINGS

ANSI C considera dos funciones para realizar el envío de strings a un archivo, y la lectura de strings del mismo. Estas funciones son `fputs()` y `fgets()`.

FUNCIÓN FPUTS

Esta función se utiliza para enviar un string terminado con carácter nulo, desde la memoria hacia el flujo. La cantidad de caracteres transferidos depende de la ubicación del carácter nulo, es decir que no se especifica *a priori* cuál será dicha cantidad. El carácter nulo de finalización de string no se envía.

Su prototipo es:

```
int fputs ( const char * string , FILE * fp ) ;
```

Los argumentos transferidos a la función son la dirección de inicio en memoria del string, y el puntero al flujo.

La función retorna el último carácter enviado al flujo, en caso de que todo haya funcionado correctamente, o `EOF` si hubo algún error.

El lector puede comparar el comportamiento de esta función con `puts()`. De hecho se puede utilizar `fputs()` para enviar un string a pantalla con la forma:

```
fputs ( "texto" , stdout ) ;
```

FUNCIÓN GETS

La función `fgets()` lee caracteres del flujo especificado hasta que encuentra un carácter de salto de línea ('`\n`'), o un `EOF`, o bien, se ha llegado al límite de caracteres especificado para la actual transferencia.

En todos los casos, almacena los caracteres en memoria, a partir de la dirección indicada, y finaliza la cadena almacenando un carácter nulo.

Requiere como argumentos la dirección de memoria a partir de la cual almacenará el string, la cantidad máxima (más uno) de los caracteres que se desean transferir, y el puntero al flujo.

El prototipo es el siguiente:

```
char * fgets ( char * buffer , int maximo , FILE * fp ) ;
```

Retorna en caso de éxito un puntero a la cadena que se transfirió, c un puntero nulo en caso de haberse producido un error o encontrar el fin de archivo.

Será necesario en este último caso ejecutar `ferror()` o `feof()` para determinar exactamente qué es lo que ocurrió.

`maximo` es la cantidad máxima de caracteres que la función podrá almacenar en el buffer asignado en memoria para la string.

Dado que hay que contemplar el almacenamiento del carácter nulo de finalización de string, la cantidad máxima de caracteres transferidos, será `maximo-1`.

EJEMPLO: USO DE FPUTS CON STDOUT

Este es un sencillo programa que muestra el comportamiento de la función `fputs()`. Observe la diferencia al reemplazarla con `puts()`.

```
/* Muestra el funcionamiento de la función fputs( ) */
#include <stdio.h>

int main()
{
    fputs ( "Hola, esta es mi primera frase" , stdout ) ;
    fputs ( "Hola, esta es mi segunda frase" , stdout ) ;
}
```

EJEMPLO: USO DE FGETS CON STDIN

En este ejemplo se muestra el efecto de aplicar `fgets()` a una lectura desde teclado utilizando el flujo estándar `stdin`.

Ingrese frases de más de 10 caracteres a fin de observar el recorte producido por la limitación de `maximo` (en este caso, 10).

Compruebe que dicho recorte se produce en `maximo-1`.

```
#include <stdio.h>

int main()
{
```

```

    char vec [ 20 ] ;
    printf ( "Ingrese una frase \n\n" ) ;
    fgets ( vec , 10 , stdin ) ;
    puts ( vec ) ;
}

```

EJEMPLO: STRINGS Y ARCHIVOS

El programa mostrado a continuación ilustra el funcionamiento de las funciones fputs y fgets manejando los strings de un archivo de texto.

El programa crea el archivo de texto FRASES.TXT al cual ingresa 4 strings utilizando la función fputs.

Posteriormente lo abre para lectura y muestra el resultado de haber realizado la lectura con fgets en primer término, y repite el proceso para una lectura realizada con getc.

```

/* Muestra el comportamiento de las funciones fputs( ) y fgets( ) */
#include <stdio.h>
int main ( )
{
    char vec[20] , C ;
    FILE * fp ;      int I ;

    /* Creación del archivo */
    fp = fopen ( "FRASES.TXT" , "w" ) ;
    printf ( "Ingrese 4 frases \n\n" ) ;
    for ( I = 0 ; I < 4 ; I++ ) {
        gets ( vec ) ;
        fputs ( vec , fp ) ;
    }
    fclose ( fp ) ;

    /* Primera lectura del archivo */
    fp = fopen ( "FRASES.TXT" , "r" ) ;
    printf ( "\n\n\n" ) ;
    while ( !feof ( fp ) ) {
        fgets ( vec , 20 , fp ) ;
        puts ( vec ) ; }
    fclose ( fp ) ;

    /* Segunda lectura del archivo */
    fp = fopen ( "FRASES.TXT" , "r" ) ;
    printf ( "\n\n\n" ) ;
    while ( !feof ( fp ) ) {
        C = getc ( fp ) ;
        putchar ( C ) ; }
    fclose ( fp ) ;
}

```

En este caso se omitieron las comprobaciones de apertura del archivo para obtener un ejemplo más corto y sencillo. Recomendamos colocarlas siempre.

Observe el comportamiento del programa y mejórelo colocando en el lugar indicado, la siguiente línea:

```
putc ( '\n', fp ) ;
```

LECTURA Y ESCRITURA CON FORMATO

Así como se disponen de las funciones de lectura y escritura con formato `printf` y `scanf` para lectura de teclado y escritura en pantalla, ANSI C contempla dos funciones más generales, de la misma naturaleza, para lectura y escritura con formato en un flujo.

Estas funciones son `fprintf()` y `fscanf()`.

Su funcionamiento es similar a las anteriores, excepto que necesitan recibir como argumento adicional un puntero a un flujo.

Sus prototipos se encuentran en `stdio.h` y se muestran a continuación :

```
int fprintf( FILE * fp, const char * cadena_de_formato, argumentos...);  
int fscanf( FILE * fp, const char * cadena_de_formato, direcciones...);
```

El valor returnedo por `fprintf` es la cantidad de caracteres enviados, mientras que para `fscanf` representa la cantidad de caracteres recibidos, o EOF en caso de haberse detectado el final del archivo.

El resto de los argumentos resulta conocido dado que son similares a los de las funciones `printf` y `scanf`, con excepción del puntero al flujo ya comentado.

Estas funciones no son muy adecuadas para almacenar información en disco, dado que al guardar con formato será necesario recuperar con el mismo formato para evitar errores.

Por otro lado, el almacenamiento numérico es poco eficiente porque se envían al flujo los caracteres ASCII que representan a los números. Esto se verá en un ejemplo posterior.

Una utilización mas adecuada es el envío de información con formato informando errores (`stderr`).

EJEMPLO: FUNCIÓN FPRINTF

En este programa se realiza la comparación entre los archivos `ARCH1` y `ARCH2` .

En ambos se almacena el valor 105. En el primer archivo, esto se realiza mediante el envío de un byte, mientras que en el segundo se utiliza una impresión con formato.

Observe el tamaño de ambos archivos desde el S.O.

```
#include <stdio.h>
int main ( )
{
    FILE *fp ;
    int NUM = 105 ;

    fp = fopen("ARCH1","wb");
    putc( NUM , fp ) ;
    fclose ( fp ) ;

    fp = fopen("ARCH2","wb");
    fprintf( fp , "%d" , NUM ) ;
    fclose ( fp ) ;

    printf("Contenido del archivo ARCH1 : " );
    fp = fopen("ARCH1","rb");
    NUM = getc ( fp ) ;
    while ( !feof ( fp ) ) {
        putchar ( NUM ) ;
        printf ( " \nInterpretacion numerica : %d \n\n" , NUM ) ;
        NUM = getc ( fp ) ;
    }
    fclose ( fp ) ;

    printf("Contenido del archivo ARCH2 : " );
    fp = fopen("ARCH2","rb");
    while ( !feof ( fp ) ) {
        NUM = getc ( fp ) ;
        putchar ( NUM ) ;
    }
    fclose ( fp ) ;
}
```

ARCHIVOS BINARIOS

Como se mencionó anteriormente, los archivos binarios son aquellos en los que la información se guarda a nivel de byte sin realizar transformación alguna. Por lo que la cantidad de bytes enviados al flujo será igual a la cantidad de bytes que se almacenarán.

Cuando se intenta transferir una cantidad de bytes determinada, o bien ubicar el n-ésimo byte dentro del archivo, es importante que el mismo se haya abierto con formato binario para evitar errores.

Si bien las funciones que se describen en esta sección también funcionan en archivos de texto, pueden ser causa de los errores antes mencionados. Por esta razón es que se las considera adecuadas sólo para flujos binarios.

TRANSFERENCIA DE BLOQUES DE BYTES

Consideremos la lectura o escritura en el archivo de un bloque de información de una cantidad de bytes determinada.

Se disponen de un par de funciones contempladas por el estándar que realizan estas operaciones: `fread()` y `fwrite()`.

Sus prototipos se muestran a continuación:

```
size_t fread(void * buffer, size_t tamaño, size_t numero, FILE * fp );  
  
size_t fwrite(const void * buffer, size_t tamaño, size_t numero, FILE * fp ) ;
```

Donde `size_t` es un tipo de dato usado para especificar tamaño de objetos en memoria. Está definido de la siguiente manera:

```
typedef unsigned int size_t ;
```

A los fines prácticos deberá ser considerado como un entero no signado.

Las funciones `fread` y `fwrite` realizan la transferencia desde y hacia el flujo especificado por `fp`, de una cantidad `numero` de bloques, de `tamaño` bytes hacia o desde un área de memoria que comienza en la dirección indicada por `buffer`.

Los valores retornados representan la cantidad de bytes efectivamente transferidos. Si en el caso de `fwrite` la cantidad retornada es menor a la esperada, se puede suponer la ocurrencia de un error.

Si esto mismo ocurre con `fread`, será necesario ejecutar `ferror` o `feof`, a fin de determinar si lo ocurrido fue un error o bien se alcanzó el fin de archivo.

Nótese que, dado que no se conoce el tipo de dato a albergar en el `buffer`, el puntero al mismo se especifica en estas funciones, como `void *`. Lo estrictamente correcto al usarlas sería realizar un casting para adecuar el tipo.

Una manera de enviar un dato, por ejemplo un flotante, a un archivo mediante `fwrite()` se muestra a continuación:

```
float F ;
F = 2.7865 ;
fwrite ( &F , sizeof(float) , 1 , fp ) ;
```

Nótese el empleo de `sizeof()` al indicar el tamaño del bloque, a fin de asegurar portabilidad del código.

Dado que `fread` y `fwrite` transfieren bloques de información de un tamaño dado, resultan ideales para transferir estructuras y de esta manera tener el equivalente a un vector de estructuras en el disco.

Esta es una de las aplicaciones primarias que se mostraron en un principio. A continuación veremos unos ejemplos que ilustran el uso de `fread` y `fwrite` para implementar esta estructura de archivo.

EJEMPLO: VECTOR DE ESTRUCTURAS EN ARCHIVO

Se declara una estructura de datos cuyos campos son el nombre, el sexo y la edad de una persona. Se ingresan por teclado los datos de una cierta cantidad de personas (este número se pregunta previamente) y se los envía al archivo DATOS.DAT.

```
/* Genera el archivo DATOS.DAT con un cierto numero de estructuras */
/* La cantidad de éstas se pide por teclado */

#include <stdio.h>
int main ( )
{
    struct {
        char nombre [ 20 ] ;
        char sexo ;
        int edad ;
    } datos ;
    FILE *puntero ;
    int i , N ;

    if ( ( puntero = fopen ( "DATOS.DAT" , "wb" ) ) == NULL ) {
        printf ( "Error apertura archivo. \n\n" ) ;
        return 1;
    }
    printf ( "Indique cuántos datos ingresará = " ) ;
    scanf( "%d" , &N ) ;

    for ( i=0 ; i < N ; i++ ) {
        printf ( "\nNOMBRE,SEXO Y EDAD \n" );
        fflush ( stdin ) ;
        gets ( datos.nombre ) ;
        datos.sexo = getchar ( ) ;
```

```

        scanf ( "%d" , &datos.edad ) ;
        fwrite ( &datos , sizeof(datos) , 1 , puntero ) ;
    }

    fclose ( puntero ) ;
}

```

EJEMPLO: LECTURA DE VECTOR DE ESTRUCTURAS DESDE ARCHIVO

Este programa complementa al anterior. Abre el archivo DATOS.DAT y lee las estructuras antes mencionadas para mostrarlas en pantalla, hasta detectar el fin de archivo.

De esta manera puede visualizarse la corrección de los procesos de escritura y lectura del archivo.

```

/* Lee e imprime las estructuras del archivo DATOS.DAT */
/* hasta llegar al final del archivo. */

#include <stdio.h>
int main()
{
    struct {
        char nombre[20] ;
        char sexo ;
        int edad ;
    } datos ;
    FILE *puntero ;
    int i ;

    if ( ( puntero = fopen ( "DATOS.DAT" , "rb" ) ) == NULL ) {
        printf ( "Error apertura archivo. \n\n" ) ;
        return 1 ;
    }

    printf ( " \n \t\t\t Contenido del archivo \n\n" ) ;
    fread ( &datos , sizeof ( datos ) , 1 , puntero ) ;
    while ( !feof ( puntero ) ) {
        printf("\n%30s%10c%10d", datos.nombre,
                           datos.sexo,datos.edad);
        fread ( &datos , sizeof(datos) , 1 , puntero ) ;
    }

    fclose ( puntero ) ;
}

```

INTERPRETACIÓN DEL MODELO DE VENTANA

Las funciones `fread` y `fwrite` le dan una opción particular al modelo de ventana. Podemos considerar que estamos eligiendo el tamaño de la ventana de transferencia mediante la selección del tamaño del bloque y la cantidad de bloques.

El **tamaño** de ventana se establece para cada transferencia particular y puede variar de una a otra.

Podemos decir que:

$$\text{Tamaño de ventana} = \text{Tamaño de bloque} * \text{Número de bloques}$$

ARCHIVOS RANDOM

Los archivos tratados hasta ahora se comportan de manera secuencial. Cada vez que se ingresa o se lee un bloque del archivo, la ventana avanza una cantidad de bytes equivalente a la longitud del bloque transferido.

Podríamos tener un acceso directo al archivo si pudieramos colocar el inicio de la ventana en una determinada posición.

Esto lo podemos lograr recurriendo al servicio de la función `fseek()`.

De esta manera completamos el manejo de la ventana, manejando no solo su tamaño sino también su posición.

Esta posibilidad no anula el comportamiento de acceso secuencial al archivo, dado que luego de la transferencia, la ventana se desplazará hacia adelante la cantidad de bytes transferida (queramos, o no).

Simplemente se agrega al comportamiento secuencial del archivo, la posibilidad de accesos *random* por reposicionamiento de la ventana.

Si deseamos acceder a posiciones sucesivas, pero separadas una cantidad de bytes distintos del tamaño de la ventana, será necesario reposicionar la misma cada vez que se realice una transferencia.

FUNCIÓN FSEEK

La función `fseek` está contemplada por el estándar ANSI C y su prototipo se encuentra en la cabecera `stdio.h`.

Considerando el modelo de ventana, podemos decir que mueve el **tope** de la misma a una posición determinada por un *offset* y una referencia a partir de la cual se considera el *offset*.

Su prototipo es como se muestra a continuación:

```
int fseek ( FILE * fp , long offset , int referencia ) ;
```

El valor retornado si el posicionamiento tuvo éxito es *cero*. Si hubo algún error, como ser en el caso de intentar un posicionamiento anterior al origen, el resultado es *distinto de cero*.

Si se intenta posicionar la ventana en un flujo en el que no tenga aplicación hacerlo (por ejemplo, en un flujo asociado con un dispositivo externo que no puede realizar accesos *random*, como podría ser una comunicación serie) el valor retornado carece de sentido.

Los valores válidos para `referencia` son tres y están definidos como macros en `stdio.h`. Se las detalla a continuación:

<i>Macro</i>	<i>Valor</i>	<i>Significado</i>
<code>SEEK_SET</code>	<code>0</code>	Principio del archivo
<code>SEEK_CUR</code>	<code>1</code>	Posición actual
<code>SEEK_END</code>	<code>2</code>	Final del archivo

En flujos de texto no se puede asegurar el correcto posicionamiento debido al reemplazo de caracteres ya estudiado.

En flujos abiertos en modo *append*, la escritura se realiza siempre al final del archivo, independientemente del posicionamiento de la ventana.

EJEMPLO: USO DE FSEEK

Se mostrará a continuación un ejemplo de acceso *random* a partir de la posición inicial.

El programa abre y muestra el archivo `DATOS.DAT` cargado anteriormente. Luego de esto solicita sucesivas posiciones para acceder en forma directa al archivo utilizando `fseek`.

Finaliza el programa con un ingreso negativo.

```
#include <stdio.h>
int main ( )
{
    struct {
        char nombre[20] ;
        char sexo ;
        int edad ;
        } datos ;
    FILE *puntero ;
    int I ;
```

```

if ( ( puntero = fopen ( "datos.dat" , "rb" ) ) == NULL ) {
    printf ( "Error apertura archivo. \n\n" ) ;
    return 1 ;
}

fread ( &datos , sizeof(datos) , 1 , puntero ) ;
while ( !feof(puntero) ) {
    printf ("\\n%12s%4c%4d" ,
           datos.nombre , datos.sexo , datos.edad ) ;
    fread ( &datos , sizeof(datos) , 1 , puntero ) ;
}
printf ( " \\n\\n\\nIngrese la posicion ( -1 para salir ) : " );
scanf ( "%d" , &I ) ;
while ( I >= 0 ) {
    fseek ( puntero , I * sizeof(datos) , SEEK_SET ) ;
    fread ( &datos , sizeof(datos) , 1 , puntero ) ;
    printf("\\n%12s%4c%4d",datos.nombre,datos.sexo,datos.edad);
    printf ( "\\n\\n\\nIngrese la posicion (-1 para salir) :");
    scanf ( "%d" , &I ) ;
}

fclose ( puntero ) ;
printf ( " \\n Fin del programa " );
}

```

EJEMPLO: USO DE FSEEK II

Este ejemplo muestra el archivo INFORMAT.DAT almacenándolo en un vector a fin de mostrarlo sucesivamente. Permite el ingreso de distintos valores de *offset* y diversas referencias a fin de visualizar el comportamiento de *fseek*.

Recuérdese que al terminar de leer el archivo, la posición de la ventana está al final del mismo. Por esta razón se realiza un *fseek* a la posición inicial.

```

#include <stdio.h>
int main ( )
{
    struct {
        char nombre[20] ;
        char sexo ;
        int edad ;
    } datos[30] , provi ;
    FILE *puntero ;
    int num , ref , I ;

    if ( ( puntero = fopen ( "inform.dat" , "rb" ) ) == NULL ) {
        printf ( "Error apertura archivo. \\n\\n" ) ;
        return 1 ;
    }

```

```

num = 0 ;
fread ( &provi , sizeof(provi) , 1 , puntero ) ;
while ( !feof(puntero) ) {
    *(datos+num) = provi ;
    num++ ;
    fread ( &provi , sizeof(provi) , 1 , puntero ) ;
}

fseek ( puntero , 0 , SEEK_SET ) ; /* Posiciona en el inicio */

while ( ref != 9 ) {
    for ( I = 0 ; I < num ; I++ )
        printf ( " \n\t%3d\t\t%12s%4c%4d" , \
                 I,datos[I].nombre,datos[I].sexo,datos[I].edad);

    printf ( "\n\nPara salir referencia = 9 " ) ;
    printf ( "\nIngrese la posicion y referencia (0,1,2)" ) ;
    printf ( "\nSEEK_SET 0 \t SEEK_CUR 1 \t SEEK_END 2 \n" ) ;
    scanf ( "%d %d" , &I , &ref ) ;
    switch ( ref ) {
        case 0: fseek(puntero,I*sizeof(provi),SEEK_SET);
                  break;
        case 1: fseek(puntero,I*sizeof(provi),SEEK_CUR);
                  break;
        case 2: fseek(puntero,(long)I*sizeof(provi),SEEK_END);
                  break;
    }

    fread ( &provi , sizeof(provi) , 1 , puntero ) ;
    printf("\n%12s%4c%4d", provi.nombre, provi.sexo, \
                      provi.edad ) ;
}

fclose ( puntero ) ;
}

```

EJEMPLO: INSPECCIÓN DE ARCHIVO BINARIO

En este ejemplo se lee en formato binario un archivo de cualquier tipo, y se lo muestra en pantalla en bloques de 128 bytes.

El nombre del archivo a inspeccionar se ingresa como argumento del `main`, de la forma: `INSPEC nombre_archivo`

El número de bloque se ingresa por teclado, y se sale del programa ingresando un valor negativo.

La función `fseek` posiciona la ventana en el byte equivalente a `128 * bloque`, y mediante la función `fread` se leen 128 bytes (tamaño de la ventana). Se comproba el valor returned por `fread` para saber cuántos caracteres se transfirieron en realidad. Este valor se almacena en la variable `numleidos` y es posteriormente transferido a la función `mostrar`.

Esta función recibe la cantidad de bytes leídos realmente del disco (pueden ser menos que 128 en caso de fin de archivo o error) y produce una impresión hexadecimal y ASCII de cada byte. Cuando la impresión tipo carácter no es posible se la reemplaza por un punto (.). Esta situación se detecta mediante el empleo de la función `isprint` cuyo prototipo se encuentra en la cabecera `ctype.h`.

```
#include <stdio.h>
#include <ctype.h>
#define TAM 128
void mostrar(int num);
unsigned char buf[TAM];

int main( int argc, char* argv[])
{
    FILE * fp;
    int sector, numleidos;

    if(argc != 2) {
        printf("Uso: INSPEC nombre_archivo");
        return 1;
    }

    if( (fp=fopen(argv[1]) , "rb")) == NULL) {
        printf("No se puede abrir el archivo");
        return 1;
    }

    printf("Para salir ingrese un negativo");
    printf("Introduzca sector: ");
    scanf("%d", &sector);
    while(sector >= 0) {
        if( fseek(fp, sector * TAM, SEEK_SET))
            printf("Error de búsqueda \n");
        if((numleidos = fread(buf, 1, TAM, fp)) != TAM)
            printf("Final del archivo \n");

        mostrar(numleidos);
        printf("\nBytes transferidos: %d", numleidos);
        printf("Introduzca sector: ");
        scanf("%d", &sector);
    }
}
```

```

void mostrar(int numleidos) {
    int i, j;

    for(i=0; i<numleidos; i++) {
        for(j=0; j<16; j++)
            printf(" %02X", buf[ i*16 + j ]);
        printf(" ");
        for(j=0; j<16; j++)
            if(isprint(buf[i*16 + j]))
                printf("%c", buf[i*16 + j]);
            else printf(".");
        printf("\n");
    }

    /* Muestra la última linea en caso de fin de archivo */
    for(j=0; j < numleidos % 16 ; j++)
        printf(" %02X", buf[i*16+j]);
    for(j=numleidos%16; j<16; j++)
        printf(" ");
    printf(" ");
    for(j=0; j<numleidos%16; j++)
        if(isprint(buf[i*16+j]))
            printf("%c", buf[i*16 + j]);
        else printf(".");
}

```

FUNCIÓN FTELL

Si bien en el acceso a archivos en modo *random* es el usuario quien posiciona la ventana, no siempre se conoce exactamente esta posición.

Puede resultar útil entonces tener la posibilidad de averiguar cuál es el lugar en que se encuentra el tope de la ventana en un momento dado.

La función `fte11` nos informa la cantidad de bytes que hay entre el inicio del archivo y la posición actual de la ventana.

Su prototipo es:

```
int fte11 ( FILE * fp ) ;
```

La función `fte11` requiere como argumento el puntero al flujo y retorna un entero. Este valor es la cantidad de bytes de *offset* que tiene la posición actual de la ventana con respecto al inicio del archivo. Si se produjo algún error retorna -1.

EJEMPLO: BUSCAR UNA STRING EN UN ARCHIVO

El siguiente programa abre un archivo en modo de lectura binaria con el objeto de buscar una string dentro de él e informar en qué posición se encuentra, todas las veces que esto ocurra, o bien informar si no se lo encontró.

El nombre del archivo a investigar y la string a buscar se ingresan por teclado.

```
#include <stdio.h>
#include <string.h>
int main()
{
    FILE * fp;
    char BUF[30], buscar[30], archivo[30];
    int L, cont=0, encontrada=0;

    printf("Ingrese palabra a buscar y nombre de archivo: ");
    gets(buscar);
    gets(archivo);

    fp = fopen(archivo, "rb"); /* Omitimos el chequeo de error */
    L = strlen(buscar);

    while(!feof(fp)) {
        fseek(fp, cont, SEEK_SET);
        cont++;
        fread(BUF, L, 1, fp);
        BUF[L] = '\0';

        if(!strcmp(BUF, buscar)) {
            printf("\nEncontrada en %d", ftell(fp)-L);
            encontrada=1;
        }
    }
    if(!encontrada) printf("\nNo se encontró");
}
```

La string a buscar se encuentra en el vector `buscar`. Se mide su longitud (sin el carácter nulo) utilizando la función `strlen` y se almacena en la variable `L`.

Se utiliza un vector auxiliar llamado `BUF`, en el cual se almacena en su posición `L` el carácter nulo, con el fin de prepararlo para la comparación de strings con la función `strcmp`.

Se trabaja con una ventana de tamaño `L`, la cual se va moviendo de un byte por vez, transfiriendo `L` bytes hacia `BUF` para comparar con la palabra buscada. Dado que la transferencia es de `L` bytes, también lo es el desplazamiento de la ventana, por lo que será necesario reposicionarla cada vez con `fseek`.

Si se detecta igualdad entre BUF y buscar ~~se informa la posición~~ de la ventana en el momento de la transferencia. Esta información se obtiene de `fseek` restando al retorno el valor L, dado que la ventana ya se desplazó esa cantidad de bytes.

El proceso continúa hasta detectar el fin de archivo con `feof`.

EJEMPLO: TAMAÑO DE UN ARCHIVO

Un ejemplo muy simple, pero interesante, es el de medir la longitud o tamaño de un archivo utilizando las funciones `fseek` y `fseek`.

Simplemente se abre el archivo a medir, para lectura binaria. Luego se posiciona la ventana en el final del archivo por medio de `fseek` y se pide a `fseek` la posición de ésta, que será retornada en bytes desde el origen.

Compare el valor obtenido con el que informa el sistema operativo. El archivo a medir puede ser de cualquier tipo.

```
#include <stdio.h>
int main() {
    FILE * fp;

    if( ! (fopen("PRUEBA.DAT", "rb")) ) {
        printf("Error al abrir el archivo");
        return 1;
    }

    fseek(fp, 0, SEEK_END);
    printf("El tamaño del archivo en bytes es %u", ftell(fp));
}
```

OTRAS FUNCIONES DE ARCHIVOS

Veremos ahora la descripción de otras funciones del estándar que se utilizan para trabajar con flujos o archivos, aunque no tan frecuentemente como las que hemos descripto hasta aquí.

REWIND

Esta función mueve el tope de la ventana al inicio del archivo, de la misma forma que `fseek`. A diferencia de ésta, también pone a cero los flags de error y fin de archivo.

Su prototipo es:

```
void rewind ( FILE * fp ) ;
```

Solamente requiere el puntero al flujo, y no retorna valores.

FFLUSH

La función `fflush` está destinada a forzar el flujo de un buffer a su salida natural. Numerosos libros de texto indican que `fflush` "fluye" los bytes que permanecen en determinado buffer hacia el archivo, cuando éste está abierto en modo escritura, y vacían el buffer si se abrió en modo lectura.

Esto último obedece al razonamiento de que "mal puedo forzar un ingreso, si yo mismo soy el responsable de la lectura... la información fluye hacia ningún lado y se pierde".

Por ejemplo, si tomamos caracteres de un buffer de entrada con `A=getc(fp)` y luego forzamos el flujo de entrada, pero no colocamos nuevamente la línea anterior, el flujo de caracteres no tiene dónde ir y se pierde.

Sin embargo, según el estándar ANSI, `fflush` es una función definida solamente para flujos de salida, y no para flujos de entrada. Puede ocurrir que cada compilador en particular le agregue la posibilidad de vaciar el buffer de entrada, o no.

Su prototipo es:

```
int fflush ( FILE * fp ) ;
```

En numerosos ejemplos hemos utilizado `fflush` para eliminar el `<Enter>` remanente en el buffer de teclado cuando se realizan lecturas numéricas y textuales alternadas.

REMOVE

Si bien esta función no está relacionada con los flujos, sí está ligada directamente al manejo de archivos de disco. Por esa razón la presentamos aquí. Nótese que no hay ningún puntero a flujo involucrado.

El efecto de esta función es eliminar del disco el archivo cuyo nombre y *path* se le entrega como argumento.

Su prototipo es:

```
int remove ( const char * archivo_a_borrar ) ;
```

Si `remove` tuvo éxito retorna *cero*, en caso contrario retorna `EOF`.

Posibles errores pueden ser: la inexistencia del archivo, la protección de éste contra borrado, la falta de permisos para completar la operación, etc.

En caso de producirse un error se carga la variable global `errno` con un número descriptivo de lo sucedido.

RENAME

Esta función también maneja archivos y no flujos, al igual que la anterior.

`rename` cambia el nombre asignado a un archivo de disco, permitiendo indicar el *path*. De esta manera, no solamente renombra el archivo sino que además lo mueve al nuevo *path* seleccionado. Es decir que esta función puede utilizarse para renombrar un archivo, para moverlo o para ambas cosas a la vez.

No se permite mover directorios completos ni tampoco son aceptados los comodines para actuar sobre varios archivos simultáneamente.

Su prototipo es:

```
int rename( const char * old_name , const char * new_name ) ;
```

El valor returnedo es *cero* si tuvo éxito, o `EOF` en caso de error. De la misma forma que `remove`, de producirse un error, se cargará la variable global `errno` con un código descriptivo de éste.

TMPFILE

La función `tmpfile` abre un **archivo temporal** en el directorio actual o en el indicado por el sistema operativo para estos fines. Este archivo está destinado a funcionar como un archivo auxiliar que, como su nombre lo indica, luego será descartado.

El archivo se crea en modo `w+b`, es decir que estará vacío y podrá leerse y escribirse como flujo binario. El archivo temporal se elimina automáticamente cuando se cierra el flujo o cuando el programa finaliza normalmente.

Dado que el archivo no va a perdurar, su nombre es irrelevante. Por esta razón no hay acceso al nombre del archivo y solamente se tendrá como referencia a él el puntero al flujo. Además, esto permite que el sistema operativo administre los nombres de archivos temporales para evitar solapamientos entre temporales de distintos procesos en ejecución. En cualquier caso, es un problema menos para el programador.

Su prototipo es:

```
FILE * tmpfile ( void ) ;
```

El valor returnedo es el puntero al flujo en caso de funcionar normalmente, o el puntero `NULL` en caso de error.

EJEMPLO: USO DE ARCHIVO TEMPORAL

Se realizará una depuración del archivo `DATOS.DAT` utilizado anteriormente, eliminando de él los datos de las personas de sexo masculino.

Lo que haremos será transferir los datos desde el original de manera selectiva, es decir sólo aquellas personas que nos interesan (las de sexo femenino), hacia un archivo temporal. Luego se procede a reabrir el original en modo escritura binaria, de manera que su contenido se destruye, y se transfieren las personas presentes en el temporal hacia el archivo original definitivo.

Para ejecutar este ejemplo tenga en cuenta que se modificará el original DATOS.DAT, de manera que puede ser una buena idea *almacenar una copia antes* de probarlo, por si el programa no funcionara como se espera. Esto vale para cualquier programa que afecte a archivos y esté en fase de pruebas.

```
#include <stdio.h>
int main()
{
    struct {
        char nombre[20];
        char sexo;
        int edad;
    } datos;
    FILE *puntero, *tmp;

    if ((puntero = fopen("DATOS.DAT", "rb")) == NULL) {
        printf("Error al abrir el archivo");
        return 1;
    }

    if ((tmp = tmpfile()) == NULL) {
        printf("Error al crear temporal");
        return 1;
    }

    /* Copia de las mujeres al temporal */
    fread(&datos, sizeof(datos), 1, puntero);
    while(!feof(puntero)) {
        if(datos.sexo=='F')
            fwrite(&datos, sizeof(datos), 1, tmp);
        fread(&datos, sizeof(datos), 1, puntero);
    }
    fclose(puntero);
    rewind(tmp);
    puntero = fopen("DATOS.DAT", "wb");

    /* Copia del temporal al original */
    fread(&datos, sizeof(datos), 1, tmp);
    while(!feof(tmp)) {
        fwrite(&datos, sizeof(datos), 1, puntero);
    }
}
```

```
        fread(&datos, sizeof(datos), 1, tmp);
    }

    fclose(puntero);  fclose(tmp);
}
```

PROBLEMAS PROPUESTOS

1. Realice un programa que cree un archivo binario con bytes aleatorios, garantizando que en algún lugar del programa se encuentre la siguiente secuencia de 7 bytes: A8 5C 69 70 70 AF EF.
2. Realice un programa que reciba el nombre de un archivo binario por línea de comandos y determine si el mismo contiene, al menos una vez, la siguiente secuencia de 7 bytes: A8 5C 69 70 70 AF EF.

En caso afirmativo imprimir en pantalla “Archivo infectado”, o “Virus no detectado” en caso contrario. Utilice el resultado del ejercicio anterior.

3. Un archivo binario contiene estructuras con información de ventas de un comercio, consistentes en nombre del vendedor y monto de la venta.
Se pide estructurar la información y mostrar en pantalla el nombre del vendedor que haya vendido más (sumando los montos).
4. Un archivo binario **PRODUCTOS.DAT** contiene la información de los productos vendidos en un supermercado, consistentes en código de artículo (**int**), proveedor (**char** de 30) y precio (**float**).

En otro archivo binario **AUMENTOS.DAT** se encuentra un vector de estructuras conformadas por nombre de proveedor (**char** de 30) y un porcentaje (**float**). Se pide aplicar a los productos del primer archivo los porcentajes de aumento indicados en el segundo archivo. Todos los productos de un mismo proveedor deben recibir el aumento indicado en el segundo archivo.

Índice

Prólogo a la segunda edición	4
Prólogo a la primera edición.....	5

1. Diagramación..... 6

Construcción de programas	6
Proceso de diagramación.....	7
Importancia de la diagramación.....	12
Programación estructurada	12
Ciclo de vida del software	13
Teorema fundamental de Bohm y Jacopini	14
Lenguajes estructurados.....	15
Diagramas de Chapin o Nassi-Schneiderman	16
Variables	19
Contadores, Acumuladores y Flags	20
Ejemplos de diagramas de Chapin	21
Pseudocódigo.....	24
Desarrollo top-down	25
Problemas propuestos	28

2. Fundamentos de C..... 29

Estructura de un programa C.....	30
Palabras reservadas.....	31
Identificadores válidos.....	31
Tipos de datos.....	32
Modificadores.....	32
Fin de sentencia	35
Operador de asignación.....	36
Conversión de tipos	37
Operador sizeof	38
Operadores aritméticos.....	39
Pre y post incrementos y decrementos	39
Precedencia de operadores aritméticos	40
Jerarquía de tipos.....	41
Otros operadores.....	42

Entrada y salida de consola	42
Funciones de salida a pantalla	43
Funciones de entrada de teclado	49
Problemas propuestos	53
3. Control de flujo.....	55
Secuencia natural	55
Toma de decisión	56
Operadores relacionales	59
Operadores lógicos	60
Decisiones anidadas	62
Desarrollo top-down	80
Problemas propuestos	88
4. Funciones	90
Macros.....	90
Subrutinas.....	92
Variables globales.....	97
Variables locales	97
Parámetros formales	98
Funciones en el lenguaje C	98
Retorno de la función	99
Argumentos y parámetros	103
Funciones anidadas.....	106
Funciones que devuelven valores no enteros.....	108
Prototipo de una función	109
Funciones que retornan void.....	110
Prototipos y archivos cabecera	111
Transferencia por valor y por referencia.....	112
Recursividad.....	113
Problemas propuestos	115
5. Vectores	117
Declaración de vectores	118
El vector en la memoria	119
Subíndice numérico.....	120
Copia de vectores	121
Vectores sin tamaño	126
Acceso y búsqueda en vectores.....	128

Búsqueda secuencial	128
Búsqueda binaria	130
Búsqueda del mínimo	131
Ordenamiento de vectores	133
Método del burbujeo	135
Vectores apareados	137
Matrices	142
Pasaje de matrices a funciones	144
Problemas propuestos	149
6. Strings.....	151
Asignación directa de strings	151
Formato tipo string	152
Funciones para lectura de strings	154
Funciones para impresión de strings	156
Funciones de tratamiento de strings	158
Errores frecuentes	162
Vector de strings	164
Problemas propuestos	170
7. Estructuras	171
Declaración de estructuras	172
Acceso a los campos	174
Comparación entre estructuras y vectores	174
Pasaje de estructuras a funciones	178
Tipos globales y variables locales	181
Estructuras retornadas desde funciones	182
Vectores apareados vs. vectores de estructuras	183
Estructuras anidadas	186
Ejemplo integral con vector de estructuras	189
Problemas propuestos	193
8. Enumeraciones, Uniones, Campos y Operadores de bit.....	196
Enumeraciones	196
Uniones	198
Campos de bit	209
Operadores de bit	213
Problemas propuestos	226

9. Punteros.....	227
Tamaño de los punteros	227
Usos de los punteros.....	228
Declaración de tipo puntero	228
Operadores puntero.....	229
Asignación de punteros	230
Comparación de punteros.....	231
Aritmética de punteros	231
Relación entre punteros y vectores	234
Punteros y subíndices.....	236
Punteros a string.....	237
Punteros a estructura	239
Pasaje de estructuras a funciones (por referencia)	242
Punteros a punteros.....	243
Vectores de punteros	246
Punteros a función.....	251
Problemas propuestos	256
10. Argumentos del main.....	257
Tipo de los argumentos	257
Argumentos por línea de comandos	258
Variables de entorno	260
Problemas propuestos	262
11. Gestión dinámica de memoria	263
Asignación dinámica de memoria	264
Funciones de gestión dinámica de memoria.....	265
malloc()	265
calloc()	267
realloc()	267
free()	268
Problemas propuestos	269
12. Formato de punto flotante IEEE 754.....	270
Números de punto fijo.....	270
Precisión múltiple.....	271
Notación científica	272
Formato de punto flotante	272
Bit oculto	274

Polarización del exponente.....	274
Norma IEEE 754	275
Precisión de los números de punto flotante	278
Representación en la recta numérica.....	279
Error absoluto y relativo en punto fijo.....	281
Error absoluto y relativo en punto flotante	281
Distintos formatos de punto flotante	282
13. Comandos del preprocesador	284
#define	284
Usos de #define	285
Reemplazo de expresiones.....	285
Definición de constantes	287
Macros.....	289
Uso de #define para declarar macros	291
Comparación entre macros y funciones	295
#undef.....	297
#include.....	297
Cabeceras propias	298
Bibliotecas de macros.....	301
Compilación condicional	303
14. Archivos de disco.....	309
Flujos	311
Archivos en ANSI C.....	314
Apertura de archivo.....	314
Cierre de un archivo.....	317
Modos de acceso a los archivos.....	318
Modelo de ventana.....	318
Escritura de un byte en el archivo	319
Lectura de un byte desde un archivo.....	321
Uso de los flujos estándar stdin y stdout	322
Determinación del final de archivo.....	323
Reemplazo de caracteres en flujos de texto.....	325
Macro feof	326
Detección de errores.....	331
Copia de archivos	332
Transferencia de strings.....	333
Lectura y escritura con formato	336

Archivos binarios.....	337
Transferencia de bloques de bytes.....	338
Archivos random	341
Otras funciones de archivos	348
Problemas propuestos	353

C para Ingeniería Electrónica

La presente obra surge como compilación ordenada y adaptada de las publicaciones anteriores del Ing. Jorge A. Argibay en forma de apuntes. Estos son el testimonio escrito de sus propias clases frente a alumnos y contienen el resultado de más de 30 años de experiencia docente tanto en el ámbito universitario como privado.

Este libro se adapta a los temas de la materia **Informática I**, y cubre en gran medida **Informática II**, ambas de la carrera de **Ingeniería Electrónica** de la Universidad Tecnológica Nacional. Asimismo cubre totalmente los temas de la materia **Programación I** y gran parte de **Programación II** de la **Tecnicatura Superior en Programación**.

Esta **segunda edición** ha sido revisada y ampliada para cubrir los aspectos de los sistemas operativos, arquitecturas de computadoras y compiladores modernos sin perder el espíritu original de la obra.



Jorge Argibay se recibió de Ingeniero Electrónico en la Universidad de Buenos Aires y de Ingeniero en Telecomunicaciones en la Escuela Superior Técnica del Ejército. Realizó posgrados de Informática, Informática aplicada a Telecomunicaciones y Técnicas Digitales, Lenguaje VHDL y circuitos FPGA, entre otros. En la actualidad se desempeña como Director de Cátedra de las asignaturas Informática I y II, y Técnicas Digitales I en la carrera de Ingeniería Electrónica en la Facultad Regional Haedo de la UTN, dictando dichas materias; y como Director de la carrera Tecnicatura Superior en Programación en la misma Facultad, donde dicta las materias Programación I y II.



Mauro Gullino es Maestrando en Ingeniería de Software por la Universidad Nacional de La Plata y Licenciado en Comunicación Visual por la Universidad de Ciencias Empresariales y Sociales. Tiene una amplia trayectoria como desarrollador, consultor y docente en diversas universidades y empresas argentinas. En la actualidad es Adjunto a cargo de las materias Laboratorio III y IV de la Tecnicatura Superior, correspondientes a programación web. Dicta además Programación I y II de dicha carrera. Es Jefe de Trabajos Prácticos de Informática II en la carrera de Ingeniería Electrónica. Se ha desempeñado como docente en diversas carreras de grado en distintas universidades.

9 7899874 · 236012

ISBN 978-987-42-3601-2