BA-UTN
INGENIERÍA
EN SISTEMAS DE
INFORMACIÓN

Algoritmos
y
Estructuras de Datos

Informe de Archivos
FILE *
stream

Lic. Hugo A. Cuello

julio-2016

ÍNDICE

Archivos

Definición y características de un archivo

El archivo es la **única estructura de datos externa**, ya que en vez de residir en la memoria interna RAM reside en una memoria denominada externa, secundaria, auxiliar o también llamada masiva. Por definición un archivo es una colección de datos que obedece a una misma naturaleza. Sus componente generalmente se denominan **registros = struct**, en el sentido más amplio de la palabra, aunque no necesariamente cada componente deba ser de tipo registro = struct. Si por ejemplo, por cada componente debemos guardar un solo dato, el tipo de ese dato puede ser de tipo simple, como ser entero con o sin signo, real, carácter, booleana o incluso cadena y como veremos más delante de tipo puntero. En cambio, si por cada componente guardamos más de un dato, y cada uno de estos datos fueran de distinto tipo entre sí, en estos casos el tipo de cada componente lo estructuramos de tipo registro = struct. Por otro lado, si por cada componente guardamos más de un dato, y cada uno de estos datos fueran de igual tipo entre sí, en estos casos el tipo de cada componente lo estructuramos de tipo arreglo, que es otra estructura de datos interna que se verá más adelante. Para entender un poco más esta cuestión tomemos un ejemplo para cada caso: si un archivo tiene 10 componentes y por cada componente guardo un valor, en el archivo habrá 10 datos; por otro lado, si un archivo tiene 10 componentes y por cada componente guardo tres valores, en el archivo habrá 10 * 3 = 30 valores.

Ventajas y desventajas con los archivos

Una **primer ventaja** radica en el hecho de que la memoria interna es volátil, esto es, si se corta el suministro de la corriente eléctrica, los datos almacenados allí se pierden, y se deberá volver a ingresar nuevamente en forma manual para su recuperación, tarea inapropiada si contamos con una cantidad importante de datos, por lo cual sería lento y tedioso volver a ingresarlos. En cambio, la memoria auxiliar es independiente del suministro eléctrico ya que este tipo de **memoria es no volátil**, los datos se pueden resguardar y a futuro rápidamente recuperados ya que es un proceso puramente electrónico para llevar a cabo esta tarea de recuperación de los datos.

Una **segunda ventaja** de los archivos, consiste en que la memoria interna es de un espacio reducido, por lo que no podremos dar cabida en forma simultánea a una gran colección de datos, por el contrario, la **memoria masiva presenta una capacidad mucho mayor**, casi podríamos decir inconmensurable, ya que podríamos incorporar memoria adicional según se lo requiera, por lo que contamos y de hecho es posible, guardar una cantidad enorme de datos sin poner el riesgo de quedarnos sin espacio de almacenamiento.

No obstante, debemos aclarar que **existe una desventaja** con los archivos siendo el **tiempo de acceso a los datos**, ya que la unidad que consideramos para acceder a una posición en la memoria RAM es el nanosegundo, esto es, 10^{-9} segundos, o sea, 1.000.000.000 (léase mil millones) de fracciones de segundo, en cambio, para acceder a una posición de memoria externa requiere del orden de 10^{-3} segundos, o sea, la unidad

de medida es el milisegundo. Por lo tanto, observamos que existe una gran diferencia de tiempo entre acceder a una posición en la memoria interna con respecto a una posición de memoria externa.

La necesidad de poder **recuperar** los datos en momentos posteriores a su creación, ya sea, por haberse cortado el suministro de la corriente eléctrica o en distintas corridas del programa o en distintos programas, por un lado, y por otro ante la **imposibilidad de poder contar con todos los datos simultáneamente en la memoria interna** debido a su limitación de espacio, son dos motivos que hacen de la necesidad de contar con este tipo de estructuras de datos.

El archivo es la única estructura de datos externa, es decir, ubicadas en un dispositivo externo, al cuál se los denomina memoria auxiliar, o memoria secundaria o memoria externa.

La desventaja principal de este tipo de estructura es el tiempo necesario para recuperar un dato, ya que estos tiempos se miden en milisegundos, esto es 10^{-3} *segundos* en comparación con el tiempo empleado para acceder a una posición en la memoria interna cuya unidad de medida es el nanosegundo, esto es 10^{-9} *segundos*; **por lo tanto, al momento de requerir un dato desde un archivo, debemos tomar muy en cuenta esta última situación, tratando de minimizar estos tiempos.**

Las componentes de un archivo se denominan registros, y en la mayoría de las situaciones, estas componentes serán de tipo registro. No obstante, en ciertas situaciones podrán ser de un tipo simple de datos como *integer, word, char, boolean, longint o punteros,* como así también de otro tipo estructurado de datos que se verán más adelante.

Un archivo es por lo tanto, una **colección de registros** que responden a una misma naturaleza, p.e. Artículos, Clientes, Proveedores, Empleados, Cuentas Contables, etc..

Clasificación de archivos de acuerdo a su función de uso

Datos

Maestros

Son archivos **permanentes** en el tiempo, es decir, no se eliminan luego de un proceso. Contienen todos los datos necesarios para el desarrollo de las actividades de una organización. Representan al **mundo real**. Con el transcurrir del tiempo deben ser actualizados. Dependiendo del momento en que se actualizó, da un grado de confiabilidad. Ejemplo de archivos maestros pueden ser, los Clientes, Proveedores, Empleado, Artículos, Cuentas Contables, etc.

Novedades o Transacciones

Son archivos transitorios, es decir, luego de ser procesados, no tiene sentido mantenerlos, por lo tanto son eliminados. Su cometido es generalmente la actualización de los archivos maestros. La eliminación se podrá realizar inmediatamente o bien luego de un período de tiempo, por ejemplo, después de una segunda actualización al maestro. No siempre existen estos archivos, esto depende del tipo de proceso que se lleve a cabo. Por ejemplo si el proceso es *interactivo* en tiempo real, esto es, en el momento en que se conoce la novedad se actualiza en el maestro, no existirá un archivo de novedades. También se podrán generar registros por cada novedad que se presente en un proceso en tiempo real interactivo para control. En cambio, si el proceso es en *batch* o por lotes,

primero se capturan las novedades durante el transcurso de un tiempo, -un día, una semana, un mes- se los ordena bajo un cierto criterio.

Históricos

Son archivos cuyo uso generalmente son para fines estadísticos, por ejemplo, las ventas realizadas por mes de un año, el seguimiento de ciertos artículos más solicitados, procesos de períodos anteriores, etc..

Tablas

Son archivos de poco volumen, ya sea en cantidad de registros o con respecto a su longitud del mismo. A efectos de ganar velocidad durante el proceso, estos tipos de archivos pueden ser volcados a la memoria interna –RAM- para acelerar el proceso, debido a que acceder a un registro en un archivo la unidad de medida es el *milisegundo*, esto es, 10^{-3} *seg.*, en cambio acceder a una ubicación en la memoria interna, la unidad de medida es el *nanosegundo*, es decir, 10^{-9} *seg.*, razón por la cual se ve la enorme diferencia existente entre acceder a una u otra fuente. El volcado del archivo se realiza en una pasada secuencial, luego al requerir acceder a un dato se accede en la propia memoria interna. Según los procesos tal vez no sea necesario bajar todos los registros como así también bajar todos los campos del mismo. Por ejemplo un archivo que contenga los códigos de las provincias y un porcentaje que fijan las mismas por las ventas realizadas a esas provincias, un proceso que requiera esos datos, podrían ser volcados a la memoria principal, recorriendo secuencialmente de inicio a fin sobre este archivo, luego cada vez que se requiera averiguar el porcentaje de una provincia se accede a la posición de memoria interna. Más adelante se verá la manera de lograr este cometido con una estructura de datos estática que se estudiará posteriormente.

Índices

Son archivos que contienen 2 ó 3 campos generalmente, un campo denominado *clave* y un campo denominado *referencia* o *dirección* el tercer campo si existe es denominado *estado*. Estos archivos se encuentran ordenados por el campo clave. Su objetivo es indexar al archivo maestro u otros archivos de datos. Esto permite una **búsqueda** más eficiente y establece un orden **lógico** de esos datos. Podrán existir varios archivos de índices para un mismo archivo de datos y c/u. de ellos estará ordenado por el campo clave que corresponda. Para mayor información ver organización indexada.

Auxiliares

Son archivos que crea y elimina el programador y que son necesarios para mejorar la eficiencia de un proceso. Por ejemplo, un archivo de vendedores desordenado y un proceso que requiera acceder a los registros en forma reiterada o en distinto orden al que se grabaron los mismos. Está claro que al buscar un vendedor debemos recorrer el archivo secuencialmente y esto por cada vendedor que requiera el proceso, nada eficiente será el proceso, por lo cual debemos recurrir a alguna técnica que permita optimizar el proceso. Una de las técnicas que podrían emplearse es crear un archivo auxiliar de tal manera que ordene esos registros de alguna manera y permita un mayor acceso a los mismos. Por ejemplo si los vendedores están identificados con un Código de Vendedor de 1 a 999, se crean anticipadamente esos registros, luego al ir leyendo

cada vendedor en el archivo original se lo ubica en la posición física en el archivo auxiliar en la posición indicada por el Código del vendedor leído. Posteriormente se realiza el proceso principal pero esta vez, al buscar un vendedor se accede en el auxiliar en la posición indicada por el código del vendedor, al leer el registro su valor indica la posición en el archivo original del vendedor para acceder a esa ubicación y leer los datos del mismo.

Informes o Reportes

Son archivos cuyo destino original era la impresora, pero debido a que ésta ya estaba ocupada por otro proceso, el sistema operativo lo redireccionó hacia otro dispositivo, es decir, un archivo en disco para que posteriormente cuando la impresora sea liberada y la prioridad le sea asignada ese archivo sea volcado a la impresora, una vez que la tarea se llevó a cabo, el mismo sistema operativo elimina ese archivo. La parte del sistema operativo que realiza este cometido es el **S.P.O.O.L.** (Simultaneous Peripherical Operation On Line), es decir, Operaciones Periféricas simultáneas En Linea, el cual redirecciona las salidas de los procesos a archivos en disco debido a que el destino original, la impresora, estaba ocupada y por ser un recurso no compartido debió ser enviado a otro destino, armando una cola de espera.

Seguridad

Son archivos que se realizaron copias de otros archivos y en caso de pérdida de uno de ellos poder recuperarlos con el otro. Esto se conoce como *back-up*.

Programas

Fuentes

Son los archivos escritos en un lenguaje de computadora y de tipo texto. La extensión de estos archivos se corresponde con el lenguaje utilizado, por ejemplo, *.Pas, .C, .Cob, .Bas, .Prg,* etc. Estos archivos son creados utilizando un editor de texto, por ejemplo el Word pero tipo texto, el bloc de notas, no son los más apropiados, otra forma es utilizar el propio editor de texto incorporado en el paquete de software del lenguaje. Así por ejemplo el **Turbo Pascal de Borland** viene un entorno de trabajo denominado **I.D.E.** –Medio ambiente de Desarrollo Integrado- en el cual no solo podremos editar nuestro código fuente, sino además compilar, ejecutar, depurar, entre otros aspectos. Además nos facilita la escritura ya que las palabras reservadas se escriben resaltadas al resto de las otras palabras.

Objetos

Son archivos resultado del proceso de compilar el código fuente. El compilador es una aplicación, es decir, un programa ejecutable que toma como parámetro el código fuente y produce como salida un archivo o programa objeto, cuya extensión es *.OBJ*, el cual aún no puede ser ejecutado debido a que le faltan las librerías a que hace referencia. Cada lenguaje posee su propio compilador.

Ejecutables

Son archivos resultado del **enlace** con las librerías para que sean incorporadas al código máquina y pueda correr en forma autosuficiente. El *link* es una aplicación que toma como parámetro el código objeto y el resultado final es un archivo ejecutable *.EXE*.

Otros

Documentos

Son archivos creados con un procesador de palabra, por ejemplo el Word.

Imágenes gráficas

Presentan diferentes formatos .GIF, .JPG, .BMP, etc. Son creados por software graficadores como el Paint, Corel, Harvard, PhotoShop, etc.

Audio

Archivos de sonido o música con extensiones como ser .MP3, .WAV, .MID, etc.

Miscelánea

Librerías dinámicas .DLL, Sistema .SYS, Dispositivos .DRV, etc..

Clasificación de lenguajes

Los lenguajes de computadoras pueden ser clasificados en cuanto al mayor acercamiento hacia la máquina o hacia el usuario en:

Bajo nivel

Son lenguajes que se acercan más a la máquina que al usuario. Cada instrucción se traduce en una única instrucción de máquina, se dice entonces que la relación es 1 a 1; el lenguaje de bajo nivel es el assembler y la aplicación que lo convierte a código máquina se denomina ensamblador. La característica más emblemática es que los programas ejecutables son los más veloces. La desventaja es que es más compleja su programación. La extensión de los archivos de código fuente presentan la extensión *.ASM*. Por ejemplo sumar un valor a una variable, se escribiría:

Medio nivel

Son lenguajes que se encuentran en un punto intermedio entre los de bajo y alto nivel. Lenguaje como C o Forth entran dentro de esta categoría. La forma en como se escribe el código en algunos casos puede llevarlo a un nivel más bajo o más alto, por ejemplo acumular un valor en una variable podría escribirse en lenguaje C de varias maneras diferentes, pero una de ellas generará un código de máquina más eficiente que las otras. El siguiente ejemplo muestra esto último: sum = sum + 1; sum+= 1 o ++sum o sum++; en los dos últimos casos generará un código más eficiente.

Alto nivel

Los lenguajes de alto nivel se acercan más al usuario que a la máquina y los programas escritos en código fuente se asemejan al lenguaje natural. Estos programas corren más lentos que los de bajo nivel. Una sentencia suele ser convertida a varias instrucciones en código máquina. Lenguajes como Pascal, C, Cobol, Basic, Fortran, Modula, Ada, Prolog entre otros son de alto nivel. Por ejemplo, acumular un valor en una variable sería:

sum := sum + 1	Pascal
inc(sum)	Pascal
add 1 to sum	Cobol
sum = sum + 1	C
sum += 1	C
sum++	C
++sum	C

Otras clasificaciones de los lenguajes podrían realizarse en cuanto al objetivo en que fueron concebidos, así existen lenguajes de propósito general, como ser el BASIC, PASCAL; otros destinados a la gestión y administración contable como el COBOL, RPG aún otros con fines científicos como el FORTRAN y otros para el desarrollo de soft de base como el C.

Traductores

Los traductores pueden ser de dos tipos diferentes:

1. **Intérpretes**
2. **Compiladores**

Intérprete

En el primer caso los lenguajes **intérpretes**, la ejecución se realiza dentro de un entorno de trabajo del lenguaje, y se ejecuta desde alli, o por medio de una aplicación, es decir un módulo de tiempo de ejecución que toma como parámetro el código fuente, cada sentencia a ejecutar primero debe ser interpretada a su equivalente en código máquina, generalmente una sentencia se divide en varias instrucciones de máquina, luego se ejecuta, esto se repite por cada sentencia que deba ser ejecutada, aún en los casos en que una misma sentencia se ejecute más de una vez deben de realizarse esos pasos; no se genera ningún código objeto en disco. Este tipo de lenguajes es oportuno cuando se está desarrollando la aplicación, en la cual tendremos que ejecutar el programa varias veces para refinarlo en detalles, hasta que quede el definitivo, entonces debido a que no hay tiempo de espera para la compilación total del programa; se hace conveniente en esos momentos; pero no cuando el programa haya quedado terminado de corregir detalles. **El tiempo de ejecución es mayor en un programa interpretado que si fuera compilado**. La ventaja es que entre cada ejecución del programa en la etapa de depuración no debemos esperar por el proceso de compilación, en la que muchas veces se demora bastante tiempo.

Compilador

Por otro lado un lenguaje **compilado** primero se compila todo el código fuente, creándose un código máquina y guardado en un archivo con extensión .OBJ, luego en un segundo proceso se le incorporan las librerías produciendo un código ejecutable y guardado en un archivo con extensión .EXE. En este momento podremos correr o ejecutar la aplicación o programa. El tiempo insumido será mucho menor a un programa interpretado, debido a que el código fuente fue traducido a código máquina con anterioridad y solamente el proceso se centra en ejecutarlo.

Existen lenguajes que son solamente interpretados y otros que son solamente compilados, pero también existen lenguajes que pueden correr con un intérprete y que además puedan ser compilados, una vez que se hayan depurados ciertos errores. Por ejemplo un programa realizado en lenguaje Basic -ciertas versiones- puede ser solo interpretado o si el programador lo desea compilado. Otros como la mayoría de los lenguajes sólo compilado, por ejemplo, COBOL, C, PASCAL, ALGOL, CLIPPER, etc.

Etapas de procesos de los archivos en el tiempo

A continuación se presenta otra clasificación de archivos de acuerdo a distintos procesos que podemos realizar:

- **Creación**
- **Actualización:**
  - Altas
  - Bajas
  - Modificaciones
- **Recuperación:**
  - Consultas
  - Informes
- **Mantenimiento:**
  - Estructuración
  - Organización

Modo de apertura de archivos

Entrada –Input-

Un archivo abierto como solo de entrada indica que solo se podrá **leer**, hacer un intento de escritura ocasionará en un error. En el diagrama de flujo el símbolo es el bloque del trapecio invertido, base menor hacia abajo.

Salida –Output-

Un archivo abierto como solo de salida indica que solo se podrá **grabar**, hacer un intento de lectura ocasionará en un error. En el diagrama de flujo el símbolo es el bloque del trapecio, base mayor hacia abajo.

Entrada / Salida -Input/Output-

Un archivo abierto en el modo de lectura-escritura indica que se podrán realizar ambas operaciones, es decir, leer y/o grabar indistintamente.

Modo de acceso a los registros

Secuencial

Se recorren los registros uno a continuación del otro, es decir, en forma adyacente o contigua. Para alcanzar el registro que ocupa la ubicación ***n*** debemos recorrer todos los registros que lo preceden. El tiempo empleado para accesar un registro n depende del lugar en que se encuentre ubicado el puntero al archivo.

Al azar

El tiempo empleado para accesar un registro **no** depende del lugar en que se encuentre ubicado el puntero al archivo, debido a que se accede al registro ***n*** directamente, vale decir que, el tiempo empleado para acceder a cualquier posición es el mismo desde el lugar en que se encuentre el puntero al archivo.

> Por ejemplo, el control remoto de una TV vía satélite o por cable permite 2 tipos de accesos, uno secuencial, y otro al azar; en el primer caso, si se oprimen las teclas CH+ o CH- permite un acceso secuencial a los canales; por otro lado, si tipeamos un número, -con las teclas numéricas-, cambiamos directamente a ese canal. ¡Imaginarse, por ejemplo, el tiempo empleado de acceder del canal 182 al canal 534 en forma secuencial!.

Organizaciones de archivos

Secuencial

La organización secuencial es aquella en la cual los registros solo pueden ser accedidos en forma secuencial. Además la apertura del archivo solo se realiza exclusivamente para entrada o exclusivamente para salida, en el primer caso solo se lo puede leer y en el segundo caso solo se lo puede grabar. Tanto la lectura o grabación se lleva a cabo hacia delante y no se permite retroceder a una posición previa. Los medios de almacenamiento por naturaleza son las cintas magnéticas pero también pueden ser los discos. En esta organización los registros suelen ser sometidos a un ordenamiento establecido. Además los registros podrán estar agrupados, contenidos en un bloque, esta técnica se conoce como *factor de bloqueo* en donde se determina un valor ***n***, siendo este valor n la cantidad de registros contenidos en un bloque, esta técnica es utilizada en archivos con acceso secuencial para lograr un menor tiempo de procesamiento cada vez que se acceda al dispositivo externo. Un bloque representa un *registro físico*, mientras que los ***n*** registros contenidos en él representan ***n*** *registros lógicos*. Este tipo de organización es oportuna cuando los registros deban ser leídos en el mismo orden en que fueron grabados y deban ser procesados la mayoría de ellos.

Indexada

La organización indexada presenta básicamente 2 archivos, uno de **datos** similar a la organización secuencial, es decir, los registros que se incorporan son agregados al final del archivo y normalmente no estarán ordenados físicamente. El otro archivo es el de **índice** y puede contener 2 ó 3 campos, a saber, un campo que contendrá la *clave* y que debe formar parte en el archivo de datos, pudiendo ser de cualquier tipo pero el tipo

preferido es el de cadenas, debido a que en muchas oportunidades se suelen combinar los valores de dos o más campos por medio de la operación de concatenación. El segundo campo representa la *referencia* o *dirección*, y que indica en donde se encuentra el valor de esa clave en el archivo de datos. Por último en caso de existir el tercer campo establece un *estado* para informar si ese registro debe ser tenido en cuenta en los procesos ya que podría habérsele dado de baja, con un valor podría indicar que estará activo o caso contrario estará inactivo, on u off, verdadero o falso, 0 ó 1. En caso en que el registro no debe tomarse en cuenta es debido a que se ha realizado una baja lógica, es decir, el registro sigue ocupando un lugar físico en el medio externo. Las bajas físicas en estos casos se realiza cuando se vayan acumulando varias bajas lógicas, ya que este proceso requiere de un mayor tiempo de proceso.

A continuación se presentará un ejemplo con valores en el archivo de datos para construir el archivo de índices. El siguiente archivo de datos puede representar datos de Alumnos y a efectos de resumir, solo se mostrará por cada registro el campo NroLeg.

Archivo de datos: Alumnos.Dat

valores del campo Nro.Leg. x c/registro.

43	27	38	75	12	89	62	31	56	19	94	83	42	5	98
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

direcciones físicas de cada registro.

Archivo de índices

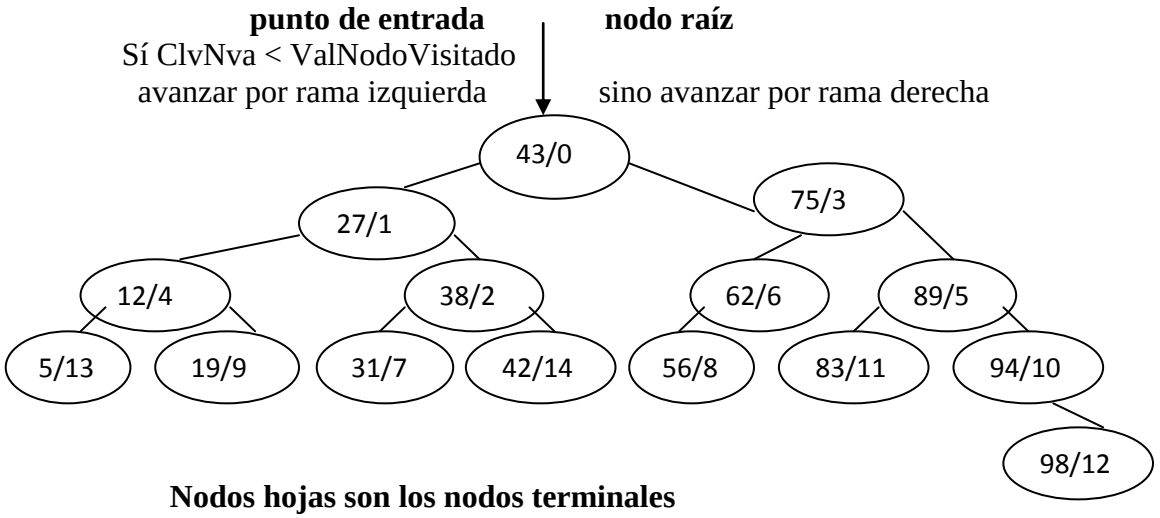
Alumnos.Idx

Para construir la tabla de índices se comenzará leyendo el primer valor del archivo, luego el siguiente y así sucesivamente, pero si vamos escribiendo en la tabla de más abajo notaremos que debemos insertar por cualquier lugar de la lista que vayamos armando y en el caso de escribirlo sobre papel es casi una tarea imposible; por esta razón vamos a implementar otro esquema gráfico que evitará la situación indicada anteriormente.

En el primer caso tendremos el siguiente problema:

CLV	REF.
~~43~~ 27	~~0~~ 1
43	0

Luego de haber copiado el valor 43 y su referencia 0, al leer el siguiente dato vemos que el valor clave 27 es menor al ingresado previamente, si queremos mantener un orden ascendente deberíamos escribir este valor en una línea previa, pero sin antes borrar el valor 43 y su referencia para luego escribirlos más abajo. Notamos luego que al continuar con los próximos valores esta tarea se vería más complicada, razón por la cual adoptaremos otro criterio gráfico.



Nodos hojas son los nodos terminales

Un nuevo nodo se inserta siempre como nodo hoja o terminal.

El gráfico representa un árbol binario, el nodo raíz es el punto de entrada para incorporar un nuevo valor, por lo tanto por cada **valor nuevo** que ingresa se compara si es **menor** al **valor del nodo actual**, si es así se dirige a la **izquierda**, **caso contrario** hacia la **derecha**, en caso de no existir un nodo se crea uno nuevo, siempre como nodo hoja. Si seguimos este criterio, los valores irán ubicándose de la forma en que quedaron arriba. Esta forma depende de cómo se vayan conociendo los valores claves, ya que si esos mismos valores claves estuvieran acomodados de una manera diferente, diferente sería entonces la estructura que adoptaría el árbol.

Ahora vamos a volcar estos valores a nuestra tabla original, para ello debemos saber como se procede a recorrer el árbol, la forma de hacerlo podría ser **in-orden IRD**[1], en **pre-orden RID**[1] o en **post-orden IDR**[1]. Vamos a recorrer in-orden. En un árbol binario tenemos un sub-árbol izquierdo y un sub-árbol derecho, así el nodo con valor 27 es el nodo raíz del sub-árbol izquierdo, y el nodo con valor 75 es el nodo raíz del sub-árbol derecho, esta división en sub-árboles se puede continuar con los restantes nodos.

El recorrido se inicia por el nodo raíz, en este momento averiguamos si hay algún nodo hacia la izquierda, si es así descendemos un nivel, nuevamente hacemos lo mismo hasta llegar a un nivel en que no haya nodos a izquierda, entonces tomamos este valor en nodo raíz del sub-árbol y marcamos al nodo como visitado, en el ejemplo este valor es 5. Luego averiguamos si hay nodos a derecha, si es así descendemos un nivel y volvemos a aplicar el mismo criterio indicado anteriormente. En el ejemplo el siguiente valor a tomar es el 12 que se encuentra un nivel más arriba, debido a que no había nodos a derecha del nodo con valor 5. Luego el recorrido es hacia la derecha del nodo con valor 12, y así seguiremos avanzando y retrocediendo. Esta técnica es conocida como *back-tracking*.

A continuación se expondrá en el archivo de datos indicado anteriormente otro valor para cada registro, por ejemplo, el Apellido de cada alumno, para simplificar la escritura lo indicamos de la siguiente manera:

[1] IRD significa: Izquierda, Raíz, Derecha. RID significa: Raíz, Izquierda, Derecha. IDR significa: Izquierda, Derecha, raíz.

valores del campo Nro.Leg. x c/registro

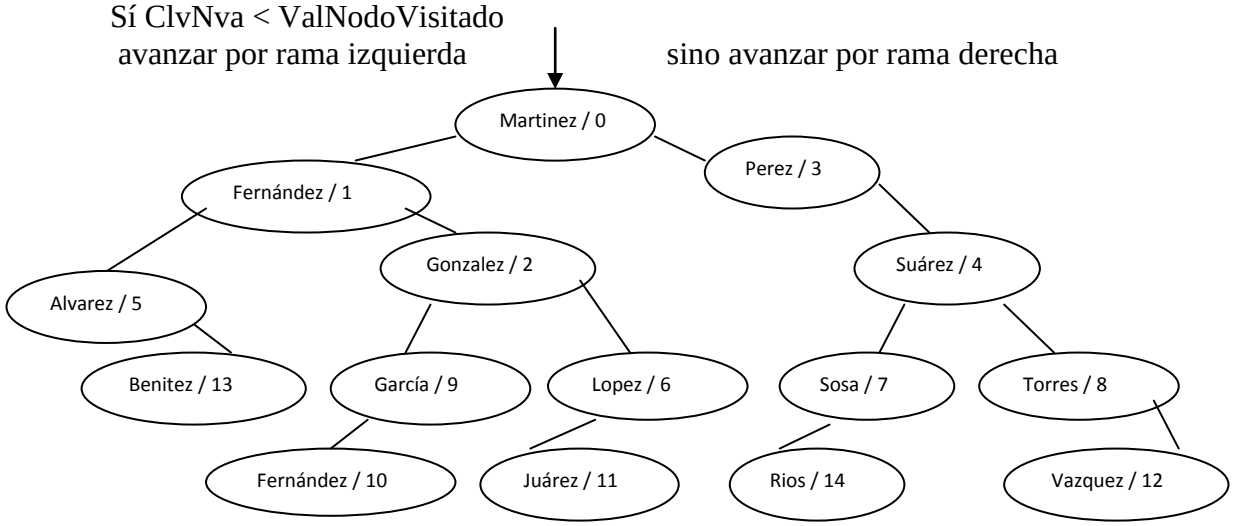
MARTINEZ	FERNÁNDEZ	GONZALEZ	PEREZ	SUAREZ	ALVAREZ	LOPEZ
0	1	2	3	4	5	6

direcciones físicas de cada registro

valores del campo Nro.Leg. x c/registro

SOSA	TORRES	GARCÍA	FERNÁNDEZ	JUÁREZ	VAZQUEZ	BENITEZ	RIOS
7	8	9	10	11	12	13	14

direcciones físicas de cada registro



Nodos hojas son los nodos terminales. Un nuevo nodo se inserta siempre como nodo hoja o terminal.

Conocido el valor de una clave, ¿qué método de búsqueda aplicar?. Si el método de búsqueda fuera **secuencial** comenzaríamos desde el primer valor en la tabla si es el que buscamos detenemos la búsqueda e indicaremos que el valor se encontró, sino puede que sea mayor el valor a buscar en ese caso seguiremos buscando con los próximos valores de la tabla hasta posiblemente encontrarlo o bien detenernos cuando aparezca un valor mayor al que buscamos o bien termine la tabla, en estos casos se informará que el valor no se encontró en la tabla.

Tabla: NroLeg

CLV NroLeg	Ref.
5	13
12	4
19	9
27	1
31	7
38	2
42	14
43	0
56	8
62	6
75	3
83	11
89	5
94	10
98	12

Tabla: ApeNom

CLV ApeNom	Ref.
Álvarez	5
Benítez	13
Fernández	1
Fernández	10
García	9
González	2
Juárez	11
López	6
Martínez	0
Pérez	3
Rios	14
Sosa	7
Suárez	4
Torres	8
Vázquez	12

Si el valor clave está, entonces el siguiente paso será acceder con la referencia indicada en la tabla, al archivo de datos a esa misma posición, para obtener los datos requeridos.

Ahora bien, ¿será este método el secuencial el más adecuado cuando la tabla de índices se encuentra ordenada por la clave a buscar?. La respuesta es **NO**. Un mejor método es realizar una búsqueda **binaria**, aquella que parte en forma sucesiva por la mitad entre los valores *mínimo* **pri** y *máximo* **ult** de las direcciones de los registros extremos, es decir, en donde posiblemente se pueda encontrar el valor de la clave. Encontrado el punto medio entre los extremos *(pri + ult) / 2*, se compara el valor a buscar con el valor de la posición de este punto medio, si se encontró se abandona la búsqueda, informando que se encontró el valor, sino puede suceder que sea mayor o menor, en cualquiera de los casos se acorta la tabla por su mitad, es decir, cambia el valor extremo menor **pri** por el de **med** + 1 o bien cambia el valor extremo mayor **ult** por el de **med** – 1, esto se repite hasta encontrar el valor a buscar, si es éste el caso se accede al archivo de datos en la dirección indicada por el **campo ref**. de la clave encontrada en la tabla de índices, o bien si el valor extremo menor **pri** se hizo mayor al valor extremo mayor **ult**, se abandona la búsqueda e informa que el valor clave no se encontró en la tabla. Se establece que la cantidad máxima de comparaciones a realizar con N componentes, está dado por la siguiente expresión: $log_2 N$, y considerando a **N = 50.000**, la cantidad máxima de comparaciones será de **16**.

En el árbol si buscamos el valor 31, debemos pasar por los siguientes nodos: 43, 27, 38 y 31.

Un árbol **unialargado** es aquel árbol en que crece solamente por una de sus ramas, izquierda o derecha para todo nodo.

Ejercicios

1. ¿Cuál sería la estructura de un árbol si las claves vienen ordenadas en forma ascendente?.
2. Idem anterior pero con las claves ordenadas en forma descendentes.

Las claves en una organización indexada pueden ser *primarias* o *secundarias*. Si la clave es primaria identifica **unívocamente** a un registro y su valor **no puede repetirse** en la tabla. En cambio si la clave es secundaria, puede que se repita o no.

Ejemplos de claves primarias y secundarias:

Claves Primarias: *NroLeg* en el archivo maestro de Empleados, *CodArt* en el archivo maestro de Artículos, etc.

Claves Secundarias: *CodPos* en al archivo maestro de Empleados o Clientes o Proveedores, notamos en este caso que diferentes empleados o clientes o proveedores pueden tener su domicilio bajo un mismo código postal, por lo tanto esta clave secundaria será con duplicación o repetición.

Para un mismo archivo podrán presentarse varias claves candidatas a ser clave primaria, la elección queda más justificada cuanto más compacta sea la misma.

Las claves también pueden ser **simples** o **compuestas**. Una clave simple es la que se forma con el valor de un solo campo; mientras que una clave compuesta se forma con la concatenación de dos o más valores de campos.

Los valores de las claves deben formar parte en el archivo de datos. Si la clave es de tipo cadena es mejor, ya que en los casos de una clave compuesta tal vez se requiera concatenar dos valores de campos diferentes. Si uno de los campos no es de tipo cadena, no hay problema, ya que se lo podrá convertir a cadena.

Relativa

La organización relativa se denomina así porque las posiciones que ocupan los registros dentro del archivo son direcciones **relativas** y no **absolutas**. Debido a que las posiciones de los registros del archivo se comienzan a contar desde el punto de inicio del archivo y no desde el punto de inicio del disco. La organización relativa está considerada como la organización de archivos con mayor velocidad de acceso a los registros generalmente, aunque no siempre será así, dependiendo de los distintos casos que se presenten, como se verá más adelante. Esta organización de archivo junto con la organización secuencial han sido las primeras organizaciones en los inicios de la informática, en este caso precisamente, porque se requería un acceso a los registros que no fuera necesariamente secuencial. En cambio la organización indexada surgió con posterioridad, hoy ampliamente utilizada esta última en las bases de datos. Para poder acceder a un registro n en una organización relativa debemos conocer el valor de una clave debiendo ser de tipo numérico, ya que las direcciones en el archivo son valores numéricas. No existe un archivo de índices como en la organización indexada, solo un archivo de datos. Este tipo de organización requiere la habilidad del programador para manejarlo adecuadamente. Esta habilidad estará emparentada con la experiencia lograda con los años de trabajo en diferentes proyectos que haya realizado en su vida profesional. No obstante, algunos detalles podrán mostrarse.

Tipos de direccionamiento en una organización relativa

Existen dos tipos de direccionamiento en una organización relativa

Direccionamiento directo

Es cuando se accede una sola vez al archivo a un registro direccionado, siendo ese el registro requerido. En ciertos casos esa dirección será igual al valor de la clave, en otros casos, esa dirección se obtiene aplicando al valor de la clave una función denominada en forma genérica función **hasing**, que es una función de mapeo y, cuando aplicando esta función de mapeo o hasing obtiene para cualquier clave conocida direcciones de memoria válidas, se dice que es una función hashing perfecta, el cual no provoca *colisiones* o *sinónimos*.

Direccionamiento indirecto

Es cuando se accede más de una vez al archivo para localizar o establecer el valor de la clave en una ubicación en el archivo, debido a que ocurrieron *colisiones* o *sinónimos*.

A continuación se darán varios casos de estudio

Caso 1: En una entrevista con un cliente, nos informa llevar el proceso de un archivo de artículos, máximo 100 y la forma de identificar a cada artículo, se combino en numerarlos de 1 a 100. Notamos aquí una **relación 1 a 1** entre el código del artículo y la dirección que le debe corresponder en el archivo. Así conocida la clave de un artículo, digamos 23 le corresponde la dirección 23 en el archivo. Notamos entonces que el acceso a dicho registro tanto para leer como para grabar lo logramos con un solo acceso. Cuando se presenta esta situación el **direccionamiento** se denomina **directo**.

Caso 2: La situación con el cliente se presenta semejante al caso anterior pero los códigos de artículos deben ser entre 1001 y 1100. En este caso no podemos decir que conocida una clave con ese valor nos ubicamos en la dirección correspondiente en el archivo, ya que no existirán direcciones en el archivo con ninguno de esos valores claves. Pero no todo está perdido aún. Notamos enseguida que el intervalo de valores es igual a la cantidad de posibles artículos que podemos contar como máximo y que solo están desplazados 1000 posiciones, por lo tanto, si restamos ese valor a cualquier clave que conozcamos solucionamos el problema y obtendremos una dirección válida en el archivo, es decir, una dirección entre 1 y 100. Por ejemplo, si conocemos una clave cuyo valor fuera 1023, le restamos 1000, nos queda la dirección 23, siendo válida esta dirección para acceder en el archivo. Por lo tanto aplicando un cálculo a la clave hemos obtenido una dirección válida en el archivo. La relación 1 a 1 sigue existiendo para este caso, solo debemos realizar un cálculo para obtener una dirección válida en el archivo. El tipo de direccionamiento también es directo, ya que solo se requiere de un acceso para localizar un artículo.

Caso 3: La situación con el cliente se presenta semejante a los casos anteriores salvo que los códigos de artículos deben estar comprendidos entre 3427 y 7965. En este caso vemos que al igual del caso 2 con el valor de una clave no podemos ubicarnos en una dirección válida en el archivo, pero además notamos que el intervalo de valores de las claves es mucho mayor a la cantidad de artículos máximos que

podemos tener. Por lo tanto esto trae aparejado un nuevo problema, el de las **colisiones** o **sinónimos**. Esto se produce cuando dos o más valores claves distintas generan una misma dirección en el archivo. Ahora bien, ¿cómo lograr obtener a partir de un valor clave una dirección válida en el archivo?. La técnica empleada se conoce como función de *mapeo*. Una función de mapeo ampliamente utilizada es la del **método del resto**, que consiste en tomar el valor clave k por un lado y el tamaño del archivo TA por otro, o bien por un número primo más cercano al tamaño del archivo; se realiza la división en donde k es el dividendo y TA o el número primo el divisor; el cociente entero es la *dirección natural* $\boldsymbol{d_n}$ para acceder al archivo de datos. Si las direcciones en el archivo comenzaran desde uno en adelante, en estos casos se le suma 1 al resto.

Notamos que valores de claves distintas digamos $\boldsymbol{k_i \neq k_j}$ pueden ocasionar una misma dirección natural d_n, produciéndose entonces una colisión o sinónimo. Este problema es solucionable existiendo distintos métodos para resolverlo. Uno de ellos sería buscar en los registros adyacentes un lugar libre y una vez localizado ubicar los datos allí, en caso de estar dando un alta. Si llegamos al final del archivo, debemos siguir buscando desde el inicio del mismo. Si lo que deseamos es buscar el valor de una clave primero accedemos a su dirección natural si es el dato que hay allí lo encontramos y listo, sino seguiremos buscando con el método empleado para solucionar colisiones, por ejemplo buscar en los registros adyacentes, así seguiremos hasta encontrarlo o bien detenernos en un punto en donde se asegura que ese valor clave no se encuentra en el archivo. Una solución a esto último sería agregar un campo adicional al registro que indique un estado del registro con un valor de V indicando que el registro está vacante, nunca fue utilizado por alguna clave, O indicando que el registro está ocupado, S indicando que el registro fue suprimido, es decir, se realizó una baja lógica. Al dar de alta si su dirección natural estuviera ocupada, la clave nueva se ubicará en el primer registro que esté como vacante o suprimido y se marca el estado con O de ocupado, siempre y cuando esa clave no estuviera en el archivo, sino sería alta existente. Al dar de baja una clave, si no es la de su dirección natural se seguirá buscando hasta que aparezca en el archivo, en ese caso se marca el estado con S de suprimido, siendo una baja lógica, o bien seguir buscando hasta que se encuentre un registro como vacante o se haya completado la vuelta, en estos casos sería una baja inexistente.

En este último caso el direcionamiento se denomina **indirecto**, debido a que encontrar una clave podrá necesitar realizar más de un acceso.

Caso 4: La situación con el cliente se presenta semejante a los casos anteriores salvo que los códigos de artículos ahora son valores de tipo cadena de 5. En este caso vemos que con el valor de una clave no podemos ubicarnos en una dirección válida en el archivo, ya que la clave no es numérica, y como fuera dicho anteriormente, es de esperar que la clave sea numérica, ya que las direcciones en el archivo son posiciones numéricas, por lo que en estas situaciones, debemos convertirla en numérica; una forma sería tomar el cad [i] tomar su código ASCII y multiplicarlo por un múmero primo diferente, para cada cad [i], y acumular los resultados parciales. De esta manera ahora contamos con un número bastante grande, por lo que es de esperar, que este valor numérico no se corresponda con ninguna dirección válida en el archivo; pero además notamos que el intervalo de valores de las claves es mucho mayor a la cantidad de artículos máximos que podemos tener. Por lo tanto esto trae aparejado el mismo problema, que en el caso 3, el de las **colisiones** o **sinónimos**, por lo que aplicamos la misma técnica vista en el caso 3, para solucionar el problema de las colisiones o sinónimos, por lo que a partir de este instante, se soluciona como fuera indicado en el caso 3.

También en este caso el direccionamiento se denomina **indirecto**, debido a que encontrar una clave podrá necesitar realizar más de un acceso.

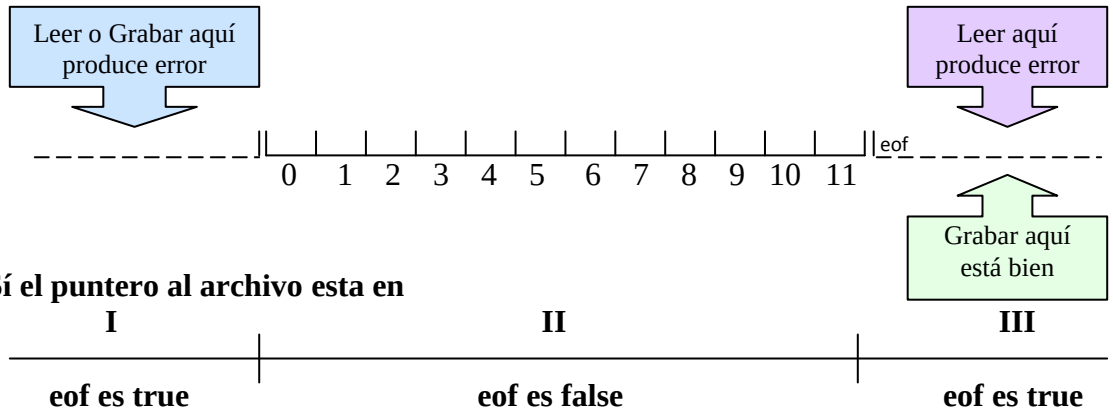
Para utilizar archivos con organización relativa se requiere amplia experiencia, debido a que es la organización de archivos más compleja de todas porque requiere una exigencia por parte del programador en el diseño del archivo, las funciones de mapeo y la resolución de las colisiones.

Estado del puntero al archivo en situación de leer o grabar

Si el puntero al archivo está ubicado en la marca de **eof()**, realizar una operación de lectura producirá un error en tiempo de ejecución y se aborta el programa, lo mismo ocurrirá si está más allá de la marca de **eof**(), pero grabar esta bién y se expande el tamaño del archivo una componente más y la marca se reubica después del registro grabado, coincidiendo con el puntero al archivo.

Si el puntero al archivo está ubicado en una dirección más alejada digamos 15 y se graba un nuevo registro, también se graban los registros 12 a 14 pero su contenido será desconocido.

Archivo binario con 12 registros con direcciones [0; 11]



Si el puntero al archivo está ubicado en una dirección menor a 0 realizar una operación de lectura o escritura producirá un error en tiempo de ejecución y se aborta el programa.

La función eof(f) retornará verdadero si el puntero al archivo se encuentra fuera de los límites del archivo, esto es, si está más allá de la marca eof o si se movió el puntero al archivo a una dirección menor a 0.

Procesos clásicos con archivos

Corte de Control

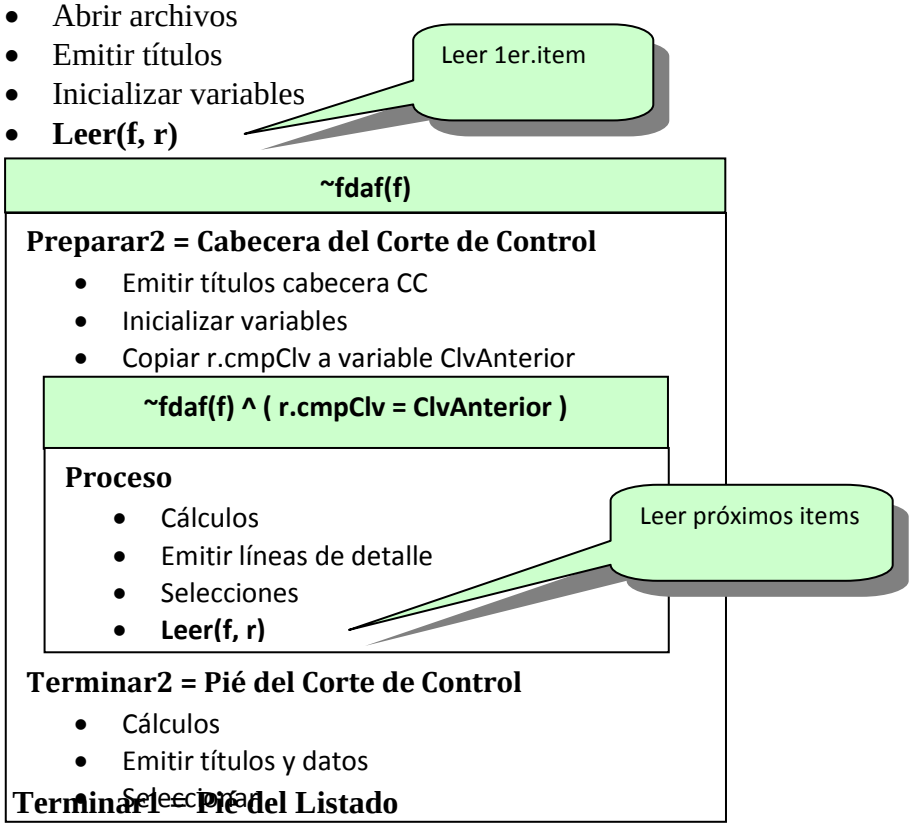
El corte de control es un proceso en el cual los registros del archivo se encuentran ordenados por el valor de uno o más campos, denominados campos **clave** o **llave**, el ordenamiento puede ser ascendente –lo más común- o descendente. Este tipo de proceso generalmente es utilizado para realizar informes o reportes, en el cual se deba emitir de cada grupo –formado por el mismo valor del campo clave- totales, promedios, máximos o mínimos, etc..

En líneas generales este proceso posee una estructura que es bastante característica y que pasaremos a detallar a continuación.

Cada corte de control establece un **nivel**, así si existen ***n*** cortes de control, habrá n niveles, y cada uno de estos n cortes o niveles están contenidos dentro de un ciclo indefinido, a estos ciclos se le suma uno más, y representa el fin del proceso, siendo este ciclo el de mayor nivel y el más externo, luego en forma anidada se van desarrollando los ciclos internos en un orden que va de mayor nivel de corte hasta llegar al menor nivel de corte. El ciclo de mayor nivel establece el fin del proceso, siendo su condición, el fin del archivo ***feof(f)*** que se está procesando. Las condiciones de los ciclos internos van **arrastrando** las condiciones de los ciclos anteriores o más externos a la que se le suma la propia condición del ciclo, es decir, la que producirá el corte de control de ese nivel, por lo tanto, cuanto más anidado sea el ciclo, más condiciones contendrá. El ciclo externo contiene una condición, y el ciclo más interno posee n + 1 condiciones y en cada uno estos ciclos las condiciones establecidad estarán relacionadas con el operador lógico "y", es decir, el operador && del lenguaje C/C++ o representado por el carácter acento circunflejo "^". Si un proceso requiere n cortes, entonces la cantidad de ciclos será de n + 1. Además **cada corte de control obliga a que los datos se encuentren ordenados por el valor del campo clave comenzando desde el de mayor nivel hasta el de menor nivel**, por ejemplo, si se requieren obtener totales de las ventas realizadas por diferentes vendedores, los datos deberían están organizados por el código de vendedor; un segundo ejemplo, si los vendedores realizan sus actuaciones en una zona y el proceso requiere informar además totales por zonas, en este caso los datos deben encontrarse ordenados primero por código de zona y luego por código de vendedor, debido a que es de esperar que primero cambien los vendedores y luego cambien las zonas. **El corte de control requiere de una lectura anticipada**. Las próximas lecturas del archivo se realizarán en el ciclo más interno y como última acción. Antes de ingresar a un ciclo denominamos a esa región del algoritmo **cabecera**. Al salir de un ciclo, denominamos a esa región del algoritmo **pié**. Dentro de un ciclo, denominamos a esa región del algoritmo **proceso**. En la cabecera generalmente realizamos las siguientes acciones: Inicializar, Emitir títulos y datos. En el pié generalmente realizamos las siguientes acciones: Cálculos, Emitir totales, promedios, máximos o mínimos, tomar alguna decisión. En el proceso generalmente realizamos las siguientes acciones: Cálculos, Emitir líneas de detalle, tomar alguna decisión. A continuación se presenta un modelo de algoritmo de Corte de Control en un formato general.

Preparar1 = Cabecera del Listado

- Abrir archivos
- Emitir títulos
- Inicializar variables
- **Leer(f, r)**



Terminar1 = Pie del Listado

- Cálculos
- Emitir títulos y datos
- Cerrar archivos

Ejercicio de Corte de Control de Universidades y Facultados

```
/*
        Id.Programa: G2Ej09.cpp Corte de Control n = 2.
        Autor.....: Lic. Hugo Cuello
        Fecha.....: Mayo-2003
        Comentario.: Examen a alumnos de distintas Universidades y Facultades.
                                                Tecnica de Corte de Control.
*/

#include<iomanip>
#include<iostream>
using namespace std;

typedef unsigned int  word;
typedef unsigned short byte;
typedef char str5[6];
typedef char str15[16];
typedef char str20[21];
struct sExa {
        str5  CodUni,
                        CodFacu;
        long  NroLeg;
```

```
        str20 ApeNom;
        byte  Nota;
};

// Prototipos ----------------------------------------------------
void Abrir(FILE **Examen);
void Inic(word &totInsGral, word &totAprGral);
void EmiteCabLst();
void IniCab(word &totIns, word &totApr, str5 Clave, str5 &ClaveAnt);
void EmiteCabCte(str15 titulo,str5 codigo);
void ProcAlum(sExa rExa, word &tInsFacu, word &tAprFacu);
void CalcPie(word totIns, word totApr, word &totInsM, word &totAprM);
void EmitePie(str15 titulo, word totInsFacu, word totAprFacu);
// Fin Prototipos -----------------------------------------------

void main() {
        FILE *Examen = NULL;
        sExa rExamen;
        word totInsGral,
             totAprGral,
             totInsUni,
             totAprUni,
             totInsFacu,
             totAprFacu;
        str5  UniAnt,
             FacuAnt;

        Abrir(&Examen);
        freopen("Examen.Lst","w",stdout);
        Inic(totInsGral,totAprGral);
        EmiteCabLst();
        fread(&rExamen,sizeof(rExamen),1,Examen);
        while (!feof(Examen)) {
           IniCab(totInsUni,totAprUni,rExamen.CodUni,UniAnt);
           EmiteCabCte("*Cod.Univ.: ",rExamen.CodUni);
           while (!feof(Examen) && strcmp(rExamen.CodUni,UniAnt) == 0) {
              IniCab(totInsFacu,totAprFacu,rExamen.CodFacu,FacuAnt);
              EmiteCabCte("**Cod.Fac.: ",rExamen.CodFacu);
              while (!feof(Examen) && strcmp(rExamen.CodUni,UniAnt) == 0 &&
                          strcmp(rExamen.CodFacu,FacuAnt) == 0) {
                 ProcAlum(rExamen,totInsFacu,totAprFacu);
                 fread(&rExamen,sizeof(rExamen),1,Examen);
              }
              CalcPie(totInsFacu,totAprFacu,totInsUni,totAprUni);
              EmitePie("Facu.: ",totInsFacu,totAprFacu);
           }
           CalcPie(totInsUni,totAprUni,totInsGral,totAprGral);
           EmitePie("Univ.: ",totInsUni,totAprUni);
        }
        EmitePie("General: ",totInsGral,totAprGral);
```

```
        freopen("CON","w",stdout);
        fclose(Examen);
} //main

void Abrir(FILE **Exa) {

        *Exa = fopen("Examen.Dat","rb");
} //Abrir

void Inic(word &tInsG, word &tAprG) {

        tInsG = tAprG = 0;
} // Inic

void EmiteCabLst() {

        cout << "Listado examen a alumnos" << endl;
} //EmiteCab

void IniCab(word &totIns, word &totApr, str5 Clave, str5 &ClaveAnt) {

        totIns = totApr = 0;
        strcpy(ClaveAnt,Clave);
} //IniCab

void EmiteCabCte(str15 titulo, str5 codigo) {

        cout << endl << titulo << " " << codigo << endl;
        if (strstr(titulo,"Fac") != NULL)
                cout << endl << setw(5) << " " << "Nro.Leg.  Nota" << endl;
} //EmiteCabCte

void ProcAlum(sExa rExa, word &tInsFacu, word &tAprFacu) {

        ++tInsFacu;
        if (rExa.Nota >= 4) {
           ++tAprFacu;
           cout << setw(6) << "  " << setw(6) << rExa.NroLeg << "    " << setw(2) <<
           rExa.Nota << endl;
        }
} //ProcAlum

void CalcPie(word totIns, word totApr, word &totInsM, word &totAprM) {

        totInsM += totIns;
        totAprM += totApr;
} //CalcPie

string replicate(char car, unsigned n) {
  string cad = "";
```

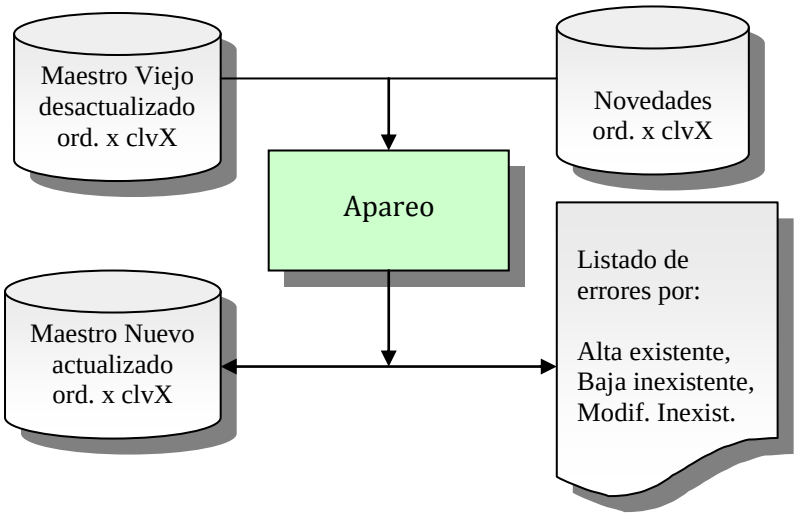
```
  for(unsigned i = 1; i <= n; i++)
      cad += car;
  return cad;
} // replicate

void EmitePie(str15 titulo, word totIns, word totApr) {
  string ast;

          if (titulo[0] == 'F')
             ast = replicate('*',2);
          else
             if (titulo[0] == 'U')
                ast = replicate('*',1);
          cout << " " << ast << "Tot.Insc. " << titulo << " " << totIns << endl;
          cout << " " << ast << "Tot.Apr.. " << titulo << " " << totApr << endl;
} //EmitePie
```

Apareo de Archivos

El apareo de archivos es un proceso que dependiendo del tipo de organización que tengan estos archivos, el algoritmo adoptará una estructura particular. Este proceso es muy empleado para la **actualización del maestro** a través del archivo de novedades. Las novedades tendrán que ver con **altas, bajas o modificaciones** o algunas de ellas solamente. Si ambos archivos poseen una organización secuencial, entonces ambos archivos deben encontrarse ordenados –ascendente o descendente- por medio del valor de una clave en común. El resultado de este proceso será un nuevo archivo de salida con la misma estructura que el maestro a actualizar, siendo este archivo el maestro actualizado. El archivo de novedades tiene la misma estructura que el maestro pero con un campo más, el cual indica el *código de movimiento*. Cada archivo trabajará con sus propios registros, es decir, un registro para el archivo maestro viejo, otro para el archivo de maestro nuevo y un registro para el archivo de novedades. Las situaciones de errores por alta existente o bajas o modificaciones inexistentes, se emitirán por medio del dispositivo de la impresora.

Este proceso que genera un nuevo archivo –el maestro actualizado- se denomina **proceso Padre – Hijo**.

El siguiente diagrama de sistema muestra el proceso Padre – Hijo



A continuación se presentará un modelo de apareo de archivos, con un archivo maestro desactualizado y un archivo de novedades, ambos con organización secuencial, con valores de campos claves sin repetición en cada uno de los archivos y ordenados ambos por el valor de una misma clave.

Al igual que con el Corte de Control, el proceso de **apareo requiere una lectura anticipada y debe ser una lectura especial** en el Lenguaje Pascal, como fue indicado en el tema de Corte de Control.

Un primer proceso se realizará mientras no haya finalizado ningún archivo y en un segundo proceso, se procesará el archivo que no haya finalizado, hasta agotarlo. Por lo tanto, habrá tres ciclos en secuencia.

Dentro del primer ciclo o proceso se compararán los valores de las claves de ambos archivos, por ejemplo, ¿la clave del maestro es igual a la clave de novedades? o ¿la clave del maestro es mayor a la clave de novedades? y por descarte será que ¿la clave de maestro es menor a la clave de novedades?.

En el primer caso si la **clave del registro maestro es igual a la clave del registro de novedades** se deberá comparar el código de movimiento, pudiendo ser igual a 'A' por alta, o a 'B' por baja o a 'M' por modificación.

Si el código de **movimiento es igual a 'A'** será un **error, por alta existente**, es decir, el valor de esa clave se encuentra ya en el archivo maestro y el **registro de maestro desactualizado deberá grabarse en el nuevo maestro** –para no perderlo-.

Si el código de **movimiento es igual a 'B'** no es error y no debe grabarse en el nuevo maestro, -de esta manera lo estamos dando de baja al no aparecer en el nuevo maestro-, por lo tanto, **no debemos realizar ninguna acción** por este caso.

Si el código de **movimiento es igual a 'M'** no es error y debemos **mover los campos a modificar indicados por el archivo de novedades al registro del maestro y luego grabarlo en el nuevo maestro**.

Por último debemos realizar la lectura especial por cada uno de los archivos, es decir, tanto del maestro desactualizado como de novedades.

A continuación se detalla el proceso a realizar si la **clave del registro maestro viejo es mayor a la clave del registro de novedades**.

En este caso resulta que existe un registro de novedades sin su registro correspondiente en el maestro. Por lo tanto se debe determinar el código de movimiento.

Si el código de **movimiento es igual a 'A'** es correcto ya que la clave no existe en el maestro viejo y debe realizarse el alta, para ello se debe a**signar los campos**

informados por la novedad al registro del maestro nuevo y grabar un nuevo registro en maestro nuevo. No asignarlo al registro del maestro viejo, ¿por qué?.

Si el código de **movimiento es igual a 'B'** es **error, por baja inexistente**.

Si el código de **movimiento es igual a 'M'** es **error, por modificación inexistente**.

Por último debemos realizar la lectura especial solo de novedades, -se lee del archivo cuya clave resultó menor, de esta manera aseguramos alcanzar el valor de la otra clave-.

A continuación por decantación se detalla el proceso a realizar si la clave del registro maestro viejo es menor a la clave del registro de novedades.

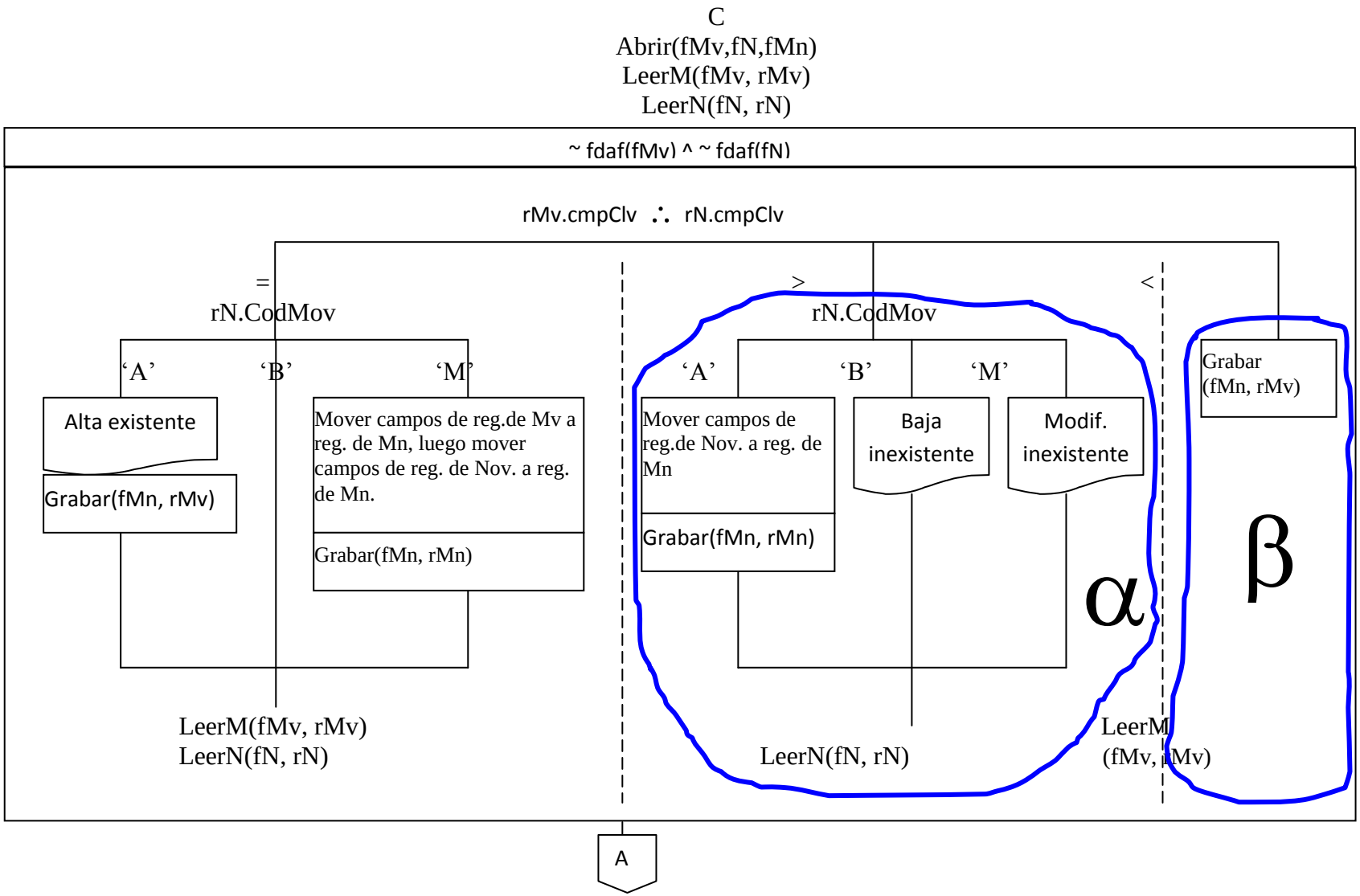
En este caso resulta que existe un registro de maestro sin su registro correspondiente en el de novedades. Por lo tanto, aquí no se deberá analizar el código de movimiento, ya que no existe. El **registro de maestro viejo se debe grabar en el archivo maestro nuevo** y se debe realizar la lectura especial solo de maestro viejo, por haber resultado su clave menor a la de novedades-.

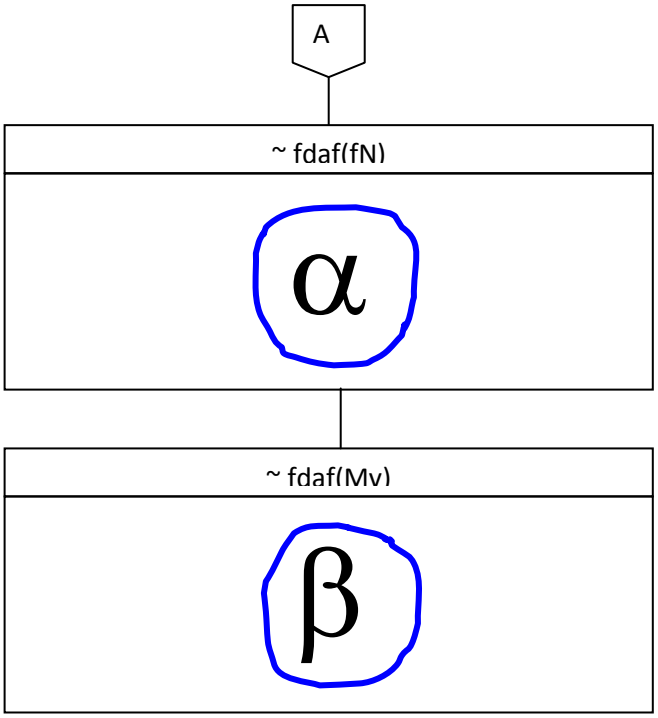
Este primer proceso finaliza cuando uno de los archivos finalice, podría darse el caso que finalicen ambos, si los valores de las claves del último registro de cada archivo son iguales.

Debido a que no se garantiza que finalicen ambos archivos en el proceso anterior, habrá que procesar los registros restantes del archivo que no finalizó.

Si es el caso del maestro viejo que no haya finalizado el proceso finalizará cuando se hayan procesados todos los registros restantes de éste archivo. El proceso consistirá en grabar todos los registros pendientes. Debido a que traemos un registro que hemos leído pero no hemos procesado, dentro del ciclo primero grabamos el registro y luego leemos con una lectura especial el próximo registro.

Por otro lado, si es el caso del archivo de novedades el que no haya finalizado el proceso finalizará cuando se hayan procesados todos los registros restantes de éste otro archivo. El proceso consistirá en realizar las mismas acciones vistas en el caso cuando la clave de maestro viejo es mayor a la clave de novedades, solo serán correctos los registros de novedades cuyos códigos de movimientos indiquen 'A'.A continuación se presenta un modelo de algoritmo de Apareo de archivos en un formato general.


C
Abrir(fMv,fN,fMn)
LeerM(fMv, rMv)
LeerN(fN, rN)
~ fdaf(fMv) ^ ~ fdaf(fN)
rMv.cmpClv ∴ rN.cmpClv
=
rN.CodMov
'A'
'B'
'M'
Alta existente
Grabar(fMn, rMv)
Mover campos de reg.de Mv a reg. de Mn, luego mover campos de reg. de Nov. a reg. de Mn.
Grabar(fMn, rMn)
LeerM(fMv, rMv)
LeerN(fN, rN)
>
rN.CodMov
'A'
'B'
'M'
Mover campos de reg.de Nov. a reg. de Mn
Grabar(fMn, rMn)
Baja inexistente
Modif. inexistente
LeerN(fN, rN)
<
Grabar (fMn, rMv)
α
β
LeerM
(fMv, rMv)
A



Cerrar(fMv, fN, fMn)
F

Técnica de HIGH_VALUE o LOW_VALUE

Este modelo de apareo de archivos puede ser optimizado si se realizan algunos ligeros cambios. En principio vemos que las acciones **alfa** y **beta** se vuelven a reiterar cuando uno de los archivos haya finalizado.

La pregunta es ¿no podrían estas acciones realizarse solamente y en forma completa dentro del primer ciclo?. La respuesta es sí, pero realizando las siguientes modificaciones.

En principio se abandonará el ciclo si **ambos** archivos finalizaron. El secreto radica en asignar el valor más alto al campo clave del archivo que finalizó, si los archivos están ordenados en forma ascendente, caso contrario se asignará el valor más bajo. Esta técnica se conoce como **HIGH VALUE** o **LOW VALUE** en el segundo caso.

La lectura especial debe tener 2 parámetros, el archivo a leer y el registro a devolver.

En caso que haya finalizado el archivo se deberá mover al campo clave el valor más alto, -o el más bajo-, es decir, un valor al cual ningún dato leído podrá alcanzar. Por ejemplo, si el campo clave indicara un código de vendedor de 3 dígitos, el valor más alto sería 1000, ya que ningún dato que se lea alcanzaría a ese valor especial; un segundo ejemplo, podría ser tratándose de fechas con mes y día en un mismo campo clave, el valor más alto sería 1232 **¿por qué?.**

Las condiciones del ciclo, deberán compararse, los valores de los campos claves con el valor más alto, y para abandonar el ciclo las condiciones podrían ser las siguientes:

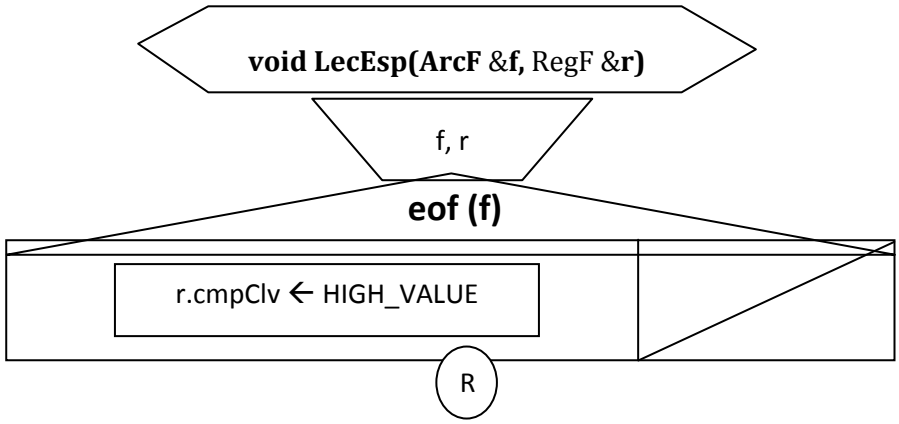
(rMv.cmpClv ≠ HIGH_VALUE) v (rN.cmpClv ≠ HIGH_VALUE)

HIGH_VALUE es una constante con nombre definida en la sección const, p.ej.:

```
const
  HIGH_VALUE = 1000;
```

De esta manera los ciclos posteriores dejarían de existir, ya que no se lo requieren. Con estos ligeros cambios el algoritmo solo quedaría con un solo ciclo.

La lectura especial podría presentar el siguiente aspecto:



Ejemplo de Apareo de archivos

```
/*
   Id.Programa: G2Ej10.cpp Apareo
   Autor.....: Lic. Hugo Cuello
   Fecha.....: ago-2014
   Comentario.: Apareo entre MaeVjo y Nov.
                Genera MaeNvo y LstErr.
*/

#include<iostream>
using namespace std;

typedef char str20[21];

struct sMae {
  long  cmpClv;
  int   cmp1,
        cmp2;
  str20 RazSoc,
        Domic,
        Local;
  long  NroCUIT;
};

struct sNov {
  sMae rMaeN;
  char codMov;
};

// Prototipos ----------------------------------------
```

```
void Abrir(FILE **, FILE **, FILE **);
void Apareo(FILE *, FILE *, FILE *);
void CerrarEliminarRenombrar(FILE *, FILE *, FILE *);
// Fin Prototipos -----------------------------------

void Abrir(FILE **MaeV, FILE **MaeN, FILE **Nov) {

  *MaeV = fopen("MaeVjo.Dat","rb");
  *MaeN = fopen("MaeNvo.Dat","wb");
  *Nov  = fopen("Noved.Dat","rb");
} //Abrir

void Apareo(FILE *MaeV, FILE *MaeN, FILE *Nov) {
  sMae rMaeV,
       rMaeN;
  sNov rNov;

  freopen("ErroresABM.Lst","w",stdout);
  fread(&rMaeV,sizeof(rMaeV),1,MaeV);
  fread(&rNov,sizeof(rNov),1,Nov);
  while (!feof(MaeV) && !feof(Nov))
    if (rMaeV.cmpClv == rNov.rMaeN.cmpClv) {
      switch (rNov.codMov) {
        case 'A': cout << "Error por Alta Existente, clave: " <<
                          rNov.rMaeN.cmpClv << "Ubicacion: " <<
                          (ftell(Nov) - sizeof(rNov)) / sizeof(rNov) << endl;
                  rMaeN = rMaeV;
                  fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
                  break;
        case 'M': rMaeN = rMaeV;
                  rMaeN.cmp1 = rNov.rMaeN.cmp1;
                  strcpy(rMaeN.Domic,rNov.rMaeN.Domic);
                  fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
                  break;
      }
      fread(&rMaeV,sizeof(rMaeV),1,MaeV);
      fread(&rNov,sizeof(rNov),1,Nov);
    }
    else
      if (rMaeV.cmpClv > rNov.rMaeN.cmpClv) {
        switch (rNov.codMov) {
          case 'A': rMaeN = rNov.rMaeN;
                    fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
                    break;
          case 'B': cout << "Error por Baja inexistente, clave: " <<
                            rNov.rMaeN.cmpClv << "Ubicacion: " <<
                            (ftell(Nov) - sizeof(rNov)) / sizeof(rNov) << endl;
                    break;
          default: cout << "Error por Modificacion inexistente, clave: " <<
                           rNov.rMaeN.cmpClv << "Ubicacion: " <<
```

```
                    (ftell(Nov) - sizeof(rNov)) / sizeof(rNov) << endl;
      }
      fread(&rNov,sizeof(rNov),1,Nov);
    }
    else {
      rMaeN = rMaeV;
      fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
      fread(&rMaeV,sizeof(rMaeV),1,MaeV);
    }
  while (!feof(Nov)) {
   switch (rNov.codMov) {
    case 'A': rMaeN = rNov.rMaeN;
            fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
            break;
    case 'B': cout << "Error por Baja inexistente, clave: " <<
                    rNov.rMaeN.cmpClv << "Ubicacion: " <<
                    (ftell(Nov) - sizeof(rNov)) / sizeof(rNov) << endl;
            break;
    default: cout << "Error por Modificacion inexistente, clave: " <<
                    rNov.rMaeN.cmpClv << "Ubicacion: " <<
                    (ftell(Nov) - sizeof(rNov)) / sizeof(rNov) << endl;
   }
   fread(&rNov,sizeof(rNov),1,Nov);
  }
  while (!feof(MaeV)) {
   rMaeN = rMaeV;
   fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
   fread(&rMaeV,sizeof(rMaeV),1,MaeV);
  }
  freopen("CON","w",stdout);
} //Apareo

void CerrarEliminarRenombrar(FILE *MaeV, FILE *MaeN, FILE *Nov) {

  fclose(MaeV);
  fclose(MaeN);
  fclose(Nov);
  remove("MaeVjo.Dat");
  rename("MaeNvo.Dat","MaeVjo.Dat");
} //CerrarEliminarRenombrar

int main() {
  FILE *MaeVjo,
       *MaeNvo,
       *Noved;

  Abrir(&MaeVjo,&MaeNvo,&Noved);
  Apareo(MaeVjo,MaeNvo,Noved);
  CerrarEliminarRenombrar(MaeVjo,MaeNvo,Noved);
```

```
  return 0;
}
```

Apareo de archivos con técnica de High Value

```
/*
   Id.Programa: G2Ej10HV.cpp Apareo
   Autor.....: Lic. Hugo Cuello
   Fecha.....: ago-2014
   Comentario.: Apareo entre MaeVjo y Nov.
                Genera MaeNvo y LstErr.
                Tecnica con HIGH_VALUE.
*/

#include<iostream>
using namespace std;

#define HIGH_VALUE 10000
typedef char str20[21];

struct sMae {
  long  cmpClv;
  int   cmp1,
        cmp2;
  str20 RazSoc,
        Domic,
        Local;
  long  NroCUIT;
};

struct sNov {
  sMae rMaeN;
  char codMov;
};

// Prototipos ----------------------------------------
void Abrir(FILE **, FILE **, FILE **);
void ApareoHV(FILE *, FILE *, FILE *);
void LecHV(FILE *, sMae &);
void LecHV(FILE *, sNov &);
void CerrarEliminarRenombrar(FILE *, FILE *, FILE *);
// Fin Prototipos -------------------------------------

void Abrir(FILE **MaeV, FILE **MaeN, FILE **Nov) {

  *MaeV = fopen("MaeVjo.Dat","rb");
  *MaeN = fopen("MaeNvo.Dat","wb");
  *Nov  = fopen("Noved.Dat","rb");
```

```
} //Abrir

void LecHV(FILE *MaeV, sMae &rMaeV) {

 fread(&rMaeV,sizeof(rMaeV),1,MaeV);
 if (feof(MaeV))
   rMaeV.cmpClv = HIGH_VALUE;
} //LecHV

void LecHV(FILE *Nov, sNov &rNov) {

  fread(&rNov,sizeof(rNov),1,Nov);
  if (feof(Nov))
    rNov.rMaeN.cmpClv = HIGH_VALUE;
} //LecHV

void ApareoHV(FILE *MaeV, FILE *MaeN, FILE *Nov) {
  sMae rMaeV,
       rMaeN;
  sNov rNov;

  freopen("ErroresABMHD.Lst","w",stdout);
  LecHV(MaeV,rMaeV);
  LecHV(Nov,rNov);
  while (rMaeV.cmpClv != HIGH_VALUE && rNov.rMaeN.cmpClv !=
HIGH_VALUE)
    if (rMaeV.cmpClv == rNov.rMaeN.cmpClv) {
      switch (rNov.codMov) {
        case 'A': cout << "Error por Alta Existente, clave: " <<
                          rNov.rMaeN.cmpClv << "Ubicacion: " <<
                          (ftell(Nov) - sizeof(rNov)) / sizeof(rNov) << endl;
                  rMaeN = rMaeV;
                  fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
                  break;
        case 'M': rMaeN = rMaeV;
                  rMaeN.cmp1 = rNov.rMaeN.cmp1;
                  strcpy(rMaeN.Domic,rNov.rMaeN.Domic);
                  fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
                  break;
      }
      LecHV(MaeV,rMaeV);
      LecHV(Nov,rNov);
    }
    else
      if (rMaeV.cmpClv > rNov.rMaeN.cmpClv) {
        switch (rNov.codMov) {
          case 'A': rMaeN = rNov.rMaeN;
                    fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
                    break;
          case 'B': cout << "Error por Baja inexistente, clave: " <<
```

```
                    rNov.rMaeN.cmpClv << "Ubicacion: " <<
                    (ftell(Nov) - sizeof(rNov)) / sizeof(rNov) << endl;
                break;
        default: cout << "Error por Modificacion inexistente, clave: " <<
                    rNov.rMaeN.cmpClv << "Ubicacion: " <<
                    (ftell(Nov) - sizeof(rNov)) / sizeof(rNov) << endl;
       }
       LecHV(Nov,rNov);
     }
     else {
       rMaeN = rMaeV;
       fwrite(&rMaeN,sizeof(rMaeN),1,MaeN);
       LecHV(MaeV,rMaeV);
     }
  freopen("CON","w",stdout);
}  //ApareoHV

void CerrarEliminarRenombrar(FILE *MaeV, FILE *MaeN, FILE *Nov) {

  fclose(MaeV);
  fclose(MaeN);
  fclose(Nov);
  remove("MaeVjo.Dat");
  rename("MaeNvo.Dat","MaeVjo.Dat");
} //CerrarEliminarRenombrar

int main() {
  FILE *MaeVjo,
          *MaeNvo,
          *Noved;

  Abrir(&MaeVjo,&MaeNvo,&Noved);
  ApareoHV(MaeVjo,MaeNvo,Noved);
  CerrarEliminarRenombrar(MaeVjo,MaeNvo,Noved);

  return 0;
}
```

Búsqueda Binaria o Dicotómica

La búsqueda binaria es un proceso en el cual los datos deben encontrarse ordenados por el valor de la clave a buscar, generalmente en orden ascendente, pero puede ocurrir que se encuentren ordenados en forma descendente. Nuestro modelo de estudio se considera ordenado en forma ascendente. El proceso consiste en tomar el valor que se encuentre en una posición media entre las posiciones de los extremos, así tratándose de elementos que se encuentran en un archivo, esos valores extremos serían **cero** y **filesize(f) – 1**. El punto medio se obtiene de aplicar la siguiente expresión:

$$med \leftarrow (pri + ult)\ div\ 2;$$

en donde: *pri* es el extremo inferior y *ult* es el extremo superior. Una vez obtenido el punto medio se accede a esa posición en el archivo y se lee el registro, luego se compara con el valor de la clave a buscar, pudiendo resultar alguna de las siguientes posibilidades:

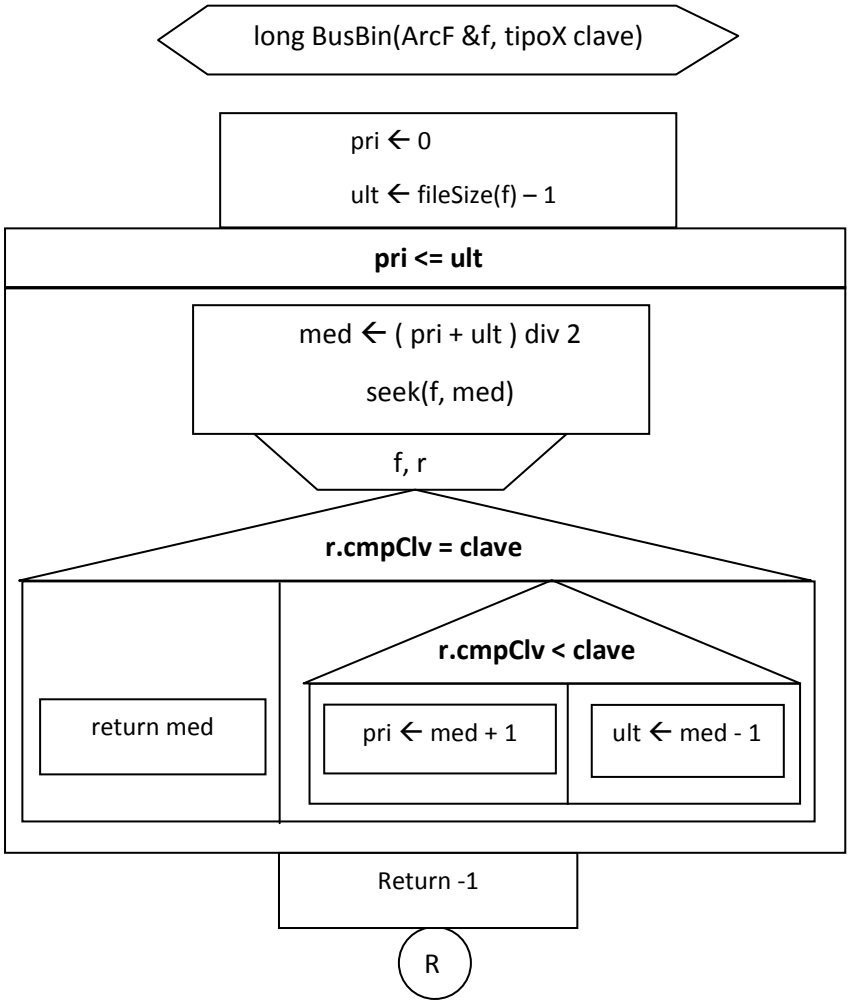
1. Que el valor del registro leído sea igual al valor clave a buscar, en ese caso, el proceso de búsqueda finalizará saliendo del ciclo cambiando el estado a una variable boolean e informaremos el dato a retornar, por ejemplo, en qué posición se encontró, o un valor boolean, o el valor de un campo o todo el registro, según lo que convenga al proceso.
2. Que el valor del registro leído sea menor al valor clave a buscar, en ese caso deberá cambiarse el valor del extremo inferior, de la siguiente manera: *pri* ← *med + 1*.
3. Caso contrario, es decir, el valor del registro leído es mayor al valor clave a buscar, en ese caso deberá cambiarse el valor del extremo superior, de la siguiente manera: *ult* ← *med – 1*.

Este método va eliminando, por cada vez que obtenemos un punto medio, una mitad del conjunto de datos que nos va quedando, y habiendo leído el registro de esta posición, su valor no es el de la clave a buscar. Por lo tanto, aquí tenemos del porque este método recibe el nombre de búsqueda binaria o dicotómica.

Si la clave no se encuentra, el proceso finalizará cuando el valor del extremo inferior sea mayor al valor del extremo superior, es decir, cuando *pri* es mayor a *ult*.

La búsqueda binaria garantiza realizar pocos accesos al disco, si ***n*** es el tamaño del archivo, la cantidad de accesos recién se duplica con n^2. Por ejemplo, si n = 50000 la cantidad de accesos máximos será de 16. La expresión que determina esto está en función de $\log_2(n)$. Recordemos entonces que **para poder aplicar este método, el conjunto de datos, deberá encontrarse ordenado por el valor de la clave a buscar**.

A continuación se describe el algoritmo para un caso general de búsqueda binaria. El mismo será una función que recibe como parámetros el archivo y la clave y retorna la posición, si se encontró un valor ≥ 0, caso contrario –1.



De acuerdo a las necesidades de un proceso, el módulo de búsqueda binaria podrá ser una función si lo que se retorne es un valor simple como una posición del archivo o un estado de verdadero o falso por si se encontró o no la clave, una cadena, un real o cualquier otro valor simple o incluso un puntero a un tipo de dato simple o estructurado. Pero además podrá devolver un valor por medio de sus parámetros pasados por referencia.

Búsqueda Lineal o Secuencial

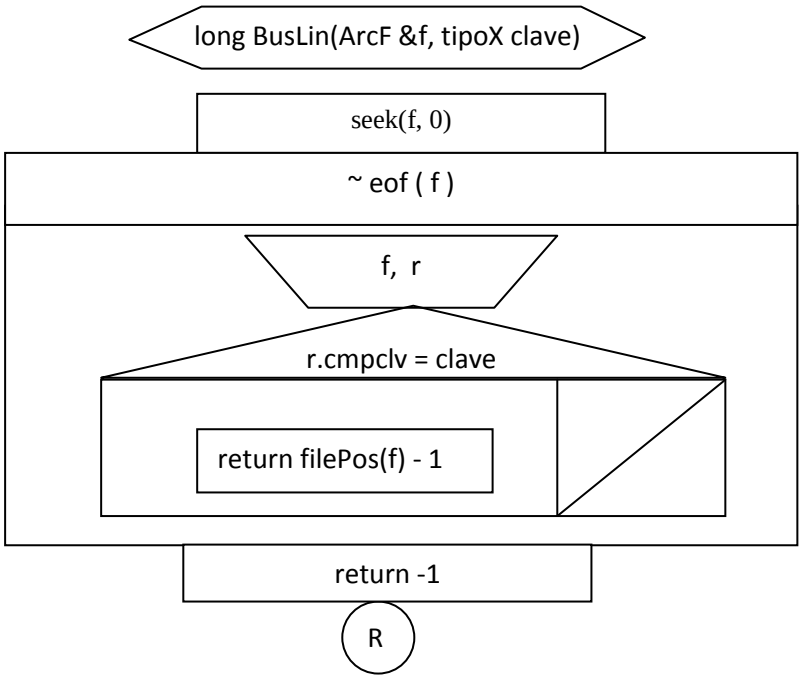
La búsqueda secuencial o lineal en archivos **no es aconsejable**, debido a que la cantidad de accesos a realizar para buscar el valor de una clave, en el peor de los casos será de n accesos, en el mejor de los casos será de 1 acceso y como promedio será de n/2 accesos, para un archivo con n registros.

Para aplicar la búsqueda binaria hemos visto en el tema anterior que el archivo debe encontrarse ordenado por el valor de un campo clave. El costo que tendremos en un archivo desordenado sería entonces tener que ordenarlo.

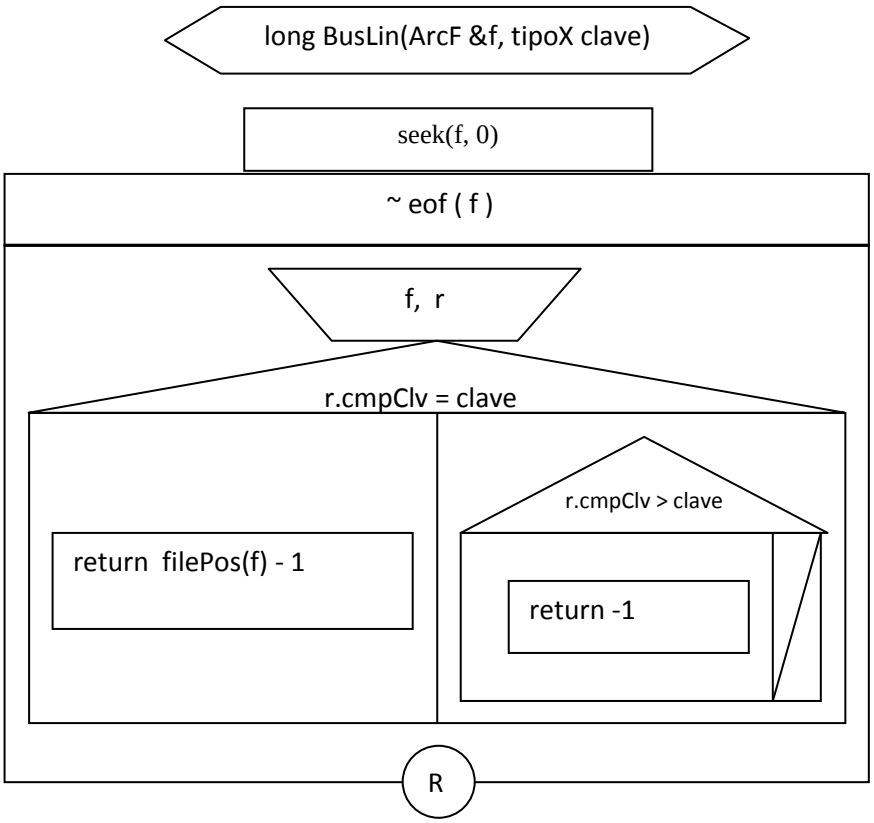
Podrían existir **casos excepcionales** en el cual podamos aplicar la búsqueda secuencial a un archivo desordenado, por lo tanto daremos el algoritmo correspondiente, para esta situación.

Buscar el valor de una clave en un archivo desordenado, primero debemos asegurar que el puntero al archivo esté ubicado al comienzo del mismo. Inicializamos el

estado de una variable de tipo boolean en falso, y al nombre de la función le asignamos –1 para indicar que aún el valor de la clave no se ha encontrado. El proceso se realizará mientras no se haya encontrado un registro que contenga el valor de la clave a buscar y mientras no sea fin de archivo. Dentro del proceso leemos un registro, luego lo comparamos con el valor de la clave, si es igual lo hemos encontrado, modificamos el estado de una variable de tipo boolean a verdadero y asignamos al nombre de la función la posición del archivo – 1. A continuación presentamos el algoritmo de búsqueda lineal.



Por último si un archivo se encuentra ordenado en concordancia con el valor de la clave a buscar, como se vio en párrafos anteriores, la búsqueda binaria es la técnica a emplear, pero, se podría emplear la búsqueda secuencial, el único impedimento es que serán necesarios más accesos al disco, ahora bien, si descartamos este hecho real, podríamos decir que el método de búsqueda secuencial visto anteriormente podría ser aplicado, pero sabiendo que el archivo ya se encuentra ordenado, existe una variante más óptima para estos casos. El proceso finalizaría ahora no solamente si se encuentra el valor de la clave a buscar, sino que también **nos detendríamos si el valor leído en una instancia se hiciera mayor al valor de la clave**, ya que no tendría razón de seguir avanzando en la búsqueda, debido a que todos los valores posteriores también serían mayores. La única razón de mostrar este método consistiría en que posteriormente se verán estructuras de datos en la cual el único acceso posible sería el secuencial y con esta estructura ordenada la búsqueda finalizaría al encontrarse el valor a buscar o bien cuando hayamos alcanzado un valor mayor al buscado. El algoritmo correspondiente de una búsqueda secuencial optimizada en archivos se detalla a continuación, -en realidad jamás utilizaremos este método- ya que la búsqueda binaria sería el método a emplear. Este método se muestra al solo efecto de que más adelante se verán estructuras de datos dinámicas lineales y ordenadas por el valor de una clave, y en estos casos buscar un elemento en estas listas será únicamente secuencial, no existiendo la posibilidad de realizar búsqueda binaria, por lo tanto, será una búsqueda secuencial optimizada, en el sentido que cuando nos topemos con un valor que se hizo mayor al buscado, en ese punto detendremos la búsqueda.



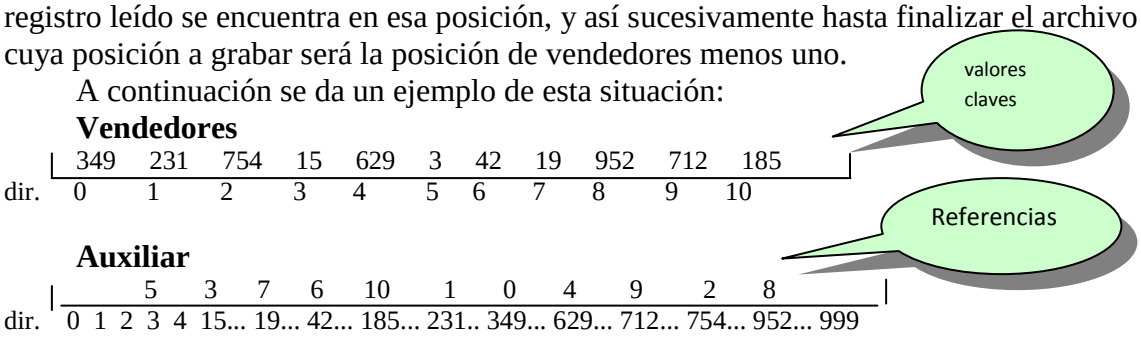
Recordar: Para archivos ordenados el método de búsqueda será el **dicotómico o binario**, en cambio si existe una relación 1:1, es decir el valor de la clave = dirección –*f. biyectiva*- o transformando el valor de la clave en una dirección válida en el archivo sin que se produzcan colisiones, el **direccionamiento directo** es el que debe emplearse. Si el archivo estuviera desordenado la creación de **archivos auxiliares** para lograr accesos más rápidos a los datos del archivo es otra de las posibilidades como técnicas a emplear.

El siguiente ejemplo ilustra una situación en la cual el archivo se encuentra desordenado. Suponiendo que este archivo fuera de vendedores conteniendo como campos el código del vendedor (3 dígitos), más otros campos de interés, y sabiendo que vamos a recurrir en forma reiterada a este archivo para buscar el valor de una clave por código de vendedor, los accesos secuenciales serían reiterados y por consecuencia la velocidad de ejecución de este proceso caería por debajo de lo óptimo o deseado. Aplicar el método de búsqueda binario no podríamos realizarlo debido a que el archivo no está ordenado y si bien existe la posibilidad de ordenarlo físicamente, aplicar ese método está fuera de las posibilidades conocidas hasta ahora, *es lo que se conoce como* ***método por copia*** *y se requieren un archivo del mismo tamaño que el original, -ahí va a quedar el archivo definitivo y ordenado-, más un espacio adicional para un archivo temporal de trabajo.* Pero como dijimos anteriormente este método no vamos a emplear. Otro método podría ser indexar el archivo a través del uso de un archivo de índices, pero acá también con los conocimientos logrados hasta ahora no contamos con la información necesaria para lograr este cometido. Otra posibilidad sería en crear un archivo en el cual pueda contener todas las claves posibles. La técnica a emplear en este último caso será de la creación de un archivo cascarón que contenga todas y solo todas las posibles claves. Para el ejemplo previamente indicado este archivo constará de 1000 registros, cuyas direcciones irán de 0 a 999.

Una vez creado el archivo cascarón que reserva los espacios suficientes, se procederá a leer secuencialmente el archivo de datos, por cada registro leído se accede con direccionamiento directo a la posición en el archivo auxiliar indicada por el valor de la clave, y grabamos la referencia o dirección del valor de esa clave en el archivo de datos, y así sucesivamente hasta finalizar el recorrido en el archivo de datos.

Por ejemplo si leemos del archivo vendedores el valor clave 349, con este valor nos ubicamos en el archivo auxiliar y grabamos 0, debido a que el primer registro leído se encuentra en esa posición, el segundo registro leído el valor clave es 231, con este valor nos ubicamos en el archivo auxiliar y grabamos 1, debido a que el segundo registro leído se encuentra en esa posición, y así sucesivamente hasta finalizar el archivo cuya posición a grabar será la posición de vendedores menos uno.

A continuación se da un ejemplo de esta situación:



Notar en el auxiliar que la *dirección-clave* 349, dice que se encuentra en la referencia 0 en vendedores, que la *dirección-clave* 231, dice que se encuentra en la dirección 1 en vendedores y así sucesivamente. Luego en el proceso, al requerir un dato de un vendedor, accedemos con la clave conocida en el auxiliar, a la dirección que coincide con su clave y con el valor que obtenemos de leer ese registro accedemos directamente a vendedores. Por ejemplo si conocemos la clave del vendedor 185 accedemos a esa posición en el auxiliar, leemos el registro y con el valor 10 accedemos a esa posición en vendedores. ¡¡¡Esto es mucho más óptimo que acceder secuencialmente a un archivo desordenado!!! incluso más rápido que acceder en forma dicotómica o binaria. El costo de aplicar este método es que debemos de contar con un espacio adicional en disco para poder crear el archivo cascarón.

Ejemplos de la G2Ej4 Crear archivo Articulos

```
/*
  Id.Programa: G2Ej04.cpp
  Autor.....: Lic. Hugo Cuello
  Fecha.....: oct-2013
  Comentario.: Creacion archivo de Articulos.
*/

#include <conio.h>
#include <iostream>
using namespace std;
#define CENTINELA 0

typedef unsigned short ushort;
typedef char str20[21];
struct sArt {
        ushort CodArt;
        str20  Descrip;
```

```
        float  PreUni;
};

// Prototipos -----------
void Abrir(FILE **);
void ObtDatos(sArt &);
// Fin Prototipos -------

main() {
        FILE *Articulos;
        sArt rArticulo;

        Abrir(&Articulos);
        ObtDatos(rArticulo);
        while (rArticulo.CodArt) {
                fwrite(&rArticulo,sizeof(rArticulo),1,Articulos);
                ObtDatos(rArticulo);
        }
        fclose(Articulos);
        return 0;
} // main

void Abrir(FILE **Art) {
        *Art = fopen("Articulos.Dat","wb");
} // Abrir

void ObtDatos(sArt &rArt) {
   do {
     gotoxy(5,1);
     cout << "Alta de Articulos.Dat";
     gotoxy(10,5); clreol();
     cout << "Cod.Articulo Fin = " << CENTINELA << ": ";
     cin >> rArt.CodArt;
   }
   while (!(rArt.CodArt <= 100));
   if (rArt.CodArt) {
      do {
          gotoxy(10,7); clreol();
          cout << "Descripcion: ";
          gets(rArt.Descrip);
      }
      while (!strcmp(rArt.Descrip,""));
      do {
          gotoxy(10,9); clreol();
          cout << "Pre.Unitario: ";
          cin >> rArt.PreUni;
      }
      while (!rArt.PreUni);
   }
} // ObtDatos
```

Ejemplo de la G2Ej5 Actualizar Precio en Articulos

```
/*
        Id.Programa: G2Ej05.cpp
        Autor.....: Lic. Hugo Cuello
        Fecha.....: oct-2013
        Comentario.: Actualizacion de Precio en archivo de Articulos.
*/

#include <conio.h>
#include <iostream>
using namespace std;

typedef char str20[21];
typedef unsigned short ushort;
struct sArt {
   ushort CodArt;
   str20   Descrip;
   float    PreUni;
};

// Prototipos ------------------
void Abrir(FILE **);
void ObtDato(float &);
void ActReg(FILE *, sArt );
void Cerrar(FILE *, FILE *);
// Fin Prototipos --------------

void Abrir(FILE **Art) {
        *Art = fopen("Articulo.Dat","r+b");
} // Abrir

void ObtDato(float &porc) {
        clrscr();
        gotoxy(10,5); clreol();
        cout << "Porcentaje: ";
        cin >> porc;
} // ObtDato

void ActReg(FILE *Art, sArt rArt) {
        fseek(Art,ftell(Art) - sizeof(rArt),SEEK_SET);
        fwrite(&rArt,sizeof(rArt),1,Art);
        fseek(Art,ftell(Art),SEEK_SET); //Necesario hacer para Actualizar posicion del
                                        //puntero
} // ActReg

void main() {
        FILE  *Articulos;
        sArt  rArticulo;
        float porcje;
```

```
        Abrir(&Articulos);
        ObtDato(porcje);
        while (fread(&rArticulo,sizeof(rArticulo),1,Articulos)) {
                rArticulo.PreUni *= (1 + porcje / 100);
                ActReg(Articulos,rArticulo);
        }
        fclose(Articulos);
}
```

Ejemplo de la G2Ej06 Mayores Precios en Articulos grabar en Mayores

```
/*
        Id.Programa: G2Ej06.cpp
        Autor.....: Lic. Hugo Cuello
        Fecha.....: oct-2013
        Comentario.: Genera archivo de Mayores precios de Articulos.
*/
#include <conio.h>
#include <iostream>
using namespace std;
typedef char str20[21];
typedef unsigned short ushort;
struct sArt {
        ushort CodArt;
        str20  Descrip;
        float  PreUni;
};

struct sMay {
  unsigned CodArt;
  float    PreUni;
};

// Prototipos -----------------
void Abrir(FILE **, FILE **);
void ObtDato(float &);
void GenReg(FILE *, sArt );
void Cerrar(FILE *, FILE *);
// Fin Prototipos -------------

void Abrir(FILE **Art, FILE **May) {
        *Art = fopen("Articulo.Dat","rb");
        *May = fopen("Mayores.Dat","wb");
} // Abrir

void ObtDato(float &impMax) {
        clrscr();
        gotoxy(10,5); clreol();
        cout << "Importe Maximo: ";
```

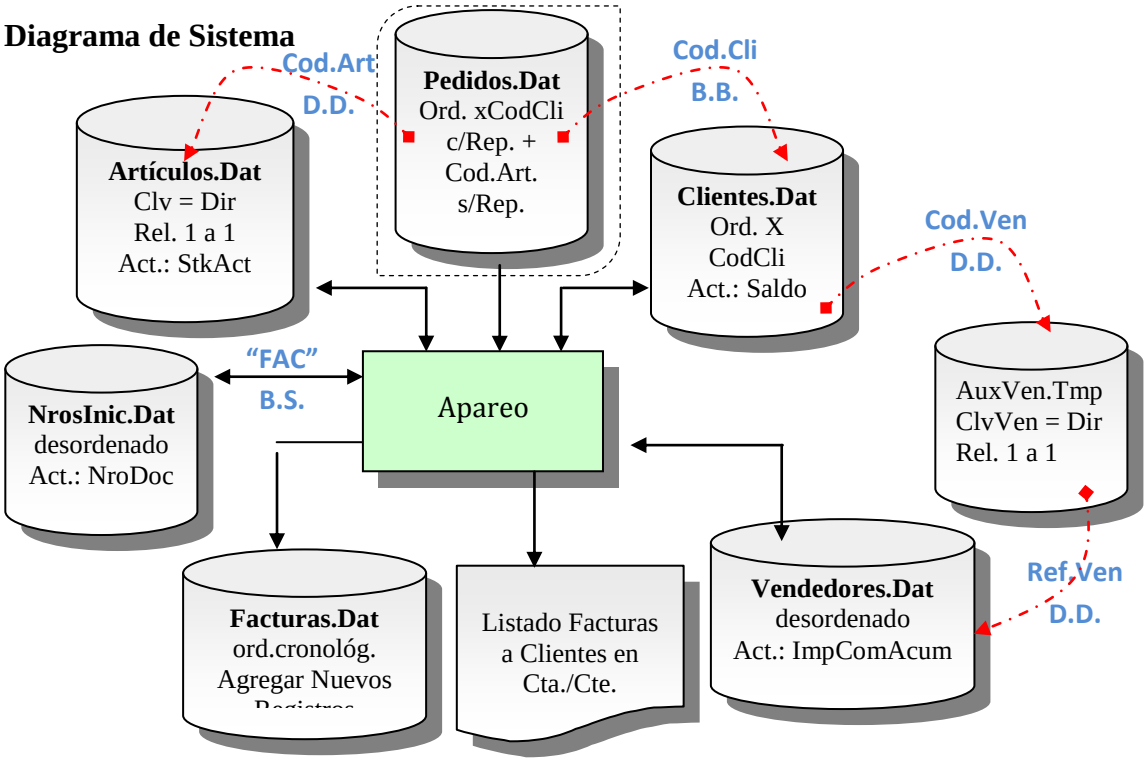
```
        cin >> impMax;
} // ObtDato

void GenReg(FILE *May, sArt rArt) {
        sMay rMay;
        rMay.CodArt = rArt.CodArt;
        rMay.PreUni = rArt.PreUni;
        fwrite(&rMay,sizeof(rMay),1,May);
} // GenReg

void Cerrar(FILE *Art, FILE *May) {
  fclose(Art);
  fclose(May);
} // Cerrar

void main() {
        FILE  *Articulos,
              *Mayores;
        sArt  rArticulo;
        float maxImp;

        Abrir(&Articulos,&Mayores);
        ObtDato(maxImp);
        while (fread(&rArticulo,sizeof(rArticulo),1,Articulos)) {
                if (rArticulo.PreUni > maxImp)
                  GenReg(Mayores,rArticulo);
        }
        Cerrar(Articulos,Mayores);
}
```

Relaciones entre archivos

Búsqueda Binaria, Secuencial, Direccionamiento Directo y Archivo Auxiliar

Ejemplo de la G2Ej11 de Facturas a Clientes en Cta./Cte.:

Diagrama de Sistema




```
/*
   Id.Programa: G2Ej11Facturas.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: ago-2014
   Comentario.: Emision de Facturas a Clientes en Cta.Cte.
                Relaciones entre archivos.
*/

#include<iomanip>
#include<iostream>
using namespace std;

#define IVA 0.21
#define not !
#define and &&

typedef char str5[6];
typedef char str10[11];
typedef char str15[16];
typedef char str20[21];
typedef unsigned short byte;
typedef unsigned word;

struct sPed {
```

```
  long codCli; //Ord. por codCli (con repeticion) + codArt (sin repeticion)
  byte codArt;
  word cant;
};

struct sCli {    //Ord. por codCli
  long  codCli;  //8 digitos.
  str20 razSoc,
        domic,
        local;
  word  codPos;
  char  codPcia;
  str15 nroCUIT;
  str20 formaPago;
  word  codVen;
  float saldo; //Actualizar el saldo
};

struct sArt {   //Relacion 1:1 clv = dir.
  byte  codArt; //1..100
  str20 marca,
        descrip;
  float precio;
  word  stockAct, //Actualizar el stockAct.
        stockMin,
        ptoRep;
  str10 uniMed;
  char  estado;
};

struct sVen {       //desordenado
  word  codVen;     //1..999
  str20 nomVen;
  float porcCom,
        impAcumCom; //Actualizar el impAcumCom.
};

struct sNro {       //desordenado
  str5 tipoDoc;
  long nroDoc;      //Actuaizar nroDoc.
};

struct sFecF {
  byte dia,
       mes;
  word year;
};

struct sFac {  //Ord.cronologico. El archivo existe y se deben agregar +registros.
  long  nroFac;
```

```
  sFecF fecFac;
  long  codCli;
  float impo;
  char  estado; //'A' = Adeuda, 'P' = Pagada.
};

struct sFec {
  int year,
      mes,
      dia,
      dsem;
};

struct sHor {
  int Hora,
      Min,
      Seg;
};

// Prototipos ----------------------------------------------------------
void Abrir(FILE **, FILE **, FILE **, FILE **,FILE **, FILE **, FILE **);
void ArmarAuxVen(FILE *, FILE *);
long GetDate(int &, int &, int &, int &);
void EmiteTapaObtFecha(sFec &);
void BusLinNrosInic(FILE *, str5 , sNro &);
void InicCabCli(byte &, float &, long &nDoc);
bool BusBinCli(FILE *, long , sCli &);
void BusDDVen(FILE *, word , sVen &, FILE *);
void EmiteCabCli(sCli , str20 , sFec , sHor, long );
void BusDDArt(FILE *, byte , sArt &);
void CalcDetFac(word , sArt &, byte &, float &,float &);
void ActArt(FILE *, sArt );
void EmiteDetFac(byte , sArt , word , float );
void CalcPieCli(float ,float &, float &, float &, sVen &);
void ActCli(FILE *, sCli );
void ActVen(FILE *, sVen );
void AgregarRegFac(FILE *, long , sFec , long , float );
void EmitePieFac(byte , float , float , float );
void ActNroI(FILE *, sNro );
void CerrarEliminar();
// Fin Prototipos ------------------------------------------------------

void Abrir(FILE **Ped, FILE **Cli, FILE **Art, FILE **Ven,FILE **nInic, FILE
                    **Fac, FILE **AuxVen) {

  *Ped    = fopen("Pedidos.Dat","rb");
  *Cli    = fopen("Clientes.Dat","r+b");
  *Art    = fopen("Articulos.Dat","r+b");
  *Ven    = fopen("Vendedores.Dat","r+b");
  *nInic  = fopen("NrosInic.Dat","r+b");
```

```
  *Fac    = fopen("Facturas.Dat","ab");
  *AuxVen = fopen("AuxVen.Tmp","w+b");
} //Abrir

void ArmarAuxVen(FILE *Ven, FILE *AuxVen) {
  int  refVen = -1;
  sVen rVen;

  for (int i = 0; i <= 999; i++)
    fwrite(&refVen,sizeof(int),1,AuxVen);
  while (fread(&rVen,sizeof(rVen),1,Ven)) {
    fseek(AuxVen,rVen.codVen * sizeof(rVen),SEEK_SET);
    refVen = (ftell(Ven) - sizeof(rVen)) / sizeof(rVen);
    fwrite(&refVen,sizeof(refVen),1,AuxVen);
  }
} //ArmarAuxVen

long GetDate(int &year, int &mes, int &dia, int &ds) {
  time_t     rawtime;
  tm *timeinfo;

  time ( &rawtime );
  timeinfo = localtime ( &rawtime );
  year = 1900 + timeinfo->tm_year;
  mes  = 1 + timeinfo->tm_mon;
  dia  = timeinfo->tm_mday;
  ds   = 1 + timeinfo->tm_wday;
  return (1900 + timeinfo->tm_year) * 10000 + (1 + timeinfo->tm_mon) * 100 +
          timeinfo->tm_mday;
} // GetDate

long GetTime(int &hh, int &mm, int &ss) {
  time_t     rawtime;
  tm *timeinfo;

  time ( &rawtime );
  timeinfo = localtime ( &rawtime );
  hh = timeinfo->tm_hour;
  mm = timeinfo->tm_min;
  ss = timeinfo->tm_sec;
  return timeinfo->tm_hour * 10000 + timeinfo->tm_min * 100 + timeinfo->tm_sec;
} // GetTime

void EmiteTapaObtFecha(sFec &rFec) {

  cout << "FACTURACION A CLIENTES" << endl;
  cout << " EN CUENTA CORRIENTE" << endl;
  GetDate(rFec.year,rFec.mes,rFec.dia,rFec.dsem);
} //EmiteTapaObtFecha
```

```
void BusLinNrosInic(FILE *nInic, str5 tDoc, sNro &rNroI) {
  bool sigo = true;

  while (sigo) {
    fread(&rNroI,sizeof(rNroI),1,nInic);
    if (strcmp(rNroI.tipoDoc,tDoc) == 0)
      sigo = false;
  }
} //BusLinNrosInic

void InicCabCli(byte &nItem, float &tFac, long &nDoc) {

  nItem = 0;
  tFac = 0.0;
  nDoc++;
} //InicCabCli

void ObtHora(sHor &rHor) {

  cout << "FACTURACION A CLIENTES" << endl;
  cout << " EN CUENTA CORRIENTE" << endl;
  GetTime(rHor.Hora,rHor.Min,rHor.Seg);
} //ObtHora

bool BusBinCli(FILE *Cli, long cCli, sCli &rCli) {
  int pri,
      ult,
      med;

  pri = 0;
  fseek(Cli,0L,SEEK_END);
  ult = ftell(Cli) / sizeof (rCli) - 1;
  while (pri <= ult) {
    med = (pri + ult) / 2;
    fseek(Cli,med * sizeof (rCli),SEEK_SET);
    fread(&rCli,sizeof (rCli),1,Cli);
    if (rCli.codCli == cCli)
      return true;
    else
      if (rCli.codCli < cCli)
        pri = med + 1;
      else
        ult = med - 1;
  }
  return false;
} // BusBinCli

void BusDDVen(FILE *Ven, word cVen, sVen &rVen, FILE *AuxiV) {
  word refVen;
```

```
  fseek(AuxiV,cVen * sizeof(rVen),SEEK_SET);
  fread(&refVen,sizeof(refVen),1,AuxiV);
  fseek(Ven,refVen * sizeof(rVen),SEEK_SET);
  fread(&rVen,sizeof(rVen),1,Ven);
} //BusDDVen

void EmiteCabCli(sCli rCli, str20 nVen, sFec rFec, sHor rHor, long nDoc) {
  cout << setw(38) << " " << "FACTURA: " << nDoc << endl;
  cout << setw(38) << " " << "FECHA..: " << setw(2) << rFec.dia << '-' <<
                                setw(2) << rFec.mes << '-' <<
                                setw(4) << rFec.year << endl;
  cout << setw(38) << " " << "HORA...: " << setw(2) << rHor.Hora << '-' <<
                                setw(2) << rHor.Min << '-' <<
                                setw(4) << rHor.Seg << endl;
  cout << "Cod. Cliente.: " << setw( 8) << rCli.codCli <<
         "Raz. Social..: " << setw(20) << rCli.razSoc <<
         "Domicilio....: " << setw(20) << rCli.domic <<
         "Forma de Pago: " << setw(20) << rCli.formaPago <<
         "Cod. Vendedor: " << setw( 3) << nVen << endl;
  cout << endl;
  cout << setw(40) << "-" << endl;
  cout << "Item  Cant.  Cod.Art. Descripcion       Pre.Uni.    T.Item" << endl;
  cout << setw(40) << "-" << endl;
} //EmiteCabCli

void BusDDArt(FILE *Art, byte cArt, sArt &rArt) {

  fseek(Art,cArt * sizeof(rArt),SEEK_SET);
  fread(&rArt,sizeof(rArt),1,Art);
} //BusDDArt

void CalcDetFac(word cant, sArt &rArt, byte &nItem, float &tItem,float &totFac) {

  nItem++;
  tItem = cant * rArt.precio;
  rArt.stockAct -= cant;
  totFac += tItem;
} //CalcDetFac

void ActArt(FILE *Art, sArt rArt) {

  fseek(Art,ftell(Art) - sizeof(rArt),SEEK_SET);
  fread(&rArt,sizeof(rArt),1,Art);
} //ActArt

void EmiteDetFac(byte nItem, sArt rArt, word cant, float tItem) {

  cout << setw(2) << nItem <<
         setw(5) << cant <<
         setw(3) << rArt.codArt <<
```

```
        setw(20) << rArt.descrip <<
        setw(8) << rArt.precio <<
        setw(8) << tItem << endl;
} //EmiteDetFac

void CalcPieCli(float tBruFac,float &impIVA, float &tNetoFac, float &saldo, sVen
                    &rVen) {

  impIVA = tBruFac * IVA;
  tNetoFac = tBruFac + impIVA;
  saldo += tNetoFac;
  rVen.impAcumCom += tBruFac * rVen.porcCom / 100;
} //CalcPieCli

void ActCli(FILE *Cli, sCli rCli) {

  fseek(Cli,ftell(Cli) - sizeof(rCli),SEEK_SET);
  fwrite(&rCli,sizeof(rCli),1,Cli);
} //ActCli

void ActVen(FILE *Ven, sVen rVen) {

  fseek(Ven,ftell(Ven) - sizeof(rVen),SEEK_SET);
  fwrite(&rVen,sizeof(rVen),1,Ven);
} //ActVen

void AgregarRegFac(FILE *Fac, long nFac, sFec rFec, long cCli, float tNetoFac) {
  sFac rFac;

  rFac.nroFac = nFac;
  rFac.fecFac.dia = rFec.dia;
  rFac.fecFac.mes = rFec.mes;
  rFac.fecFac.year = rFec.year;
  rFac.codCli = cCli;
  rFac.impo = tNetoFac;
  rFac.estado = 'A';
  fwrite(&rFac,sizeof(rFac),1,Fac);
} //AgregarRegFac

void EmitePieFac(byte nItem, float tBruFac, float impIVA, float tNetoFac) {
  byte i;

  for (i = 1; i <= 10 - nItem; i++)
    cout << endl;
  cout << setw(38) << " " << "Sub-Total.: " << setw(8) << tBruFac <<
        setw(38) << " " << "I.V.A.....: " << setw(8) << impIVA <<
        setw(38) << " " << "Total Neto: " << setw(8) << tNetoFac << endl;
} //EmitePieFac

void ActNroI(FILE *nInic, sNro rNroI) {
```

```
  fseek(nInic,ftell(nInic) - sizeof(rNroI),SEEK_SET);
  fwrite(&rNroI,sizeof(rNroI),1,nInic);
} //ActNroI

void CerrarEliminar() { //FILE *Ped, FILE *Cli, FILE *Art, FILE *Ven,FILE *nInic,
                        //FILE *Fac, FILE *AuxVen)

  fcloseall();
  remove("AuxVen.Tmp");
} //CerrarEliminar

int main() {
  FILE *Pedidos,
       *Clientes,
       *Articulos,
       *Vendedores,
       *NrosInic,
       *Facturas,
       *AuxVendedores;
  sFec rFecha;
  sHor rHora;
  sPed rPedido;
  sNro rNroInic;
  sCli rCliente;
  sVen rVendedor;
  sArt rArticulo;
  byte nroItem;
  float totItem,
        totBrutoFac,
        impIVA,
        totNetoFac;

Abrir(&Pedidos,&Clientes,&Articulos,&Vendedores,&NrosInic,&Facturas,
        &AuxVendedores);
  freopen("FacturasCtaCteCli.Lst","w",stdout);
  ArmarAuxVen(Vendedores,AuxVendedores);
  EmiteTapaObtFecha(rFecha);
  BusLinNrosInic(NrosInic,"FAC",rNroInic);
  fread(&rPedido,sizeof(rPedido),1,Pedidos);
  while (not feof(Pedidos)) {
    InicCabCli(nroItem,totBrutoFac,rNroInic.nroDoc);
    ObtHora(rHora);
    BusBinCli(Clientes,rPedido.codCli,rCliente);
    BusDDVen(Vendedores,rCliente.codVen,rVendedor,AuxVendedores);
    EmiteCabCli(rCliente,rVendedor.nomVen,rFecha,rHora,rNroInic.nroDoc);
    while (not feof(Pedidos) and rPedido.codCli == rCliente.codCli) {
      BusDDArt(Articulos,rPedido.codArt,rArticulo);
      CalcDetFac(rPedido.cant,rArticulo,nroItem,totItem,totBrutoFac);
      ActArt(Articulos,rArticulo);
```

```
     EmiteDetFac(nroItem,rArticulo,rPedido.cant,totItem);
     fread(&rPedido,sizeof(rPedido),1,Pedidos);
    }
    CalcPieCli(totBrutoFac,impIVA,totNetoFac,rCliente.saldo,rVendedor);
    ActCli(Clientes,rCliente);
    ActVen(Vendedores,rVendedor);
    AgregarRegFac(Facturas,rNroInic.nroDoc,rFecha,rCliente.codCli,totNetoFac);
    EmitePieFac(nroItem,totBrutoFac,impIVA,totNetoFac);
  }
  ActNroI(NrosInic,rNroInic);
  freopen("CON","w",stdout);
  CerrarEliminar();//usa fcloseall()  Pedidos, Clientes,Articulos, Vendedores, NrosInic,
                   //                  Facturas, AuxVendedores);
  return 0;
}
```

Ejemplo G2Ej13 Listado de Gastos ordenado Mes-Dia ImpAcum

```
/*
   Id.Programa: G2Ej13Gtos.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: ago-2014
   Comentario.: Listado de Gastos ord. x Mes y Dia Imp.Acum.
*/
#include<iomanip>
#include<iostream>
using namespace std;
  typedef unsigned short byte;
  struct sGto {
     byte  mes,
           dia;
     float impo;
};

// Prototipos ------------------------
void Abrir(FILE **, FILE **);
void ArmarAuxG(FILE *);
int CantDias(byte );
void AcumGto(FILE *, sGto );
string MesStr(unsigned );
void Listado(FILE *);
void CerrarEliminar(FILE *, FILE *);
// Fin Prototipos ---------------------

void Abrir(FILE **Gtos, FILE **AuxG) {

  *Gtos = fopen("Gastos.Dat","rb");
  *AuxG = fopen("AuxGtos.Tmp","w+b");
} //Abrir
```

```
void ArmarAuxG(FILE *AuxG) {
  float impAcum = 0;

  for (int i = 1; i <= 365; i++)
    fwrite(&impAcum,sizeof(impAcum),1,AuxG);
} //ArmarAuxG

int CantDias(byte mes) {
  int sumDias = 0;

  for (int mes_i = 1; mes_i < mes; mes_i++)
    switch (mes_i) {
      case  4:
      case  6:
      case  9:
      case 11: sumDias += 30;
               break;
      case  2: sumDias += 28;
               break;
      default: sumDias += 31;
    }
  return sumDias;
} //CantDias

void AcumGto(FILE *AuxG, sGto rGto) {
  float impAcum;
  int   pos;

  pos = CantDias(rGto.mes) + rGto.dia - 1;
  fseek(AuxG,pos * sizeof(float),SEEK_SET);
  fread(&impAcum,sizeof(impAcum),1,AuxG);
  impAcum += rGto.impo;
  fseek(AuxG,pos * sizeof(float),SEEK_SET);
  fwrite(&impAcum,sizeof(float),1,AuxG);
} //AcumGto

string MesStr(unsigned mes) {

  switch (mes) {
    case  1: return "Enero";
    case  2: return "Febrero";
    case  3: return "Marzo";
    case  4: return "Abril";
    case  5: return "Mayo";
    case  6: return "Junio";
    case  7: return "Julio";
    case  8: return "Agosto";
    case  9: return "Septiembre";
    case 10: return "Octubre";
    case 11: return "Noviembre";
```

```
   case 12: return "Diciembre";
   default: return "";
  }
} // MesStr

void Listado(FILE *AuxG) {
  float impAcum,
        totGral = 0.0,
        totMes,
        impMenDia,
        impMayMes = 0.0;
  byte  nroDiaMen,
        nroMesMay;

  rewind(AuxG);
  freopen("GtosMesDiaAcum.Lst","w",stdout);
  cout.precision(2);
  cout.setf(ios::fixed);
  cout << "Listado de Gastos ord. por Mes y Dia por Imp.Acum." << endl;
  for (int mes = 1; mes <= 12; mes++) {
    totMes = 0;
    impMenDia = 1E6;
    cout << "Mes: " << MesStr(mes) << endl;
    cout << setw(7) << " " << "Dia    Imp.Acu." << endl;
    for (byte dia = 1; dia <= CantDias(mes + 1) - CantDias(mes); dia++) {
      fread(&impAcum,sizeof(impAcum),1,AuxG);
      if (impAcum) {
        if (impAcum < impMenDia) {
          impMenDia = impAcum;
          nroDiaMen = dia;
        }
        cout << setw(8) << " " << setw(2) << dia
             << setw(4) << " " << setw(8) << impAcum << endl;
        totMes += impAcum;
      }
    }
    totGral += totMes;
    if (impMenDia > impMayMes) {
      impMayMes = impMenDia;
      nroMesMay = mes;
    }
    cout << "Tot. Mes: " << totMes << endl;
    cout << "Nro. Dia < $: " << nroDiaMen << endl << endl;
  }
  cout << "Tot. Anual: " << totGral << endl;
  cout << "Nro. Mes de > $ de los < Dias: " << setw(2) <<
          nroMesMay << "(" << MesStr(nroMesMay) << ")" << endl;
  freopen("CON","w",stdout);
} //Listado
```

```
void CerrarEliminar(FILE *Gtos, FILE *AuxG) {

  fclose(Gtos);
  fclose(AuxG);
  remove("AuxGtos.Tmp");
} //CerrarEliminar

int main() {
  FILE *Gastos,
       *AuxGtos;
  sGto rGto;

  Abrir(&Gastos,&AuxGtos);
  ArmarAuxG(AuxGtos);
  while (fread(&rGto,sizeof(rGto),1,Gastos))
    AcumGto(AuxGtos,rGto);
  Listado(AuxGtos);
  CerrarEliminar(Gastos,AuxGtos);
  return 0;
}
```

Archivos: FILE archivos en C/C++

Clasificación de archivos en el lenguaje C/C++

- ✓ **FILE* (variante para C / C++)**
- ✓ **STREAM (variante exclusiva para C++)**

En ambos casos se presentan las siguientes variantes:

1. **Texto**
2. **Binario**

Estilo FILE *

Archivo de texto

Un archivo de texto es un archivo con organización secuencial, solo se permite acceso secuencial a sus componentes. Las componentes de un archivo de texto se denominan líneas y cada línea finaliza con una marca, denominada fin de línea (**eoln**). El fin de un archivo está indicado por otra marca, denominada fin de archivo (**eof**). Cada una de las líneas de un archivo de texto es de longitud variable. Cada línea puede leerse en forma completa y almacenarse en una variable de tipo cadena. También es posible leer caracteres individuales de una línea, almacenada en una variable de tipo carácter. Otra alternativa de leer una línea es, si sabemos de antemano como está constituida cada una de las líneas, debe tener un formato fijo preestablecido, podremos leer cada uno de estos valores en una línea en correspondientes variables de acuerdo al tipo de valor a leer, p.e. si sabemos que por cada línea se registraron 4 valores y, siendo estos valores, un entero, un carácter, un valor real y una cadena de 20 caracteres como máximo, entonces para leer esta línea utilizamos cuatro variables según los tipos indicados recientemente, como ser **int** a, **char** b, **float** c, **char** d[20], luego por cada línea de este archivo de texto, presentan las mismas situaciones regulares como fue indicado. El archivo de texto debe su nombre porque al grabar datos en el archivo con los datos que están previamente en la memoria interna, **éstos datos están en una representación binaria, pero al grabarse en el archivo deben ser convertidos a texto** y **cuando se lean del archivo para volcar esos datos a la memoria interna también debe existir una conversión**. En el primer caso, esto es cuando grabamos la **conversión de datos es de binario a texto**, y en el segundo caso, esto es cuando leemos, la **conversión de datos es de texto a binario**.

Un archivo de texto es abierto solo para leer o solo para grabar.

Al abrir un archivo de texto de **solo lectura**, el archivo indicado debe existir y en este caso ubica el puntero al archivo al comienzo del mismo. En caso que no exista el archivo indicado ocasionará un error, por archivo inexistente.

Al abrir un archivo de texto **solo para grabar**, si no existe se crea desde cero o vacío. En caso que exista el archivo indicado se destruye todo su contenido y se lo crea desde cero o vacío. Por lo que se deberá tener sumo cuidado al abrir un archivo de texto

en esta modalidad ya que podríamos perder todos los datos, salvo que estemos completamente seguros de esta situación.

Un tercer modo de apertura de archivos de texto es el modo **añadir** o **agregar**, esto hace que el puntero, si el archivo existe previamente, se sitúa al final del mismo, es decir, sobre la marca de fin de archivo. Esto permite agregar nuevas componentes al archivo de texto. Si no existe previamente el archivo de texto se lo crea desde cero o vacío y ubica el puntero al final, sobre la marca de fin de archivo.

Ejemplo

```
void main() {
  int a;
  FILE *ft;

  ft = fopen("Entero.Txt","w");
  a = 23;
  fprintf(ft,"%d\n",a);
  fclose(ft);
}
```

Representemos en un gráfico las acciones realizadas en la codificación previa:

Al abrir el archivo

Vemos que el puntero coincide con la marca de EOF (End Of File = Fin De Archivo), el tamaño del archivo es cero.

Luego de grabar el dato

Observamos que el puntero luego de grabar una componente avanza una posición y la marca de EOF es movida al final del archivo. La longitud del archivo es de uno. La posición inicial está indicada por el valor cero. El contenido de la primer componente línea son dos caracteres, el carácter 2 y el carácter 3. Se observa también que al finalizar la línea se incorpora una marca de fin de línea (End Of Line = Fin De Línea).

El ejemplo de **programa** indicado más arriba, nos muestra la declaración de la variable ft como un puntero a un tipo **FILE**. Esta variable representa el nombre lógico del archivo, y este nombre es el que se utilizará posteriormente con las sentencias referidas al archivo.

fopen()

FILE * fopen(const char *nombreArchivo, const char *modoApertura)

fopen retorna un puntero a un tipo FILE, generalmente esto se asigna a una variable de ese tipo FILE, p.e., la siguiente definición: **FILE** *f; hace que f sea una variable puntero a un tipo archivo. Luego al abrir un archivo realizamos la siguiente sentencia:

f = fopen("Ejemplo.Txt","w")

Observamos que el valor del primer argumento "Ejemplo.Txt", indica el nombre físico del archivo, al que se le podrá incluir una unidad de disco y su ruta, el valor de este primer argumento podrá ser un valor constante, como en el ejemplo, o una variable o una expresión de cadena. El valor del segundo argumento indica el modo de apertura del archivo, en este caso, el archivo se abre para grabar, y precisamente es un archivo de texto, también podríamos haber escrito en el modo de apertura equivalente al ejemplo como "wt", pero, generalmente usamos la forma indicada en el ejemplo, ya que es la opción por defecto. La función fopen asocia un flujo con un manejador de archivo y esta retorna un puntero que identifica el flujo con el archivo.

Antes de operar con el archivo debemos abrirlo, esto se logra por medio de la función **fopen**, que establece la apertura del archivo, el cual debemos indicar como primer argumento el nombre físico del archivo, el cual puede incluir la unidad de disco y la ruta y, como segundo argumento indicamos el **modo de apertura** del archivo, los cuales podrán ser;

Modo de apertura

"r", "rt" solo lectura,
"w", "wt" solo escritura,
"a", "at" agregar o añadir más componentes al final del archivo,
"r+", "r+t", "rt+" para lectura/escritura,
"w+", "w+t", "wt+" para crearlo y leer,
"a+", "a+t", "at+" para agregar nuevas componentes y leer.

Estos modos de apertura de archivos son para **archivos de texto**, ya que si no indicamos el formato de archivo, asume por defecto el de texto. Una alternativa para indicar que nos referimos a archivos de texto, es indicando la letra "t" por texto, como en "rt", "wt", "at", aunque no es necesario indicarlo de esta manera, en cambio si indicamos el modo de apertura como, "r+t", "w+t", "a+t", o sus equivalentes "rt+", "wt+", "at+", abre un archivo para leer y grabar, o abre un archivo para grabar y leer, o abre un archivo para agregar más componentes y leer. Esta función retorna un puntero a FILE el cual es asignado a la variable de tipo FILE.

freopen()

FILE *freopen(**const char** *nombreArchivo, **const char** *modo, **FILE** *flujo)

Asocia un nuevo archivo con un flujo ya abierto. Reemplaza un archivo por un flujo abierto, cerrando el flujo. Es sumamente poderoso freopen para cambiar los flujos stdin, stdout o strerr. El valor retornado por la función si la operación fue exitosa es el argumento al flujo, caso contrario, retorna NULL.

Ejemplo

```
FILE *ft;
ft = freopen("Salida.Txt","w",stdout)
ft = freopen("CON","w",staout)
```

El primer ejemplo reencauza la salida por defecto que es la pantalla a otro dispositivo, hacia el disco, como archivo de texto. Es decir las salidas producidas por ejemplo con fprintf o con cout, en lugar de ir a la pantalla, ahora van a un archivo de texto.

El segundo ejemplo reencauza la salida hacia el dispositivo estándar o por defecto, es decir, las salidas producidas por ejemplo con fprintf o cout, dejan de ir al arhivo en disco, y vuelven a salir hacia la pantalla o monitor.

Las **operaciones** de lectura y escritura, permiten leer datos o grabar datos al archivo respectivamente y para ello se pueden utilizar diferentes funciones. En el ejemplo, el archivo se abrió para crearlo como archivo nuevo, en el cual solo se graban registros. La función utilizada para grabar en el ejemplo es, **fprintf**, función semejante a printf, solo que en este caso se indica como primer argumento el dispositivo al que hace referencia, en este caso, la variable que representa el nombre lógico del archivo de texto ft. También observamos que esta función graba datos en el archivo con formato, indicado como segundo argumento, y finalmente el tercer argumento o siguientes indican los valores a grabar, que se corresponde c/u. con el segundo argumento que establecen los formatos de c/u. de las componentes a guardar en el archivo. Una vez finalizadas todas las operaciones de lectura/escritura con el archivo, debemos cerrarlo, esta acción se realiza por medio de la función **fclose**. No solo cierra el archivo indicado, sino, que además vacía el buffer del archivo, forzando a enviarlo al archivo lo que quedó allí, si obviamente el archivo en cuestión fuese de escritura, debido a que éste no se llenó completamente, razón por la cual fclose obliga a forzar el vaciado, para no perder esos datos, que en caso de no haberse grabado, lo hubiéramos perdido. La función utilizada para leer datos de un archivo de texto es **fscanf** que funciona de forma semejante a su equivalente para leer datos desde el dispositivo estándar que es el teclado, pero en este caso lee datos de un archivo de texto. Otras funciones para lectura en un archivo de texto son, **fgetc** que lee un carácter, **fgets** que lee una cantidad de caracteres de un string y para grabar las funciones adicionales son **fputc** el cual graba un carácter, **fputs** que graba una cantidad de caracteres de un string.

Las sintaxis de estas funciones se indican a continuación:

Funciones de lectura en archivo de texto

int fgetc(**FILE** *flujo)
char *fgets(**char** *cad, **int** n, **FILE** *flujo)
int fscanf(**FILE** *flujo, **const char** *formato [,dirección, …])

Funciones de escritura en archivo de texto

int fputc(**int** car, **FILE** *flujo)
int fputs(**const char** *cad, **FILE** *flujo)
int fprintf(**FILE** *flujo, **const char** *formato [,argumento, …])

Ejemplo de archivo de texto

Agrega nuevas componentes al archivo luego lo recorre para leer

```
/*
   Id.Programa: fileTexto01.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2016
   Comentario.: Operaciones de lectura/escritura
                en archivo de texto
*/

#include <iostream>
using namespace std;

main() {
  FILE *t;
  char car,
       cad20[21];

  t = fopen("CadCar.Txt","a+"); // Si no existe lo crea o
                                // Si existe ubica puntero al final.
  cout << "Ingresar 5 caracteres..." << endl;
  for (int i = 1; i <= 5; i++) {
    cout << "Car: ";
    cin >> car;
    fputc(car,t);
    fputc(' ',t);
    if (i == 3)
      fputc('\n',t);
  }
  fputc('\n',t);
  cout << "Ingresar 5 cadenas -no frases-" << endl;
```

```
  for (int i = 1; i <= 5; i++){
    cout << "Cad: ";
    cin >> cad20;
    fputs(cad20,t); // corta al leer un espacio en blanco
    fputc(' ',t);
    if (i == 3)
      fputc('\n',t);
  }
  fputc('\n',t);
  cout << "Ingresar 5 cadenas o frases" << endl;
  for (int i = 1; i <= 5; i++){
    cout << "*Cad: ";
    gets(cad20); // lee incluso espacios en blanco
    fprintf(t,"%s",cad20);
    fputc(' ',t);
    if (i == 3)
      fputc('\n',t);
  }
  fputc('\n',t);
  rewind(t);
  car = fgetc(t);
  while (!feof(t)){ // o (car != EOF)
    while (car != '\n') {
      cout << car;
      car = fgetc(t);
    }
    cout << endl;
    system("pause");
    car = fgetc(t);
  }
  fclose(t);
} //main
```

Lee archivo de texto y muestra en pantalla

```
/*
   Id.Programa: 1.LeeTxtMstPan.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2014
   Comentario.: Lee archivo de texto y muestra datos en Pantalla
*/
```

```
#include <iomanip.h>
#include <iostream>
using namespace std;
typedef char str20[21];
 struct sVen {
   int   codVen,
         cant;
   float preUni;
   str20 descrip; // eq. char descrip[21];
 };

int main () {
 FILE *VentasTxt; // es el nombre logico del archivo.
 sVen rVenta;
 char car;

 VentasTxt = fopen("Ventas.Txt","r");
 cout.precision(2);
 cout.setf(ios::fixed);
 cout << "           Listado de Ventas" << endl;
 cout << endl << "Cod.Ven. Cant.  Pre. Uni.  Descripcion" << endl;
 fscanf(VentasTxt,"%d%d%f%c",&rVenta.codVen,
                        &rVenta.cant,
                        &rVenta.preUni,
                        &car);
 fgets(rVenta.descrip,22,VentasTxt);
 while (!feof(VentasTxt)) {
  //cout.setf(ios::right);
  cout << "  " << setw(3) << rVenta.codVen << " ";
  cout << "    " << setw(4) << rVenta.cant << " ";
  cout << "   " << setw(7) << rVenta.preUni << " ";
  rVenta.descrip[strlen(rVenta.descrip) - 1] = '\0';
  cout << " " << rVenta.descrip << endl;
  fscanf(VentasTxt,"%d%d%f%c",&rVenta.codVen,
                        &rVenta.cant,
                        &rVenta.preUni,
                        &car);
  fgets(rVenta.descrip,22,VentasTxt);
 }
 fclose(VentasTxt);

 return 0;
}
```

Ingresa datos por teclado y graba en archivo de texto

```
/*
   Id.Programa: 2.IngTecGraTxt.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2014
   Comentario.: Ingresa datos por teclado y graba en archivo de texto
*/

#include <conio.h>
#include <iomanip.h>
#include <iostream>
using namespace std;
typedef char str20[21];

struct sVen {
   int   codVen,
         cant;
   float preUni;
   str20 descrip;
  };

// Prototipos
void Titulos();
void ObtDatos(sVen &);
// Fin Prototipos

void Titulos() {
  gotoxy(10,1);
  cout << "Ingreso de Ventas de Articulos";
  gotoxy(10,5);
  cout << "Cod.Art.Fin=0:";
  gotoxy(10,8);
  cout << "Cantidad.....:";
  gotoxy(10,10);
  cout << "Descripcion..:";
  gotoxy(10,12);
  cout << "Pre.Unitario.:";
} // Titulos

void ObtDatos(sVen &rVen) {

  Titulos();
  gotoxy(25,5); clreol();
```

```
  cin >> rVen.codVen;
  if(rVen.codVen) {
    gotoxy(25,8); clreol();
    cin >> rVen.cant;
    gotoxy(25,10); clreol();
    gets(rVen.descrip);
    gotoxy(25,12); clreol();
    cin >> rVen.preUni;
  }
} // ObtDatos

int main () {
  FILE *aVentasTxt;
  sVen rVenta;

  aVentasTxt = fopen("VentasTec.Txt","w");
  cout.precision(2);
  cout.setf(ios::fixed);

  ObtDatos(rVenta);
  while (rVenta.codVen) {
    fprintf(aVentasTxt,"%3d %4d %7.2f %-20s\n",rVenta.codVen,
                                    rVenta.cant,
                                    rVenta.preUni,
                                    rVenta.descrip);
    ObtDatos(rVenta);
  }
  fclose(aVentasTxt);

  return 0;
}
```

Lee archivo de texto y graba en archivo binario

```
/*
   Id.Programa: 3.LeeTxtGraBin.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2014
   Comentario.: Lee archivo de texto y graba en archivo binario
*/

#include <iomanip.h>
```

```
#include <iostream>
using namespace std;
typedef char str20[21];

struct sVen {
   int    codVen,
          cant;
   float  preUni;
   str20  descrip;
 };

int main () {
 FILE  *aVentasTxt,
       *aVentas;
 sVen  rVenta;
 char  car;

 cout.precision(2);
 cout.setf(ios::fixed);
 aVentasTxt = fopen("Ventas.Txt","r");
 aVentas    = fopen("Ventas.Dat","wb");
 fscanf(aVentasTxt,"%d%d%f%c",&rVenta.codVen,&rVenta.cant,&rVenta.preUni,
          &car);
 fgets(rVenta.descrip,22,aVentasTxt);
 while (!feof(aVentasTxt)) {
    fwrite(&rVenta, sizeof(rVenta),1,aVentas);
     fscanf(aVentasTxt,"%d%d%f%c",&rVenta.codVen,&rVenta.cant, &rVenta.preUni,
              &car);
    fgets(rVenta.descrip,22,aVentasTxt);
 }
 fclose(aVentasTxt);
 fclose(aVentas);
 return 0;
}
```

Lee archivo binario y muestra en pantalla

```
/*
   Id.Programa: 4.LeeBinMstPan.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2014
   Comentario.: Lee archivo binario y muestra datos en pantalla
*/
```

```
#include <iomanip.h>
#include <iostream>
using namespace std;
typedef char str20[21];
struct sVen {
   int   codVen,
         cant;
   float preUni;
   str20 descrip;
  };

int main () {
  FILE *aVentas;
  sVen rVenta;
  char car;

  aVentas = fopen("Ventas.Dat","rb");
  cout.precision(2);
  cout.setf(ios::fixed);
  fread(&rVenta,sizeof(rVenta),1,aVentas);
  while (!feof(aVentas)) {
    cout.setf(ios::right);
    cout << setw(3) << rVenta.codVen << " ";
    cout << setw(4) << rVenta.cant << " ";
    cout << setw(7) << rVenta.preUni << " ";
    rVenta.descrip[strlen(rVenta.descrip) - 1] = '\0';
    cout << rVenta.descrip << endl;
    fread(&rVenta,sizeof(rVenta),1,aVentas);
  }
  fclose(aVentas);

  return 0;
}
```

Lee archivo binario y graba en archivo de texto

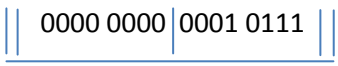
```
/*
   Id.Programa: 5.LeeBinGraTxt.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2014
   Comentario.: Lee archivo binario graba en archivo de texto
*/
```

```
#include <iomanip.h>
#include <iostream>
using namespace std;

typedef char str20[21];
struct sVen {
   int   codVen,
         cant;
   float preUni;
   str20 descrip;
  };

int main () {
  FILE *aVentas,
        *aVentasTxt;
  sVen rVenta;
  char car;

  aVentas = fopen("Ventas.Dat","rb");
  aVentasTxt = fopen("VentasCpy.Txt","w");
  cout.precision(2);
  cout.setf(ios::fixed);
  fprintf(aVentasTxt,"%5sListado de Ventas de Articulos\n\n"," ");
  fprintf(aVentasTxt,"Cod.Art. Cant.  Pre.Uni.   Descripcion\n");
  fread(&rVenta,sizeof(rVenta),1,aVentas);
  while (!feof(aVentas)) {
    rVenta.descrip[strlen(rVenta.descrip) - 1] = '\0';
    fprintf(aVentasTxt,"   %3d   %4d   %7.2f    %-20s\n",rVenta.codVen,
                                        rVenta.cant,
                                        rVenta.preUni,
                                        rVenta.descrip);
    fread(&rVenta,sizeof(rVenta),1,aVentas);
  }
  fclose(aVentas);
  fclose(aVentasTxt);

  return 0;
}
```

Archivo binario

Un archivo binario es la representación o imagen de la memoria interna, es decir, no existe una conversión de formato de los datos, como si existe en los archivos de

texto. Por lo que podríamos decir que el procesamiento de archivos binarios en más rápida que con los archivos de texto, debido a que no existen conversiones, tanto para volcar los datos del archivo a la memoria interna o desde la memoria interna al disco. Tomemos el siguiente ejemplo:



La representación anterior está representada en el formato de dato binario entero con signo, con un tamaño de 2 bytes, el bit de mayor peso del byte izquierdo, representa el signo del número en este caso es cero indicando positivo y la representación del número está en forma directa y los restantes 15 bits son utilizados para representar el número; si tomamos aquellos bits 1's, tenemos los siguientes valores de izquierda a derecha; 16+4+2+1 = 23 en base 10, por lo que la interpretación de la representación binaria es el valor positivo entero 23. Ahora bien, al grabar este dato en un archivo binario, y si luego abrimos este archivo con un editor de texto, notaremos que la representación está formada por dos caracteres que representan, por un lado el carácter nulo, es decir sin representación gráfica, seguida por el símbolo ↕

Los archivos binarios están estructurados por **componentes que denominamos registros en su sentido más amplio de la palabra**, aunque no necesariamente de tipo registro. Si por cada componente guardamos un solo dato, este dato puede que sea de tipo simple, como ser un entero con o sin signo, real, carácter, booleano o cadena. En cambio, si por cada componente guardamos más de un dato y, si cada uno de estos datos son de distinto tipo, en estos casos el tipo de componente lo definimos de tipo registro, en C/C++ tipo **struct** y, si por cada componente guardamos más de un dato y, si cada uno de estos datos son de igual tipo, en este otro caso el tipo de componente lo definimos de tipo arreglo, estructura de datos que se analizará más adelante. Un archivo binario no contiene marcas de fin de línea ya que sus componentes no se los denomina líneas, pero si contiene una marca al final del archivo, denominada marca de fin de archivo (eof). Las posiciones de cada componente se enumeran a partir de la **posición cero**, luego uno, dos y así sucesivamente. Podremos acceder a cada componente del archivo con acceso secuencial o bien con acceso al azar o aleatorio (random).

El **acceso secuencial** presenta la siguiente característica, se lee o se graba la componente en donde se encuentre localizado el puntero al archivo, luego de la operación de lectura o escritura el puntero del archivo avanza automáticamente a la próxima posición o componente. Este tipo de acceso es muy veloz cuando queremos acceder a las componentes en el mismo orden en que estas fueron creadas. Si por el contrario, queremos acceder a las componentes en distinto orden a como estos fueron grabados, este acceso es muy lento y no es aconsejable su implementación.

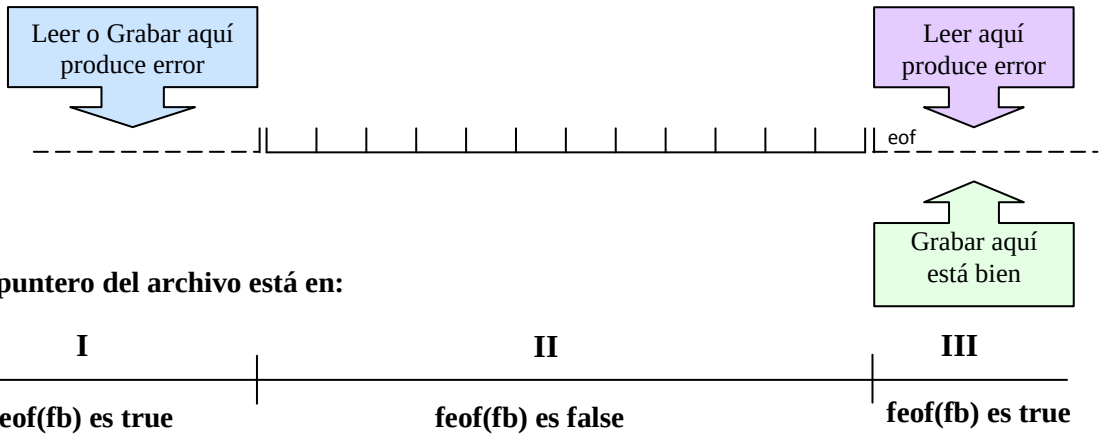
Por otro lado, para acceder a los registros en distinto orden en que fueron grabadas el acceso conveniente es el **acceso al azar**, este acceso permite acceder a una posición n, sin necesidad de pasar por aquellas componentes que le preceden, por lo que el tiempo de acceso es independiente del lugar en donde se encuentre el puntero del

archivo para llegar a una componente cualquiera. Vale decir, si estamos al comienzo del archivo, posición cero, y queremos ubicarnos en la posición 1563, el tiempo de acceso es igual, si el puntero del archivo se encuentra en la posición 1562 y queremos ubicarnos en la misma posición 1563. Cabe aclarar que el cambio de ubicación puede realizarse hacia adelante o hacia atrás, de donde nos encontramos para ir a la nueva posición. Además existe la posibilidad de desplazarnos a la nueva posición, tomando en cuenta desde la primer posición del archivo o desde la posición actual en donde está situado el puntero del archivo en ese instante.

Cuando leemos o grabamos en un archivo binario, se lee o se graba toda la componente completa, esto significa que si el tipo de componente fuese de tipo registro, entonces al leer o grabar esa componente, se leerán o grabarán todos los campos representados en el tipo registro.

Recordemos lo que fue indicado en párrafos previos en situaciones de lectura o escritura, si estamos fuera de los límites del archivo antes del inicio, despu{es del inicio, o dentro de su tamaño actual.

Cuando recorramos el archivo en forma secuencial para leer sus componente, sabremos que hemos llegado al final, al detectar la marca de fin de archivo. Esta marca se detecta por medio de la función **feof**, que presenta la siguiente forma:



Abrir un archivo

fopen()

FILE * fopen(**const char** *nombreArchivo, **const char** *modoApertura)

fopen retorna un puntero a un tipo FILE, generalmente esto se asigna a una variable de ese tipo FILE, p.e., la siguiente definición: **FILE** *fb; hace que fb sea una variable puntero a un tipo archivo. Luego al abrir un archivo realizamos la siguiente sentencia:

fb = fopen(“Ejemplo.Dat”,”wb”)

Observamos que el valor del primer argumento “Ejemplo.Dat”, que es una cadena de caracteres, indica el nombre físico del archivo, al que se le podrá incluir una unidad de disco y su ruta, el valor de este primer argumento podrá ser un valor constante, como en el ejemplo, o una variable o una expresión de cadena. El valor del segundo argumento indica el modo de apertura del archivo, que también es una cadena de

caracteres, en este caso, el archivo se abre para grabar, y precisamente es un archivo binario. La función fopen asocia un flujo con un manejador de archivo y esta retorna un puntero que identifica el flujo con el archivo.

Antes de operar con el archivo debemos abrirlo, esto se logra por medio de la función **fopen**, que establece la apertura del archivo, el cual debemos indicar como primer argumento el nombre físico del archivo, el cual puede incluir la unidad de disco y la ruta y, como segundo argumento indicamos el **modo de apertura** del archivo, los cuales podrán ser;

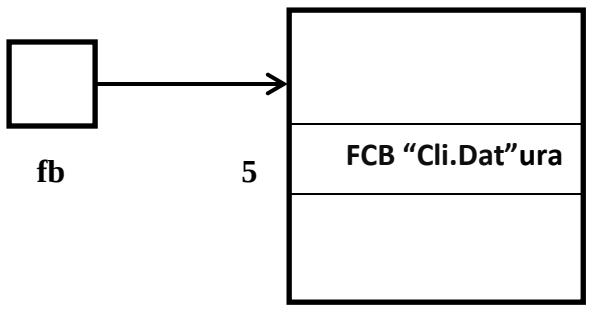
Modo de apertura

"rb" solo lectura,
"wb" solo escritura,
"rb+", "r+b" para lectura/escritura,
"wb+", "w+b" para crearlo y leer,
"ab" agrega, abre o crea solo grabar al final.
"ab+" agrega , abre o crea para actualizar en binario, graba al final del archivo.
Permite leer. Esta función retorna un puntero a FILE el cual es asignado a la variable de tipo FILE.

Si el archivo pudo ser abierto, entonces la función retorna un puntero genérico a FILE, caso contrario, la función retorna el valor **NULL**.

¿Qué sucede con fb luego de abrir el archivo del ejemplo anterior?

La función fopen retorna un puntero a la estructura FILE definido en <stdio.h> el cual contiene un descriptor, un entero, que indica un índice a la tabla de apertura de archivo. Al hacer una llamada de E/S como fprintf(fb,"%d",i), se localiza el **FCB** (*File Control Block*) en la tabla de apertura de archivo. Los datos localizados en este lugar se copian desde el FCB al disco, al abrir el archivo. El usuario no puede acceder a esta tabla, solo el S.O.



Detectar marca fin de archivo

feof()

int feof(**FILE** *flujo)

Macro que retorna un indicador de true, valor distinto de cero, si se alcanzó el fin de archivo de un flujo y este permanece hasta tanto no se realice un rewind, o sea, un rebobinado o no se realice un cierre del archivo, caso contrario, retorna un valor igual a cero, indicando que no se alcanzó el fin de archivo o **EOF**. El uso típico de esta macro es cuando recorremos para leer en forma secuencial un archivo, y en la cabecera del ciclo establecemos la condición, si no se alcanzó la marca de fin de archivo, como p.e. !feof(f), recordando que la primer lectura la realizamos antes de ingresar al ciclo, esto es diferente al caso del lenguaje Pascal, en que realizamos las lecturas dentro del ciclo y como primera acción de esta estructura de control.

Ejemplo:

```
// Lectura secuencial de un archivo binario
…
FILE *fb;

fb = fopen("ArcBin.Dat","rb");
fread(&rComp,sizeof(rComp),1,fb); // Observar 1er. lectura antes de ingresar al ciclo.
while (!feof(fb)) { // compara con distinto de cero !=0
  ProcReg(rComp);
  fread(&rComp,sizeof(rComp),1,fb); // Observar próx. lecturas, últ. acción en el ciclo.
}
fclose(fb);
```

Determinar posición del puntero y el tamaño del archivo

ftell()

long ftell(**FILE** *flujo)

Existe la posibilidad que ante el requerimiento del tamaño del archivo, podamos conocer este valor, por medio de la función **ftell**, cuando el puntero se encuentre posicionado sobre la última componente grabada en el archivo, **retorna la cantidad de bytes** hasta esa posición con respecto al comienzo del archivo, en cambio sí ocurre un error la función retorna -1L y pone a **errno** en un valor positivo.

Para saber cuántas componentes contiene basta hacer una división, en donde el numerador es **ftell** y el denominador es el tamaño de cada componente **sizeof**. Por ejemplo, si contamos con un archivo con 10 componentes y por cada componente guardamos 3 valores como ser **short** a, **float** b, **char** c, el tamaño de cada componente se puede determinar con la función sizeof, y si las variables a, b y c son campos de rComp con estructura struct sComp, entonces el sizeof(rComp) retornará 7 bytes, por lo que si la función ftell nos regresa 70, ya que el puntero del archivo asumimos que se encuentra ubicado en la componente 10; entonces el tamaño del archivo en cantidad de componentes 10 en este caso, quedaría determinado por la siguiente expresión:

Calcular el tamaño en cantidad de componentes

long ftell(fb) / sizeof(rComp)

Suponiendo que rComp, su tamaño, sizeof(rComp) = 16, y si el puntero al archivo asociado con este tipo de registro estuviese ubicado en su primera componente, o sea, posición = cero, entonces ftell(fb) retornará cero, si el puntero estuviese ubicado en la segunda componente, posición = uno, ftell(fb) retornará 16, si estuviese ubicado en la tercer componente, posición = dos, entonces ftell(fb) retornará 32, y así sucesivamente. Por lo tanto, si quisiéramos averiguar en un instante dado en qué posición se encuentra el puntero del archivo, basta realizar una división entre el valor informado por ftell(f) dividido por el tamaño de la componente, es decir, sizeof(rComp), según el ejemplo que hemos tomado para este caso.

Ejemplos

considerando el tamaño de rComp = 7 bytes.

Si ftell(fb) informa cero, entonces, ftell(fb) / sizeof(rComp) = 0.

Si ftell(fb) informa 7, entonces, ftell(fb) / sizeof(rComp) = 1.

Si ftell(fb) informa 14, entonces, ftell(fb) / sizeof(rComp) = 2.

Mover el puntero del archivo a una nueva posición

fseek()

int fseek(**FILE** *flujo, **long** Ofs, **int** ptoRef)

También podremos averiguar en qué posición se encuentra el puntero del archivo en un instante dado, esto se logra a través de la función **ftell**, p.e., si estamos en la posición 4, establecido de la siguiente manera, fseek(fb,4 * sizeof(rComp),SEEK_SET); en el cual, la función fseek realiza la acción de mover el puntero del archivo indicado en la dirección establecida como segundo argumento en cantidad de bytes, considerando a partir de una ubicación absoluta en relación al comienzo del archivo, establecida por la constante **SEEK_SET (valor 0)**, si en cambio quisiéramos indicarlo con una ubicación relativa, es decir, desde la posición que se encuentre el puntero actualmente, lo indicamos con la constante, **SEEK_CUR (valor 1)**, y si quisiéramos ubicar el puntero del archivo al final del mismo, se lo indica con la constante, **SEEK_END (valor 2)**; ftell(fb) regresará el valor 28 en cantidad de bytes, ya que, 4 * 7 = 28, siendo 7 el tamaño de rComp, luego aplicando la expresión ftell(fb) / sizeof(rComp) el resultado será que el puntero del archivo fb se encuentra en la posición cuatro.

Si la acción de mover el puntero fue exitosa, entonces, fseek retorna cero, caso contrario, retorna un valor distinto de cero.

Para **determinar el tamaño del archivo en cantidad de componentes**, debemos mover el puntero al final del archivo y dividirlo por el tamaño de la componente, como se indica a continuación:

fseek(fb,0L,SEEK_END)

ftell(fb) / sizeof(rComp)

Si quisiéramos ubicarnos en una **posición anterior del lugar en que nos encontramos** en un momento dado, se establece de la siguiente manera:

fseek(fb,0L,SEEK_END)

fseek(fb,ftell(fb) - sizeof(rComp),SEEK_SET)

Mover el puntero del archivo al comienzo

rewind

rewind(fb)

Para ubicarnos al **comienzo del archivo**, podemos hacerlo de las siguientes formas:

fseek(fb,0L,SEEK_SET)
rewind(**FILE** *flujo)
rewind(fb)

En ambos casos ubica el puntero sobre el primer byte del archivo, o dicho de otra manera en su primer componente, no obstante observamos que la segunda forma es más compacta su escritura.

Para mover el puntero a una posición n, podemos hacerlo de la siguiente forma:

fseek(fb,n * sizeof(rComp),SEEK_SET)

Para ubicarnos al final del archivo, la sentencia será

fseek(fb,0L,SEEK_END)

Ubicar el puntero sobre la última componente grabada

fseek(fb,0L,SEEK_END)

fseek(fb,ftell(fb) – sizeof(rComp),SEEK_SET)

La siguiente función de usuario determina el tamaño del archivo en cantidad de bytes:

Función fileSize

```
// Función que retorna el tamaño del archivo en cantidad de bytes.
long fileSize(FILE *flujo) {
  long curpos,
       length;

  curpos = ftell(flujo);
  fseek(flujo,0L,SEEK_END);
  length = ftell(flujo);
  fseek(flujo,curpos,SEEK_SET);
```

```
  return length;
}
```

Utilizando esta función de usuario, para ubicarnos al final del archivo, haríamos ahora:

fseek(fb,fileSize(fb),SEEK_SET);

Y para ubicarnos en la última componente grabada en el archivo, utilizando la función de usuario fileSize, haríamos ahora:

fseek(fb,fileSize(fb) – sizeof(rComp),SEEK_SET)

Operación de lectura en un archivo binario

fread()

size_t fread(**void** *bufer, **size_t** tamaño, **size_t** cantidadComponentes, **FILE** *flujo)

La función fread, lee una componente completa de un archivo binario en donde se encuentra el puntero del archivo en un instante dado, luego de esta operación de lectura, el puntero avanza automáticamente a la próxima posición. Cabe aclarar que si queremos recorrer leyendo secuencialmente debemos, antes de ingresar al proceso cíclico realizar la primer lectura en forma directa, a continuación la condición del ciclo determinará que si no es fin de archivo ingresar al ciclo, para procesar el dato leído y como última acción de este ciclo, realizar las próxima lecturas, también en forma directa. Esta operación de lectura notar que es diferente en el caso del lenguaje Pascal, ya que los mecanismos de lectura funcionar en forma diferente, recordando que en Pascal, todas las lecturas se realizan dentro del ciclo como primer acción y luego se procesa el registro leído. Solo en los casos particulares que requieran lecturas anticipadas, como en los procesos de corte de control y apareo de archivos, que imperiosamente sí requieren lectura anticipada, en Pascal debemos realizar una lectura especial, es decir, un módulo que primero pregunte si no es fin de archivo, en estos casos sí leeremos un registro y a una variable booleana le asignaremos el valor de falso, caso contrario, no leeremos ningún registro más y a esa misma variable booleana le asignaremos el valor de verdadero, la condición del ciclo en vez de preguntar por la función eof() de Pascal, lo reemplazaremos por la variable booleana, como p.e. **not** fdaf. Luego las próximas lecturas se realizarán dentro del ciclo como última acción invocando nuevamente a este módulo de lectura especial. Queda claro entonces, que en el lenguaje C/C++ este módulo especial no es necesario implementarlo, ya que, el mecanismo de lectura funciona de manera diferente al Pascal, debido que naturalmente, leemos en forma directa antes de ingresar al ciclo siempre.

Cabe aclarar que la cantidad de bytes leídos está dada por la siguiente expresión cantidadComponentes * tamaño. Si la operación de lectura fue exitosa, entonces, retorna la cantidad de ítems leídos y no cantidad de bytes. En cambio, si la operación falló, ya sea por un error o por haber encontrado la marca de fin de archivo, entonces, retorna un valor pequeño, posiblemente cero.

Ejemplo

fread(&rComp, sizeof(rComp), 1,fb)

Operación de escritura en un archivo binario

fwrite()

size_t fwrite(**const void** *bufer, **size_t** tamaño, **size_t** cantidadComponentes, **FILE** *flujo)

La función fwrite, graba una componente completa de un archivo binario en donde se encuentra el puntero del archivo en un instante dado, luego de esta operación de escritura, el puntero avanza automáticamente a la próxima posición.

Cabe aclarar que la cantidad de bytes grabados está dada por la siguiente expresión cantidadComponentes * tamaño. Si la operación de lectura fue exitosa, entonces, retorna la cantidad de ítems leídos y no cantidad de bytes. En cambio, si la operación falló, ya sea por un error o por haber encontrado la marca de fin de archivo, entonces, retorna un valor pequeño, posiblemente cero.

Ejemplo

fwrite(&rComp, sizeof(rComp), 1,fb)

Cerrar el archivo

fclose()

int fclose(**FILE** *flujo)

La función fclose, antes de cerrar el archivo vacía el búfer forzando a enviar su contenido al archivo, luego lo cierra. Retorna cero, si la operación fue exitosa, caso contrario, retorna EOF.

Cerrar todos los archivos abiertos

fcloseall()

fcloseall()

Cierra todos los flujos abiertos y retorna un valor positivo que indica la cantidad de flujos cerrados, en cambio sí ocurrió un error retorna **EOF**.

Renombrar un archivo

rename()

int rename(**const char** *ViejoNombre, **const char** *NuevoNombre)

La función rename, renombra el archivo indicado como ViejoNombre, por otro nombre indicado como NuevoNombre, en donde ViejoNombre y NuevoNombre son cadena de caracteres. No se permite el uso de comodines. Si la operación fue correcta,

entonces retorna cero, caso contrario, retorna -1L y pone a **errno** a uno de los siguientes valores:

ENOENT	Ningún archivo o directorio.
EACCES	Permiso negado.
ENOTSAM	Ningún dispositivo.

Eliminar un archivo

remove()

int remove(**const char** *NombreArchivo)

Esta macro, elimina el archivo indicado como NombreArchivo, antes de eliminar el archivo, se debe asegurar que el mismo esté cerrado. El NombreArchivo puede incluir la unidad de disco y la ruta en donde se encuentra el archivo. Si la operación fue exitosa, entonces, retorna cero, caso contrario, retorna -1L y pone a **errno** a uno de los siguientes valores:

ENOENT	Ningún archivo o directorio.
EACCES	Permiso negado.

Función de usuario equivalente a la sentencia de Pascal: truncate()

```
#include <dos.h>

struct REGPACK reg;

void truncate(FILE *flujo) {
  fseek(flujo,ftell(flujo),SEEK_SET); // Posición actual en cantidad de bytes.
  reg.r_ax = 0x4000; // Servicio grabar aleatorio.
  reg.r_bx = fileno(flujo); // Número de handle.
  reg.r_cx = 0; // Actualizar el tamaño del archivo.
  reg.r_ds = FP_SEG(fileno(flujo)); // Segmento del buffer de dato.
  reg.r_dx = FP_OFF(fileno(flujo)); // Desplazamiento del buffer de dato.
  intdos(&reg,&reg); // Invoca int 21h.
} // truncate
```

Definimos las siguientes variables puntero a FILE, local al módulo main:

```
FILE *f1 = NULL, *f2 = NULL, *f3 = NULL, *f4 = NULL, *f5 = NULL;
FILE *f6 0 NULL, *f7 = NULL, *f8 = NULL, *f9 = NULL, *f10 = NULL;
```

Invocamos al módulo desde la función main, para abrir estos archivos:

```
Abrir(&f1,&f2,&f3,&f4,&f5,&f6,&f7,&f8,&f9,&f10);
```

Módulo Abrir, cabecera y cuerpo de la función:

```
void Abrir(FILE **f1, FILE **f2, FILE **f3, FILE **f4, FILE **f5,
```

```
FILE **f6, FILE **f7, FILE **f8, FILE **f9, FILE **f10) {
  *f1 = fopen("Arcf1.Dat","rb");   // Abre f1 binario para solo lectura, sino existe, es error.
  *f2 = fopen("Arcf2.Dat","wb"); // Abre f2 binario para solo escritura, si existe, lo borra.
  *f3 = fopen("Arcf3.Dat","rb+"); // Abre f3 binario para lectura/escritura, error si no existe.
  *f4 = fopen("Arcf4.Dat","wb+"); //Abre f4 binario para escritura/lectura, borra si existe.
  *f5 = fopen("Arcf5.Dat","r+b"); // Abre f5 binario para lectura/escritura, error si no existe.
  *f6 = fopen("Arcf6.Txt","r");    // Abre f6 texto para solo lectura, sino existe, es error.
  *f7 = fopen("Arcf7.Txt","w");  // Abre f7 texto para solo escritura, si existe, lo borra.
  *f8 = fopen("Arc8.Txt","a");    // Abre f8 texto para agregar al final más componentes.
  *f9 = fopen("Arc9.Txt","rt");   // Abre f9 texto para solo lectura, sino existe, es error.
  *f10 = fopen("Arc10.Txt","rt+"); //Abre f10 texto para lectura/escritura, error si no existe.
}
```

Ejemplo de posición del puntero:

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

/*
  Id.Programa: PosFile.CPP
  Autor.....: Lic. Hugo Cuello
  Fecha.....: Junio-2013
  Comentario.: Grabar, Leer, Mover puntero de archivo al final, a
          una ubicación i, al inicio, en forma absoluta SEEK_SET,
          en forma relativa SEEK_CUR hacia adelante +n, 0 en la
          misma posición, o -1 una posición anterior; SEEK_END
          mueve al final del archivo. ftell(f) posición actual en
          cantidad de bytes en todos los casos.
          FILE *fopen(NombreArchivo,modoApertura) // Abrir archivo.
          FILE *f // Variable de tipo puntero a archivo.
          modo de apertura:
            r leer.
       w grabar.
       a agregar más componentes en archivo de texto.
            r+ abre para leer y grabar.
       w+ crea para grabar, luego poder leer en archivo de
         texto. También se y puede agregar una t. rt, wt, r+t,w+t.
            Si se agrega b entonces el archivo es binario:
            rb solo lee archivo binario.
            wb solo graba archivo binario.
       ab agrega registros al archivo binario.
       a+b agrega registros y leer archivo binario.
            r+b abre para leer o grabar archivo binario.
            w+b crea para grabar luego leer archivo binario.
            fclose(f) : Cierra un archivo.
            fread(&r,sizeof r,1,f)
            fwrite(&r,sizeof r,1,f)
            fseek(f,n * sizeof r,SEEK_SET | SEEK_CUR | SEEK_END)
            ftell(f)
```

```
        rewind(f)
*/
```

```
long fileSize(FILE *stream);

void main() {
  FILE *f;
  int  i;

  clrscr();
  f = fopen("Enteros.Dat","w+b");
  for (i = 0; i < 10; i++)
    fwrite(&i,sizeof i,1,f);
  printf("fileSize: %ld\n",fileSize(f)); //fileSize: 20
  rewind(f); //Mueve ptr.arch.a inicio, equivale a
  fseek(f,0L,SEEK_SET).
  printf("* \n"); //*
  for (i = 0; i < 3; i++) {
    fread(&i,sizeof i,1,f);
    printf("%d \n",i);  //0, 1, 2
  }
  fseek(f,0L,SEEK_SET); //Mueve ptr.arch.a inicio.
  printf("+ \n"); //+
  for (i = 0; i < 3; i++) {
    fread(&i,sizeof i,1,f);
    printf("%d \n",i); //0, 1, 2
  }
  fseek(f,0L,SEEK_SET);
  fread(&i,sizeof i,1,f);
  printf("->%d\n",i);
  fseek(f,0L,SEEK_END); //Mueve ptr.arch.final p/agregar+reg.x ej.
  printf("- \n");  //-
  for (i = 10; i < 15; i++)
    fwrite(&i,sizeof i,1,f);
  fseek(f,5*sizeof i,SEEK_SET);
  fread(&i,sizeof i,1,f);
  printf(":%d \n",i); //:5
  fseek(f,2*sizeof i,SEEK_CUR);
  fread(&i,sizeof i,1,f);
  printf(">%d \n",i); //>8
  fseek(f,ftell(f) - sizeof i,SEEK_SET); //Mueve ptr. pos.actual-1
  fread(&i,sizeof i,1,f);
  printf("<%d \n",i); //<8
  //fseek(f,0L,SEEK_END);
  fseek(f,fileSize(f) - sizeof i,SEEK_SET);
  fread(&i,sizeof i,1,f);
  printf(":) %d\n",i);   //:) 14
  printf(":( %ld \n",ftell(f) / sizeof i);  //:( 15
```

```
  printf("fileSize: %ld\n",fileSize(f)); //fileSize: 30
  fseek(f,0L,SEEK_SET);
  for (i = 0; i < 15; i++) {
    fread(&i,sizeof i,1,f);
    printf("%d ",i);
  }
  fclose(f);
}

long fileSize(FILE *stream) {
   long curpos,
        length;

   curpos = ftell(stream);
   fseek(stream, 0L, SEEK_END);
   length = ftell(stream);
   fseek(stream, curpos, SEEK_SET);
   return length;
}
```

Archivo de cabecera <stdio.h> o <cstdio>

Esta librería de C ejecuta operaciones de Entrada / Salida, las que tambi{en podrán ser ejecutadas en C++ utilizando la librería <cstdio>. Esta librería utiliza los llamados streams para operar con los dispositivos periféricos como ser teclados, impresoras, terminales o con cualquier otro tipo de archivos soportados por el sistema. Los streams son abstracciones para interactuar con ellos de un modo uniforme; todos los streams tienen propiedades similares independientemente de sus características individuales del medio físico con los que están asociados.

Los streams son manejados por la librería <cstdio> como punteros a objetos FILE. Un objeto a FILE identifica únivocamente a un stream, y es utilizado como un parámetro en las operaciones referente a los streams.

Existen tres streams estándar: stdin, stdout y stderr, los cuales son creados automáticamente y abiertod por todos los programas que utilizan la librería.

Propiedades de los stream

Los streams tienen ciertas propiedades que definen que funciones pueden ser utilizadas sobre ellos y como estos pueden tratar las entradas y salidas de datos. Algunas de estas propiedades están definidas en el momentos en que el stream es asociado a un archivo –abierto- utilizando la función fopen.

Acceso para Leer / Grabar -Read/Write-

Indica si el stream tiene acceso para leer o grabar –o ambos- al medio físico al cual está asociado.

Texto/ Binario -***Text / Binary***-

Un archivo de texto es un conjunto de líneas de texto, en donde, cada línea finaliza con un carácter de nueva línea.

Un archivo binario es una secuencia de caracteres grabados o leídos desde el medio físico sin ninguna conversión, teniendo una correspondencia uno a uno con los caracteres leídos o grabados al stream.

Buffer

Un buffer es un bloque de memoria en donde los datos son ubicados antes de iniciar una lectura o grabación con el arvhivo o dispositivo asociado.

Indicadores

Los stream tienen ciertos indicadores internos que indican su estado actual y que afectan a su comportamiento en las operaciones de entrada – salida.

Indicador de Error

Este indicador es puesto cuando un error ha ocurrido en una operación relacionada con el stream. El indicador puede ser verificado con la función ferror, y puede ser reseteado llamando a algunas de las siguientes funciones, **clearerr, freopen o rewind**.

Indicador Fin-De-Archivo

Este indicador cuando es puesto, indica después de la última operación de lectura o escritura encontró la marca de End of File o Fin de Archivo. Este idnciador puede ser verificado con la función **feof**, y puede ser reseteado llamando a las siguientes funciones, **clearerr o freopen** o llamando a cualquier función de reposicionamiento, **rewind, fseek y fsetpos**.

Indicador de posición

Es un puntero interno para cada stream el cual apunta al próximo carácter a ser leído o grabado en la próxima operación de I/O. Su valor puede ser obtenido por las funciones **ftell y fgetpos**, y puede ser cambiado utilizando las funciones de reposicionamiento **rewind, fseek y fsetpos**.

Funciones streams

En esta tabla se han dejado los comentarios explicativos de los distintos elementos en su idioma original. Ud. podrá consultar en otros puntos de este mismo documento los comentarios en el idioma nativo de estos mismos elementos. Solo se ha dejado una breve traducción del significado del elemento.

Operaciones con Archivos	
1	*int remove(const char * nomFis)* Elimina físicamente un archivo. Deletes the file whose name is specified in *filename*. This is an operation performed directly on a file identified by its *filename*; No streams are involved in the operation. Proper file access shall be available. ## Parameters `filename` C string containing the name of the file to be deleted. Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system). ## Return value If the file is successfully deleted, a zero value is returned. On failure, a nonzero value is returned. On most library implementations, the errno variable is also set to a system-specific error code on failure. ## Example `/* remove example: remove myfile.txt */` `#include <stdio.h>` `int main ()` `{` `  if( remove( "myfile.txt" ) != 0 )` `    perror( "Error deleting file" );` `  else` `    puts( "File successfully deleted" );` `  return 0;` `}` If the file `myfile.txt` exists before the execution and the program has write access to it, the file would be deleted and this message would be written to `stdout`: `File successfully deleted` Otherwise, a message similar to this would be written to `stderr`: `Error deleting file: No such file or directory`

2 *int rename(const char * nomVjo, const char * nomNvo)*
Renombra físicamente un archivo.

Changes the name of the file or directory specified by *oldname* to *newname*.

This is an operation performed directly on a file; No streams are involved in the operation.

If *oldname* and *newname* specify different paths and this is supported by the system, the file is moved to the new location.

If *newname* names an existing file, the function may either fail or override the existing file, depending on the specific system and library implementation.

Proper file access shall be available.

Parameters

`oldname`

C string containing the name of an existing file to be renamed and/or moved.
Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

`newname`

C string containing the new name for the file.
Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

Return value

If the file is successfully renamed, a zero value is returned.
On failure, a nonzero value is returned.
On most library implementations, the errno variable is also set to a system-specific error code on failure.

Example

```
1  /* rename example */
2  #include <stdio.h>
3
4  int main ()
5  {
6    int result;
7    char oldname[] ="oldname.txt";
8    char newname[] ="newname.txt";
9    result= rename( oldname , newname );
10   if ( result == 0 )
11     puts ( "File successfully renamed" );
12   else
13     perror( "Error renaming file" );
14   return 0;
```

```
15 }
```

If the file `oldname.txt` could be successfully renamed to `newname.txt` the following message would be written to `stdout`:

```
File successfully renamed
```

Otherwise, a message similar to this will be written to `stderr`:

```
Error renaming file: Permission denied
```

3 *int fclose(FILE * stream)*

Cierra un archivo y quita la asociación. Stream es un puntero a FILE. Retorna cero si el cierre fue correcto, sino EOF.

Closes the file associated with the *stream* and disassociates it.

All internal buffers associated with the stream are disassociated from it and flushed: the content of any unwritten output buffer is written and the content of any unread input buffer is discarded.

Even if the call fails, the stream passed as parameter will no longer be associated with the file nor its buffers.

Parameters

`stream`

Pointer to a FILE object that specifies the stream to be closed.

Return Value

If the stream is successfully closed, a zero value is returned. On failure, EOF is returned.

```
1 /* fclose example */
2 #include <stdio.h>
3 int main ()
4 {
5   FILE * pFile;
6   pFile = fopen ("myfile.txt","wt");
7   fprintf (pFile, "fclose example");
8   fclose (pFile);
9   return 0;
```

	`}`
4	*fcloseall()* Cierra todos los archivos abiertos por el usuario. Función propia de Borland.
5	*int fflush (FILE * stream)* Vacía un stream. If the given *stream* was open for writing (or if it was open for updating and the last i/o operation was an output operation) any unwritten data in its output buffer is written to the file. If *stream* is a null pointer, all such streams are flushed. In all other cases, the behavior depends on the specific library implementation. In some implementations, flushing a stream open for reading causes its input buffer to be cleared (but this is not portable expected behavior). The stream remains open after this call. When a file is closed, either because of a call to fclose or because the program terminates, all the buffers associated with it are automatically flushed. **Parameters** `stream` Pointer to a FILE object that specifies a buffered stream. **Return Value** A zero value indicates success. If an error occurs, EOF is returned and the error indicator is set (see ferror). **Example** In files open for update (i.e., open for both reading and writing), the stream shall be

flushed after an output operation before performing an input operation. This can be done either by repositioning (fseek, fsetpos, rewind) or by calling explicitly `fflush`, like in this example:

```
/* fflush example */
#include <stdio.h>
char mybuffer[80];
int main()
{
   FILE * pFile;
   pFile = fopen ("example.txt","r+");
   if (pFile == NULL) perror ("Error opening file");
   else {
     fputs ("test",pFile);
     fflush (pFile);    // flushing or repositioning
                                          required
     fgets (mybuffer,80,pFile);
     puts (mybuffer);
     fclose (pFile);
     return 0;
  }
}
```

6

*FILE *fopen(const char * nomArc, const char * modoAper)*
Abre un archivo. Retorna un puntero, sino NULL si el archivo no pudo ser abierto.

Opens the file whose name is specified in the parameter *filename* and associates it with a stream that can be identified in future operations by the FILE pointer returned.

The operations that are allowed on the stream and how these are performed are defined by the *mode* parameter.

The returned stream is *fully buffered* by default if it is known to not refer to an interactive device (see setbuf).

The returned pointer can be disassociated from the file by calling fclose or freopen. All opened files are automatically closed on normal program termination.

The running environment supports at least FOPEN_MAX files open simultaneously.

Parameters

`filename`

C string containing the name of the file to be opened.
Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

`mode`

C string containing a file access mode. It can be:

"r"	**read:** Open file for input operations. The file must exist.
"w"	**write:** Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a

	new empty file.
"a"	**append:** Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist.
"r+"	**read/update:** Open a file for update (both for input and output). The file must exist.
"w+"	**write/update:** Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
"a+"	**append/update:** Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

With the *mode* specifiers above the file is open as a *text file*. In order to open a file as a *binary file*, a "b" character has to be included in the *mode* string. This additional "b" character can either be appended at the end of the string (thus making the following compound modes: "rb", "wb", "ab", "r+b", "w+b", "a+b") or be inserted between the letter and the "+" sign for the mixed modes ("rb+", "wb+", "ab+").

The new C standard (C2011, which is not part of C++) adds a new standard subspecifier ("x"), that can be appended to any "w" specifier (to form "wx", "wbx", "w+x" or "w+bx"/"wb+x"). This subspecifier forces the function to fail if the file exists, instead of overwriting it.
If additional characters follow the sequence, the behavior depends on the library implementation: some implementations may ignore additional characters so that for example an additional "t" (sometimes used to explicitly state a *text file*) is accepted.
On some library implementations, opening or creating a text file with update mode may treat the stream instead as a binary file.
Text files are files containing sequences of lines of text. Depending on the environment where the application runs, some special character conversion may occur in input/output operations in *text mode* to adapt them to a system-specific text file format. Although on some environments no conversions occur and both *text files* and *binary files* are treated the same way, using the appropriate mode improves portability.
For files open for update (those which include a "+" sign), on which both input and output operations are allowed, the stream shall be flushed (fflush) or repositioned (fseek, fsetpos, rewind) before a reading operation that follows a writing operation. The stream shall be repositioned (fseek, fsetpos, rewind) before a writing operation that follows a reading operation (whenever that operation did not reach the end-of-file).

Return Value

If the file is successfully opened, the function returns a pointer to a FILE object that can be used to identify the stream on future operations.
Otherwise, a null pointer is returned.

On most library implementations, the errno variable is also set to a system-specific error code on failure.

Example

```
/* fopen example */
#include <stdio.h>
int main ()
{
  FILE * pFile;
  pFile = fopen ("myfile.txt","w");
  if (pFile!=NULL)
  {
    fputs ("fopen example",pFile);
    fclose (pFile);
  }
  return 0;
}
```

7

*FILE *freopen(const char * filename, const char * mode, FILE * stream)*

Reabre un stream con diferente archivo o modo.

Reuses *stream* to either open the file specified by *filename* or to change its access *mode*. If a new *filename* is specified, the function first attempts to close any file already associated with *stream* (third parameter) and disassociates it. Then, independently of whether that stream was successfuly closed or not, `freopen`opens the file specified by *filename* and associates it with the *stream* just as fopen would do using the specified *mode*.

If *filename* is a null pointer, the function attempts to change the *mode* of the stream. Although a particular library implementation is allowed to restrict the changes permitted, and under which circumstances.

The *error indicator* and *eof indicator* are automatically cleared (as if clearerr was called).

This function is especially useful for redirecting predefined streams like `stdin`, `stdout` and `stderr` to specific files (see the example below).

Parameters

`filename`

C string containing the name of the file to be opened.
Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).
If this parameter is a null pointer, the function attempts to change the mode of the *stream*, as if the file name currently associated with that stream had been used.

`mode`

C string containing a file access mode. It can be:

"r"	**read:** Open file for input operations. The file must exist.
"w"	**write:** Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.

"a"	**append:** Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist.
"r+"	**read/update:** Open a file for update (both for input and output). The file must exist.
"w+"	**write/update:** Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
"a+"	**append/update:** Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

With the *mode* specifiers above the file is open as a *text file*. In order to open a file as a *binary file*, a "b" character has to be included in the *mode* string. This additional "b" character can either be appended at the end of the string (thus making the following compound modes: "rb", "wb", "ab", "r+b", "w+b", "a+b") or be inserted between the letter and the "+" sign for the mixed modes ("rb+", "wb+", "ab+").

The new C standard (C2011, which is not part of C++) adds a new standard subspecifier ("x"), that can be appended to any "w" specifier (to form "wx", "wbx", "w+x" or "w+bx"/"wb+x"). This subspecifier forces the function to fail if the file exists, instead of overwriting it.
If additional characters follow the sequence, the behavior depends on the library implementation: some implementations may ignore additional characters so that for example an additional "t" (sometimes used to explicitly state a *text file*) is accepted.
On some library implementations, opening or creating a text file with update mode may treat the stream instead as a binary file.

`stream`

pointer to a FILE object that identifies the stream to be reopened.

Return value

If the file is successfully reopened, the function returns the pointer passed as parameter *stream*, which can be used to identify the reopened stream.
Otherwise, a null pointer is returned.
On most library implementations, the errno variable is also set to a system-specific error code on failure.

Example

```
1 /* freopen example: redirecting stdout */
2 #include <stdio.h>
3
4 int main ()
5 {
6   freopen ("myfile.txt","w",stdout);
7   printf ("This sentence is redirected to a file.");
8   fclose (stdout);
9   return 0;
```

```
10 }
```

This sample code redirects the output that would normally go to the standard output to a file called `myfile.txt`, that after this program is executed contains:

```
This sentence is redirected to a file.
```

8

*int fprintf (FILE * stream, const char * formato, ...)*

Graba datos formateados al stream.

Writes the C string pointed by *format* to the *stream*. If *format* includes *format specifiers* (subsequences beginning with `%`), the additional arguments following *format* are formatted and inserted in the resulting string replacing their respective specifiers.

After the *format* parameter, the function expects at least as many additional arguments as specified by *format*.

Parameters

`stream`

Pointer to a FILE object that identifies an output stream.

`format`

C string that contains the text to be written to the stream.

It can optionally contain embedded *format specifiers* that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A *format specifier* follows this prototype:

`%[flags][width][.precision][length]specifier`

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding

specifier	**Output**	**Example**
`d` *or* `i`	Signed decimal integer	392
`u`	Unsigned decimal integer	7235
`o`	Unsigned octal	610
`x`	Unsigned hexadecimal integer	7fa
`X`	Unsigned hexadecimal integer (uppercase)	7FA
`f`	Decimal floating point, lowercase	392.65
`F`	Decimal floating point, uppercase	392.65
`e`	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
`E`	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
`g`	Use the shortest representation: `%e` or `%f`	392.65
`G`	Use the shortest representation: `%E` or `%F`	392.65
`a`	Hexadecimal floating point, lowercase	-0xc.90fep-2
`A`	Hexadecimal floating point, uppercase	-0XC.90FEP-2
`c`	Character	a
`s`	String of characters	sample
`p`	Pointer address	b8000000
`n`	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
`%`	A `%` followed by another `%` character will write a single `%` to the stream.	%

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

flags	**description**

–	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or –) even for positive numbers. By default, only negative numbers are preceded with a – sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier).

width	**description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**description**
.number	For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, 0 is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

The *length* sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without *length* specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

	specifiers						
length	**d i**	**u o x X**	**f F e E g G a A**	**c**	**s**	**p**	**n**
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed	unsigned					signed

	char	char					char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Note that the `c` specifier takes an `int` (or wint_t) as argument, but performs the proper conversion to a `char` value (or a `wchar_t`) before formatting it for output.

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.
See <cinttypes> for the specifiers for extended types.

`... (additional arguments)`

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for `n`).
There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the *error indicator* (ferror) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, errno is set to `EILSEQ` and a negative number is returned.

Example

Edit & Run

```
1  /* fprintf example */
2  #include <stdio.h>
3
4  int main ()
5  {
6     FILE * pFile;
7     int n;
8     char name [100];
9
10    pFile = fopen ("myfile.txt","w");
11    for (n=0 ; n<3 ; n++)
```

```
12    {
13      puts ("please, enter a name: ");
14      gets (name);
15      fprintf (pFile, "Name %d [%-10.10s]\n",n+1,name);
16    }
17    fclose (pFile);
18
19    return 0;
20  }
```

This example prompts 3 times the user for a name and then writes them to `myfile.txt` each one in a line with a fixed length (a total of 19 characters + newline).
Two format tags are used:
`%d` : Signed decimal integer
`%-10.10s` : left-justified (`-`), minimum of ten characters (`10`), maximum of ten characters (`.10`), string (`s`).
Assuming that we have entered `John`, `Jean-Francois` and `Yoko` as the 3 names, `myfile.txt` would contain:

```
Name 1 [John      ]
Name 2 [Jean-Franc]
Name 3 [Yoko      ]
```

9 *int fscanf(FILE * stream, const char * formato, ...)*
Lee datos formateados desde el stream.

```
int fscanf ( FILE * stream, const char * format, ... );
```

Read formatted data from stream
Reads data from the *stream* and stores them according to the parameter *format* into the locations pointed by the additional arguments.

The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the *format* string.

Parameters

`stream`

Pointer to a FILE object that identifies the input stream to read data from.

`format`

C string that contains a sequence of characters that control how characters extracted from the stream are treated:

- Whitespace character: the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters -- seeisspace). A single whitespace in the *format* string validates any quantity of whitespace characters extracted from the *stream* (including none).
- Non-whitespace character, except format specifier (%): Any character

that is not either a whitespace character (blank, newline or tab) or part of a *format specifier* (which begin with a `%` character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of *format*. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.

- Format specifiers: A sequence formed by an initial percentage sign (`%`) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from the *stream* and stored into the locations pointed by the additional arguments.

A *format specifier* for `fscanf` follows this prototype:

```
%[*][width][length]specifier
```

Where the *specifier* character at the end is the most significant component, since it defines which characters are extracted, their interpretation and the type of its corresponding argument:

specifier	**Description**	**Characters extracted**
`i`, `u`	Integer	Any number of digits, optionally preceded by a sign (`+` or `-`). Decimal digits assumed by default (`0-9`), but a `0` prefix introduces octal digits (`0-7`), and`0x` hexadecimal digits (`0-f`).
`d`	Decimal integer	Any number of decimal digits (`0-9`), optionally preceded by a sign (`+` or `-`).
`o`	Octal integer	Any number of octal digits (`0-7`), optionally preceded by a sign (`+` or `-`).
`x`	Hexadecimal integer	Any number of hexadecimal digits (`0-9`, `a-f`, `A-F`), optionally preceded by `0x` or `0X`, and all optionally preceded by a sign (`+` or `-`).
`f`, `e`, `g` `a`	Floating point number	A series of decimal digits, optionally containing a decimal point, optionally preceeded by a sign (`+` or `-`) and optionally followed by the `e` or `E` character and a decimal integer (or some of the other sequences supported by `strtod`). Implementations complying with C99 also support hexadecimal floating-point format when preceded by `0x` or `0X`.
`c`	Character	The next character. If a *width* other than `1` is specified, the function reads exactly*width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
`s`	String of characters	Any number of non-whitespace characters, stopping at the first whitespace character found. A

		terminating null character is automatically added at the end of the stored sequence.
p	Pointer address	A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format %p in fprintf.
[*characters*]	Scanset	Any number of the characters specified between the brackets. A dash (-) that is not the first character may produce non-portable behavior in some library implementations.
[^*characters*]	Negated scanset	Any number of characters none of them specified as *characters* between the brackets.
n	Count	No input is consumed. The number of characters read so far from *stream* is stored in the pointed location.
%	%	A % followed by another % matches a single %.

Except for n, at least one character shall be consumed by any specifier. Otherwise the match fails, and the scan ends there.

The *format specifier* can also contain sub-specifiers: *asterisk* (*), *width* and *length* (in that order), which are optional and follow these specifications:

sub-specifier	**description**
*	An optional starting asterisk indicates that the data is to be read from the stream but ignored (i.e. it is not stored in the location pointed by an argument).
width	Specifies the maximum number of characters to be read in the current reading operation (optional).
length	One of hh, h, l, ll, j, z, t, L (optional). This alters the expected type of the storage pointed by the corresponding argument (see below).

This is a chart showing the types expected for the corresponding arguments where input is stored (both with and without a *length* sub-specifier):

	specifiers					
length	**d i**	**u o x**	**f e g a**	**c s [] [^]**	**p**	**n**
(none)	int*	unsigned int*	float*	char*	void**	int*
hh	signed char*	unsigned char*				signed char*
h	short int*	unsigned short int*				short int*
l	long int*	unsigned long int*	double*	wchar_t*		long int*
ll	long long int*	unsigned long long int*				long long int*
j	intmax_t*	uintmax_t*				intmax_t*
z	size_t*	size_t*				size_t*
t	ptrdiff_t*	ptrdiff_t*				ptrdiff_t*
L			long			

			`double*`			

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.

`...` *(additional arguments)*

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type. There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.

These arguments are expected to be pointers: to store the result of a `fscanf` operation on a regular variable, its name should be preceded by the *reference operator* (`&`) (see example).

Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set (feof or ferror). And, if either happens before any data could be successfully read, EOF is returned.

If an encoding error happens interpreting wide characters, the function sets errno to `EILSEQ`.

```
/* fscanf example */
#include <stdio.h>

int main () {
  char str [80];
  float f;
  FILE * pFile;

  pFile = fopen ("myfile.txt","w+");
  fprintf (pFile, "%f %s", 3.1416, "PI");
  rewind (pFile);
  fscanf (pFile, "%f", &f);
  fscanf (pFile, "%s", str);
  fclose (pFile);
  printf ("I have read: %f and %s \n",f,str);
  return 0;
}
```

10 *int printf(const char * formato, ...)*
Graba datos formateados al stdout.

Writes the C string pointed by *format* to the standard output (stdout).
If *format* includes *format specifiers* (subsequences beginning with `%`), the additional arguments following *format* are formatted and inserted in the resulting string replacing their respective specifiers.

Parameters

`format`

C string that contains the text to be written to stdout.
It can optionally contain embedded *format specifiers* that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A *format specifier* follows this prototype: [see compatibility note below]

```
%[flags][width][.precision][length]specifier
```

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	**Output**	**Example**
`d` *or* `i`	Signed decimal integer	`392`
`u`	Unsigned decimal integer	`7235`
`o`	Unsigned octal	`610`
`x`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (uppercase)	`7FA`
`f`	Decimal floating point, lowercase	`392.65`
`F`	Decimal floating point, uppercase	`392.65`
`e`	Scientific notation (mantissa/exponent), lowercase	`3.9265e+2`
`E`	Scientific notation (mantissa/exponent), uppercase	`3.9265E+2`
`g`	Use the shortest representation: `%e` or `%f`	`392.65`
`G`	Use the shortest representation: `%E` or `%F`	`392.65`
`a`	Hexadecimal floating point, lowercase	`-0xc.90fep-2`
`A`	Hexadecimal floating point, uppercase	`-0XC.90FEP-2`
`c`	Character	`a`
`s`	String of characters	`sample`
`p`	Pointer address	`b8000000`
`n`	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
`%`	A `%` followed by another `%` character will write a single `%` to the stream.	`%`

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

flags	**description**
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or −) even for positive numbers. By default, only negative numbers are preceded with a − sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier).

width	**description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**description**
.number	For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, 0 is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

The *length* sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without *length* specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

	specifiers						
length	**d i**	**u o x X**	**f F e E g G a A**	**c**	**s**	**p**	**n**
(none)	int	unsigned int	double	int	char*	void*	int*

hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Note regarding the `c` specifier: it takes an `int` (or wint_t) as argument, but performs the proper conversion to a `char` value (or a `wchar_t`) before formatting it for output.

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.
See <cinttypes> for the specifiers for extended types.

`... (additional arguments)`

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for `n`).
There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

On success, the total number of characters written is returned.

Return value

If a writing error occurs, the *error indicator* (ferror) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, errno is set to `EILSEQ` and a negative number is returned.

Example

```
1  /* printf example */
2  #include <stdio.h>
3
4  int main()
5  {
6     printf ("Characters: %c %c \n", 'a', 65);
7     printf ("Decimals: %d %ld\n", 1977, 650000L);
8     printf ("Preceding with blanks: %10d \n", 1977);
9     printf ("Preceding with zeros: %010d \n", 1977);
10    printf ("Some different radices: %d %x %o %#x %#o \n", 100,
11 100, 100, 100, 100);
```

```
12    printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416,
13 3.1416);
14    printf ("Width trick: %*d \n", 5, 10);
15    printf ("%s \n", "A string");
      return 0;
   }
```

Salida:

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick:    10
A string
```

11 *int scanf(const char * formato, ...)*

Lee datos formateados desde el stdin.

Reads data from stdin and stores them according to the parameter *format* into the locations pointed by the additional arguments.

The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the *format* string.

Parameters

`format`

C string that contains a sequence of characters that control how characters extracted from the stream are treated:

- Whitespace character: the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters -- seeisspace). A single whitespace in the *format* string validates any quantity of whitespace characters extracted from the *stream* (including none).
- Non-whitespace character, except format specifier (%): Any character that is not either a whitespace character (blank, newline or tab) or part of a *format specifier* (which begin with a `%` character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of *format*. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.
- Format specifiers: A sequence formed by an initial percentage sign (`%`) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from the *stream* and stored into the locations

pointed by the additional arguments.

A *format specifier* for `scanf` follows this prototype:

`%[*][width][length]specifier`

Where the *specifier* character at the end is the most significant component, since it defines which characters are extracted, their interpretation and the type of its corresponding argument:

specifier	**Description**	**Characters extracted**
`i`	Integer	Any number of digits, optionally preceded by a sign (`+` or `-`). Decimal digits assumed by default (`0-9`), but a `0` prefix introduces octal digits (`0-7`), and`0x` hexadecimal digits (`0-f`). *Signed* argument.
`d` *or* `u`	Decimal integer	Any number of decimal digits (`0-9`), optionally preceded by a sign (`+` or `-`). `d` is for a *signed* argument, and `u` for an *unsigned*.
`o`	Octal integer	Any number of octal digits (`0-7`), optionally preceded by a sign (`+` or `-`). *Unsigned* argument.
`x`	Hexadecimal integer	Any number of hexadecimal digits (`0-9`, `a-f`, `A-F`), optionally preceded by `0x` or `0X`, and all optionally preceded by a sign (`+` or `-`). *Unsigned* argument.
`f`, `e`, `g` `a`	Floating point number	A series of decimal digits, optionally containing a decimal point, optionally preceeded by a sign (`+` or `-`) and optionally followed by the `e` or `E` character and a decimal integer (or some of the other sequences supported by `strtod`). Implementations complying with C99 also support hexadecimal floating-point format when preceded by `0x` or `0X`.
`c`	Character	The next character. If a *width* other than `1` is specified, the function reads exactly*width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
`s`	String of characters	Any number of non-whitespace characters, stopping at the first whitespace character found. A terminating null character is automatically added at the end of the stored sequence.
`p`	Pointer address	A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format `%p` in fprintf.
`[`*characters*`]`	Scanset	Any number of the characters specified between the brackets. A dash (`-`) that is not the first character may produce non-portable behavior in some library implementations.

`[^characters]`	Negated scanset	Any number of characters none of them specified as *characters* between the brackets.
`n`	Count	No input is consumed. The number of characters read so far from `stdin` is stored in the pointed location.
`%`	`%`	A `%` followed by another `%` matches a single `%`.

Except for `n`, at least one character shall be consumed by any specifier. Otherwise the match fails, and the scan ends there.

The *format specifier* can also contain sub-specifiers: *asterisk* (`*`), *width* and *length* (in that order), which are optional and follow these specifications:

sub-specifier	**description**
`*`	An optional starting asterisk indicates that the data is to be read from the stream but ignored (i.e. it is not stored in the location pointed by an argument).
width	Specifies the maximum number of characters to be read in the current reading operation (optional).
length	One of `hh`, `h`, `l`, `ll`, `j`, `z`, `t`, `L` (optional). This alters the expected type of the storage pointed by the corresponding argument (see below).

This is a chart showing the types expected for the corresponding arguments where input is stored (both with and without a *length* sub-specifier):

	specifiers					
length	**d i**	**u o x**	**f e g a**	**c s [] [^]**	**p**	**n**
(none)	`int*`	`unsigned int*`	`float*`	`char*`	`void**`	`int*`
`hh`	`signed char*`	`unsigned char*`				`signed char*`
`h`	`short int*`	`unsigned short int*`				`short int*`
`l`	`long int*`	`unsigned long int*`	`double*`	`wchar_t*`		`long int*`
`ll`	`long long int*`	`unsigned long long int*`				`long long int*`
`j`	`intmax_t*`	`uintmax_t*`				`intmax_t*`
`z`	`size_t*`	`size_t*`				`size_t*`
`t`	`ptrdiff_t*`	`ptrdiff_t*`				`ptrdiff_t*`
`L`			`long double*`			

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.

`... (additional arguments)`

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type. There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the

function.
These arguments are expected to be pointers: to store the result of a `scanf` operation on a regular variable, its name should be preceded by the *reference operator* (`&`) (see example).

Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set (feof or ferror). And, if either happens before any data could be successfully read, EOF is returned.

If an encoding error happens interpreting wide characters, the function sets errno to `EILSEQ`.

Example

```
1  /* scanf example */
2  #include <stdio.h>
3
4  int main ()
5  {
6    char str [80];
7    int i;
8
9    printf ("Enter your family name: ");
10   scanf ("%79s",str);
11   printf ("Enter your age: ");
12   scanf ("%d",&i);
13   printf ("Mr. %s , %d years old.\n",str,i);
14   printf ("Enter a hexadecimal number: ");
15   scanf ("%x",&i);
16   printf ("You have entered %#x (%d).\n",i,i);
17
18   return 0;
19 }
```

This example demonstrates some of the types that can be read with `scanf`:

```
Enter your family name: Soulie
Enter your age: 29
Mr. Soulie , 29 years old.
Enter a hexadecimal number: ff
You have entered 0xff (255).
```

12 *int sprintf (char * str, const char * formato, ...)*
Graba datos formateados a string.

Composes a string with the same text that would be printed if *format* was used on printf, but instead of being printed, the content is stored as a *C string* in the buffer pointed by *str*.

The size of the buffer should be large enough to contain the entire resulting string (see snprintf for a safer version).

A terminating null character is automatically appended after the content.

After the *format* parameter, the function expects at least as many additional arguments as needed for *format*.

Parameters

`str`

Pointer to a buffer where the resulting C-string is stored.
The buffer should be large enough to contain the resulting string.

`format`

C string that contains a format string that follows the same specifications as *format* in printf (see printf for details).

`... (additional arguments)`

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for `n`).
There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

Return Value

On success, the total number of characters written is returned. This count does not include the additional null-character automatically appended at the end of the string.
On failure, a negative number is returned.

Example

```
1  /* sprintf example */
2  #include <stdio.h>
3
4  int main ()
5  {
6    char buffer [50];
7    int n, a=5, b=3;
8    n = sprintf (buffer, "%d plus %d is %d", a, b, a+b);
9    printf ("[%s] is a string %d chars long\n",buffer,n);
10   return 0;
11 }
```

Salida:

```
[5 plus 3 is 8] is a string 13 chars long
```

13 *int sscanf (const char * s, const char * formato, ...)*
Lee datos formateados desde una cadena.

Reads data from *s* and stores them according to parameter *format* into the locations given by the additional arguments, as if scanf was used, but reading from *s* instead of the standard input (stdin).

The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the *format* string.

Parameters

`s`

C string that the function processes as its source to retrieve the data.

`format`

C string that contains a format string that follows the same specifications as *format* in scanf (see scanf for details).

`... (additional arguments)`

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type. There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.

Return Value

On success, the function returns the number of items in the argument list successfully filled. This count can match the expected number of items or be less (even zero) in the case of a matching failure.
In the case of an input failure before any data could be successfully interpreted, EOF is returned.

Example

```
1  /* sscanf example */
2  #include <stdio.h>
3
4  int main ()
5  {
6    char sentence []="Rudolph is 12 years old";
7    char str [20];
8    int i;
9
10   sscanf (sentence,"%s %*s %d",str,&i);
11   printf ("%s -> %d\n",str,i);
12
13   return 0;
14 }
```

Salida:

```
Rudolph -> 12
```

14 *int vfprintf (FILE * stream, const char * formato, va_list arg)*

Graba datos formateados desde el stream en una lista de argumentos variable.

Writes the C string pointed by *format* to the *stream*, replacing any *format specifier* in the same way as printf does, but using the elements in the variable argument list identified by *arg* instead of additional function arguments.

Internally, the function retrieves arguments from the list identified by *arg* as if va_arg was used on it, and thus the state of *arg* is likely altered by the call.

In any case, *arg* should have been initialized by va_start at some point before the call, and it is expected to be released byva_end at some point after the call.

Parameters

`stream`

Pointer to a FILE object that identifies an output stream.

`format`

C string that contains a format string that follows the same specifications as *format* in printf (see printf for details).

`arg`

A value identifying a variable arguments list initialized with va_start.
va_list is a special type defined in <cstdarg>.

Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the *error indicator* (ferror) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, errno is set to `EILSEQ` and a negative number is returned.

Example

```
1  /* vfprintf example */
2  #include <stdio.h>
3  #include <stdarg.h>
4
5  void WriteFormatted (FILE * stream, const char * format, ...)
6  {
7    va_list args;
8    va_start (args, format);
9    vfprintf (stream, format, args);
10   va_end (args);
11 }
12
```

```
13 int main ()
14 {
15    FILE * pFile;
16
17    pFile = fopen ("myfile.txt","w");
18
19    WriteFormatted (pFile,"Call with %d variable
20 argument.\n",1);
21    WriteFormatted (pFile,"Call with %d variable
22 %s.\n",2,"arguments");
23
24    fclose (pFile);
25
    return 0;
}
```

The example demonstrates how the `WriteFormatted` can be called with a different number of arguments, which are on their turn passed to the `vfprintf` function. `myfile.txt` would contain:

myfile.txt
`Call with 1 variable argument.` `Call with 2 variable arguments.`

15

*int vprintf (const char * formato, va_list arg)*

Graba datos formateados desde una lista de argumentos variable a stdout.

Writes the C string pointed by *format* to the standard output (stdout), replacing any *format specifier* in the same way asprintf does, but using the elements in the variable argument list identified by *arg* instead of additional function arguments.

Internally, the function retrieves arguments from the list identified by *arg* as if va_arg was used on it, and thus the state of *arg* is likely altered by the call.

In any case, *arg* should have been initialized by va_start at some point before the call, and it is expected to be released byva_end at some point after the call.

Parameters

`format`

C string that contains a format string that follows the same specifications as *format* in printf (see printf for details).

`arg`

A value identifying a variable arguments list initialized with va_start. va_list is a special type defined in <cstdarg>.

Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the *error indicator* (ferror) is set and a negative number is

returned.

If a multibyte character encoding error occurs while writing wide characters, errno is set to EILSEQ and a negative number is returned.

Example

```
/* vprintf example */
#include <stdio.h>
#include <stdarg.h>

void WriteFormatted ( const char * format, ... )
{
  va_list args;
  va_start (args, format);
  vprintf (format, args);
  va_end (args);
}

int main ()
{
   WriteFormatted ("Call with %d variable argument.\n",1);
   WriteFormatted ("Call with %d variable
%s.\n",2,"arguments");

   return 0;
}
```

The example illustrates how the WriteFormatted can be called with a different number of arguments, which are on their turn passed to the vprintf function, showing the following output:

```
Call with 1 variable argument.
Call with 2 variable arguments.
```

16 *int vsprintf (char * s, const char * formato, va_list arg)*

Graba datos formateados de una lista de argumentos variable de cadena.

Composes a string with the same text that would be printed if *format* was used on printf, but using the elements in the variable argument list identified by *arg* instead of additional function arguments and storing the resulting content as a *C string* in the buffer pointed by *s*.

Internally, the function retrieves arguments from the list identified by *arg* as if va_arg was used on it, and thus the state of *arg* is likely to be altered by the call.

In any case, *arg* should have been initialized by va_start at some point before the call, and it is expected to be released byva_end at some point after the call.

Parameters

s

Pointer to a buffer where the resulting C-string is stored.

The buffer should be large enough to contain the resulting string.

`format`

C string that contains a format string that follows the same specifications as *format* in printf (see printf for details).

`arg`

A value identifying a variable arguments list initialized with va_start. va_list is a special type defined in <cstdarg>.

Return Value

On success, the total number of characters written is returned.
On failure, a negative number is returned.

Example

```
1  /* vsprintf example */
2  #include <stdio.h>
3  #include <stdarg.h>
4
5  void PrintFError ( const char * format, ... )
6  {
7    char buffer[256];
8    va_list args;
9    va_start (args, format);
10   vsprintf (buffer,format, args);
11   perror (buffer);
12   va_end (args);
13 }
14
15 int main ()
16 {
17   FILE * pFile;
18   char szFileName[]="myfile.txt";
19
20   pFile = fopen (szFileName,"r");
21   if (pFile == NULL)
22     PrintFError ("Error opening '%s'",szFileName);
23   else
24   {
25     // file successfully open
26     fclose (pFile);
27   }
28   return 0;
29 }
```

In this example, if the file `myfile.txt` does not exist, perror is called to show an error message similar to:

```
Error opening file 'myfile.txt': No such file or directory
```

17 *int fgetc (FILE * stream)*
Obtiene carácter desde el stream.

Returns the character currently pointed by the internal file position indicator of the

specified *stream*. The internal file position indicator is then advanced to the next character.

If the stream is at the end-of-file when called, the function returns EOF and sets the *end-of-file indicator* for the stream (feof).

If a read error occurs, the function returns EOF and sets the *error indicator* for the stream (ferror).

`fgetc` and getc are equivalent, except that getc may be implemented as a macro in some libraries.

Parameters

`stream`

Pointer to a FILE object that identifies an input stream.

Return Value

On success, the character read is returned (promoted to an `int` value).
The return type is `int` to accommodate for the special value EOF, which indicates failure:
If the position indicator was at the *end-of-file*, the function returns EOF and sets the *eof indicator* (feof) of *stream*.
If some other reading error happens, the function also returns EOF, but sets its *error indicator* (ferror) instead.

Example

```
/* fgetc example: money counter */
#include <stdio.h>
int main () {
  FILE * pFile;
  int c;
  int n = 0;

  pFile = fopen ("myfile.txt","r");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    do {
      c = fgetc (pFile);
      if (c == '$') n++;
    } while (c != EOF);
    fclose (pFile);
    printf ("The file contains %d dollar sign characters
($).\n",n);
  }
  return 0;
}
```

This program reads an existing file called `myfile.txt` character by character and uses the `n` variable to count how many dollar characters (`$`) the file contains.

18	*char * fgets (char * str, int num, FILE * stream)* Obtiene una cadena desde el stream. Reads characters from *stream* and stores them as a C string into *str* until (*num*-1) characters have been read or either a newline or the *end-of-file* is reached, whichever happens first. A newline character makes `fgets` stop reading, but it is considered a valid character by the function and included in the string copied to *str*. A terminating null character is automatically appended after the characters copied to *str*. Notice that `fgets` is quite different from gets: not only `fgets` accepts a *stream* argument, but also allows to specify the maximum size of *str* and includes in the string any ending newline character. Parameters `str` Pointer to an array of `char`s where the string read is copied. `num` Maximum number of characters to be copied into *str* (including the terminating null-character). `stream` Pointer to a FILE object that identifies an input stream. stdin can be used as argument to read from the *standard input*. Return Value On success, the function returns *str*. If the *end-of-file* is encountered while attempting to read a character, the *eof indicator* is set (feof). If this happens before any characters could be read, the pointer returned is a null pointer (and the contents of *str* remain unchanged). If a read error occurs, the *error indicator* (ferror) is set and a null pointer is also returned (but the contents pointed by *str*may have changed). Example `/* fgets example */` `#include <stdio.h>` `int main()` `{` `  FILE * pFile;` `  char mystring [100];` `  pFile = fopen ("myfile.txt" , "r");`

```
10    if (pFile == NULL) perror ("Error opening file");
11    else {
12      if ( fgets (mystring , 100 , pFile) != NULL )
13        puts (mystring);
14      fclose (pFile);
15    }
16    return 0;
17 }
```

This example reads the first line of `myfile.txt` or the first 99 characters, whichever comes first, and prints them on the screen.

19 *int fputc (int car, FILE * stream)*

Graba un carácter al stream.

Writes a *character* to the *stream* and advances the position indicator.

The character is written at the position indicated by the *internal position indicator* of the *stream*, which is then automatically advanced by one.

Parameters

`character`

The `int` promotion of the character to be written.
The value is internally converted to an `unsigned char` when written.

`stream`

Pointer to a FILE object that identifies an output stream.

Return Value

On success, the *character* written is returned.
If a writing error occurs, EOF is returned and the *error indicator* (ferror) is set.

Example

```
1  /* fputc example: alphabet writer */
2  #include <stdio.h>
3
4  int main ()
5  {
6    FILE * pFile;
7    char c;
8
9    pFile = fopen ("alphabet.txt","w");
10   if (pFile!=NULL) {
11
12     for (c = 'A' ; c <= 'Z' ; c++)
13       fputc ( c , pFile );
14
15     fclose (pFile);
16   }
17   return 0;
18 }
```

This program creates a file called `alphabet.txt` and
writes `ABCDEFGHIJKLMNOPQRSTUVWXYZ` to it.

20 *int fputs (const char * str, FILE * stream)*
Graba una cadena al stream.

Writes the *C string* pointed by *str* to the *stream*.

The function begins copying from the address specified (*str*) until it reaches the terminating null character (`'\0'`). This terminating null-character is not copied to the stream.

Notice that `fputs` not only differs from puts in that the destination *stream* can be specified, but also `fputs` does not write additional characters, while puts appends a newline character at the end automatically.

Parameters

`str`

C string with the content to be written to *stream*.

`stream`

Pointer to a FILE object that identifies an output stream.

Return Value

On success, a non-negative value is returned.
On error, the function returns EOF and sets the *error indicator* (ferror).

Example

```
1  /* fputs example */
2  #include <stdio.h>
3
4  int main () {
5     FILE * pFile;
6     char sentence [256];
7
8     printf ("Enter sentence to append: ");
9     fgets (sentence,256,stdin);
10    pFile = fopen ("mylog.txt","a");
11    fputs (sentence,pFile);
12    fclose (pFile);
13    return 0;
14 }
15
```

This program allows to append a line to a file called mylog.txt each time it is run.

21 *int getc (FILE * stream)*

Obtiene un carácter desde el stream.

Returns the character currently pointed by the internal file position indicator of the specified *stream*. The internal file position indicator is then advanced to the next character.

If the stream is at the end-of-file when called, the function returns EOF and sets the *end-of-file indicator* for the stream (feof).

If a read error occurs, the function returns EOF and sets the *error indicator* for the stream (ferror).

`getc` and fgetc are equivalent, except that `getc` may be implemented as a macro in some libraries. See getchar for a similar function that reads directly from stdin.

Parameters

`stream`

Pointer to a FILE object that identifies an input stream.
Because some libraries may implement this function as a macro, and this may evaluate the *stream* expression more than once, this should be an expression without side effects.

Return Value

On success, the character read is returned (promoted to an `int` value).
The return type is `int` to accommodate for the special value EOF, which indicates failure:
If the position indicator was at the *end-of-file*, the function returns EOF and sets the *eof indicator* (feof) of *stream*.
If some other reading error happens, the function also returns EOF, but sets its *error indicator* (ferror) instead.

Example

```
1  /* getc example: money counter */
2  #include <stdio.h>
3  int main ()
4  {
5    FILE * pFile;
6    int c;
7    int n = 0;
8    pFile=fopen ("myfile.txt","r");
9    if (pFile==NULL) perror ("Error opening file");
10   else
11   {
12     do {
13       c = getc (pFile);
14       if (c == '$') n++;
15     } while (c != EOF);
16     fclose (pFile);
17     printf ("File contains %d$.\n",n);
```

```
18    }
19    return 0;
20  }
```

This program reads an existing file called `myfile.txt` character by character and uses the `n` variable to count how many dollar characters (`$`) does the file contain.

22 *int getchar (void)*

Obtiene un carácter desde stdin.

Returns the next character from the standard input (stdin).

It is equivalent to calling getc with stdin as argument.

Parameters

(none)

Return Value

On success, the character read is returned (promoted to an `int` value).
The return type is `int` to accommodate for the special value EOF, which indicates failure:
If the standard input was at the *end-of-file*, the function returns EOF and sets the *eof indicator* (feof) of stdin.
If some other reading error happens, the function also returns EOF, but sets its *error indicator* (ferror) instead.

Example

```
/* getchar example : typewriter */
#include <stdio.h>

int main () {
  int c;
  puts ("Enter text. Include a dot ('.') in a sentence to
exit:");
  do {
    c = getchar();
    putchar (c);
  } while (c != '.');
  return 0;
}
```

A simple typewriter. Every sentence is echoed once ENTER has been pressed until a dot (.) is included in the text.

23 *char * gets (char * str)*

Obtiene una cadena desde stdin.

Reads characters from the *standard input* (stdin) and stores them as a C string into *str* until a newline character or the*end-of-file* is reached.

The newline character, if found, is not copied into *str*.

A terminating null character is automatically appended after the characters copied to *str*.

Notice that `gets` is quite different from fgets: not only `gets` uses stdin as source, but it does not include the ending newline character in the resulting string and does not allow to specify a maximum size for *str* (which can lead to buffer overflows).

Parameters

`str`

Pointer to a block of memory (array of `char`) where the string read is copied as a *C string*.

Return Value

On success, the function returns *str*.
If the *end-of-file* is encountered while attempting to read a character, the *eof indicator* is set (feof). If this happens before any characters could be read, the pointer returned is a null pointer (and the contents of *str* remain unchanged).
If a read error occurs, the *error indicator* (ferror) is set and a null pointer is also returned (but the contents pointed by *str*may have changed).

Compatibility

The most recent revision of the C standard (2011) has definitively removed this function from its specification.
The function is deprecated in C++ (as of 2011 standard, which follows C99+TC3).

Example

```
1  /* gets example */
2  #include <stdio.h>
3
4  int main() {
5    char string [256];
6    printf ("Insert your full address: ");
7    gets (string);      // warning: unsafe (see fgets instead)
8    printf ("Your address is: %s\n",string);
9    return 0;
10 }
11
```

24 *int putc (int car, FILE * stream)*
Graba un carácter al stream.

The character is written at the position indicated by the *internal position indicator* of the *stream*, which is then automatically advanced by one.

`putc` and fputc are equivalent, except that `putc` may be implemented as a macro in

some libraries. See putchar for a similar function that writes directly to stdout.

Parameters

`character`

The `int` promotion of the character to be written.
The value is internally converted to an `unsigned char` when written.
Because some libraries may implement this function as a macro, and this may evaluate the *stream* expression more than once, this should be an expression without side effects.

`stream`

Pointer to a FILE object that identifies an output stream.

Return Value

On success, the *character* written is returned.
If a writing error occurs, EOF is returned and the *error indicator* (ferror) is set.

Example

```
/* putc example: alphabet writer */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  char c;

  pFile=fopen("alphabet.txt","wt");
  for (c = 'A' ; c <= 'Z' ; c++) {
    putc (c , pFile);
    }
  fclose (pFile);
  return 0;
}
```

This example program creates a file called `alphabet.txt` and writes ABCDEFGHIJKLMNOPQRSTUVWXYZ to it.

25 *int putchar (int character)*
Graba un carácter al stdout.

Writes a *character* to the *standard output* (stdout).

It is equivalent to calling putc with stdout as second argument.

Parameters

`character`

The `int` promotion of the character to be written.
The value is internally converted to an `unsigned char` when written.

Return Value

On success, the *character* written is returned.
If a writing error occurs, EOF is returned and the *error indicator* (ferror) is set.

Example

```
/* putchar example: printing the alphabet */
#include <stdio.h>

int main () {
  char c;
  for (c = 'A' ; c <= 'Z' ; c++)
    putchar (c);

  return 0;
}
```

This program writes `ABCDEFGHIJKLMNOPQRSTUVWXYZ` to the standard output.

26 *int puts (const char * str)*
Graba una cadena al stdout.

The function begins copying from the address specified (*str*) until it reaches the terminating null character (`'\0'`). This terminating null-character is not copied to the stream.

Notice that `puts` not only differs from fputs in that it uses stdout as destination, but it also appends a newline character at the end automatically (which fputs does not).

Parameters

`str`
C string to be printed.

Return Value

On success, a non-negative value is returned.
On error, the function returns EOF and sets the *error indicator* (ferror).

Example

```
/* puts example : hello world! */
#include <stdio.h>

int main ()
{
  char string [] = "Hello world!";
  puts (string);
}
```

27 *int ungetc (int car, FILE * stream)*
Devuelve un carácter desde el stream.

A *character* is virtually put back into an input *stream*, decreasing its *internal file position* as if a previous getc operation was undone.

This *character* may or may not be the one read from the *stream* in the preceding input operation. In any case, the next character retrieved from *stream* is the *character* passed to this function, independently of the original one.

Notice though, that this only affects further input operations on that *stream*, and not the content of the physical file associated with it, which is not modified by any calls to this function.

Some library implementations may support this function to be called multiple times, making the characters available in the reverse order in which they were *put back*. Although this behavior has no standard portability guarantees, and further calls may simply fail after any number of calls beyond the first.

If successful, the function clears the *end-of-file indicator* of *stream* (if it was currently set), and decrements its internal*file position indicator* if it operates in binary mode; In text mode, the *position indicator* has unspecified value until all characters put back with `ungetc` have been read or discarded.

A call to fseek, fsetpos or rewind on *stream* will discard any characters previously put back into it with this function.

If the argument passed for the *character* parameter is EOF, the operation fails and the input *stream* remains unchanged.

Parameters

`character`

The `int` promotion of the character to be put back.
The value is internally converted to an `unsigned char` when put back.

`stream`

Pointer to a FILE object that identifies an input stream.

Return Value

On success, the *character* put back is returned.
If the operation fails, EOF is returned.

Example

```
1 /* ungetc example */
2 #include <stdio.h>
3
4 int main () {
```

```
  FILE * pFile;
  int c;
  char buffer [256];

  pFile = fopen ("myfile.txt","rt");
  if (pFile==NULL)
    perror ("Error opening file");
  else
    while (!feof (pFile)) {
      c=getc (pFile);
      if (c == EOF)
        break;
      if (c == '#')
        ungetc ('@',pFile);
      else
        ungetc (c,pFile);
      if (fgets (buffer,255,pFile) != NULL)
        fputs (buffer,stdout);
      else
        break;
  }
  return 0;
}
```

This example opens an existing file called `myfile.txt` for reading and prints its lines, but first gets the first character of every line and puts it back into the stream replacing any starting `#` by an `@`.

28 *size_t fread (void * ptr, size_t size, size_t count, FILE * stream)*

Lee un bloque de datos desde el stream.

Reads an array of *count* elements, each one with a size of *size* bytes, from the *stream* and stores them in the block of memory specified by *ptr*.

The position indicator of the stream is advanced by the total amount of bytes read.

The total amount of bytes read if successful is `(size*count)`.

Parameters

`ptr`

Pointer to a block of memory with a size of at least `(size*count)` bytes, converted to a `void*`.

`size`

Size, in bytes, of each element to be read.
size_t is an unsigned integral type.

`count`

Number of elements, each one with a size of *size* bytes.
size_t is an unsigned integral type.

`stream`

Pointer to a FILE object that specifies an input stream.

Return Value

The total number of elements successfully read is returned.
If this number differs from the *count* parameter, either a reading error occurred or the *end-of-file* was reached while reading. In both cases, the proper indicator is set, which can be checked with ferror and feof, respectively.
If either *size* or *count* is zero, the function returns zero and both the stream state and the content pointed by *ptr* remain unchanged.
size_t is an unsigned integral type.

Example

```
/* fread example: read an entire file */
#include <stdio.h>
#include <stdlib.h>

int main () {
  FILE * pFile;
  long lSize;
  char * buffer;
  size_t result;

  pFile = fopen ( "myfile.bin" , "rb" );
  if (pFile==NULL) {
    fputs ("File error",stderr); exit (1);
  }
  // obtain file size:
  fseek (pFile , 0 , SEEK_END);
  lSize = ftell (pFile);
  rewind (pFile);
  // allocate memory to contain the whole file:
  buffer = (char*) malloc (sizeof(char)*lSize);
  if (buffer == NULL) {
    fputs ("Memory error",stderr); exit (2);
  }
  // copy the file into the buffer:
  result = fread (buffer,1,lSize,pFile);
  if (result != lSize) {
    fputs ("Reading error",stderr); exit (3);
  }
/* the whole file is now loaded in the memory buffer. */
  // terminate
  fclose (pFile);
  free (buffer);
  return 0;
}
```

This code loads `myfile.bin` into a dynamically allocated memory buffer, which can be used to manipulate the content of a file as an array.

29 *size_t fwrite (const void * ptr, size_t size, size_t count, FILE * stream)*
Graba un bloque de datos al stream.

Writes an array of *count* elements, each one with a size of *size* bytes, from the block of memory pointed by *ptr* to the current position in the *stream*.

The *position indicator* of the stream is advanced by the total number of bytes written.

Internally, the function interprets the block pointed by *ptr* as if it was an array of `(size*count)` elements of type `unsigned char`, and writes them sequentially to *stream* as if fputc was called for each byte.

Parameters

`ptr`
Pointer to the array of elements to be written, converted to a `const void*`.

`size`
Size in bytes of each element to be written.
size_t is an unsigned integral type.

`count`
Number of elements, each one with a size of *size* bytes.
size_t is an unsigned integral type.

`stream`
Pointer to a FILE object that specifies an output stream.

Return Value

The total number of elements successfully written is returned.
If this number differs from the *count* parameter, a writing error prevented the function from completing. In this case, the*error indicator* (ferror) will be set for the *stream*.
If either *size* or *count* is zero, the function returns zero and the *error indicator* remains unchanged.
size_t is an unsigned integral type.

Example

```
1  /* fwrite example : write buffer */
2  #include <stdio.h>
3
4  int main ()
5  {
6    FILE * pFile;
7    char buffer[] = { 'x' , 'y' , 'z' };
8    pFile = fopen ("myfile.bin", "wb");
9    fwrite (buffer , sizeof(char), sizeof(buffer), pFile);
10   fclose (pFile);
11   return 0;
12 }
```

A file called `myfile.bin` is created and the content of the buffer is stored into it. For simplicity, the buffer contains `char`elements but it can contain any other type. `sizeof(buffer)` is the length of the array in bytes (in this case it is three, because the

	array has three elements of one byte each).
30	*int fgetpos (FILE * stream, fpos_t * pos)* Obtiene la posición actual del stream. Retrieves the current position in the *stream*. The function fills the fpos_t object pointed by *pos* with the information needed from the *stream*'s *position indicator* to restore the *stream* to its current position (and multibyte state, if *wide-oriented*) with a call to fsetpos. The ftell function can be used to retrieve the current position in the *stream* as an integer value. ## Parameters `stream` Pointer to a FILE object that identifies the stream. `pos` Pointer to a fpos_t object. This should point to an object already allocated. ## Return Value On success, the function returns zero. In case of error, errno is set to a platform-specific positive value and the function returns a non-zero value. ## Example ``` /* fgetpos example */ #include <stdio.h> int main () { FILE * pFile; int c; int n; fpos_t pos; pFile = fopen ("myfile.txt","r"); if (pFile==NULL) perror ("Error opening file"); else { c = fgetc (pFile); printf ("1st character is %c\n",c); fgetpos (pFile,&pos); for (n=0;n<3;n++) { fsetpos (pFile,&pos); c = fgetc (pFile); printf ("2nd character is %c\n",c); } fclose (pFile); } return 0; ```

```
}
```

Possible output (with `myfile.txt` containing `ABC`):

```
1st character is A
2nd character is B
2nd character is B
2nd character is B
```

The example opens `myfile.txt`, then reads the first character once, and then reads 3 times the same second character.

31 *int fseek (FILE * stream, long int offset, int origen)*
Reposiciona el indicador de posición al stream.

For streams open in binary mode, the new position is defined by adding *offset* to a reference position specified by *origin*.

For streams open in text mode, *offset* shall either be zero or a value returned by a previous call to ftell, and *origin* shall necessarily be `SEEK_SET`.

If the function is called with other values for these arguments, support depends on the particular system and library implementation (non-portable).

The *end-of-file internal indicator* of the *stream* is cleared after a successful call to this function, and all effects from previous calls to ungetc on this *stream* are dropped.

On streams open for update (read+write), a call to `fseek` allows to switch between reading and writing.

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

`offset`

Binary files: Number of bytes to offset from *origin*.
Text files: Either zero, or a value returned by ftell.

`origin`

Position used as reference for the *offset*. It is specified by one of the following constants defined in <cstdio>exclusively to be used as arguments for this function:

Constant	Reference position
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer

	SEEK_END End of file * * Library implementations are allowed to not meaningfully support SEEK_END (therefore, code using it has no real standard portability). ## Return Value If successful, the function returns zero. Otherwise, it returns non-zero value. If a read or write error occurs, the *error indicator* (ferror) is set. ## Example ``` /* fseek example */ #include <stdio.h> int main () { FILE * pFile; pFile = fopen ("example.txt" , "wb"); fputs ("This is an apple." , pFile); fseek (pFile , 9 , SEEK_SET); fputs (" sam" , pFile); fclose (pFile); return 0; } ``` After this code is successfully executed, the file example.txt contains: ``` This is a sample ```
32	*int fsetpos (FILE * stream, const fpos_t * pos)* Pone el indicador de posición al stream. Restores the current position in the *stream* to *pos*. The *internal file position indicator* associated with *stream* is set to the position represented by *pos*, which is a pointer to an fpos_t object whose value shall have been previously obtained by a call to fgetpos. The *end-of-file internal indicator* of the *stream* is cleared after a successful call to this function, and all effects from previous calls to ungetc on this *stream* are dropped. On streams open for update (read+write), a call to fsetpos allows to switch between reading and writing. A similar function, fseek, can be used to set arbitrary positions on streams open in binary mode. ## Parameters ``` stream ```

Pointer to a FILE object that identifies the stream.

`position`

Pointer to a fpos_t object containing a position previously obtained with fgetpos.

Return Value

If successful, the function returns zero.
On failure, a non-zero value is returned and errno is set to a system-specific positive value.

Example

```
1  /* fsetpos example */
2  #include <stdio.h>
3
4  int main ()
5  {
6    FILE * pFile;
7    fpos_t position;
8
9    pFile = fopen ("myfile.txt","w");
10   fgetpos (pFile, &position);
11   fputs ("That is a sample",pFile);
12   fsetpos (pFile, &position);
13   fputs ("This",pFile);
14   fclose (pFile);
15   return 0;
16 }
```

After this code is successfully executed, a file called `myfile.txt` will contain:

```
This is a sample
```

33 *long int ftell (FILE * stream)*

Obtiene la posición actual del stream.

Returns the current value of the position indicator of the *stream*.

For binary streams, this is the number of bytes from the beginning of the file.

For text streams, the numerical value may not be meaningful but can still be used to restore the position to the same position later using fseek (if there are characters put back using ungetc still pending of being read, the behavior is undefined).

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

Return Value

On success, the current value of the position indicator is returned.
On failure, `-1L` is returned, and errno is set to a system-specific positive value.

Example

```
/* ftell example : getting size of a file */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  long size;

  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    fseek (pFile, 0, SEEK_END);   // non-portable
    size = ftell (pFile);
    fclose (pFile);
    printf ("Size of myfile.txt: %ld bytes.\n",size);
  }
  return 0;
}
```

This program prints out the size of `myfile.txt` in bytes (where supported).

34 *void rewind (FILE * stream)*

Pone la posición del stream al inicio.

Sets the position indicator associated with *stream* to the beginning of the file.

The *end-of-file* and *error* internal indicators associated to the *stream* are cleared after a successful call to this function, and all effects from previous calls to ungetc on this *stream* are dropped.

On streams open for update (read+write), a call to `rewind` allows to switch between reading and writing.

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

Return Value

none

Example

```
/* rewind example */
#include <stdio.h>

int main ()
{
  int n;
```

```
  FILE * pFile;
  char buffer [27];

  pFile = fopen ("myfile.txt","w+");
  for ( n='A' ; n<='Z' ; n++)
    fputc ( n, pFile);
  rewind (pFile);
  fread (buffer,1,26,pFile);
  fclose (pFile);
  buffer[26]='\0';
  puts (buffer);
  return 0;
}
```

A file called `myfile.txt` is created for reading and writing and filled with the alphabet. The file is then rewinded, read and its content is stored in a buffer, that then is written to the standard output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

35

*void clearerr (FILE * stream)*
Limpia los indicadores de error.

Resets both the *error* and the *eof* indicators of the *stream*.

When a i/o function fails either because of an error or because the end of the file has been reached, one of these internal indicators may be set for the *stream*. The state of these indicators is cleared by a call to this function, or by a call to any of: rewind, fseek, fsetpos and freopen.

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

Return Value

None

Example

```
/* writing errors */
#include <stdio.h>
int main () {
  FILE * pFile;

  pFile = fopen("myfile.txt","r");
  if (pFile==NULL)
    perror ("Error opening file");
  else {
    fputc ('x',pFile);
    if (ferror (pFile)) {
      printf ("Error Writing to myfile.txt\n");
      clearerr (pFile);
```

```
    }
    fgetc (pFile);
    if (!ferror (pFile))
      printf ("No errors reading myfile.txt\n");
    fclose (pFile);
  }
  return 0;
}
```

This program opens an existing file called `myfile.txt` for reading and causes an I/O error trying to write on it. That error is cleared using `clearerr`, so a second error checking returns false.

Salida:

```
Error Writing to myfile.txt
No errors reading myfile.txt
```

36

*int feof (FILE * stream)*

Verifica el final del archivo.

Checks whether the *end-of-File indicator* associated with *stream* is set, returning a value different from zero if it is.

This indicator is generally set by a previous operation on the *stream* that attempted to read at or past the end-of-file.

Notice that *stream*'s internal position indicator may point to the *end-of-file* for the next operation, but still, the *end-of-file*indicator may not be set until an operation attempts to read at that point.

This indicator is cleared by a call to clearerr, rewind, fseek, fsetpos or freopen. Although if the *position indicator* is not repositioned by such a call, the next i/o operation is likely to set the indicator again.

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

Return Value

A non-zero value is returned in the case that the *end-of-file indicator* associated with the stream is set.
Otherwise, zero is returned.

Example

```
/* feof example: byte counter */
#include <stdio.h>

```

```
4  int main () {
5    FILE * pFile;
6    int n = 0;
7
8    pFile = fopen ("myfile.txt","rb");
9    if (pFile==NULL)
10     perror ("Error opening file");
11   else {
12     while (fgetc(pFile) != EOF) {
13       ++n;
14     }
15     if (feof(pFile)) {
16       puts ("End-of-File reached.");
17       printf ("Total number of bytes read: %d\n", n);
18     }
19     else
20       puts ("End-of-File was not reached.");
21     fclose (pFile);
22   }
23   return 0;
}
```

This code opens the file called `myfile.txt`, and counts the number of characters that it contains by reading all of them one by one. The program checks whether the end-of-file was reached, and if so, prints the total number of bytes read.

37 *int ferror (FILE * stream)*

Verifica el indicador de error.

Checks if the *error indicator* associated with *stream* is set, returning a value different from zero if it is.

This indicator is generally set by a previous operation on the *stream* that failed, and is cleared by a call to clearerr, rewindor freopen.

Parameters

`stream`

Pointer to a FILE object that identifies the stream.

Return Value

A non-zero value is returned in the case that the *error indicator* associated with the stream is set.
Otherwise, zero is returned.

Example

```
1  /* ferror example: writing error */
2  #include <stdio.h>
3  int main ()
4  {
5    FILE * pFile;
```

```
  pFile=fopen("myfile.txt","r");
  if (pFile==NULL) perror ("Error opening file");
  else {
    fputc ('x',pFile);
    if (ferror (pFile))
      printf ("Error Writing to myfile.txt\n");
    fclose (pFile);
  }
  return 0;
}
```

This program opens an existing file called `myfile.txt` in read-only mode but tries to write a character to it, generating an error that is detected by `ferror`.

Salida:

```
Error Writing to myfile.txt
```

38 *void perror (const char * str)*
Imprime un mensaje de error.

Interprets the value of errno as an error message, and prints it to stderr (the standard error output stream, usually the console), optionally preceding it with the custom message specified in *str*.

errno is an integral variable whose value describes the error condition or diagnostic information produced by a call to a library function (any function of the C standard library may set a value for errno, even if not explicitly specified in this reference, and even if no error happened), see errno for more info.

The error message produced by `perror` is platform-depend.

If the parameter *str* is not a null pointer, *str* is printed followed by a colon (:) and a space. Then, whether *str* was a null pointer or not, the generated error description is printed followed by a newline character (`'\n'`).

`perror` should be called right after the error was produced, otherwise it can be overwritten by calls to other functions.

Parameters.

`str`

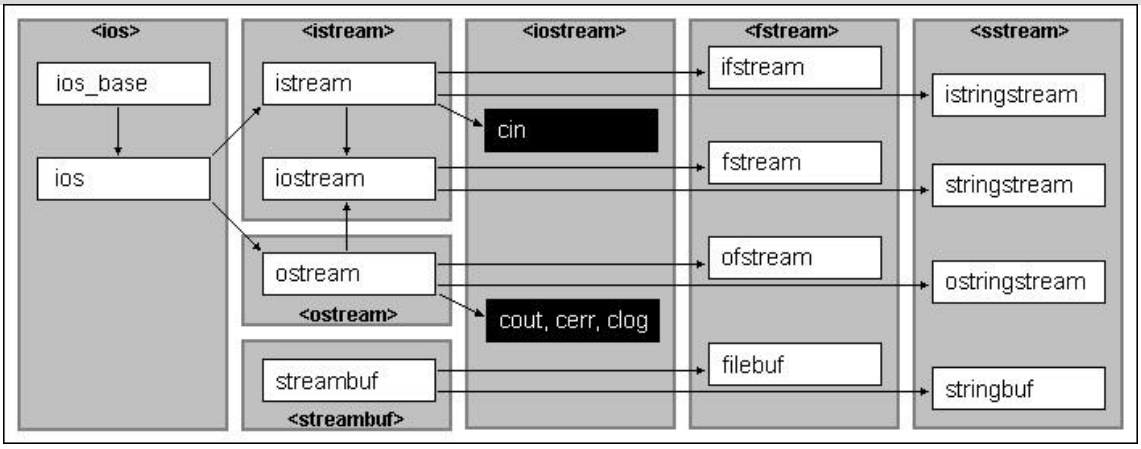
C string containing a custom message to be printed before the error message itself.
If it is a null pointer, no preceding custom message is printed, but the error message is still printed.
By convention, the name of the application itself is generally used as parameter.

Return Value

	none Example `/* perror example */` `#include <stdio.h>` `int main ()` `{` `  FILE * pFile;` `  pFile=fopen ("unexist.ent","rb");` `  if (pFile==NULL)` `  perror ("The following error occurred");` `  else` `  fclose (pFile);` `  return 0;` `}` If the file `unexist.ent` does not exist, something similar to this may be expected as program output: `The following error occurred: No such file or directory`
39	UFSIZ Tamaño del buffer.
40	EOF Macro que indica Fin del Archivo, generalmente representado por el valor -1. It is a macro definition of type `int` that expands into a negative integral constant expression (generally, `-1`). It is used as the value returned by several functions in header <cstdio> to indicate that the End-of-File has been reached or to signal some other failure conditions. It is also used as the value to represent an invalid character. In C++, this macro corresponds to the value of `char_traits<char>::eof()`.
41	FILENE_MAX Máxima longitud del nombre de archivo.
42	FOPEN_MAX Cantidad máxima de archivos abiertos simultáneamente.
43	NULL Macro que indica Puntero nulo, establecido por un valor de cero.
44	SEEK_SET A partir del comienzo del archivo mueve el puntero.
45	SEEK_CUR A partir de la posición actual del puntero mueve el puntero hacia adelante si es positivo o hacia atrás si es negativo.
46	SEEK_END A partir del final del archivo mueve el puntero hacia adelante si es positivo o hacia atrás si es negativo.
47	FILE

	Objeto que contiene información para controlar un stream.
48	fpos_t Objeto que contiene información para indicar la posición dentro del archivo. This type of object is used to specify a position within a file. An object of this type is capable of specifying uniquely any position within a file. The information in `fpos_t` objects is usually filled by a call to fgetpos, which takes a pointer to an object of this type as argument. The content of an `fpos_t` object is not meant to be read directly, but only to be used as an argument in a call to fsetpos.
49	size_t Tipo entero sin signo. Alias of one of the fundamental unsigned integer types. It is a type able to represent the size of any object in bytes: `size_t` is the type returned by the `sizeof` operator and is widely used in the standard library to represent sizes and counts. In <cstdio>, it is used as the type of some parameters in the functions fread, fwrite and setvbuf, and in the case of freadand fwrite also as its returning type.

Archivos: STREAM



Stream o flujo de datos externo

Un **stream** o **flujo**, se puede definir como un dispositivo que produce información., el cual siempre está ligado a un dispositivo físico, como ser, pantalla, disco, impresora…, comportándose de manera análoga con total independencia del dispositivo en cuestión.

Todo programa que se ejecute en forma automática estarán disponibles los siguientes flujos:

stream	comentario
cin	Entrada estándar por teclado.
cout	Salida estándar por pantalla.
cerr	Salida estándar de mensajes de error .

Indicadores y Manipuladores

Manipuladores definidos en <iomanip.h>

	Manipuladores parametrizados de <iomanip.h>
1	*setbase(base)* base es un valor entre 1 y 36.
2	*setfill(car)*
3	*setprecision(cdig)* cdig es la cantidad de dígitos a mostrar en su parte decimal.
4	setw(ancho) ancho es la cantidad de lugares mínimo de formato.
5	setiosflag(flag) flag es una lista de 15 items a setear, ios::flag usa | para usar más de un flag. Ver lista de flags más abajo en Manipuladores_iomanip
6	resetiosflag(flag) Idem anterior.

Ejemplos de uso manipuladores iomanip

```
cout << setbase(16) << setfill('*') << setprecision(2);
cout << setiosflag(ios::rigth | ios::showpoint);
```

Manipuladores sin parámetros definidos en la clase ios

Los cambios son permanentes hasta que se produzca un nuevo cambio.

Ejemplos de uso de manipuladores sin parámetros

```
cout << hex << 123 << oct << 123 << endl;
cout.fill('*');
cout.width(20);
cout << right << "Hola" << endl;
```

Ord	Manipuladores	Manipuladores sin parámetros de la clase ios
1	dec	Cambia la base de numeración a decimal.
2	*hex*	Cambia la base de numeración a hexadecimal.
3	*oct*	Cambia la base de numeración a octal.
4	*ws*	Solo para streams de entrada.
5	*ends*	Agrega carácter nulo a una cadena.
6	*flush*	Vacía el buffer.
7	*endl*	Vacía el buffer y además cambia de línea.
8	*fixed*	Muestra reales en notación punto fijo.
9	*scientific*	Muestra reales en notación punto flotante o científica.
10	*boolalpha*	Emite false o true según corresponda en lugar de 0 o 1.
11	*left*	Alínea a izquierda.
12	*right*	Alínea a derecha.
13	*internal*	Inserta relleno internamente en un formato de salida.
14	*showbase*	Muestra 0 o 0x antes del valor en octal o si es hexadecimal.
15	*showpos*	Muestra el signo del valor siempre.
16	*showpoint*	Muestra punto decimal siempre.
17	*uppercase*	Convierte solo a mayúsculas la representación numérica.
18	*skipws*	Al ingresar datos desde un stream espacios iniciales los ignora.
19	*unitbuf*	Vacía todos los buffers.
20	*nouppercase*	Desactiva uppercase.
21	*noboolalpha*	Desactiva boolalpha.
22	*noshowbase*	Desactiva showbase.
23	*noshowpoint*	Desactiva showpoint.
24	*noshowpos*	Desactiva showpos.

25	*noskipws*	Desactiva skipws.
26	*nounitbuf*	Desactiva unitbuf.

Ejemplos de manipuladores sin parámetros de la clase ios

Codificación de manipuladores sin parámetros de la clase ios

```
/*
   Id.Programa: Manipuladores_sin_parametros.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: julio-2016
   Comentario.: Manipuladores sin parametros
                1. dec
                2. hex
                3. oct
                4. ws
                5. ends
                6. endl
                7. flush
                8. boolalpha
                9. left
               10. right
               11. fixed
               12. internal
               13. scientific
               14. showbase
               15. showpoint
               16. showpos
               17. skipws
               18. unitbuf
               19. uppercase
               20. noboolalpha
               21. noshowbase
               22. noshowpoint
               23. noshowpos
               24. noskipws
               25. nounitbuf
               26. nouppercase
*/

#include <iostream>
```

```
using namespace std;

int main() {
  bool existe = true;
  char a[21], b[21];

  cout << showbase << hex << 123 << endl;
  cout.width(10);
  cout << right << "Hola" << '*' << endl << boolalpha << existe << endl;
  cout.width(10);
  cout.fill('*');
  cout << showpos << dec << 123 << endl;
  cout << internal << -123 << endl;
  cout << uppercase << hex << 0x1ba9 << endl;
  cout << nouppercase << 0x1ba2 << endl;
  cin >> skipws >> a >> b;
  cout << a << '*' << endl << b << '*' << endl;
}

/* Salida
0x7b
     Hola*
true
******+123
-123
0X1BA9
0x1ba2
    Hola
       que tal?
Hola
que tal?
*/
```

Funciones miembro definidas en la clase ios

Su forma de uso presenta la siguiente notación

```
stream.función()
```

Máscaras de modificadores

Permiten trabajar con grupos de modificadores afines.

```
enum {
  basefield = dec+oct+hex,
  floatfield = scientific+fixed,
  adjustfield = left+right+internal
};
```

Funciones Miembro de la clase ios		
1	*stream.bad()*	Devuelve un valor distinto de cero si ha ocurrido un error. Sólo se comprueba el bit de estado ios::badbit, de modo que esta función no equivale a !good().
2	*stream.clear(estado=0)*	Sirve para modificar los bits de estado de un stream, normalmente para eliminar un estado de error. Se suelen usar constantes definidas en el enum io_state definido en ios, usando el operador de bits OR para modificar varios bits a la vez.
3	*filebuf* stream.close()*	Cierra el filebuf y el fichero asociados.
4	*stream.eof()*	Devuelve un valor distinto de cero si se ha alcanzado el fin de fichero. Esta función únicamente comprueba el bit de estado ios::eofbit.
5	*stream.fail()*	Devuelve un valor distinto de cero si una operación sobre el stream ha fallado. Comprueba los bits de estado ios::badbit y ios::failbit.
6	*stream.fill()* *stream.fill(car)*	Cambia el carácter de relleno que se usa cuando la salida es más ancha de la necesaria para el dato actual.
7	*stream.flags()* *stream.flags(long)*	Permite cambiar o leer los flags de manipulación de formato. La primera forma devuelve el valor actual de los flags. La segunda cambia el valor actual por valor, el valor de retorno es el valor previo de los flags.
8	*stream.flush()*	Vacía el buffer asociado al stream. Procesa todas las salidas pendientes.
9	*int stream.gcount()*	Devuelve el número de caracteres sin formato de la última lectura. Las lecturas sin formato son las realizadas mediante las funciones get, getline y read.

10	*int stream.get()* *stream.get(char *,int len,char = '\n')* *stream.get(char &)* *stream.get(streambuf &,char = 'n')*	La primera forma extrae el siguiente carácter o EOF si no hay disponible ninguno. La segunda forma extrae caracteres en la dirección proporcionada en el parámetro char* hasta que se recibe el delimitador del tercer parámetro, el fin de fichero o hasta que se leen len-1 bytes. Siempre se añade un carácter nulo de terminación en la cadena de salida. El delimitador no se extrae desde el stream de entrada. La función sólo falla si no se extraen caracteres. La tercera forma extrae un único carácter en la referencia a char proporcionada. La cuarta forma extrae caracteres en el streambuf especificado hasta que se encuentra el delimitador.
11	*stream.getline(char *,int,char = '\n')*	Extrae caracteres hasta que se encuentra el delimitador y los coloca en el buffer, elimina el delimitador del stream de entrada y no lo añade al buffer.
12	*stream.good()*	Devuelve un valor distinto de cero si no ha ocurrido ningún error, es decir, si ninguno de los bits de estado está activo. Aunque pudiera parecerlo, (good significa bueno y bad malo, en inglés) esta función no es exactamente equivalente a !bad(). En realidad es equivalente a rdstate() == 0.
13	*stream.ignore(int n=1,INT delim=EOF)*	Hace que los siguientes n caracteres en el stream de entrada sean ignorados; la extracción se detiene antes si se encuentra el delimitador delim. El delimitador también es extraído del stream.
14	*filebuf* stream.open(nomFis,modoAper)*	Abre un fichero para el objeto especificado. Para el parámetro mode se pueden usar los valores del enum open_mode definidos en la clase ios. open_mode –modo de apertura- enum open_mode { in, out, ate, app, trunc, nocreate, noreplace, binary };

15	*int stream.peek()*	Devuelve el siguiente carácter sin extraerlo del stream.
16	*stream.precision()* *stream.precision(car)*	Permite cambiar el número de caracteres significativos que se mostrarán cuando trabajemos con números en coma flotante: float o double. La primera forma devuelve el valor actual de la precisión, la segunda permite modificar la precisión para las siguientes salidas, y también devuelve el valor actual.
17	*stream.put(car)*	Inserta un carácter en el stream de salida.
18	*stream.putback(car)*	Devuelve un carácter al stream.
19	*stream.rdstate()*	Devuelve el estado del stream. Este estado puede ser una combinación de cualquiera de los bits de estado definidos en el enum ios::io_state, es decir ios::badbit, ios::eofbit, ios::failbit e ios::goodbit. El goodbit no es en realidad un bit, sino la ausencia de todos los demás. De modo que para verificar que el valor obtenido por rdstate es ios::goodbit tan sólo hay que comparar. En cualquier caso es mejor usar la función good(). En cuanto a los restantes bits de estado se puede usar el operador & para verificar la presencia de cada uno de los bits. Aunque de nuevo, es preferible usar las funciones bad(), eof() o fail().
20	*stream.read(char *,int)*	Extrae el número indicado de caracteres en el array char*. Se puede usar la función gcount() para saber el número de caracteres extraídos si ocurre algún error.
21	*stream.seekg(pos)* *stream.seekg(offset,dir)*	La primera forma se mueve a posición absoluta, tal como la proporciona la función tellg. La segunda forma se mueve un número offset de bytes la posición del cursor del stream relativa a dir. Este parámetro puede tomar los valores definidos en el enum seek_dir: {beg, cur, end}; Para streams de salida usar ostream::seekp. Usar seekpos o seekoff para moverse en un

		buffer de un stream.
22	*stream.seekp(dir)* *stream.seekp(dir,donde)*	La primera forma mueve el cursor del stream a una posición absoluta, tal como la devuelve la función tellp. La segunda forma mueve el cursor a una posición relativa desde el punto indicado mediante el parámetro donde -seek_dir-, que puede tomar los valores del enum seek_dir: beg, cur, end.
23	*stream.setf(long)* *stream.setf(valor,máscara)*	Permite modificar los flags de manipulación de formato. La primera forma activa los flags que estén activos tanto en el parámetro y deja sin cambios el resto. La segunda forma activa los flags que estén activos tanto en valor como en máscara y desactiva los que estén activos en mask, pero no en valor. Podemos considerar que mask contiene activos los flags que queremos modificar y valor los flags que queremos activar. Ambos devuelven el valor previo de los flags. Máscara enum { basefield = dec+oct+hex, floatfield = scientific+fixed, adjustfield = left+right+internal };
24	*stream.tellg()*	Devuelve la posición actual del stream.
25	*stream.tellp()*	Devuelve la posición absoluta del cursor del stream.
26	*stream.unsetf(máscara)*	Permite eliminar flags de manipulación de formato. Desactiva los flags que estén activos en el parámetro.
27	*stream.width()* *stream.width(entero)*	Cambia la anchura en caracteres de la siguiente salida de stream. La primera forma devuelve el valor de la anchura actual, la segunda permite cambiar la anchura para las siguientes salidas, y también devuelve el valor actual de la anchura.
28	*stream.write(const char *, int n)*	Inserta n caracteres (aunque sean nulos) en

		el stream de salida.

Ejemplos con funciones miembro de la clase ios
Codificación de funciones miembro de la clase ios

Indicadores de la clase ios

Se utilizan con las funciones miembro:

- **setf()**
- **unsetf()**
- **resetiosflag()**

y su formato es cout.setf(ios:: flag | ios::flag …)

Ord.	Indicadores (flag) de la clase ios	
1	boolalpha	Muestra false o true el reemplazo de 0 o 1.
2	dec	Muestra el valor entero en base decimal.
3	oct	Muestra el valor entero en base octal.
4	hex	Muestra el valor entero en base hexadecimal.
5	internal	Rellena con el carácter seleccionado internamente.
6	left	Alínea a la izquierda la próxima salida.
7	right	Alínea a la derecha la próxima salida.
8	scientific	Muestra el valor real en notación de punto flotante.
9	fixed	Muestra el valor real en notación de punto fijo.
10	showbase	Muestra la base numérica del valor entero.
11	showpoint	Muestra el punto decimal siempre.
12	showpos	Muestra el signo del número siempre.
13	skipws	Saltea o quita los espacios iniciales.
14	unitbuf	Vacía todos los buffers.
15	uppercase	Convierte a mayúscula base u valor entero.

Algunas funciones miembros <ios.h> y definidas en las clases istream, ostream, iostream

*cout.**width**(num)*
*cout.**width**()*
*cout.**precision**(num)*
*cout.**precision**()*
*cout.**fill**(car)*
*cout.**fill**()*

cout.width(n)

Obtiene (1) streamsize width() const;
Establece (2) streamsize width (streamsize wide);

La primer forma (1) retorna el valor actual del ancho del campo.

La segunda forma (2) también pone un nuevo ancho de campo al stream.

El ancho del campo determina el número mínimo de caracteres a ser escritos en una representación de salida. Si el ancho estándar de la representación es más corta que el ancho del campo, la representación es cubierto con caracteres de relleno en el lugar determinado por el formato de bandera adjustfield(uno de left, right o internal). El carácter de relleno puede ser establecido con la función miembro fill.

El ancho del campo también puede ser modificando utilizando el manipulador parametrizado *setw*().

Retorna: Valor del ancho del campo antes de la llamada.

```
// ancho del campo
#include <iostream>     // cout, left
using namespace std;

int main () {
  cout << 100 << '\n';
  cout.width(10);
  cout << 100 << '\n';
  cout.fill('x');
  cout.width(15);
  cout << left << 100 << '\n';
  return 0;
}
```

Emite

```
100
       100
100xxxxxxxxxxxx
```

cout.precision()

Obtener (1) streamsize precision() const;
Establecer (2) streamsize precision (streamsize prec);

La primer forma (1) retorna el valor actual del campo precisión punto flotante del stream.

La segunda forma (2) también establece el nuevo valor.

La precisión en punto flotante determina el número máximo de dígitos a ser escritos en operaciones de inserción.

La precisión decimal también puede ser modificada utilizando el manipulador parametrizado *setprecision*().

Retorna: La precisión indicada en el stream antes de la llamada.

```
// modify precision
#include <iostream>     // std::cout, std::ios
using namespace std;

int main () {
  double f = 3.14159;
  cout.unsetf ( ios::floatfield );              // floatfield not set
  cout.precision(5);
  cout << f << '\n';
  cout.precision(10);
  cout << f << '\n';
  cout.setf( ios::fixed,  ios::floatfield ); // floatfield set to fixed
  cout << f << '\n';
  return 0;
}
```

Emite

```
3.1416
3.14159
3.1415900000
```

cout.fill()

Obtener (1) `char fill() const;`
Establecer (2) `char fill (char fillch);`

La primer forma (1) retorna el carácter de relleno.

La segunda forma (2) establece el carácter de relleno como el nuevo carácter de relleno y retorna el carácter de relleno usado antes de la llamada.

También puede ser utilizado el manipulador parametrizado *setfill*().

Retorna: El valor del carácter de relleno antes de la llamada.

```
// using the fill character
#include <iostream>     // std::cout
using namesapace std;

int main () {
  char prev;
```

```
  cout.width (10);
  cout << 40 << '\n';
  prev = cout.fill ('x');
  cout.width (10);
  cout << 40 << '\n';
  cout.fill(prev);
  return 0;
}
```

Emite

```
        40
xxxxxxxx40
```

Indicadores y Manipuladores de Entrada / Salida

Pueden ser empleados en operaciones de entrada como así también de salida. Los manipuladores pueden tener o no argumentos, los que no tienen están definidos en **<iostream.h>**, mientras que los que sí tienen están definidos en **<iomanip.h>**. La dificultad de los manipuladores contra los indicadores es que no permiten guardar las configuraciones anteriores. Un manipulador solo afecta al stream al que se aplica (cin, cout, cerr). Además un manipulador permanece activo en el flujo correspondiente hasta que se cambia por otro manipulador, la excepción es setw().

Entrada/Salida con formato <iostream>

Se utilizan los siguientes métodos o funciones:

*cout.**setf()***
*cout.**unsetf()***
*cout.**flags()***

Cada stream tienen asociados unos **indicadores** que son constantes de tipo **enum** que se almacena en un tipo long los cuales, los bits pueden activarse o desactivarse, a continuación se presenta dichos indicadores, utilizados con las funciones métodos indicados anteriormente (stream.setf(), stream.unsetf(), stream.flags(), como por ejemplo, cout.setf(ios::x), en donde x son algunos de los indicadores indicados en la siguiente tabla:

Tabla de Indicadores con cout.setf(), cout.unsetf(), cout.flags()

Items	Indicadores valor	Comentario
	Indicadores:	Permite:

	cout.setf(ios::indic$_1$ | ios::indic$_2$…) cout.unsetf(ios::indic$_1$ | ios::indic$_2$…) cout.flags(ios::indic$_1$ | ios::indic$_2$…) El símbolo (|) barra vertical, lease o lógico, el cual permite indicar más de una bandera en una misma función miembro de la clase ios.		Activar banderas indicadas sin modificar las otras. Desactivar banderas indicadas sin modificar las otras. Resetear todo y activa banderas indicadas.
	Ejemplos de formas de uso cout.setf(ios::boolalpha | ios::fixed); cout.unsetf(ios::boolalpha); cout.flags(ios::left)		
1	skipws	0x0001	Se descartan blancos iniciales a la entrada.
2	left	0x0002	La salida se alinea a la izquierda.
3	right	0x0004	La salida se alinea a la derecha.
4	internal	0x0008	Se alinea el signo y los caracteres indicativos por la izquierda y los dígitos por la derecha.
5	dec	0x0010	Salida decimal para enteros por omisión.
6	oct	0x0020	Salida octal para enteros.
7	hex	0x0040	Salida hexadecimal para enteros.
8	showbase	0x0080	Se muestra la base de los valores numéricos.
9	showpoint	0x0100	Se muestra el punto decimal.
10	uppercase	0x0200	Los caracteres de formato aparecen en mayúsculas.
11	showpos	0x0400	Se muestra el signo + en los valores positivos.
12	scientific	0x0800	Notación científica para coma flotante.
13	fixed	0x1000	Notación normal para coma flotante.
14	unitbuf	0x2000	Salida sin buffer (se vuelca cada operación).
15	stdio	0x4000	Vacía los buffers de stdout y stderr luego de una inserción.

Ejemplos de indicadores

```
cout.setf(ios::left);
cout.setf(ios::fixed | ios::showpos | ios::showbase);
cout.setf(ios::skipws | ios::showpoint | ios::hex);
```

Además existen otros indicadores adicionales (adjustfield, basefield, floatfield) que actúan como combinaciones de las anteriores (máscaras), a saber:

Items	Indicadores compuestos	Combina excluyente (solo una en on)
1	adjustfield	left, rigth, internal
2	basefield	dec, oct, hex
3	floatfield	scientific, fixed

Por defecto todos los indicadores anteriores están desactivados, excepto skipws y dec.

Establecer (1) `fmtflags setf (fmtflags fmtfl);`
Máscara (2) `fmtflags setf (fmtflags fmtfl, fmtflags mask);`

La primer forma de *setf*(1) es generalmente utilizado para poner banderas de formato independiente: *boolalpha*, *showbase*, *showpoint*, *showpos*, *skipws*, *unitbuf* and *uppercase*, which can also be unset directly with member *unsetf*().

La segunda forma de *setf*(2) es generalmente utilizada para poner un valor para una de las banderas indicadas, utilizando uno de los campos de máscara de bit.

Retorna: El formato de bandera selecionada en el stream antes de la llamada.

Ejemplo

```
// modifying flags with setf/unsetf
#include <iostream>     // cout, ios
using namespace std;

int main () {
  cout.setf ( ios::hex, ios::basefield );  // set hex as the basefield
  cout.setf ( ios::showbase );             // activate showbase
  cout << 100 << '\n';
  cout.unsetf ( ios::showbase );           // deactivate showbase
  cout << 100 << '\n';
  return 0;
}
```

Emite

```
0x64
64
```

El formato para *flags*() es igual que para *setf*(). La diferencia es que *flags*() previamente inicializa en cero los bits, mientras que, *setf*(), solo modifica la configuración indicada, y mantiene todo lo que no se ha cambiado explícitamente, por lo que es una alternativa más segura.

Ejemplos adicionales

cout.setf(ios::showpos); // mostrar positivo, el signo +.
cout.setf(ios::showpoint | ios::fixed); // establece mostrar punto decimal y punto fijo.
cout << 100.0; // muestra +100.000000

Manipuladores sin argumentos

Se utilizan en stream como ser cout y como argumentos como si fueran expresiones, por ejemplo, cout << boolalpha << a << left << b << endl;

<table>
<tr><th>Ord</th><th>Indicadores <ios.h></th><th colspan="2">Establece</th></tr>
<tr><td></td><td colspan="2">Manipuladores:
Ejemplos de formas de uso
cout << boolalpha << existe << endl;
cout << left << setw(10) << “Hola” << ‘ ‘ << 123 << endl;</td><td>Permite:
Activar banderas indicadas sin modificar las otras.
Desactivar banderas indicadas sin modificar las otras.
Resetear todo y activa banderas indicadas.</td></tr>
<tr><td>1</td><td>boolalpha</td><td colspan="2">Valor false o true a cambio de 0, 1.
<pre>
1 // modify boolalpha flag
2 #include <iostream>   // scout,
3 boolalpha, noboolalpha
4 using namespace std;
5
6 int main () {
7   bool b = true;
8   cout << boolalpha << b << '\n';
9   cout << noboolalpha << b <<
'\n';
   return 0;
}
</pre></td></tr>
<tr><td>2</td><td>dec</td><td colspan="2">Valor numérico en decimal.
<pre>
#include <iostream>
using namespace std;

int main() {
   int a = 123, c = 432, b = 543;

   cout << "Decimal: " << dec
        << a << ", " << b
        << ", " << c << endl;
   cout << "Hexadecimal: " << hex
        << a << ", " << b
        << ", " << c << endl;
   cout << "Octal: " << oct
        << a << ", " << b
</pre></td></tr>
</table>

```
        << ", " << c << endl;

   return 0;
}
```

La salida tendrá éste aspecto:

```
Decimal:     123, 543, 432
Hexadecimal: 7b, 21f, 1b0
Octal:       173, 1037, 660
```

3	endl	Retorno de carro y vacia el buffer.
4	ends	Sirve para añadir el carácter nulo de fin de cadena.
5	fixed	Notación en punto fijo.

```
// example on insertion
#include <iostream>      // cout,
right, endl
#include <iomanip>       // setw
using namespace std;

int main () {
  int val = 65;

  cout << right;         // right-
adjusted (manipulator)
  cout << setw(10);      // set
width (extended manipulator)

  cout << val << std::endl; //
multiple insertions

  return 0;
}
```

Output:

```
        65
```

```
// modify precision
#include <iostream>      // cout,
ios
using namespace std;

int main () {
  double f = 3.14159;
  cout.unsetf ( ios::floatfield
);                 // floatfield
not set
```

```
  cout.precision(5);
  cout << f << '\n';
  cout.precision(10);
  cout << f << '\n';
  cout.setf(          ios::fixed,
ios::floatfield ); // floatfield
set to fixed
  cout << f << '\n';
  return 0;
}
```

Possible output:

```
3.1416
3.14159
3.1415900000
```

6	flush	Vacia el buffer.

```
// Flushing files
#include  <fstream>           //
std::ofstream
using namespace std;

int main () {

  ofstream outfile ("test.txt");

  for (int n=0; n<100; ++n) {
    outfile << n;
    outfile.flush();
  }
  outfile.close();

  return 0;
}
```

7	hex	Valor numérico en hexadecimal.

```
// modify flags
#include <iostream>      // cout,
ios
using namespace std;

int main () {
  cout.flags(ios::right|ios::hex
                | ios::showbase );
  cout.width (10);
  cout << 100 << '\n';
  return 0;
}
```

```
      0x64
```

8	internal	Relleno intermedio. `// modify adjustfield using manipulators` `#include <iostream> // cout, std::internal, left, right` `using namespace std;` `int main () {` `  int n = -77;` `  cout.width(6);` `  cout <<internal<< n << '\n';` `  cout.width(6); cout << left << n << '\n';` `  cout.width(6); cout << right << n << '\n';` `  return 0;` `}` Output: `-  77` `-77` `  -77`
9	left	Alineación a la izquierda `// modify adjustfield using manipulators` `#include <iostream> // cout, internal, left, right` `using namespace std;` `int main () {` `  int n = -77;` `  cout.width(6); cout << internal << n << '\n';` `  cout.width(6); cout << left << n << '\n';` `  cout.width(6); cout << right << n << '\n';` `  return 0;` `}`
10	noboolalpha	Desafecta false y true por 0 y 1. `// modify boolalpha flag` `#include <iostream> // cout, boolalpha, noboolalpha` `using namespace std;` `int main () {` `  bool b = true;`

```
  cout << boolalpha << b << '\n';
  cout << noboolalpha << b << '\n';
  return 0;
}
```

Salida:

```
true
1
```

11	noshowbase	Desactiva showbase, es decir, no muestra la base del número.

```
// modify showbase flag
#include <iostream>     //
std::cout, std::showbase,
std::noshowbase
using namespace std;

int main () {
  int n = 20;
  cout << hex << showbase << n
<< '\n';
  cout << hex << noshowbase << n
<< '\n';
  return 0;
}
```

Edit & Run

Output:

```
0x14
14
```

12	noshowpoint	

```
// modify showpoint flag
#include <iostream>     // cout,
showpoint, noshowpoint
using namespace std;

int main () {
  double a = 30;
  double b = 10000.0;
  double pi = 3.1416;
  cout.precision (5);
  cout << showpoint     << a <<
'\t' << b << '\t' << pi << '\n';
  cout << noshowpoint << a <<
'\t' << b << '\t' << pi << '\n';
  return 0;
}
```

		Possible output: `30.000  10000.  3.1416` `30      10000   3.1416`
13	noshowpos	Desafecta mostrar signo. `// modify showpos flag` `#include <iostream>     // cout, showpos, noshowpos` `using namespace std;` `int main () {` `  int p = 1;` `  int z = 0;` `  int n = -1;` `  cout << showpos   << p << '\t' << z << '\t' << n << '\n';` `  cout << noshowpos << p << '\t' << z << '\t' << n << '\n';` `  return 0;` `}` Salida: `+1      +0      -1` `1       0       -1`
14	noskipws	Desactiva saltear blancos o tabulados. `// skipws flag example` `#include <iostream>     // cout, skipws, std::noskipws` `#include  <sstream>           // istringstream` `using namesapce std;` `int main () {` `  char a, b, c;` `  istringstream iss ("  123");` `  iss >>skipws >> a >> b >> c;` `  cout << a << b << c << '\n';` `  iss.seekg(0);` `  iss >> noskipws >> a >> b >> c;` `  cout << a << b << c << '\n';` `  return 0;` `}` Salida: `123`

<table>
<tr><td></td><td></td><td>

```
  1
```

</td></tr>
<tr><td>15</td><td>noinitbuf</td><td></td></tr>
<tr><td>16</td><td>nouppercase</td><td>Desactiva mayúsculas

```
1 // modify uppercase flag
2 #include <iostream>      // cout,
3 showbase, hex
4 using namespace std;
5
6 //uppercase, nouppercase
7 int main () {
8   cout << showbase << hex;
9   cout  <<  uppercase  <<  77  <<
'\n';
   cout  <<  nouppercase  <<  77  <<
'\n';
   return 0;
}
```

</td></tr>
<tr><td>17</td><td>oct</td><td>Valor numérico octal.

```
1  // modify basefield
2  #include <iostream>      // cout,
3  std::dec, std::hex, oct
4  using namespace std;
5
6  int main () {
7    int n = 70;
8    cout << dec << n << '\n';
9    cout << hex << n << '\n';
10   cout << oct << n << '\n';
     return 0;
   }
```

Output:

```
70
46
106
```

</td></tr>
<tr><td>18</td><td>right</td><td>Alineación a derecha.

```
1  //   modify   adjustfield   using
2  manipulators
3  #include <iostream>      // cout,
4  internal, left, right
5  using namespace std;
6
7  int main () {
8    int n = -77;
9    cout.width(6);      cout      <<
10 internal << n << '\n';
     cout.width(6); cout << left <<
   n << '\n';
```

</td></tr>
</table>

```
  cout.width(6);  cout  <<  right
<< n << '\n';
  return 0;
}
```

Salida:

```
-   77
-77
   -77
```

19	scientific	Habilita notación científica o punto flotante.

```
1  // modify floatfield
2  #include <iostream>     // cout,
3  fixed, scientific
4  using namespace std;
5
6  int main () {
7    double a = 3.1415926534;
8    double b = 2006.0;
9    double c = 1.0e-10;
10
11   cout.precision(5);
12
13   cout << "default:\n";
14   cout << a << '\n' << b << '\n'
15 << c << '\n';
16
17   cout << '\n';
18
19   cout << "fixed:\n" << fixed;
20   cout << a << '\n' << b << '\n'
21 << c << '\n';
22
23   cout << '\n';
24
   cout   <<   "scientific:\n"   <<
  scientific;
   cout << a << '\n' << b << '\n'
  << c << '\n';
   return 0;
  }
```

Salida:

```
default:
3.1416
2006
1e-010

fixed:
```

<table>
<tr><td></td><td></td><td>

```
3.14159
2006.00000
0.00000

scientific:
3.14159e+000
2.00600e+003
1.00000e-010
```

</td></tr>
<tr><td>20</td><td>showbase</td><td>Activa bandera para mostrar base 0xddd para hexadecimal, 0dddd para octal.

```
1 // modify uppercase flag
2 #include <iostream>      // cout,
3 showbase, hex
4 //uppercase, nouppercase
5 using namespace std;
6
7 int main () {
8   cout << showbase << hex;
9   cout  <<  uppercase  <<  77  <<
'\n';
  cout  <<  nouppercase  <<  77  <<
'\n';
  return 0;
}
```

Salida:

```
0X4D
0x4d
```

</td></tr>
<tr><td>21</td><td>showpoint</td><td>Activa bandera.

```
1  // modify showpoint flag
2  #include <iostream>     // cout,
3  showpoint, noshowpoint
4  using namespaces std;
5
6  int main () {
7    double a = 30;
8    double b = 10000.0;
9    double pi = 3.1416;
10   cout.precision (5);
11   cout <<     showpoint << a <<
12 '\t' << b << '\t' << pi << '\n';
   cout  <<  noshowpoint  <<  a  <<
'\t' << b << '\t' << pi << '\n';
   return 0;
}
```

</td></tr>
</table>

<table>
<tr><td>22</td><td>showpos</td><td>Activa bandera para mostrar signo.

```
// modify showpos flag
#include <iostream>      // cout,
showpos, noshowpos
using namespace std;

int main () {
  int p = 1;
  int z = 0;
  int n = -1;
  cout << showpos   << p << '\t'
<< z << '\t' << n << '\n';
  std::cout << std::noshowpos <<
p << '\t' << z << '\t' << n <<
'\n';
  return 0;
}
```

</td></tr>
<tr><td>23</td><td>skipws</td><td>Habilita blancos.

```
// skipws flag example
#include <iostream>      // cout,
skipws, noskipws
#include  <sstream>           //
istringstream
using namespace std;

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

  istringstream iss ("  123");
  iss >> skipws >> a >> b >> c;
  cout << a << b << c << '\n';

  iss.seekg(0);
  iss  >>  noskipws  >>  a  >>  b  >>
c;
  cout << a << b << c << '\n';
  return 0;
}
```

Output:

```
123
  1
```

</td></tr>
<tr><td>24</td><td>unitbuf</td><td>Activa bandera.</td></tr>
<tr><td>25</td><td>uppercase</td><td>Habilita mayúsculas.

```
// modify uppercase flag
#include <iostream>      // cout,
std::showbase, hex
```

</td></tr>
</table>

		4 //uppercase, nouppercase 5 using namespace std; 6 7 int main () { 8 cout << showbase << hex; 9 cout << uppercase << 77 << '\n'; cout << nouppercase << 77 << '\n'; return 0; }
26	ws	Extrae los espacios en blanco y tabuladores.

Ejemplos

cout << *right* << fill('*') << setw(20) << "Hola" << ' ' << 123 << endl;

Manipuladores parametrizados de <iomanip>

Tabla Manipuladores de <iomanip>

Ord.	Manipulador	Comentario
1	resetioflags(n)	Permite desactivar los flags (15) de formato de salida. enum { skipws left right internal dec oct hex showbase showpoint uppercase showpos scientific fixed unitbuf stdio } Veamos un ejemplo:

<table>
<tr>
<td></td>
<td></td>
<td>

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   float x = 121.0/3;
   int y = 123;

   cout << "#" <<
setiosflags(ios::left)
        << setw(12) <<
setprecision(4)
        << x << "#" << endl;
   cout << "#"
        << resetiosflags(ios::left
| ios::dec)
        << setiosflags(ios::hex |
            ios::showbase |
ios::right)
        << setw(8) << y << "#"
        << endl;
   return 0;
}
```

La salida tendrá este aspecto:

```
#40.33       #
#    0x7b#
```

</td>
</tr>
<tr>
<td>2</td>
<td>setbase(n)</td>
<td>

Permite cambiar la base de numeración que se usará para la salida. Sólo se admiten tres valores: 8, 10 y 16, es decir, octal, decimal y hexadecimal. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int x = 123;

   cout << "#" << setbase(8) << x
        << "#" << setbase(10) << x
        << "#" << setbase(16) << x
```

</td>
</tr>
</table>

		`<< "#" << endl;` `return 0;` `}` La salida tendrá este aspecto: `#173#123#7b#`
3	setfill(car)	Permite especificar el carácter de relleno cuando la anchura especificada sea mayor de la necesaria para mostrar la salida. Por ejemplo: `#include <iostream>` `#include <iomanip>` `using namespace std;` `int main() {` `int x = 123;` `cout << "#" << setw(8) << setfill('0')` `<< x << "#" << endl;` `cout << "#" << setw(8) << setfill('%')` `<< x << "#" << endl;` `return 0;` `}` La salida tendrá este aspecto: `#00000123#` `#%%%%%123#`
4	setprecision(n)	Permite especificar el número de dígitos significativos que se muestran cuando se imprimen números en punto flotante: float o double. Por ejemplo: `#include <iostream>` `#include <iomanip>` `using namespace std;` `int main() {`

```
   float x = 121.0/3;

   cout << "#" << setprecision(3)
        << x << "#" << endl;
   cout << "#" << setprecision(1)
        << x << "#" << endl;
   return 0;
}
```

La salida tendrá este aspecto:

```
#40.3#
#4e+01#
```

5	setw(n)	Permite cambiar la anchura en caracteres de la siguiente salida de cout. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int x = 123, y = 432;

   cout << "#" << setw(6) << x <<
"#"
        << setw(12) << y << "#" <<
endl;
   return 0;
}
```

La salida tendrá este aspecto:

```
#   123#         432#
```

6	setiosflag(n)	Permiten activar o desactivar, respectivamente, los flags de formato de salida. Existen quince flags de formato a los que se puede acceder mediante un enum definido en la clase ios:

flag	Acción
skipws	ignora espacios en operaciones de lectura

left	ajusta la salida a la izquierda
right	ajusta la salida a la derecha
internal	deja hueco después del signo o el indicador de base
dec	conversión a decimal
oct	conversión a octal
hex	conversión a hexadecimal
showbase	muestra el indicador de base en la salida
showpoint	muestra el punto decimal en salidas en punto flotante
uppercase	muestra las salidas hexadecimales en mayúsculas
showpos	muestra el signo '+' en enteros positivos
scientific	muestra los números en punto flotante en notación exponencial
fixed	usa el punto decimal fijo para números en punto flotante
unitbuf	vacía todos los buffers después de una inserción
stdio	vacía los buffers stdout y stderr después de una inserción

Veamos un ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   float x = 121.0/3;
   int y = 123;

   cout << "#" <<
setiosflags(ios::left)
        << setw(12) <<
setprecision(4)
        << x << "#" << endl;
   cout << "#"
        << resetiosflags(ios::left
```

```
| ios::dec)
        << setiosflags(ios::hex |
            ios::showbase |
ios::right)
        << setw(8) << y << "#"
        << endl;
   return 0;
}
```

La salida tendrá este aspecto:

```
#40.33       #
#    0x7b#
```

Ejemplos

cout << setprecision(2);
cout.setf(ios::fixed); // **equivalente a** cout << fixed;
cout << setfill('*') << setw(10) << "Hola " << setw(12) << 123.3456 << endl;

Codificaciòn de Manipuladores en <iomanip.n>

```
/*
   Id.Programa: Manipuladores_iomanip.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha…....: julio-2016
   Comentario.: Manipuladores
         1. setw          Permite cambiar la anchura en caracteres de la
                          siguiente salida de cout.
         2. setprecision  Permite especificar el número de dígitos
                          significativos que se muestran cuando se imprimen
                          números en punto flotante: float o double.
         3. setfill       Permite especificar el carácter de relleno cuando
                          la anchura especificada sea mayor de la necesaria
                          para mostrar la salida.
         4. setbase       Permite cambiar la base de numeración que se usará
                          para la salida. Sólo se admiten tres valores:
                          8, 10 y 16, es decir, octal, decimal y hexadecimal.
         5. setiosflags   Permiten activar o desactivar, respectivamente, los
                          flags de formato de salida. Existen quince flags
                          de formato a los que se puede acceder mediante
```

```
                    un enum definido en la clase ios.
          6. resetiosflags Idem anterior.
            Flags utilizados como argumentos en setiosflags y resetiosflags
   Flag          Acción
   skipws        ignora espacios en operaciones de lectura
   left          ajusta la salida a la izquierda
   right         ajusta la salida a la derecha
   internal      deja hueco después del signo o el indicador de base
   dec           conversión a decimal
   oct           conversión a octal
   hex           conversión a hexadecimal
   showbase      muestra el indicador de base en la salida
   showpoint     muestra el punto decimal en salidas en punto flotante
   uppercase     muestra las salidas hexadecimales en mayúsculas
   showpos       muestra el signo '+' en enteros positivos
   scientific    muestra los números en punto flotante en notación exponencial
   fixed         usa el punto decimal fijo para números en punto flotante
   unitbuf       vacía todos los buffers después de una inserción
   stdio         vacía los buffers stdout y stderr después de una inserción
*/

#include <iostream>
#include <iomanip>
using namespace std;

int main() {
  float x = 121.0/3;
  int y = 123;

  cout << "#" << setiosflags(ios::left)
       << setw(12) << setprecision(4) << setfill('*') << setbase(16)
       << x << "#" << endl;
  cout << "#" << 123
       << resetiosflags(ios::left | ios::dec)
       << setiosflags(ios::hex | ios::showbase | ios::right)
       << setw(8) << y << "#"
       << endl;
   cout << y;
}

/* Salida
```

```
#40.33*******#
#7b****0x7b#
0x7b
*/
```

Funciones miembro de la clase ios

	Funciones miembro de <ios.h> Ejemplos de formas de uso stream.indicador() cout.indicador(); cout.bad(); cout.eof(); cout.fill();	
1	bad() Clase ios	`int bad();` Retorna un valor distinto de cero si ha ocurrido un error.

Sólo se comprueba el bit de estado *ios::badbit*, de modo que esta función no equivale a !good().

```
// error state flags
#include <iostream>     // cout, ios
#include <sstream>      // stringstream
using namespace std;

void print_state (const std::ios& stream) {
  cout << " good()=" << stream.good();
  cout << " eof()=" << stream.eof();
  cout << " fail()=" << stream.fail();
  cout << " bad()=" << stream.bad();
}

int main () {
  stringstream stream;

  stream.clear (stream.goodbit);
  cout << "goodbit:"; print_state(stream);cout << '\n';
  stream.clear (stream.eofbit);
  cout << " eofbit:"; print_state(stream);cout << '\n';
  stream.clear (stream.failbit);
  cout << "failbit:"; print_state(stream);cout << '\n';
  stream.clear (stream.badbit);
  cout << " badbit:"; print_state(stream);cout << '\n';
  return 0;
}
```

2	clear() Clase ios	`void clear(iostate state=0);` Sirve para modificar los bits de estado de un *stream*, normalmente para eliminar un estado de error. Se suelen usar constantes definidas en el enum *io_state*definido en *ios*, usando el operador de bits OR para modificar varios bits a la vez.

```
// clearing errors
#include <iostream>     // cout
#include <fstream>      //fstream
using namespace std;

int main () {
  char buffer [80];
```

```
  fstream myfile;

  myfile.open ("test.txt",fstream::in);
  myfile << "test";
  if (myfile.fail()){
    cout << "Error writing to
                          test.txt\n";
    myfile.clear();
  }
  myfile.getline (buffer,80);
  cout << buffer << " successfully
                 read from file.\n";

  return 0;
}

```

3	close() Clase fstream	`filebuf* close();` Actualiza la información actualmente en el buffer y cierra el fichero. En caso de error, retorna el valor 0.
4	eof() Clase ios	`int eof();` Retorna un valor distinto de cero si se ha alcanzado el fin de fichero.

```
// print the content of a text file.
#include <iostream>     // cout
#include <fstream>      //ifstream

int main () {
  ifstream ifs;

  ifs.open ("test.txt");
  char c = ifs.get();
  while (ifs.good()) {
    cout << c;
    c = ifs.get();
  }
  ifs.close();
  return 0;
}
```

		Esta función únicamente comprueba el bit de estado *ios::eofbit*. `//eof example` `#include <iostream>  //cout` `#include <fstream>  //ifstream` `int main () {` `  ifstream  is("example.txt");  // open file` `  char c;` `  while (is.get(c)) // loop getting single characters` `    cout << c;` `  if (is.eof())  // check for EOF` `    cout << "[EoF reached]\n";` `  else` `    cout << "[error reading]\n";` `  is.close();  // close file` `  return 0;` `}`
5	fail() Clase ios	`int fail();` Retorna un valor distinto de cero si una operación sobre el *stream* ha fallado. Comprueba los bits de estado *ios::badbit* y *ios::failbit*.
6	fill() Clase ios	Cambia el carácter de relleno que se usa cuando la salida es más ancha de la necesaria para el dato actual. `int fill();` `int fill(char);` La primera forma devuelve el valor actual del carácter de relleno, la segunda permite cambiar el carácter de relleno para las siguientes salidas, y también devuelve el valor actual. Ejemplo: `  int x = 23;`

```
  cout << "|";
  cout.width(10);
  cout.fill('%');
  cout << x << "|" << x << "|" << endl;
```

7	flags() Clase ios_base	Permite cambiar o leer los flags de manipulación de formato.

```
long flags () const;
long flags (long valor);
```

La primera forma devuelve el valor actual de los flags.

La segunda cambia el valor actual por valor, el valor de retorno es el valor previo de los flags.

Ejemplo:

```
   int x = 235;
   long f;

   cout << "|";
   f = flags();
   f &= !(ios::adjustfield);
   f |= ios::left;
   cout.flags(f);
   cout.width(10);
   cout << x << "|" << endl;
```

8	flush() Clase ostream	

```
ostream& flush();
```

Vacía el buffer asociado al stream. Procesa todas las salidas pendientes.

```
1  // istream::peek example
2  #include <iostream>     //cin,cout
3  #include <string>       //string
4  #include <cctype>       //isdigit
5  using namespace std;
6
7  int main () {
8
9    stcout << "Please, enter a number or
10 a word: ";
11   cout.flush();     // ensure output is
12 written
```

```
13
14   cin >>ws;   // eat up any leading
15 white spaces
16   int   c   =cin.peek();      //   peek
17 character
18
19   if ( c == EOF )
20      return 1;
21   if (isdigit(c) ) {
22     int n;
23     cin >> n;
24     cout << "You entered the number: "
25                   << n << '\n';
26   }
27   else {
28     string str;
29     cin >> str;
     cout << "You entered the word: "
          << str << '\n';
   }
   return 0;
}
```

9 gcount()
Clase istream

```
int gcount();
```

Retorna el número de caracteres sin formato de la última lectura. Las lecturas sin formato son las realizadas mediante las funciones get, getline y read.

```
1  // cin.gcount example
2  #include <iostream>     //cin,cout
3  using namespace std;
4
5  int main () {
6    char str[20];
7
8    cout << "Please, enter a word: ";
9    cin.getline(str,20);
10   cout <<cin.gcount() << " characters
11 read: " << str << '\n';
12
     return 0;
   }
```

10	get() Clase istream	`int get();` `istream& get(char*, int len, char = '\n');` `istream& get(char&);` `istream& get(streambuf&, char = '\n');`

La primera forma extrae el siguiente carácter o EOF si no hay disponible ninguno.

La segunda forma extrae caracteres en la dirección proporcionada en el parámetro char* hasta que se recibe el delimitador del tercer parámetro, el fin de fichero o hasta que se leen len-1 bytes. Siempre se añade un carácter nulo de terminación en la cadena de salida. El delimitador no se extrae desde el stream de entrada. La función sólo falla si no se extraen caracteres.

La tercera forma extrae un único carácter en la referencia a char proporcionada.

La cuarta forma extrae caracteres en el streambuf especificado hasta que se encuentra el delimitador.

```
1  // print the content of a text file.
2  #include <iostream>     // std::cout
3  #include <fstream>      //ifstream
4  using namespace std;
5
6  int main () {
7    ifstream ifs;
8
9    ifs.open ("test.txt",ifstream::in);
10
11   char c = ifs.get();
12
13   while (ifs.good()) {
14     cout << c;
15     c = ifs.get();
16   }
17
18   ifs.close();
19
20   return 0;
   }
```

<table>
<tr>
<td>11</td>
<td>getline()
Clase istream</td>
<td>

```
istream& getline(char*, int, char = '\n');
```

Extrae caracteres hasta que se encuentra el delimitador y los coloca en el buffer, elimina el delimitador del stream de entrada y no lo añade al buffer.

```
// cin.gcount example
#include <iostream>     //cin,cout
using namespace std;

int main () {
  char str[20];

  cout << "Please, enter a word: ";
  cin.getline(str,20);
  cout <<cin.gcount() << " characters
read: " << str << '\n';

  return 0;
}
```

</td>
</tr>
<tr>
<td>12</td>
<td>good()
Clase ios</td>
<td>

Retorna un valor distinto de cero si no ha ocurrido ningún error, es decir, si ninguno de los bits de estado está activo.

Aunque pudiera parecerlo, (*good* significa bueno y *bad* malo, en inglés) esta función no es exactamente equivalente a !*bad*().
En realidad es equivalente a *rdstate*() == 0.

```
// print the content of a text file.
#include <iostream>     // std::cout
#include   <fstream>                //
std::ifstream
using namespace std;

int main () {
  ifstream ifs;

  ifs.open ("test.txt",ifstream::in);

  char c = ifs.get();

  while (ifs.good()) {
    cout << c;
    c = ifs.get();
```

</td>
</tr>
</table>

```
  }

  ifs.close();

  return 0;
}
```

13 ignore()
Clase istream

```
istream& ignore(int n = 1, int delim = EOF);
```

Hace que los siguientes n caracteres en el stream de entrada sean ignorados; la extracción se detiene antes si se encuentra el delimitador delim.

El delimitador también es extraído del stream.

```
// istream::ignore example
#include <iostream>     //cin,cout
using namespace std;

int main () {
  char first, last;

  cout << "Please, enter your first name followed by your surname: ";

  first =cin.get();        // get one character
  cin.ignore(256,' ');      // ignore until space

  last =cin.get();         // get one character

  cout << "Your initials are " << first << last << '\n';

  return 0;
}
```

14 open()
Clase fstream

```
filebuf* open(const char *name, int mode,
   int prot = filebuf::openprot);
```

Abre un fichero para un objeto de una clase específica, con el nombre *name*. Para el parámetro *mode* se puede usar el enum open_mode definido en la clase

ios.

Parámetro mode	Efecto
ios::app	(append) Se coloca al final del fichero antes de cada operación de escritura.
ios::ate	(at end) Se coloca al final del stream al abrir el fichero.
ios::binary	Trata el stream como binario, no como texto.
ios::in	Permite operaciones de entrada en un stream.
ios::out	Permite operaciones de salida en un stream.
ios::trunc	(truncate) Trunca el fichero a cero al abrirlo.

El parámetro *prot* se corresponde con el permiso de acceso DOS y es usado siempre que no se indique el modo *ios::nocreate*. Por defecto se usa el permiso para leer y escribir. En algunas versiones de las bibliotecas de streams no existe este parámetro, ya que está íntimamente asociado al sistema operativo.

```
enum open_mode { in, out, ate, app, trunc,
nocreate,
   noreplace, binary };
```

```
1  // print the content of a text file.
2  #include <iostream>     // std::cout
3  #include <fstream>      //ifstream
4  using namespace std;
5
6  int main () {
7    ifstream ifs;
8
9    ifs.open ("test.txt",ifstream::in);
10
11   char c = ifs.get();
12
```

```
  while (ifs.good()) {
    cout << c;
    c = ifs.get();
  }

  ifs.close();

  return 0;
}
```

15 | peek()
Clase istream

```
int peek();
```

Retorna el siguiente carácter sin extraerlo del stream.

```
// istream::peek example
#include <iostream>     //cin,cout
#include <string>       //string
#include <cctype>       //isdigit
using namespace std;

int main () {

  cout << "Please, enter a number or a word: ";
  cout.flush();    // ensure output is written

  cin >> std::ws;    // eat up any leading white spaces
  int c = std::cin.peek();   // peek character

  if ( c == EOF )
     return 1;
  if (isdigit(c) ) {
    int n;
    cin >> n;
    cout << "You entered the number: " << n << '\n';
  }
  else {
    string str;
    cin >> str;
    cout << "You entered the word: " << str << '\n';
  }

  return 0;
}
```

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td>16</td><td>precisión()
Clase ios_base</td><td>Permite cambiar el número de caracteres significativos que se mostrarán cuando trabajemos con números en coma flotante: float o double.

<pre>int precision();
int precision(char);</pre>
La primera forma retorna el valor actual de la precisión, la segunda permite modificar la precisión para las siguientes salidas, y también devuelve el valor actual.

<pre>1 // modify precision
2 #include <iostream>   //cout, ios
3
4 int main () {
5   double f = 3.14159;
6   cout.unsetf   (ios::floatfield   );
7 // floatfield not set
8   cout.precision(5);
9   cout << f << '\n';
10   cout.precision(10);
11   cout << f << '\n';
12   cout.setf(ios::fixed,ios::floatfield
13); // floatfield set to fixed
14   cout << f << '\n';
     return 0;
   }</pre></td></tr>
<tr><td>17</td><td>put()
Clase ostream</td><td><pre>ostream& put(char ch);</pre>
Inserta un carácter en el stream de salida.

<pre>   char l = 'l';
   unsigned char a = 'a';

   cout.put('H').put('o').put(l).put(a) <<
endl;</pre>
<pre>// typewriter
#include <iostream>   // std::cin, std::cout
#include <fstream>   // std::ofstream</pre></td></tr>
</table>

```
int main () {
  ofstream outfile ("test.txt");
  char ch;

  cout << "Type some text (type a dot to
finish):\n";
  do {
    ch = std::cin.get();
    outfile.put(ch);
  } while (ch!='.');

  return 0;
}
```

18	putback() Clase istream	`istream& putback(char);` Retorna un carácter al stream.

```
1  // istream::putback example
2  #include <iostream>     //cin,cout
3  #include <string>       //string
4
5  int main () {
6    cout << "Please, enter a number or a
7  word: ";
8    char c =cin.get();
9
10   if ( (c >= '0') && (c <= '9') ) {
11     int n;
12     cin.putback (c);
13     cin >> n;
14     cout << "You entered a number: "
15 << n << '\n';
16   }
17   else {
18     string str;
19     cin.putback (c);
20     getline (std::cin,str);
21     cout << "You entered a word: " <<
22 str << '\n';
23   }
24   return 0;
   }
```

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td>19</td><td>rdstate()
Clase ios</td><td>

```
int rdstate();
```

Retorna el estado del *stream*. Este estado puede ser una combinación de cualquiera de los bits de estado definidos en el enum *ios::io_state*, es decir*ios::badbit*, *ios::eofbit*, *ios::failbit* e *ios::goodbit*. El goodbit no es en realidad un bit, sino la ausencia de todos los demás. De modo que para verificar que el valor obtenido por rdstate es *ios::goodbit* tan sólo hay que comparar. En cualquier caso es mejor usar la función good().
En cuanto a los restantes bits de estado se puede usar el operador & para verificar la presencia de cada uno de los bits. Aunque de nuevo, es preferible usar las funciones bad(), eof() o fail().
</td></tr>
<tr><td>20</td><td>read()
Clase istream</td><td>

```
istream& read(char*, int);
```

Extrae el número indicado de caracteres en el array char*. Se puede usar la función gcount() para saber el número de caracteres extraídos si ocurre algún error.

```
1  // read a file into memory
2  #include <iostream>     //cout
3  #include <fstream>      //ifstream
4
5  int main () {
6
7    ifstream                              is
8  ("test.txt",ifstream::binary);
9    if (is) {
10     // get length of file:
11     is.seekg (0, is.end);
12     int length = is.tellg();
13     is.seekg (0, is.beg);
14
15     char * buffer = new char [length];
16
17     cout << "Reading " << length << "
18 characters... ";
19     // read data as a block:
20     is.read (buffer,length);
21
```
</td></tr>
</table>

```
    if (is)
      cout << "all characters read
successfully.";
    else
      cout << "error: only " <<
is.gcount() << " could be read";
    is.close();

    // ...buffer contains the entire
file...

    delete[] buffer;
  }
  return 0;
}
```

21 | seekg()
Clase istream

```
istream& seekg(streampos pos);
istream& seekg(streamoff offset, seek_dir
dir);
```

La primera forma se mueve a posición absoluta, tal como la proporciona la función tellg.

La segunda forma se mueve un número *offset* de bytes la posición del cursor del stream relativa a dir. Este parámetro puede tomar los valores definidos en el enum *seek_dir*: {*beg*, *cur*, *end*};
Para streams de salida usar ostream::seekp.

Usar *seekpos* o *seekoff* para moverse en un buffer de un stream.

```
// read a file into memory
#include <iostream>     //cout
#include <fstream>      //ifstream
using namespace std;

int main () {
  ifstream                          is
("test.txt",ifstream::binary);
  if (is) {
    // get length of file:
    is.seekg (0, is.end);
    int length = is.tellg();
    is.seekg (0, is.beg);

    // allocate memory:
```

```
16    char * buffer = new char [length];
17
18    // read data as a block:
19    is.read (buffer,length);
20
21    is.close();
22
23    // print content:
24    cout.write (buffer,length);
25
26    delete[] buffer;
27  }
28
  return 0;
}
```

22 | seekp()
Clase ostream

```
ostream& seekp(streampos);
ostream& seekp(streamoff, seek_dir);
```

La primera forma mueve el cursor del stream a una posición absoluta, tal como la devuelve la función tellp.

La segunda forma mueve el cursor a una posición relativa desde el punto indicado mediante el parámetro *seek_dir*, que puede tomar los valores del enum*seek_dir*: *beg*, *cur*, *end*.

```
1  // position in output stream
2  #include <fstream>      //ofstream
3  using namespace std;
4
5  int main () {
6
7    ofstream outfile;
8    outfile.open ("test.txt");
9
10   outfile.write    ("This    is    an
11 apple",16);
12   long pos = outfile.tellp();
13   outfile.seekp (pos-7);
14   outfile.write (" sam",4);
15
16   outfile.close();
17
   return 0;
}
```

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td>23</td><td>setf()
Clase ios_base</td><td>Permite modificar los flags de manipulación de formato.


```
long setf(long);
long setf(long valor, long mascara);
```


La primera forma activa los flags que estén activos tanto en el parámetro y deja sin cambios el resto.

La segunda forma activa los flags que estén activos tanto en valor como en máscara y desactiva los que estén activos en mask, pero no en valor. Podemos considerar que mask contiene activos los flags que queremos modificar y valor los flags que queremos activar.
Ambos devuelven el valor previo de los flags.


```
int x = 235;

cout << "|";
cout.setf(ios::left, ios::left |
   ios::right | ios::internal);
cout.width(10);
cout << x << "|" << endl;
```

</td></tr>
<tr><td>24</td><td>tellg()
Clase istream</td><td>

```
long tellg();
```


Retorna la posición actual del stream.


```
1  // read a file into memory
2  #include <iostream>     //cout
3  #include <fstream>      //ifstream
4  using namespace std;
5
6  int main () {
7    ifstream       is       ("test.txt",
8  ifstream::binary);
9    if (is) {
10     // get length of file:
11     is.seekg (0, is.end);
12     int length = is.tellg();
13     is.seekg (0, is.beg);
```

</td></tr>
</table>

```
14
15    // allocate memory:
16    char * buffer = new char [length];
17
18    // read data as a block:
19    is.read (buffer,length);
20
21    is.close();
22
23    // print content:
24    cout.write (buffer,length);
25
26    delete[] buffer;
27  }
28
  return 0;
}
```

25	tellp() Clase ostream	`streampos tellp();` Retorna la posición absoluta del cursor del stream.

```
t// position in output stream
 #include <fstream>      //ofstream
 using namespace std;

 int main () {

   ofstream outfile;
   outfile.open ("test.txt");

   outfile.write     ("This     is     an
apple",16);
   long pos = outfile.tellp();
   outfile.seekp (pos-7);
   outfile.write (" sam",4);

   outfile.close();

   return 0;
 }
```

26	unsetf() Clase ios_base	Permite eliminar flags de manipulación de formato: `void unsetf(long mascara);`

<table>
<tr><td></td><td></td><td>Desactiva los flags que estén activos en el parámetro.


```
1  // modifying flags with setf/unsetf
2  #include <iostream>     //cout,ios
3  using namespace std;
4
5  int main () {
6    cout.setf  (  ios::hex,ios::basefield
7  );  // set hex as the basefield
8    cout.setf   (  std::ios::showbase  );
9  // activate showbase
10   cout << 100 << '\n';
11   cout.unsetf  (  std::ios::showbase  );
   // deactivate showbase
     cout << 100 << '\n';
     return 0;
   }
```

</td></tr>
<tr><td>27</td><td>width()
Clase ios_base</td><td>Cambia la anchura en caracteres de la siguiente salida de stream:


```
int width();
int width(int);
```


La primera forma retorna el valor de la anchura actual, la segunda permite cambiar la anchura para las siguientes salidas, y también devuelve el valor actual de la anchura.


```
   int x = 23;

   cout << "#";
   cout.width(10);
   cout << x << "#" << x << "#" << endl;
```

</td></tr>
<tr><td>28</td><td>write()
Clase ostream</td><td>

```
ostream& write(const char*, int n);
```


Inserta n caracteres (aunque sean nulos) en el stream de salida.


```
t  // position in output stream
   #include <fstream>      //ofstream
   using namespace std;

   int main () {
```

</td></tr>
</table>

```
  ofstream outfile;
  outfile.open ("test.txt");

  outfile.write     ("This     is     an
apple",16);
  long pos = outfile.tellp();
  outfile.seekp (pos-7);
  outfile.write (" sam",4);

  outfile.close();

  return 0;
}
```

Funciones miembro booleanas

- ✓ **good()**
- ✓ **eof()**
- ✓ **fail()**
- ✓ **bad()**
- ✓ **rdstate()**

Tabla de estado de I-O

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
goodbit	No errors (zero value iostate)	true	false	false	false	goodbit
eofbit	End-of-File reached on input operation	false	true	false	false	eofbit
failbit	Logical error on i/o operation	false	false	true	false	failbit
badbit	Read/writing error on i/o operation	false	false	true	true	badbit

Ejemplos

cout << hex << 100 << endl; // emite: 64 valor en base 16.
cout << setfill << '*' << setw(10) << 2343.0; // emite: ??????2343

Funciones útiles: good(), fail(), eof(), bad(), rdstate() o clear()

Hay varios flags de estado que podemos usar para comprobar el estado en que se encuentra un stream.

Concretamente nos puede interesar si hemos alcanzado el fin de fichero, o si el stream con el que estamos trabajando está en un estado de error.

La función principal para esto es *good()*, de la clase *ios*.

Después de ciertas operaciones con streams, a menudo no es mala idea comprobar el estado en que ha quedado el stream. Hay que tener en cuenta que ciertos estados de error impiden que se puedan seguir realizando operaciones de entrada y salida.

Otras funciones útiles son *fail()*, *eof()*, *bad()*, *rdstate()* o *clear()*.

En el ejemplo de archivos de acceso aleatorio hemos usado *clear()* para eliminar el bit de estado *eofbit* del fichero de entrada, si no hacemos eso, las siguientes operaciones de lectura fallarían.

Otra condición que conviene verificar es la existencia de un fichero. En los ejemplos anteriores no ha sido necesario, aunque hubiera sido conveniente, verificar la existencia, ya que el propio ejemplo crea el fichero que después lee.

Cuando vayamos a leer un fichero que no podamos estar seguros de que existe, o que aunque exista pueda estar abierto por otro programa, debemos asegurarnos de que nuestro programa tiene acceso al stream. Por ejemplo:

```
#include <fstream>
using namespace std;

int main() {
   char mes[20];
   ifstream fich("meses1.dat", ios::in | ios::binary);

   // El fichero meses1.dat no existe, este programa es
   // una prueba de los bits de estado.

   if(fich.good()) {
      fich.read(mes, 20);
      cout << mes << endl;
   }
   else {
      cout << "Fichero no disponible" << endl;
      if(fich.fail()) cout << "Bit fail activo" << endl;
      if(fich.eof())  cout << "Bit eof activo" << endl;
      if(fich.bad())  cout << "Bit bad activo" << endl;
   }
   fich.close();

   return 0;
}
```

Ejemplo de fichero previamente abierto

```
#include <fstream>
using namespace std;

int main() {
   char mes[20];
   ofstream fich1("meses.dat", ios::out | ios::binary);
   ifstream fich("meses.dat", ios::in | ios::binary);

   // El fichero meses.dat existe, pero este programa
```

```
   // intenta abrir dos streams al mismo fichero, uno en
   // escritura y otro en lectura. Eso no es posible, se
   // trata de una prueba de los bits de estado.

   fich.read(mes, 20);
   if(fich.good())
      cout << mes << endl;
   else {
      cout << "Error al leer de Fichero" << endl;
      if(fich.fail()) cout << "Bit fail activo" << endl;
      if(fich.eof())  cout << "Bit eof activo" << endl;
      if(fich.bad())  cout << "Bit bad activo" << endl;
   }
   fich.close();
   fich1.close();

   return 0;
}
```

Entradas y Salidas de archivos stream

Las operaciones de acceso a archivos o ficheros requiere incluir el archivo de cabecera <fstream.h> o bien <fstream>. En este archivo están incluidas las siguientes librerías el cual permitirá la utilización de los archivos, a saber:

- ✓ **ifstream**
- ✓ **ofstream**
- ✓ **fstream**

las cuales derivan de la clase ios, que fuera representado en la tabla indicada más arriba.
Los pasos a seguir para poder empezar a utilizar archivos se indican a continuación:

1. **Incluir el archivo de cabecera** <fstream> o <fstream.h>
2. **Abrir el o los archivos necesarios (**ifstream, ofstream, fstream**)**
3. **Operar con el o los archivos para leer o grabar (**read, write**)**
4. **Cerrar el o los archivos cuando ya no se requiera utiliazarlo(s) (**close**)**

Funciones miembro para abrir archivo

Nomenclatura utilizada

NF: Nombre Físico del archivo. Es el rótulo que aparece en el directorio del disco. Es una cadena de caracteres (char*)
NL: Nombre Lógico del archivo. Es un stream representado por una variable.
MA: Modo de Apertura del archivo. Hay 6 modos los cuales son:

- ✓ **ios::in**
- ✓ **ios::out**
- ✓ **ios::app**

- ✓ **ios::ate**
- ✓ **ios::trunc**
- ✓ **ios::binary**

Apertura del archivo

A diferencia de los stream estándar cin, cout, cerr, los cuales no requieren que se los abra en forma explícita, debido a que todos estos stream son abiertos en forma automática y disponibles al usuario para su utilización en la operaciones de lectura y/o escritura y tampoco es necesario cerrarlos ya que son cerrados automáticamente; los archivos en disco serán necesarios realiar en forma explícita estas acciones de apertura y cierre. Para abrir un stream existen las siguientes alternativas:

1. ifstream **strL(NF, MA) ;** ofstream **strG(NF, MA);** fstream **strLG(NF, MA);**
2. ifstream **stream(NF);** ofstream **strG(NF);** fstream **strLG(NF);**
3. ifstream **stream;** ofstream **strG;** fstream **strLG;**

Primero aclaremos que las diferencias de los nombres de los identificadores de clases corresponde a los siguientes casos:

- ✓ ifstream **Abre un stream para entrada (input), es decir, para leer.**
- ✓ ofstream **Abre un stream para salida (outup), es decir, para grabar.**
- ✓ fstream **Abre un stream para entrada/salida (i-o), es decir, para leer/grabar.**

El primer caso emplea dos argumentos, el primero indica el nombre físico del archivo, el cual es una cadena de caracteres (char*) el que podrá incluir además del nombre del archivo, su ruta completa, formada por la unidad de disco y sus directorios por los cuales atraviesa su camino. El segundo argumento indica el modo de apertura como fuera indicado anteriormente, el cual puede presentar más de un modo separado por la barra vertical (|) el cual indica el 'o' lógico.

El segundo caso emplea un único argumento que es el nombre físico del archivo, mientras que el modo de apertura se da por sobreentendido ya que la clase ifstream establece que el modo de apertura es de entrada o input indicado por la primer letra del identificador ifstream.

El tercer caso no se emplea ningún argumento, en este caso se deberá abrir el archivo por medio de la función método 'open', aplicado al objeto stream, el cual presenta dos argumentos, el primero el nombre físico del archivo y el segundo el modo de apertura.

La sintaxis de apertura de archivo de Entrada es

Omisión (1)	**ifstream**();
Inicialización (2)	explicit **ifstream** (const char*

	`filename, ios_base::openmode mode = ios_base::in);`

Apertura de archivos en modo Entrada

Ejemplos

ifstream strL (“Alumnos.txt”, ios::in);
ifstream strL2 (“Alumnos2.txt”);
ifstream strL3;
strL3.**open**(“Alumnos3.txt”,ios::in | ios::binary);

Se espera que los archivos abiertos en modo de entrada existan, ya que de no existir se informará de un error. En caso de existir el puntero se ubica en la primer componente del archivo, es decir la componente cero, aún así siempre el proceso interno se referirá a un byte y no a una componente, por lo que la posición del puntero se ubicará en el byte cero.

Los dos primeros ejemplos abren archivos de texto.

El tercer ejemplo abre un archivo binario. Además como al instante de crear el stream strL3 no se lo asoció a un nombre físico, se lo debe realizar como se establece en la siguiente línea el cual emplea el método open que va precedido por el stream y el punto que separa el objeto stream con el método.

```
// ifstream constructor.
#include <iostream>     // std::cout
#include <fstream>      // std::ifstream

int main () {

  ifstream ifs ("test.txt", ifstream::in);

  char c = ifs.get();

  while (ifs.good()) {
    cout << c;
    c = ifs.get();
  }

  ifs.close();

  return 0;
}
```

Los mismos ejemplos pueden ser aplicados a las aperturas de stream en modo salidas y al modo combinado de entrada/salida.

La sintaxis de apertura de archivo de Salida es

Omisión (1)	**ofstream**();
Inicialización (2)	explicit **ofstream** (const char* filename, ios_base::openmode mode = ios_base::out);

Apertura de archivo en modo Salida

ofstream strL (“Alumnos.txt”, ios::out);
ofstream strL2 (“Alumnos2.txt”);
ofstream strL3;
strL3.**open**(“Alumnos3.txt”,ios::out | ios::binary);

Si los archivos abiertos en modo de salida no existieran, no será error y el archivo es creado desde cero, sin embargo, en caso de que sí existieran, éstos se destruirán, perdiéndose todos los datos allí almacenados y a continuación los archivos se crearán desde cero. El puntero en estos casos se ubica sobre la marca de finde archivo, eof.

```
// ofstream constructor.
#include <fstream>      // std::ofstream

int main () {

  std::ofstream ofs ("test.txt", std::ofstream::out);

  ofs << "lorem ipsum";

  ofs.close();

  return 0;
}
```

La sintaxis de apertura de archivo de Entrada-Salida es

Omisión (1)	**fstream**();
Inicialización (2)	explicit **fstream** (const char* filename,ios_base::openmode mode = ios_base::in | ios_base::out);

Apertura de archivos en modo Entrada-Salida

fstream strL (“Alumnos.txt”, ios::in | ios::out);
fstream strL2 (“Alumnos2.txt”);
fstream strL3;

strL3.**open**(“Alumnos3.txt”, ios:: in | ios::out | ios::binary);

Los archivos deben existir al momento de su apertura, ya que si no existieran, ocasionaría un error. Si la apertura fue existosa entonces el puntero se ubica en su byte de posición cero, como fuera indicado anteriormente en los casos previos.

```
// fstream constructor.
#include <fstream>      // std::fstream

int main () {
```

```
  std::fstream fs ("test.txt", std::fstream::in | std::fstream::out);

  // i/o operations here

  fs.close();

  return 0;
}
```

Operaciones de lectura en archivos de texto –incluye dispositivo por omisión-

Se trata de un objeto global definido en "iostream.h".

En ejemplos anteriores ya hemos usado el operador >>.

El operador >>

Ya conocemos el operador >> lo hemos usado para capturar variables.

```
istream &operator>>(int&)
```

Este operador está sobrecargado en cin para los tipos estándar: int&, short&, long&, double&, float&, char& y char*.

Además, el operador << devuelve una referencia objeto ostream, de modo que puede asociarse. Estas asociaciones se evalúan de izquierda a derecha, y permiten expresiones como:

```
cin >> var1 >> var2;
cin >> variable;
```

Cuando se usa el operador >> para leer cadenas, la lectura se interrumpe al encontrar un carácter '\0', ' ' o '\n'.

Hay que tener cuidado, ya que existe un problema cuando se usa el operador >> para leer cadenas: cin no comprueba el desbordamiento del espacio disponible para el

almacenamiento de la cadena, del mismo modo que la función gets tampoco lo hace. De modo que resulta poco seguro usar el operador >> para leer cadenas.

Por ejemplo, declaramos:

```
char cadena[10];
cin >> cadena;
```

Si el usuario introduce más de diez caracteres, los caracteres después de décimo se almacenarán en una zona de memoria reservada para otras variables o funciones.

Existe un mecanismo para evitar este problema, consiste en formatear la entrada para limitar el número de caracteres a leer:

```
char cadena[10];
cin.width(sizeof(cadena));
cin >> cadena;
```

De este modo, aunque el usuario introduzca una cadena de más de diez caracteres sólo se leerán diez.

Funciones interesantes de cin

Hay que tener en cuenta que cin es un objeto de la clase "istream", que a su vez está derivada de la clase "ios", así que heredará todas las funciones y operadores de ambas clases. Se mostrarán todas esas funciones con más detalle en la documentación de las bibliotecas, pero veremos ahora las que se usan más frecuentemente.

Formatear la entrada

El formato de las entradas de cin, al igual que sucede con cout, se puede modificar mediante flags. Estos flags pueden leerse o modificarse mediante las funciones flags, setf y unsetf.

Otro medio es usar manipuladores, que son funciones especiales que sirven para cambiar la apariencia de una operación de salida o entrada de un stream. Su efecto sólo es válido para una operación de entrada o salida. Además devuelven una referencia al stream, con lo que pueden ser insertados en una cadena entradas o salidas.

Por el contrario, modificar los flags tiene un efecto permanente, el formato de salida se modifica hasta que se restaure o se modifique el estado del flag.

Funciones manipuladoras con parámetros

Para usar estos manipuladores es necesario incluir el fichero de cabecera iomanip.

Existen cuatro de estas funciones manipuladoras aplicables a cin: setw, setbase, **setiosflags y resetiosflags.**

Todas trabajan del mismo modo, y afectan sólo a la siguiente entrada o salida.

En el caso de cin, no todas las funciones manipuladoras tienen sentido, y algunas trabajan de un modo algo diferentes que con streams de salida.

Manipulador setw

Permite establecer el número de caracteres que se leerán en la siguiente entrada desde cin. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   char cad[10];

   cout << "Cadena:"
   cin >> setw(10) >> cad;

   cout << cad << endl
   return 0;
}
```

La salida tendrá este aspecto, por ejemplo

```
Cadena: 1234567890123456
123456789
```

Hay que tener en cuenta que el resto de los caracteres no leídos por sobrepasar los diez caracteres, se quedan en el buffer de entrada de cin, y serán leídos en la siguiente operación de entrada que se haga. Ya veremos algo más abajo cómo evitar eso, cuando veamos la función "ignore".

El manipulador setw no tiene efecto cuando se leen números, por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int x;

   cout << "Entero:"
   cin >> setw(3) >> x

   cout << x << endl
```

```
   return 0;
}
```

La salida tendrá este aspecto, por ejemplo

```
Entero: 1234567
1234567
```

Manipulador setbase

Permite cambiar la base de numeración que se usará para la entrada de números enteros. Sólo se admiten tres valores: 8, 10 y 16, es decir, octal, decimal y hexadecimal. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int x;

   cout << "Entero: ";
   cin >> setbase(16) >> x;

   cout << "Decimal: " << x << endl;
   return 0;
}
```

La salida tendrá este aspecto

```
Entero: fed4

Decimal: 65236
```

Manipuladores setiosflags y resetiosflags

Permiten activar o desactivar, respectivamente, los flags de formato de entrada. Existen quince flags de formato a los que se puede acceder mediante un enum definido en la clase ios:

	flag	Acción
1	skipws	ignora espacios en operaciones de lectura
2	left	ajusta la salida a la izquierda

3	right	ajusta la salida a la derecha
4	internal	deja hueco después del signo o el indicador de base
5	dec	conversión a decimal
6	oct	conversión a octal
7	hex	conversión a hexadecimal
8	showbase	muestra el indicador de base en la salida
9	showpoint	muestra el punto decimal en salidas en punto flotante
10	uppercase	muestra las salidas hexadecimales en mayúsculas
11	showpos	muestra el signo '+' en enteros positivos
12	scientific	muestra los números en punto flotante en notación exponencial
13	fixed	usa el punto decimal fijo para números en punto flotante
14	unitbuf	vacía todos los buffers después de una inserción
15	stdio	vacía los buffers stdout y stderr después de una inserción

De los flags de formato listados, sólo tienen sentido en cin los siguientes: skipws, dec, oct y hex.

Ejemplo

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   char cad[10];

   cout << "Cadena: ";
   cin >> setiosflags(ios::skipws) >> cad;
   cout << "Cadena: " << cad << endl;

   return 0;
}
```

La salida tendrá este aspecto

```
Cadena:         prueba
Cadena: prueba
```

Manipuladores sin parámetros

Existen otro tipo de manipuladores que no requieren parámetros, y que ofrecen prácticamente la misma funcionalidad que los anteriores. La diferencia es que los cambios

son permanentes, es decir, no sólo afectan a la siguiente entrada, sino a todas las entradas hasta que se vuelva a modificar el formato afectado.

Manipuladores dec, hex y oct

```
inline ios& dec(ios& i)
inline ios& hex(ios& i)
inline ios& oct(ios& i)
```

Permite cambiar la base de numeración de las entradas de enteros:

Función	Acción
dec	Cambia la base de numeración a decimal
hex	Cambia la base de numeración a hexadecimal
oct	Cambia la base de numeración a octal

El cambio persiste hasta un nuevo cambio de base. Ejemplo:

```
#include <iostream>
using namespace std;

int main() {
   int x, y, z;

   cout << "Entero decimal (x y z): ";
   cin >> dec >> x >> y >> z;
   cout << "Enteros: " << x << ", "
        << y << ", " << z << endl;
   cout << "Entero octal (x y z): ";
   cin >> oct >> x >> y >> z;
   cout << "Enteros: " << x << ", "
        << y << ", " << z << endl;
   cout << "Entero hexadecimal (x y z): ";
   cin >> hex >> x >> y >> z;
   cout << "Enteros: " << x << ", "
        << y << ", " << z << endl;

   return 0;
}
```

La salida tendrá éste aspecto

```
Entero decimal (x y z): 10 45 25
Enteros: 10, 45, 25
Entero octal (x y z): 74 12 35
Enteros: 60, 10, 29
Entero hexadecimal (x y z): de f5 ff
Enteros: 222, 245, 255
```

Función ws

```
extern istream& ws(istream& ins);
```

Ignora los espacios iniciales en una entrada de cadena. Ejemplo

```
#include <iostream>
using namespace std;

int main() {
   char cad[10];

   cout << "Cadena: ";
   cin >> ws >> cad;
   cout << "Cadena: " << cad << endl;

   return 0;
}
```

La salida tendrá éste aspecto:

```
Cadena:      hola
Cadena: hola
```

Función width()

Cambia la anchura en caracteres de la siguiente entrada de stream:

```
int width();
int width(int);
```

La primera forma devuelve el valor de la anchura actual, la segunda permite cambiar la anchura para la siguiente entrada, y también devuelve el valor actual de la anchura. Esta función no tiene efecto con variables que no sean de tipo cadena.

```
   char cadena[10];
```

```
   cin.width(sizeof(cadena));
   cin >> cadena;
```

Función setf()

Permite modificar los flags de manipulación de formato:

```
long setf(long);
long setf(long valor, long mascara);
```

La primera forma activa los flags que estén activos tanto en el parámetro y deja sin cambios el resto.

La segunda forma activa los flags que estén activos tanto en valor como en máscara y desactiva los que estén activos en mask, pero no en valor. Podemos considerar que mask contiene activos los flags que queremos modificar y valor los flags que queremos activar.

Ambos devuelven el valor previo de los flags.

```
   int x;

   cin.setf(ios::oct, ios::dec | ios::oct | ios::hex);
   cin >> x;
```

Función unsetf()

Permite eliminar flags de manipulación de formato:

```
void unsetf(long mascara);
```

Desactiva los flags que estén activos en el parámetro.

Nota: en algunos compiladores he comprobado que esta función tiene como valor de retorno el valor previo de los flags.

```
   int x;

   cin.unsetf(ios::dec | ios::oct | ios::hex);
   cin.setf(ios::hex);
   cin >> x;
```

Función flags()

Permite cambiar o leer los flags de manipulación de formato:

```
long flags () const;
long flags (long valor);
```

La primera forma devuelve el valor actual de los flags.

La segunda cambia el valor actual por valor, el valor de retorno es el valor previo de los flags.

```
   int x;
   long f;

   f = flags();
   f &= !(ios::hex | ios::oct | ios::dec);
   f |= ios::dec;
   cin.flags(f);
   cin >> x;
```

Función get

La función get() tiene tres formatos:

```
int get();
istream& get(char& c);
istream& get(char* ptr, int len, char delim = '\n');
```

Sin parámetros, lee un carácter, y lo devuelve como valor de retorno:

Nota: Esta forma de la función get() se considera obsoleta.

Con un parámetro, lee un carácter:

En este formato, la función puede asociarse, ya que el valor de retorno es una referencia a un stream. Por ejemplo:

```
char a, b, c;

cin.get(a).get(b).get(c);
```

Con tres parámetros: lee una cadena de caracteres:

En este formato la función get lee caracteres hasta un máximo de 'len' caracteres o hasta que se encuentre el carácter delimitador.

```
char cadena[20];

cin.get(cadena, 20, '#');
```

Función getline

Funciona exactamente igual que la versión con tres parámetros de la función get(), salvo que el carácter delimitador también se lee, en la función get() no.

```
istream& getline(char* ptr, int len, char delim = '\n');
```

Función read

Lee n caracteres desde el cin y los almacena a partir de la dirección ptr.

```
istream& read(char* ptr, int n);
```

Función ignore

Ignora los caracteres que aún están pendientes de ser leídos:

```
istream& ignore(int n=1, int delim = EOF);
```

Esta función es útil para eliminar los caracteres sobrantes después de hacer una lectura con el operador >>, get o getline; cuando leemos con una achura determinada y no nos interesa el resto de los caracteres introducidos. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   char cad[10];
   int i;

   cout << "Cadena: ";
   cin >> setw(10) >> cad;
   cout << "Entero: ";
   cin.ignore(100, '\n') >> i;
   cout << "Cadena: " << cad << endl;
   cout << "Entero: " << i << endl;

   cin.get();
   return 0;
}
```

La salida podría tener este aspecto:

```
Cadena: cadenademasiadolarga
```

```
Entero: 123
Cadena: cadenadem
Entero: 123
```

Función peek()

Esta función obtiene el siguiente carácter del buffer de entrada, pero no lo retira, lo deja donde está.

```
int peek();
```

Función putback()

Coloca un carácter en el buffer de entrada:

```
istream& putback(char);
```

Función get()

single character (1)	`int get();` `istream& get (char& c);`
c-string (2)	`istream& get (char* s, streamsize n);` `istream& get (char* s, streamsize n, char delim);`
stream buffer (3)	`istream& get (streambuf& sb);` `istream& get (streambuf& sb, char delim);`

1) Un caracter

Extrae un carácter desde el stream.

El carácter es retornado primer caso, o puesto como valor en su argumento segundo caso.

(2) c-string

Extrae caracteres desde el stream y lo almacena en s como c-string, hasta n-1 caracteres que han sido extraidos o delimitado por el carácter encontrado por el carácter de nueva línea '\n' o limitado si se indico en su argumento.

El carácter null '\0' es automáticamente agregado para la secuencia si n es mayor a cero, aún si el string es vacio.

(3) stream buffer

Extrae caracteres desde el stream y lo inserta en la secuencia de salida controlado por el objeto sb buffer del stream, deteniéndose tan pronto como falle la inserción o tan pronto como el carácter delimitador es encontrado en la secuencia de entrada.

```
// istream::get example
#include <iostream>     // std::cin, std::cout
#include <fstream>      // std::ifstream

int main () {
  char str[256];

  cout << "Enter the name of an existing text file: ";
  cin.get (str,256);    // get c-string

  ifstream is(str);     // open file

  char c;
  while (is.get(c))          // loop getting single characters
    cout << c;

  is.close();                // close file

  return 0;
}
```

```
istream& getline (char* s, streamsize n );
```

```
istream& getline (char* s, streamsize n, char delim );
```

```
// istream::getline example
#include <iostream>     // std::cin, std::cout

int main () {
  char name[256], title[256];

  cout << "Please, enter your name: ";
  cin.getline (name,256);

  cout << "Please, enter your favourite movie: ";
  cin.getline (title,256);

  cout << name << "'s favourite movie is " << title;

  return 0;
}
```

Versión stream G2Ej04 Creación archivo de Artículos

```
/*
        Id.Programa: G2Ej04stream.cpp
        Autor.....: Lic. Hugo Cuello
        Fecha.....: jun-2016
        Comentario.: Creacion stream archivo de Articulos.
*/
```

```
#include <fstream>
#include <conio.h>
#include <iostream>
using namespace std;

#define CENTINELA 0

typedef char str20[21];
typedef unsigned short ushort;

template <typename T>
ostream &writeblock(ostream &out, const T &block) {
  return out.write(reinterpret_cast <const char*> (&block), sizeof block);
}

struct sArt {
        ushort CodArt;
        str20  Descrip;
        float  PreUni;
};

void ObtDatos(sArt &rArt) {
        do {
     gotoxy(5,1);
     cout << "Alta de Articulos.Dat";
          gotoxy(10,5); clreol();
          cout << "Cod.Articulo FIN = " << CENTINELA << ": ";
          cin >> rArt.CodArt;
        }
        while (!(rArt.CodArt <= 100));
        if (rArt.CodArt) {
                do {
                        gotoxy(10,7); clreol();
                        cout << "Descripcion: ";
                        gets(rArt.Descrip);
                }
                while (!strcmp(rArt.Descrip,""));
                do {
                        gotoxy(10,9); clreol();
                        cout << "Pre.Unitario: ";
                        cin >> rArt.PreUni;
                }
                while (!rArt.PreUni);
        }
} // ObtDatos
```

```
main() {
   sArt rArticulo;
   ofstream Articulos("Articulos2.Dat",ios::binary);

        ObtDatos(rArticulo);
        while (rArticulo.CodArt) {
                Articulos.write((const char*) (&rArticulo), sizeof rArticulo);
                ObtDatos(rArticulo);
        }
        Articulos.close();
        return 0;
}
```

Versión stream G2Ej05 Actualización Precio en Artículos

```
/*
        Id.Programa: G2Ej05stream.cpp
        Autor.....: Lic. Hugo Cuello
        Fecha.....: jun-2016
        Comentario.: Actualizacion de Precio en archivo de Articulos.
*/

#include <conio.h>
#include <iomanip.h>
#include <fstream>
#include <iostream>
using namespace std;

#define modeOpen ios::in | ios::out | ios::binary

typedef char str20[21];
typedef unsigned short ushort;

struct sArt {
        ushort CodArt;
        str20  Descrip;
        float  PreUni;
};

// Prototipos ------------------

void ObtDato(float &);
void ActReg(fstream &, sArt );
// Fin Prototipos --------------
```

```
void ObtDato(float &porc) {
   clrscr();
   gotoxy(10,5); clreol();
   cout << "Porcentaje: ";
   cin >> porc;
} // ObtDato

void ActReg(fstream &Art, sArt rArt) {
   long tam = Art.tellg();

   tam -= sizeof (rArt);
   Art.seekg(tam,ios::beg);
   Art.write((const char *) &rArt, sizeof rArt);
} // ActReg

main() {
  sArt  rArticulo;
  float porcje;
   long pos;

   cout.precision(2);
   cout.setf(ios::fixed);
   fstream Articulos;
   Articulos.open("Articulos.Dat", modeOpen);
   ObtDato(porcje);
   while (Articulos.read((char *) &rArticulo,sizeof(rArticulo))) {
      rArticulo.PreUni *= (1 + porcje / 100);
      ActReg(Articulos,rArticulo);
      cout << left << setw(20) << rArticulo.Descrip << " " << right << setw(8)
           << rArticulo.PreUni << endl;
   }
   Articulos.seekg(0L,ios::end);
   cout << Articulos.tellg() << endl;
   Articulos.close();
   return 0;
}
```

Versión stream G2Ej06 Precios Mayores de Articulos grabar en Mayores

```
/*
        Id.Programa: G2Ej06stream.cpp
        Autor.....: Lic. Hugo Cuello
        Fecha.....: jun-2016
        Comentario.: Genera archivo de Mayores precios de Articulos.
*/
```

```
#include <conio.h>
#include <fstream>
#include <iostream>
using namespace std;

typedef unsigned short ushort;
typedef char str20[21];

struct sArt {
  ushort CodArt;
  str20  Descrip;
  float  PreUni;
};

struct sMay {
  ushort CodArt;
  float  PreUni;
};

// Prototipos -----------------
void Abrir(ifstream &, ofstream &);
void ObtDato(float &);
void GenReg(ofstream &, sArt );
void Cerrar(ifstream &, ofstream &);
// Fin Prototipos -------------

void Abrir(ifstream &Art, ofstream &May) {
  Art.open("Articulos.Dat",ios::in | ios::binary);
  May.open("Mayores.Dat",ios::out | ios::binary);
} // Abrir

void ObtDato(float &impMax) {
  clrscr();
  gotoxy(10,5); clreol();
  cout << "Importe Maximo: ";
  cin >> impMax;
} // ObtDato

void GenReg(ofstream &May, sArt rArt) {
  sMay rMay;

  rMay.CodArt = rArt.CodArt;
  rMay.PreUni = rArt.PreUni;
  May.write((const char*) (&rMay),sizeof rMay);
} // GenReg
```

```
void Cerrar(ifstream &Art, ofstream &May) {
  Art.close();
  May.close();
} // Cerrar

main() {
  ifstream Articulos;
  ofstream Mayores;
  sArt  rArticulo;
  float maxImp;

  Abrir(Articulos,Mayores);
  ObtDato(maxImp);
  while (Articulos.read((char*) (&rArticulo),sizeof(rArticulo)))
    if (rArticulo.PreUni > maxImp)
      GenReg(Mayores,rArticulo);
  Cerrar(Articulos,Mayores);
  return 0;
}
```

Versión Corte de Control con stream

```
/*
        Id.Programa: G2Ej09stream.cpp Corte de Control n = 2.
        Autor.....: Lic. Hugo Cuello
        Fecha.....: jun-2016
        Comentario.: Examen a alumnos de distintas Universidades y Facultades.
             Tecnica de Corte de Control.
*/

#include <iomanip>
#include <fstream>
#include <iostream>
using namespace std;

typedef unsigned int  word;
typedef unsigned short byte;
typedef char str5[6];
typedef char str15[16];
typedef char str20[21];
struct sExa {
         str5  CodUni,
        CodFacu;
         long  NroLeg;
         str20 ApeNom;
         byte  Nota;
```

```
};

// Prototipos ----------------------------------------------------
void Abrir(ifstream &Examen);
void Inic(word &totInsGral, word &totAprGral);
void EmiteCabLst();
void IniCab(word &totIns, word &totApr, str5 Clave, str5 &ClaveAnt);
void EmiteCabCte(str15 titulo,str5 codigo);
void ProcAlum(sExa rExa, word &tInsFacu, word &tAprFacu);
void CalcPie(word totIns, word totApr, word &totInsM, word &totAprM);
void EmitePie(str15 titulo, word totInsFacu, word totAprFacu);
// Fin Prototipos -----------------------------------------------

void main() {
        ifstream Examen;
        sExa rExamen;
        word totInsGral,
      totAprGral,
      totInsUni,
      totAprUni,
      totInsFacu,
      totAprFacu;
        str5 UniAnt,
      FacuAnt;

        Abrir(Examen);
        freopen("Examen.Lst","w",stdout);
        Inic(totInsGral,totAprGral);
        EmiteCabLst();
        Examen.read((char*) (&rExamen),sizeof(rExamen));
        while (!Examen.eof()) {
                IniCab(totInsUni,totAprUni,rExamen.CodUni,UniAnt);
                EmiteCabCte("*Cod.Univ.: ",rExamen.CodUni);
                while (!Examen.eof() && strcmp(rExamen.CodUni,UniAnt) == 0) {
                        IniCab(totInsFacu,totAprFacu,rExamen.CodFacu,FacuAnt);
                        EmiteCabCte("**Cod.Fac.: ",rExamen.CodFacu);
                        while (!Examen.eof() && strcmp(rExamen.CodUni,UniAnt) == 0
                                && strcmp(rExamen.CodFacu,FacuAnt) == 0) {
                           ProcAlum(rExamen,totInsFacu,totAprFacu);
                           Examen.read((char*) (&rExamen),sizeof(rExamen));
                        }
                        CalcPie(totInsFacu,totAprFacu,totInsUni,totAprUni);
                        EmitePie("Facu.: ",totInsFacu,totAprFacu);
                }
                CalcPie(totInsUni,totAprUni,totInsGral,totAprGral);
                EmitePie("Univ..: ",totInsUni,totAprUni);
        }
```

```
        EmitePie("General: ",totInsGral,totAprGral);
        freopen("CON","w",stdout);
        Examen.close();
} //main

void Abrir(ifstream &Exa) {

        Exa.open("Examen.Dat",ios::binary);
} //Abrir

void Inic(word &tInsG, word &tAprG) {

        tInsG = tAprG = 0;
} // Inic

void EmiteCabLst() {

        cout << "Listado examen a alumnos" << endl;
} //EmiteCab

void IniCab(word &totIns, word &totApr, str5 Clave, str5 &ClaveAnt) {

        totIns = totApr = 0;
        strcpy(ClaveAnt,Clave);
} //IniCab

void EmiteCabCte(str15 titulo, str5 codigo) {

        cout << endl << titulo << " " << codigo;
        if (strstr(titulo,"Fac") != NULL)
                cout << endl << setw(5) << " " << "Nro.Leg.  Nota" << endl;
} //EmiteCabCte

void ProcAlum(sExa rExa, word &tInsFacu, word &tAprFacu) {

        ++tInsFacu;
        if (rExa.Nota >= 4) {
                ++tAprFacu;
                cout << setw(6) << "  " << setw(6) << rExa.NroLeg << "    " << setw(2)
                        << rExa.Nota << endl;
        }
} //ProcAlum

void CalcPie(word totIns, word totApr, word &totInsM, word &totAprM) {

        totInsM += totIns;
        totAprM += totApr;
```

```
} //CalcPie

string replicate(char car, unsigned n) {
  string cad = "";

  for(unsigned i = 1; i <= n; i++)
    cad += car;
  return cad;
} // replicate

void EmitePie(str15 titulo, word totIns, word totApr) {
  string ast;

    if (titulo[0] == 'F')
      ast = replicate('*',2);
    else
      if (titulo[0] == 'U')
        ast = replicate('*',1);
  cout << " " << ast << "Tot.Insc. " << titulo << " " << setw(6) << totIns << endl;
  cout << " " << ast << "Tot.Apr.. " << titulo << " " << setw(6) << totApr << endl;
} //EmitePie
```

Versión apareo de archivos técnica High Value y stream

```
/*
   Id.Programa: G2Ej10HVstream.cpp Apareo
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2016
   Comentario.: Apareo entre MaeVjo y Nov.
                Genera MaeNvo y LstErr.
                Tecnica con HIGH_VALUE.
*/

#include <fstream>
#include <iostream>
using namespace std;

#define HIGH_VALUE 10000
typedef char str20[21];

struct sMae {
  long  cmpClv;
  int   cmp1,
        cmp2;
  str20 RazSoc,
        Domic,
```

```
      Local;
 long  NroCUIT;
};

struct sNov {
 sMae rMaeN;
 char codMov;
};

// Prototipos -----------------------------------------
void Abrir(ifstream &, ofstream &, ifstream &);
void ApareoHV(ifstream &, ofstream &, ifstream &);
void LecHV(ifstream &, sMae &);
void LecHV(ifstream &, sNov &);
void CerrarEliminarRenombrar(ifstream &, ofstream &, ifstream &);
// Fin Prototipos -------------------------------------

void Abrir(ifstream &MaeV, ofstream &MaeN, ifstream &Nov) {

 MaeV.open("MaeVjo.Dat",ios::binary);
 MaeN.open("MaeNvo.Dat",ios::binary);
 Nov.open("Noved.Dat",ios::binary);
} //Abrir

void LecHV(ifstream &MaeV, sMae &rMaeV) {

 if (!MaeV.eof())
   MaeV.read((char *) &rMaeV,sizeof(rMaeV));
 else
   rMaeV.cmpClv = HIGH_VALUE;
} //LecHV

void LecHV(ifstream &Nov, sNov &rNov) {

 if (!Nov.eof())
   Nov.read((char *) &rNov,sizeof(rNov));
 else
   rNov.rMaeN.cmpClv = HIGH_VALUE;
} //LecHV

void ApareoHV(ifstream &MaeV, ofstream &MaeN, ifstream &Nov) {
 sMae rMaeV,
      rMaeN;
 sNov rNov;

 freopen("ErroresABMHD.Lst","w",stdout);
 LecHV(MaeV,rMaeV);
```

```
  LecHV(Nov,rNov);
  while (rMaeV.cmpClv != HIGH_VALUE && rNov.rMaeN.cmpClv !=
                                                    HIGH_VALUE)
    if (rMaeV.cmpClv == rNov.rMaeN.cmpClv) {
      switch (rNov.codMov) {
        case 'A': cout << "Error por Alta Existente, clave: " <<
                        rNov.rMaeN.cmpClv << "Ubicacion: " <<
                        (Nov.tellg() - (long) sizeof(rNov)) / sizeof(rNov) << endl;
                  rMaeN = rMaeV;
                  MaeN.write((const char *) &rMaeN,sizeof(rMaeN));
                  break;
        case 'M': rMaeN = rMaeV;
                  rMaeN.cmp1 = rNov.rMaeN.cmp1;
                  strcpy(rMaeN.Domic,rNov.rMaeN.Domic);
                  MaeN.write((const char *) &rMaeN,sizeof(rMaeN));
                  break;
      }
      LecHV(MaeV,rMaeV);
      LecHV(Nov,rNov);
    }
    else
      if (rMaeV.cmpClv > rNov.rMaeN.cmpClv) {
        switch (rNov.codMov) {
          case 'A': rMaeN = rNov.rMaeN;
                    MaeN.write((const char *) &rMaeN,sizeof(rMaeN));
                    break;
          case 'B': cout << "Error por Baja inexistente, clave: " <<
                          rNov.rMaeN.cmpClv << "Ubicacion: " <<
                          (Nov.tellg() - (long) sizeof(rNov)) / sizeof(rNov) << endl;
                    break;
          default: cout << "Error por Modificacion inexistente, clave: " <<
                        rNov.rMaeN.cmpClv << "Ubicacion: " <<
                        (Nov.tellg() - (long) sizeof(rNov)) / sizeof(rNov) << endl;
        }
        LecHV(Nov,rNov);
      }
      else {
        rMaeN = rMaeV;
        MaeN.write((const char *) &rMaeN,sizeof(rMaeN));
        LecHV(MaeV,rMaeV);
      }
  freopen("CON","w",stdout);
}  //ApareoHV

void CerrarEliminarRenombrar(ifstream &MaeV, ofstream &MaeN, ifstream &Nov) {

  MaeV.close();
```

```
  MaeN.close();
  Nov.close();
  remove("MaeVjo.Dat");
  rename("MaeNvo.Dat","MaeVjo.Dat");
} //CerrarEliminarRenombrar

int main() {
  ifstream MaeVjo,
           Noved;
  ofstream MaeNvo;

  Abrir(MaeVjo,MaeNvo,Noved);
  ApareoHV(MaeVjo,MaeNvo,Noved);
  CerrarEliminarRenombrar(MaeVjo,MaeNvo,Noved);

  return 0;
}
```

Versión stream ejercicio Facturación a Clientes en Cta./Cte.

```
/*
   Id.Programa: G2Ej11FacturasStream.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2016
   Comentario.: Emision de Facturas a Clientes en Cta.Cte.
                Relaciones entre archivos.
*/

#include<iomanip>
#include <fstream>
#include<iostream>
using namespace std;

#define IVA 0.21

typedef char str5[6];
typedef char str10[11];
typedef char str15[16];
typedef char str20[21];
typedef unsigned short byte;
typedef unsigned word;

struct sPed {
```

```
  long codCli; //Ord. por codCli (con repeticion) + codArt (sin repeticion)
  byte codArt;
  word cant;
};

struct sCli {    //Ord. por codCli
  long  codCli;  //8 digitos.
  str20 razSoc,
        domic,
        local;
  word  codPos;
  char  codPcia;
  str15 nroCUIT;
  str20 formaPago;
  word  codVen;
  float saldo; //Actualizar el saldo
};

struct sArt {   //Relacion 1:1 clv = dir.
  byte  codArt; //1..100
  str20 marca,
        descrip;
  float precio;
  word  stockAct, //Actualizar el stockAct.
        stockMin,
        ptoRep;
  str10 uniMed;
  char  estado;
};

struct sVen {       //desordenado
  word  codVen;     //1..999
  str20 nomVen;
  float porcCom,
        impAcumCom; //Actualizar el impAcumCom.
};

struct sNro {       //desordenado
  str5 tipoDoc;
  long nroDoc;      //Actuaizar nroDoc.
};
```

```
struct sFecF {
  byte dia,
       mes;
  word year;
};

struct sFac {  //Ord.cronologico. El archivo existe y se deben agregar +registros.
  long  nroFac;
  sFecF fecFac;
  long  codCli;
  float impo;
  char  estado; //'A' = Adeuda, 'P' = Pagada.
};

struct sFec {

  int year,
      mes,
      dia,
      dsem;
};

struct sHor {
  int Hora,
      Min,
      Seg;
};

// Prototipos ----------------------------------------------------------
void Abrir(ifstream &, fstream &, fstream &, fstream &,fstream &, ofstream &, fstream
&);
void ArmarAuxVen(fstream &, fstream &);
long GetDate(int &, int &, int &, int &);
void EmiteTapaObtFecha(sFec &);
void BusLinNrosInic(fstream &, str5 , sNro &);
void InicCabCli(byte &, float &, long &nDoc);
bool BusBinCli(fstream &, long , sCli &);
void BusDDVen(fstream &, word , sVen &, fstream &);
void EmiteCabCli(sCli , str20 , sFec , long );
void BusDDArt(fstream &, byte , sArt &);
```

```
void CalcDetFac(word , sArt &, byte &, float &,float &);
void ActArt(fstream &, sArt );
void EmiteDetFac(byte , sArt , word , float );
void CalcPieCli(float ,float &, float &, float &, sVen &);
void ActCli(fstream &, sCli );
void ActVen(fstream &, sVen );
void AgregarRegFac(fstream &, long , sFec , long , float );
void EmitePieFac(byte , float , float , float );
void ActNroI(fstream &, sNro );
void CerrarEliminar();
// Fin Prototipos -------------------------------------------------------

void Abrir(ifstream &Ped, fstream &Cli, fstream &Art, fstream &Ven, fstream &nInic,
ofstream &Fac, fstream &AuxVen) {

  Ped.open("Pedidos.Dat",ios::binary);
  Cli.open("Clientes.Dat",ios::binary);
  Art.open("Articulos.Dat",ios::binary);
  Ven.open("Vendedores.Dat",ios::binary);
  nInic.open("NrosInic.Dat",ios::binary);
  Fac.open("Facturas.Dat",ios::app | ios::binary);
  AuxVen.open("AuxVen.Tmp",ios::binary); //w+b
} //Abrir

void ArmarAuxVen(fstream &Ven, fstream &AuxVen) {
  int  refVen = -1;
  sVen rVen;

  for (int i = 0; i <= 999; i++)
    Ven.write((const char *) &refVen,sizeof(int));
  while (Ven.read((char *) &rVen,sizeof(rVen))) {
    AuxVen.seekp(rVen.codVen * sizeof(rVen),ios::beg);
    refVen = (Ven.tellg() - (long) sizeof(rVen)) / sizeof(rVen);
    AuxVen.write((const char *) &refVen,sizeof(refVen));
  }
} //ArmarAuxVen

long GetDate(int &year, int &mes, int &dia, int &ds) {
  time_t     rawtime;
  struct tm *timeinfo;
```

```
  time ( &rawtime );
  timeinfo = localtime ( &rawtime );
  year = 1900 + timeinfo->tm_year;
  mes  = 1 + timeinfo->tm_mon;
  dia  = timeinfo->tm_mday;
  ds   = 1 + timeinfo->tm_wday;
  return (1900 + timeinfo->tm_year) * 10000 + (1 + timeinfo->tm_mon) * 100 + timeinfo-
>tm_mday;
} // GetDate

long GetTime(int &hh, int &mm, int &ss) {
  time_t     rawtime;
  tm *timeinfo;

  time ( &rawtime );
  timeinfo = localtime ( &rawtime );
  hh = timeinfo->tm_hour;
  mm = timeinfo->tm_min;
  ss = timeinfo->tm_sec;
  return timeinfo->tm_hour * 10000 + timeinfo->tm_min * 100 + timeinfo->tm_sec;
} // GetTime

void EmiteTapaObtFecha(sFec &rFec) {

  cout << "FACTURACION A CLIENTES" << endl;
  cout << " EN CUENTA CORRIENTE" << endl;
  GetDate(rFec.year,rFec.mes,rFec.dia,rFec.dsem);
} //EmiteTapaObtFecha

void BusLinNrosInic(fstream &nInic, str5 tDoc, sNro &rNroI) {
  bool sigo = true;

  while (sigo) {
    nInic.read((char *) &rNroI,sizeof(rNroI));
    if (strcmp(rNroI.tipoDoc,tDoc) == 0)
      sigo = false;
  }
} //BusLinNrosInic

void InicCabCli(byte &nItem, float &tFac, long &nDoc) {
```

```
  nItem = 0;
  tFac = 0.0;
  nDoc++;
} //InicCabCli

void ObtHora(sHor &rHor) {

  cout << "FACTURACION A CLIENTES" << endl;
  cout << " EN CUENTA CORRIENTE" << endl;
  GetTime(rHor.Hora,rHor.Min,rHor.Seg);
} //ObtHora

bool BusBinCli(fstream &Cli, long cCli, sCli &rCli) {
  int pri,
      ult,
      med;

  pri = 0;
  Cli.seekg(0L,ios::end);
  ult = Cli.tellg() / sizeof (rCli) - 1;
  while (pri <= ult) {
    med = (pri + ult) / 2;
    Cli.seekg(med * sizeof (rCli),ios::beg);
    Cli.read((char *) &rCli,sizeof (rCli));
    if (rCli.codCli == cCli)
      return true;
    else
      if (rCli.codCli < cCli)
        pri = med + 1;
      else
        ult = med - 1;
  }
  return false;
} // BusBinCli

void BusDDVen(fstream &Ven, word cVen, sVen &rVen, fstream &AuxiV) {
  word refVen;

  AuxiV.seekg(cVen * sizeof(rVen),ios::beg);
  AuxiV.read((char *) &refVen,sizeof(refVen));
  Ven.seekg(refVen * sizeof(rVen),ios::beg);
```

```
  Ven.read((char *) &rVen,sizeof(rVen));
} //BusDDVen

void EmiteCabCli(sCli rCli, str20 nVen, sFec rFec, sHor rHor, long nDoc) {
  cout << setw(38) << " " << "FACTURA: " << nDoc << endl;
  cout << setw(38) << " " << "FECHA..: " << setw(2) << rFec.dia << '-' <<
                                 setw(2) << rFec.mes << '-' <<
                                 setw(4) << rFec.year << endl;
  cout << setw(38) << " " << "HORA...: " << setw(2) << rHor.Hora << '-' <<
                                 setw(2) << rHor.Min << '-' <<
                                 setw(4) << rHor.Seg << endl;
  cout << "Cod. Cliente.: " << setw(8) << rCli.codCli <<
         "Raz. Social..: " << setw(20) << rCli.razSoc <<
         "Domicilio....: " << setw(20) << rCli.domic <<
         "Forma de Pago: " << setw(20) << rCli.formaPago <<
         "Cod. Vendedor: " << setw(3) << nVen << endl;
  cout << endl;
  cout << setw(40) << "-" << endl;
  cout << "Item  Cant.  Cod.Art. Descripcion       Pre.Uni.    T.Item" << endl;
  cout << setw(40) << "-" << endl;
} //EmiteCabCli

void BusDDArt(fstream &Art, byte cArt, sArt &rArt) {

  Art.seekg(cArt * sizeof(rArt),ios::beg);
  Art.read((char *) &rArt,sizeof(rArt));
} //BusDDArt

void CalcDetFac(word cant, sArt &rArt, byte &nItem, float &tItem,float &totFac) {

  nItem++;
  tItem = cant * rArt.precio;
  rArt.stockAct -= cant;
  totFac += tItem;
} //CalcDetFac

void ActArt(fstream &Art, sArt rArt) {

  Art.seekg(Art.tellg() - (long) sizeof(rArt),ios::beg);
  Art.read((char *) &rArt,sizeof(rArt));
} //ActArt
```

```
void EmiteDetFac(byte nItem, sArt rArt, word cant, float tItem) {

  cout << setw(2) << nItem <<
          setw(5) << cant <<
          setw(3) << rArt.codArt <<
          setw(20) << rArt.descrip <<
          setw(8) << rArt.precio <<
          setw(8) << tItem << endl;
} //EmiteDetFac

void CalcPieCli(float tBruFac,float &impIVA, float &tNetoFac, float &saldo, sVen
&rVen) {

  impIVA = tBruFac * IVA;
  tNetoFac = tBruFac + impIVA;
  saldo += tNetoFac;
  rVen.impAcumCom += tBruFac * rVen.porcCom / 100;
} //CalcPieCli

void ActCli(fstream &Cli, sCli rCli) {

  Cli.seekp(Cli.tellp() - (long) sizeof(rCli),ios::beg);
  Cli.write((const char *) &rCli,sizeof(rCli));
} //ActCli

void ActVen(fstream &Ven, sVen rVen) {

  Ven.seekp(Ven.tellp() - (long) sizeof(rVen),ios::beg);
  Ven.write((const char *) &rVen,sizeof(rVen));
} //ActVen

void AgregarRegFac(ofstream &Fac, long nFac, sFec rFec, long cCli, float tNetoFac) {
  sFac rFac;

  rFac.nroFac = nFac;
  rFac.fecFac.dia = rFec.dia;
  rFac.fecFac.mes = rFec.mes;
  rFac.fecFac.year = rFec.year;
  rFac.codCli = cCli;
  rFac.impo = tNetoFac;
```

```
  rFac.estado = 'A';
  Fac.write((const char *) &rFac,sizeof(rFac));
} //AgregarRegFac

void EmitePieFac(byte nItem, float tBruFac, float impIVA, float tNetoFac) {
  byte i;

  for (i = 1; i <= 10 - nItem; i++)
    cout << endl;
  cout << setw(38) << " " << "Sub-Total.: " << setw(8) << tBruFac <<
        setw(38) << " " << "I.V.A.....: " << setw(8) << impIVA <<
        setw(38) << " " << "Total Neto: " << setw(8) << tNetoFac << endl;
} //EmitePieFac

void ActNroI(fstream &nInic, sNro rNroI) {

  nInic.seekp(nInic.tellp() - (long) sizeof(rNroI),ios::beg);
  nInic.write((const char *) &rNroI,sizeof(rNroI));
} //ActNroI

void CerrarEliminar() { //FILE *Ped, FILE *Cli, FILE *Art, FILE *Ven,FILE *nInic,
                        //FILE *Fac, FILE *AuxVen)

  fcloseall();
  remove("AuxVen.Tmp");
} //CerrarEliminar

int main() {
  ifstream Pedidos;
  fstream  Clientes,
           Articulos,
           Vendedores,
           NrosInic,
           AuxVendedores;
  ofstream Facturas;
  sFec rFecha;
  sHor rHora;
  sPed rPedido;
  sNro rNroInic;
  sCli rCliente;
  sVen rVendedor;
```

```
  sArt  rArticulo;
  byte  nroItem;
  float totItem,
        totBrutoFac,
        impIVA,
        totNetoFac;

  Abrir(Pedidos,Clientes,Articulos,Vendedores,NrosInic,Facturas,AuxVendedores);
  freopen("FacturasCtaCteCli.Lst","w",stdout);
  ArmarAuxVen(Vendedores,AuxVendedores);
  EmiteTapaObtFecha(rFecha);
  BusLinNrosInic(NrosInic,"FAC",rNroInic);
  Pedidos.read((char *) &rPedido,sizeof(rPedido));
  while (!Pedidos.eof()) {
   InicCabCli(nroItem,totBrutoFac,rNroInic.nroDoc);
   ObtHora(rHora);
   BusBinCli(Clientes,rPedido.codCli,rCliente);
   BusDDVen(Vendedores,rCliente.codVen,rVendedor,AuxVendedores);
   EmiteCabCli(rCliente,rVendedor.nomVen,rFecha,rHora,rNroInic.nroDoc);
   while (!Pedidos.eof() && rPedido.codCli == rCliente.codCli) {
    BusDDArt(Articulos,rPedido.codArt,rArticulo);
    CalcDetFac(rPedido.cant,rArticulo,nroItem,totItem,totBrutoFac);
    ActArt(Articulos,rArticulo);
    EmiteDetFac(nroItem,rArticulo,rPedido.cant,totItem);
    Pedidos.read((char *) &rPedido,sizeof(rPedido));
   }
   CalcPieCli(totBrutoFac,impIVA,totNetoFac,rCliente.saldo,rVendedor);
   ActCli(Clientes,rCliente);
   ActVen(Vendedores,rVendedor);
   AgregarRegFac(Facturas,rNroInic.nroDoc,rFecha,rCliente.codCli,totNetoFac);
   EmitePieFac(nroItem,totBrutoFac,impIVA,totNetoFac);
  }
  ActNroI(NrosInic,rNroInic);
  freopen("CON","w",stdout);
  CerrarEliminar(); //usa fcloseall() Pedidos, Clientes, Articulos, Vendedores, NrosInic,
                    //Facturas, AuxVendedores);
  return 0;
}
```

Versión stream de Gastos anuales

```
/*
   Id.Programa: G2Ej1e3GtosStream.cpp
   Autor.....: Lic. Hugo Cuello
   Fecha.....: jun-2016
   Comentario.: Listado de Gastos ord. x Mes y Dia Imp.Acum.
*/

#include<iomanip>
#include<fstream>
#include<iostream>
using namespace std;

typedef unsigned short byte;
struct sGto {
  byte  mes,
        dia;
  float impo;
};

// Prototipos ------------------------
void Abrir(ifstream &, fstream &);
void ArmarAuxG(fstream &);
int CantDias(byte );
void AcumGto(fstream &, sGto );
string MesStr(unsigned );
void Listado(fstream &);
void CerrarEliminar(ifstream &, fstream &);
// Fin Prototipos ---------------------

void Abrir(ifstream &Gtos, fstream &AuxG) {

  Gtos.open("Gastos.Dat",ios::in | ios::binary);
  AuxG.open("AuxGtos.Tmp",ios::in | ios::out | ios::binary);
} //Abrir

void ArmarAuxG(fstream &AuxG) {
  float impAcum = 0;

  for (int i = 1; i <= 365; i++)
```

```
   AuxG.write((const char *) &impAcum,sizeof(impAcum));
} //ArmarAuxG

int CantDias(byte mes) {
  int sumDias = 0;

  for (int mes_i = 1; mes_i < mes; mes_i++)
    switch (mes_i) {
      case  4:
      case  6:
      case  9:
      case 11: sumDias += 30;
               break;
      case  2: sumDias += 28;
               break;
      default: sumDias += 31;
    }
  return sumDias;
} //CantDias

void AcumGto(fstream &AuxG, sGto rGto) {
  float impAcum;
  int   pos;

  pos = CantDias(rGto.mes) + rGto.dia - 1;
  AuxG.seekg(pos * sizeof(float),ios::beg);
  AuxG.read((char *) &impAcum,sizeof(impAcum));
  impAcum += rGto.impo;
  AuxG.seekp(pos * sizeof(float),ios::beg);
  AuxG.write((const char *) &impAcum,sizeof(float));
} //AcumGto

string MesStr(unsigned mes) {

  switch (mes) {
    case  1: return "Enero";
    case  2: return "Febrero";
    case  3: return "Marzo";
    case  4: return "Abril";
    case  5: return "Mayo";
    case  6: return "Junio";
```

```
   case  7: return "Julio";
   case  8: return "Agosto";
   case  9: return "Septiembre";
   case 10: return "Octubre";
   case 11: return "Noviembre";
   case 12: return "Diciembre";
   default: return "";
  }
} // MesStr

void Listado(fstream &AuxG) {
  float impAcum,
        totGral = 0.0,
        totMes,
        impMenDia,
        impMayMes = 0.0;
  byte  nroDiaMen,
        nroMesMay;

  AuxG.seekg(0,ios::beg);
  freopen("GtosMesDiaAcum.Lst","w",stdout);
  cout.precision(2);
  cout.setf(ios::fixed);
  cout << "Listado de Gastos ord. por Mes y Dia por Imp.Acum." << endl;
  for (int mes = 1; mes <= 12; mes++) {
    totMes = 0;
    impMenDia = 1E6;
    cout << "Mes: " << MesStr(mes) << endl;
    cout << setw(7) << " " << "Dia    Imp.Acu." << endl;
    for (byte dia = 1; dia <= CantDias(mes + 1) - CantDias(mes); dia++) {
      AuxG.read((char *) &impAcum,sizeof(impAcum));
      if (impAcum) {
        if (impAcum < impMenDia) {
          impMenDia = impAcum;
          nroDiaMen = dia;
        }
        cout << setw(8) << " " << setw(2) << dia
             << setw(4) << " " << setw(8) << impAcum << endl;
        totMes += impAcum;
      }
    }
```

```
   totGral += totMes;
   if (impMenDia > impMayMes) {
     impMayMes = impMenDia;
     nroMesMay = mes;
    }
   cout << "Tot. Mes: " << totMes << endl;
   cout << "Nro. Dia < $: " << nroDiaMen << endl << endl;
  }
  cout << "Tot. Anual: " << totGral << endl;
  cout << "Nro. Mes de > $ de los < Dias: " << setw(2) <<
          nroMesMay << "(" << MesStr(nroMesMay) << ")" << endl;
  freopen("CON","w",stdout);
} //Listado

void CerrarEliminar(ifstream &Gtos, fstream &AuxG) {

  Gtos.close();
  AuxG.close();
  remove("AuxGtos.Tmp");
} //CerrarEliminar

int main() {
  ifstream Gastos;
  fstream  AuxGtos;
  sGto rGto;

  Abrir(Gastos,AuxGtos);
  ArmarAuxG(AuxGtos);
  while (Gastos.read((char *) &rGto,sizeof(rGto)))
    AcumGto(AuxGtos,rGto);
  Listado(AuxGtos);
  CerrarEliminar(Gastos,AuxGtos);
  return 0;
}
```

FUNCIONES PARA ARCHIVOS DE TEXTO Y BINARIO –estilo FILE*-

TABLAS – ARCHIVOS FILE*

Funciones para archivos de Texto y Binario estilo FILE*

ARCHIVOS
NL indica Nombre Lógico, variable de tipo archivo. **NF** indica Nombre Físico. **NR** indica Nombre de Registro. **Dir** indica dirección o referencia o posición. **LUGAR** indica **SEEK_SET** considera que el puntero es desde el inicio, **SEEK_CUR** considera que el puntero es desde el lugar actual, **SEEK_END** considera que el puntero es desde el final. **IMPORTANTE**: Si se actualiza un archivo, en la modalidad "r+b" o "w+b", se debe indicar después de grabar el registro previamente leído, la próxima posición con fseek(NL,ftell(NL),SEEK_SET), ya que no avanza el puntero automáticamente, a efectos de continuar con la lectura secuencial de los próximas componentes. Pág.266 Apéndice B "El Lenguaje de Programación C" 2da.Ed. – Kernighan & Ritchie. La ruta del nombre físico del archivo se utiliza '/' para cambiar de niveles entre las carpetas, y NO '\' o en su lugar "\\" doble barra invertida. Ej.: "C:/Borland/BCC55/prgsFtes/Articulos.Dat". Ej.: "C:\\Borland\\BCC55\\prgsFtes\\Articulos.Dat".

Motivo	Texto en Pascal	Texto en C/C++	Binario en Pascal	Binario en C/C++
Tipo	Tipo *text* Un archivo de texto se compone de líneas y cada línea finaliza con una marca de fin de línea **eoln**. Función **eoln**(NL). Retorna true si el puntero está sobre la marca eoln, sino, false. El fin del archivo está indicado por otra marca, en este caso la marca de fin de archivo **eof**. Función **eof**(NL). Retorna true si el puntero está sobre la marca de eof, sino, false.	En el modo de apertura "t".	Tipo **file of** *tipo* *tipo* puede ser cualquier tipo simple de dato primitivo o cadena de caracteres, o puntero; como cualquier tipo estructurado de dato, registro, conjunto o arreglo o combinadas arreglo de registros, arreglo de conjuntos. La única excepción es el tipo archivo, sea de text o file of.	En el modo de apertura "b".
Asociar NL, NF	assign(NL,NF)		Assign(NL,NF)	
Apertura	**reset**(NL) Debe existir, sino es error. Ubica el puntero al archivo en la primer componente o primera línea. **rewrite**(NL) Si existe lo elimina y lo crea desde cero. Si no existe no es error y lo crea desde cero. Ubica el puntero sobre la marca eof. **append**(NL) Si existe ubica el puntero al final sobre la marca eof, permitiendo agregar nuevas componentes. Un archivo de texto abierto con reset, solo puede leerse. Un archivo de texto abierto con rewrite, solo puede grabarse. Un archivo de texto abierto con append, solo puede agregar componentes al final del archivo.	aTxt=**fopen**(NF,"r"), o fopen(NL,"w"), o fopen(NL,"a"), o fopen(NL,"rt"), o fopen(NL,"wt"), o fopen(NL,"at"), o fopen(NL,"r+"), o fopen(NL,"w+"), o fopen(NL,"a+"), o fopen(NL,"r+t"), o fopen(NL,"w+t"), o fopen(NL,"a+t");, o fopen(NL,"rt+"), o fopen(NL,"wt+), o fopen(NL,"at+"). **freopen**(NF,modo,flujo) flujo puede ser stdin, stdout, stderr. Asocia un nuevo archivo con un flujo ya abierto, al cual cerrará. Retorna NULL si no pudo llevarse a cabo la reapertura. Para cerrar un flujo estandar redireccionado a otro flujo, hacer, **fclose**(stdout) esto restituye la salida por defecto que es la pantalla. También se puede usar **freopen**("CON","w",stdout).	**reset**(NL) **rewrite**(NL) *filemode* es una variable primitiva cuyo valor por omisión es 2, indicando que puede leer o grabar si se abre con reset. Si se asigna un valor de 0 solo lectura, un valor de 1 solo grabar. Si se abre con rewrite siempre se considera el valor 2, o dicho de otra manera, no se considera el valor de filemode, por lo que se puede leer o grabar.	aBin=**fopen**(NF,"rb"), o fopen(NF,"wb"), o fopen(NL,"r+b"), o fopen(NL,"w+b"), o fopen(NL,"rb+"), o fopen(NL,"wb+") o fopen(NL,"ab") o fopen (NL,"ab+") o fopen(NL,"a+b").
Operación	**Leer**:	**Leer**:	**Leer**:	**Leer**:

Motivo	Texto en Pascal	Texto en C/C++	Binario en Pascal	Binario en C/C++
	read(NL,var,…) **readln**(NL,var,…) **Grabar**: **write**(NL,exp,…) **writeln**(NL,exp,…)	**fscanf**(NL,*formato*,var,…) **int fgetc**(NL) o retorna EOF si es fin de archive o error. **char* fgets** (cad, cant, NL) lee hasta n-1 cars, si encuentra nueva linea se detiene, incluyéndose en cad terminado por '\0', retorna cad o NULL si es fin de archivo o error. **Int getc**(FILE *f) equivalente a fgetc. **getchar**() // teclado+ENTER **getch**() // sin eco. **getche**() // con eco. Getw() **scanf**(form,&var) // teclado **cin** >> var // teclado C++ Grabar: **Int fprintf** (NL, *formato*, exp,…) **int fputc**(car,NL) **int fputs**(cad3,NL) **int putc**(car,FILE *f) equivalente a fputc. **int putchar**(car) // pantalla.Equivalente a putc(car, stdout) **putw**(int,f) // 2 bytes. **printf**(form,exp) // pantalla **cout** << exp // pantalla C++	**read**(NL,NR) El registro a leer es donde está ubicado el puntero del archivo, luego de esta operación el puntero avanza automáticamente a la próxima componente. Leer fuera de los límites del archivo siempre es error. **Grabar**: **write**(NL,NR) El registro a grabar es en donde se encuentra ubicado el puntero del archivo, luego de esta operación el puntero avanza automáticamente a la próxima componente. Grabar antes de la posición cero es error, pero grabar más allá de la marca de eof no es error y además de crear la componente se generar todas aquellas componentes entre la marca eof hasta la componente recién creada.	**fread**(&NR,tamaño,cant,NL) **Grabar**: **fwrite**(&NR,tamaño,cant,NL)
Cierre	**close**(NL) Cerrar un archivo de texto implica vaciar el buffer y luego cierra el archivo. Si un archivo de texto no es cerrado y si el buffer no se había completado implica que se perderán esos datos, ya que no se grabarán en el archivo. En un archivo binario esta situación no sucede. De todas maneras es conveniente cerrar siempre un archivo abierto.	**fclose**(NL) **fcloseall**()	**close**(NL)	**fclose**(NL) **fcloseall**()

Motivo	Texto en Pascal	Texto en C/C++	Binario en Pascal	Binario en C/C++
Estado de fin de archivo	**eof**(NL) Retorna true si el puntero está sobre la marca eof, sino, false. Es útil utilizar esta función cuando recorremos secuencialmente el archivo para leer.	**feof**(NL) **ferror**(FILE *f)	**eof**(NL)	**feof**(NL) **ferror**(FILE *f)
Estado posición del puntero			**filepos**(NL) Retorna la posición del puntero en cantidad de componentes.	**ftell**(NL) Retorna la posición del puntero en cantidad de bytes. Para convertir a cantidad de componentes, se debe realizar el siguiente cálculo: **ftell**(NL) / **sizeof**(NR)
Estado tamaño del archivo			**filesize**(NL) Retorna el tamaño del archivo en cantidad de componentes.	No existe un comando específico para esta acción. No obstante, puede realizarse con otros comandos, **fseek** y ftell.
Mover el puntero a otra posición			**seek**(NL,DIR) Mueve el puntero del archivo indicado a la posición indicada por DIR	**fseek**(NL,DIR,ORIGEN) ORIGEN es **SEEK_SET** la cantidad de componentes a desplazar se considera desde el incio del archivo; si es **SEEK_CUR** se considera desde la posición actual del puntero; si es **SEEK_END** se considera desde el final del archivo. Ejemplos: Suponiendo que el archivo contiene 15 componentes posiciones de 0 a 14, y que el puntero se encuentra en la posición 7, que es la componente 8, luego de hacer: fseek(NL,5,SEEK_SE

Motivo	Texto en Pascal	Texto en C/C++	Binario en Pascal	Binario en C/C++
				T) el puntero se ubica en la posición 5 que es la componente 6. fseek(NL,5,SEEK_CUR) el puntero se ubica en la posición 7 + 5 = 12 o componente 13. fseek(NL,5,SEEK_END) el puntero se ubica en la posición 15 + 5 = 20 o componente 21. **rewind**(NL); Mueve el puntero al inicio.
Cortar el archivo en una posición seleccionada			**truncate**(NL)	No existe una función equivalente al procedimiento truncate de Turbo Pascal. En cambio, si se accede a los servicios brindados por el D.O.S. interrupción 21h mediante la siguiente función de usuario se obtiene su equivalente. #include <dos.h> **struct** REGPACK rf; **void** truncate(**FILE** *f) { fseek(f,ftel(f),SEEK_SET); // Posición actual en // cantidad de bytes. rf.r_ax = 0x4000; // Servicio // grabar aleatorio. Rf.r_bx = fileno(f); // Número de hadle. Rf.r_cx = 0; // Actualizar el tamño // del archivo. Rf.r_ds =

Motivo	Texto en Pascal	Texto en C/C++	Binario en Pascal	Binario en C/C++
				FP_SEG(fileno(f)); // Segmento del buffer // de dato. Rf.r_dx = FP_OFF(fileno(f)); // Desplazamiento del // buffer de dato. Intdos(&rf,&rf); // Invoca int 21h. } // truncate
Cambiar el nombre físico a un archivo.	**rename**(NL,NvoNF);	**rename**(VjoNF,NvoNF);	**rename**(NL,NvoNF)	**rename**(VjoNom,NvoNom); VjoNom y NvoNom son cadenas, no se permite uso de comodines, retorna cero si la operación fue correcta, sino -1L y errno se pone a **ENOENT** indica, Ningún archivo o directorio. **EACCES** indica, Permiso negado. **ENOTSAM** indica, Ningún dispositivo.
Eliminar físicamente un archivo.	**erase**(NL);	**remove**(NF);	**erase**(NL);	**remove**(NF); El archivo debe estar cerrado, se permite indicar la ruta de acceso, retorna cero sil a operación fue correcta, sino, -1L y errno se pone a : ENOENT indica, Ningún archivo o directorio. EACCES indica, Permiso negado.

FUNCIONES PARA ARCHIVOS DE TEXTO Y BINARIO –estilo C++ -stream-

TABLAS ARCHIVOS ESTILO C++ STREAMS

Funciones para archivos de Texto y Binario estilo C++

NL indica Nombre Lógico, variable de tipo archivo. NF indica Nombre Físico. NR indica Nombre de Registro. Dir indica dirección o referencia o posición. LUGAR indica SEEK_SET considera que el puntero es desde el inicio, SEEK_CUR considera que el puntero es desde el lugar actual, SEEK_END considera que el puntero es desde el final.

IMPORTANTE: Si se actualiza un archivo, en la modalidad "r+b" o "w+b", se debe indicar después de grabar el registro previamente leído, la próxima posición con fseek(NL,ftell(NL),SEEK_SET), ya que no avanza el puntero automáticamente, a efectos de continuar con la lectura secuencial de los próximas componentes.

Pág.266 Apéndice B "El Lenguaje de Programación C" 2da.Ed. – Kernighan & Ritchie.

La ruta del nombre físico del archivo se utiliza '/' para cambiar de niveles entre las carpetas, y NO '\' o en su lugar "\\" doble barra invertida.

Ej.: "C:/Borland/BCC55/prgsFtes/Articulos.Dat".

Ej.: "C:\\Borland\\BCC55\\prgsFtes\\Articulos.Dat".

Input/Output library
click on an element for detailed information

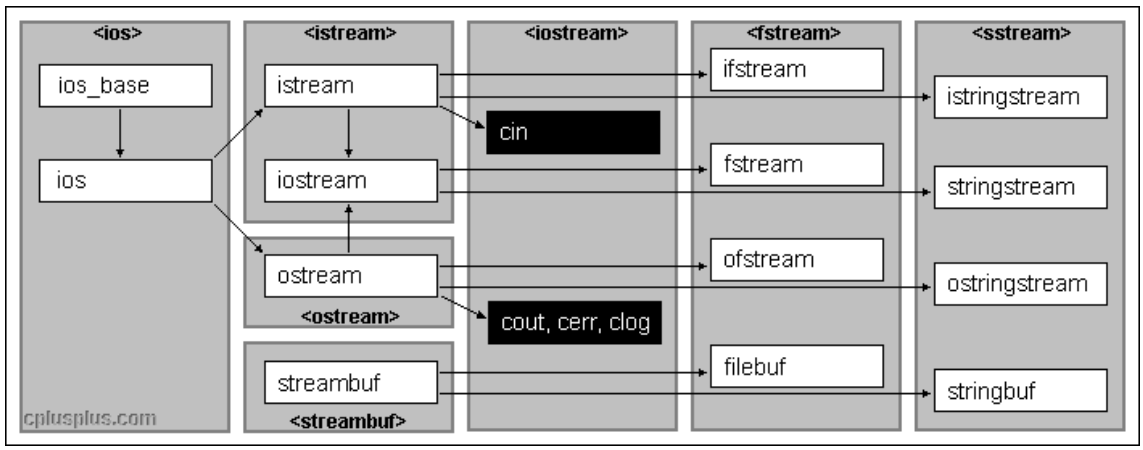


Tabla funciones miembro archivos de Texto y Binario estilo C++ stream

Acción	FILE *	STREAM	Comentario
ABREVIATURAS NF : Nombre Lógico NF : Nombre Físico NR : Nombre de Registro MA : Modo de Apertura NFV : Nombre Físico Viejo NFN : Nombre Físico Nuevo DIR : Dirección			
Definir variable (NL)	FILE *f;	ifstream f; ofstream f; fstream f;	f para leer f para grabar f para leer/grabar
	Ejemplos: ifstream Articulos("Articulos.Dat",ios::out | ios::app); Se definió un stream y se lo asoció al nombre físico. ifstream Articulos; Articulos.open("Articulos.Dat",ios::binary | ios::in); Primero se definió un stream pero sin asociarlo a un nombre físico, por lo que será necesario utilizar posteriormente la función miembro open.		
Abrir	f = fopen(NF, MA);	f.open(NF, MA);	MA: ios::in | ios::out ios::binary | ios::trunc ios::app | ios::ate
Redirccionar Salida	freopen("NF",MA,dispositivo)		freopen("Salida.Txt","w",stdout) Las salidas de printf o cout van a un archivo de texto "Salida.Txt".
Leer	fread(NL, NR);	f.read((char*) &buf, size); f.read(**reinterpret_cast** <char*> (&block), sizeof block);	Leer desde el stream una cantidad de size bytes o caracteres.
Grabar	fwrite(NL, NR);	f.write((const char*) &buf, size); f.write(**reinterpret_cast** <const char*> (&block), sizeof block);	Graba al stream una cantidad de size bytes o caracteres.
Cerrar	fclose(NL);	f.close();	Cierra un stream

Cerrar todos los archivos	fcloseall()	fcloseall()	Cierra todos los archivos abiertos.
Fin de archivo	feof(NL)	f.eof()	Retorna true si el puntero al archivo está ubicado sobre la marca de fin de archivo, sino, false.
Decir posición	ftell(NL)	f.tellg() f.tellp()	Abierto para leer Abierto para grabar
Mover posición Acceso al azar	fseek(NL, DIR)	f.seekg(pos, whereas); f.seekp(pos, whereas);	Abierto para leer **whereas**: ios::beg, cur, end Abierto para grabar **whereas**: ios::beg, cur, end
Mover puntero al inicio del archivo	rewind(NL)	f.seekg(0,ios::beg) f.seekp(0,ios::beg)	Regresa el puntero del archivo indicado como NL al inicio del archivo, es decir, rebobina la cinta del carrete.
Detectar abierto		is_open()	false o true dependiendo de si no pudo o pudo abrir el archivo respectivamente.
Leer carácter		f.get(car);	Lee un carácter desde el stream.
Grabar carácter		f.put(car);	Graba un carácter al stream.
Estado apertura		f.bad();	true si ha ocurrido un error de apertura.
		f.good();	true si no ha ocurrido un error en la apertura.
		f.fail();	true si ha ocurrido un error de apertura.
Leer línea car's.		f.getline(char*, streamsize n); f.getline(char* ,streamzise n, char delim);	Lee una línea desde el stream o n caracteres.
Renombrar	rename(NFN,NFV)	rename(NFN,NFV)	Renombra el archivo NFV con un nuevo nombre indicado en NFN.
Eliminar	remove(NF)	remove(NF)	Elimina físicamente el archivo indicado como NF.

Sitios web de obtención de información utilizadas en esta documentación

Sitios web de información empleada en el documento

http://c.conclase.net/curso/
Teoría con ejemplos de los temas explicados de C / C++ en castellano.

http://www.cplusplus.com/
Teoría con ejemplos de los temas explicados de C / C++ en inglés.

http://pseint.sourceforge.net/
Software que permite codificar en lenguaje natural y generación de los gráficos algorítmicos en programación lineal o la forma Nassi – Shneiderman.