

SEP

SEIT

DGIT

INSTITUTO TECNOLÓGICO DE NUEVO
LAREDO

DEPTO. DE SISTEMAS Y COMPUTACIÓN



"Manejo de Archivos en Lenguaje C++"

Por:

Ing. Bruno López Takeyas, M.C.

<http://www.itnuevolaredo.edu.mx/takeyas>

Email: takeyas@itnuevolaredo.edu.mx

TABLA DE CONTENIDO

	Pág.
Tabla de figuras.....	5
Prefacio.....	7
1.- CONCEPTOS BÁSICOS DE ARCHIVOS.....	8
1.1. ¿Cómo surge la necesidad de utilizar archivos?.....	8
1.2. Relación entre la memoria principal, el microprocesador y dispositivos de almacenamiento secundario.....	9
1.3. Definiciones de datos, registros y archivos.....	10
1.4. Analogías de archivos y archiveros.....	12
1.5. Apertura de archivos.....	16
1.6. Clasificación de archivos por tipo de contenido.....	17
1.6.1. Archivos de texto.....	18
1.6.2. Archivos binarios.....	18
1.7. Clasificación de archivos por tipos de acceso.....	19
1.7.1. Archivos secuenciales.....	20
1.7.1.1. Consulta o recorrido secuencial.....	20
1.7.2. Archivos directos (relativos, de acceso directo o aleatorios)...	22
1.7.2.1. Direcciones lógicas y direcciones físicas.....	22
1.7.2.2. Cálculo de direcciones físicas.....	23
1.7.2.3. Consulta directa.....	24
2.- FUNCIONES DE MANEJO DE ARCHIVOS EN LENGUAJE C++.....	26
2.1. Declaración del alias del archivo.....	26
2.2. Funciones de manejo de archivos en C++.....	26
2.2.1. La función <code>fopen()</code> y modos de apertura de archivos.....	27
2.2.2. Validar la apertura de un archivo.....	28
2.2.3. Cierre de archivos usando <code>fclose()</code> y <code>fcloseall()</code>	29
2.2.4. Escritura de registros usando <code>fwrite()</code>	29
2.2.4.1. Vaciando los buffers con <code>fflush()</code>	31
2.2.5. Lectura de registros usando <code>fread()</code>	31
2.2.6. Reposicionando el apuntador mediante <code>fseek()</code>	31
2.2.6.1. Puntos de referencia de la función <code>fseek()</code>	32
2.2.6.2. Conociendo la posición del apuntador del archivo con la función <code>ftell()</code>	34
2.2.6.3. Colocando el apuntador del archivo al principio con la función <code>rewind()</code>	34

2.2.7. Detectando el final del archivo con <code>feof()</code>	35
2.2.8. Cambiando nombres de archivos mediante <code>rename()</code>	35
2.2.9. Eliminando archivos con la función <code>remove()</code>	36
3.- APLICACIONES DE ARCHIVOS EN C++.....	38
3.1. Declaraciones globales.....	38
3.2. Archivos secuenciales en Lenguaje C++.....	39
3.2.1. ALTAS secuenciales.....	40
3.2.1.1. Diagrama de flujo de la rutina de ALTAS secuenciales.....	40
3.2.1.2. Codificación de la rutina de ALTAS secuenciales.....	42
3.2.2. CONSULTAS secuenciales.....	43
3.2.2.1. Diagrama de flujo de la rutina de CONSULTAS secuenciales.....	43
3.2.2.2. Codificación de la rutina de CONSULTAS secuenciales.....	43
3.2.3. LISTADO secuencial.....	44
3.2.3.1. Diagrama de flujo de la rutina de LISTADO secuencial.....	44
3.2.3.2. Codificación de la rutina de LISTADO secuencial.....	46
3.2.4. MODIFICACIONES de datos en un archivo secuencial.....	47
3.2.4.1. Diagrama de flujo de la rutina de MODIFICACION secuencial.....	47
3.2.4.2. Codificación de la rutina de MODIFICACIÓN secuencial.....	49
3.2.5. BAJAS de registros en un archivo secuencial (bajas ógicas y bajas físicas).....	50
3.2.5.1. Diagrama de flujo de la rutina de BAJAS lógicas en un archivo secuencial.....	51
3.2.5.2. Codificación de la rutina de BAJAS lógicas en un archivo secuencial.....	53
3.2.5.3. Diagrama de flujo de la rutina de BAJAS físicas en un archivo secuencial (compactar).....	54
3.2.5.4. Codificación de la rutina de BAJAS físicas en un archivo secuencial (compactar).....	56
3.3. Archivos directos en Lenguaje C++.....	57
3.3.1. ALTAS directas.....	57
3.3.1.1. Diagrama de flujo de la rutina de ALTAS directas.....	57
3.3.1.2. Codificación de la rutina de ALTAS directas.....	59
3.3.2. CONSULTAS directas.....	60
3.3.2.1. Diagrama de flujo de la rutina de CONSULTAS directas.....	60
3.3.2.2. Codificación de la rutina de CONSULTAS directas....	62
3.3.3. MODIFICACIONES directas.....	63
3.3.3.1. Diagrama de flujo de la rutina de MODIFICACIONES directas.....	63

3.3.3.2. Codificación de la rutina de MODIFICACIONES directas.....	65
3.3.4. BAJAS de registros en un archivo de acceso directo (bajas lógicas).....	66
3.3.4.1. Diagrama de flujo de la rutina de BAJAS lógicas directas.....	67
3.3.4.2. Codificación de la rutina de BAJAS lógicas directas..	69
4.- CONCLUSIONES.....	71
5.-BIBLIOGRAFÍA.....	72

TABLA DE FIGURAS

No.	Descripción	Pág.
1	Interacción entre la memoria principal, el microprocesador y los archivos.....	9
2	Formato del registro de Productos.....	11
3	Declaración del registro de Productos.....	11
4	Formato del registro de Productos.....	12
5	Cuadro comparativo de archivos y archiveros.....	13
6	Apertura de archivos.....	17
7	Clasificación de archivos por contenido.....	18
8	Clasificación de archivos por tipo de acceso.....	20
9	Diagrama de flujo de rutina de consulta secuencial.....	21
10	Ejemplo de cálculo del espacio ocupado por un registro.....	22
11	El lenguaje C++ maneja archivos con direcciones físicas.....	23
12	Direcciones lógicas y físicas de un archivo.....	23
13	Cálculo de la dirección física a partir de la dirección lógica.....	24
14	Diagrama de flujo de rutina de consulta directa.....	25
15	Modos de apertura para archivos de texto y binarios.....	27
16	La función fopen.....	28
17	Validar la apertura de un archivo.....	29
18	La función fwrite.....	30
19	Puntos de referencia de la función fseek.....	33
20	La función fseek.....	33
21	La función ftell.....	34
22	La función rewind.....	34
23	La función rename.....	36
24	La función remove.....	37
25	Declaraciones globales de las aplicaciones.....	39
26	Diagrama de flujo de rutina de alta secuencial.....	41
27	Codificación de la rutina de altas secuenciales.....	42
28	Codificación de la rutina de consultas secuenciales.....	43
29	Diagrama de flujo de rutina de listado secuencial.....	45
30	Codificación de la rutina de listado secuencial.....	46
31	Diagrama de flujo de rutina de modificación secuencial.....	48
32	Codificación de rutina de modificación secuencial.....	49
33	Diagrama de flujo de rutina de baja lógica secuencial.....	52
34	Codificación de rutina de baja lógica secuencial.....	53
35	Diagrama de flujo de rutina de baja física secuencial (compactar)...	55
36	Codificación de rutina de baja física secuencial (compactar).....	56
37	Diagrama de flujo de rutina de altas directas.....	58
38	Codificación de rutina de altas directas.....	59
39	Diagrama de flujo de rutina de consultas directas.....	61
40	Codificación de rutina de consultas directas.....	62

41	Diagrama de flujo de rutina de modificación directa.....	64
42	Codificación de rutina de modificaciones directas.....	65
43	Diagrama de flujo de rutina de baja lógica directa.....	68
44	Codificación de rutina de baja lógica directa.....	69

PREFACIO

Durante el tiempo que he impartido la materia de “Administración de Archivos” en la carrera de Ingeniería en Sistemas Computacionales (ISC) en el Instituto Tecnológico de Nuevo Laredo (ITNL), me he percatado de las deficiencias de los alumnos para programar archivos y, aunque es necesario dominar este aspecto de programación para aplicarlo en la materia, no es limitante o requisito estricto para cursarla, ya que la retícula así lo señala. Además estoy enterado que los temas de archivos pertenecen a la última unidad programática de las materias previas de Programación I y II y que debido a lo extenso de esos programas de estudio, no se comprenden completamente los temas relacionados con archivos. Debido a lo anterior, presento este documento basado en un cúmulo de experiencias y dudas planteadas por alumnos que tiene como finalidad reforzar los conocimientos de programación de archivos en Lenguaje C++ para aplicarlos a necesidades específicas de la materia “Administración de Archivos”.

Ing. Bruno López Takeyas, M.C.

<http://www.itnuevolaredo.edu.mx/takeyas>

takeyas@itnuevolaredo.edu.mx

1. CONCEPTOS BÁSICOS DE ARCHIVOS

Esta sección presenta las generalidades relacionadas con archivos antes de empezar a utilizarlos y programarlos. Es necesario involucrarse con la terminología relacionada como archivo, registro, campo, etc. También es recomendable conocer las clasificaciones generales y las operaciones fundamentales con archivos.

1.1. ¿Cómo surge la necesidad de utilizar archivos?

Hasta antes de la materia de Administración de Archivos, muchas de las aplicaciones que los alumnos de ISC han programado han sido usando la memoria principal o memoria RAM como único medio de almacenamiento (usando variables, arreglos o estructuras de datos mas complejas), con el inconveniente que esto representa: la volatilidad de la memoria RAM; es decir, si se apaga la computadora se pierden los datos. Además, algunas aplicaciones exigen transportar los datos de una computadora a otra. De ahí surge la necesidad de almacenar dichos datos de forma permanente que permita retenerlos en ciertos dispositivos de almacenamiento secundario por un período de tiempo largo sin necesidad de suministrarles energía, de tal forma que permitan transportarlos y utilizarlos en otro equipo computacional.

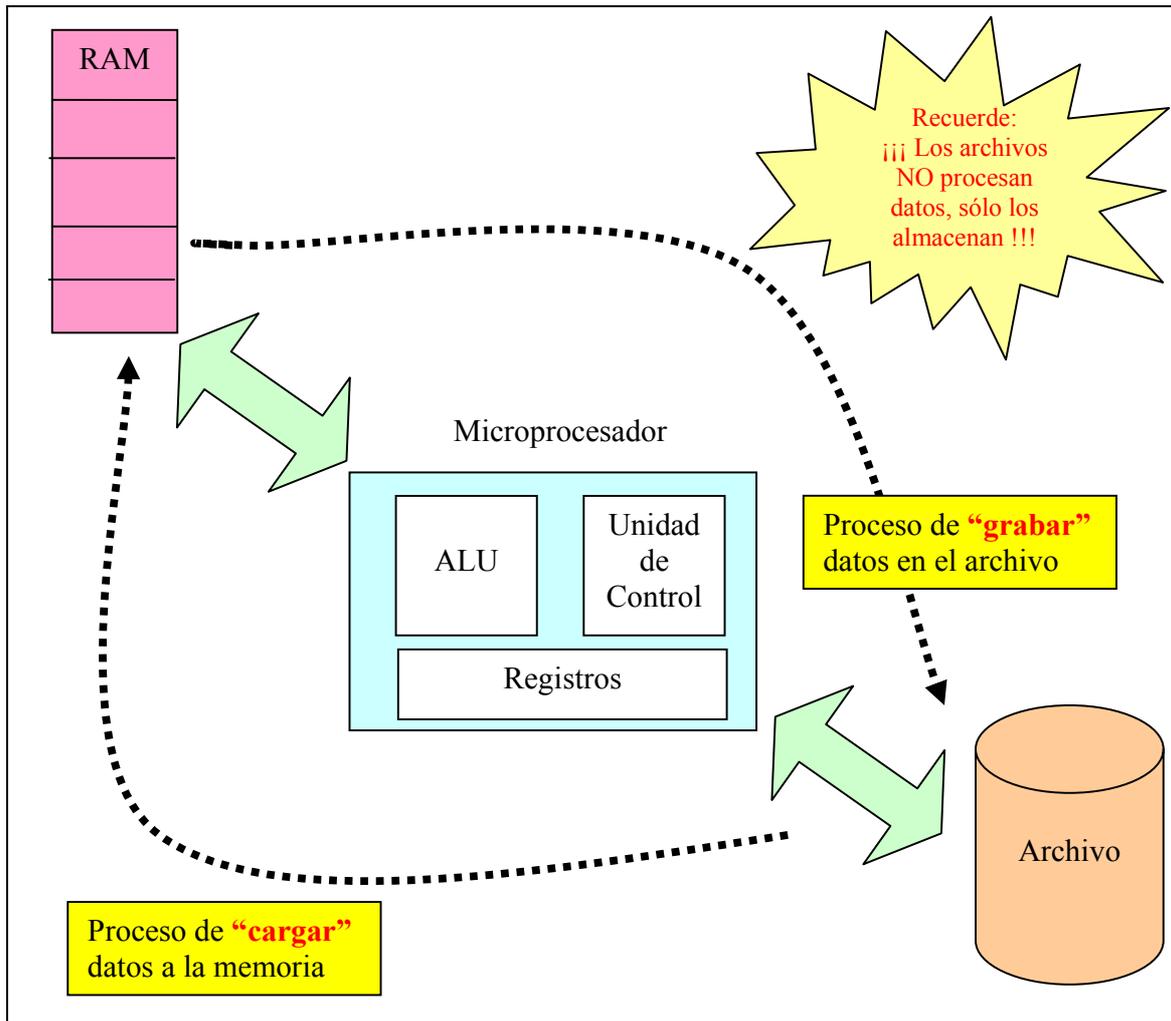


Fig. 1. Interacción entre la memoria principal, el microprocesador y los archivos

1.2. Relación entre la memoria principal, el microprocesador y dispositivos de almacenamiento secundario

Existe una estrecha relación entre la memoria principal, el microprocesador y los dispositivos de almacenamiento secundario ya que el procesamiento que realiza una computadora es tarea absoluta del microprocesador en conjunción con

la memoria principal; es decir, los dispositivos de almacenamiento secundario (diskettes, discos duros, CDs, flash drives, etc.) **no procesan datos, sólo los almacenan.** En estos dispositivos sólo se reflejan los datos previamente procesados y funcionan exclusivamente como una bodega. Esto repercute de manera significativa al momento de programar archivos, ya que para hacerle modificaciones a los datos de un registro previamente almacenado es necesario primero **“cargarlo”** en la memoria principal, es decir, localizar el registro en el archivo y leerlo para colocar sus datos en la memoria RAM, ahí modificarlo y posteriormente **grabarlo en la misma posición** en la que se encontraba, sin embargo estas operaciones no se realizan directamente, sino a través de la unidad aritmética-lógica, la unidad de control y los registros del microprocesador (Fig. 1).

1.3. Definiciones de datos, registros y archivos

- **Datos:** Básicamente se refieren a los testimonios individuales relacionados con hechos, ya sean características de ciertos objetos de estudio o condiciones particulares de situaciones dadas. Los elementos individuales de los archivos se llaman datos o campos. Por ejemplo un cheque de un banco tiene los siguientes campos: Cuenta habiente, Número de cheque, Fecha, Persona a la que se le paga, Monto numérico, Monto con letra, Nota, Identificación del banco, Número de cuenta y Firma. Cada campo es definido por un tipo de dato.
- **Registro:** Es el conjunto completo de datos relacionados pertenecientes a una entrada. P. ejem. Un almacén puede retener los datos de sus productos en registros de acuerdo al formato mostrado en la Fig. 2.

No_prod	Descrip	Cantidad	Precio	Garantia
Entero	Cadena [30]	Entero	Real	Caracter

Fig. 2. Formato del registro de Productos

El registro de la Fig. 2 puede ser declarado globalmente (por encima de la función main) el Lenguaje C++ utilizando struct (Fig. 3).

```

struct tipo_registro
{
    int no_prod;
    char descrip[30];
    int cantidad;
    float precio;
    char garantia;
};
struct tipo_registro Registro;
    
```

Declaración del tipo de dato (en este caso del tipo de registro)

Nótese la ubicación del símbolo ;

Declaración de la variable "Registro" de tipo "tipo_registro"

Fig. 3. Declaración del registro de Productos.

- **Archivo:** En términos computacionales es una colección de datos que tiene un nombre y se guardan en dispositivos de almacenamiento secundario, los cuales pueden ser magnéticos, ópticos, electrónicos, etc. P. ejem. Diskettes, discos duros, CD's, ZIP drives, flash drives, memory sticks, etc.

P. ejem. La Fig. 4 muestra la estructura de un archivo con registros homogéneos de Productos (con la misma estructura) declarados previamente (Fig. 3).

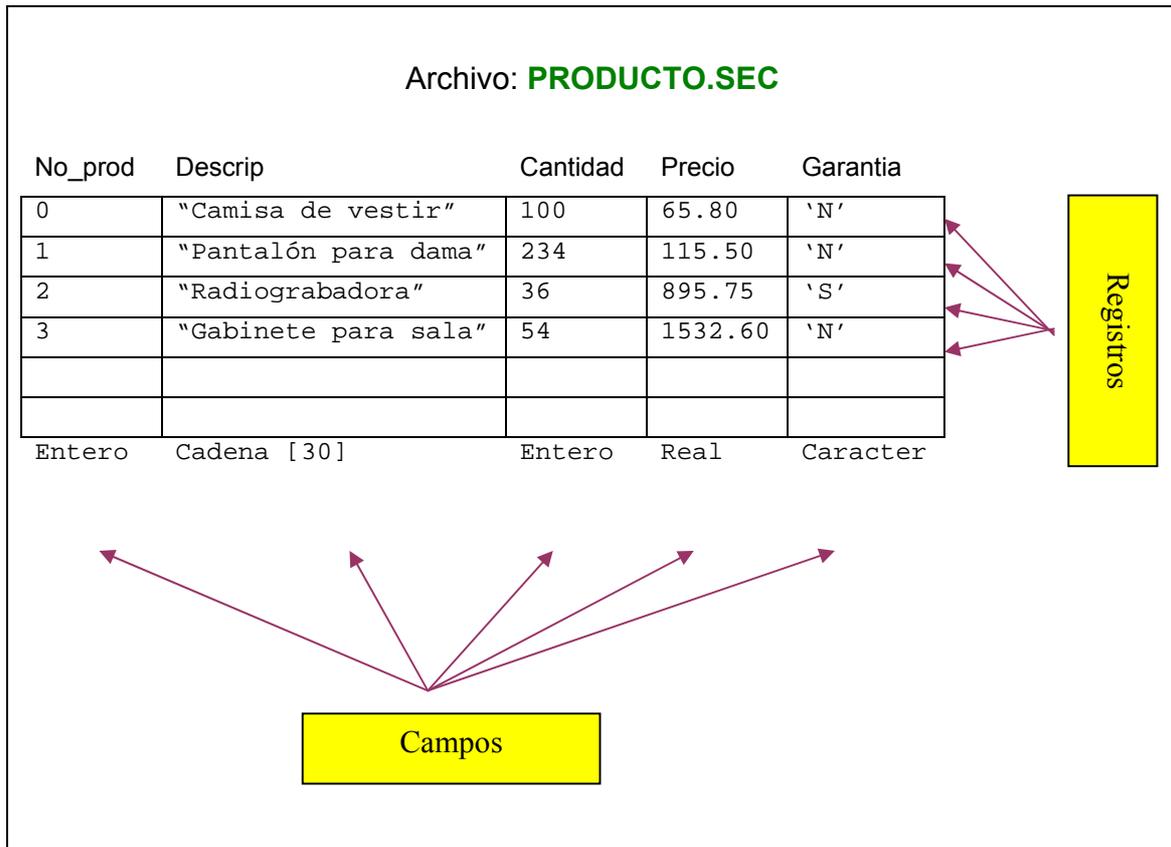
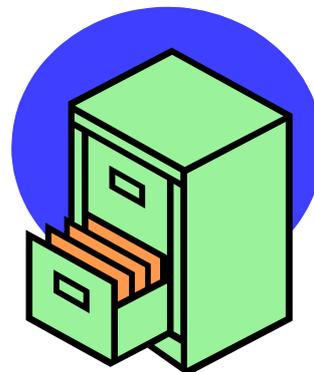


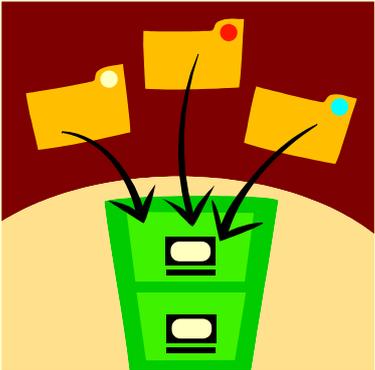
Fig. 4. Formato del registro de Productos

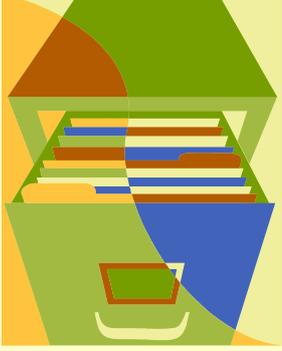
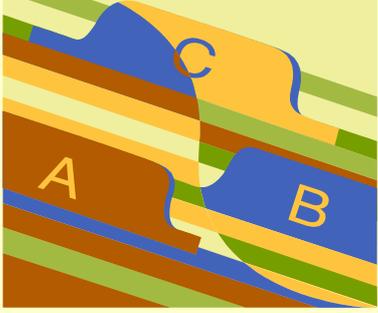
1.4. Analogías de archivos y archiveros

El modo de operación de un archivo puede ser asociado con el de un archivero en una oficina, ya que ambos almacenan datos y operan de forma semejante. De tal forma que muestran las siguientes operaciones, acciones similares y comparaciones:



Operación o acción	Archivero	Archivo computacional
<p data-bbox="233 577 603 663">Identificar la localización de la información</p> 	 <p data-bbox="635 524 1007 981">Localizando el archivero en particular que contiene las carpetas con la información que se solicita, ya que una oficina puede tener varios archiveros debidamente clasificados e identificados</p>	 <p data-bbox="1021 524 1375 1039">Identificando la base de datos correspondiente a la información que se solicita. Una base de datos es una colección de archivos relacionados. P. Ejem. Profesores, alumnos y materias están correlacionados.</p>
<p data-bbox="233 1115 603 1249">Identificar el lugar exacto donde se encuentra la información</p>	<p data-bbox="635 1066 1007 1308">Regularmente un archivero contiene varios cajones, cada uno con información debidamente clasificada y ordenada.</p>	<p data-bbox="1021 1066 1375 1415">Se recomienda que los archivos contengan datos relacionados con un objeto de interés en particular y no de varios. P. Ejem. Sólo datos de ALUMNOS.</p>

<p style="text-align: center;">Nombres</p> 	<p>Se pueden colocar etiquetas a los cajones de los archiveros para identificar su contenido de tal forma que indique el contenido o la clasificación de las carpetas que contiene.</p>	<p>Los nombres de los archivos están regidos por el sistema operativo, pero regularmente se componen del nombre principal y su extensión, la cual puede ser de cualquier tipo, es decir, el usuario puede colocarle la extensión que desee ya sea DAT, TXT, BIN, JK8, etc. Sin embargo se recomienda que tanto el nombre como la extensión sean relevantes al contenido del archivo. P. Ejem. ALUMNOS.DAT, ARCHIVO.TXT</p>
<p style="text-align: center;">Operaciones</p> 	<p>En un archivero se pueden agregar, extraer o cambiar documentos de las carpetas.</p>	<p>Básicamente un archivo solo tiene 2 operaciones:</p> <ul style="list-style-type: none"> • Lectura • Escritura <p>Las demás operaciones se realizan como consecuencia de éstas.</p>

<p style="text-align: center;">Apertura</p> 	<p>Obviamente cuando se requiere agregar o consultar carpetas del cajón de un archivero, es necesario primero abrirlo.</p>	<p>Para acceder los datos de un archivo es necesario abrirlo. Existen varios modos de apertura de los archivos dependiendo de las operaciones que se deseen realizar en él.</p>
<p>Clasificación de los datos</p> 	<p>Los cajones de los archiveros tienen separadores o pequeñas pestañas para identificar las carpetas. Estas facilitan el acceso, ya sea la inserción o la extracción de un carpeta en particular.</p>	<p>Los datos pueden ser almacenados de muchas formas diferentes en los archivos y de esto depende la facilidad (o dificultad) que el archivo muestre para ciertas operaciones de acceso. A estas formas de almacenamiento se les conoce como "organización del archivo".</p>

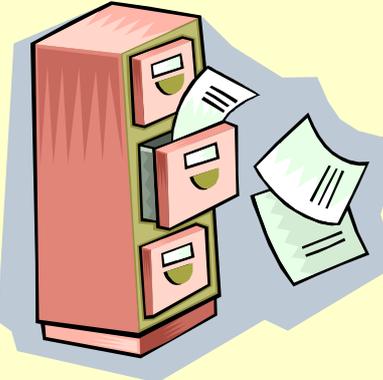
<p style="text-align: center;">Cierre</p> 	<p>Cuando ya no se desea utilizar un cajón de un archivero es necesario cerrarlo, ya que de no hacerlo, se corre el riesgo de dañar o perder la información.</p>	<p>Cuando se termina de utilizar un archivo es necesario cerrarlo. De esa forma se vacía la memoria caché y se asegura almacenar y proteger los datos.</p>
<p style="text-align: center;">Seguridad</p> 	<p>Algunos archiveros cuentan con un candado que permite asegurar los cajones de aperturas no deseadas por personal no autorizado.</p>	<p>Dependiendo del sistema operativo, se pueden aplicar restricciones de seguridad y acceso a los archivos dependiendo de los usuarios; es decir, se establecen políticas de acceso a usuarios.</p>

Fig. 5. Cuadro comparativo de archivos y archiveros

1.5. Apertura de archivos

Antes de escribir o leer datos de un archivo es necesario abrirlo. Al abrir el archivo se establece comunicación entre el programa y el sistema operativo a cerca de cómo accesarlo. Es necesario que el programa le proporcione al sistema operativo el nombre completo del archivo y la intención de uso (leer o escribir datos), entonces se definen áreas de comunicación entre ellos. Una de estas

áreas es una estructura que controla el archivo (**alias del archivo**), de esta forma cuando se solicita una operación del archivo, se recibe una respuesta que informa el resultado mediante un apuntador. Cada archivo abierto requiere un alias para poder realizar operaciones en él (Fig.6).

La estructura del archivo contiene información del archivo que se está usando, así como el tamaño actual y la localización de los buffers de datos.

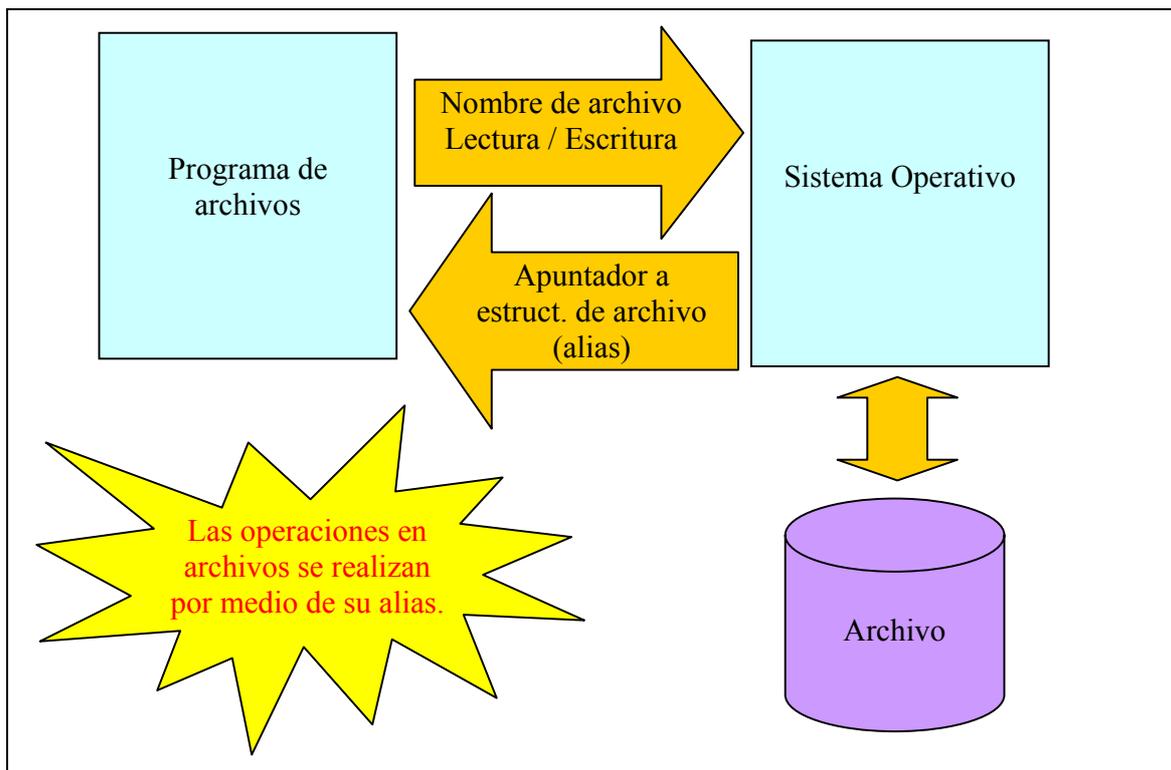


Fig. 6. Apertura de archivos

1.6. Clasificación de archivos por tipo de contenido

Existen muchas clasificaciones de archivos de acuerdo a diferentes criterios o aplicaciones. Aquí se presenta una muy importante: de acuerdo al contenido.

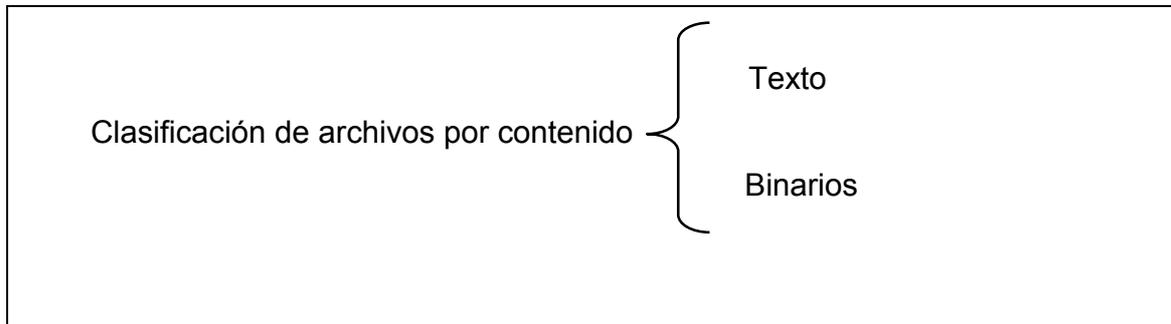


Fig. 7 Clasificación de archivos por contenido

1.6.1. Archivos de texto

Son aquellos que pueden contener cualquier clase de datos y de tal manera que son “entendibles” por la gente. Los datos en un archivo de texto se almacenan usando el código ASCII, en el cual cada carácter es representado por un simple byte. Debido a que los archivos de texto utilizan el código ASCII, se pueden desplegar o imprimir. En este tipo de archivos, todos sus datos se almacenan como cadenas de caracteres, es decir, los números se almacenan con su representación ASCII y no su representación numérica, por lo tanto no se pueden realizar operaciones matemáticas directamente con ellos. P. ejem. Si se guarda el dato 3.141592 en un archivo de texto, se almacena como “3.141592” y nótese que

...

3.141592 ≠ “3.141592”

1.6.2. Archivos binarios

Este tipo de archivos almacenan los datos numéricos con su representación binaria. Pueden ser archivos que contienen instrucciones en lenguaje máquina

listas para ser ejecutadas. Por ejemplo, cuando escribimos un programa en un lenguaje en particular (como C++, Pascal, Fortran, etc), tenemos las instrucciones almacenadas en un archivo de texto llamado programa fuente, pero una vez que lo sometemos a un proceso de compilación y ejecución nuestro programa lo trasladamos a un programa ejecutable (en lenguaje máquina), que es directamente entendido por la computadora y se crea un archivo binario. En este tipo de archivos también se pueden almacenar diferentes tipos de datos incluyendo datos numéricos; sin embargo, cabe destacar que los datos numéricos se graban con su representación binaria (no con su representación ASCII), por tal razón, cuando se despliegan con un editor de textos o por medio de comandos del sistema operativo, aparecen caracteres raros que no se interpretan. P. ejem. Si se guarda el dato 27 en un archivo binario, se almacena como 00001111 y no como "27".

1.7. Clasificación de archivos por tipo de acceso

De acuerdo a la forma de acceder los datos de los archivos, éstos se clasifican en secuenciales o directos (también conocidos como de acceso directo, relativos o aleatorios). En esta sección no se pretende analizar las diferentes estructuras de datos involucradas en estas clasificaciones de archivos ni desarrollar aplicaciones complejas debidamente diseñadas usándolos, sino conocer esencialmente sus conceptos teóricos y la forma de manejarlos.

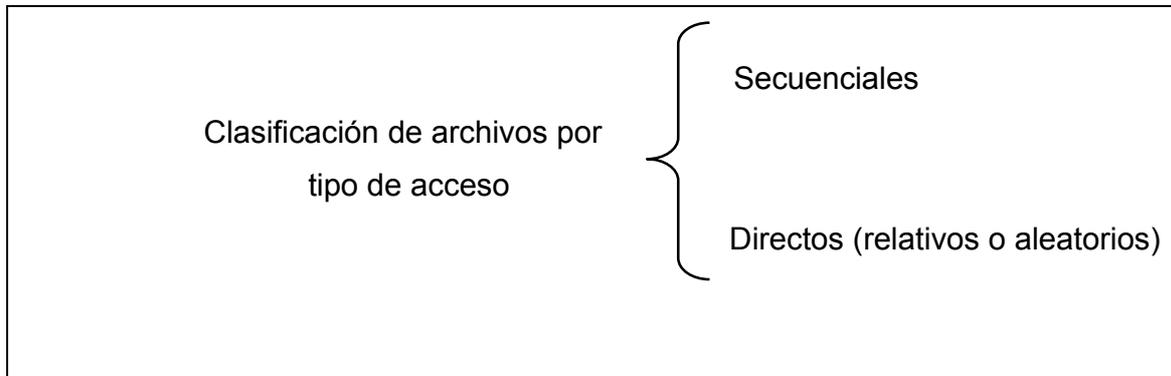


Fig. 8. Clasificación de archivos por tipo de acceso

1.7.1. Archivos secuenciales

Como su nombre lo indica, en este tipo de archivos los registros se graban en secuencia o consecutivamente y deben accesarse de ese mismo modo, es decir, conforme se van insertando nuevos registros, éstos se almacenan al final del último registro almacenado; por lo tanto, cuando se desea consultar un registro almacenado es necesario recorrer completamente el archivo leyendo cada registro y comparándolo con el que se busca. En este tipo de archivo se utiliza una marca invisible que el sistema operativo coloca al final de los archivos: **EOF** (End of File), la cual sirve para identificar dónde termina el archivo

1.7.1.1. Consulta o recorrido secuencial

A continuación se muestra un diagrama de flujo de una rutina de consulta de un registro en particular mediante un recorrido secuencial. En esta rutina se maneja un archivo que almacena datos de productos (**PRODUCTO.SEC**) según el registro declarado en la Fig. 3 y lo recorre completamente en forma secuencial (registro por registro) hasta encontrar el producto solicitado (Fig. 7).

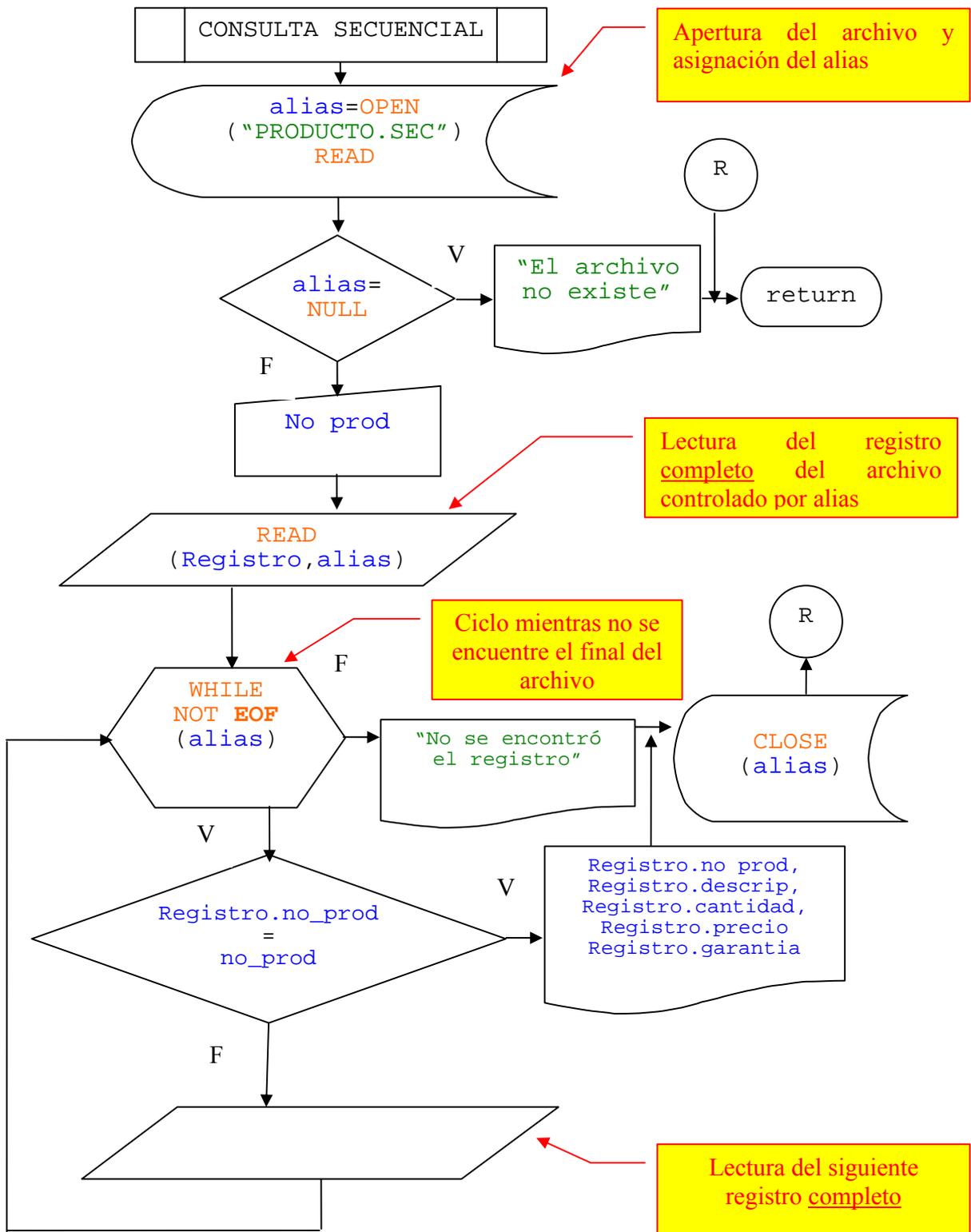


Fig. 9. Diagrama de flujo de rutina de consulta secuencial

1.7.2. Archivos directos (relativos, de acceso directo o aleatorios)

A diferencia de los archivos secuenciales, en los archivos directos no es necesario recorrerlo completamente para acceder un registro en particular, sino se puede colocar el apuntador interno del archivo directamente en el registro deseado, permitiendo con esto mayor rapidez de acceso. Cabe destacar que para reposicionar este apuntador se utiliza el comando `SEEK` indicándole la dirección del registro que se desea.

1.7.2.1. Direcciones lógicas y direcciones físicas

El lenguaje C++ utiliza direcciones físicas para los archivos (a diferencia de direcciones lógicas de otros lenguajes como PASCAL), esto es, que el direccionamiento consiste en el espacio ocupado por los datos en el archivo (calculado en bytes) no en el renglón al que se asignó dicho registro. P. ejem. Suponga que tiene un archivo llamado “**PRODUCTO.SEC**” que almacena registros con el formato mostrado en la Fig. 3. Dicho registro tiene cinco campos, cada uno con un tipo de dato determinado, un tamaño específico y que al sumarlos se obtiene el espacio ocupado por cada registro con este formato (Fig.10).

Campo	Tipo de dato	Tamaño en bytes
no_prod	Int	2
Descrip	char [30]	30
Cantidad	int	2
Precio	float	4
Garantia	char	1
TOTAL		39

Fig. 10. Ejemplo de cálculo del espacio ocupado por un registro.

Los archivos en Lenguaje C++ usan direcciones físicas y no lógicas como en PASCAL.



Fig.11. El lenguaje C++ maneja archivos con direcciones físicas

1.7.2.2. Cálculo de direcciones físicas

Para poder reposicionar el apuntador de un archivo en un registro específico es necesario calcular su dirección física correspondiente de acuerdo al espacio ocupado por sus registros predecesores.

Archivo: **PRODUCTO.SEC**

Dir.	Dir.	No_prod	Descrip	Cantidad	Precio	Garantia
Lóg.	Fís.					
0	0	0	"Camisa de vestir"	100	65.80	'N'
1	41	1	"Pantalón para dama"	234	115.50	'N'
2	82	2	"Radiograbadora"	36	895.75	'S'
3	123	3	"Gabinete para sala"	54	1532.60	'N'
4	164					
5	205					

Entero Cadena [30] Entero Real Caracter

Fig. 12. Direcciones lógicas y físicas de un archivo.

De esta forma, para convertir direcciones lógicas en direcciones físicas se utiliza la fórmula mostrada en la Fig. 13.

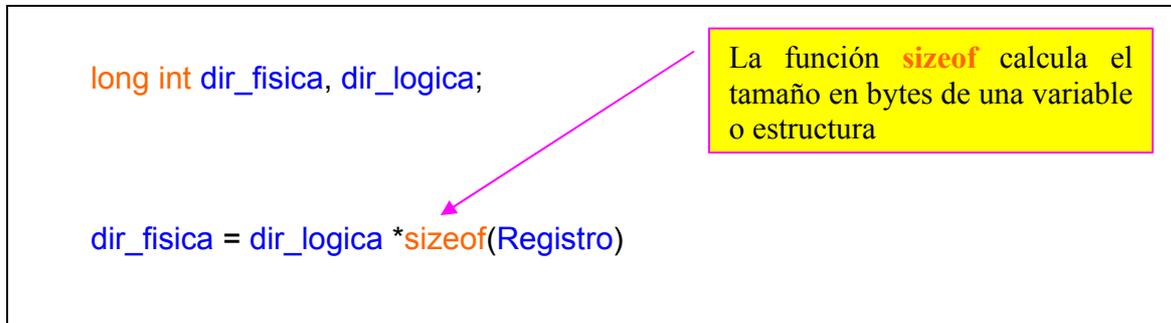


Fig. 13. Cálculo de la dirección física a partir de la dirección lógica

1.7.2.3. Consulta directa

A continuación se muestra un diagrama de flujo de una rutina de consulta directa de un registro en particular. En esta rutina se maneja el archivo **PRODUCTO.SEC** según el registro declarado en la Fig. 3. A diferencia del recorrido secuencial, en el que es necesario leer cada registro del archivo, en la consulta directa se calcula la dirección física del registro solicitado y se posiciona el apuntador del archivo directamente en ese registro para leerlo (Fig. 14).

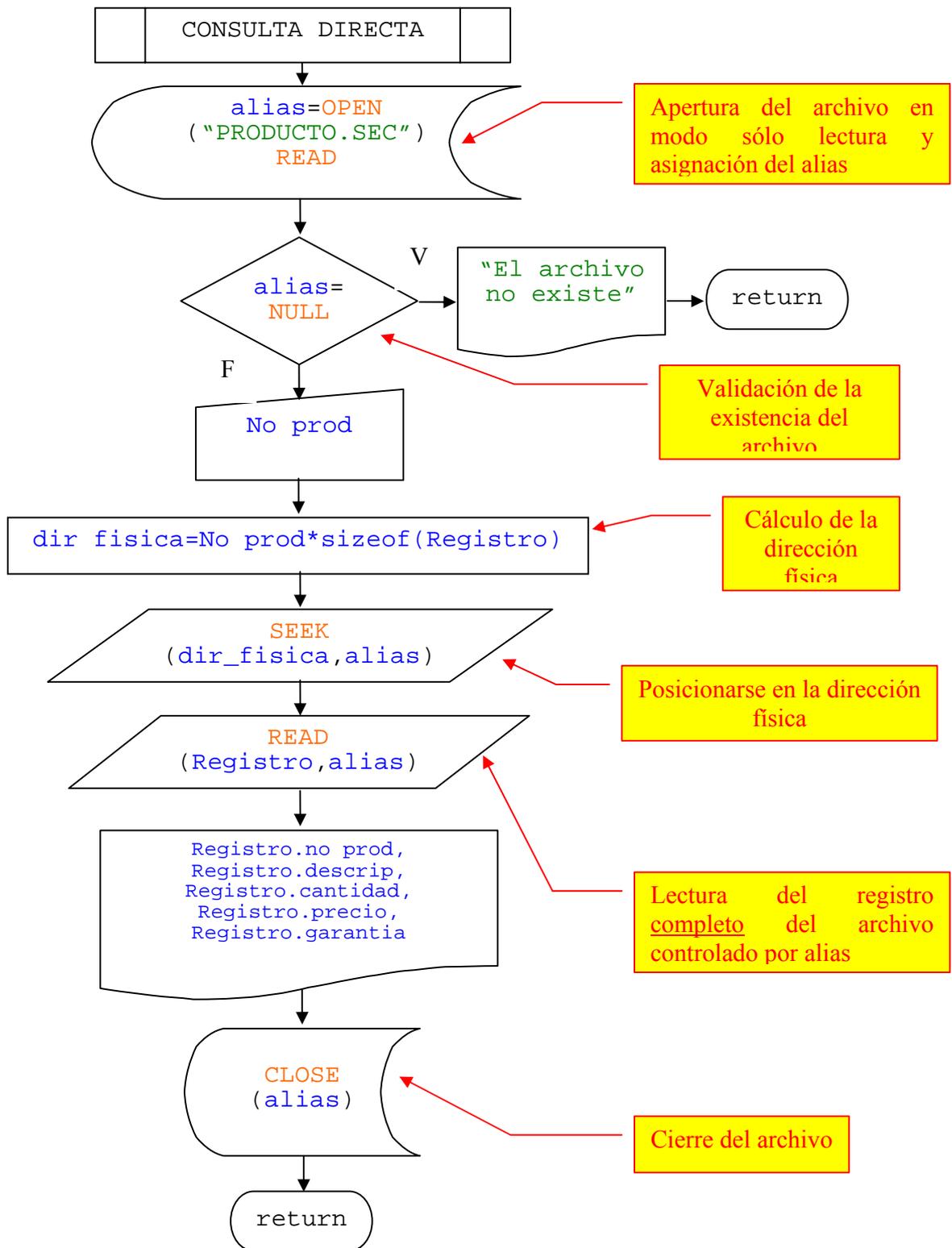


Fig. 14. Diagrama de flujo de rutina de consulta directa

2.FUNCIONES DE MANEJO DE ARCHIVOS EN LENGUAJE C++

Esta sección presenta los aspectos generales de la implementación de programas de archivos en Lenguaje C++. Aunque se puede almacenar cualquier tipo de datos en archivos, aquí se muestran las operaciones del manejo de registros (`struct`) en archivos, por lo que las funciones que se mencionan a continuación son las mas importantes para este tipo de datos.

2.1. Declaración del alias del archivo

Para realizar programas de manejo de archivos en Lenguaje C++ se requiere el encabezado “Standard I/O” y se necesita incluirlo de la sig. forma:

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

además es necesario declarar una variable de tipo `FILE` que opere como el apuntador a la estructura del archivo (alias), esto se logra con la sig. línea:

```
FILE *alias;
```

2.2. Funciones de manejo de archivos

En esta sección se presentan las funciones más importantes para el manejo y control de registros en archivos.

2.2.1 La función `fopen ()` y modos de apertura de archivos

Se usa la función `fopen` para abrir un archivo, determinar el modo de apertura y establecer la vía de comunicación mediante su alias correspondiente. Además determinar el tipo de contenido del archivo (texto o binario). Esta función tiene dos argumentos: el nombre del archivo y su modo (Fig. 16). La Fig. 15 muestra los modos de apertura de archivos de texto y binarios.

Modo de apertura (archivos de texto)	Modo de apertura (archivos binarios)	Operación
"r"	"rb"	Apertura en modo de sólo lectura. El archivo debe existir.
"w"	"wb"	Apertura en modo de sólo escritura. Si el archivo existe, se reescribirá (pierde el contenido anterior). Si el archivo no existe, lo crea.
"a"	"ab"	Apertura en modo de agregar. Si el archivo existe, los datos se agregan al final del archivo, en caso contrario, el archivo se crea.
"r+"	"rb+"	Apertura en modo de lectura/escritura. El archivo debe existir.
"w+"	"wb+"	Apertura en modo de lectura/escritura. Si el archivo existe, se reescribirá (pierde el contenido anterior).
"a+"	"ab+"	Apertura en modo de lectura/agregar. Si el archivo no existe lo crea.

Fig. 15. Modos de apertura para archivos de texto y binarios.

```
#include <stdio.h>
FILE *alias1, *alias2, *alias3;

alias1 = fopen("EJEMPLO.DAT", "rb");
alias2 = fopen("ARCHIVO.TXT", "ab");
alias3 = fopen("c:\\\\tarea\\PRODUCTO.005", "w");
```

Abre el archivo **binario** **EJEMPLO.DAT** en modo de sólo lectura y lo asigna al apuntador

Nótese que se necesitan dos '\\' ya que el backslash indica el inicio de una secuencia de escape en C++

Crea el archivo de **texto** **PRODUCTO.005** en modo de sólo escritura y lo asigna al apuntador **alias3**. El archivo lo crea en el subdirectorio c:\tarea

Fig. 16. La función fopen

2.2.2. Validar la apertura de un archivo

Algunas funciones requieren la existencia del archivo para realizar operaciones, por ello es necesario verificar que cuando se intenta abrir un archivo haya tenido éxito la operación. Si un archivo no se puede abrir, la función `fopen` devuelve el valor de 0 (cero), definido como `NULL` en `stdio.h`. La Fig. 17 muestra un ejemplo de la manera de detectar la apertura de un archivo.

```
#include <stdio.h>
FILE *alias;

alias = fopen("EJEMPLO.DAT", "rb");
if (alias==NULL)
{
    cout << "\n\r No se puede abrir el archivo !!!";
    getch();
    return();
}
```

Fig. 17. Validar la apertura de un archivo.

2.2.3. Cierre de archivos usando `fclose()` y `fcloseall()`

Antes de dejar de utilizar un archivo es necesario cerrarlo. Esto se logra mediante las funciones `fclose` o `fcloseall`. Si se usa `fclose` es necesario indicarle el alias del archivo que se desea cerrar. La función `fcloseall` cierra todos los archivos abiertos.

2.2.4. Escritura de registros usando `fwrite()`

La función `fwrite` proporciona el mecanismo para almacenar **todos** los campos de un registro en un archivo. Cabe destacar que al utilizar esta función, se almacena una variable (de tipo `struct`) que representa un bloque de datos o campos; es decir, no se almacena campo por campo. Esta función tiene cuatro argumentos: la variable que se desea grabar, su tamaño en bytes, la cantidad de variables y el alias del archivo donde se desea almacenar (Fig. 18).

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

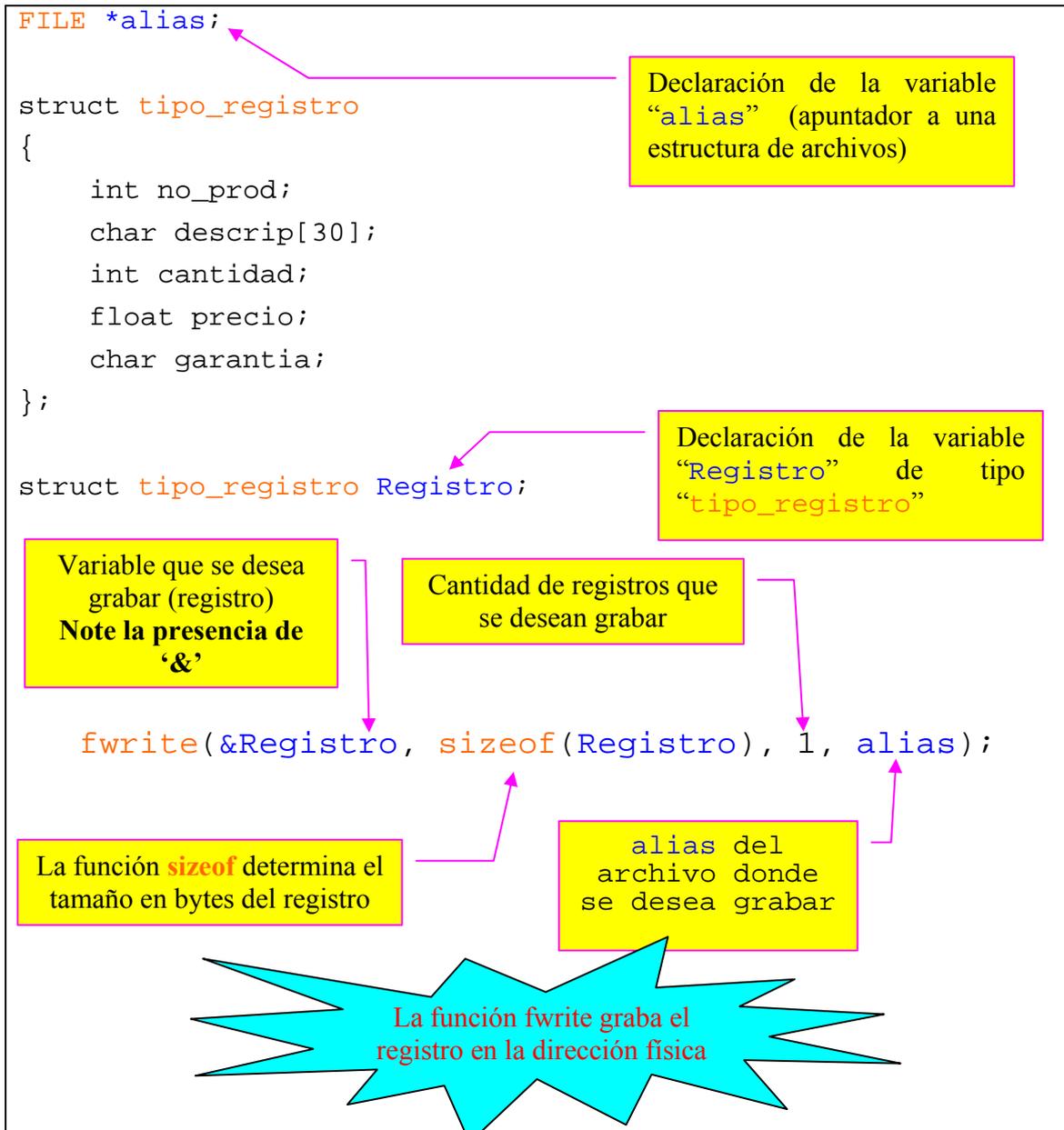


Fig. 18. La función fwrite

2.2.4.1. Vaciando los buffers con fflush()

Un buffer es un área de almacenamiento temporal en memoria para el conjunto de datos leídos o escritos en el archivo. Estos buffers retienen datos en tránsito desde y hacia al archivo y tienen la finalidad de hacer más eficiente las

operaciones de entrada/salida en los archivos de disco, provocando menor cantidad de accesos, los cuales son más lentos que la memoria. P. ejem. Si se requiere consultar constantemente un dato del archivo, no es necesario calcular su dirección física, reposicionar el apuntador del archivo, “cargar” el dato en memoria mediante una operación de lectura cada vez que se necesita, sino que el sistema operativo controla y mantiene este dato en los buffers de memoria y de ahí lo toma cuando lo requiera. Sólo hay una consideración importante al utilizar los buffers, los datos escritos en ellos no se reflejan exactamente en los archivos de disco en forma inmediata, sino hasta que se “vacía” el buffer. Para ello se utiliza la función `fflush` y basta enviarle el alias del archivo como argumento. Los buffers también se vacían cuando se cierra el archivo.

2.2.5. Lectura de registros usando `fread()`

La función `fread` permite “cargar” **todos** los campos de un registro en un archivo, es decir, lee un registro y lo copia en la memoria RAM (Fig. 1). Esta función tiene los mismos argumentos que la función `fwrite` (Fig. 18).

2.2.6. Reposicionando el apuntador mediante `fseek()`

Para comprender la operación de esta función es necesario estar relacionado con el apuntador del archivo, el cual indica la posición dentro del archivo. Cuando se abre un archivo en modo de sólo lectura, sólo escritura o lectura/escritura, el apuntador del archivo se posiciona al inicio del mismo y cuando un archivo se abre en modo agregar se posiciona al final, sin embargo, se puede reposicionar este apuntador del archivo mediante la función `fseek`. También es importante mencionar que cada vez que se realiza una operación de lectura o de escritura de cualquier tipo de datos (caracter, cadena, estructura, etc.), el apuntador del archivo se mueve al final de dicho dato, de tal forma que está posicionado en el

siguiente, por lo que es muy importante asegurarse que se encuentre en la posición deseada antes de realizar cualquier operación.

Como se mencionó en las secciones 1.7.2.1. y 1.7.2.2., los archivos en lenguaje C++ son controlados por direcciones físicas (no lógicas) indicadas en bytes y la función `fseek` permite reposicionar el apuntador del archivo en la dirección física deseada mediante tres argumentos: el alias del archivo, la dirección física (en bytes) y el punto de referencia. Para poder reposicionar el apuntador del archivo es necesario que éste se encuentre abierto y se le haya asignado el **alias** correspondiente, también es necesario calcular la **dirección física** donde se desea colocar (ver sección 1.7.2.2) e indicarle el **punto de referencia** de donde se partirá en el conteo de la cantidad de bytes indicado por la dirección física, los cuales pueden ser desde el principio, desde donde se encuentre el apuntador del archivo y desde el final. La Fig. 20 ilustra mejor esto.

2.2.6.1. Puntos de referencia de la función `fseek()`

El último argumento de la función `fseek` es conocido como el punto de referencia o el modo y se refiere desde dónde se iniciará el conteo de bytes determinado por la dirección física



Modo	Nombre	Operación
0	SEEK_SET	Desde el principio del archivo
1	SEEK_CUR	Desde la posición actual del apuntador del archivo
2	SEEK_END	Desde el final del archivo

Fig. 19. Puntos de referencia de la función `fseek`



Fig. 20. La función `fseek`

2.2.6.2. Conociendo la posición del apuntador del archivo con la función `ftell()`

Se usa la función `ftell` para conocer la posición actual del apuntador de un archivo abierto. La posición se expresa en bytes (dirección física) contados desde el principio del archivo (Fig. 21).

```

long int dir_fisica;

dir_fisica = ftell(alias);
    
```

alias del archivo

La variable `dir_fisica` debe ser declarada como `long int`

Fig. 21. La función `ftell`

2.2.6.3. Colocando el apuntador del archivo al principio con la función `rewind()`

Se usa la función `rewind` para colocar el apuntador del archivo al principio de un archivo abierto sin necesidad de usar la función `fseek`. Basta con enviar el alias del archivo como argumento (Fig. 22).

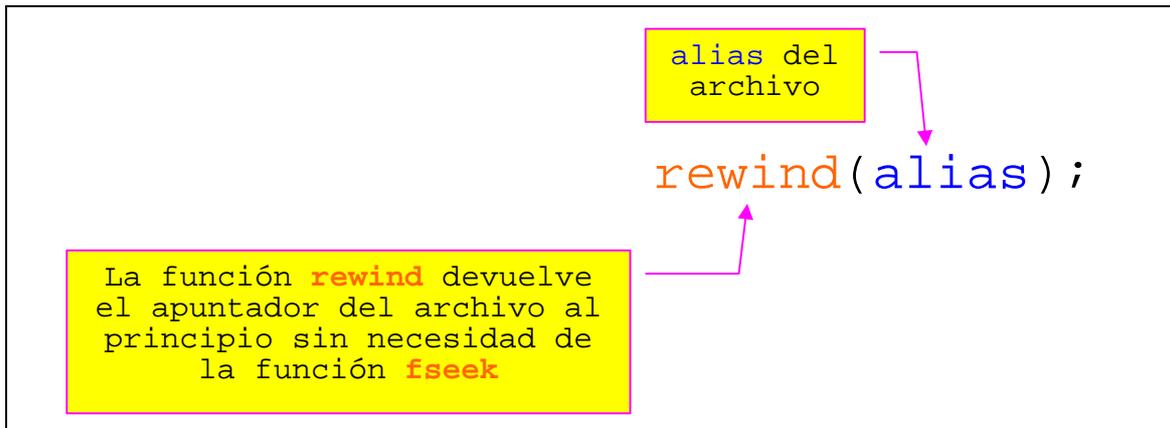


Fig. 22. La función *rewind*

2.2.7. Detectando el final del archivo con `feof()`

Se usa el macro `feof()` (definido en `stdio.h`) para determinar si se ha encontrado el final de un archivo. Si se encuentra el final de un archivo, devuelve un valor diferente de cero y cero en caso contrario. Para invocarlo es necesario colocar el alias del archivo abierto como argumento. Se utiliza mucho esta función en recorridos de tipo secuencial (Ver sección 1.7.1).

2.2.8. Cambiando nombres de archivos mediante

`rename()`

Esta función tiene como objetivo cambiar el nombre de un archivo o subdirectorios especificado por su ruta de acceso. Sólo necesita dos argumentos: el nombre anterior del archivo y el nuevo nombre. **Es importante destacar que esta función sólo puede aplicarse a archivos cerrados.**

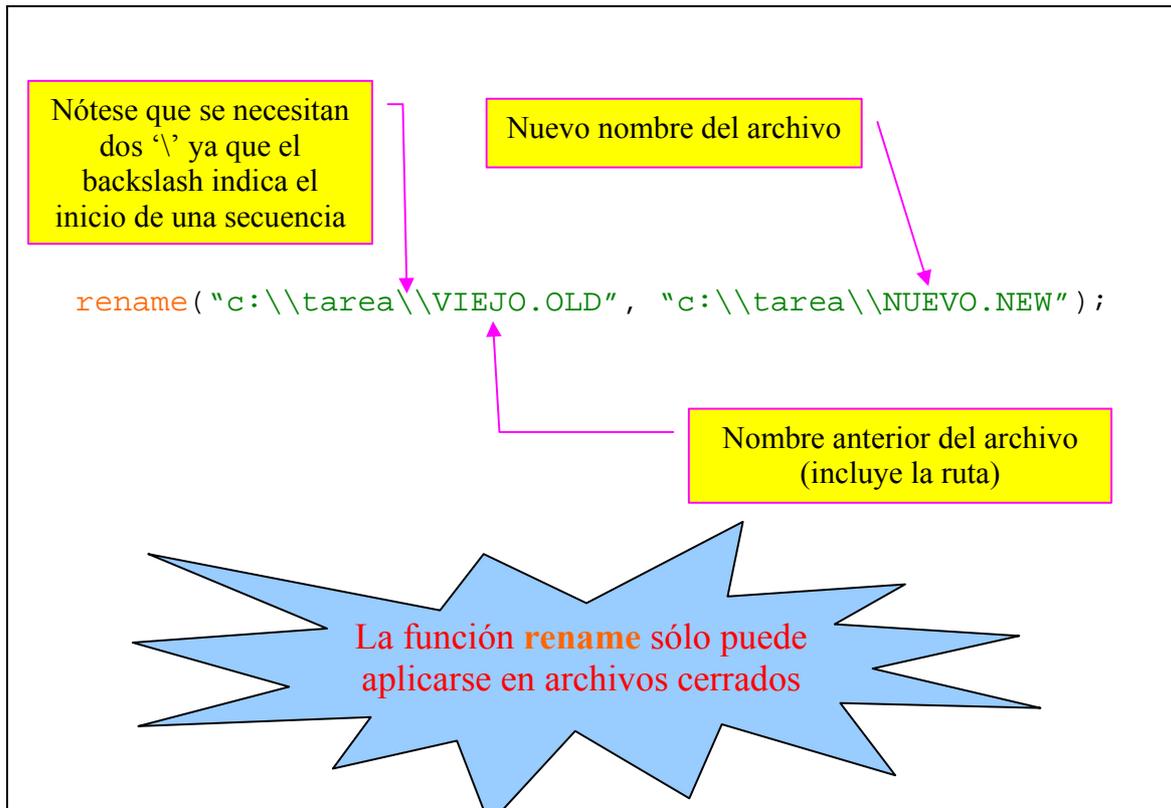


Fig. 23. La función rename

2.2.9. Eliminando archivos con la función remove ()

La función remove elimina definitivamente un archivo especificando su nombre (Fig. 24). **Es importante destacar que esta función sólo puede aplicarse a archivos cerrados.**

Nótese que se necesitan dos ‘\’ ya que el backslash indica el inicio de una secuencia

Nombre del archivo que se desea eliminar

```
remove( "c:\\tarea\\VIEJO.OLD" );
```

La función remove sólo puede aplicarse en archivos cerrados

The diagram shows a C++ code snippet for removing a file. Two yellow boxes with arrows point to the backslashes in the file path string. The first box explains that two backslashes are needed because a single backslash is an escape character. The second box identifies the file name. A blue starburst contains a warning that the remove function only works on closed files.

Fig. 24. La función remove

3. APLICACIONES DE ARCHIVOS EN C++

A continuación se presentan algunas de las aplicaciones prácticas más comunes con archivos. Están basadas en los aspectos teóricos analizados en la sección 1 y en las funciones de manejo de archivos en lenguaje C++ de la segunda sección. Aquí se analizarán los procedimientos de inserción, consulta y eliminación de registros tanto en archivos secuenciales como directos, así como el listado secuencial. En ambos tipos de archivos se tratarán como archivos binarios.

Para apoyar el análisis, diseño e implementación de estas rutinas, se tomará el ejemplo del control de PRODUCTOS que se mencionó anteriormente así como la definición del registro de la Fig. 3.

En cada procedimiento se muestra el diagrama de flujo y la codificación íntegra en lenguaje C++.

3.1. Declaraciones globales

Antes de analizar, diseñar e implementar las rutinas es necesario establecer las consideraciones iniciales tales como el uso de los encabezados (*.h), las declaraciones globales del registro de productos y el alias correspondiente. La fig. 25 muestra las declaraciones globales necesarias para estas aplicaciones. Para efectos de identificación, se usará el nombre **“PRODUCTO.SEC”** para el archivo secuencial y el nombre **“PRODUCTO.DIR”** para el archivo directo.

```

#include <stdio.h> // Para el manejo de archivos
#include <string.h> // Para el manejo de cadenas
#include <conio.h> // Para el manejo de clrscr
#include <iostream.h> // Para el manejo de cout
#include <ctype.h> // Para el uso de toupper

struct tipo_registro
{
    int no_prod;
    char descrip[30];
    int cantidad;
    float precio;
    char garantia;
};

struct tipo_registro Registro;

FILE *alias;

void main(void)
{
    .....
}

```

Fig. 25. Declaraciones globales de las aplicaciones

3.2. Archivos Secuenciales en Lenguaje C++

En esta sección se analizará la manera de diseñar rutinas que manipulen registros de productos o artículos en un archivo secuencial. Como su nombre lo indica, en este tipo de archivo se hace un recorrido secuencial para localizar la dirección del registro solicitado, es decir, se lee registro por registro hasta llegar al deseado.

3.2.1. Altas Secuenciales

Aquí se presenta una rutina que inserta registros de productos en un archivo secuencial. Se considera un número de producto (campo `no_prod`) que servirá como referencia para identificarlo y diferenciarlo de otros productos y no se permite que se graben dos productos diferentes con el mismo número, por lo que es necesario realizar un recorrido secuencial completo del archivo para asegurarse que no haya duplicados.

La primera ocasión que se intente insertar registros en un archivo, éste debe crearse; sin embargo **debe cuidarse no crear el archivo cada vez que se invoque esta rutina** porque debe tenerse presente que si se crea un archivo existente, se pierde su contenido anterior.

3.2.1.1. Diagrama de flujo de la rutina de Altas Secuenciales

La Fig. 26 ilustra el diagrama de flujo de la rutina de altas en un archivo secuencial.

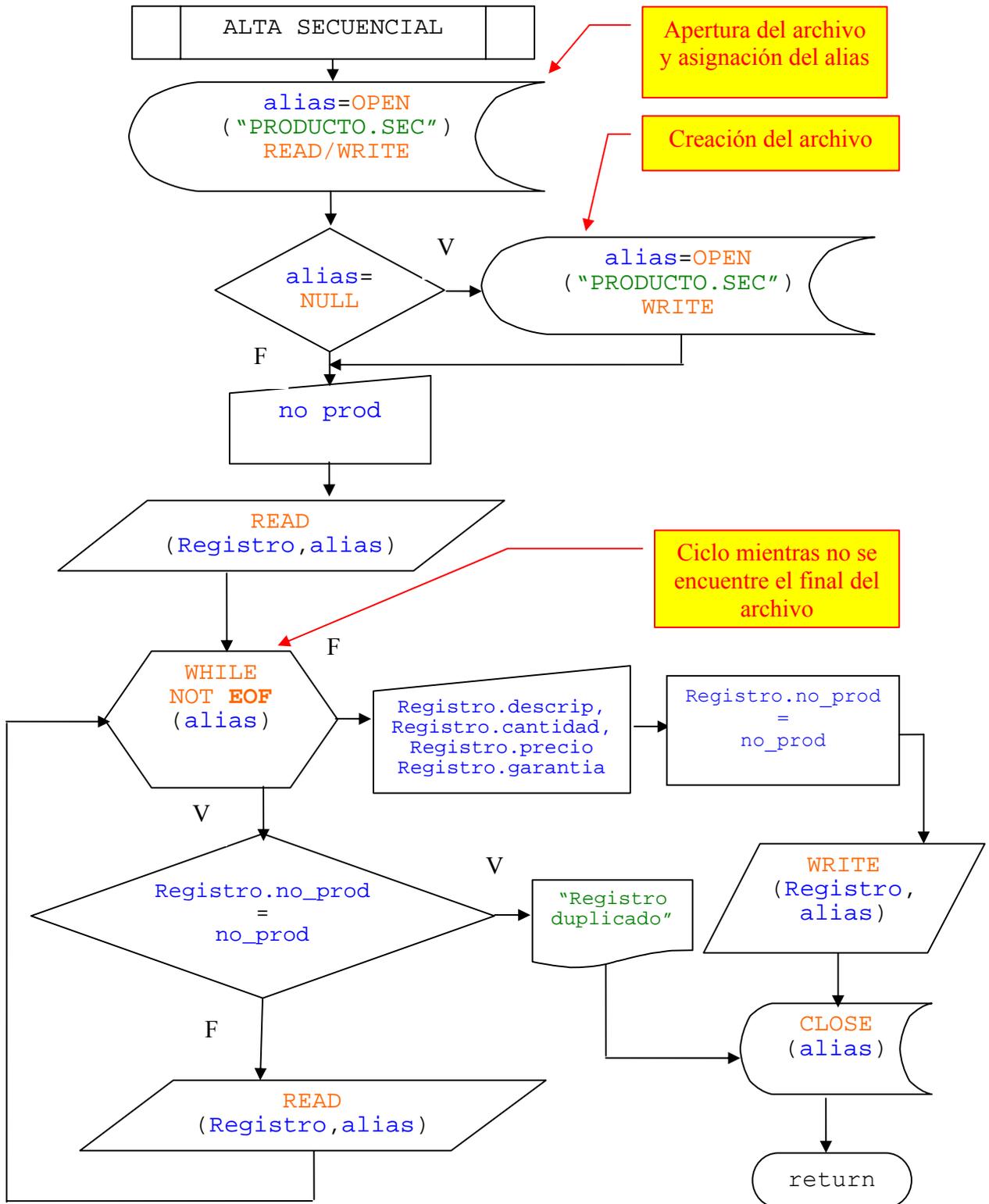


Fig. 26. Diagrama de flujo de rutina de alta secuencial

3.2.1.2. Codificación de la rutina de ALTAS secuenciales

La Fig. 27 muestra el código de la rutina de ALTAS secuenciales de acuerdo al diagrama de la Fig. 26.

```

void ALTA_SECUENCIAL(void)
{
    int no_prod; // Variable local para el numero de producto
    clrscr();
    cout << "\n\rALTAS DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.SEC","rb+"); // Intenta abrir el archivo
                                        // en modo de lectura/escritura
    if(alias==NULL)
        alias=fopen("PRODUCTO.SEC","wb"); // Crea el archivo en caso de no
                                            // existir
    cout << "\n\n\rNumero de producto: "; cin >> no_prod;

    fread(&Registro,sizeof(Registro),1,alias);
    // Lee el "Registro", de tamaño=sizeof(Registro) del archivo "alias"

    while(!feof(alias)) // Ciclo mientras no se encuentre el final del
                        // archivo
    {
        if(Registro.no_prod==no_prod)
        {
            cout << "\n\n\rRegistro duplicado !!!";
            fclose(alias);
            getch();
            return;
        }
        fread(&Registro,sizeof(Registro),1,alias);
    }
    cout << "\n\rDescripcion: "; gets(Registro.descripcion);
    cout << "\n\rCantidad : "; cin >> Registro.cantidad;
    cout << "\n\rPrecio : "; cin >> Registro.precio;
    do
    {
        cout << "\n\rGarantia : "; Registro.garantia=toupper(getche());
    }while(Registro.garantia!='S' && Registro.garantia!='N');
    Registro.no_prod=no_prod;

    fwrite(&Registro,sizeof(Registro),1,alias); // Grabar el Registro
    fclose(alias); // Cierra el archivo

    cout << "\n\n\rProducto registrado !!!";
    cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
    getch();
    return;
}

```

Fig. 27. Codificación de la rutina de altas secuenciales

3.2.2. CONSULTAS secuenciales

En esta sección se analiza una rutina que busca un registro particular de un producto en un archivo secuencial. En este tipo de archivo se inicia el recorrido desde el primer registro almacenado y se detiene cuando se localiza el registro solicitado o cuando se encuentre el final del archivo.

3.2.2.1. Diagrama de flujo de la rutina de CONSULTAS secuenciales

El diagrama de flujo de esta rutina se planteó en la sección 1.7.1.1 y se muestra en la Fig. 9.

3.3.2.2. Codificación de la rutina de CONSULTAS secuenciales

La Fig. 28 muestra el código de la rutina de CONSULTAS secuenciales de acuerdo al diagrama de la Fig. 7.

```
void CONSULTA_SECUENCIAL(void)
{
    int no_prod;
    clrscr();

    cout << "\n\rCONSULTA DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.SEC","rb"); // Intenta abrir el archivo
PRODUCTO.SEC
                                     // en modo de solo lectura
    if(alias==NULL)
    {
        cout << "\n\n\n\rNo existe el archivo !!!";
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }

    cout << "\n\n\n\rNumero de producto: "; cin >> no_prod;

    fread(&Registro,sizeof(Registro),1,alias);
```

```

// Lee el "Registro", de tamaño=sizeof(Registro) del archivo "alias"
while(!feof(alias))
{
    if(Registro.no_prod==no_prod)
    {
        cout << "\n\rNo Prod                Descripcion  Cantidad
Precio          Garantia";
        cout << "\n\r-----"
-----";

printf("\n\r%3d\t%30s\t%3d\t\t$%4.2f\t%c",Registro.no_prod,Registro.descr
ip,
Registro.cantidad,Registro.precio,Registro.garantia);
        fclose(alias);
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }
    fread(&Registro,sizeof(Registro),1,alias);
}
cout << "\n\rNo se encuentra ese registro !!!";
fclose(alias); // Cierra el archivo
cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
getch();
return;
}

```

Fig. 28. Codificación de la rutina de consultas secuenciales

3.2.3. LISTADO secuencial

Esta rutina es muy semejante a la de CONSULTAS secuenciales, solo que en el LISTADO se despliegan todos los registros lógicos que contiene el archivo. En este tipo de archivo se inicia el recorrido desde el primer registro almacenado y se detiene cuando se encuentre el final del archivo.

3.2.3.1. Diagrama de flujo de la rutina de LISTADO secuencial

El diagrama de flujo de esta rutina se muestra en la Fig. 29.

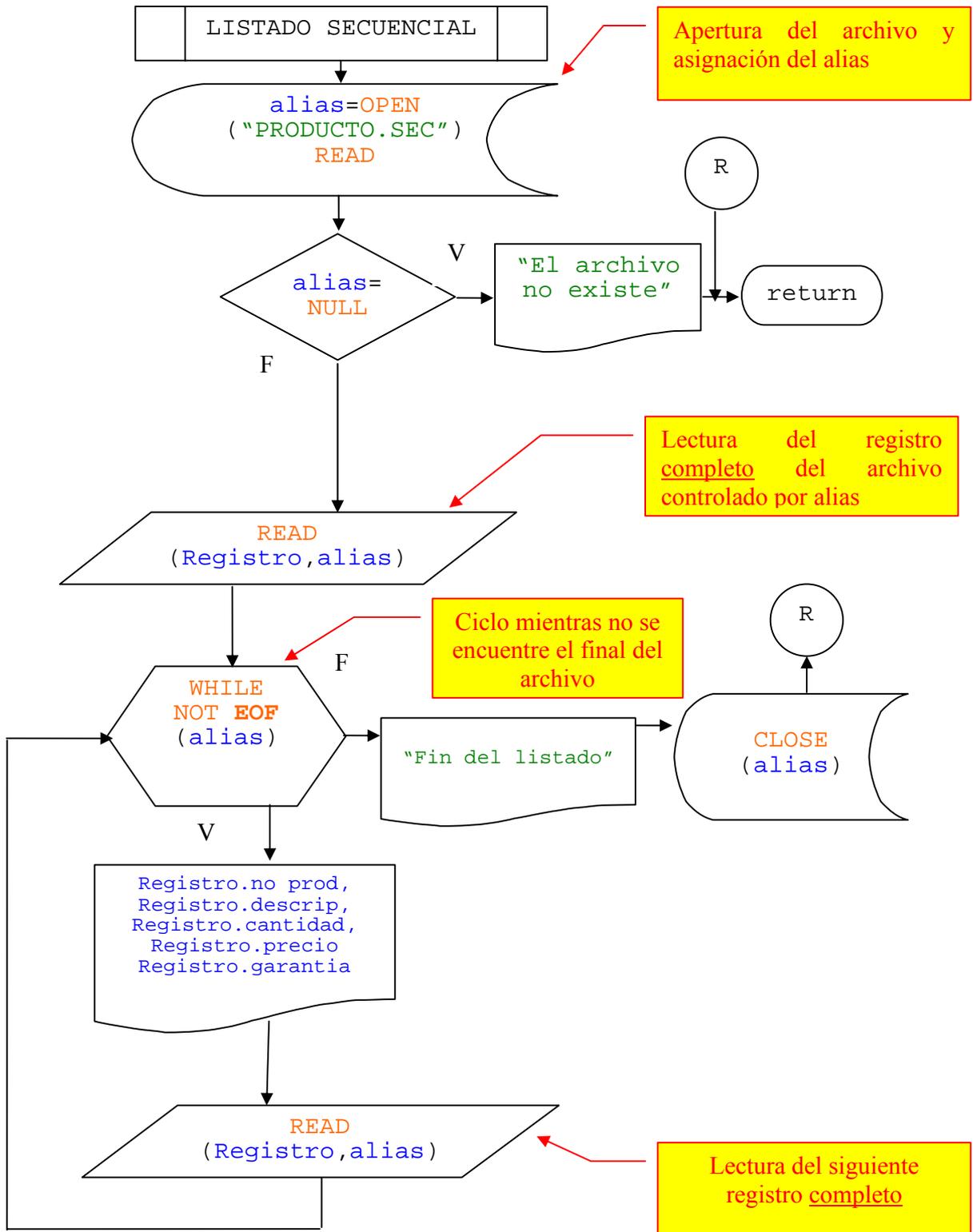


Fig. 29. Diagrama de flujo de rutina de listado secuencial

3.3.3.2. Codificación de la rutina de LISTADO secuencial

La Fig. 30 muestra el código de la rutina de LISTADO secuencial de acuerdo al diagrama de la Fig. 29.

```

void LISTADO_SECUENCIAL(void)
{
    clrscr();

    cout << "\n\rLISTADO DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.SEC","rb"); // Intenta abrir el archivo
PRODUCTO.SEC
                                     // en modo de solo lectura

    if(alias==NULL)
    {
        cout << "\n\n\n\rNo existe el archivo !!!";
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }

    cout << "\n\rNo Prod                Descripcion  Cantidad
Precio  Garantia";
    cout << "\n\r-----
-----";

    fread(&Registro,sizeof(Registro),1,alias);
    // Lee el "Registro", de tamaño=sizeof(Registro) del archivo "alias"
    while(!feof(alias)) // Ciclo mientras no se encuentre el final del
archivo
    {

printf("\n\r%3d\t%30s\t%3d\t\t%$4.2f\t%c",Registro.no_prod,Registro.descr
ip,
Registro.cantidad,Registro.precio,Registro.garantia);
        fread(&Registro,sizeof(Registro),1,alias);
    }
    fclose(alias); // Cierra el archivo
    cout << "\n\r-----
-----";

    cout << "\n\rFin del listado !!!";
    cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
    getch();
    return;
}

```

Fig. 30. Codificación de la rutina de listado secuencial

3.2.4. MODIFICACIONES de datos en un archivo secuencial

La forma de modificar el contenido de los campos de registros de un archivo, depende mucho de la rutina de consulta, ya que es necesario localizar previamente el registro que se desea modificar, capturar los nuevos valores y posteriormente **grabar el registro completo en la misma posición que se encontraba.** Esto último es muy importante porque recuerde que cuando se termina de leer un registro del archivo, el apuntador se posiciona al inicio del siguiente registro, por lo que, antes de grabar el registro modificado, es necesario reposicionar el apuntador del archivo en la dirección correcta.

3.2.4.1. Diagrama de flujo de la rutina de MODIFICACIÓN secuencial

La Fig. 31 muestra el diagrama de flujo de la rutina de modificaciones de campos de un registro particular en un archivo secuencial.

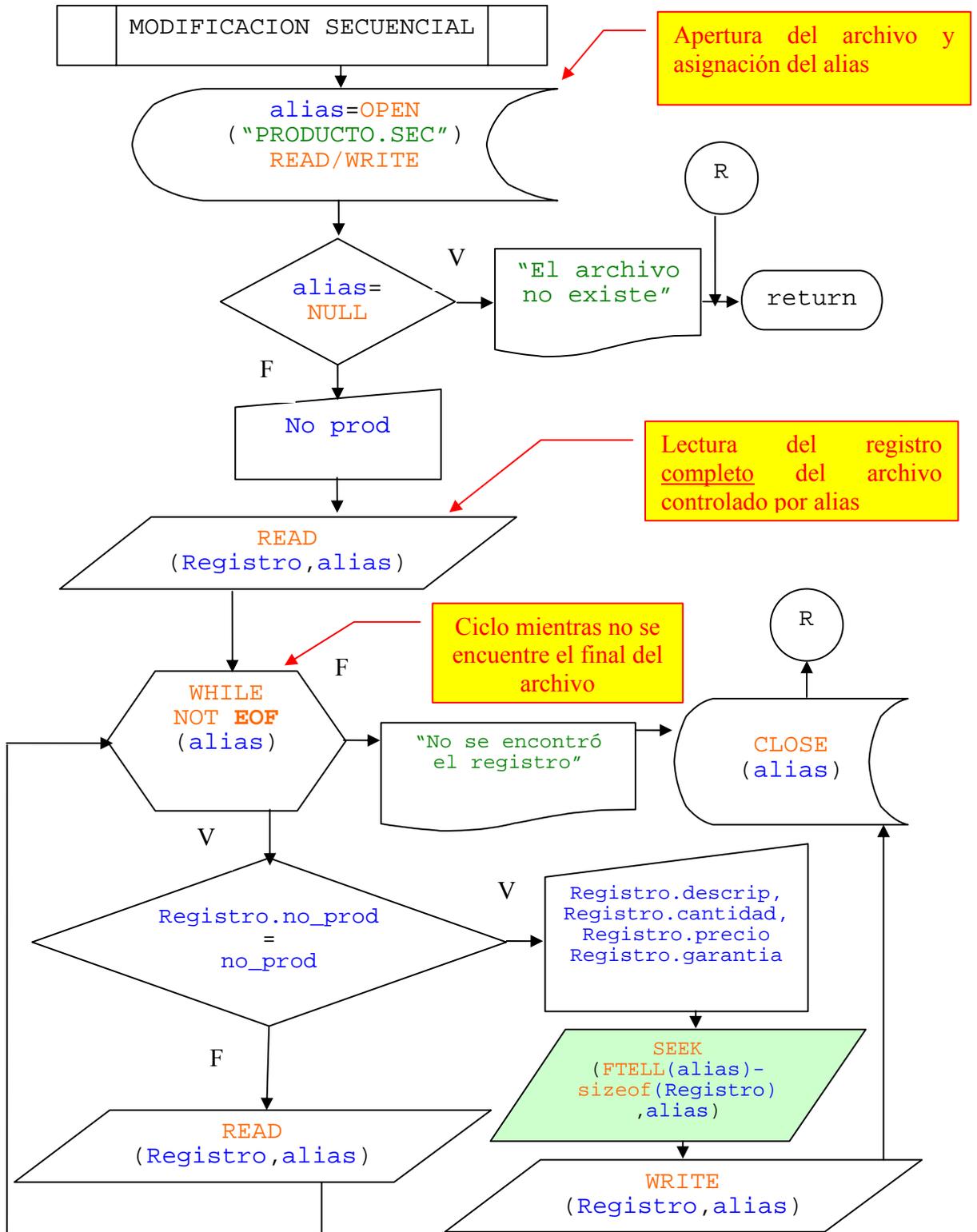


Fig. 31. Diagrama de flujo de rutina de modificación secuencial

3.2.4.2. Codificación de la rutina de MODIFICACIÓN secuencial

La Fig. 32 muestra la codificación de la rutina de modificaciones de campos de un registro particular en un archivo secuencial.

```

void MODIFICACION_SECUENCIAL(void)
{
    int no_prod; // Variable local para el numero de producto que desea
    modificar
    clrscr();

    cout << "\n\rMODIFICACION DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.SEC","rb+"); // Intenta abrir el archivo
    PRODUCTO.SEC
                                // en modo de lectura/escritura
    if(alias==NULL) // Valida la existencia del archivo
    {
        cout << "\n\n\n\rNo existe el archivo !!!";
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }

    cout << "\n\n\n\rNumero de producto: "; cin >> no_prod;

    fread(&Registro,sizeof(Registro),1,alias);
    // Lee el "Registro", de tamaño=sizeof(Registro) del archivo "alias"
    while(!feof(alias)) // Ciclo mientras no se encuentre el final del
    archivo
    {
        if(Registro.no_prod==no_prod)
        {
            cout << "\n\rNo Prod                Descripcion  Cantidad
Precio  Garantia";
            cout << "\n\r-----
-----";

            printf("\n\r%3d\t%30s\t%3d\t\t\t$%.2f\t%c",Registro.no_prod,Registro.descripcion,
Registro.cantidad,Registro.precio,Registro.garantia);

            cout << "\n\n\n\rAnote los nuevos datos ...";
            cout << "\n\rDescripcion: "; gets(Registro.descripcion);
            cout << "\n\rCantidad    : "; cin >> Registro.cantidad;
            cout << "\n\rPrecio      : "; cin >> Registro.precio;
            do
            {
                cout << "\n\rGarantia    : "; Registro.garantia=toupper(getche());
            }while(Registro.garantia!='S' && Registro.garantia!='N');

            // Es necesario reposicionar el apuntador del archivo al principio

```

```

del
    // registro que desea modificar, ya que al leer un registro, el
    // apuntador se posiciona en el registro siguiente
    // La funcion ftell(alias) devuelve la posicion donde se encuentra
el
    // apuntador
    fseek(alias,ftell(alias)-sizeof(Registro),SEEK_SET);
    fwrite(&Registro,sizeof(Registro),1,alias); // Graba el registro
con
    // los nuevos campos
    fclose(alias); // Cierra el archivo
    cout << "\n\n\n\rRegistro modificado !!!";
    cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
    getch();
    return;
}
    fread(&Registro,sizeof(Registro),1,alias);
}
    cout << "\n\rNo se encuentra ese registro !!!";
    fclose(alias); // Cierra el archivo
    cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
    getch();
    return;
}

```

Fig. 32. Codificación de rutina de modificación secuencial

3.2.5. Bajas de registros en un archivo secuencial (bajas lógicas y bajas físicas)

Básicamente existen dos formas de eliminar registros en un archivo: lógica y físicamente. La diferencia radica en que cuando se elimina un registro en forma lógica de un archivo, sólo se le coloca una marca especial en alguno de sus campos que lo identifique como registro “borrado”, sin embargo, el registro sigue existiendo en el archivo y por lo tanto ocupa espacio. En cambio en la baja física de registros, se elabora una rutina que elimine físicamente el registro del archivo, es decir, que el archivo solo conserve los registros válidos. Las bajas físicas también son conocidas como rutinas de compactación o compresión del archivo.

3.2.5.1. Diagrama de flujo de la rutina de BAJAS lógicas en un archivo secuencial

Como se mencionó en el punto anterior, las bajas lógicas consisten en “marcar” los registros eliminados. En el ejemplo práctico que se muestra a continuación, el registro borrado se “limpia”, dejando en blanco todos sus campos (colocando el valor cero en los campos numéricos y blancos en las cadenas o de tipo carácter).

Para fines prácticos, la rutina de bajas lógicas se asemeja mucho a la rutina de modificaciones, sólo que en las bajas no se capturan los nuevos valores, sino se les asigna valores en blanco y se graba el registro completo en la misma posición en la que se encontraba. El diagrama de la Fig. 33 muestra esta rutina.

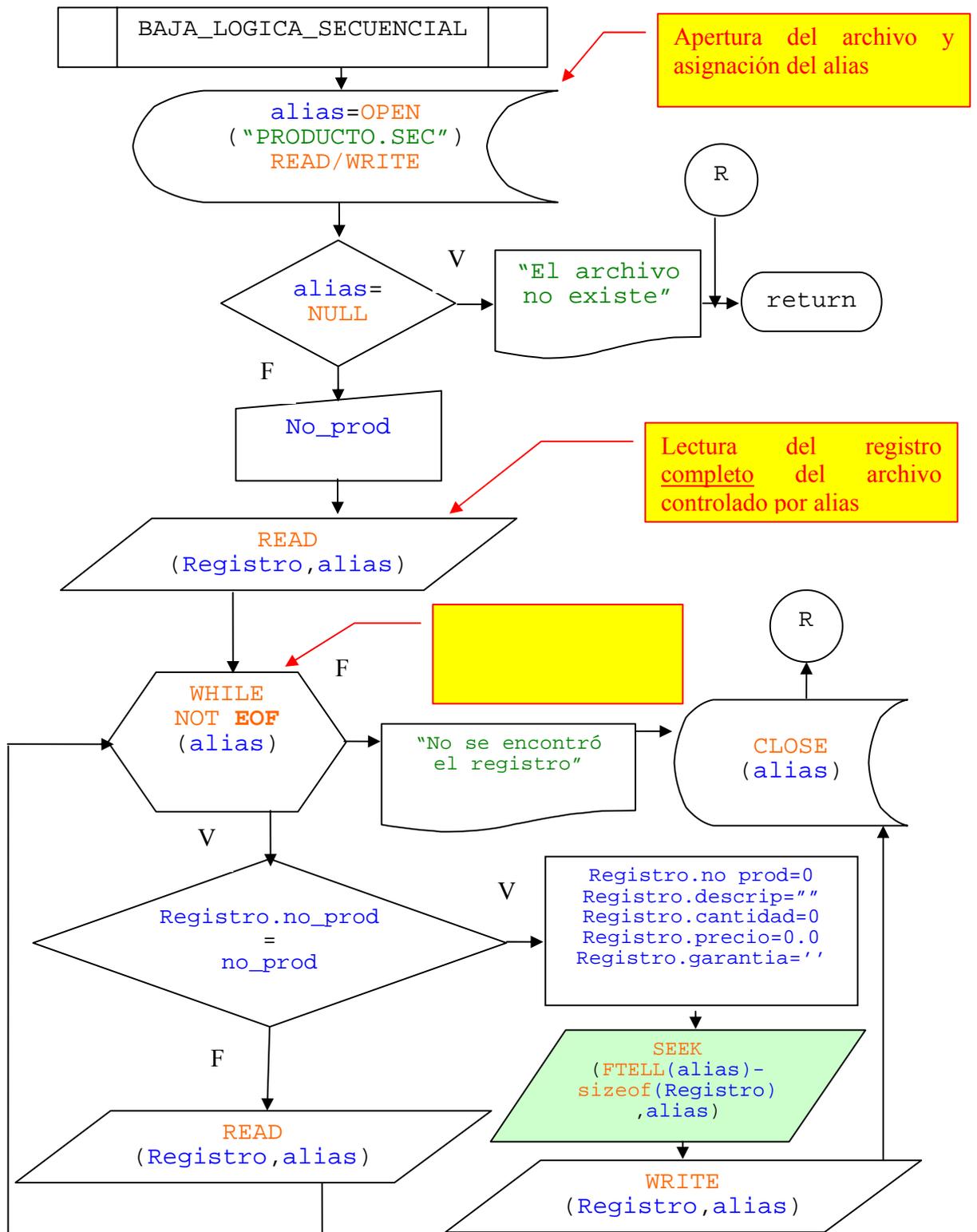


Fig. 33. Diagrama de flujo de rutina de baja lógica secuencial

3.2.5.2. Codificación de la rutina de BAJAS lógicas en un archivo secuencial

La Fig. 34 muestra la codificación íntegra de esta rutina.

```

void BAJA_LOGICA_SECUENCIAL(void)
{
    int no_prod; // Variable local para el numero de producto que desea
eliminar
    char op; //Variable local
    clrscr();

    cout << "\n\rBAJAS LOGICAS DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.SEC","rb+"); // Intenta abrir el archivo
PRODUCTO.SEC
                                // en modo de lectura/escritura
    if(alias==NULL) // Valida la existencia del archivo
    {
        cout << "\n\n\n\rNo existe el archivo !!!";
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }

    cout << "\n\n\n\rNumero de producto: "; cin >> no_prod;

    fread(&Registro,sizeof(Registro),1,alias);
    // Lee el "Registro", de tamano=sizeof(Registro) del archivo "alias"
    while(!feof(alias)) // Ciclo mientras no se encuentre el final del
archivo
    {
        if(Registro.no_prod==no_prod)
        {
            cout << "\n\rNo Prod                                Descripcion  Cantidad
Precio  Garantia";
            cout << "\n\r-----";
            -----";

printf("\n\r%3d\t%30s\t%3d\t\t$%4.2f\t%c",Registro.no_prod,Registro.descr
ip,Registro.cantidad,Registro.precio,Registro.garantia);

            Registro.no_prod=0;
            strcpy(Registro.descripcion, "");
            Registro.cantidad=0;
            Registro.precio=0.0;
            Registro.garantia=' ';

            do {
                cout << "\n\n\rEsta seguro que desea borrarlo? [S/N] ---> ";
                op=toupper(getche());
            }while(op!='S' && op!='N');

            if(op=='S')

```

```

{
// Es necesario repositionar el apuntador del archivo al principio
del
// registro que desea modificar, ya que al leer un registro, el
// apuntador se posiciona en el registro siguiente
// La funcion ftell(alias) devuelve la posicion donde se encuentra
el
// apuntador
fseek(alias,ftell(alias)-sizeof(Registro),SEEK_SET);
fwrite(&Registro,sizeof(Registro),1,alias); // Graba el registro
con
// los nuevos campos
cout << "\n\n\n\rRegistro eliminado !!!";
}
fclose(alias); // Cierra el archivo
cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
getch();
return;
}
fread(&Registro,sizeof(Registro),1,alias);
}
cout << "\n\rNo se encuentra ese registro !!!";
fclose(alias); // Cierra el archivo
cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
getch();
return;
}
}

```

Fig. 34. Codificación de rutina de baja lógica secuencial

3.2.5.3. Diagrama de flujo de la rutina de BAJAS físicas en un archivo secuencial (compactar)

La rutina de bajas físicas (también conocida como compactar el archivo), consiste en eliminar definitivamente los espacios dejados por los registros borrados lógicamente. Cuando se elimina un registro en forma lógica dejando en blanco sus campos, sigue ocupando espacio en el archivo, sin embargo se puede diseñar una rutina que se apoye de un archivo auxiliar (también secuencial), en el cual se graben todos los registros válidos, es decir, los registros no eliminados, para posteriormente eliminar el **archivo** original y renombrar este archivo auxiliar como el archivo original. La Fig. 35 muestra este diagrama de flujo.

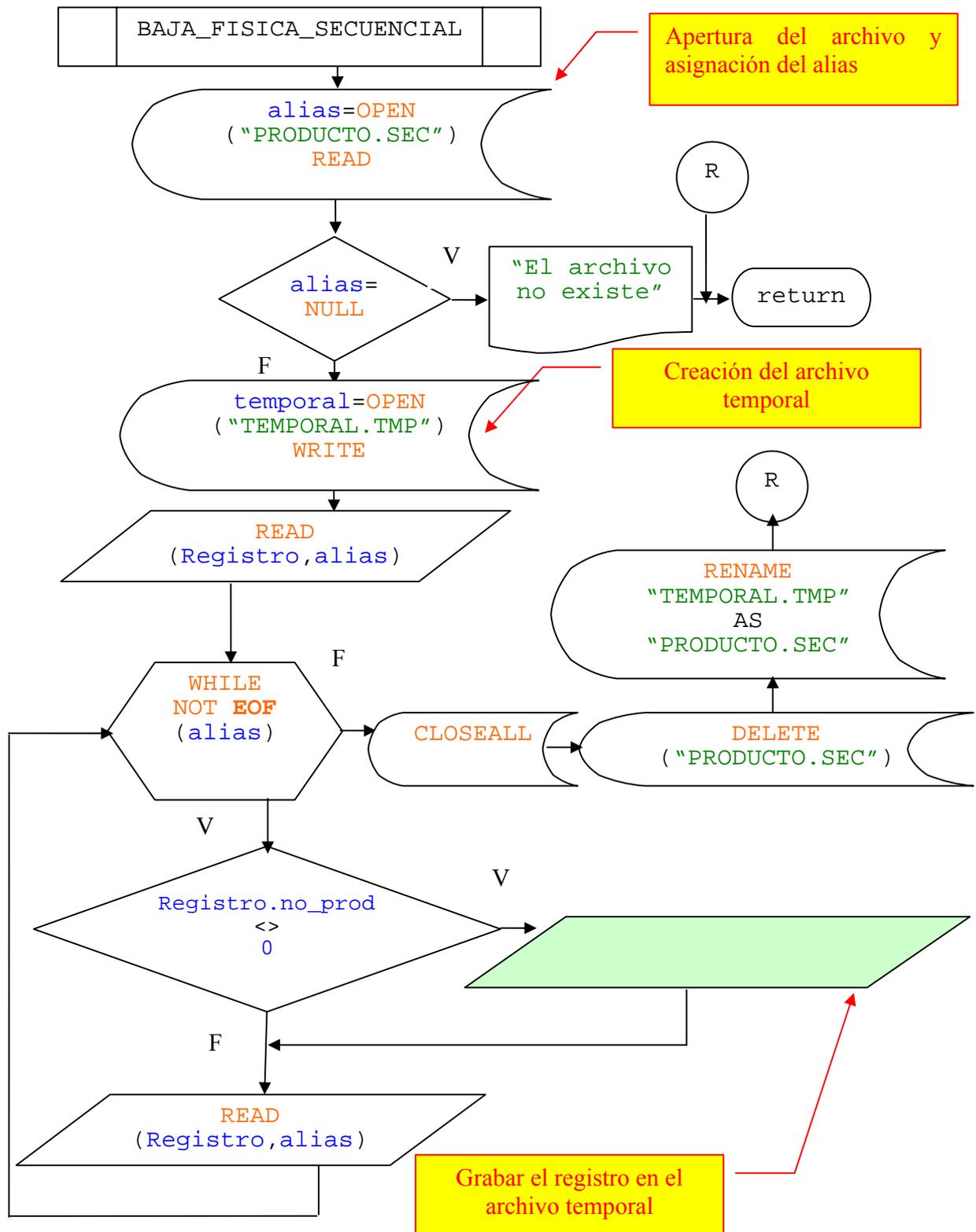


Fig. 35. Diagrama de flujo de rutina de baja física secuencial (compactar)

3.2.5.4. Codificación de la rutina de BAJAS físicas en un archivo secuencial (compactar)

La Fig. 36 muestra la codificación íntegra de esta rutina.

```
void BAJA_FISICA_SECUENCIAL(void)
{
FILE *temporal; //Declaracion local de una variable para controlar el
                // archivo temporal
    clrscr();

    cout << "\n\rBAJAS FISICAS DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.SEC","rb"); // Intenta abrir el archivo
PRODUCTO.SEC
                                // en modo de solo lectura
    if(alias==NULL) // Valida la existencia del archivo
    {
        cout << "\n\n\n\rNo existe el archivo !!!";
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }

    temporal=fopen("TEMPORAL.TMP","wb"); // Crea el archivo TEMPORAL.TMP

    fread(&Registro,sizeof(Registro),1,alias);
        // Lee el "Registro", de tamaño=sizeof(Registro) del archivo "alias"
    while(!feof(alias)) // Ciclo mientras no se encuentre el final del
archivo
    {
        if(Registro.no_prod!=0)
            fwrite(&Registro,sizeof(Registro),1,temporal);
            // Graba el registro valido en el archivo temporal

        fread(&Registro,sizeof(Registro),1,alias);
    }
    fcloseall(); // Cierra todos los archivos abiertos
    remove("PRODUCTO.SEC"); //Elimina el archivo original
    rename("TEMPORAL.TMP","PRODUCTO.SEC");
    //Renombra el archivo temporal con el nombre del archivo original

    cout << "\n\n\n\rArchivo compactado !!!";
    cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
    getch();
    return;
}
```

Fig. 36. Codificación de rutina de baja física secuencial (compactar)

3.3. Archivos directos en Lenguaje C++

En esta sección se analizará la manera de diseñar rutinas que manipulen registros de productos o artículos en un archivo directo. A diferencia del archivo secuencial en el que se hace un recorrido secuencial para localizar la dirección del registro solicitado, en el archivo de acceso directo, se calcula la dirección física y se posiciona el apuntador del archivo directamente en el registro solicitado usando la función `fseek`.

3.3.1. Altas Directas

Aquí se presenta una rutina que inserta registros de productos en un archivo directo. Se tomará el mismo ejemplo del registro de producto que se usó en el archivo secuencial. En este caso se nombrará el archivo como **“PRODUCTO.DIR”**

La primera ocasión que se intente insertar registros en un archivo, éste debe crearse; sin embargo **debe cuidarse no crear el archivo cada vez que se invoque esta rutina** porque debe tenerse presente que si se crea un archivo existente, se pierde su contenido anterior.

3.2.1.1. Diagrama de flujo de la rutina de Altas directas

La Fig. 37 ilustra el diagrama de flujo de la rutina de altas en un archivo relativo o de acceso directo.

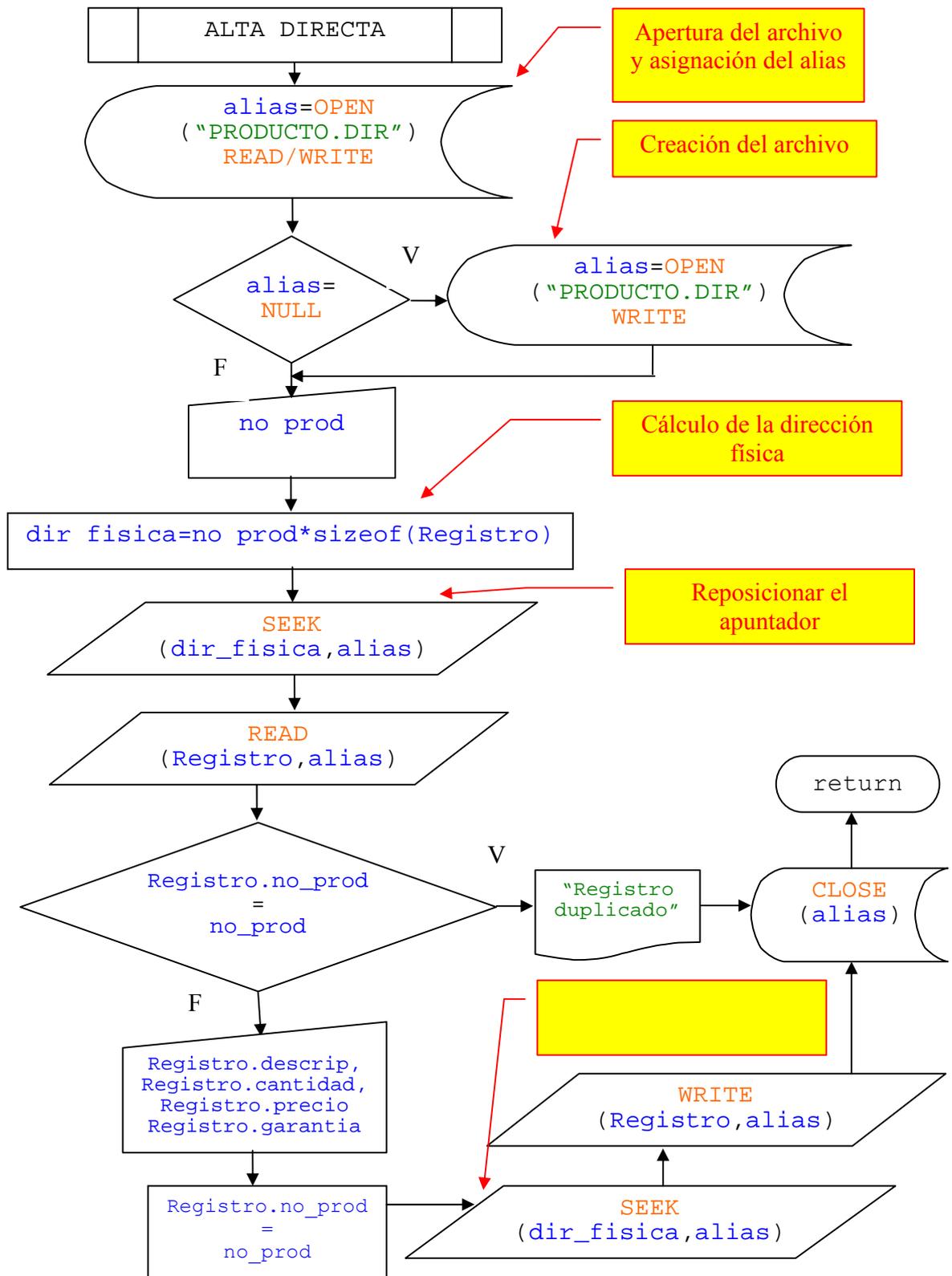


Fig. 37. Diagrama de flujo de rutina de altas directas

3.2.1.2. Codificación de la rutina de Altas directas

La Fig. 38 muestra la codificación de la rutina de altas en un archivo relativo o de acceso directo.

```

void ALTA_DIRECTA(void)
{
    int no_prod; // Variable local para el numero de producto
    clrscr();

    cout << "\n\rALTAS DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.DIR","rb+"); // Intenta abrir el archivo
    PRODUCTO.DIR
                                // en modo de lectura/escritura
    if(alias==NULL)
        alias=fopen("PRODUCTO.DIR","wb"); // Crea el archivo en caso de no
    existir

    cout << "\n\n\n\rNumero de producto: "; cin >> no_prod;

    dir_fisica=no_prod*sizeof(Registro); // Calculo de la dir. fisica
    fseek(alias,dir_fisica,SEEK_SET); //Posicionar el apuntador del archivo

    fread(&Registro,sizeof(Registro),1,alias);
    // Lee el "Registro", de tamano=sizeof(Registro) del archivo "alias"
    if(Registro.no_prod==no_prod)
    {
        cout << "\n\n\n\rRegistro duplicado !!!";
        fclose(alias);
        getch();
        return;
    }

    cout << "\n\rDescripcion: "; gets(Registro.descripcion);
    cout << "\n\rCantidad : "; cin >> Registro.cantidad;
    cout << "\n\rPrecio : "; cin >> Registro.precio;
    do
    {
        cout << "\n\rGarantia : "; Registro.garantia=toupper(getche());
    }while(Registro.garantia!='S' && Registro.garantia!='N');

    Registro.no_prod=no_prod;

    fseek(alias,dir_fisica,SEEK_SET); //Posicionar el apuntador del archivo
    fwrite(&Registro,sizeof(Registro),1,alias); // Grabar el Registro
    completo
    fclose(alias); // Cierra el archivo

    cout << "\n\n\n\rProducto registrado !!!";
    cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
    getch();
}

```

```
return;  
}
```

Fig. 38. Codificación de rutina de altas directas

3.3.2. CONSULTAS directas

En esta sección se analiza una rutina que busca un registro particular de un producto en un archivo directo. En este tipo de archivo no es necesario hacer el recorrido secuencial desde el primer registro almacenado, sino se calcula la dirección física del registro que se desea consultar y se posiciona el apuntador del archivo directamente.

3.3.2.1. Diagrama de flujo de la rutina de CONSULTAS directas

El diagrama de flujo de esta rutina se muestra en la Fig. 39.

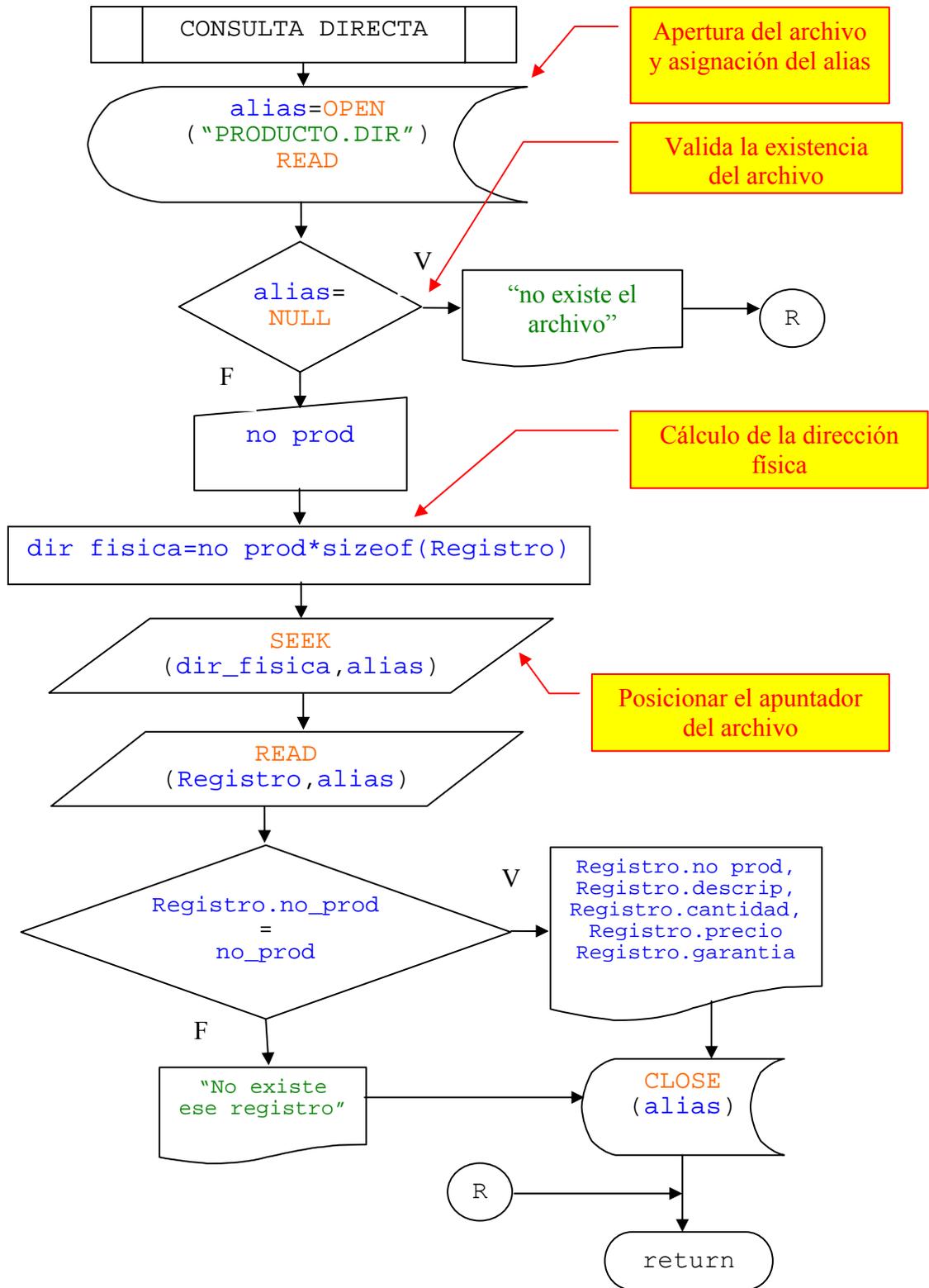


Fig. 39. Diagrama de flujo de rutina de consultas directas

3.3.2.2. Codificación de la rutina de CONSULTAS directas

La Fig. 40 muestra el código de la rutina de CONSULTAS directas de acuerdo al diagrama de la Fig. 39.

```

void CONSULTA_DIRECTA(void)
{
    int no_prod; // Variable local para el numero de producto que desea
    consultar
    clrscr();

    cout << "\n\nCONSULTA DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.DIR","rb"); // Intenta abrir el archivo
    PRODUCTO.DIR
                                // en modo de solo lectura

    if(alias==NULL)
    {
        cout << "\n\n\nNo existe el archivo !!!";
        cout << "\n\n<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }

    cout << "\n\n\nNumero de producto: "; cin >> no_prod;

    dir_fisica=no_prod*sizeof(Registro); // Calculo de la dir. fisica
    fseek(alias,dir_fisica,SEEK_SET); //Posicionar el apuntador del archivo
    fread(&Registro,sizeof(Registro),1,alias);
    // Lee el "Registro", de tamaño=sizeof(Registro) del archivo "alias"
    if(Registro.no_prod==no_prod)
    {
        cout << "\n\nNo Prod                Descripcion  Cantidad
Precio  Garantia";
        cout << "\n\n-----
-----";

        printf("\n\n%3d\t%30s\t%3d\t\t\t$%4.2f\t%c",Registro.no_prod,Registro.descr
ip,Registro.cantidad,Registro.precio,Registro.garantia);
    }
    else
    {
        cout << "\n\n\nNo existe ese registro !!!";
    }
    fclose(alias);
    cout << "\n\n\n\n\n<<< Oprima cualquier tecla para continuar >>>";
    getch();
    return;
}

```

Fig. 40. Codificación de rutina de consultas directas

3.3.3. MODIFICACIONES directas

Al igual que en el archivo secuencial, la forma de modificar el contenido de los campos de registros de un archivo, depende mucho de la rutina de consulta, ya que es necesario localizar previamente el registro que se desea modificar, capturar los nuevos valores y posteriormente **grabar el registro completo en la misma posición que se encontraba.** Esto último es muy importante porque recuerde que cuando se termina de leer un registro del archivo, el apuntador se posiciona al inicio del siguiente registro, por lo que, antes de grabar el registro modificado, es necesario reposicionar el apuntador del archivo en la dirección correcta.

3.3.3.1. Diagrama de flujo de la rutina de MODIFICACIONES directas

La Fig. 41 muestra el diagrama de flujo de la rutina de modificaciones de campos de un registro particular en un archivo directo.

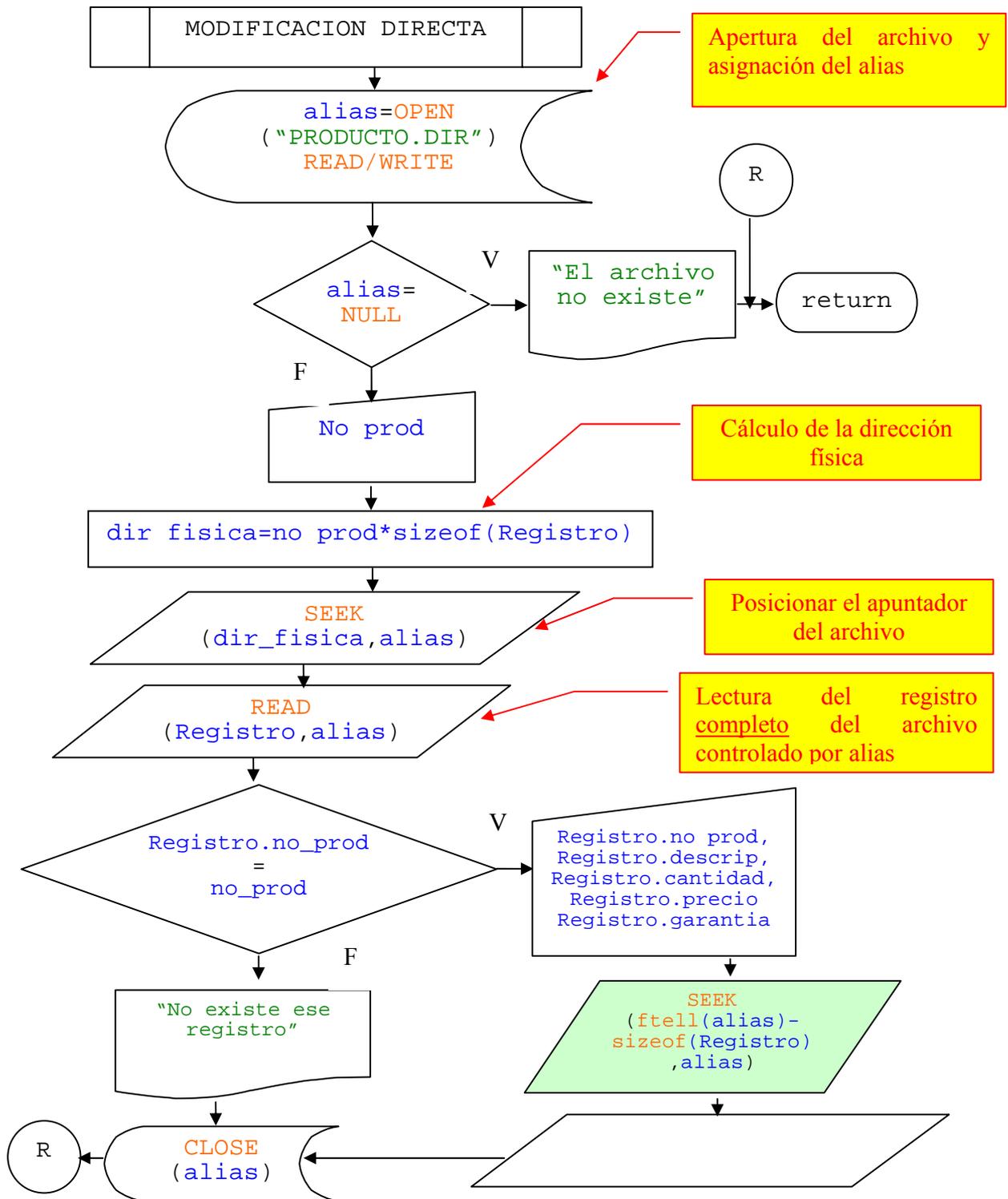


Fig. 41. Diagrama de flujo de rutina de modificación directa

3.3.3.2. Codificación de la rutina de MODIFICACIONES directas

La Fig. 42 muestra el diagrama de flujo de la rutina de modificaciones de campos de un registro particular en un archivo directo.

```

void MODIFICACION_DIRECTA(void)
{
    int no_prod; // Variable local para el numero de producto que desea
    modificar
    clrscr();

    cout << "\n\rMODIFICACION DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.DIR","rb+"); // Intenta abrir el archivo
    PRODUCTO.DIR
                                // en modo de lectura/escritura
    if(alias==NULL) // Valida la existencia del archivo
    {
        cout << "\n\n\n\rNo existe el archivo !!!";
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }

    cout << "\n\n\n\rNumero de producto: "; cin >> no_prod;

    dir_fisica=no_prod*sizeof(Registro); // Calculo de la dir. fisica
    fseek(alias,dir_fisica,SEEK_SET); //Posicionar el apuntador del archivo
    fread(&Registro,sizeof(Registro),1,alias);
    // Lee el "Registro", de tamano=sizeof(Registro) del archivo "alias"
    if(Registro.no_prod==no_prod)
    {
        cout << "\n\rNo Prod                Descripcion  Cantidad
Precio  Garantia";
        cout << "\n\r-----
-----";

        printf("\n\r%3d\t%30s\t%3d\t\t\t$%4.2f\t%c",Registro.no_prod,Registro.descr
ip,Registro.cantidad,Registro.precio,Registro.garantia);

        cout << "\n\n\n\rAnote los nuevos datos ...";
        cout << "\n\rDescripcion: "; gets(Registro.descripcion);
        cout << "\n\rCantidad    : "; cin >> Registro.cantidad;
        cout << "\n\rPrecio      : "; cin >> Registro.precio;
        do
        {
            cout << "\n\rGarantia    : "; Registro.garantia=toupper(getche());
            }while(Registro.garantia!='S' && Registro.garantia!='N');

        // Es necesario reposicionar el apuntador del archivo al principio
del
        // registro que desea modificar, ya que al leer un registro, el

```

```

        // apuntador se posiciona en el registro siguiente
        // La funcion ftell(alias) devuelve la posicion donde se encuentra
el
        // apuntador
        fseek(alias,dir_fisica,SEEK_SET); //Posicionar el apuntador del
archivo
        fwrite(&Registro,sizeof(Registro),1,alias); // Graba el registro
con
                                // los nuevos campos
        fclose(alias); // Cierra el archivo
        cout << "\n\n\n\rRegistro modificado !!!";
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }
    else
    {
        cout << "\n\n\n\rNo se encuentra ese registro !!!";
    }
    fclose(alias); // Cierra el archivo
    cout << "\n\n\n\n\r<<< Oprima cualquier tecla para continuar >>>";
    getch();
    return;
}

```

Fig. 42. Codificación de rutina de modificaciones directas

3.3.4. Bajas de registros en un archivo de acceso directo (bajas lógicas)

Para el ejemplo que se ha estado manejando en este archivo directo, NO se pueden realizar bajas físicas (únicamente bajas lógicas), ya que existe una relación directa entre la dirección lógica y el número del producto: esto es, que el producto 1 se almacena en el registro lógico 1, o sea que cada producto se almacena en su respectivo registro de acuerdo al número de producto y si se aplicase la rutina de bajas físicas (compactación), los registros se recorrerían a otras direcciones que no corresponderían con su número de producto y dichos registros no podrían ser localizados por las rutinas de consultas, modificaciones y las mismas bajas lógicas.

Al igual que en las bajas lógicas de los archivos secuenciales, cuando se elimina un registro en forma lógica de un archivo, sólo se le coloca una marca

especial en alguno de sus campos que lo identifique como registro “borrado”, sin embargo, el registro sigue existiendo en el archivo y por lo tanto ocupa espacio.

3.3.4.1. Diagrama de flujo de la rutina de BAJAS lógicas directas

Como se mencionó en el punto anterior, las bajas lógicas consisten en “marcar” los registros eliminados. En el ejemplo práctico que se muestra a continuación, el registro borrado se “limpia”, dejando en blanco todos sus campos (colocando el valor cero en los campos numéricos y blancos en las cadenas o de tipo carácter).

Para fines prácticos, la rutina de bajas lógicas se asemeja mucho a la rutina de modificaciones, sólo que en las bajas no se capturan los nuevos valores, sino se les asigna valores en blanco y se graba el registro completo en la misma posición en la que se encontraba. La diferencia entre las rutinas de bajas lógicas del archivo secuencial y el archivo directo radica en el mecanismo utilizado para localizar el registro que se desea eliminar. El diagrama de la Fig. 43 muestra esta rutina.

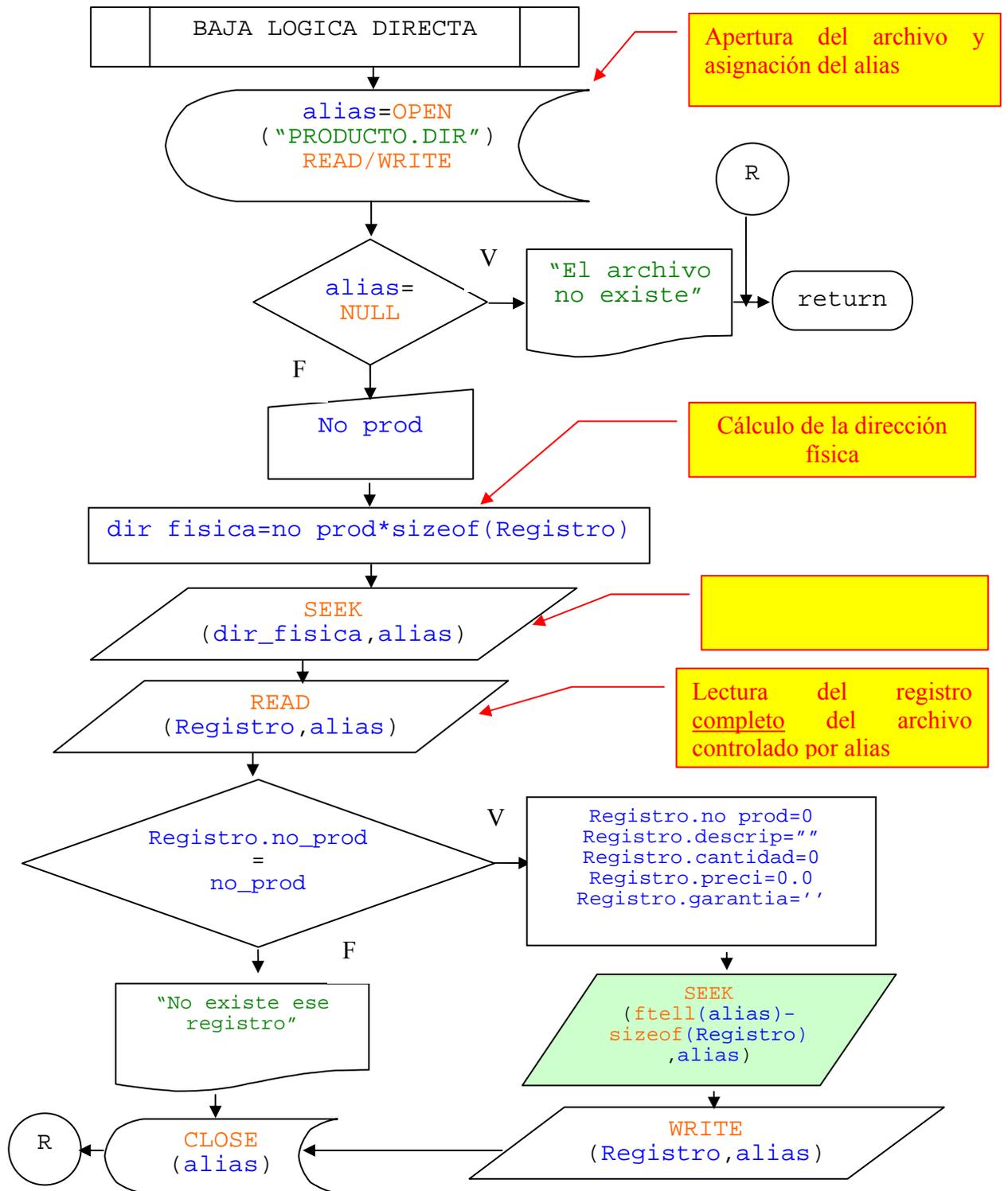


Fig. 43. Diagrama de flujo de rutina de baja lógica directa

3.3.4.2. Codificación de la rutina de BAJAS lógicas directas

La Fig. 44 muestra el diagrama de flujo de la rutina de bajas lógicas en un archivo directo.

```

void BAJA_LOGICA_DIRECTA(void)
{
    int no_prod; // Variable local para el numero de producto que desea
eliminar
    char op;
    clrscr();

    cout << "\n\rBAJA LOGICA DE REGISTROS DE PRODUCTOS";
    alias=fopen("PRODUCTO.DIR","rb+"); // Intenta abrir el archivo
PRODUCTO.DIR
                                // en modo de lectura/escritura
    if(alias==NULL) // Valida la existencia del archivo
    {
        cout << "\n\n\n\rNo existe el archivo !!!";
        cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
        getch();
        return;
    }

    cout << "\n\n\n\rNumero de producto: "; cin >> no_prod;

    dir_fisica=no_prod*sizeof(Registro); // Calculo de la dir. fisica
    fseek(alias,dir_fisica,SEEK_SET); //Posicionar el apuntador del archivo
    fread(&Registro,sizeof(Registro),1,alias);
    // Lee el "Registro", de tamaño=sizeof(Registro) del archivo "alias"
    if(Registro.no_prod==no_prod)
    {
        cout << "\n\rNo Prod                Descripcion  Cantidad
Precio  Garantia";
        cout << "\n\r-----
-----";

        printf("\n\r%3d\t%30s\t%3d\t\t$%4.2f\t%c\n\n\n\n\r",Registro.no_prod,Regi
stro.descrip,Registro.cantidad,Registro.precio,Registro.garantia);

        Registro.no_prod=0;
        strcpy(Registro.descrip,"");
        Registro.cantidad=0;
        Registro.precio=0.0;
        Registro.garantia=' ';

        // Es necesario reposicionar el apuntador del archivo al principio
del
        // registro que desea modificar, ya que al leer un registro, el
// apuntador se posiciona en el registro siguiente
// La funcion ftell(alias) devuelve la posicion donde se encuentra

```

```
el
    // apuntador
    do
    {
    cout << "\n\rEsta seguro que desea eliminar este registro? [S/N] --
-> ";
    op=toupper(getche());
    }while(op!='S' && op!='N');
    if(op=='S')
    {
    fseek(alias,dir_fisica,SEEK_SET); //Posicionar el apuntador del
archivo
    fwrite(&Registro,sizeof(Registro),1,alias); // Graba el registro
con
// los nuevos campos
    cout << "\n\n\n\rRegistro eliminado logicamente !!!";
    }
    else
    {
    cout << "\n\n\n\rRegistro NO eliminado !!!";
    }
    fclose(alias); // Cierra el archivo
    cout << "\n\r<<< Oprima cualquier tecla para continuar >>>";
    getch();
    return;
}
else
{
    cout << "\n\n\n\rNo se encuentra ese registro !!!";
}
fclose(alias); // Cierra el archivo
cout << "\n\n\n\n\r<<< Oprima cualquier tecla para continuar >>>";
getch();
return;
}
```

Fig. 44. Codificación de rutina de baja lógica directa

4.CONCLUSIONES

Aunque existe una gran diversidad de aplicaciones que se pueden desarrollar con manejo de archivos que pueden ser sumamente completas y complejas, estos apuntes presentan, de una forma sencilla y comprensible, los aspectos básicos de programación de archivos usando lenguaje C++. De tal forma, que no presenta lógica abrumadora de control de detalles, sino la base fundamental del material es entender y utilizar las funciones básicas de manejo de archivos en este lenguaje para posteriormente usarlas en el curso de “Administración de Archivos” y en cursos posteriores.

Cabe destacar que la codificación de las rutinas que se muestran se pueden obtener en forma íntegra en el sitio <http://www.itnuevolaredo.edu.mx/takeyas> y ejecutarse para reafirmar el contenido.

5. BIBLIOGRAFIA

- García Badell, J. Javier. **"Turbo C. Programación en manejo de archivos"**. Macrobit.
- Joyanes Aguilar, Luis. **"Problemas de Metodología de la Programación"**. McGraw Hill. 1990.
- Lafore, Robert. **"Turbo C. Programming for the PC"**.
- Loomis, Mary E.S. **"Estructura de Datos y Organización de Archivos"**. Prentice Hall. México. 1991.
- Martin, James. **"Organización de las bases de datos"**. Prentice Hall. 1993.
- Rose, Cesar E. **"Archivos. Organización y Procedimientos"**. Computec. 1993.
- Sedgewick, Robert. **"Algorithms"**. Second edition. Addison Wesley. USA. 1988.
- Sedgewick, Robert. **"Algoritmos en C++"**. Addison Wesley. USA. 1995.
- Tsai, Alice Y. H. **"Sistemas de bases de datos. Administración y uso"**. Prentice Hall. 1988.