

Algoritmos y

Estructuras

De

Datos

Datos

en memoria externa:

ARCHIVOS (ficheros)

ESTRUCTURAS DE DATOS: Clasificaciones

- Según donde se almacenan
 - Internas (en memoria principal)
 - Externas (en memoria auxiliar)
- Según tipos de datos de sus componentes
 - Homogéneas (todas del mismo tipo)
 - No homogéneas (pueden ser de distinto tipo)
- Según la implementación
 - Provistas por los lenguajes (básicas)
 - Abstractas (TDA - Tipo de dato abstracto que puede implementarse de diferentes formas)
- Según la forma de almacenamiento
 - Estáticas (ocupan posiciones fijas y su tamaño nunca varía durante todo el módulo)
 - Dinámicas (su tamaño varía durante el módulo y sus posiciones también)

Introducción

- Un **archivo** es una colección de datos almacenados juntos bajo un nombre común.
 - ✓ Los programas en C++ son ejemplos de archivos.
- Los archivos **almacenan información de manera permanente** en dispositivos de almacenamiento de memoria secundaria, tales como dispositivos magnéticos (discos duros, cintas), discos ópticos (CDROM, DVD), memorias permanentes de estado sólido (memorias flash USB), etc.
 - ✓ La información puede ser tanto programas (software) como datos que serán utilizados por los programas.
- Los archivos de datos pueden ser creados, leídos y actualizados por programas en C++.

Archivos (archivos)

- Para tratar con un archivo se utilizan, aparte de otras operaciones, unas operaciones básicas que son: **abrir, cerrar, escribir y leer**.
- Los archivos pueden ser **de lectura**, en cuyo caso diremos que el tipo de acceso del archivo es de entrada, o bien **de escritura**, con lo cual tendremos un archivo de salida.
- Los archivos de salida pueden **crearse** (o sobrescribirse, si ya existen) o bien **actualizarse**.
 - ✓ También se dispone de archivos de **entrada / salida**.
- El tipo de acceso determinará las **operaciones disponibles** para el archivo.

La biblioteca que permite la manipulación de archivos recibe el nombre de **fstream**.

Esta biblioteca nos provee de objetos que serán de entrada (**ifstream**) o de salida (**ofstream**).

Todos los archivos disponen de operaciones para abrirlos, cerrarlos y preguntar si estamos al final.

Información relativa a un archivo

Los objetos del tipo archivo son objetos lógicos (variables de nuestro programa) que representan un archivo físico.

Para poder identificar de forma unívoca el archivo físico que manejan, se necesita un nombre de archivo.

Información en archivos



Tipos de Archivos

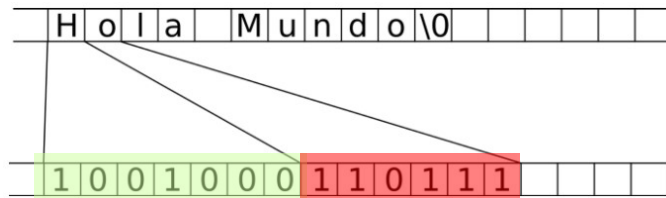
Existen dos tipos de archivos, que **se distinguen de acuerdo a la codificación o formato** en que almacenan la información:

- 1) Archivos de texto
- 2) Archivos binarios

Ambos tipos de archivos almacenan datos usando un código binario; la diferencia está en lo que representan los códigos.

Archivos de Texto

- La información se almacena como una secuencia de caracteres.
- Cada carácter individual (como una letra, dígito, signo de pesos, punto decimal, etc.) se almacena utilizando una codificación estándar (usualmente basada en la codificación ASCII).
- Al tratarse de un formato estandarizado, otros programas diferentes de aquel que creó el archivo (un procesador de palabras o un editor de texto) podrán entender y procesar su contenido.
- En el caso del software, los programas en código fuente codificados en un lenguaje de programación suelen ser almacenados como archivos de texto. Sin embargo, el resultado de compilar estos programas fuente a programas ejecutables se almacenan en archivos binarios.



Los archivos de texto al fin y al cabo son también archivos binarios, pero los tratamos como un tipo de archivo distinto, porque los podemos ver y editar desde un editor de texto.

Archivos Binarios

- La información se almacena con el mismo formato y codificación utilizada por el compilador C++ para almacenar sus datos primarios en memoria RAM:
 - Los *números* aparecen en su forma binaria verdadera
 - Las *cadenas* conservan su forma ASCII
- La ventaja de los archivos binarios es su *compactibilidad*: usan menos espacio para almacenar más números utilizando su código binario en lugar de valores de caracteres individuales.
- Están concebidos para ser procesados automáticamente por programas que conocen su formato interno.
- Se utilizan cuando no estamos interesados en que la información sea visible por otros programas.
- El procesamiento de archivos binarios es más eficiente que el de archivos de texto porque la información está representada directamente en código binario (exactamente tal y como se encuentra internamente en la memoria principal). De esta forma se evita la pérdida de tiempo que ocasionaría su conversión a un formato estándar (ASCII, UTF-8, etc), como ocurre con los archivos de texto.

Si queremos guardar el número 123456 en un archivo de texto, cada dígito ocupará 8 bytes. Mientras que, usando un formato binario, el número 123456 ocupará 4 bytes (es decir, lo que ocupa un int en C++).

Clasificación según tipo de acceso

Por la manera de acceder a los archivos, los podemos clasificar en función de:

1) La dirección del flujo de los datos:

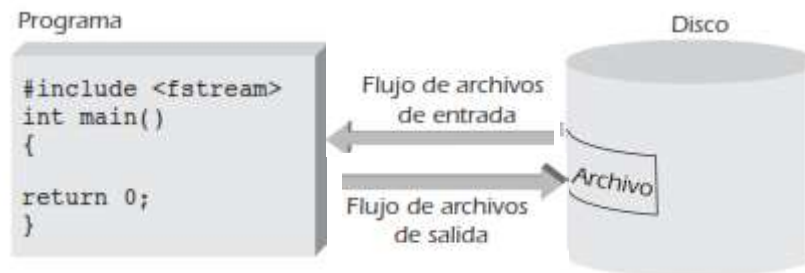
- **De entrada:** Son aquellos cuyos datos se leen por parte del programa.
- **De salida:** Aquellos archivos que el programa escribe.
- **De entrada/salida:** archivos que se pueden leer y escribir.

2) Dependiendo de cómo se accede a los datos en sí:

- **Secuencial:** El orden de acceso a los datos está determinado, primero se accede al primer elemento y luego se puede ir accediendo a los siguientes, de uno en uno.
- **Directo / Aleatorio:** Se puede acceder de forma directa a un elemento concreto del archivo. En este caso, el acceso es similar al de los arreglos.

Entrada y salida de información

- Un programa codificado en C++ realiza la entrada y salida de información a través de **flujos de entrada y salida** respectivamente.
- La entrada y salida de datos es a través de los **flujos estándares** de entrada y salida (**cin y cout**, respectivamente), usualmente conectados con el teclado y la pantalla de la consola.
- **Todo lo visto respecto a la entrada y salida** básica con los flujos estándares, o la entrada y salida de cadenas de caracteres, también **es aplicable** a los flujos de entrada y salida vinculados a **archivos**.



La dirección, o modo, se define en relación con el programa y no con el archivo

Ejemplo 1. Archivo de texto de entrada

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    char z; int a; double c;

    ifstream f;
    f.open ("archi.txt");

    cout << "Lectura de un archivo" << endl;
    f >> z >> a >> c;

    cout << "Salida: " << endl;
    cout << z << " " << a << " " << c;

    f.close( );
    return 0;
}
```

archi: Bloc de notas

Archivo Edición Formato Ver Ayuda

h 23 45.7

Archivo de texto conteniendo un carácter,
un valor entero y un valor double

Lectura desde el archivo

Lectura de un archivo
Salida:
h 23 45.7

Ejemplo 2. Archivo de texto de salida

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main(void) {
    char z; int a; double c;
    z='h'; a=20; c=5.8;

    ofstream f;
    f.open ("sali.txt");

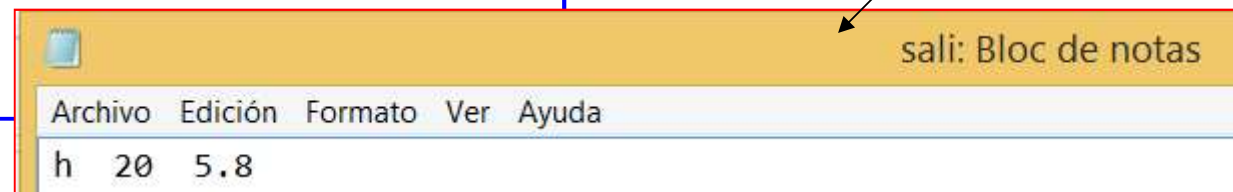
    cout << "Escritura en archivo" << endl;
    f << z << " " << a << " " << c;

    f.close();
    return 0;
}
```

Escritura en archivo

Escritura en el archivo

Archivo de texto generado



open: Cuando es para salida, si el archivo no existe lo crea, si ya existe lo sobrescribe vacío.

Ejemplo 3. Lectura de archivo con varios renglones

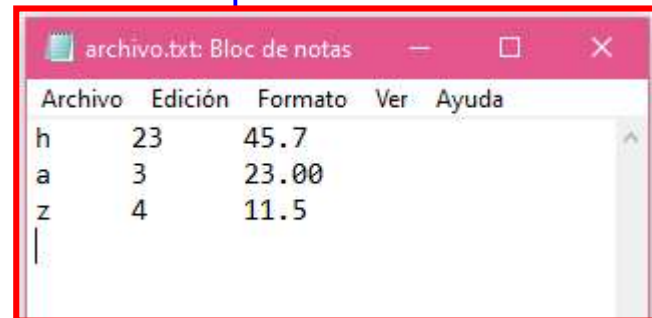
```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main() {
    char z; int a; double c;

    ifstream f;
    f.open("archivo.txt");

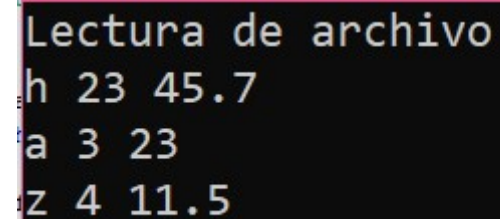
    cout << "Lectura de archivo" << endl;
    f >> z >> a >> c;
    while (!f.eof()) {
        cout << z << " " << a << " " << c << endl;
        f >> z >> a >> c;
    }

    f.close();
    return 0;
}
```



archivo.txt: Bloc de notas

Archivo	Edición	Formato	Ver	Ayuda
h	23	45.7		
a	3	23.00		
z	4	11.5		



```
Lectura de archivo
h 23 45.7
a 3 23
z 4 11.5
```

Comprobar la apertura del archivo

Un archivo físico que se haya intentado abrir y que no exista no tendrá su correspondiente archivo lógico asociado.

- Para comprobarlo se usa la instrucción **fail** que consulta si el archivo falló:

```
if (archivoTexto.fail())  
    // el archivo no se ha abierto
```

*archivoTexto es la variable
u objeto lógico de tipo
ifstream / ofstream.*

fail(): Devuelve true si el archivo no se ha abierto con éxito;
de lo contrario, devuelve false.

- O bien se puede hacer un **if** sobre la **variable** del tipo archivo:

```
if (!archivoTexto)  
    // el archivo no se ha abierto
```

Ambas posibilidades son equivalentes.

Lectura

Una vez que el archivo está abierto, podemos **leer elementos** con el **operador de entrada: “>>”** .

Se usa igual que en las instrucciones de entrada estándar (cin » variable) pero substituyendo el flujo *cin* por el *nombre de la variable del tipo ifstream*.

Para leer un carácter: **archivoTexto » caract;**

Dicha operación pone en la variable *caract* el siguiente elemento leído del archivo.

Por defecto, la lectura con el operador “>>” *ignora* los espacios en blanco, tabulaciones y saltos de línea.

- Para que no los ignore: **archivoTexto.unsetf(ios::skipws);**
- Para que vuelva a ignorar los espacios en blanco:
archivoTexto.setf(ios::skipws);

Ejemplo 4. Lectura de un archivo de texto

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    char caract;
    ifstream archivo_entr;
    archivo_entr.open("datos.txt");
    if (!archivo_entr)
        cout << "Error al abrir el fichero" << endl;
    else {
        archivo_entr.unsetf(ios::skipws);
        archivo_entr >> caract;
        while (!archivo_entr.eof()){
            cout << caract;
            archivo_entr >> caract;
        }
    }
    archivo_entr.close();
    return 0;
}
```

datos.txt: Bloc de notas

Archivo	Edición	Formato	Ver
12 bc def ghi			
ff			
hh			
b			

```
12 bc def ghi
ff
hh
b
```

Tratamiento de Archivos

Cuando un programa quiere trabajar con un determinado archivo, debe realizar las siguientes acciones:

1. Incluir la biblioteca `<fstream>`, que contiene los elementos necesarios para procesar el archivo.
2. Usar el espacio de nombres `std`.
3. Declarar las variables que actuarán como manejadores de archivos.
4. Abrir el flujo de datos, vinculando la variable correspondiente con el archivo especificado. Esta operación establece un vínculo entre la variable (manejador de archivo) definida en nuestro programa y el archivo físico.

Toda transferencia de información entre el programa y un archivo se realizará a través de la variable manejador que ha sido vinculada con dicho archivo.

5. Comprobar que la apertura del archivo se realizó correctamente. Si la vinculación con el archivo especificado no pudo realizarse por algún motivo (por ejemplo, si queremos hacer entrada de datos desde un archivo que no existe, o si es imposible crear un archivo en el que escribir datos), entonces la operación de apertura fallaría.

Tratamiento de Archivos

6. Realizar la transferencia de información (de entrada o de salida) con el archivo a través de la variable de flujo vinculada al mismo.

Normalmente, tanto la entrada como la salida de datos se realizan mediante un **proceso iterativo**. En el caso de entrada dicho proceso suele requerir la lectura de todo el contenido del archivo.

7. Comprobar que el procesamiento del archivo del paso previo se realizó correctamente.

En el caso de procesamiento para entrada ello suele consistir en comprobar si el estado de la variable manejador indica que **se ha alcanzado el final del archivo**.

El procesamiento para salida suele consistir en comprobar si el estado de la variable manejador indica que **se ha producido un error de escritura**.

8. Finalmente, cerrar el flujo para liberar la variable manejador de su vinculación con el archivo.

En caso de no cerrar el flujo, éste será *cerrado automáticamente* cuando termine el ámbito de vida de la variable manejador del archivo.

Tratamiento de Ficheros

Todo tratamiento de ficheros consta de tres pasos:

1. **Apertura del archivo:** Momento en el cual se mapea el objeto lógico (la variable que representa el archivo en nuestro programa) con el objeto físico (el archivo en disco). Del modo de apertura dependerá si un archivo es de lectura o escritura.
2. **Acceso al archivo:** Etapa en la cual se van leyendo o escribiendo los elementos que interesen.
3. **Cierre del archivo:** Actualiza el archivo y elimina la información relativa a él que se haya creado en el momento de abrirlo, tanto por parte del compilador como del sistema operativo.

Para abrir un fichero se usa la instrucción **open** y como parámetro se pasa el nombre del fichero y el modo de apertura.

Para cerrar un fichero se usa la instrucción **close** sin ningún argumento.

Opciones para abrir archivos

Para **abrir un archivo** usamos el método:

```
void open (const char* nombre, int nModo = ios::in)
```

- (**nombre**) es el nombre del archivo
- (**nModo**) es el modo de apertura

Modo	Significado
<code>ios::in</code>	Abre para lectura (valor por defecto para <code>ifstream</code>). Se sitúa al principio.
<code>ios::out</code>	Abre para crear nuevo archivo (valor por defecto para <code>ofstream</code>). Se sitúa al principio y pierde los datos.
<code>ios::app</code>	Abre para añadir al final.
<code>ios::trunc</code>	Crea un archivo para escribir/leer (si ya existe se pierden los datos).
<code>ios::binary</code>	Archivo binario.

Lectura: get

La instrucción ">>" se saltea los espacios en blanco.

Si se requiere leer todo el archivo tal y como está sin saltarse espacios, tabulaciones, etc., se puede usar la función **get**.

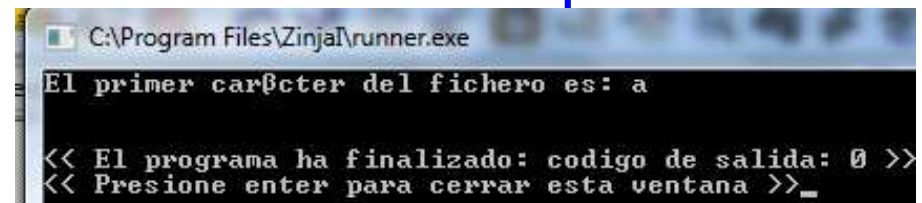
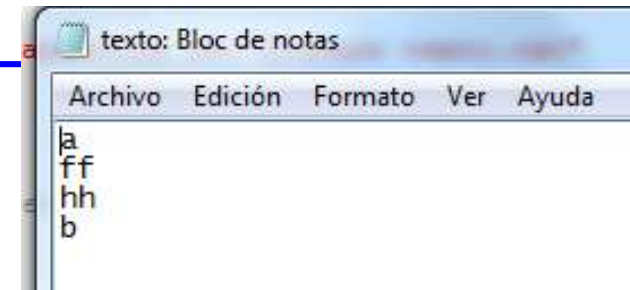
La operación **get** se puede llamar de 3 formas:

- **int get ()**: Lee un sólo carácter y lo retorna. No se salta espacios en blanco.
- **get(char& car)**: Lee un sólo carácter de un flujo de entrada y lo pone en la variable **car**.
- **get (char* cad, int numCar, char delimitador='\n')**: Se usa para leer un número determinado de caracteres dado en *numCad* y termina si encuentra un salto de línea.
 - ✓ En este caso lo que se leerá es como máximo el número de caracteres indicado en *numCar-1* o el carácter delimitador, y los almacenará en la cadena de caracteres **cad**.
 - ✓ El carácter delimitador es opcional y sirve para definir un delimitador, que por defecto es el salto de línea. Si se encuentra el delimitador, **no es extraído** de la secuencia de entrada, y permanece ahí como el siguiente carácter a ser extraído.

Ejemplo 5. Uso de get()

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main() {
    ifstream ficheroTexto;
    char c;
    ficheroTexto.open ("texto.txt");
    if (!ficheroTexto)
        cout << "Error al abrir el archivo" << endl;
    else {
        //Comprobamos que el fichero no esté vacío
        if (!ficheroTexto.eof()) {
            ficheroTexto.get(c); //Leemos el 1er caracter
            cout << "El primer caracter del fichero es: "
                 << c << endl;
        }
    }
    ficheroTexto.close();
    return 0;
}
```



Ejemplo 6. Uso de get con parámetros

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main(){
    ifstream ficheroTexto;
    char car[30];

    ficheroTexto.open ("texto.txt");
    if (!ficheroTexto)
        cout << "Error al abrir el archivo" << endl;
    else {
        //Comprobamos que el fichero no esté vacío
        if (!ficheroTexto.eof()){
            ficheroTexto.get(car,3, '\n');
            cout << "Los caracteres leídos del fichero son: "
                << car << endl;
        }
    }
    ficheroTexto.close();
    return 0;
}
```

texto.txt: Bloc de notas

Archivo	Edición	Formato	Ver
abcd			
ff			
hh			
b			

C:\Program Files (x86)\Zinjal\bin\run

Los caracteres leídos del fichero son: ab

Lectura: getline

Permite la lectura de toda una línea.

Tiene la siguiente sintaxis:

```
getline( char* cad, int numCar, char delim = '\n' );
```

Cuando se ejecuta almacena en la cadena **cad** los caracteres del archivo hasta que:

- ✓ encuentre el delimitador **delim** (indicado como 3er parámetro), o
- ✓ como máximo haya leído **numCar - 1** caracteres, o
- ✓ encuentre el final del archivo

En el último elemento de la cadena **cad** se guarda **el carácter final de cadena: '\0'**.

El 3er parámetro es opcional y sirve para definir un delimitador, que por defecto es el salto de línea. Si se encuentra el delimitador, **se extrae** de la secuencia de entrada pero es descartado, **no se copia** a la cadena **cad**.

Ejemplo 7. Uso de getline

The image shows a C++ program in a code editor, a Notepad window, and a terminal window. The code in the editor opens a file named 'texto2.txt' and reads its contents line by line using `getline`. The Notepad window shows the contents of 'texto2.txt' as 'affjrpot', 'hh', and 'b'. The terminal window shows the output of the program.

```
int main() {  
    ifstream ficheroTexto;  
    char car[30];  
  
    ficheroTexto.open ("texto2.txt");  
    if (!ficheroTexto)  
        cout << "Error al abrir el archivo" << endl;  
  
    else {  
        if (!ficheroTexto.eof()) {  
            ficheroTexto.getline(car, '\n');  
            cout << car << endl;  
        }  
    }  
    ficheroTexto.close();  
    return 0;  
}
```

texto2.txt: Bloc de notas

Archivo Edición Formato Ver

affjrpot
hh
b

Los caracteres leídos del fichero son: affjrpot

<< El programa ha finalizado: código de salida: 0 >>
<< Presione enter para cerrar esta ventana >>

Diferencias entre get y getline

Entonces, ¿cual es la diferencia? Sutil pero importante:

- La función `get()` se detiene cuando ve el delimitador en el flujo de entrada, pero no lo extrae del flujo de entrada. Entonces, si se hace otro `get()` usando el mismo delimitador, retornará inmediatamente sin ninguna entrada contenida.
- La función `getline()`, por el contrario, sí extrae el delimitador del flujo de entrada, pero no lo almacena en la cadena resultante.

Diferencias entre get y getline

```
ficheroTexto.open ("texto2.txt");  
if (!ficheroTexto)  
    cout << "Error al abrir el archivo" << endl;  
else {  
    if (!ficheroTexto.eof()){  
        ficheroTexto.get(car, '\n');  
        cout << "Los caracteres leidos del fichero son: "  
            << car << endl;  
        ficheroTexto.get(car, '\n');  
        cout << "Los caracteres leidos del fichero son: "  
            << car << endl;  
    }  
}  
ficheroTexto.close();
```

texto2.txt: Bloc de notas

Archivo	Edición	Formato	Ver
affjrpot hh b			

```
Los caracteres leidos del fichero son: affjrpot  
Los caracteres leidos del fichero son:  
  
<< El programa ha finalizado: codigo de salida: 0 >>  
<< Presione enter para cerrar esta ventana >>
```

```
ficheroTexto.open ("texto2.txt");  
if (!ficheroTexto)  
    cout << "Error al abrir el archivo" << endl;  
else {  
    if (!ficheroTexto.eof()){  
        ficheroTexto.getline(car, '\n');  
        cout << "Los caracteres leidos del fichero son: "  
            << car << endl;  
        ficheroTexto.getline(car, '\n');  
        cout << "Los caracteres leidos del fichero son: "  
            << car << endl;  
    }  
}  
ficheroTexto.close();
```

```
Los caracteres leidos del fichero son: affjrpot  
Los caracteres leidos del fichero son: hh
```

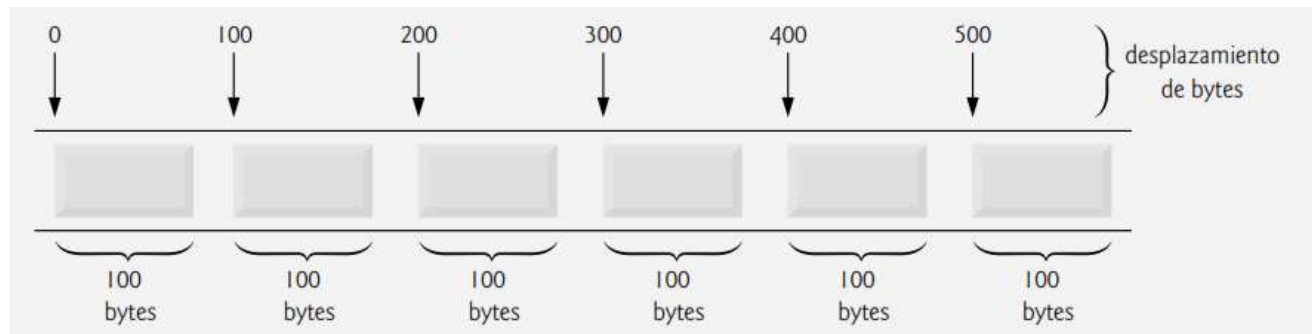
```
<< El programa ha finalizado: codigo de salida: 0 >>  
<< Presione enter para cerrar esta ventana >>
```

Archivos de Acceso Directo o Aleatorio

Los **archivos secuenciales no son apropiados** para las aplicaciones de acceso directo, en las que un registro específico se debe localizar inmediatamente: sistemas de reservación de aerolíneas, sistemas bancarios, sistemas de punto de venta, cajeros automáticos, etc.

Para estos casos se requiere de **archivos de acceso aleatorio**, cuyos registros individuales se pueden acceder de manera directa y rápida, sin tener que buscar en otros registros.

C++ no impone una estructura sobre un archivo de acceso aleatorio. La aplicación que desee utilizar este tipo de archivos debe crearlos. El método más sencillo es requerir que todos los registros en el archivo sean de la misma longitud fija. De este modo, es más fácil para el programa calcular la ubicación exacta de cualquier registro relativo al inicio del archivo facilitando el acceso inmediato a registros específicos, incluso en archivos extensos.



Representación gráfica de un archivo de acceso directo en C++

Archivos Binarios: La función de lectura

Prototipo: `istream& read (char* s, streamsize n);`

Lee un bloque de datos desde un archivo binario.

Extrae n caracteres (indicados en el 2do parámetro) desde un archivo de entrada y los almacena en la cadena de caracteres s (indicada en el 1er parámetro).

- ✓ Para leer un tipo de dato que no sea una cadena de caracteres se deberá hacer un *casting*.
- ✓ Esta función copia un bloque de datos sin chequear su contenido (si tiene caracteres nulos o de final).

Cuando no sepamos la medida en bytes de la información que queremos leer, deberemos usar la función **sizeof** que dado un tipo nos devolverá su tamaño en bytes.

Ejemplo: tenemos una variable var de tipo *double* y queremos leer de un archivo un valor de tipo double y ponerlo en var . Haremos:

`archivo.read ((char*)& var, sizeof(double));`

Variable auxiliar en la que se
almacenan los caracteres leídos

Cantidad de caracteres que se
leen de un archivo binario

Archivos Binarios: La función de escritura

Prototipo: `ostream& write (const char* s, streamsize n);`

Escribe un bloque de datos en un archivo binario.

Inserta n caracteres (indicados en el 2do parámetro) en el archivo de salida, desde la cadena de caracteres s (indicada en el 1er parámetro).

✓ Esta función tampoco valida contenido.

Ejemplo 8. Escritura en archivos binarios con registros

```
#include <iostream>
#include <fstream>
using namespace std;

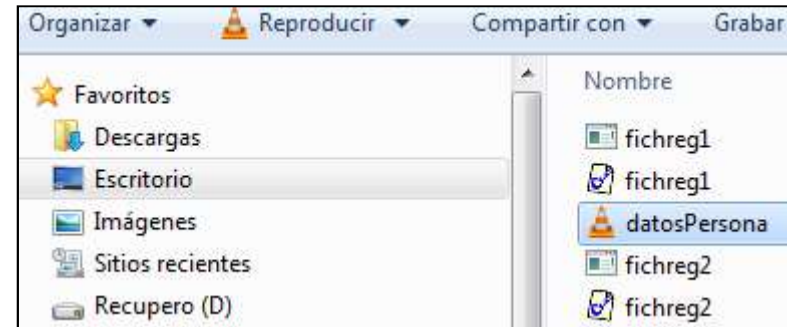
typedef char id[20];

struct persona {
    id nombre, apellido;
    int edad;
};

int main() {
    ofstream fPers;
    fPers.open("datosPersona.bin");
    if(!fPers)
        cout << "Error al abrir el fichero" << endl;
    else {
        persona p;
        strcpy(p.nombre, "Agustina");
        strcpy(p.apellido, "Soler");
        p.edad = 34;

        // Escribimos una persona
        fPers.write((char *)(&p), sizeof(p));

        fPers.close();
    }
    return 0;
}
```



Graba en el archivo de salida una cantidad de bytes (indicadas por sizeof), obtenidos de la dirección de memoria (&p) donde están guardados los datos que se escribirán en el archivo, tratándola como una cadena de caracteres (puntero a char)

Ejemplo 9. Lectura de archivos binarios con registros

```
#include <iostream>
#include <fstream>
using namespace std;

typedef char id[20];
struct persona {
    id nombre, apellido;
    int edad;
};

int main() {
    persona p;
    ifstream fEntrada;
    fEntrada.open("datosPersona.bin");
    if (!fEntrada)
        cout << "Error al abrir el fichero" << endl;
    else {
        // Comprobamos que no este vacio
        if (!fEntrada.eof()) {
            fEntrada.read((char*)(&p), sizeof(p));
            cout << p.nombre << endl;
            cout << p.apellido << endl;
            cout << p.edad << endl;
        }
    }
    return 0;
}
```

```
Agustina
Soler
34

<< El programa ha finalizado:
```

Lee desde el archivo de entrada una cantidad de bytes (del sizeof), tratándola como caracteres (puntero a char) y lo almacena en la dirección de memoria del registro (p)

Funciones de posicionamiento

- En el **acceso aleatorio (directo)**, cualquier carácter en el archivo abierto puede leerse en forma directa sin tener que leer primero, en forma secuencial, todos los caracteres almacenados antes que él.
- Para proporcionar acceso aleatorio a los archivos, cada objeto **ifstream** crea en forma automática un marcador de posición de archivo, **seekg/seekp**.
- Este marcador es un **numero entero largo** que representa un **desplazamiento** desde el principio de cada archivo e indica el **lugar desde donde se va a leer o a escribir el siguiente carácter**.

Función de posicionamiento: seek

Para leer y escribir archivos de acceso directo usaremos las operaciones read y write, y tenemos disponible la instrucción que nos **permite posicionarnos en una posición aleatoria en el archivo**:

seekg : “seek get”,

se puede aplicar sobre archivos de entrada, así como archivos de E/S, e indica la posición del próximo *get*.

Para archivos de salida, la posición para el próximo *put* se puede indicar con:

seekp : “seek put”,

Ambas tienen el mismo formato:

Posición donde se hará la próxima lectura

seekg(pos_type* posi) → (Esta forma es siempre relativa al *inicio* del archivo)

O bien:

Cantidad *off* de posiciones a desplazarse

seekg(off_type* off, ios_base::direcc)

Desde dónde se va a desplazar

Lugares desde donde se van a realizar los desplazamientos:

Valor	Descripción
ios::beg	El desplazamiento es relativo al inicio del fichero.
ios::end	El desplazamiento es relativo al final del fichero.
ios::cur	El desplazamiento es relativo a la posición actual en el fichero.

Función de posicionamiento: seek

La función **seek** nos permite acceder a cualquier posición del archivo, no tiene por qué ser exactamente al principio de un registro.

Cada carácter en un archivo de datos se localiza por su posición en el archivo. El primer carácter en el archivo se localiza en la posición 0, el siguiente en la posición 1, etc. Se hace referencia a la posición de un carácter como su **desplazamiento desde el inicio** del archivo. Por tanto, el primer carácter tiene un desplazamiento de 0, el segundo carácter tiene un desplazamiento de 1, etc., para cada carácter en el archivo.

El desplazamiento de la función **seek** es de 1 byte.

Un desplazamiento positivo significa avanzar en el archivo y un desplazamiento negativo significa retroceder.

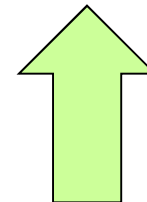
Función de posicionamiento: seek

Por ejemplo:

- Si nos queremos desplazar al inicio del archivo haremos `seekg(0)`,
- Si nos queremos desplazar al final del archivo: `seekg (0, ios:: end)`.

Hay que tener en cuenta que el final de un archivo no es la posición del último elemento sino la **posición del cursor después de leer el último elemento**.

Esto quiere decir que si queremos consultar el último elemento tendríamos que ponernos en la posición: `seekg(-1, ios::end)`.



Último elemento

Posición del marcador de posicionamiento: tell

Para saber, en un momento determinado, **la posición en la que estamos**, ésta se puede consultar con la instrucción **tell**

tellg () : devuelve la posición del cursor (o el valor de desplazamiento del marcador de posición) dentro del archivo de lectura.

Para los **archivos de escritura** tendremos **tellp ()** que funciona igual.

Ejemplo: si se leyeron diez caracteres desde un archivo de entrada llamado *archivo_entr*, la siguiente sentencia devolverá el número entero largo 10:

archivo_entr.tellg();

Esto significa que el siguiente carácter que se va a leer está desplazado diez posiciones desde el inicio del archivo, y es el undécimo carácter en el archivo.

Ejemplo 10. Búsqueda en archivos estructurados en registros con acceso directo

```
typedef char id[20];  
struct persona {  
    id nombre, apellido;  
    int edad;  
};
```

```
int main() {  
    ifstream fEntrada;  
    int pos;  
    persona p;  
    fEntrada.open("datosPersona.bin");  
    if (!fEntrada)  
        cout << "Error al abrir el fichero" << endl;  
    else {  
        cout << "Ingrese el nro de persona a mostrar: ";  
        cin >> pos;  
        // Comprobamos que no este vacio  
        if ((pos) * sizeof(persona) > fEntrada.seekg(0, ios::end).tellg())  
            cout << "Ese elemento no existe" << endl;  
        else{  
            fEntrada.seekg((pos-1) * sizeof(persona));  
            fEntrada.read((char*)(&p), sizeof(p));  
            cout << p.nombre << endl;  
            cout << p.apellido << endl;  
            cout << p.edad << endl;  
        }  
    }  
    return 0;  
}
```

```
Ingrese el nro de persona a mostrar: 1  
Agustina  
Soler  
34
```

```
<< El programa ha finalizado: codigo de salida
```

```
Ingrese el nro de persona a mostrar: 7  
Ese elemento no existe
```

```
<< El programa ha finalizado: codigo de salida
```

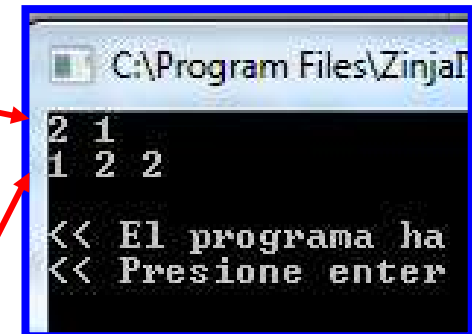

Ejemplo 10. Búsqueda en archivos estructurados en registros con acceso directo

```
cout << "Ingrese el nro de persona a mostrar: ";
cin >> pos;
// Comprobamos que no este vacio
if ((pos) * sizeof(persona) > fEntrada.seekg(0, ios::end).tellg())
    cout << "Ese elemento no existe" << endl;
else{
    fEntrada.seekg((pos-1) * sizeof(persona));
    fEntrada.read((char*)(&p), sizeof(p));
    cout << p.nombre << endl;
    cout << p.apellido << endl;
    cout << p.edad << endl;
}
```

Ubicamos el puntero en la posición de la persona numero *pos*

Ejemplo 11. Lectura y escritura de arreglos

```
int main(void) {  
    int a[3][4]={{1,2,3,4},{1,1,1,1},{2,2,2,2}};  
    int b[3][4];  
    int c[12];  
  
    //-----  
    ofstream f;  
    f.open ("buf1.bin");  
  
    f.write ( (char*)a, sizeof(a) );  
    f.close( );  
    //-----  
    ifstream g;  
    g.open ("buf1.bin");  
  
    g.read ( (char*) b, sizeof(b) );  
    cout << b[2][3]<< " " << b[0][0] << endl;  
    g.close( );  
    // -----  
    ifstream h;  
    h.open ("buf1.bin");  
  
    h.read ( (char*) c, sizeof(c) );  
    cout << c[5] << " " << c[10] << " " << c[11];  
    h.close( );  
    return 0;  
}
```



C:\Program Files\Zinjal
2 1
1 2 2
<< El programa ha
<< Presione enter

Ejemplo 12. Escritura de 200 pares de valores

```
int main() {  
    ofstream archi;  
    archi.open("grupo2.dat", ios::binary);  
    struct par {  
        float c1;  
        int c2;  
    };  
    par vector[200];  
  
    for (int c=1; c<=200; c++){  
        vector[c-1].c1 = rand()/(1000.0);  
        vector[c-1].c2 = rand()%1000;  
        cout << vector[c-1].c1 << " - " << vector[c-1].c2 << endl;  
    }  
    archi.write((char *)vector, sizeof(par)*200);  
    archi.close();  
    system("pause");  
    return 0;  
}
```

```
18.787 - 905  
17.958 - 391  
10.202 - 625  
26.477 - 414  
9.314 - 824  
29.334 - 874  
24.372 - 159  
11.833 - 70  
7.487 - 297  
7.518 - 177
```

```
<< El programa ha finalizado:
```

Otros ejemplos

Algoritmos sobre archivos - Fusión de 2 fich de caracteres

```
#include <iostream>
#include <fstream>
using namespace std;
void fusionarFicheros(ifstream& f1, ifstream& f2, ofstream& fResu);

int main()
{
    char nombre1[100] , nombre2[100];
    cout << "Introduce el nombre del primer fichero: ";
    cin >> nombre1;
    ifstream f1;
    f1.open(nombre1);
    cout << "Introduce el nombre del segundo fichero: ";
    cin >> nombre2;
    ifstream f2;
    f2.open(nombre2);
    char nombreDest[100];
    cout << "Introduce el nombre del fichero destino: ";
    cin >> nombreDest;
    ofstream fDest;
    fDest.open(nombreDest);
    fusionarFicheros(f1, f2, fDest);
    f1.close ();
    f2.close ();
    fDest.close ( );
    return 0;
}
```

Algoritmos sobre archivos- Fusión de 2 fich de car

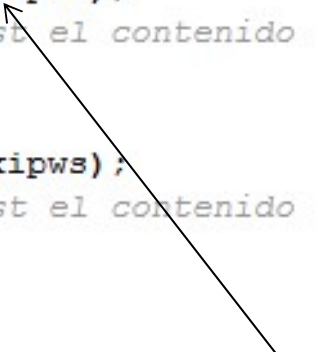
```
#include <iostream>
#include <fstream>
using namespace std;
void fusionarFicheros(ifstream& f1, ifstream& f2, ofstream& fResu);

int main()
{
    char nombre1[100];
    cout << "Introduzca nombre1: ";
    cin >> nombre1;
    ifstream f1;
    f1.open(nombre1);
    cout << "Introduzca nombre2: ";
    cin >> nombre2;
    ifstream f2;
    f2.open(nombre2);
    char nombreDest[100];
    cout << "Introduzca nombreDest: ";
    cin >> nombreDest;
    ofstream fDest;
    fDest.open(nombreDest);
    fusionarFicheros(f1, f2, fDest);
    f1.close ();
    f2.close ();
    fDest.close ();
    return 0;
}

void fusionarFicheros(ifstream& f1, ifstream& f2, ofstream& fDest)
{
    char e;
    f1.unsetf(ios::skipws);
    // Ponemos enfDest el contenido del primer fichero
    while (f1 >> e)
        fDest << e;
    f2.unsetf(ios::skipws);
    // Ponemos enfDest el contenido del segundo fichero
    while (f2 >> e)
        fDest << e;
}
```

Algoritmos sobre archivos- Fusión de 2 fich de car

```
void fusionarFicheros(ifstream& f1, ifstream& f2, ofstream& fDest)
{
    char e;
    f1.unsetf(ios::skipws);
    // Ponemos enfDest el contenido del primer fichero
    while (f1 >> e)
        fDest << e;
    f2.unsetf(ios::skipws);
    // Ponemos enfDest el contenido del segundo fichero
    while (f2 >> e)
        fDest << e;
}
```



Cualquier bandera establecida puede desactivarse.

Para desactivar una bandera usamos la función **unsetf**.

Cuando la bandera de formato **skipws** se establece, los espacios en blanco seguidos se leen y se descartan del flujo hasta leer algo que no sea blanco.

Esto se aplica a todas las operaciones de entrada con formato realizado con el operador >> en la secuencia.

Espacios de tabulación, de retornos de carro y espacios en blanco son todos considerados espacios en blanco (ver isspace).

Para flujos estándares, la bandera **skipws** se encuentra en la inicialización.

Ejercicios con pares de números

1. Mediante un programa C++, escribir un archivo binario llamado **grupo.dat**, formado por un conjunto de 200 pares de números generados aleatoriamente. Cada par de datos se conforma por un flotante y un entero.
2. Escribir un programa que abra el archivo generado en el ejercicio anterior, y solicite al usuario que ingrese un flotante, un entero y una posición.

El programa debe **sobreescribir** el par **en la posición** ingresada por el usuario, por el nuevo par.

Luego, debe mostrar la lista de datos por pantalla mostrando un par por línea.

1.

```
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <cstdlib>
using namespace std;

int main() {
    ofstream archi;
    archi.open("grupo.dat", ios::binary);
    float f; int i;

    for (int c=1; c<=200; c++){
        f=rand()/(1000.0);
        i=rand()%1000;
        cout << f << " - " << i << endl;
        archi.write((char *)&f, sizeof(f));
        archi.write((char *)&i, sizeof(i));
    }
    archi.close();

    system("pause");
}
```

2.

```
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <cstdlib>
using namespace std;

int main(){
    fstream archi;
    archi.open("grupo.dat", ios::binary | ios::in | ios::out);
    float f,nf; int i,ni, posicion;

    cout << "Posicion a sobrescribir (en 0..199) ? ";
    cin >> posicion;
    cout << "Nuevo float ? ";
    cin >> nf;
    cout << "Nuevo int ";
    cin >> ni;

    archi.seekg(posicion*(sizeof(f)+sizeof(i)),ios::beg);

    archi.read((char *) &f, sizeof(float));
    archi.read((char *) &i, sizeof(int));

    cout<<"Par de valores de la posicion " << posicion << " : " <<f<<" "<<i<<endl;

    archi.seekp(posicion*(sizeof(f)+sizeof(i)),ios::beg);
    archi.write((char *)&nf, sizeof(float));
    archi.write((char *)&ni, sizeof(int));

    archi.close();

    system("pause");
}
```

Ejercicio de transacciones



Funcionalidad:

- actualiza las cuentas existentes,
- agrega nuevas cuentas,
- elimina cuentas, y
- guarda un listado con formato de todas las cuentas actuales en un archivo de texto

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

// clientData structure definition
- struct clientData {
    unsigned int acctNum; // account number
    char lastName[ 15 ]; // account last name
    char firstName[ 10 ]; // account first name
    double balance; // account balance
};
typedef struct clientData cliente;

// prototypes
unsigned int enterChoice();
void textFile (fstream &);
void updateRecord( fstream & );
void newRecord( fstream & );
void deleteRecord( fstream & );
void mostrarLinea (ostream&, const cliente &);
int obtenerCuenta (const char * );
```

```
int main( void )
{
    fstream credito;
    credito.open("credit.dat", ios::in|ios::out);
    if ( !credito ) {
        cout << "File could not be opened." ;
    }
    else {
        unsigned int choice; // user's choice// enable user to specify action
        while ( ( choice = enterChoice() ) != 5 ) {
            switch ( choice ) {
                case 1:
                    textFile( credito );break;
                case 2:
                    updateRecord( credito );break;
                case 3:
                    newRecord( credito );break;
                case 4:
                    deleteRecord( credito );break;
                default:
                    cout << "Incorrect choice" ;break;
            }
        }
        credito.close();
    }
} // end main
```

```
unsigned int enterChoice( void )
{
    unsigned int menuChoice;
    cout <<"\nEnter your choice\n"
        <<"1 - store a formatted text file of accounts called\n"
        <<"    \"accounts.txt\" for printing\n"
        <<"2 - update an account\n"
        <<"3 - add a new account\n"
        <<"4 - delete an account\n"
        <<"5 - end program\n? " ;

    cin >> menuChoice;
    return menuChoice;
}
```

```
// create formatted text file for printing
```

```
void textFile( fstream & leerDeArchivo)
```

```
{
```

```
    ofstream salida; // accounts.txt file pointer
```

```
    cliente client ;
```

```
    salida.open( "accounts.txt", ios::out);
```

```
    if (!salida){
```

```
        cout << "File could not be opened." ;
```

```
    }
```

```
    else {
```

```
        salida << left << setw(10) << "Acct"<<setw(16) <<"Last Name"
                << setw(14)<<"First Name"<< right <<setw(10) <<"Balance"<<endl;
```

```
        leerDeArchivo.seekg (0);
```

```
        leerDeArchivo.read((char *) &client, sizeof(cliente));
```

```
        while ( !leerDeArchivo.eof() ) {
```

```
            mostrarLinea (salida,client);
```

```
            leerDeArchivo.read((char *) &client, sizeof(cliente));
```

```
        }
```

```
        leerDeArchivo.close();
```

```
    }
```

```
} // end function textFile
```

Se escriben
los títulos
del listado




```

void mostrarLinea( ostream & salida, const cliente& registro)
{
    salida << left << setw(10) << registro.acctNum<<setw(16) <<registro.lastName
        << setw(11)<< registro.firstName << right <<setw(10) <<registro.balance<<endl;
}

int obtenerCuenta (const char * indicador)
{
    int num;
    do {
        cout << indicador <<"(1-100): ";
        cin >> num;
    } while (num <1 || num > 100);
    return num;
}

```



```
// update balance in record
```

```
void updateRecord( fstream & actualizarArch )
```

```
{
```

```
    unsigned int account = obtenerCuenta ("Escriba la cuenta que desea actualizar");
```

```
    double transaction;
```

```
    // create clientData with no information
```

```
    cliente client = { 0, "", "", 0.0 };
```

```
    actualizarArch.seekg ( ( account - 1 ) * sizeof(cliente));
```

```
    actualizarArch.read((char *) &client, sizeof(cliente));
```

```
    if ( client.acctNum != 0 ) {
```

```
        mostrarLinea (cout,client);
```

```
        cout <<endl <<"Enter charge ( + ) or payment ( - ): " ;
```

```
        cin >>transaction ;
```

```
        client.balance += transaction; // update record balance
```

```
        actualizarArch.seekp ( ( account - 1 ) * sizeof(cliente));
```

```
        actualizarArch.write((char *) &client, sizeof(cliente));
```

```
        mostrarLinea (cout,client);
```

```
    }
```

```
    else cout << "La cuenta " << account <<" no tiene información" << endl;
```

```
    // create and insert record
```

```
}
```

```
void newRecord( fstream & insertarEnArch)
{
    unsigned int account = obtenerCuenta ("Escriba la cuenta que desea actualizar");
    insertarEnArch.seekg ( ( account - 1 ) * sizeof(cliente));
    cliente client ;
    insertarEnArch.read((char *) &client, sizeof(cliente));
    if ( client.acctNum != 0 ) {

        cout<< " Escriba apellido, nombre y saldo" << endl;
        cin >> client.lastName >> client.firstName>> client.balance ;

        insertarEnArch.seekp ( ( account - 1 ) * sizeof(cliente));
        insertarEnArch.write((char *) &client, sizeof(cliente));
    }
    else cout << "La cuenta " << account <<" ya tiene informacion" << endl;
}
```

```

void deleteRecord( fstream & eliminarEnArch)
{
    unsigned int account = obtenerCuenta ("Escriba la cuenta que desea eliminar");
    eliminarEnArch.seekg ( ( account - 1 ) * sizeof(cliente));
    cliente client ;
    eliminarEnArch.read((char *) &client, sizeof(cliente));
    if ( client.acctNum != 0 ) {
        cliente client = { 0, "", "", 0.0 };
        eliminarEnArch.seekp ( ( account - 1 ) * sizeof(cliente));
        eliminarEnArch.write((char *) &client, sizeof(cliente));
        cout << " La cuenta"<<account << "ha sido eliminada";
    }
    else cout << "La cuenta " << account <<" esta vacia" << endl;
}

```

LEER

Capítulo 8

“Flujos de archivos de E/S y archivos de datos”

Libro: “C++ para ingeniería y ciencias” 2da edición -

Gary J. Bronson, pág. 443