Tema 2: Ficheros

Índice

Introducción

¿Por qué estudiar los ficheros?

Los Tipos de Datos (TDs) vistos hasta ahora (es decir, tipos de datos básicos, vectores y registros) no son suficientes para resolver cualquier problema. El TD fichero nos permitirá subsanar algunas de las deficiencias que presentan los **TD vistos hasta ahora**.

Los algoritmos no trabajan directamente con datos sino con representaciones de éstos, es decir, con las variables. Al ejecutar el algoritmo, para cada variable se reserva una zona en memoria RAM. Gracias a esto, el tiempo de acceso a los datos es pequeño.

Todos los programas vistos hasta ahora trabajan con información almacenada en memoria principal:

Esta forma de trabajar presenta **dos limitaciones**:

1. La falta de información permanente: una vez finalizada la ejecución del programa, la información desaparece (memoria volátil). Si interesa mantener la información después de cada ejecución, ¿cómo lo hacemos?.
2. Limitación en cuanto al espacio disponible para los datos, en dos aspectos:
    a. El tamaño de la memoria es limitado. Puede darse el caso de que los datos con los que necesita trabajar el programa son demasiado grandes (ocupan mucha memoria) para que entren en memoria principal.
    b. El tamaño (declarado) de la estructura de datos (fijo) también es limitado.

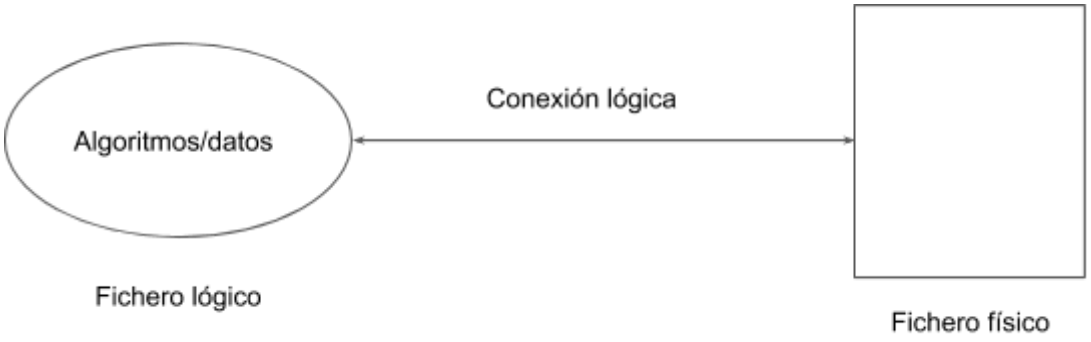
En dichos casos se utilizarán ficheros para contener información en memoria secundaria (disco duro, memorias USB, etc.).

Los ficheros tienen **las siguientes características principales** que vienen a corregir las limitaciones de los TDs vistos hasta ahora:

- Número de datos indefinido (indeterminado) y limitado únicamente por el tamaño del dispositivo de almacenamiento.
- Información permanente.

El uso de ficheros tiene la desventaja de que el tiempo de acceso a la información aumenta.

Intuitivamente un fichero es “algo” que nos permitirá almacenar datos en soportes permanentes de información de forma que una vez que acabe la ejecución del programa los datos permanezcan. Pero a la hora de hablar de ficheros, hay que hacer una distinción entre **ficheros físicos o externos** y **ficheros lógicos o internos**.

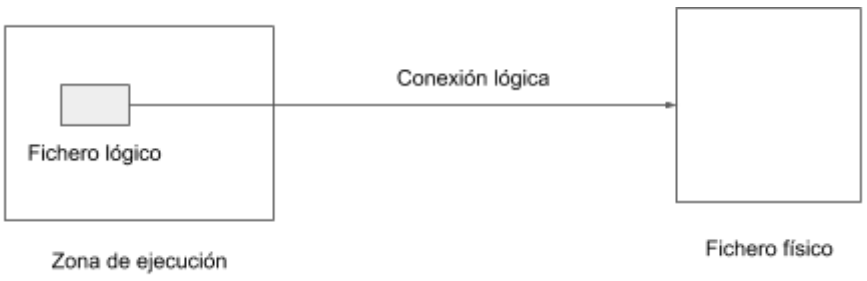


Un **fichero físico** es un conjunto de valores almacenados en algún dispositivo de almacenamiento externo. La gestión la hace el sistema operativo, por eso los ficheros físicos tienen nombre (y extensión) según las reglas del sistema operativo.

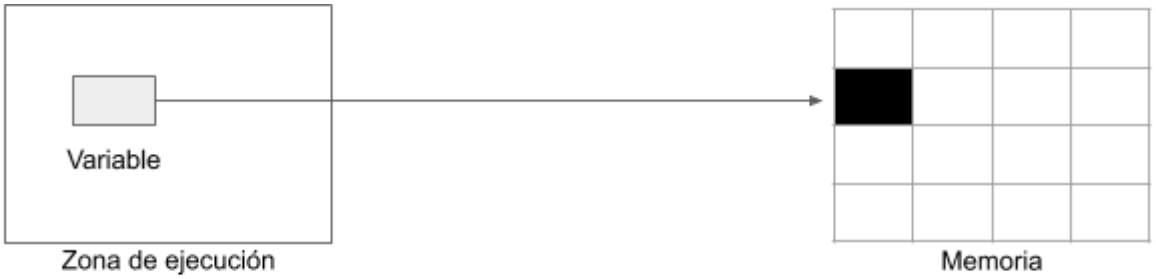
Pero, para trabajar con un fichero físico o externo dentro de un algoritmo tendremos que declarar una variable que lo represente y con la cual trabajar. Dicha variable será un **fichero lógico o interno**.

Los algoritmos trabajan con el contenido del fichero físico pero no directamente, sino a través de una variable (fichero lógico) que representa al fichero físico dentro del algoritmo (o programa).

Para trabajar con un fichero en un algoritmo hay que asociar el fichero físico al lógico, es decir, **establecer una conexión lógica**. Para ello (es decir, para relacionar el fichero lógico con el fichero físico) necesitamos realizar una **operación de apertura del fichero**.



Se puede realizar el siguiente símil con las variables y las posiciones de memoria. Al igual que no trabajamos directamente con la memoria, sino con el nombre de la variable; no trabajamos directamente con el fichero físico, sino con el nombre lógico del fichero.



Ficheros físicos

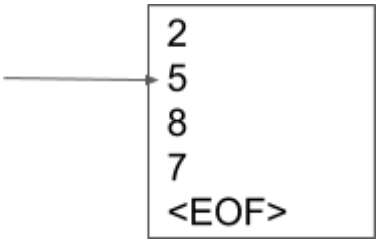
Características

Los ficheros físicos tienen las siguientes características:

- El tamaño de un fichero es indeterminado y limitado únicamente por la capacidad del dispositivo de almacenamiento. Esto implica que el tamaño del fichero NO es conocido por lo que no puede usarse.
- Todos los datos de nuestros ficheros serán del mismo tipo (a pesar de que el lenguaje sea más permisivo). Como veremos más adelante los algoritmos que trabajan con ficheros son muy dependientes de la estructura del fichero por lo que son susceptibles a múltiples errores.

Elementos especiales de un fichero físico

Supongamos el siguiente ejemplo de fichero físico:



En dicho fichero podemos distinguir los siguientes elementos especiales:

1. **Apuntador** (representado por una flecha) es un elemento que nos indica el dato del fichero al que tenemos acceso. Indica cuál es el dato que se va a tomar si hacemos una operación de lectura sobre el fichero (Ej: si leemos, leeremos el dato 5. Al leerlo, el apuntador avanzará al siguiente dato).
2. **<EOF>** es la marca de fin de fichero (similar a la marca de fin de secuencia). No es algo físico. Se puede interpretar como algo booleano que toma el valor:
    - **verdad** cuando el apuntador apunta al final del fichero, o

- **falso** cuando el apuntador apunta a cualquier elemento (incluida la marca de fin).

Clasificación de los ficheros físicos

Atenderemos a dos criterios para clasificar los ficheros: según el modo de acceso a los datos y según el modo de almacenamiento de los datos.

Según el modo de acceso a los datos

Al estar en memoria secundaria, no todos los elementos del fichero son accesibles de forma inmediata. Solamente se puede acceder cada vez a un único elemento del fichero, el indicado por el apuntador (llamado ventana del fichero).

Dependiendo de cómo se desplaza la ventana por el fichero, podemos distinguir dos tipos de acceso:

1. **Ficheros secuenciales (acceso secuencial):** para acceder a un dato pasaremos previamente por todos los anteriores (como ocurría con el tratamiento secuencial). Nuestros ficheros serán secuenciales (por eso, nuestros esquemas serán análogos a los de las secuencias). *La ventana del fichero o apuntador sólo puede moverse hacia delante a partir del primer elemento y siempre de uno en uno.*
2. **Ficheros de acceso directo (acceso directo):** se puede acceder directamente a cualquier dato. La ventana del fichero o apuntador se puede situar directamente en cualquier posición del fichero. Es el mismo acceso utilizado en los **arrays**.

El **acceso directo** suele ser más eficiente, ya que para leer un dato no hace falta leer antes todos los anteriores. La razón por la que existe el **acceso secuencial** es que existen dispositivos de memoria secundaria que sólo admiten acceso secuencial (como por ejemplo las cintas). Además, el acceso secuencial se utiliza también en dispositivos que admiten acceso directo cuando queremos leer los elementos de forma secuencial, ya que **este acceso es más sencillo**.

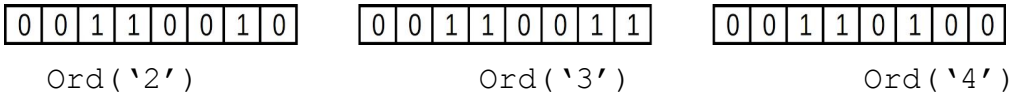
Según el modo de almacenamiento de los datos

Aquí también distinguiremos dos categorías:

1. **Ficheros binarios (o sin formato):** Los elementos se almacenan en el fichero exactamente igual que están almacenados en memoria principal, es decir, al leer o escribir no se realiza ningún tipo de conversión (los datos se almacenan como sucesión de bytes, codificados del mismo modo que en memoria). Ej: 234

Aunque pueden almacenar cualquier tipo de dato, nosotros trabajaremos con ficheros cuyos datos sean todos del mismo tipo.

2. **Ficheros de texto *(o con formato)***: La información se almacena como si se escribiese por pantalla, carácter a carácter. Todo lo que se codifica son caracteres. Pueden almacenarse números pero tratados como secuencias de caracteres. Ej: 234.



Codifican en cada byte el código ASCII del carácter. Los datos estructurados (**vectores y registros**, entre otros) no pueden almacenarse como estructuras sino que debemos tratarlos como una colección de datos simples.

El teclado (entrada estándar) y la pantalla (salida estándar) se consideran también ficheros de texto.

Procesamiento de un fichero

Como hemos comentado anteriormente, siempre que queramos trabajar con un fichero físico o externo dentro de un algoritmo tenemos que declararnos una variable que lo represente o **fichero lógico** con la cual trabajaremos. Para cualquier operación que queramos realizar sobre el fichero se debe seguir el siguiente esquema:

1. **Declarar** una variable de tipo fichero.
2. **Apertura** de Fichero – establecer conexión lógica.
    - Comprobar que conexión se ha establecido
3. Operaciones.
4. **Cierre** del fichero – cerrar conexión lógica.

Notar que a la hora de establecer la conexión lógica entre el fichero físico y el lógico, es decir, realizar la apertura del fichero, puede haber problemas; es por eso que hay que comprobar que la conexión se ha establecido.

Ficheros en C++

C++ soporta los dos tipos de ficheros: de texto y binarios. Como ya comentamos anteriormente, los primeros almacenan datos como códigos ASCII. Los valores simples tales como números y caracteres están separados por espacios en blanco o retornos de carro. Los segundos almacenan bits de forma directa (por lo que no se necesitan separadores) y se necesita usar la dirección de una posición de almacenamiento.

En C++ un fichero es simplemente un flujo externo que se puede abrir para ***entrada*** (dando lugar a lo que llamaremos **archivo o fichero de entrada**), para ***salida*** (dando lugar a lo que llamaremos **archivo o fichero de salida**) o para ***entrada/salida*** (**archivo o fichero de entrada/salida o E/S**).

Observación: una biblioteca en C++ que proporciona "funciones" y operadores para el manejo de ficheros es la biblioteca **fstream**. Luego, para trabajar con ficheros tendremos que incluir este fichero de cabecera.

```
#include <fstream>
```

Declaración de variables de tipo fichero

La declaración de variables de tipo fichero (fichero lógico) dependerá de si queremos declarar un fichero de salida o un fichero de entrada: **se declara un fichero de entrada (o de lectura)** para, a través de él, obtener los datos que hay almacenados en el fichero físico. **Se declara un fichero de salida (o de escritura)** para, a través de él, introducir datos en el fichero físico.

En C++ utilizaremos la siguiente sintaxis.

- Para **declarar un fichero de entrada**

  ```
  ifstream <nombre_fichero_logico>;
  ```

- Para **declarar un fichero de salida**

  ```
  ofstream <nombre_fichero_logico>;
  ```

- Para declarar un fichero como "genérico", es decir, sin especificar si va a ser para lectura o escritura:

  ```
  fstream <nombre_fichero_logico>;
  ```

El hecho de que sea de entrada o de salida (lectura o escritura) se indica en la apertura del fichero.

Apertura del fichero - Estableciendo la conexión lógica

Distinguiremos si el fichero es de texto o binario. Primero veremos el caso de los **ficheros de texto** y luego indicaremos cómo se realiza la apertura para ficheros binarios.

La apertura se realiza con la instrucción **open** utilizando la siguiente sintaxis:

```
<nombre_fichero_logico>.open(<nombre_fichero_fisico>,modo);
```

Siendo:

- `<nombre_fichero_logico>` una variable declarada de tipo fichero.
- `<nombre_fichero_fisico>` una cadena de caracteres que representa el nombre del fichero físico. Esta cadena de caracteres puede ser:
  - Una constante → "..."
  - Una variable de tipo cadena. En C++ un vector de caracteres (donde `\0` marca el fin de la cadena).

    ```
    char <nombre_fichero_físico> [dimensión];
    ```
- `modo` puede ser una de las siguientes opciones o una combinación de varias de ellas usando |:
  - `ios::in` - modo entrada (lectura).
  - `ios::out` - modo salida (escritura).
  - `ios::app` - modo añadir al final.
  - `ios::binary` - fichero binario (si no se pone, por defecto es de texto)

Notas.

- Por defecto los **ficheros de entrada** (`ifstream`) se abren **sólo para lectura (o en modo lectura)** poniendo el apuntador al principio. Éste apuntará al primer elemento del fichero (si el fichero no está vacío) o a <EOF> si está vacío. Además el fichero debe existir, si no se genera un error. A continuación se muestra un ejemplo:

  ```
  ifstream <nombre_fichero_lógico>;
  <nombre_fichero_lógico>.open(<nombre_fichero_físico>,ios::in);
  <nombre_fichero_lógico>.open(<nombre_fichero_físico>,ios::in|ios::binary);
  ```

- Por defecto, los **ficheros de salida** (`ofstream`) se abren **sólo para escritura (o en modo escritura)** poniendo el apuntador al principio. Si el fichero con ese nombre no existe, se crea un fichero físico vacío con ese nombre. Si el fichero ya existe lo vacía.

  ```
  ofstream <nombre_fichero_lógico>;
  <nombre_fichero_lógico>.open(<nombre_fichero_físico>,ios::out):
  <nombre_fichero_lógico>.open(<nombre_fichero_físico>,ios::out|ios::binary);
  ```

- Si queremos abrir un **fichero de salida en modo añadir** utilizaremos el modo `ios::app`. En este caso no se borra el fichero y el apuntador del fichero se pone después del último elemento. Sin el fichero no existe da error.

  ```
  ofstream <nombre_fichero_lógico>;
  <nombre_fichero_lógico>.open(<nombre_fichero_físico>,ios::app);
  ```

A continuación veremos una serie de ejemplos.

1. Abrir para lectura (como fichero de entrada) el fichero de texto “mio.txt” y lo asocia al fichero lógico `canalE`.

```
ifstream canalE;
canalE.open("mio.txt",ios::in);
```

2. Abrir para escritura (modo salida) el fichero de texto “salida.dat” (si el fichero no existe lo crea y si existe borra su contenido) asociándolo al descriptor `canalS`.

```
ofstream canalS;
canalS.open("salida.dat",ios::out);
```

3. Abrir el fichero binario “F2.txt” como fichero de salida en modo añadir.

```
fstream canalES;
canalES.open("F2.txt",ios::app|ios::binary);
```

Apertura segura de ficheros

Si al intentar abrir un fichero en modo lectura se produce algún error (por ejemplo que no exista), cualquier operación de lectura posterior dará error. Antes de empezar a leer o escribir en un fichero es siempre conveniente verificar que la operación de apertura se realizó con éxito.

Si hay algún problema en la *apertura*, por ejemplo abrir en modo lectura un *fichero* que no existe, el descriptor (la variable) toma un valor que se interpreta como `false`. Por ello, para controlar que la apertura del fichero no ha dado error haremos lo siguiente:

```
if (<nombre_fichero_lógico>) {
   // Se opera sobre el fichero
}
```

La condición anterior se debe poner siempre que abramos un fichero, puesto que si ha habido un error, no se podrá realizar ninguna operación con él.

Operaciones sobre ficheros

Existen una serie de esquemas sobre ficheros que veremos más adelante.

Cierre del fichero

Un fichero anteriormente abierto y con un descriptor (fichero lógico) asociado deber ser cerrado con el fin de liberar los recursos asociados a él de la siguiente forma:

```
<nombre_fichero_lógico>.close();
```

La operación anterior tiene la siguiente semántica:

1. Se rompe la conexión lógica.
2. Se "cierra" el fichero (para poder volver a utilizarlo).

Detección de fin de fichero

¿Cómo saber cuándo se ha llegado al final del fichero? Para ello debemos detectar el <EOF>, pero ¿cómo? En C++ para detectar el <EOF> hay que leerlo. ¿Cómo saber si se ha leído la marca fin?

Lo primero que se nos ocurre es comparar el dato leído x con <EOF> pero: `x=<EOF>` produce un **error**, ya que <EOF> no es del tipo de los elementos del fichero. Por lo tanto, es necesario utilizar una función que nos permita detectar el fin del fichero. En concreto la sintaxis de dicha función es la siguiente:

```
<nombre_fichero_lógico>.eof();
```

La función `eof()` devuelve un valor distinto de cero (`true`) si el dato que hemos leído era un final de fichero y `false` en caso contrario. Por esta razón hay que leer primero el carácter y después comprobar si hemos llegado a final de fichero.

Si el apuntador apunta a `<EOF>` y leemos del fichero no da error pero el valor de la variable sobre la que se ha leído es indeterminado.

Todas las instrucciones vistas hasta ahora sirven tanto para ficheros binarios como para los de texto. La única diferencia que hemos visto es que en la función `open` debemos especificar el tipo de fichero. Las diferencias entre ambos están en las operaciones de lectura y escritura.

Operaciones sobre ficheros

A continuación veremos las operaciones de eliminación, renombrado, lectura y escritura sobre ficheros de texto y binarios.

Eliminación

Para eliminar un fichero se utiliza la función `remove`:

```
remove(<nombre_fichero_fisico>);
```

Renombrado

Para cambiar el nombre de un fichero se utiliza la función `rename`:

```
rename(<nombre_fichero_anterior>,<nombre_fichero_nuevo>);
```

La instrucción anterior renombra el fichero cuyo nombre se pasa con el primer parámetro con el nombre que se le pasa como segundo parámetro.

Lectura y escritura de ficheros de texto

Las operaciones de lectura y escritura sobre ficheros de texto son exactamente las mismas que se utilizan sobre `cin` (para ficheros de entrada) y `cout` (para ficheros de salida), es decir:

- el operador << para escritura y
- el operador >> para lectura.

De hecho `cin` y `cout` son ficheros predefinidos, que están asociados con la entrada estándar y la salida estándar del sistema operativo.

Ejemplo.
Para escribir el número 10 en el fichero `f`:

```
f << 10;
```

Para leer un entero del fichero `f`:

```
int dato;
f >> dato;
```

Sin embargo, lo normal es que no sepamos el número de elementos que vamos a leer, sino que queremos leer hasta que lleguemos al final del fichero. Para ello se puede utilizar un bucle `while` de la siguiente forma.

```
int dato;
f >> dato;
while(!f.eof()){
    // Procesar dato
    f >> dato;
}
```

De esta forma, el código anterior leerá (hasta llegar al final del fichero) todos los números del fichero.

Esta forma de leer el fichero se puede utilizar con cualquier tipo de lectura con >> y también con `getline` (comando que almacena el resultado en un objeto `string`).

Queda ahora la cuestión del tipo que vamos a utilizar para declarar `dato`. Para ello existen dos opciones.

1. Declarar dato de tipo `char`. De este modo se puede mostrar por pantalla el contenido del fichero, pero para almacenar los datos, no lo hemos hecho carácter a carácter como hemos visto anteriormente.
2. Dar a `dato` el tipo original, de esta manera la información se puede recuperar tal y como se almacenó. Aquí es importante resaltar dos cosas:
   - ¿Qué hace `cin >> dato` (o en general `f >> dato`)? Va al buffer original y toma caracteres que pueden formar parte del `dato`. Ejemplo: si el buffer es 34.4a26, el resultado será distinto dependiendo del tipo de `dato`:
     - `int dato` → 34
     - `float dato` → 34.4
     - `string dato` → 34.4a26
   - Es muy importante que en la especificación (precondición) del problema aparezca el tipo de los datos del fichero, de lo contrario se pueden producir errores. Por ejemplo, si consideramos el siguiente programa con el siguiente fichero, al llegar a la letra A del fichero se produce un bucle infinito ya que el programa no consigue avanzar.

```
int dato;
...
if(f){
   f >> dato;
   while(!f.eof()){
      cout << dato <<endl;
      f >> dato;
   }
}
```

Fichero

```
2
35
100
A
26
<EOF>
```

Ejemplo.
Crear un fichero de números enteros llamado (NUMEROS.DAT) con los números enteros del 1 al 100. Modificar el algoritmo para que el nombre del fichero se solicite por teclado.

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

using namespace std;

int main(){
   int i;
   ofstream canalS;
   canalS.open("NUMEROS.DAT",ios::out);
   if(canalS){
```

```
      for(i=1;i<=100;i++){
         canalS<<i<<endl;
      }
      canalS.close();
   }else{
     cout << "Incapaz de crear o abrir el fichero." << endl;
   }
   system("Pause");
   return 0;
}
```

Variante leyendo el nombre desde teclado.

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

using namespace std;

int main(){
   char nombreFichero[20];
   int i;
   ofstream canalS;
   cout<<"Introduce el nombre del fichero :"<<endl;
   cin>>nombreFichero;
   canalS.open(nombreFichero, ios::out);
   if(canalS){
      for(i=1;i<=100;i++){
         canalS<<i<<endl;
      }
      canalS.close();
   }else{
     cout << "Incapaz de crear o abrir el fichero." << endl;
   }
   system("Pause");
   return 0;
}
```

Ejemplo.

Diseñar un programa que cuente el número de elementos que hay almacenados en un fichero de reales. El nombre del fichero deberá solicitarse por teclado.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    char nombreFichero[20];
    float dato;
    int i=0;
    ifstream canalE;
    cout<<"Introduce el nombre del fichero :"<<endl;
    cin>>nombreFichero;
    canalE.open(nombreFichero, ios::in);
    if(canalE)
    {
        canalE>>dato;
        while(!canalE.eof())
        {
            i++;
            canalE>>dato;
        }
```

```
        cout<<"El fichero tiene "<<i<<" entradas."<<endl;
        canalE.close();
    }
    else{
        cout<<"Incapaz de crear o abrir el fichero "<<nombreFichero<<endl;
    }
    system("Pause");
    return 0;
}
```

Lectura y escritura en ficheros binarios

Recordad que un fichero binario es una colección de bytes. Por lo tanto, en un fichero binario los datos se codifican del mismo modo que en la memoria, es decir, en binario utilizando un determinado número de bytes para cada tipo de dato. Así que al escribir o leer los datos de un fichero, *los datos se leen o escriben sin ningún tipo de transformación; es decir, es un volcado directo de memoria a fichero o de fichero a memoria.*

Ejemplo. Entero → 2 bytes; real → 4 bytes; carácter → 1 byte; etc.

Observad que en la declaración de variables de tipo fichero en C++ no se especifica el tipo de los elementos del fichero. Éste tendremos que indicarlo en las operaciones de lectura y escritura. Ejemplos:

- Para leer un entero de un fichero de enteros → coge dos bytes y decodifica el entero correspondiente.
- Para escribir un entero → codifica el entero usando dos bytes.

Por lo tanto, va a ser necesario conocer el número de bytes utilizados para codificar cada elemento de un fichero. Normalmente, para expresar el tamaño o número de bytes a leer/escribir se utiliza la función `sizeof` con la siguiente sintaxis:

```
sizeof(<tipo_dato>)
```

La instrucción `sizeof(<tipo_dato>)` devuelve el número de bytes utilizados para almacenar un dato de tipo `<tipo_dato>`. Ejemplos:

- `sizeof(short)` → `2`
- `sizeof(int)` → `4`
- `sizeof(float)` → `4`
- `sizeof(char)` → `1`
- `struct tPersona{...}; ¿sizeof(tPersona)?`

También es posible utilizar la función `sizeof` con la siguiente sintaxis:

```
sizeof(<expresion>)
```

La instrucción `sizeof(<expresion>)` devuelve el número de bytes utilizados para almacenar un dato del tipo devuelto por `<expresion>`. Ejemplos:

- `sizeof(4)` → `4` (ya que `4` es de tipo `int`, y por lo tanto se almacena con 4 bytes).
- `sizeof(4.2+5.3)` → `4` (ya que el resultado de `4.2+5.3` es de tipo `float`, y por lo tanto se almacena con 4 bytes).

En las operaciones de lectura/escritura debemos indicar cuántos bytes recoger/escribir del/en el fichero.

En C++ la **lectura en binario** se realiza mediante el método `read` y la **escritura** mediante el método `write`. En ambos casos hay que pasar como primer parámetro un puntero de tipo char a la variable a leer o escribir (de cualquier tipo) (pasada por referencia), y como segundo dato el número de bytes a leer o escribir.

Para leer un dato del fichero utilizaremos la siguiente sintaxis:

```
<nombre_fichero_logico>.read((char *)& dato, sizeof(dato));
```

Y para escribir en el fichero, la sintaxis es la siguiente:

```
<nombre_fichero_logico>.write((char *)& dato, sizeof(dato));
```

Ejemplo.
Crear un fichero binario de números enteros llamado (NUMEROS.DAT) con los números enteros del 1 al 100, y posteriormente leer dicho fichero.

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

using namespace std;

int main()
{
    int i,j;
    ofstream canalS;
    canalS.open("NUMEROS.DAT", ios::out|ios::binary);
    if(canalS)
    {
        for(i=1; i<=100; i++)
        {
            canalS.write((char *)& i, sizeof(i));
        }
        canalS.close();
        ifstream canalE("NUMEROS.DAT", ios::out | ios::binary);

        if(canalE)
        {
            canalE.read((char *)& j, sizeof(j));
            while(!canalE.eof())
            {
                cout<<" tiene "<<j<<endl;
                canalE.read((char *)& j, sizeof(j));
```

```
            }
            canalE.close();
        }
        else
        {
            cout << "Incapaz de abrir el fichero." << endl;
        }
    }
    else
    {
        cout << "Incapaz de crear el fichero." << endl;
    }
    system("Pause");
    return 0;
}
```

Lectura y escritura de estructuras

A la hora de almacenar registros en ficheros, podemos trabajar tanto con ficheros de texto como con ficheros binarios. En el caso de ficheros de texto, consideramos que cada línea del fichero contiene los datos de un registro. Por ejemplo, si tenemos en un fichero de texto los datos de los empleados de un empresa, y cada empleado lo representamos con el registro:

```
struct tEmpleado{
    char dni[9];
    char nombre[20];
    int horasTrabajadas;
    int precioHora;
};
```

Nuestro fichero almacenará algo como lo siguiente:

```
12345678 Bea 12 24
23456789 Jose 23 12
...
```

Si queremos definir un subprograma que lea el contenido del fichero y lo muestre por pantalla deberíamos utilizar algo como lo siguiente:

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

using namespace std;

int main()
{
    tEmpleado empl;
    ifstream canalE;//Declaramos una variable de tipo fichero de entrada
    canalE.open("Empleados.txt",ios::in); //Establecemos la conexión lógica
    if(canalE)
    {
        canalE>>empl.dni;
        while(!canalE.eof())
        {
            //Terminamos de leer la fila
            canalE>>empl.nombre;
            canalE>>empl.horasTrabajadas;
```

```
            canalE>>empl.precioHora;
            cout<<"-----------------------------------------------"<<endl;
            cout<<"DNI: "<<empl.dni<<endl;
            cout<<"Nombre: "<<empl.nombre<<endl;
            cout<<"Horas trabajadas: "<<empl.horasTrabajadas<<endl;
            cout<<"Precio hora: "<<empl.precioHora<<endl;
            cout<<"-----------------------------------------------"<<endl;
            canalE>>empl.dni;
        }
        canalE.close();
    }
    else
    {
        cout<<"Problemas a la hora de abrir el fichero Empleados.txt"<<endl;
    }
}
```

Por el contrario, si hemos almacenado nuestros empleados en un fichero binario, la forma correcta de leerlos sería la siguiente:

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

using namespace std;

int main()
{
    tEmpleado empl;
    ifstream canalE;//Declaramos una variable de tipo fichero de entrada
    canalE.open("Empleados.dat",ios::in|ios::binary); //Establecemos la conexión lógica
    if(canalE)
    {
        canalE.read((char *)&empl,sizeof(empl));
        while(!canalE.eof())
        {
            cout<<"-----------------------------------------------"<<endl;
            cout<<"DNI: "<<empl.dni<<endl;
            cout<<"Nombre: "<<empl.nombre<<endl;
            cout<<"Horas trabajadas: "<<empl.horasTrabajadas<<endl;
            cout<<"Precio hora: "<<empl.precioHora<<endl;
            cout<<"-----------------------------------------------"<<endl;
        canalE.read((char *)&empl,sizeof(empl));
        }
        canalE.close();
    }
    else
    {
        cout<<"Problemas a la hora de abrir el fichero Empleados.dat"<<endl;
    }
}
```

Esquemas de operaciones

Existen una serie de operaciones muy habituales al trabajar sobre ficheros que pueden ser clasificadas en dos tipos: operaciones de consulta y operaciones de modificación. A continuación, vamos a dar esquemas que sirvan para entender la filosofía de la operación a realizar (los esquemas que daremos suponen que estamos trabajando con ficheros binarios, pero el caso de ficheros de texto es

análogo). En general, estos esquemas requerirán de una adaptación al problema concreto.

Operaciones de consulta

Existen dos clases de operaciones de consulta: de recorrido y de búsqueda.

Recorrido

En el esquema de *recorrido* partimos del principio y llegamos al final del fichero. Para ello utilizamos el ciclo: leer - comprobar - procesar:

```
ifstream f; char nombref[30];
…
f.open(nombref,ios::in|ios::binary);
if(f){
  f.read((char*) & dato,sizeof(dato));
  while(!f.eof()){
     // tratamiento
     f.read((char*) & dato,sizeof(dato));
  }
  …
  f.close()
}
```

Búsqueda

En el esquema de *búsqueda* se recorre parte de un fichero, aunque en ocasiones la búsqueda debe recorrer todo el fichero. Existen dos esquemas de búsqueda.

Caso 1. Tratamiento dentro del ciclo

```
ifstream f; char nombref[30];
bool encontrado = false;
f.open(nombref,ios::in|ios::binary);
if(f){
  f.read((char*) & dato,sizeof(dato));
  while(!f.eof() && !encontrado){
     if(dato es elemento buscado){
        encontrado=true;
        // tratamiento
     }
     f.read((char*) & dato,sizeof(dato));
  }
  f.close()
}
```

Caso 2. Tratamiento fuera del ciclo

```
ifstream f; char nombref[30];
bool encontrado = false;
f.open(nombref,ios::in|ios::binary);
```

```
if(f){
  f.read((char*) & dato,sizeof(dato));
  while(!f.eof() && !encontrado){
     if(dato es elemento buscado){
        encontrado=true;
     }else{
        f.read((char*) & dato,sizeof(dato));
     }
  }
  if(!f.eof() && encontrado){
     // Tratamiento
  }
  f.close()
}
```

Operaciones de modificación

Las operaciones de modificación suponen siempre modificar el contenido de un fichero existente. Para ello se utiliza un fichero auxiliar en el que ir escribiendo. Los pasos para las operaciones de modificación son los siguientes:

1. Abrir fichero original en modo lectura.
2. Crear fichero auxiliar en modo escritura.
3. Manipular los ficheros de forma que al final el contenido del fichero auxiliar sea el del fichero resultado.
4. Cerrar ambos ficheros.
5. Borrar el fichero original.
6. Cambiar el nombre del auxiliar al nombre del fichero original.

Existen tres operaciones de modificación: inserción, eliminación y modificación. Sus esquemas son los siguientes.

Inserción

Añadir datos

```
ifstream fo; ofstream faux; char nombref[30];
fo.open(nombref,ios::in|ios::binary);
faux.open("Auxiliar.DAT",ios::out|ios::binary);
if(fo && faux){
  fo.read((char *) &dato,sizeof(dato));
  while(!fo.eof() && dato < datoAinsertar){
     faux.write((char *) &dato,sizeof(dato));
     fo.read((char *) &dato,sizeof(dato));
  }
  faux.write((char *) &datoAinsertar,sizeof(dato));
  while(!fo.eof()){
     faux.write((char *) &dato,sizeof(dato));
     fo.read((char *) &dato,sizeof(dato));
  }
  fo.close(); faux.close();
```

```
  remove(nombref); rename("Auxiliar.DAT",nombref);
}
```

Eliminación

Eliminar datos

```
ifstream fo; ofstream faux; char nombref[30];
fo.open(nombref,ios::in|ios::binary);
faux.open("Auxiliar.DAT",ios::out|ios::binary);
if(fo && faux){
  fo.read((char *) &dato,sizeof(dato));
  while(!fo.eof()){
     if(dato no es datoAeliminar){
        faux.write((char *) &dato,sizeof(dato));
     }
     fo.read((char *) &dato,sizeof(dato));
  }
  fo.close(); faux.close();
  remove(nombref); rename("Auxiliar.DAT",nombref);
}
```

Modificación

Cambiar algún dato del fichero.

```
ifstream fo; ofstream faux; char nombref[30];
fo.open(nombref,ios::in|ios::binary);
faux.open("Auxiliar.DAT",ios::out|ios::binary);
if(fo && faux){
  fo.read((char *) &dato,sizeof(dato));
  while(!fo.eof()){
     if(dato es datoAmodificar){
         // Modificar dato
     }
     faux.write((char *) &dato,sizeof(dato));
     fo.read((char *) &dato,sizeof(dato));
  }
  fo.close(); faux.close();
  remove(nombref); rename("Auxiliar.DAT",nombref);
}
```

Ficheros como parámetros de subalgoritmos

Si se tienen subalgoritmos que manejan ficheros y un algoritmo principal que realiza llamadas a los subalgoritmos, una de las cuestiones fundamentales es decidir **dónde se establece la conexión lógica** entre el fichero físico y lógico.

Hay dos posibilidades:

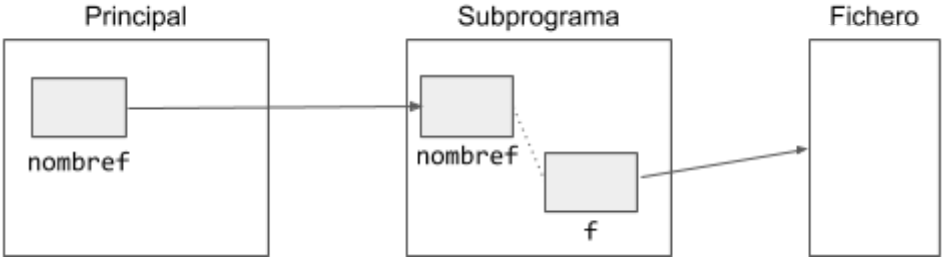
- Establecerla en el subalgoritmo.
- Establecerla en el principal (en la zona donde se realizan las llamadas al subalgoritmo).

En algunos casos podrá elegirse cualquiera de las dos opciones. En otros será necesario, o por lo menos conveniente, hacerlo de una de las dos formas.

Vamos a analizar ambos casos. Para ello, *suponemos que el nombre del fichero físico es una variable y que está leída en en principal.*

Apertura en el subalgoritmo

La conexión lógica se establece en el subalgoritmo, por lo que el subalgoritmo necesita el nombre del fichero físico, información que tenemos en el principal. ¿Cómo se pasa esa información al subalgoritmo? A través de parámetros, en este caso un parámetro de entrada de tipo cadena. Además será necesario declarar una variable de tipo fichero en el subalgoritmo.



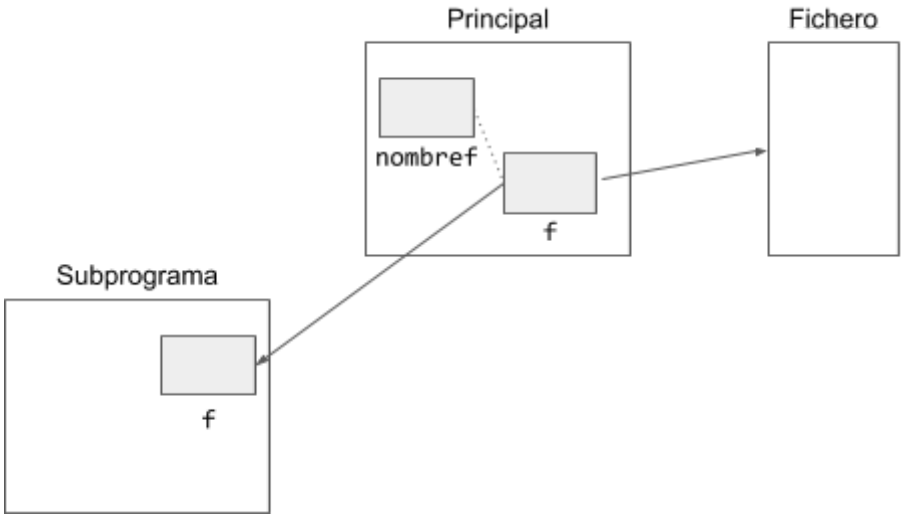
Definición del subalgoritmo:

```
void nombreAccion (char nombreFisico[])
{
    //Declarar una variable de tipo fichero
    // Apertura del fichero físico con nombre nombreFisico
    .....
    //Cierre del fichero físico
}
```

La llamada desde el principal la haremos mediante la instrucción: `nombreAccion(<nombre_fichero_fisico>)`.

Apertura en el principal

La conexión lógica se establece en el principal, por lo que el subalgoritmo recibe el fichero ya abierto y asociado. Así, el subalgoritmo **no** necesita el nombre del fichero físico.



El fichero pasa abierto y como parámetro de tipo fichero de e/s (ya sea porque se escribe en él o porque se lee, en ambos casos se modifica el apuntador). **Los ficheros se pasan siempre por referencia**.

Al contrario que en el caso anterior los ficheros se cierran en el principal.

Definición del subalgoritmo:

```
void nombreAccion (ifstream/ofstream ficheroLogico)
{
    // Para referirnos al fichero físico con ficheroLogico
}
```

La llamada desde el principal la haremos mediante la instrucción: `nombreAccion(<nombre_fichero_logico>)`.

Ahora queda la cuestión de cómo decidir qué opción usar ante un problema. Utilizaremos la segunda opción cuando:

1. El subalgoritmo necesite recibir el fichero con el apuntador en una posición determinada (que puede no ser al principio del fichero).
2. El subalgoritmo modifica el apuntador y es necesario que la posición de éste pase al principal.

Ejemplo.
Dados dos ficheros de números enteros de nombres UNO.DAT y DOS.DAT, posiblemente de tamaños diferentes, formar un tercer fichero cuyos elementos resultan de tomar alternativamente un elemento de cada fichero. El nombre del fichero a crear se ha de solicitar por teclado.

Notas.

Debemos recorrer ambos ficheros. Podemos hacerlo en dos pasos:

1. Mientras que queden elementos en ambos ficheros (no hemos llegado al final de ninguno), leemos un dato de cada fichero y lo escribimos en el fichero resultado.
2. Cuando se termina de recorrer (al menos) un fichero, debemos terminar de recorrer el otro, escribiendo en el fichero resultado. Pueden darse los casos:
    a. Se termina ficheroA: copiamos el resto de ficheroB.
    b. Se termina ficheroB: copiamos el resto de ficheroA.
    c. Se terminan ambos: no hacer nada.

Esto nos lleva a diseñar un subalgoritmo al que se le pasan dos ficheros abiertos: uno en modo lectura y otro en modo escritura y debe copiar en el segundo fichero los datos del primer fichero que quedan por recorrer.

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

using namespace std;

void copiarResto(ifstream & canalE, ofstream & canalS);

int main(){
    int datoA,datoB;
    char nombreFicheroA[20], nombreFicheroB[20], nombreFicheroS[20];
    ifstream canalEA;//Declaramos un fichero lógico de entrada
    ifstream canalEB;//Declaramos un fichero lógico de entrada
    ofstream canalS;//Declaramos un fichero lógico de salida
    cout<<"Introduce el nombre del primer fichero a combinar "<<endl;
    cin>> nombreFicheroA;
    cout<<"Introduce el nombre del segundo fichero a combinar "<<endl;
    cin>> nombreFicheroB;
    cout<<"Introduce el nombre del fichero de salida "<<endl;
    cin>> nombreFicheroS;
    canalEA.open(nombreFicheroA,ios::in);
    canalEB.open(nombreFicheroB,ios::in);
    canalS.open(nombreFicheroS,ios::out);
    if(canalEA && canalEB && canalS)
    {
        canalEA>>datoA;
        canalEB>>datoB;
        while(!canalEA.eof() && !canalEB.eof())
        {
            canalS<<datoA<< endl;
            canalS<<datoB<< endl;
            canalEA>>datoA;
            canalEB>>datoB;
        }
        if(!canalEA.eof())
        {
            canalS<<datoA<< endl;
            copiarResto(canalEA, canalS);
        }
        else if(!canalEB.eof())
        {
            canalS<<datoB<< endl;
            copiarResto(canalEB, canalS);
        }
        canalEA.close();
```

```
        canalEB.close();
        canalS.close();

    }
    else
    {
        cout<<"Problemas a la hora de abrir alguno de los ficheros"<<endl;
    }
    system("Pause");
    return 0;
}

void copiarResto(ifstream & canalE, ofstream & canalS)
{
    int dato;
    canalE>>dato;
    while(!canalE.eof())
    {
        canalS<<dato<< endl;
        canalE>>dato;
    }
}
```