



GRADO EN INGENIERÍA INFORMÁTICA

ESTRUCTURAS DE DATOS I

Práctica 1

Ficheros y Tablas Dinámicas

Esta primera práctica tiene por objetivo el afianzar los conceptos relativos al uso de **ficheros** y **memoria dinámica**.

La Dirección General de Tráfico (DGT) nos ha solicitado el desarrollo de una aplicación para generación y gestión de sanciones administrativas por exceso de velocidad, a partir de la información registrada por un sistema de radares distribuidos a lo largo de la geografía nacional.

Según la información reportada por la DGT, cada radar consta de dos puntos de captación; uno de entrada y otro de salida. Una vez se registra el paso de un vehículo por ambos puntos, a sabiendas de la distancia que los separa, basta con calcular el tiempo transcurrido entre ambas captaciones para determinar la velocidad media con la que éste ha circulado por dicho tramo. Este dato permite finalmente dictaminar si se ha cometido o no una infracción por exceso de velocidad.

El sistema de información en que se basará dicha aplicación parte de un fichero binario que contiene información sobre cada uno de los radares considerados en el programa. Este fichero se estructura como un conjunto indeterminado de elementos efectivos, cada uno de los cuales es aproximable como una instancia del tipo registro (*struct*) *radartramo*, cuya definición es la que sigue:

```
struct radartramo {
    int codigo; //código identificador único del radar.
    cadena nombre; //nombre del radar.
    cadena provincia; //provincia en la que se ubica el radar.
    cadena localizacion; //localización exacta del radar.
    float distancia; //distancia (en km) que separa los puntos de captura.
    int velocidadMediaMaxima; //velocidad máxima permitida en el tramo.
    cadena ficheropunto1; //fichero de lecturas de coches en el punto 1.
    cadena ficheropunto2; //fichero de lectura de coches en el punto 2.
};
```

donde el tipo ad-hoc *cadena* se utiliza para almacenar cadenas de caracteres, y se declara como sigue:

```
typedef char cadena[50];
```

En la siguiente figura, se muestra de forma ilustrativa la manera en la que está estructurado el fichero de radares.

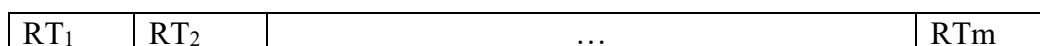


Fig 1. Estructura del fichero binario contenedor de los radares considerados en el sistema.

donde RT_i es la instancia del tipo registro *radartramo* correspondiente al radar i -ésimo almacenado en el sistema. Los campos *ficheropunto1* y *ficheropunto2* de un radar se refieren al nombre de los ficheros binarios que almacenan, respectivamente, los registros de los coches que pasan por el punto de entrada, y los de aquellos que hacen lo propio por el segundo punto de salida.

Cuando un vehículo pasa por alguno de estos puntos de control, los dispositivos de adquisición de datos instalados por la DGT realizan la captura de información. Tras procesar dicha captura genera una lectura que es almacenada en el *ficheropunto* correspondiente. En el sistema de información a desarrollar, cada lectura se articula como una instancia del tipo registro *lecturavehiculo*, cuya declaración es la siguiente:

```
struct lecturavehiculo {
    cadena matricula; //matrícula del vehículo.
    tlectura lec; //fecha y hora a la que se ha generado la lectura.
};
```

La definición tipo registro *tlectura*, utilizado para almacenar la fecha y hora a la que se registra el paso de un vehículo por alguno de los puntos de un radar, es la que sigue:

```
struct tfecha{
    int dia;
    int mes;
    int anno;};

struct thora{
    int hora;
    int min;
    int seg;
};

struct tlectura {
    tfecha fecha;
    thora hora;
};
```

Como puede apreciarse, el tipo *tlectura* consta de dos campos: uno de tipo *tfecha* para almacenar la fecha de captura (día, mes y año), y otro de tipo *thora* para almacenar la hora (hora, minuto y segundo).

Con esto, los ficheros binarios de capturas de cualquiera de los radares se organizan como un conjunto indeterminado de elementos, cada uno de los cuales es aproximable como una instancia del tipo registro *tlectura*. Estos elementos se organizan dentro de cada fichero ordenados de forma ascendente, según la fecha de captura (campo *lec*).

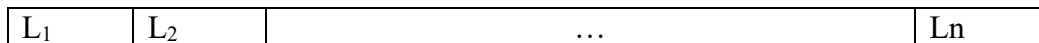


Fig 2. Estructura un fichero binario contenedor de las lecturas realizada por un radar en alguno de sus dos puntos, dónde cada elemento L_i se corresponde con una instancia del tipo registro *tlectura*.

En el sistema de información a implementar, también es necesario tener acceso al conjunto de vehículos registrados por la DGT. Para ello, este organismo ha puesto a nuestra disposición un fichero también binario que almacena para cada vehículo una instancia del tipo registro *coche*, cuya declaración es la propuesta a continuación:

```
struct coche {
    cadena matricula;//matrícula del vehículo.
    tfecha fecha; //fecha de caducidad de la última inspección realizada del vehículo.
    cadena marca;//marca del vehículo.
    cadena modelo;//modelo del vehículo.
};
```

Este fichero se estructura como un conjunto indeterminado de elementos efectivos, donde cada de ellos es aproximable como una instancia del tipo arriba definido. Cada vehículo se almacena en una posición específica dentro de dicho fichero, calculable a partir de la parte numérica de la matrícula del mismo. Así, sea un vehículo v , para el que la parte numérica de su matrícula se corresponde con el valor entero positivo m , se tiene que su posición dentro del fichero binario, p_v , se calcula como sigue:

$$p_v = m \% 1000$$

dónde el elemento almacenado justo al comienzo del fichero binario estaría alojado en la posición 0.

A modo de ejemplo, el vehículo con matrícula 1234ABC (ejemplo además del formato único de matrícula considerado en la presente práctica) almacenará su instancia registro correspondiente en la posición 234 del fichero, siendo 1234 el valor entero correspondiente a la parte numérica de dicha matrícula.

Por otra parte, las sanciones generadas a partir del procesamiento de los datos registrados por los distintos radares son también almacenadas en un fichero binario, con la siguiente estructura:

S_1	S_2	...	S_p
-------	-------	-----	-------

Fig 3. Estructura del fichero binario contenedor de las sanciones generadas por el sistema, dónde S_i es la sanción generada para un determinado vehículo para el que ha registrado una infracción por exceso de velocidad.

Cada elemento efectivo contenido en dicho fichero es aproximable como una instancia del tipo registro *sanciones*. A continuación, se presenta la declaración de dicho tipo:

```
struct sanciones {
    cadena matricula;//matrícula del vehículo sancionado.
    float euros;//cuantía en euros de la multa emitida por la infracción.
    int puntos;//número de puntos de carné sancionados (si procede).
    int codRadar;//código identificador del radar que registró la infracción.
    tlectura fh;//fecha/hora de la infracción (fecha/hora del paso del vehículo por el
                //punto 2 del radar).
};
```

La cuantificación de las sanciones es actualizada año a año. Así, para cada año se establecen tres franjas de sanción, que dependen del porcentaje con el que la velocidad máxima es rebasada:

- Sanción leve. Cuando la velocidad máxima permitida es superada en, como máximo, un 10%.
- Sanción estándar. Cuando la velocidad registrada supera a la velocidad máxima permitida en un exceso que oscila entre el 10 y el 20% de ésta.
- Sanción grave. Cuando la velocidad máxima permitida es superada en más de un 20%.

También se contempla la generación de sanciones para vehículos para los que se ha vencido la fecha límite para realizar su inspección técnica. Estas sanciones conllevan únicamente la resta de puntos de carné. El número de puntos sancionado también es actualizado cada año.

Con esto, la magnitud de las sanciones por año también es almacenada en un fichero binario. Este fichero se estructura como un conjunto indeterminado de elementos del tipo registro *tipossanciones*, que se define como sigue:

```
struct tipossanciones {
    int anno;//año de vigencia de las sanciones.
    float euros[3];
    int puntos[3];
    int puntositv;
};
```

Los elementos 0, 1 y 2 del campo-vector *eurosv* contienen la cuantía de la multa para, respectivamente, las sanciones leve, estándar y grave. Los elementos 0, 1 y 2 del campo-vector *puntosv* contienen el número de puntos sancionados para, respectivamente, las sanciones leve, estándar y grave. Finalmente, el campo *puntositv* contiene el número de puntos sancionables cuando se detecta que un vehículo ha caducado su fecha de inspección.

Cada registro de tipo *tipossanciones* se almacena en una posición concreta dentro del fichero, que viene determinada por el año al que corresponde. Así, sea el *tipo de sanción* t , correspondiente al año a , se tiene que su posición dentro del fichero binario, p_t , se calcula como sigue:

$$p_t = a \% 2000$$

dónde el elemento almacenado justo al comienzo del fichero binario estaría alojado en la posición 0.

A modo de ejemplo, la estructura que concreta la cuantía de las sanciones correspondientes al año 2018 estaría almacenada en la posición 18 del fichero.

La lógica de negocio a implementar se fundamenta en el conjunto de ficheros binarios comentados. De acuerdo con los requisitos trasladados por la DGT, la aplicación mantendrá abierto en tiempo de ejecución un flujo a través del cual operar sobre el fichero de sanciones. Cuando sea necesario, se procederá a procesar las lecturas registradas por un determinado radar. Para cada vehículo registrado por el radar, será necesario comprobar si éste ha superado la velocidad máxima permitida en el tramo correspondiente. Igualmente, será necesario comprobar si ha sido alcanzada la fecha de inspección del vehículo. Para esto último, será necesario consultar el dato correspondiente en el fichero binario que contiene la información de los vehículos. Con esto, si en base a las lecturas registradas para un vehículo y/o la consulta de la fecha de inspección, se estima que éste ha incurrido en una infracción, será generada una nueva sanción, que deberá ser añadida al fichero de sanciones.

Clase *GestorSanciones*

Con objeto de abordar definitivamente el desarrollo de aplicación, se ha convenido durante una etapa inicial de diseño la implementación de la clase *GestorSanciones*, cuya declaración es la que sigue:

```
class GestorSanciones {
    fstream ficheroSanciones; //flujo de acceso al fichero de sanciones.
    cadena nomFicheroRadares; //nombre del fichero de radares.
    cadena nomFicheroVehiculos; //nombre del fichero de vehículos.
    cadena nomFicheroTipoSancion; //nombre del fichero de cuantías de sanción por año.
    bool ficheroSancionesActivo; //estado del flujo; true indica que el fichero de sanciones
                                //ha sido abierto con éxito en el flujo ficheroSanciones.

public:
    ~GestorSanciones();
    GestorSanciones(cadena nFSanciones, cadena nFRadares, cadena nFVehiculos, cadena
                    nFTipoSancion);
    bool getFicheroSancionesActivo();
    void getNomFicheroVehiculos(cadena nF);
    void getNomFicheroTipoSancion(cadena nF);
    void mostrarRadares();
    bool mostrarRadar(int c);
    bool mostrarLecturasRadar(int c);
    bool procesarRadar(int c);
    bool mostrarVehiculo(cadena m);
    bool anyadirVehiculo(coche v);
    bool mostrarTipoSancion(int a);
    void mostrarSanciones();
};
```

A continuación, se concreta la funcionalidad de cada uno de estos métodos, así como los requisitos de implementación que deben cumplir:

- ***~GestorSanciones()***; Método destructor de la clase. Deberá cerrar el flujo *ficheroSanciones*.
- ***GestorSanciones(cadena nFSanciones, cadena nFRadares, cadena nFVehiculos, cadena nFTipoSancion)***; Método constructor de la clase. Deberá abrir de forma adecuada el fichero de nombre *nFSanciones* en el flujo *ficheroSanciones*, atributo de la clase. Si tiene éxito en la apertura del fichero, inicializará el atributo *ficheroSancionesActivo* a *true*; *false* en caso contrario. Finalmente, inicializará los atributos *nomFicheroRadares*, *nomFicheroVehiculos* y *nomFicheroTipoSancion* con las cadenas de caracteres contenidas en, respectivamente, los parámetros *nFRadares*, *nFVehiculos* y *nFTipoSancion*.
- ***bool getFicheroSancionesActivo()***; Método que devuelve el valor del atributo *ficheroSancionesActivo*.
- ***void getNomFicheroVehiculos(cadena nF)***; Método que devuelve el valor del atributo *nomFicheroVehiculos* a través del parámetro de entrada/salida *nF*.

- ***void getNomFicheroTipoSancion(cadena nF)***; Método que devuelve el valor del atributo *nomFicheroTipoSanción* a través del parámetro de entrada/salida *nF*.
- ***void mostrarRadares()***; Método que mostrará por pantalla la información de cada uno de los radares del sistema, contenidos en el fichero cuyo nombre está almacenado en *nomFicheroRadares*, atributo de la clase. Si no hay ningún radar en el fichero, se notificará de dicha circunstancia mostrando por pantalla el mensaje que sigue: *“ERROR. No hay radares registrados en el sistema.”*.
- ***bool mostrarRadar(int c)***; Método que mostrará por pantalla la información del radar con código identificador único igual al indicado por el parámetro *c*, confirmando la operación con el retorno de un *true*. En caso contrario, esto es, cuando no encuentre un radar coincidente con el identificador indicado, el método no mostrará nada por pantalla, y finalizará su ejecución devolviendo *false*.
- ***bool mostrarLecturasRadar(int c)***; Método que mostrará por pantalla las lecturas registradas por el radar con código identificador único igual al indicado por el parámetro *c*, tanto en su punto de entrada como en su punto de salida. El método devolverá *true* para confirmar que la operación se ha realizado con éxito; *false* en caso contrario, bien por no encontrar el radar indicado, o bien por encontrarse con algún problema en la apertura de los ficheros que contienen las correspondientes lecturas.
- ***bool procesarRadar(int c)***; Método encargado de procesar las lecturas de un radar para generar las sanciones correspondientes a las potenciales infracciones registradas. Para ello, en primer lugar, el método buscará el radar cuyo identificador coincide con el valor del parámetro *c*. Si no encuentra ningún radar coincidente, el método finalizará devolviendo *false*. En caso de éxito, se intentará abrir en flujos de lectura los ficheros que contienen las capturas registradas en los puntos 1 (entrada) y 2 (salida) del radar. Si hay algún problema en el proceso de apertura, el método finalizará devolviendo *false*.

En caso de éxito en la apertura de ambos ficheros, el método continuará con el procedimiento. Primero volcará una a una las lecturas del fichero del punto 2 (salida) en un vector dinámico de tipo *lecturavehiculo*, respetando el orden con el que están almacenados en la estructura binaria. Este vector será inicialmente creado con un tamaño de 2 elementos, y deberá ser redimensionado cada vez que se agote el espacio disponible, incrementando su tamaño en 2. Finalizado el volcado, se procederá a recorrer el fichero 1 elemento a elemento. Para cada lectura de entrada, se buscará en el vector dinámico el registro de salida correspondiente, y se computará si ha habido infracción por exceso de velocidad.

En base a la ordenación de los ficheros de lecturas (ascendente según el campo lec, fecha de captura de la lectura), dada una lectura del fichero 1, el registro de salida correspondiente será el primer elemento que encuentre en el vector dinámico que coincida en la matrícula y tenga una fecha de captura posterior.

Además, será necesario consultar la fecha de inspección del vehículo considerado en cada iteración. Esta información es esperable que esté contenida en el fichero de vehículos, cuyo nombre contiene el atributo de la clase *nomFicheroVehiculos*. Si la fecha de inspección es anterior a la actual se asumirá que ha vencido, y por tanto el titular del vehículo deberá ser sancionado por dicho motivo. Si se estima que el vehículo ha incurrido en alguna infracción, será necesario comprobar la magnitud de la sanción aplicable, en base al año en el que se registró la captura por parte del radar. Para ello será necesario consultar el fichero correspondiente, cuyo nombre es esperable que esté contenido en *nomFicheroTipoSancion*, atributo de la clase. Accedida esta información, será utilizada para generar una nueva instancia del tipo registro *sanciones*, y escrita al final del fichero correspondiente (fichero de sanciones), accesible a través del flujo *ficheroSanciones*. Finalizado el recorrido y procesamiento de fichero del punto1(entrada), el método finalizará devolviendo *true*, confirmando el éxito de la operación.

- ***bool mostrarVehículo(cadena m)***; Método que mostrará por pantalla la información del vehículo con matrícula igual a la indicada por el parámetro *m*, contenida en el fichero de vehículos. Devolverá *true* para confirmar que la operación ha finalizado con éxito; *false* en caso contrario, cuando no encuentre

ningún vehículo con la matrícula indicada, o bien haya sido imposible acceder al fichero de vehículos para consultar los datos correspondientes.

- ***bool anyadirVehiculo(coche v)***; Método que añadirá el vehículo *v* en el fichero de vehículos. Esto solo será posible en caso de que no haya ya almacenado un vehículo en la posición que, debido a su matrícula, le corresponde. El método devolverá *true* para confirmar que la operación de actualización del fichero de vehículos ha finalizado de forma adecuada; *false* en caso contrario, bien porque la posición que corresponde al vehículo ya está ocupada, o bien porque ha ocurrido algún tipo de problema con el manejo del fichero de vehículos.
- ***bool mostrarTipoSancion(int a)***; Dado el fichero con las cuantías de las sanciones, este método mostrará la información correspondiente al elemento del mismo cuyo año (campo *anno*) coincide con el valor del parámetro *a*. Si no encuentra ningún registro coincidente, el método devolverá *false*; en caso contrario (éxito en la búsqueda), mostrará los datos de la instancia correspondiente, y finalizará devolviendo un *true*. Igualmente, si por algún motivo resulta imposible abrir el fichero con las cuantías de las sanciones, el método finalizará devolviendo *false*.
- ***void mostrarSanciones()***; Este método mostrará todas las sanciones registradas en el sistema, contenidas en el fichero de sanciones, y accesible a través del flujo *ficheroSanciones*, atributo de la clase.

Programa principal. Experiencia de usuario.

La DGT también nos ha trasladado los requisitos y funcionalidades esperables del programa a implementar, junto con una captura a modo de prototipo, que ilustra el aspecto pretendido para la aplicación final.

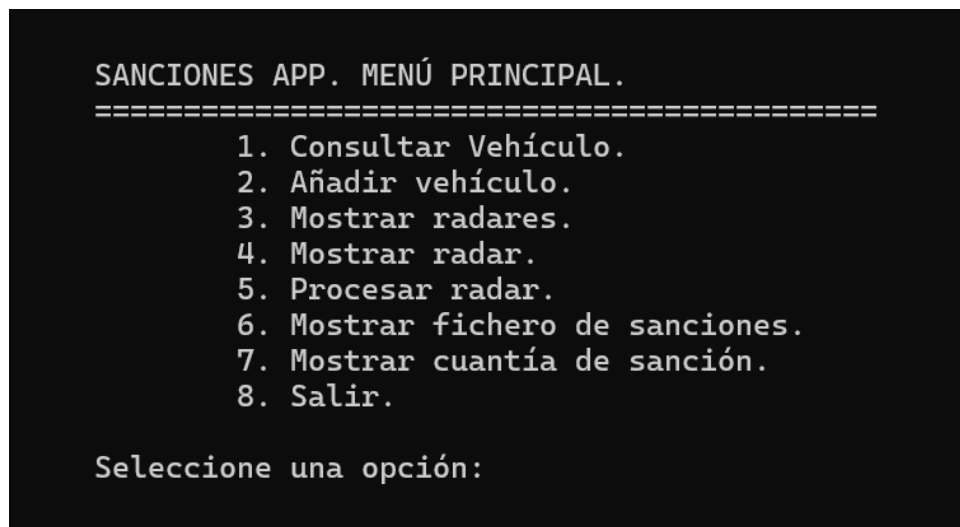


Fig. 4. Imagen prototipo del aspecto del menú principal de la aplicación a desarrollar.

Como detalle fundamental de implementación del programa principal, hay que indicar que toda la lógica de negocio relacionada con la gestión de las sanciones estará gobernada por un único objeto de la clase *GestorSanciones*. Al crear dicho objeto, será necesario pasarle al método constructor correspondiente los nombres de los ficheros de sanciones, de radares, de vehículos y de cuantías de sanción. Está permitido especificar el nombre de dichos ficheros directamente en código.

La aplicación, tras arrancar y generar el objeto de la clase *GestorSanciones*, mostrará de forma inmediata por pantalla un menú similar al mostrado en la figura 4. Tras esto quedará a la espera de que el usuario seleccione alguna de las opciones propuestas. Para elegir una opción, el usuario escribirá el número correspondiente y pulsará la tecla *intro*. De forma automática se desencadenará la ejecución de las acciones correspondientes

asociadas a la opción seleccionada. Finalizadas éstas, se volverá a renderizar en pantalla el menú principal, quedando el programa nuevamente a la espera de que el usuario seleccione una opción. De introducir un valor sin opción asociada, el usuario será notificado con un mensaje de error por pantalla, tras el cual, volverá de nuevo a renderizarse el menú.

A continuación, se detalla el funcionamiento de cada una de las opciones disponibles en la aplicación:

1. **Consultar vehículo.** Una vez seleccionada esta opción, la aplicación pedirá al usuario que introduzca una matrícula válida, y seguidamente mostrará la información almacenada para el vehículo correspondiente, según los datos almacenados para el mismo en el fichero binario de vehículos. Si no hay ningún vehículo registrado en el sistema con la matrícula indicada, el usuario deberá ser notificado por pantalla con un mensaje de error.
 2. **Añadir vehículo.** Una vez seleccionada esta opción, la aplicación pedirá al usuario que introduzca los datos del nuevo vehículo a registrar; concretamente su matrícula, la fecha límite de realización de su próxima inspección (día, mes y año), la marca y el modelo. La aplicación notificará mediante un mensaje por pantalla el éxito de la operación, o bien mostrará un mensaje de error en caso de que ésta no se haya podido completar, bien por existir ya un vehículo ocupando la posición que le correspondería al nuevo en el fichero de vehículos, o bien por algún problema derivado de la manipulación del propio fichero.
 3. **Mostrar radares.** A través de esta opción, se mostrará por pantalla la información correspondiente a todos los radares registrados en el sistema. Para cada radar, sólo se mostrarán los nombres de los ficheros que almacenan las lecturas de los pasos de vehículos, pero no el contenido de los mismos.
 4. **Mostrar radar.** Una vez seleccionada esta opción, la aplicación pedirá al usuario que introduzca el código identificador único del radar para el que quiere realizar la consulta. De existir dicho radar, la aplicación mostrará la información completa del mismo, incluyendo los registros de sus ficheros de lecturas. Si no se encuentra ningún radar con el identificador indicado por el usuario, éste será notificado con un mensaje de error.
 5. **Procesar radar.** Una vez seleccionada esta opción, la aplicación pedirá al usuario que introduzca el código identificador único del radar que pretende procesar. De no existir radar alguno con el identificador indicado por el usuario, el programa mostrará por pantalla un mensaje de error. En caso contrario, se procederá al proceso de los ficheros de lecturas del radar especificado, de acuerdo con la lógica comentada para el método *bool GestorSanciones::procesarRadar(int c)*, actualizándose si procede el fichero de sanciones.
 6. **Mostrar fichero de sanciones.** Una vez seleccionada esta opción, se mostrará por pantalla todas las sanciones recogidas en el fichero de sanciones.
 7. **Mostrar cuantía de sanción.** Una vez seleccionada esta opción, la aplicación pedirá al usuario que introduzca el año para el que se desea consultar la magnitud de las sanciones por exceso de velocidad y por caducidad de la fecha de inspección. En base a dicho año, se consultará el registro correspondiente en el fichero de cuantía de sanciones, y se mostrará los datos del mismo; esto es, la cuantía económica y el número de puntos sancionados para las infracciones leve, estándar y grave, y los puntos sancionados por caducidad de la fecha de revisión. Si no hay almacenado registro alguno para el año indicado, será necesario mostrar por pantalla un mensaje de error, notificando dicha situación.
 8. **Salir.** Permite salir de la aplicación.
-

Realización de la práctica.

Se pide:

1. **Implementar la clase *GestorSanciones***, según la declaración propuesta en el enunciado. Esta implementación deberá realizarse respetando los principios de modularidad vistos en la práctica 0, proponiendo en este caso la declaración de la clase en un fichero de cabecera *GestorSanciones.h*, y el código de implementación de sus métodos un fichero fuente diferente, *GestorSanciones.cpp*. Está permitida la implementación de métodos auxiliares en la clase, siempre y cuando éstos sean privados.
 2. **Implementar el programa principal propuesto en el enunciado.**
-

Notas de implementación:

- En código, tras generar el objeto de la clase *GestorSanciones* en la función principal *main*, será necesario comprobar el valor del atributo *ficheroSancionesActivo* de dicho objeto. Si dicho valor es *false*, se tendrá la certeza de que ha habido algún tipo de problema al intentar abrir el fichero de sanciones especificado, y por tanto no tendrá sentido continuar con la ejecución de la aplicación. Por tanto, en ese caso, el programa deberá notificar la ocurrencia del problema de carga del fichero con un mensaje de error, y finalizar seguidamente su ejecución.
- Está permitido el uso de las bibliotecas *fstream*, *iostream*, *cstring*, *ctime* y *stdlib*. Para utilizar cualquier otra biblioteca será necesario consultarlo previamente con el profesor.
- Está permitido añadir nuevos métodos auxiliares a la clase *GestorSanciones*, siempre que éstos sean declarados miembros privados.
- Está permitida la implementación de funciones auxiliares.