Problemática del uso de Ficheros en C++: un Enfoque Educativo

José Baltasar García Perez-Schofield
Manuel Pérez Cota
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Vigo
{jbgarcia; mpcota}@uvigo.es {URL: http://www.lsi.uvigo.es}

Resumen

En este artículo abordamos la problemática de la enseñanza del uso de ficheros en un lenguaje Orientado a Objetos como el C++ directamente relacionado con este paradigma de programación. Como podemos encontrar en la bibliografía, existen múltiples sistemas de acceso a ficheros (casi uno por lenguaje), y casi ninguno se encuentra integrado de una forma que podamos decir coherente. Aquí intentamos, pues, dar una solución al problema docente que implica dar una perspectiva real al alumno de lo que sucede en el manejo de los ficheros dentro del lenguaje de programación C++. Veremos como afecta el cambio de plataforma a los programas realizados según el estándar (supuestamente portable), y, finalmente, una solución que hace más transparente su entendimiento.

Introducción.

En el lenguaje C++, muchas veces encontramos sutiles diferencias entre distintas implementaciones del estándar ANSI; estas diferencias hacen que programas escritos mediante herramientas ANSI, en principio, portables a nivel de código fuente (compilables en cualquier plataforma) no lo sean en realidad. Este problema se trata en la primera parte del artículo, mientras que en las siguientes secciones se desarrollarán unas clases de ejemplo de tratamiento de ficheros, diseñadas para sustituir a las fstream, las clases que acompañan a la distribución ANSI de C++ [3][4], para manejo de ficheros. El objetivo de este artículo es, finalmente, mostrar unas clases que proporcionen un acceso coherente según la OOP a ficheros, y que, a la vez, sean portables según el estándar ANSI.

El estándar C++

Existen ciertos aspectos del lenguaje estándar[6] C++ que no están completamente definidos. Es de notar, por ejemplo, el uso de lo que en UNIX son llamadas al sistema como open(), read(), write() en la gestión de archivos; éstas "funciones" se mantienen en C++ (al ser un superconjunto de C). En principio, no son estándar, si bien existen en todos los compiladores (podríamos decir que son estándar *de facto*), puesto que en otro caso, una gran mayoría de programas escritos en C (e incluso en C++), no compilarían en absoluto, al utilizar estas llamadas. Las funciones que se pensaron que sustituirían a las llamadas al sistema fueron fopen(), fread(), fwrite(), fclose()... etc. En C++, se crearon las clases streams para recubrir estas llamadas [1].

Al no formar estas llamadas parte del estándar, se dan casos curiosos, que eliminan la portabilidad, a nivel de código fuente, del C++, como el siguiente ejemplo:

las cabeceras que es necesario incluir en un programa C++ para UNIX para acceder a las llamadas al sistema de ficheros son:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

Mientras que en Borland C++, un compilador muy difundido en plataformas PC, sólo se incluye la cabecera:

```
#include <unistd.h>
```

Este tipo de detalles, hacen que la portabilidad del lenguaje se ponga en entredicho muchas veces (recordemos que una de las razones que han popularizado al lenguaje JAVA ha sido la portabilidad que muestra, incluso en este caso, a nivel de código ejecutable (interpretado)).

Por otra parte, los streams de C++, aunque a primera vista parecen completos, en realidad no lo son (en [1] se explica con mucho más detalle este punto), al no cubrir estos *streams* la gran mayoría de las características deseables en un lenguaje OO. Pese a que las *streams* proporcionan un acceso muy bueno a archivos en modo texto (consola incluida), la parte de archivos binarios queda completamente huérfana (cabe recordar aquí que los operadores de extracción >> y << traducen sus argumentos a formato texto), y se reduce a dos métodos (dependiendo el tipo de *stream* que se esté utilizando), como son por ejemplo, en lectura y escritura(*fstream*), fstream.read() y fstream.write(), con el siguiente formato:

```
read(void *,unsigned long);
write(void *,unsigned long);
```

Es decir, exactamente igual a fread() y fwrite(), en el acceso básico a ficheros de C (y sus correspondientes read() y write(), en llamadas al sistema UNIX). La conclusión es que en cuanto al acceso a ficheros binarios (es decir, acceso a datos organizado, un tema muy importante puesto que de él derivará cualquier esquema realizado en C++ sobre Bases de Datos), es como si estuviésemos programando en C, sin aprovechar las características del paradigma OO.

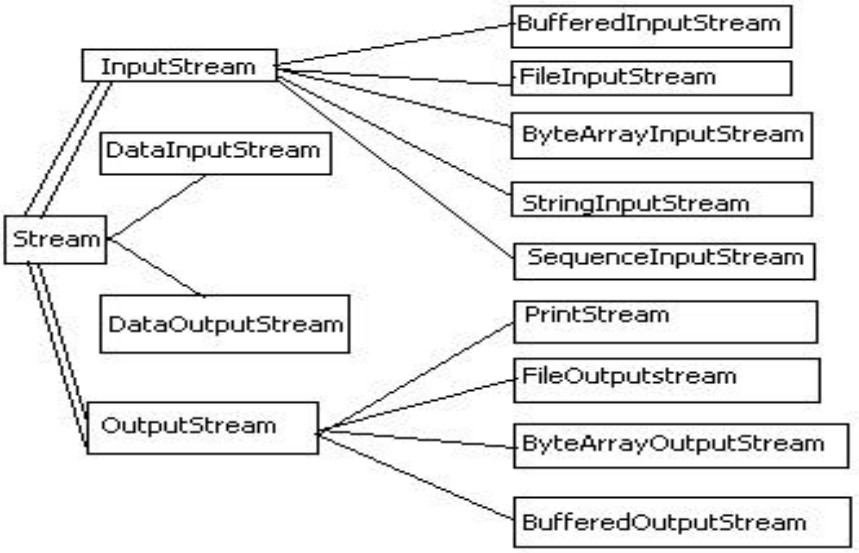

Fig. 1 Clases de Entrada/Salida en Java

Consideraciones sobre el sistema de ficheros

El sistema de ficheros ideal debería ser capaz de almacenar y recuperar **objetos**, tener capacidad de gestionar las diferentes capacidades de la OOP, (*polimorfismo, encapsulación,* …) y ser **portable**. En este artículo, propondremos una solución con la que intentaremos aproximarnos a estas premisas, inspirándonos en varios conceptos como son las clases de acceso a ficheros de Java, o la programación incremental[2].

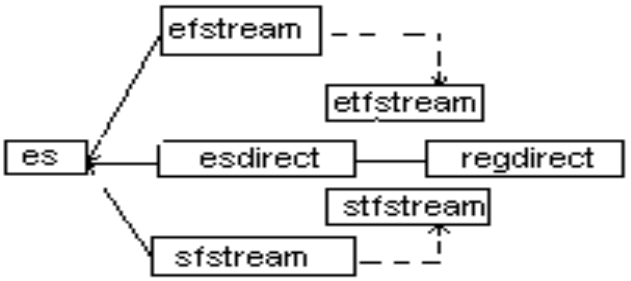

Fig. 2. Clases estream para C++

Así, como puede verse en la figura 1, en esta misma página, escogeremos la disposición de JAVA [5]: tres clases principales; lectura, escritura, y acceso directo (sólo tiene sentido hacer lectura y escritura en un acceso directo). Las clases siempre rabarán en formato byte, en vez del
doble formato de las stream, a elegir entre texto y bytes.

La razón de hacer esto así no es otra que la de pensar que la disposición de clases de C++ [3] [4], no es la correcta [1], por ser excesivamente compleja. Así, mientras en C++ se utiliza la herencia múltiple, para crear la mayoría de estas clases, en este sistema no se utilizará este mecanismo de herencia, y además no será necesario

escoger el comportamiento de la clase mediante argumentos como en C++, puesto que cada clase tendrá un comportamiento perfectamente definido; sólo será necesario instanciar la clase adecuada. Recordemos que a la clase de e/s a fichero fstream es necesario pasarle por parámetros si se desea lectura y/o escritura (ios::in | ios::out), mientras en Java, y en este sistema, cada clase cumple una única función. En la figura 2, puede verse el esquema resultante obtenido.

Así, disponemos de las tres clases principales mencionadas (efstream, entrada; sfstream, salida; esdirect, e/s), y de clases que proporcionan una funcionalidad suplementaria, como son etfstream y stfstream (texto); también regdirect, sobre archivos directos.

El funcionamiento es el siguiente: si el usuario desea leer de un fichero de entrada binario, instancia la clase *efstream*; si es de salida *sfstream*. Si se desea leer/escribir texto, (flotantes, enteros... en modo texto), se añade otra clase que funciona como "filtro" de la anterior, convirtiendo los datos a texto antes de pasarlos a escritura, o convirtiéndolos adecuadamente después de ser leídos (en la figura se ve con línea punteada la asociación de tipo agregación).

Si se desea acceso directo, se instancia la clase *esdirect*, que permite la e/s y movimiento por el fichero; *regdirect*, permite la gestión de archivos sencillos de registros de tamaño fijo (posiblemente, objetos).

Todas estas clases derivan de *es*, la clase madre que soporta acceso a ficheros. Gracias a la metáfora de ficheros, es posible controlar la pantalla y el teclado, incluso con sólo esta base.

En las sucesivas secciones, veremos como se han creado las clases, y ejemplos, de un uso contrastado con los streams de C++.

Descripción de las clases

A continuación, se explican las características de las clases aquí presentadas, que se espera que sean una aproximación a la resolución de los problemas indicados.
La clase base.

La clase base de la que derivan todas las demás proporciona de por sí un acceso básico a ficheros binarios, similar a la capacidad binaria de *fstream*, que, como ya dijimos, es, de por sí, incompleta. Por tanto, no se espera que la clase *es* sea usada (*instanciada*), en un programa, ya que sólo ofrece la encapsulación del número de *handle* del fichero. Es de resaltar, que, para ofrecer la mayor portabilidad posible, ésta clase, sobre la que se asientan todas las demás, debe estar construida utilizando, de la forma más completa posible, componentes estándares del lenguaje. La clase se ha implementado utilizando las instrucciones del lenguaje, por tanto, fopen(), fread(), fwrite(), y fclose(). La compatibilidad se garantiza al formar parte del estándar ANSI de C, que está perfectamente definido y asentado. Por definición, del C++ como superconjunto de C, sabemos que estas funciones estarán presentes en cualquier estándar de C++[6].

Las clases de Entrada y Salida básicas

Las clases *efstream* y *sfstream*, proporcionan, respectivamente, funcionalidad completa en cuanto a lectura y escritura. Son la base del resto de las clases, y sólo proporcionan escritura y lectura en modo binario: la única forma de escribir en un fichero es en modo binario. Veremos cómo se resuelve el "problema" de escribir texto (en un modo muy parecido a cómo lo hace JAVA), y que en las *streams* de C++ está presente como una de las dos formas de escribir en modo básico en un fichero (binario y texto).

Otra de las características de estas clases, es que a pesar de ofrecer escritura y lectura, por separado, sólo en modo binario, sí ofrecen, en cambio, una funcionalidad completa en cuanto a los operadores de inserción y extracción (<< y >>), pudiendo utilizarlos para casi cualquier tipo de datos, sin necesidad de utilizar el método correspondiente al tipo. Esto proporciona un grado de abstracción ya conocido, en C++, con relación a los ficheros de texto, y que por otra parte, forma parte de la filosofía de la Orientación a Objetos: tratar con un objeto sin saber en realidad cómo ha sido construido (**encapsulación**), de forma transparente al resto de los objetos.

Las clases de tratamiento de texto.

Las clases *etfstream* y *stfstream*, que en el diagrama de clases presentado se ven separadas de sus correspondientes en modo binario, son en realidad filtros o convertidores que se anteponen o aplican a una clase de entrada o salida binaria, respectivamente, para leer o escribir texto.

Al igual que en Java, estas clases aceptan, en sus constructores, un objeto de tipo *efstream* o *sfstream* (recordemos que mediante **polimorfismo,** cualquier objeto evolucionado a partir de éstas clases es válido), y la escritura/lectura en modo texto se realiza convirtiendo a priori/a posteriori los datos en binario/texto. De esta forma, el modo de escritura en el fichero sigue siendo único (binario), tal y como es en la realidad.

Estas clases, además, al igual que las anteriores, en modo binario, también soportan de modo completo los operadores de extracción e inserción. El diseño de esta colección de clases hace posible que estos operadores puedan aplicarse a los dos tipos de datos.

Las clases de acceso en lectura y escritura

Las clases *esdirect* y, *regdirect*, derivada directamente de ella, proporcionan el acceso de lectura/escritura en estas clases de acceso a ficheros.

Al igual que en el diseño de Java, sólo se permite el acceso de lectura y escritura en modo binario: en C++ también, ningún uso de un fichero de texto incluye las operaciones *fseek() y ftell()*. Nunca va a ser necesario leer y escribir a la vez en un fichero de texto. Por lo tanto, cuando el fichero sea de acceso de lectura y escritura, éste será única y exclusivamente binario, seguramente con el propósito de utilizarlo de

alguna forma organizada, aunque, por supuesto, para hablar de una base de datos, sería necesaria la agregación de más capas de software.

La clase *regdirect* añade un nivel más de abstracción, grabando siempre lo que a un nivel físico serían bloques de bytes de tamaño fijo, y que a este nivel son objetos, que hacen el papel del registro clásico.

Consideraciones sobre estas clases de recubrimiento de acceso a ficheros.

Un aspecto importante, que no presentan las clases *stream* de **C++,** ni las clases de Java, es el manejo de objetos desde un punto de vista de homogeneidad respecto al resto de los tipos de datos.

Aunque en principio debiera ser sencillo poder grabar el estado (es decir, lo que en C++ se correspondería con la instancia) de un objeto, no lo es tanto, cuando nos enfrentamos al problema de que C++ no ofrece ningún mecanismo estándar para obtener ese tipo de información de un objeto. Es decir, no disponemos de la capacidad de metainformación[7] acerca de un objeto (nombre de la clase, tamaño… etc.). Por supuesto, esto puede implementarse, aunque conlleva ciertos problemas que se comentarán más adelante. Aquí se ha resuelto de la siguiente forma: se ha creado la clase *Tobject*, de la que deben heredar todas las clases del sistema, o cuando menos, aquellas que serán susceptibles de ser tratadas en los ficheros. Aunque en principio esto pueda parecer una restricción importante, casi todos los sistemas tienen una clase de referencia de este tipo, con métodos como éste, que devuelven la longitud de la clase, y/o métodos de copia, que devuelven otro objeto exactamente igual al destinatario del mensaje (de utilidad en el caso de que existan componentes dinámicos dentro del objeto). Por otra parte, el uso de esta clase padre del resto de las clases puede ocasionar problemas a la hora de utilizar herencia múltiple; una posible solución es (al margen de no utilizar herencia múltiple) limitar este padre tan sólo a aquellas clases susceptibles de que sus instancias sean almacenadas en ficheros, aunque esto, por supuesto no resuelve el problema en su totalidad. Otra posible solución es eliminar la clase Tobject, aunque esto eliminaría la posibilidad de calcular la longitud de un objeto que utilice memoria dinámica.

Por supuesto, siempre es posible sobrecargar los operadores de extracción para cada clase y eliminar totalmente el uso de Tobject, pero para ésto no es necesario el uso de las clases aquí presentadas.

El método *dev_len()*, derivado de *Tobject*, obligatorio para conocer la longitud del objeto, puede ser tan sencillo como 'return sizeof(T…)' o tan complejo como un cálculo de las longitudes de los componentes dinámicos que contenga el objeto.

Una vez remarcada esta salvedad, estas clases son capaces de leer / escribir objetos directamente, y gracias a las características de la programación orientada a objetos, estas capacidades pueden sobrecargarse fácilmente para realizar grabaciones especiales, formatos, o cualquier otro tipo de modificación respecto a la típica *'write(&a,1,sizeof(….'*. Se ha desechado en principio la idea de incluir en la clase *Tobject* sendos métodos virtuales de lectura/escritura en fichero, los cuáles sí podrían implicar cambios más importantes en una estructura de clases ya creada.

Consideraciones acerca del punto de vista educacional

Desde un punto de vista educativo, los autores pensamos que estas clases se acercan más al ideal de Orientación a Objetos que las que ofrece el "estándar" de C++ (o su borrador). Por supuesto, no se trata de condenar o criticar un lenguaje, sino de intentar aportar una solución o una aproximación, pensamos que, más adecuada.

Por tanto, al nivel de la explicación docente, las clases de e/s forman un todo más armónico con el universo objetual que las rodea. Quizá, al nivel del alumno que comienza desde cero con un lenguaje orientado a objetos, las clases *stream* sean más sencillas de asimilar, puesto que, salvo en caso de ficheros de texto, tiene pocas características reales de orientación a objetos. Pensemos, que a la hora de almacenar un objeto en un fichero, el alumno utilizará los mismos mecanismos (write(&a,1,sizeof(T…)) que utilizaba en C. Aún así, las ventajas de tener un sistema de ficheros adecuado al paradigma de programación utilizado son más que evidentes. Las clases son, además, portables a otras plataformas debido a la preocupación señalada por el respeto hacia el estándar, aspecto que pensamos que sería deseable inculcar a los alumnos, de nuevo desde el punto de vista de un aprendizaje adecuado al paradigma que intenta explicárseles; como ya se ha dicho, a la hora de implementar ciertos aspectos estándar *de facto*, podemos encontrar esas pequeñas diferencias que den al traste con la portabilidad de un programa y que hacen más difícil la docencia y la asimilación de un concepto tan importante como es el sistema de ficheros de un lenguaje.

Consideraciones acerca de la persistencia de objetos

Por supuesto, las presentes clases descritas en el artículo, no pretenden por sí mismas resolver el problema de la persistencia de objetos. Esto es algo que se escapa del alcance de este artículo dada su complejidad; sin embargo, dada la referencia acerca del almacenamiento de objetos en ficheros binarios, es necesario aclarar ciertas cuestiones técnicas.

Así, las clases que aquí se tratan, han sido diseñadas para almacenar y recuperar lo que se entiende por instancias en un paradigma de orientación a objetos basado en clases (recordemos que también existen los basados en prototipos): es decir, lo que se almacena en el fichero es el estado del objeto, mientras en la clase se almacena la estructura del mismo. Esto, por sí mismo contiene la implicación de que la clase debe estar presente a la hora de almacenar el objeto, y, también, y este es el problema, a la hora de recuperarlo. Esto último puede no ser posible en todos los casos, haciendo inservible lo almacenado. Estas clases actúan obviando este problema, asumiendo que el programador tendrá disponible la definición de la clase en ambos momentos.

Conclusiones.

Ha intentado presentarse en este artículo un conjunto de clases reales que sustituyen a las clases *stream* de C++, proporcionando un acceso a ficheros adecuado "al medio", es decir, a un lenguaje de programación orientado a objetos como es el C++.

Estas clases intentan corregir las carencias, relativas a la orientación a objetos, que tiene la librería de C++ mencionada, y también proporcionar esta solución desde el estándar de C++.

Finalmente, se ha destacado la adecuación de este sistema de clases, a la enseñanza, resaltando las mejoras educativas que conlleva el estudio de un todo (en este caso, un lenguaje OO) homogéneo.

Por supuesto, la intención de este artículo no es criticar en ningún modo, a un lenguaje, C++, que ha probado ya su aceptación por parte de los programadores, ni muchísimo menos la labor de los autores del lenguaje, sino en todo caso, señalar una carencia y realizar una aportación de cara al mundo de la enseñanza del C++ y la programación Orientada a Objetos.

Apéndice

A continuación, un extracto de las fuentes de las clases de entrada / salida. Las fuentes completas, pueden ser pedidas a los autores, con ejemplos sobre su uso.

La clase *Tobject* se utiliza como padre en el resto. Contiene *dev_len()*, para obtener la longitud de un objeto.

```
class Tobject {
        public:
                Tobject() {};
                virtual unsigned long dev_len(void)=0;
};
```

La clase *es* es la clase base de la que derivan las clases específicas de acceso a ficheros. No se espera que esta clase sea instanciada.

```
class es : public Tobject {
protected:
  FILE * arch;
  char modo[5];
  void attach_file(FILE *x)  { arch = x; };
  void attach_file(char *x)  { arch = fopen(x,modo); };

public:
  es() { arch = NULL;     setmode(esEXIST);};
  ~es(){ if (arch!=NULL) { fclose(arch); } };
  unsigned long read(unsigned char *,unsigned long);
  unsigned long write(unsigned char *,unsigned long);
  unsigned long moveto(unsigned long);
};
```

La clase siguiente proporciona la salida binaria básica a fichero, y también la sobrecarga del operador <<, para cada tipo, incluido Tobject.

```
class sfstream : public es {
public:
```

```
  sfstream()                {};
int write(char x);
int write(int x);
virtual int write(unsigned char *x,unsigned long len);
int write(float x);
virtual int write(Tobject &);
sfstream &operator <<(Tobject &x)
                {
                        this->write(x);
                        return *this;
                };
};
```

De la misma forma, se define *efstream*, para entrada binaria. Se muestra como ejemplo una de las sobrecargas de >> para la clase.

```
class efstream : public es {
int read(char &x);
int read(int &x);
virtual int read(unsigned char *x,unsigned long len);
int read(float &x);
virtual int read(Tobject &x);
efstream &operator >>(Tobject &x)
                {
                        this->read(x);
                        return *this;
                };
```

La clase *stfstream* actúa como "filtro" de peticiones de acceso a ficheros binarios, transformando los datos de/a texto. Los constructores, que no se muestran, aceptan un objeto *sfstream* como salida de datos, o directamente, el nombre de un fichero. En modo texto, no se considera la entrada ni salida de objetos.

```
class stfstream: public Tobject {
protected:
  int vvalid;
  class sfstream *sal;
public:
  virtual int write(char x);
  virtual int write(int x);
  virtual int write(unsigned char *x,unsigned long len);
  virtual int write(float x);
  stfstream &operator <<(char x)
                {
                        this->write(x);
                        return *this;
                };
};
```

La siguiente clase es la correspondiente a la anterior, pero en cuanto a entrada. El miembro *get*() devuelve el siguiente *token*, según el tipo de entrada especificada (real, char.. etc).

```
class etfstream: public Tobject {
protected:
  int vvalid;
```

```
  class efstream *ent;
  int get(char *rec,char *ctrl);
public:
  int valid(void) { return vvalid; };
virtual int read(char &x);
virtual int read(int &x);
virtual int read(unsigned char *x,unsigned long len);
virtual int read(float &x);
virtual int read(char *x);
etfstream &operator >>(char &x)
                {
                        this->read(x);
                        return *this;
                };
etfstream &operator >>(int &x)
                {
                        this->read(x);
                        return *this;
                };
unsigned long dev_len(void) { return sizeof(class etfstream); };
```

La clase *esdirect* , permite el acceso a ficheros binarios, de entrada y salida, y por lo tanto, según el diseño de las clases, de acceso obligatoriamente directo.

```
class esdirect : public es

{
public:
  esdirect()                {};
  esdirect(char *x);
  esdirect(FILE *x);
  int read(char &x);
  int read(int &x);
  virtual int read(unsigned char *x,unsigned long len);
  int read(float &x);
  int read(char *x);
  virtual int read(Tobject &);
  esdirect &operator >>(Tobject &x);
  esdirect &operator >>(char &x);
  int write(char x);
  int write(char *);
  int write(int x);
  virtual int write(unsigned char *x,unsigned long len);
  int write(float x);
  virtual int write(Tobject &);
  esdirect &operator <<(Tobject &x)
                {
                        this->write(x);
                        return *this;
                };
};
```

A continuación, se expone la implementación de algunos métodos, que puede facilitar la comprensión de las clases descritas.

Siguen los métodos leer y escribir genéricos utilizados en la clase *es*:

```
unsigned long es::read(unsigned char *x,unsigned long y)

{
        if (arch!=NULL)
                return fread((void *)x,y,1,arch);
        else return 0;
}

unsigned long es::write(unsigned char *x,unsigned long y)

{
        if (arch!=NULL)
                return fwrite((void *)x,y,1,arch);
        else return 0;
}
```

Los métodos de la clase *efstream*, que utilizan las llamadas de su case padre, la *es*:

```
int efstream::read(char &x)

{
        if (es::read((unsigned char *) &x,sizeof(char))==sizeof(char))
                return esTRUE;
        else
                return esFALSE;
}

int efstream::read(Tobject &x)

{
        if (es::read((unsigned char *) &x,x.dev_len())==x.dev_len())
                return esTRUE;
        else
                return esFALSE;
}
```

Asimismo, los de la clase *sfstream*:

```
int sfstream::write(Tobject &x)

{
        if (es::write((unsigned char*)&x,x.dev_len())==x.dev_len())
                return esTRUE;
        else
                return esFALSE;
}
int sfstream::write(int x)
{
        if (es::write((unsigned char*)&x,sizeof(int))==sizeof(int))
                return esTRUE;
```

```
        else
                return esFALSE;
}
```

Los métodos de la clase *stfstream* convierten los datos a texto antes de escribirlos.

```
int stfstream::write(char x)
{
        return this->write((unsigned char *)&x,1);
}
int stfstream::write(int x)

{
        char temp[esMAXTEMP];

        memset((void *)temp,0,esMAXTEMP*sizeof(char));
        sprintf(temp,"%d",x);

        return this->write((unsigned char *)temp,strlen(temp));
}
```

A continuación, se exponen los mismos métodos, de conversión de texto, pero en este caso, para permitir la entrada.

```
int etfstream::read(char &x)
{
        return this->read((unsigned char*) &x,1);
}
int etfstream::read(int &x)
{
        char temp[esMAXTEMP];

        int dev = get(temp,esCARSINT);

        x = (int) atoi(temp);

        return dev;
}
```

A continuación, algunos métodos de la clase de acceso directo a ficheros. *esdirect* sólo provee el acceso básico, siendo recomendable el derivar una clase desde ésta, para sólo escribir y leer objetos.

```
int esdirect::read(char &x)
{
        if (es::read((unsigned char *) &x,sizeof(char))==sizeof(char))
                return esTRUE;
        else
                return esFALSE;
}
int esdirect::read(Tobject &x)
{
        if (es::read((unsigned char *) &x,x.dev_len())==x.dev_len())
                return esTRUE;
        else
                return esFALSE;
}
```

```
int esdirect::read(int &x)
{
        if (es::read((unsigned char *) &x,sizeof(int))==sizeof(int))
                return esTRUE;
        else
                return esFALSE;
}
int esdirect::write(char x)
{
        if (es::write((unsigned char*)&x,sizeof(char))==sizeof(char))
                return esTRUE;
        else
                return esFALSE;
}
```

Bibliografía.

[1] García Perez-Schofield, José Baltasar; Pérez Cota, Manuel. *Problemática de la E/S en C++, comparación con Java.* Actas del Congreso "Jornadas sobre Objetos'98", Universidad de Deusto, Bilbao. Septiembre 1998.

[2] García Perez-Schofield, José Baltasar; Pérez Cota, Manuel. *Implantación Orientada a Objetos de un Gestor de Bases de Datos Multiplataforma.* Actas del Congreso "IV Jornadas de Informática", Universidad de Las Lagunas, Las palmas de Gran Canaria. Julio de 1998.

[3] Borland Intl. *Manual de Borland C++*, versión 5.00.

[4] GNU. *Manual de Gcc* versión 2.7.1.

[5] Naughton, Patrick. *Manual de Java.* Ed. McGraw-Hill, 1996

[6] Stroustrup, Bjarne. *A perspective on ISO C++*. The C++ Report, Vol. 17/8. Octubre de 1995. Http:://www.research.att.com/~bs/papers.htm

[7] Günther Blascheck, *Object Oriented Programming with Prototypes.* Editorial Springer-Verlay 1994. ISBN 3-540-56469-1.