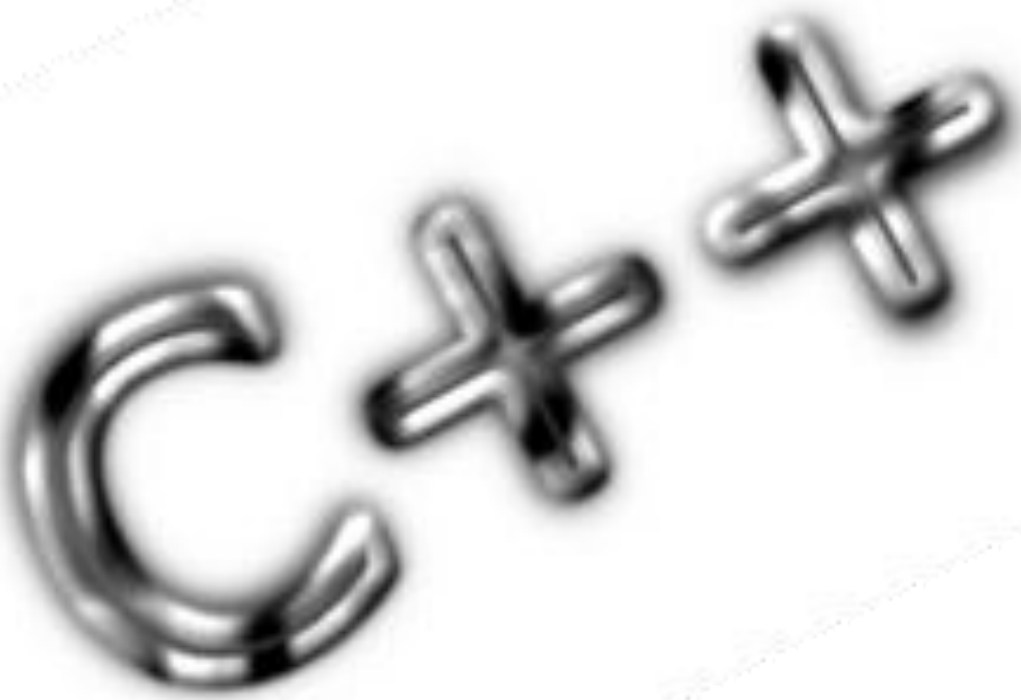

FICHEROS C++

TRANSVERSAL DE PROGRAMACIÓN BÁSICA

INGENIERÍA DE SISTEMAS

En el presente documente se hace una breve presentación del manejo de ficheros y archivos bajo el lenguaje de c++. Aunque existe más información referente al tema, solo se presentan los aspectos básicos.

CRISTIAN GUILLERMO GARCÍA
MONITORIA 2012-3 | Universidad Distrital francisco José de Caldas

FICHEROS C++

Tabla de contenido

FICHEROS C++

1. INTRODUCCIÓN

El presente documento pretende servir de guía para la enseñanza en la programación bajo el lenguaje C++. Pese a que se ha desarrollado siguiendo los lineamientos establecidos en el syllabus de la asignatura transversal de programación básica de la universidad distrital Francisco José de Caldas, es posible utilizarla como un manual en cualquier campo o ámbito siempre y cuando se relacione con el aprendizaje del lenguaje mencionado.

Es necesario resaltar que este documento es completamente tomado de [1] de quien no se precisa autor. Sin embargo, se han hecho algunas modificaciones en cuanto al formato de presentación.

2. ARCHIVOS EN C++

En los temas anteriores hemos visto que los datos de entrada o salida en un programa C++ son obtenidos o enviados directamente a través de lo que llamamos flujos de entrada o salida. Hasta hora hemos manejado los flujos estándares *cin* y *cout*, y una serie de operadores (<< y >>) y funciones (como *get* o *getline* por ejemplo) disponibles a partir de la biblioteca *iostream* y que son útiles para la entrada/salida de la información.

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

C++ soporta dos tipos de archivos: de texto y binarios. Los primeros almacenan datos como códigos ASCII. Los valores simples, tales como números y caracteres están separados por espacios. Los segundos almacenan bits de forma directa y se necesita usar la dirección de una posición de almacenamiento.

Una biblioteca en C++ que proporciona "funciones" y operadores para el manejo de ficheros es la biblioteca *fstream.* En general emplearemos ésta para la gestión básica de ficheros por lo que deberemos incluir el archivo de cabecera correspondiente en nuestros programas mediante la declaración adecuada, o sea, incluyendo la directiva de pre-procesamiento siguiente

#include <fstream>

La biblioteca *fstream* ofrece un conjunto de funciones y operadores comunes a todas las operaciones de Entrada/Salida (E/S) de ficheros.

2.1. Apertura de Ficheros

Antes de que un programa pueda manipular un fichero para leer o escribir información se debe abrir (o crear si es necesario) el fichero para identificar la posición del mismo en el programa (o sea, la dirección de memoria a partir de la cual almacenaremos o leeremos el contenido del fichero). Ya hemos indicado anteriormente que los archivos son tratados como flujos de entrada/salida por lo que, al igual que sucede con los flujos estándares y por defecto *cin* y *cout*, la información "debe" ser transferida en una sola dirección por lo que, salvo en ciertos casos, los ficheros deben abrirse bien para entrada o bien para salida.

2.1.1. Ficheros de Entrada o Salida

Supongamos que tenemos un fichero cuyo nombre en el sistema operativo es "*nombre.extensión".* Entonces hay varias formas de abrirlo.

(A) **Como fichero de entrada:** Para ello empleamos la siguiente declaración

ifstream descriptor ("nombre.extensión");

(B) **Como fichero de salida:** Para ello empleamos la sentencia

ofstream descriptor ("nombre.extensión");

En ambos casos *descriptor* es una variable que se asocia al fichero cuyo nombre es "*nombre.extensión".*

Comentario:

Observe que a partir de la apertura de un fichero, el descriptor del fichero será utilizado en todas las operaciones que se realicen sobre el fichero (OJO! no utilizamos el nombre original del fichero).

Otra forma de abrir un fichero para operar sobre él consiste en crear primero el flujo de entrada asociado al mismo, o sea, en asociar una variable *descriptor* a un fichero en una forma similar a como se declara una variable en C++, es decir, mediante una sentencia tal como

ifstream descriptor; *// Para ficheros de entrada*
ofstream descriptor; *// Para ficheros de salida*

y, posteriormente, abrir el fichero (en el modo deseado: de entrada o salida por ejemplo) mediante el uso de la función *open* aplicada a ese flujo de entrada, o sea en la forma

descriptor.open("nombre.extensión",int modo);

donde la variable *modo* indica el modo de apertura del fichero y los modos de apertura, posiblemente combinados mediante el símbolo '**|**' como veremos a continuación, pueden ser:

ios:: in // Modo entrada
ios:: out // Modo salida
ios:: app // Modo añadir, o sea, posicionar el cursor del fichero (ver abajo)
// al final del fichero antes de cada escritura
ios:: binary // El archivo se abre en modo binario
ios:: nocreate // Genera un error si el fichero no existe
ios:: noreplace // Genera un error si el fichero existe ya

Un archivo abierto con *ofstream* puede serlo en dos modos:

1. **Modo salida**, usando *ios::out*. En este caso todos los datos en el fichero se descartan (o sea, se borra el contenido del fichero si existe previamente). Naturalmente si el fichero no existe lo crea. En ambos casos el cursor del fichero (o sea, la cabeza de lectura/escritura del mismo) se posiciona al principio del fichero. Este es el modo por defecto de apertura de un fichero de salida.

2. **Modo añadir**, usando *ios::app*. En este caso los datos adicionales se añaden a partir del final del fichero (o sea, el cursor se sitúa al final del fichero y es ahí a partir de donde se empieza a realizar la escritura).

También se pueden combinar las dos formas anteriormente vistas de forma que el descriptor de fichero es declarado a la vez que el fichero es abierto. Para ello empleamos las siguientes declaraciones (según el fichero se abra para entrada o para salida):

ifstream descriptor("nombre.extensión",int modo); // para entrada
ofstream descriptor("nombre.extensión",int modo); // para salida

Donde *modo* es como se ha mostrado anteriormente.

EJEMPLO 1

Las dos líneas siguientes abren el fichero "mio.txt" como fichero de entrada (para lectura) y lo asocian al descriptor *in*.

```
ifstream in; // descriptor del fichero a abrir
in.open("mio.txt"); // Apertura del fichero;
```

Esas dos líneas equivalen a la siguiente:

```
ifstream in (“mio.txt”); // Apertura del fichero;
```

EJEMPLO 2

Para abrir el fichero “salida.dat” en modo salida (si el fichero no existe lo crea, y si existe borra su contenido) asociándolo al descriptor *out* podemos usar la siguiente sentencia;

```
ofstream out("salida.dat");
```

O la siguiente

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

O también

```
ofstream out;
out.open("salida.dat");
```

2.1.2. Ficheros de Entrada/Salida

Un fichero puede ser también abierto para entrada-salida. En este caso emplearemos una declaración *fstream* combinada con la operación de apertura correspondiente, o sea:

fstream descriptor;
descriptor.open(“nombrefichero.ext”, ios::in | ios::out)

O alternativamente podemos combinar la declaración y la apertura en una sola sentencia. Por

fstream descriptor(“nombre.extensión”,ios::in | ios:: out); // para entrada-salida

La declaración *fstream* puede ser usada arbitrariamente tanto sobre archivos de escritura como sobre archivos de lectura, aunque recomendamos que para archivos de solo-lectura se use *ifstream* y para aquellos de solo-escritura se use *ofstream*.

Comentario:

Cuando un fichero de E/S se declara con fstream, abrirlo con open debemos especificar el modo de apertura que queremos (entrada, salida o bien entrada.salida).

EJEMPLO 3

// Abrimos el fichero “F1.dat” como fichero de entrada

```
fstream inout;
inout.open("F1.dat", ios::in);
```

// Intentamos abrir el fichero “F1.dat” pero tenemos un error puesto que no
// hemos especificado el modo de apertura

```
fstream inout;
inout.open("F1.dat");
```

// Abrimos el fichero “F2.txt” como fichero de salida en modo AÑADIR.

```
fstream inout;
inout.open("F2.txt", ios::app);
```

// Abrimos el fichero “binario.dat” para entrada-salida binaria y le asociamos el
// descriptor ejemplo.

```
fstream ejemplo ("binario.dat", ios::in | ios:: out | ios::binary);
```

2.2. Cierre de Ficheros

Un fichero anteriormente abierto y con un descriptor asociado a él debe ser cerrado con el fin de liberar los recursos asociado a él de la siguiente forma:

descriptor.close()

2.3. Detección de fin de fichero y otras funciones

Además de las funciones *open* y *close*, existen otras funciones disponibles en la biblioteca *fstream*, que pueden ser aplicadas directamente al descriptor de un fichero en la forma

Descriptor.función();

Donde ***función*** es una de las siguientes:

- La función ***eof()*** que devuelve "true" si se ha alcanzado el final del fichero y falso en cualquier otro caso.
  LECTURA ADELANTADA: Para que la función **eof()** devuelva un valor de verdad (actualizado) es necesario, en muchos casos, realizar una operación de lectura previa. Esto permite que se actualice el valor a devolver por la función *eof* comprobándose si tras

realizar dicha operación se ha llegado al final del fichero. A este proceso lo llamaremos **lectura adelantada** (ver el ejemplo 6).

- La funciones ***fail()*** o ***bad()*** que devuelven "true" si existe un error en una operación de flujo asociada al fichero identificado por la variable *descriptor* (bien sea en una apertura, un cierre, una escritura o una lectura entre otras posibles operaciones) y "false" en caso contrario. La diferencia entre estas dos funciones es muy sutil. En realidad un flujo puede estar en un estado de fallo (i.e., *fail*) pero no en un estado erróneo (i.e. *bad*) en el caso de que el flujo no se considere corrupto ya que no se han perdido los caracteres que provienen del flujo (en un estado erróneo, esto puede no ocurrir). Recomendamos un vistazo a algún manual de C++ para aclarar las diferencias tan sutiles.

- La función ***good()*** que devuelve "true" si no existe un error en una operación de flujo y "false" en caso contrario.

2.4. Comprobación de Apertura Correcta

Antes de empezar a leer o escribir en un fichero es siempre conveniente verificar que la operación de apertura se realizó con éxito. La comprobación del "buen estado" de un determinado flujo asociado a un fichero se puede realizar preguntando directamente sobre el descriptor de fichero asociado (al igual que se hace con los flujos estándares *cin* y *cout*) en la forma siguiente:

```
if (descriptor){
      //Buen estado. Continuamos
      // operando  sobre el fichero
      .....
}
```

```
if (! descriptor){
      // Mal estado. Escribimos un mensaje
      // a la salida estándar
  cout << "Error en la apertura "
  .....
}
```

O mejor aún, utilizando las funciones *good* y *bad* (o *fail*) vistas anteriormente, o sea:

```
if (descriptor.good()){
      //Buen estado. Continuamos
      // operando  sobre el fichero
      .....
}
```

```
if  (descriptor.bad()){  // o descriptor.fail()
      // Mal estado. Escribimos un mensaje
      // a la salida estándar
  cout << "Error en la apertura "
}
```

.

EJEMPLO 4

```
ifstream in("F1.dat");
if (!in)          // O bien,  if (in.bad())
{
  cout << endl <<"Incapaz de crear o abrir el fichero " << endl;
                  // salida estándar
  cout << " para entrada " << endl;
}
else{
    …. // Se opera sobre el fichero
}
```

3. LECTURA-ESCRITURA EN FICHEROS (DE TEXTO)

3.1. Avance del Cursor

Tanto para los ficheros de texto como para los ficheros binarios existe lo que se denomina el *cursor* o *apuntador* del fichero que es un índice que direcciona una posición relativa del fichero. El cursor marca en todo momento la posición actual del fichero a partir de la cual va a tener lugar la siguiente operación de entrada-salida a ejecutar sobre el mismo (bien sea una operación de lectura o una de escritura). Cada vez que se realiza una operación de entrada o salida, el cursor del fichero avanza automáticamente el número de bytes leídos o escritos.

3.2. Ficheros de Texto

La lectura y la escritura en un archivo de texto se puede realizar directamente con los operadores **<<** y **>>** al igual que se realiza sobre los flujos estándares *cin* y *cout*.

EJEMPLO 5

El siguiente programa escribe tres líneas en un fichero llamado “EJEMPLO5.TXT” que se crea en el programa (si ya existe borramos su contenido). Cada línea consta de un entero, un real y una cadena de caracteres. Los datos en cada línea están separados por espacios en blanco.

```
#include <fstream>   // Biblioteca para el manejo de ficheros
#include <iostream>  // Biblioteca para la entrada-salida estándar

using namespace std;

int main(){
  ofstream fichout("EJEMPLO5.TXT",ios::out);
  if (!fichout){
    cout << endl << "Incapaz de crear este o abrir el fichero " << endl;
  }else {
    fichout << 1 << " " << 5.0 << " APROBADO" << endl;
                                          // Escritura en el fichero
    fichout << 2 << " " << 1.1 << " SUSPENSO" << endl;
    fichout << 3 << " " << 8.0 << " NOTABLE " << endl;
    fichout.close();
  }
  system("PAUSE");
  return 0;
}
```

EJEMPLO 6

El siguiente programa lee el fichero de texto "EJEMPLO5.TXT", creado en el ejemplo anterior, y visualiza su contenido en el monitor.

Comentario:

El operador >> omite los espacios en blanco al igual que ocurría en la entrada estándar.

```
#include <fstream>   // Libreria para el manejo de ficheros
#include <iostream>

using namespace std;

typedef char TCadena[30];

int main(){
  int i;
  float r;
  TCadena cad;

  ifstream fichin("EJEMPLO5.TXT");   // declaracion y apertura del fichero
  if (!fichin){
    cout << endl << "Incapaz de crear o abrir el fichero ";
  }
  else{
    fichin >> i;                     // Observe la lectura adelantada!!!
    while (!fichin.eof()){
      cout << i << " ";              // Lectura de valores en el fichero
      fichin >> r;
      cout << r << " ";              // Lectura de valores en el fichero
      fichin >> cad;
      cout << cad << endl;           // Lectura de valores en el fichero
      fichin >> i;
    }
    fichin.close();
  } // Fin del else
  system("PAUSE");
  return 0;
}  // Fin del main
```

Comentarios:

(1) Observe en el ejemplo anterior que ha sido necesario realizar una lectura adelantada previamente al chequeo del fin de fichero. Si no se realiza de esta forma podríamos tener problemas.

(2) Observe también que no es necesario realizar una lectura para "saltarse" los espacios en blanco que fueron introducidos en el fichero "EJEMPLO5.TXT" en el ejemplo 5. Esto es debido a que, como ya se comentó anteriormente, el operador **>>** omite los espacios en blanco (3) No olvide cerrar los ficheros!!

4. FICHEROS BINARIOS

Hasta ahora hemos trabajado con ficheros de texto, éstos pueden ser vistos como una serie de caracteres que pertenecen a un determinado código de entrada/salida, en nuestro caso al código ASCII. También hemos visto cómo se abre un fichero binario. En esta sección vamos a ver cómo podemos leer y escribir datos en binario.

4.1. Utilidad de los ficheros Binarios

En esta sección explicamos la diferencia de los ficheros binarios con los de texto y lo hacemos a partir de un ejemplo sencillo. Por ejemplo, si creamos un fichero con

```
ofstream fich("F1.dat",ios::out);
if (!fich) {
    cout << endl << "Incapaz de crear este o abrir el fichero ";
    cout << " para salida " << endl;
}
else{
    fich << "HOLA ADIOS";  // Escribe la cadena en el fichero
                           // "F1.dat"
    fich.close();
}
```

"F1.dat" será un fichero que contiene 10 bytes, donde cada uno será el código ASCII correspondiente a cada uno de los caracteres que hemos escrito en el fichero (un carácter ocupa un byte). Esta característica hace que un fichero de texto pueda ser usado por otros programas que supongan que el contenido del fichero es un conjunto de caracteres del código ASCII.

Sabemos que los datos se almacenan en memoria según las características del tipo con que han sido declarados. Así, para:

```
char C;
unsigned N;
```

La variable C usa un byte en memoria principal, y en el se almacena el código ASCII de un carácter, mientras que la variable N, al ser de tipo entero sin signo, usa cuatro bytes, y en ellos se almacena el código binario correspondiente a un número entero sin signo. Esta característica sugiere que para almacenar en el fichero de texto el número contenido en la variable N no basta con grabar directamente los cuatro bytes en el fichero (Ejemplo, si N contiene el número 13864, para almacenarlo en el fichero se requieren 5 caracteres- o sea, 5 bytes-, mientras que en memoria, la variable N ocupa cuatro bytes). Así pues, una instrucción

```
fich << N;
```

Ocasionará dos pasos:

1) Convertir N a su configuración en caracteres ASCII.

2) Escribir en el fichero los caracteres ASCII obtenidos.

En el caso del tipo CHAR, no es necesario efectuar la conversión pues la variable en memoria principal se corresponde con (es decir, contendrá como valor) el código ASCII correspondiente al carácter almacenado en la misma.

Hay situaciones en las que en lugar de convertir los valores a escribir en el fichero en una serie de caracteres, puede resultar útil efectuar la grabación en el fichero directamente a partir contenido de la memoria. A los ficheros creados de esta forma se les llama **ficheros binarios**.

Ventajas del uso de ficheros binarios:

- El proceso de lectura o escritura es más rápido, pues nos ahorramos el tiempo necesario para la conversión.
- Normalmente un fichero binario ocupa menos memoria que su correspondiente fichero de texto (Ejemplo, escribir 32534 en formato de texto ocuparía cinco bytes, mientras que hacerlo en formato binario ocuparía cuatro bytes).

Inconvenientes del uso de ficheros binarios:

- Su contenido no es directamente legible, por lo que no podrán ser usados por programas de carácter general. Por ejemplo, si intento usar la orden `TYPE` (de MS-DOS) o la orden `cat` (de UNIX) con un fichero binario, la información que obtengo por pantalla no es legible.

De todo lo dicho anteriormente se desprende que es conveniente usar ficheros de texto cuando pueda interesarnos tratar la información contenida en éstos con programas de carácter general (`Edit`, `TYPE`, `PRINT`, etc.), y en otro caso usar ficheros binarios.

Tratamiento de ficheros binarios

Puesto que no hay que distinguir entre diferentes formatos para diferentes tipos de datos, sólo se necesitan instrucciones que lean y escriban un cierto número de bytes por lo que en realidad sólo necesitamos un tipo de instrucción para lectura y un tipo de instrucción para escritura. Aún así veremos dos formas diferentes de leer y escribir desde y en ficheros binarios.

4.2. Lectura/Escritura Byte a Byte

4.2.1. Lectura

Para leer un carácter (es decir un byte) desde un fichero usamos la función *get* aplicada sobre el descriptor del mismo de la siguiente manera

descriptor.get(ch);

Esta instrucción extrae un único carácter del flujo de entrada, incluyendo el espacio en blanco y lo almacena en la variable *ch* (que debe ser declarada de tipo carácter). Esta función avanza además un byte la posición del cursor del archivo identificado por la variable *descriptor*.

EJEMPLO 7

El siguiente programa lee (byte a byte) un fichero binario que contiene caracteres y visualiza su contenido en el monitor.

```
#include <fstream>   // Librería para el manejo de ficheros
#include <iostream>

using namespace std;

int main()
{
  char c;
  ifstream fichin("F1.dat",ios::in | ios::binary);
  if (!fichin){
    cout << endl <<"Incapaz de Abrir el fichero ";
  }
  else{
    fichin.get(c);              // LECTURA ADELANTADA!!
    while (fichin){             // Tambien vale 'while (!fichin.eof())'
       cout << c;
       fichin.get(c);
    };
    fichin.close();
  } // Fin del else
  system("PAUSE");
  return 0;

 }  // Fin del main
```

4.2.2. Escritura

Para escribir o mandar un carácter (es decir un byte) a un archivo de de salida usamos la función *put* aplicada al descriptor de fichero del archivo de la siguiente manera

descriptor.put(ch);

Esta instrucción coloca el carácter contenido en *ch* en el archivo de salida identificado por la variable *descriptor* (o sea, en su caso en el fichero asociado a dicho descriptor) y lo hace a partir de la posición actual del cursor. A continuación avanza el cursor del fichero al siguiente byte.

EJEMPLO 8

El siguiente programa escribe un texto (byte a byte) en el fichero "Ejemplo8.dat".

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

using namespace std;

int main(){

  char  cad[17]="TEXTO A ESCRIBIR";
  ofstream fichout("Ejemplo8.dat", ios::out | ios::binary);

  if (!fichout){
    cout << endl << "Incapaz de Crear o Abrir el fichero ";
  }
  else{
    for (int i=0;i<=16;i++){
      fichout.put(cad[i]);                  // Escritura en el fichero
    }
    fichout.close();
  }   // Fin del else
  system("PAUSE");
  return 0;
}  // Fin del main
```

4.3. Lectura/Escritura por Bloque de Bytes

Otra manera alternativa de leer y escribir en ficheros binarios consiste en utilizar las funciones *read()* y *write()*.

4.3.1. Lectura

La función *read* tiene varios formatos aunque a continuación explicamos el formato básico de forma intuitiva, una llamada de la forma

*descriptor.read((char *)&c, num);*

donde *c* es una variable de un tipo arbitrario (por ejemplo de un tipo *TElemento*), pasada por referencia (su dirección se extrae con el operador &), y *num* es un entero o una variable de tipo entero, ejecuta una lectura (a partir de la posición actual del cursor del fichero asociado a la variable *descriptor*) de *num* bytes del fichero y los coloca en la variable *c*. En definitiva la llamada lee *num* bytes del fichero y los coloca en la variable *c*. Esta función puede extraer caracteres hasta alcanzar el fin del fichero.

Nótese además que la función read requiere que la dirección de la variable *c* sea de tipo puntero a carácter. Para conseguir esto, se utiliza la conversión de tipos o *casting*, anteponiendo el nuevo tipo que reemplazará al devuelto por el operador &: *(char *)&c* .

4.3.2. Escritura

Asumiendo que c es una variable de un tipo arbitrario (por ejemplo de un tipo *TElemento*) pasada por referencia, que *num* es un entero o una variable conteniendo un entero que *descriptor* está asociado a un fichero, una llamada de la forma

Descriptor.write((char *)&c, num);

Escribe, a partir de la posición del cursor, el contenido de *c* en el fichero asociado a *descriptor.* En realidad sólo escribe *num* bytes que corresponden a los *num* bytes siguientes que se encuentran en memoria a partir de la dirección en la que se encuentra almacenado el contenido de *c*. De nuevo, obsérvese el uso de casting a puntero a carácter.

Esta instrucción escribe a partir de la posición indicada por el puntero de lectura/escritura del fichero asociado a *descriptor* los *num* bytes contenidos en el parámetro *c*.

Se puede observar que en el segundo parámetro hay que indicar el número de bytes que ocupa el valor que se desea escribir. Para conocer éste dato de una forma fácil , podremos usar la función *sizeof*.

EJEMPLO 9

El siguiente programa declara el fichero "F.dat" para entrada-salida, graba en dicho fichero el valor 1234.86 en binario y después los veinte primeros enteros. Posteriormente, lee el fichero visualizando su información en la salida estándar (el monitor).

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

using namespace std;

int main(){
  float R=1234.86;
  int i,N;

  fstream fichbin("F.dat",ios::binary | ios::out);  // Apertura como salida
  fichbin.write((char*)&R,sizeof(float));
  for (i=1;i<=20;i++){
    fichbin.write((char*)&i,sizeof(int));
  }

  fichbin.close();

  fichbin.open("F.dat",ios::binary | ios::in);     // Apertura como entrada
  fichbin.read((char*)&R,sizeof(float));
  cout <<endl << "R= " << R << endl;
  for (i=1;i<=20;i++){
      fichbin.read((char*)&N,sizeof(int));
      cout <<endl << i << "= " << N << endl;
  }
  system("PAUSE");
  return 0;
}  // Fin del main
```

Observa que la apertura y el cierre de ficheros binarios se puede efectuar con las mismas operaciones estudiadas para ficheros de texto.

7. REFERENCIAS

[1] Departamento de Lenguajes y Ciencias de la Computación, 2005-2006. [En línea]. Available: altair.lcc.uma.es/clases/laboratorio/curso200102/tema9.pdf. [Último acceso: 30 11 2012].