Flujos (Archivos) Binarios

Cuando hablamos de flujos binarios significa que los datos **NO** sufren ninguna transformación. En los flujos de textos una variable entera, que puede representarse internamente con 4 bytes y en complemento a 2, se transforma en una cadena de dígitos. Así, si el contenido de esos 4 bytes, representados en hexadecimal, es `00017BE8` al escribir a un flujo de texto se transformará en la cadena `97256`, cambiando en forma y tamaño. Si en cambio el entero tiene el patrón binario `0000000F` la cadena correspondiente es 15, esta vez de menor largo que la representación binaria. Por el contrario en un flujo binario se respeta el patrón de bits, simplemente se pasa tal cual de memoria al archivo o viceversa.

Características de los flujos binarios:

- Al no haber transformaciones son más eficientes
- Al guardar valores numéricos de un determinado tipo de datos, el largo es siempre el mismo.

Esta última característica permite usar un archivo binario como una arreglo de datos de una dimensión, conociendo el tamaño de cada elemento y usando las funciones `seekp` y `seekg` que vimos al tratar flujos de texto, uno puede leer directamente el iésimo elemento.

Manejo en C++

Es muy similar a los flujos de texto, las variables que vamos a declarar son del mismo tipo. Cambia el modo de apertura, hay que indicar explícitamente que se va a tratar el flujo en modo binario, para ello usamos `ios::binary` en el modo de apertura:

```
ofstream archiesc; //archivo de escritura
archiesc.open("Archivo.bin", ios::binary); //Solo falta agregar binario

ifstream archilec; //archivo de lectura
archilec.open("Archivo.bin", ios::binary);

fstream archi; //debo especificar como lo abro, además de binario
archi.open("Archivo.bin", ios::binary | ios::out); //Para abrirlo de escritura
archi.open("Archivo.bin", ios::binary | ios::in); //Para abrirlo de lectura

//Para abrirlo de lectura y escritura
archi.open("Archivo.bin", ios::binary | ios::in | ios::out);
```

Para leer y escribir "en binario" usamos las funciones `read` y `write`. Ambas llevan los mismos parámetros: el primero es un puntero a la zona de memoria donde están los datos que quiero escribir, en el caso de write, o donde quiero poner los datos leídos, en caso de read. Este parámetro es de tipo `char *` por tanto lo habitual es hacer un casting para adecuar el tipo de puntero. Casting es la operación que transforma un tipo de dato en otro, en particular en el ejemplo que nos ocupa es considerar que un puntero a un tipo de datos en realidad apunta a otro tipo, lo que no cambia al puntero en sí, sigue guardando la misma dirección de memoria, simplemente interpreta los datos en esa dirección de otro modo (como una sucesión de caracteres). El segundo parámetro es un entero que indica el tamaño en bytes a escribir o leer, el cual calcularemos usando el operador `sizeof`.
Entonces si quiero grabar un int con valor `97256` tal como comentamos antes haríamos:

```
int nro = 97256;
archi.write((char *) &nro, sizeof(nro));
```

El casting usando `(char *)` es el "modo C" de hacerlo, es más corto de escribir que el "modo C++" y por eso muchos lo utilizan, pero sería más correcto usar el "modo C++" :

```
archi.write(reinterpret_cast<char *>(&nro), sizeof(nro));
```

Y para leer el valor desde el archivo:

```
archi.read((char *) &nro, sizeof(nro));//Casting en C
archi.read(reinterpret_cast<char *>(&nro), sizeof(nro));//Casting en C++
```

Escribir y leer estructuras

Es habitual que lo que se grabe en un archivo binario sea una estructura, porque esta aglutina información en modo similar a un registro en una base de datos. Como las funciones `read` y `write` trabajan con una cantidad de bytes contiguos en memoria, en tanto el `struct` que usemos se guarde en forma contigua en memoria, todo funciona como queremos.
Así en el ejemplo 01-StructSimple tenemos un `struct` formado por un `int` y un `double`, esto no nos trae ningún problema. Notar que abajo de los ciclos for donde escribe y lee hay comentada una línea de código que aprovecha el hecho de que un vector ubica sus elementos en memoria contigua, lo que permite escribir o leer el vector completo con una sola sentencia.
Dependiendo el caso podemos preferir el ciclo for, si tenemos que hacer algún tratamiento a cada dato leído antes de acceder al siguiente, o bien leer todo junto. Notar que en ambos casos puedo controlar si tuve éxito consultado la variable de flujo.
En el caso del primer for, el de escritura, usé `sizeof(Alumno)` lo que es claro y entendible, pero es preferible usar como en los otros casos `sizeof(vec[0])` porque es más fácil de mantener. Si cambiamos el tipo de datos de `vec` igualmente dará el tamaño correcto. Por otra parte no se pueden definir vectores sin elementos, por tanto el elemento de índice cero siempre existe.

Estructuras no contiguas en memoria

El problema lo tenemos con estructuras que tienen campos que no son contiguos en memoria. En nuestro caso nos complican los campos que sean de tipo `string` que dependiendo del tamaño de la cadena guardada y de la implementación de compilador puede guardarse en forma contigua algunas veces, pero en general no (nota: para cadenas largas, digamos más de 15 o 20 caracteres está casi garantizado que no se guardará en forma contigua). Para comprobarlo tenemos el ejemplo 02-StructNoContigua que es el mismo código del ejemplo anterior pero agregando un campo `string` nombre. Si corremos el ejemplo como viene al terminar da error. Esto es porque en el archivo se guardaron direcciones de memoria que aún están activas en el programa, pero falla al querer liberarlas dos veces (una por `vec` y otra por `vacio`) al final del programa.
Si descomentamos el for en líneas 55 y 56 el error es más evidente. Una mejor prueba aún es, una vez que el archivo ya se generó, comentar la primer parte donde se escribe el archivo (líneas 38 a 50) y que directamente lea, entonces las direcciones que carga son inválidas y falla en cuanto intenta mostrar (acceder) a las direcciones inválidas.

Por supuesto hay varias maneras de resolver el problema, veremos algunas y discutiremos alternativas. Genéricamente hablando el tema se lo denomina serialización y consiste en distinguir "el dato" (nuestra estructura) en memoria, del dato guardado o transmitido (los problemas de almacenar y transmitir son muy similares). No siempre voy a querer guardar todos los campos de una estructura y puede que algunos campos deba transformarlos para poder guardarlos (este es el caso que nos ocupa).

Modo C

En general la cátedra resuelve el problema usando lenguaje C y su biblioteca estándar. No es que C resuelva cosas que C++ no, lo que sucede es que cambian el uso de `string` por el uso de vectores de caracteres, que si cumple con ser contiguo en memoria.
Pero este enfoque tiene los siguientes inconvenientes. Las funciones de manejo de archivo cambian, usamos `FILE *`, `fopen`, `fwrite`, etc. Todas similares a las de C++ pero diferentes. Este es en realidad el menor de los problemas. El mayor inconveniente es que perdemos las capacidades de `string`, al usar un vector de caracteres debo cuidar en todo momento de no pasarme de la capacidad del vector, mientras que `string` maneja dinámicamente la memoria permitiendo cualquier largo. Perdemos también la capacidad de comparar con operadores, si `s1` y `s2` son strings puedo preguntar si `s1 == s2` o si `s1 > s2`. En cambio en C debo usar las rutinas `strcmp`. Con strings puedo asignar como con cualquier variable y hacer `s1 = s2;` en C debo usar la función `strcpy`.
En el ejemplo 03-BinarioCPuro vemos también el tema de aprovechar que el vector guarda todo en forma contigua (línea 48 comentada) además las líneas siguientes también comentadas muestran un posible control de éxito (para el for podría ver si `i` termina valiendo `dim`).

Veamos entonces que podemos hacer si no queremos perder el uso de string. Hay dos estrategias, la primera es tener una estructura diseñada para serializar y cada vez que debo guardar una de mis estructuras la "copio" haciendo las transformaciones necesarias en la de serialización, y esa es la que guardo. Obviamente para leer, leo en la estructura de serialización y luego la "copio" en mi estructura de trabajo. Esto se puede hacer, por supuesto, tanto en C como en C++. El ejemplo 04-BinarioCSerial muestra un posible modo de hacerlo usando el "modo C". Notar que el alumno de legajo 3 tiene un nombre más largo que lo que guardamos en el archivo, por eso al recuperarlo aparece recortado a ese largo.

En el ejemplo 05-BinarioCppSerial usamos la misma estrategia, pero todo escrito en C++ y con rutinas de conversión de una estructura a otra.

Modo C++

La segunda estrategia es tener una función específica para grabar y otra para leer, que graban o leen campo a campo y hacen las conversiones necesarias antes de grabar o después de leer. En el ejemplo 06-BinarioCPPfun lo hacemos así y en "modo C++". Notar que el tema de recortar algo al grabar se repite igual que en el ejemplo anterior.
Las funciones `writebin` y `readbin` reciben el flujo donde escribir y la estructura que queremos manipular. En el caso de `writebin` la estructura no era necesario pasarla por referencia, pero es más eficiente hacerlo así.
El flujo se pasa por referencia porque tiene los datos de control de flujo, pasarlo por valor no tiene sentido y traería problemas, para evitar eso `fstream` está diseñado de modo tal que no se puede pasar por valor, si se intenta da error de compilación. Devolver como resultado una referencia al flujo es lo que permite usar las funciones para controlar el for (ver línea 81).
Comentario: en la función `writebin` se usa el `string aux` porque declaramos `alu` como una referencia constante, es decir, que no será modificada dentro la función. Si hubiese declarado el parámetro por valor, no sería necesario.

Analizando el ejemplo que acabamos de ver, se nota que las funciones `writebin` y `readbin` tienen las

conversiones incluidas, lo que hace menos legible el código. Si tengo varios campos `string` esta observación se hace aun más notoria. En el ejemplo 07-BinarioCPPfun2 abordamos este problema, separamos el nombre en nombre y apellido para tener dos campos string. Las funciones `writebin` y `readbin` quedan más largas y repiten el código de conversión, haciendo evidente lo que comentamos antes, conviene sacar la conversión a otra función. Notar que al tener dos largos distintos para nombre y apellido se optó por tener un único buffer estático para los dos. Esta es una opción muy eficiente, el problema latente es que con el tiempo se agregue un campo que guarde más caracteres que el tamaño de ese buffer y no se actualice la dimensión del mismo.
Se incluye también las funciones `writebinf` y `readbinf` que cumplen el mismo papel que `writebin` y `readbin` (en main puedo usar cualquiera de las dos y obtener los mismos resultados) pero tienen la conversión separada en otra función. Notar como ahora quedan mucho más legibles y simples.
Usando las funciones auxiliares `writestring` y `readstring` ahora todo lo que hay que hacer es un `write` o un `read` por cada campo contiguo en memoria (en nuestro caso: que no sea string) y y usar las funciones auxiliares en los casos de string.
Notar que para el legajo 3 ahora queda recortado tanto el nombre como el apellido.

Sobrecarga

En nuestra versión "final" el ejemplo 08-BinarioCPPSobrec tomamos el mismo código del ejemplo anterior y lo terminamos de acomodar. Cuando teníamos flujos de texto usábamos los operadores `>>` para leer y `<<` para escribir. Podemos hacer lo mismo con flujos binarios, solo hay que sobrecargar esos operadores con el código de `writebinf` y `readbinf`, para lo cual basta con reemplazar el nombre de la función con `operator >>` u `operator <<` según sea el caso. Noten que prolijo quedó `main` ahora usando esos operadores.
En este ejemplo las funciones auxiliares las puse en un archivo aparte, el cual lo agrego al `main` mediante la directiva `#include` la que a su vez la hago después de `using namespace std;` porque uso cosas de `std` y si no debería poner en varios lados `std::`
En cuanto a `writestring` no hay mucho para probar, simplemente hace `resize`, lo que trunca el `string` si es más largo y agrega `'\0'` hasta completar la cantidad de caracteres pedidos si el `string` es más corto.
En cambio `readstring` presenta variantes. La primera es simple y eficiente aunque requiere cierto cuidado. Lo ideal sería hacer lo que está comentado en línea 17 de `rwstring.hpp`, definir un buffer de `largo + 1` de modo tal que se adapta al largo pasado como parámetro, lo que nos da cero problemas de mantenibilidad. Pero el estándar no permite una dimensión que no se conoce en tiempo de compilación. En línea 18 definimos un buffer con tamaño `bufsize` que debemos definir con un valor adecuado. Notar que en main antes de incluir rwstring.h lo definí con el mayor valor posible para este ejemplo. Definirlo con un valor que cumpla, también vale, como se muestra en línea 7 de main.c, que está comentada, o haber puesto directamente un valor 11, preferentemente con un comentario que indique que es `lape + 1` ya que lape es el largo máximo con el que vamos a serializar.
La desventaja de esta primera versión es que si cambian los largos a serializar hay que tener cuidado de actualizar correctamente a `bufsize`. La variante `readstring2` es menos eficiente pero tiene la ventaja de adecuarse dinámicamente al largo pedido en el parámetro. Es menos eficiente porque tiene que copiar a `str` el buffer antes de borrarlo. Esta variante es la consideramos recomendada para uso en clase, en TP o en exámenes.

Resumen

Lo importante es sobrecargar los operadores `>>` y `<<` como dijimos antes, usando un `write` o `read`

para todos los campos menos los que sean `string`, y para éstos usar `readstring` o `writestring`.
El ejemplo 09-Resumen es el ejemplo 8 pasado en limpio, usando la segunda variante de `readstring` y tiene la ventaja que puedo hacer el `include` al principio sin preocuparme por definir nada antes, ni calcular el tamaño del buffer, es la más apropiada.