UNIDAD 12

Archivos de Disco

Manejo de la clase fstream

1.- Introducción.

El manejo de archivos usando **Programación Orientada a Objetos** nos brindan una forma de leer y escribir información en un disco. La librería proporcionada por la mayoria de los compiladores de C++ es diseñada para trabajar solamente con archivos de texto, sin embargo este hecho no nos limita su uso con archivos binarios, como veremos en el desarrollo del tema.

2.- Archivos de texto.

Los archivos de texto son simples y facil de usar, al estudiar este tipo de archivos debemos cubrir 3 áreas principales:

+ Creación de un archivo de texto.

+ Escritura a un archivo de texto.

+ Lectura de un archivo de texto.

En la mayoría de los casos debemos de usar las clases *ifstream* y *ofstream* para llevar a cabo estos trabajos. Debido a que los archivos de texto se organizan por líneas de longitud variable no es conveniente intentar leer y escribir simultameamente en un archivo utilizando la clase *fstream*

2.1) Creación y escritura de un archivo de texto.

Para crear un archivo de texto nuevo es necesario definir un objeto del tipo *ostream* y pasar dos argumentos al constructor de la clase:

1.- El nombre del archivo, es un puntero a char.

2.- El modo de apertura del archivo, es una constante enumerada.

Ejemplo:
```
ofstream A("MIARCH.DAT", ios::out);
```

Al construir el objeto del tipo *ofstream*, se crea el archivo, si no existe, y si existe se sobre-escribe. Note que en este caso no se define un manejador de archivo, como en “C”, el objeto hace las veces de éste.

El segundo argumento *ios::out* selecciona un modo de acceso para el archivo. En la siguiente tabla se muestran los modos de apertura representados mediante constantes enumeradas. Estos modos se pueden combinar mediante el uso del operador **OR** en C++, (|).

Constante	Estandar	Efecto
app		La información es escrita al final del archivo. Probablemente tiene el mismo efecto que *ate*.
ate	+	Salta al final del archivo cuando éste es abierto. La palabra es contracción de *at end*
binary	+	Puede especificarse como *bin* en algunas implementaciones. Abre el archivo en modo binario.
in	+	Abre el archivo para lectura.
nocreate		Si el archivo no existe, no se crea un nuevo archivo.
noreplace		Si el archivo ya existe, no lo sobre-escribe.
out	+	Abre el archivo para escritura.
trunc		Abre y trunca un archivo existente. La nueva información remplaza al contenido anterior.

Desafortunadamente las constantes enumeradas no son las mismas en todas las implementaciones de la clase *ios*, a excepción de los marcados con el signo de [+], los cuales son disponibles universalmente. Los restantes pueden o no estar definidos o puede haber otras constantes agregadas para propósitos especiales.

Para ayudar a asegurar la portabilidad se pueden definir constantes simbólicas, por ejemplo:

```
#define OFSMODE ops::out | ios::app
```

ó

```
const ops::open_mode OFSMODE = ios::out | ios::app;
```

La creación del objeto sería de la forma:

```
ofstream A(“MIARCH.DAT”, OFSMODE);
```

Una vez creado o sobre-escrito el archivo es conveniente verificar si el objeto de la clase *ofstream* esta listo para ser utilizado. Para ésto utilice la siguiente sentencia:

```
ofstream A("MIARCH.DAT", ios::out);
if(!A) {
     cerr << "Imposible escribir en el archivo" << endl;
     exit(1);
}
```

La primera línea instancia al objeto **A**.

La sentencia `if` prueba para ver si el archivo fue abierto adecuadamente, si no lo fue, el programa escribe un mensaje de error y sale del programa. Note la sobre-escritura del *operator!()*, la cual tal vez sea implementada de la siguiente manera:

```
int operator!() { return fail(); }
```

En realidad la función esta regresando el resultado de la función miembro de *ios,* **fail().**

La apertura de un archivo para lectura será de la forma:

```
ifstream A("MIARCH.DAT", ios::in);
if(!A) {
     cerr << "Imposible leer en el archivo" << endl;
     exit(1);
}
```

A continuación vamos a ver algunos ejemplos donde se aplican las técnicas vistas:

2.1.1) Lectura de texto, un carácter a la vez

```
// PLCP108.CPP Lectura de un carácter a la vez

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>


void main()
{
    char narch[15];

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);
    ifstream A(narch, ios::in);
```

```
    if(!A) {
        cerr << "Error en apertura de: " << narch << endl;
        exit(1);
    }

    char c;
    long nc = 0, nl = 0;
    while(A.get(c)) {
        if(c == '\n')
            nl++;
        else
            nc++;
        cout.put(c);
    }

    cout << "Número de lineas:     " << nl << endl;
    cout << "Número de caracteres: " << nc << endl;
}
```

OBSERVACIONES.

El programa anterior además de leer un archivo ASCII, cuenta el número de caracteres y el número de líneas en el archivo, lo cual se basa en el carácter “Salto de línea” (‘\n’).

La lectura del archivo se realiza en el ciclo **while**, tomando del archivo carácter por carácter, si es ‘\n’, incrementa la cuenta de los renglones, si no, incrementa la cuenta de los caracteres y lo muestra en pantalla. El ciclo termina cuando la función miembro regresa EOF.

2.1.2) Escribiendo texto, un caracter a la vez

```
// Programa PLCP109.CPP. Escribe en un archivo
// un caracter a la vez
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main()
{
    char narch[15];

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);
    ifstream A(narch, ios::in);
    if(A) {
        cerr << "Error el archivo: " << narch << endl;
```

```
        cerr << "Ya existe, especifique un nombre diferente"
             << endl;
        exit(1);
    }

    ofstream B(narch, ios::out);
    if(!B) {
        cerr << "Error en apertura de: " << narch << endl;
        exit(2);
    }

    B << "Escribiendo a archivo\n";
    B.put('C');
    B.put('a');
    B.put('r');
    B.put('a');
    B.put('c');
    B.put('t');
    B.put('e');
    B.put('r');
    B.put(' ');
    B.put('h').put('o').put('l').put('a');
    B << endl;
}
```

OBSERVACIONES.

Para prevenir que al tratar de escribir en un archivo, vayamos a destruir uno ya existente, se trata de abrir el archivo indicado, si se puede abrir, quiere decir que el archivo ya existe, se manda un mensaje y se termina el programa.

Si el archivo no se pudo abrir, entonces no existe y se puede crear para escritura, se verifica que haya sido abierto.

El escribir en el archivo, primero escribimos una cadena usando el operador sobrecargado (<<) heredado de la clase *ofstream*. Posteriormente escribimos carácter por caracter, terminando con un salto de línea.

2.1.3) Lectura de un archivo de texto, una línea a la vez.

Al leer o escribir en un archivo una línea a la vez se mejora la velocidad de transferencia de información, el siguiente programa también cuenta el número de caracteres y líneas leídas.

```
// PLCP110.CPP, Lectura por línea de un archivo de texto.
```

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

#define BUFLEN 128

void main()
{
    char narch[15];

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);
    ifstream A(narch, ios::in);
    if(!A) {
        cerr << "Error en apertura de: " << narch << endl;
        exit(1);
    }

    char buffer[BUFLEN];
    long nc = 0, nl = 0;
    while(!A.eof()) {
        A.getline(buffer, sizeof(buffer), '\n');
        if(!(A.eof() && strlen(buffer) == 0)) {
            nc += strlen(buffer);
            nl++;
            cout << buffer << endl;
        }
    }
    cout << "\nTotal caracteres: " << nc;
    cout << "\nNúmero de lineas: " << nl << endl;
}
```

OBSERVACIONES.

Cuando se desea leer un archivo línea por línea se puede usar la siguiente sentencia:

```
A >> buffer;
```

Sin embargo si el arreglo *buffer* no es lo suficientemente grande para almacenar la línea se escribe fuera del *buffer*, lo que puede ocasionar problemas en la ejecución del programa, es mejor usar la sentencia:

```
A.getline(buffer, sozeof(buffer), '\n');
```

Otra forma sería utilizar el manipulador **setw**.

```
A >> setw(BUFLEN) >> buffer;
```

Mediante un ciclo **while** se lee todo el archivo hasta encontrar el EOF.

2.1.4) Escritura de un archivo de texto, una línea a la vez.

La última técnica de escritura de un archivo es hacerlo una línea a la vez, lo que también incrementa la velocidad de transferencia de información.

```
// PLCP111.CPP. Escritura de un archivo por línea
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#define STR "2: Esta es otra cadena de caracteres"

void main()
{
    char narch[15];

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);
    ifstream A(narch, ios::in);
    if(A) {
        cerr << "Error el archivo: " << narch << endl;
        cerr << "Ya existe, especifique un nombre diferente"
             << endl;
        exit(1);
    }

    ofstream B(narch, ios::out);
    if(!B) {
        cerr << "Error en apertura de: " << narch << endl;
        exit(2);
    }

    B << "1: Esta es una cadena de caracteres" << endl;
    B.write(STR, strlen(STR));
    B << endl;

    char *a = "Esta es una cadena manejada por puntero";
    B << "3: " << a << endl;
}
```

OBSERVACIONES.

En este programa se verifica primero si el archivo ya existe intentando abrirlo para lectura, ésto se hace con la finalidad de no ir a destruir un archivo que ya exista. Si el archivo no existe entonces lo crea abriéndolo para escritura.

Para escribir una línea a un archivo se pueden usar dos métodos:

a) Se puede escribir una cadena literal:

```
B << "Escribe esta cadena a disco" << endl;
```

Es necesario usar el manipulador **endl**, para mandar toda la línea al archivo.

b) Se puede llamar a la función miembro **write()**, la cual requiere dos argumentos, un puntero a char y el número de caracteres a escribir al archivo.

```
char *s = "Esta es una cadena de caracteres";

B.write(s, strlen(s));
```

Aunque esta función por lo general se usa para escribir solo un cierto número de caracteres al archivo, (un número de caracteres menor que el tamaño de la cadena).

```
Ejemplo:
          if(strlen(s) >= 5)
               B.write(s, 5);
```

3.- Archivos Binarios.

Este tipo de archivos no están definidos en la librería de C++, aunque permite la apertura de archivos binarios, no cuenta con métodos para manejar datos reales, o estructuras con elementos de cualquier tipo.

Para la manipulación de este tipo de archivos en operaciones de lectura y escritura es necesario agregar algunas capacidades a la clase para el manejo de archivos.

Se requiere desarrollar alguna programación extra para:

+ Escribir uno o mas byte de cualquier tipo a un archivo.
+ Leer uno o mas bytes de cualquier tipo de un archivo.
+ Transformar un objeto de cualquier tamaño a bytes.
+ Transformar un conjunto de bytes a un objeto.

3.1) Usando un manejador para archivos binarios.

La construcción de un objeto para un archivo binario es casi igual a la construcción de un objeto para un archivo de texto, solo que en este caso es necesario adicionar la constante enumerada ios::binary ó ios:bin de la siguiente manera:

a) Escritura en un archivo binario.

```
ofstream A("miarch.dat", ios::out | ios::binary);
if(!A) {
     cerr << "ERROR: imposible abrir el archivo\n";
     exit(1);
}
```

b) Lectura de un archivo binario.

```
ifstream B("miarch.dat", ios::in | ios::binary);
if(!B) {
     cerr << "ERROR: imposible abrir el archivo\n";
     exit(1);
}
```

Desafortunadamente los objetos **A** y **B** no estan listos para ser utilizados con archivos de datos binarios, es necesario realizar algunas acciones adicionales.

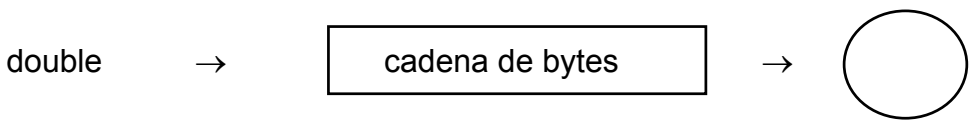
Para leer y escribir datos binarios en su *forma nativa*, debemos hacer lo siguiente:

1) Cree una clase propia derivada de las clases *ifstream* o de *ofstream*

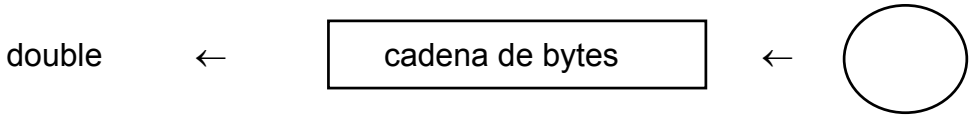
2) Para la escritura. El contenido de una variable por ejemplo del tipo *double* deberá ser convertida a una cadena de bytes, y entonces mandada al archivo como una cadena de bytes.

3) Para la lectura. Se lee del archivo un cierto número de bytes del disco, éste se coloca en una cadena y posteriormente se almacena por ejemplo en una variable del tipo double.

Escritura a disco. Un valor double en su *forma binaria nativa* es copiado a una cadena de bytes y luego esta cadena es escrita al archivo.



Lectura de Disco. En la lectura de un doble de disco, la cadena de bytes en el disco es copiada a una cadena de bytes en memoria, luego se asigna a una variable del tipo double.



Esta técnica puede utilizarse con cualquier tipo de dato inclusive con objetos.

En el siguiente programa se ejemplifica el método anteriormente descrito, primero para la escritura a disco de un tipo *double* y luego para la lectura de estos valores de disco.

a) Escritura de un archivo binario.

```
// PLCP112.CPP Clase bofstream para escritura de
// un archivo binario

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>


class bofstream : public ofstream {
// No contiene datos miembro.
public:
    bofstream(const char *fn)
    : ofstream(fn, ios::out | ios::binary) {}
    void writebyte(const void *, int);
    bofstream &operator<<(double);
};


void bofstream::writebyte(const void *p, int len)
{
    if(!p) return;
    if(len == 0) return;
    write((char *)p, len);
}



inline bofstream &bofstream::operator<<(double d)
{
    writebyte(&d, sizeof(d));
    return *this;
```

```
}



void main()
{
    char narch[15];

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);

    bofstream A(narch);
    if(!A) {
        cerr << "ERROR, imposible abrir archivo: "
             << narch << endl;
        exit(1);
    }

    double d = 3.14159;
    A << d;
    A << d*d;
    A << 9.998877;
    d = 4.7E-8;
    A << d;
}
```

OBSERVACIONES.

Se escribe la clase **bofstream** para proveer una clase de salida al archivo, esta clase se deriva de la clase ofstream. La nueva clase creada contiene tres funciones miembro.

+ El constructor de la clase **bofstream**, por simplicidad solo se requiere como argumento el nombre del archivo, el modo de apertura del archivo es fijo.

+ La función miembro **writebyte()**, escribe uno o mas bytes al archivo a partir de la dirección **void ***, haciendo uso de la función **write()**, definida para la clase **ios**.

+ El operador miembro sobrecargado **operator<<()**, él cual llama a la función **writebyte**(), la cual recibe como argumento la dirección de la variable del tipo double, y su número de bytes (8 en este caso). Por lo general este tipo de operadores se manejan como *friend* para hacerlo compatible con las sobrecargas ya definidas para el operator (**<<**).
  El objeto de regresar ***this**, es el poder manejar el operador encadenandolo con otros operadores (**<<**).

La forma de llamar a este operador es:

```
A << d;
```

b) Lectura de un archivo binario.

```
// PLCP113.CPP Clase bifstream para lectura
// de un archivo binario

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>


class bifstream : public ifstream {
// No contienwe datos miembro.
public:
    bifstream(const char *fn)
    : ifstream(fn, ios::in | ios::binary) {}
    void readbyte(void *, int);
    bifstream &operator>>(double &d);
};


void bifstream::readbyte(void *p, int len)
{
    if(!p) return;
    if(len == 0) return;
    read((char *)p, len);
}


inline bifstream &bifstream::operator>>(double &d)
{
    readbyte(&d, sizeof(d));
    return *this;
}


void main()
{
    char narch[15];

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);

    bifstream A(narch);
```

```
    if(!A) {
        cerr << "ERROR, imposible abrir archivo: "
             << narch << endl;
        exit(1);
    }

    double d;
    long cont = 0;

    cout.precision(8);


// Proceso de lectura y escritura en pantalla.
    A >> d;
    while(!A.eof()) {
        cout << ++cont << ": " << d << endl;
        A >> d;
    }
}
```

OBSERVACIONES.

Con la finalidad de proveer un objeto que maneje el archivo de entrada se crea la clase *bifstream* derivada de *ifstream* con un constructor similar al de la clase *bofstream*, que nos permita abrir el archivo binario para lectura, también se agrega la función **readbyte()**, y un operador (**>>**) sobrecargado para la lectura del archivo.

El operator inline (**>>**) llama a la función **readbyte()** copia bytes directamente de un archivo binario a una variable del tipo *double* en memoria. Luego de abrir el archivo, mediante una sentencia simple se lee el valor *double* como se muestra:

```
A >> d;
```

La función miembro **readbyte()** es similar a la función miembro **writebyte()** en el programa PLCP110.CPP, solo que esta vez se llama a la función heredada **read()**, la cual solo lee un número determinado de caracteres de un archivo, depositándolos en la dirección indicada por (void *p).

4.- Clases para archivos binarios de E/S.

Mejorando los conceptos vertidos en la sección anterior, es posible crear una versión mejorada de las clases *bofstream* y *bifstream* para leer y escribir cualquier tipo de dato definido en C++. A continuación se muestran estas clases las cuales fueron tomadas del libro:

Tom Swan's Code Secrets
Tom Swan
SAMS Publishing
1993.

4.1) Interface.

```
/* ------------------------------------------------------------ *\
**  bstream.h -- Header for bofstream and bifstream classes     **
** ------------------------------------------------------------ **
**                                                              **
** ------------------------------------------------------------ **
**     Copyright (c) 1993 by Tom Swan. All rights reserved      **
\* ------------------------------------------------------------ */

#ifndef __BSTREAM_H
#define __BSTREAM_H     // Prevent multiple #includes

#include <iostream.h>
#include <fstream.h>

// Binary output file stream

class bofstream: public ofstream {
public:
  bofstream(const char *fn)
      : ofstream(fn, ios::ate | ios::binary) { }
  bofstream(const char *fn, int)
      : ofstream(fn, ios::out | ios::binary) { }
  void writeBytes(const void *, int);
  friend bofstream & operator<< (bofstream &, signed char);
  friend bofstream & operator<< (bofstream &, unsigned char);
  friend bofstream & operator<< (bofstream &, signed short);
  friend bofstream & operator<< (bofstream &, unsigned short);
  friend bofstream & operator<< (bofstream &, signed int);
  friend bofstream & operator<< (bofstream &, unsigned int);
  friend bofstream & operator<< (bofstream &, signed long);
  friend bofstream & operator<< (bofstream &, unsigned long);
  friend bofstream & operator<< (bofstream &, float);
  friend bofstream & operator<< (bofstream &, double);
  friend bofstream & operator<< (bofstream &, long double);
};

// Binary input file stream

class bifstream: public ifstream {
public:
  bifstream(const char *fn)
      : ifstream(fn, ios::in | ios::binary) { }
  void readBytes(void *, int);
  friend bifstream & operator>> (bifstream &, signed char &);
  friend bifstream & operator>> (bifstream &, unsigned char &);
  friend bifstream & operator>> (bifstream &, signed short &);
```

```
  friend bifstream & operator>> (bifstream &, unsigned short &);
  friend bifstream & operator>> (bifstream &, signed int &);
  friend bifstream & operator>> (bifstream &, unsigned int &);
  friend bifstream & operator>> (bifstream &, signed long &);
  friend bifstream & operator>> (bifstream &, unsigned long &);
  friend bifstream & operator>> (bifstream &, float &);
  friend bifstream & operator>> (bifstream &, double &);
  friend bifstream & operator>> (bifstream &, long double &);
};

inline bofstream & operator<< (bofstream &bofs, signed char q)
{
  bofs.writeBytes(&q, sizeof(signed char));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, unsigned char q)
{
  bofs.writeBytes(&q, sizeof(unsigned char));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, signed short q)
{
  bofs.writeBytes(&q, sizeof(signed short));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, unsigned short q)
{
  bofs.writeBytes(&q, sizeof(unsigned short));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, signed int q)
{
  bofs.writeBytes(&q, sizeof(signed int));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, unsigned int q)
{
  bofs.writeBytes(&q, sizeof(unsigned int));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, signed long q)
{
  bofs.writeBytes(&q, sizeof(signed long));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, unsigned long q)
{
  bofs.writeBytes(&q, sizeof(unsigned long));
  return bofs;
}
```

```
inline bofstream & operator<< (bofstream &bofs, float q)
{
  bofs.writeBytes(&q, sizeof(float));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, double q)
{
  bofs.writeBytes(&q, sizeof(double));
  return bofs;
}

inline bofstream & operator<< (bofstream &bofs, long double q)
{
  bofs.writeBytes(&q, sizeof(long double));
  return bofs;
}

inline bifstream & operator>> (bifstream &bifs, signed char &q)
{
  bifs.readBytes(&q, sizeof(signed char));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, unsigned char &q)
{
  bifs.readBytes(&q, sizeof(unsigned char));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, signed short &q)
{
  bifs.readBytes(&q, sizeof(signed short));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, unsigned short &q)
{
  bifs.readBytes(&q, sizeof(unsigned short));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, signed int &q)
{
  bifs.readBytes(&q, sizeof(signed int));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, unsigned int &q)
{
  bifs.readBytes(&q, sizeof(unsigned int));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, signed long &q)
{
  bifs.readBytes(&q, sizeof(signed long));
  return bifs;
}
```

```
inline bifstream & operator>> (bifstream &bifs, unsigned long &q)
{
  bifs.readBytes(&q, sizeof(unsigned long));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, float &q)
{
  bifs.readBytes(&q, sizeof(float));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, double &q)
{
  bifs.readBytes(&q, sizeof(double));
  return bifs;
}

inline bifstream & operator>> (bifstream &bifs, long double &q)
{
  bifs.readBytes(&q, sizeof(long double));
  return bifs;
}

#endif  // __BSTREAM_H


//-----------------------------------------------------------------
// Copyright (c) 1993 by Tom Swan. All rights reserved.
// Revision 1.00    Date: 02/12/1993   Time: 07:51 am
```

b) La implementación.

En este caso solo corresponde a las funciones **WriteByte()** y **ReadByte()**, ya que las demás funciones son inline y deben por lo tanto ser definidas en el archivo de la interfase.

```
/* ------------------------------------------------------------ *\
**  bstream.cpp -- Implement bofstream and bifstream classes    **
** ------------------------------------------------------------ **
**                                                              **
** ------------------------------------------------------------ **
**     Copyright (c) 1993 by Tom Swan. All rights reserved      **
\* ------------------------------------------------------------ */

#include "bstream.h"

void bofstream::writeBytes(const void *p, int len)
```

```
{
  if (!p) return;
  if (len <= 0) return;
  write((char *)p, len);
}

void bifstream::readBytes(void *p, int len)
{
  if (!p) return;
  if (len <= 0) return;
  read((char *)p, len);
}


//------------------------------------------------------------
// Copyright (c) 1993 by Tom Swan. All rights reserved.
// Revision 1.00    Date: 02/12/1993   Time: 07:51 am
```

OBSERVACIONES.

Note que en la clase *bofstream* existen 2 constructores, el primero solo lleva como argumento el nombre del archivo a escribir. Este constructor permite si el archivo ya fue creado, agregar elementos a él, al final de los ya existentes.

El segundo constructor, además recibe como argumento un número entero, él cual solo le permite al compilador definir la llamada de este constructor. Al llamarlo, el archivo se abre, y el *apuntador a archivo* se sitúa al principio del archivo, por lo que si había información ésta se elimina.

Estas nuevas clases son similares a las expuestas en los programas PLCP112.CPP y PLCP113.CPP, solo que ahora se incluye operadores de entrada y salida para los 11 tipos de datos nativos de C++.

En la implementación se usaron las mismas funciones **ReadByte()** y **WriteByte()** definidas en los programas PLCP110.CPP y PLCP111.CPP, se manejan en un módulo separado, pero este puede ser enlazado con los demás módulos del programa.

Al usar las técnicas descritas para manejo de archivos binarios, es responsabilidad del programador la escritura y lectura de los valores en el orden adecuado. Si se escriben los siguientes valores en el orden indicado; double, char y long int, se deberán leer en el mismo orden.

5.- Almacenamiento de objetos en archivos binarios.

Mediante la sobrecarga de los operadores **<<** y **>>** es posible el diseño de clases las cuales cuenten con sus propios operadores de entrada y salida. Analicemos el siguiente código:

```
// PLCP114.h Almacenamiento de Objetos en un archivo
// binario.

#ifndef _POBJETO    // Evita posible redefinición.
#define _POBJETO

#include <iostream.h>
#include "bstream.h"     // Archivo antes mencionado.

class pobjeto {     // Se define una clase llamada
    int x;          // Objetos Persistentes.
    float y;
public:
    pobjeto() : x(0), y(0) {}
    pobjeto(int a, float b) {
        x = a;
        y = b;
    }
    friend ostream   &operator<<(ostream   &a, pobjeto &b);
    friend istream   &operator>>(istream   &a, pobjeto &b);
    friend bofstream &operator<<(bofstream &a, pobjeto &b);
    friend bifstream &operator>>(bifstream &a, pobjeto &b);
};

/*   Se definen operadores friend sobrecargados tanto para
     el manejo del archivo como para ser utilizados con los
     objetos de inserción y de extracción.
*/

inline ostream   &operator<<(ostream   &a, pobjeto &b)
{
    a << "Valor de x = " << b.x << endl;
    a << "Valor de y = " << b.y << endl;
    return a;
}


inline istream   &operator>>(istream   &a, pobjeto &b)
{
    cout << "Introdusca el valor de x (entero): ";
    a >> b.x;
    cout << "Introdusca el valor de y (real)  : ";
    a >> b.y;
    return a;
}


inline bofstream &operator<<(bofstream &a, pobjeto &b)
```

```
{
    a << b.x;
    a << b.y;
    return a;
}


inline bifstream &operator>>(bifstream &a, pobjeto &b)
{
    a >> b.x;
    a >> b.y;
    return a;
}

#endif  // _POBJETO
```

La implementación de la aplicación resulta:

```
// Programa PLCP114.CPP. Lectura escritura de un objeto
// en archivo.
// Aplicación de la clase pobjeto
#include "plcp114.h"
#include <conio.h>
#include <stdlib.h>

void main()
{
    pobjeto C(10, 20.5F), D;
    char narch[15];

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);

    bofstream A(narch);
    if(!A) {
        cerr << "Error en apertura de: " << narch << endl;
        exit(1);
    }
    cout << "Escribiendo al archivo: " << narch << endl;
    cout << C;
    A << C;
    A.close();
    cout << "Oprima una tecla para continuar...";
    getch();
    cout << endl;

    bifstream B(narch);
```

```
    if(!B) {
        cerr << "Error en aoertura de archivo: "
             << narch << endl;
        exit(2);
    }

    B >> D;
    cout << "Contenido del archivo " << narch << endl;
    cout << D;
    B.close();
}
```

OBSERVACIONES.

En este programa se definen 2 objetos del tipo *pobjeto*, el primero se inicializa con 2 valores, el segundo se inicializa a CERO en ambos valores, ésto nos es útil para almacenar la información leída del archivo.

El programa escribe en el archivo cuyo nombre es dado al inicio, el archivo se abre en modo de agregar, al objeto C, el archivo es cerrado, de nuevo es abierto, y ahora se lee la información en él almacenada y se deposita en el objeto D. Posteriormente se muestra en pantalla.

Note que con solo definir 2 funciones *friend*, que reciben como argumento a objetos del tipo *bifstream* y *bofstream*, es suficiente para manejar el objeto en el archivo. Las otras 2 funciones *friend*, a *istream* y *ostream*, solo nos permiten manejar de manera mas fácil objetos de la clase *pobjeto*.

La compilación y enlazado se hace por partes, si utilizamos el compilador en línea de comandos del borland, escribimos:

```
C:\USER\BCC PLCP112 BSTREAM.OBJ
```

Suponemos que el archivo *bstream.cpp* ya fue compilado a .OBJ.

Si se llama varias veces al programa PLCP112.EXE y al mismo archivo, se va almacenando de manera repetida la misma información, ya que como mencionamos, el archivo fue abierto en modo de agregar.

Para leer la información completa se puede utilizar el siguiente programa:

```
// Programa PLCP115.CPP Lectura de varios objetos en archivo
// Aplicación de la clase pobjeto
#include "plcp114.h"
#include <conio.h>
#include <stdlib.h>
```

```
void main()
{
    pobjeto C;
    char narch[15];
    int i = 0;

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);

    bifstream A(narch);
    if(!A) {
        cerr << "Error en apertura de archivo: " << narch <<
endl;
        exit(2);
    }

    cout << "Contenido del archivo " << narch << endl;
    while(!A.eof()) {
        A >> C;
        if(A.eof()) break;
        cout << "Dato: " << i++ << endl << C;
    }
    A.close();
}
```

OBSERVACIONES.

Note que en el lazo, se verifica 2 veces el fin de archivo, cuando se entra al ciclo y cada vez que es leido un objeto. El ciclo termina cuando se encuentra el fin de archivo.

En el siguiente programa se escriben varios objetos al archivo utilizando el operador sobrecargado(**<<**). Posteriormente son leídos mediante un lazo.

```
// Programa PLCP116.CPP Escritura y lectura de varios
// objetos almacenados en el archivo
// La lectura termina cuando se encuentra el EOF.
// Aplicación de la clase pobjeto

#include "plcp114.h"
#include <conio.h>
#include <stdlib.h>


void main()
{
    pobjeto C(10, 20.5F), D(12, 56.78), E(20, 50.55);
```

```
    pobjeto F(1, 1.5),  G(100, 100.10), H(2, 222.22), J;
    char narch[15];
    int i = 0;

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);

    bofstream A(narch);
    if(!A) {
        cerr << "Error en apertura de: " << narch << endl;
        exit(1);
    }
    cout << "Escribiendo al archivo: " << narch << endl;
    cout << C << D << E << F << G << H;
    A << C << D << E << F << G << H;
    A.close();
    cout << "Oprima una tecla para continuar...";
    getch();
    cout << endl;


    bifstream B(narch);
    if(!B) {
        cerr << "Error en aoertura de archivo: " << narch <<
endl;
        exit(2);
    }

    cout << "Contenido del archivo " << narch << endl;
    while(!B.eof()) {
        B >> J;
        if(B.eof()) break;
        cout << "Dato: " << i++ << endl << J;
    }
    B.close();
}
```

OBSERVACIONES.

Se instanciaron e inicializaron 6 objetos del tipo *pobjeto*, el séptimo se inicializó a CERO y nos sirve para almacenar los objetos leídos del archivo.

Los 6 primeros objetos, se muestran en pantalla y luego se almacenan en el archivo, cuyo nombre se dá al inicio del programa.

Posteriormente son leídos y mostrados de nuevo en pantalla.

6.- Posicionamiento del apuntador a archivo.

Cuando se trabaja con Base de Datos, es frecuentemente necesario situar el *apuntador a archivo* en un registro especifico en el archivo.

Existen algunas funciones miembro que nos permiten realizar las siguientes operaciones:

1) Posicionarse en un registro especifico en un archivo de lectura.
2) Posicionarse en un registro especifico en un archivo de escritura.
3) Leer la posición actual del apuntador a archivo.

Veamos cada uno de estos casos.

1) Posicionarse en un registro especifico en un archivo de lectura.

Existen dos funciones miembro sobrecargadas en la clase *istream.*

```
#include <iostream.h>

istream &seekg(streampos p);
istream &seekg(streamoff p, ios::seek_dir);
```

streampos y *streamoff*, son tipos definidos como *long int*.

La primera forma posiciona al *apuntador a archivo* en un byte específico, definido por **p,** (con respecto al inicio del archivo).

La segunda forma posiciona al *apuntador a archivo* en un **offset** con respecto al enumerado ios::seek_dir. seek_dir toma los siguientes valores:

Constante	Valor	Referencia
beg cur end	0 1 2	inicio del archivo posición actual fin del archivo

2) Posicionarse en un registro especifico en un archivo de escritura.

Existen dos funciones miembro sobrecargadas en la clase *ostream.*

```
#include <iostream.h>

ostream &seekp(streampos p);
ostream &seekp(streamoff p, ios::seek_dir);
```

streampos y *streamoff*, son tipos definidos como *long int*.

La primera forma posiciona al *apuntador a archivo* en un byte específico, definido por **p,** (con respecto al inicio del archivo).

La segunda forma posiciona al *apuntador a archivo* en un **offset** con respecto al enumerado ios::seek_dir.

3) Leer la posición actual del apuntador a archivo.

Existen dos versiones de la función:

a) Para la clase istream

```
streampos tellg();
```

b) para la clase ostream

```
streampos tellp();
```

Veamos un ejemplo de uso de algunas de estas funciones.

```
// Programa PLCP117.CPP, Lectura de un objeto del disco
// Definiendo la posición en que se localiza dicho objeto.
// Aplicación de la clase pobjeto
#include "plcp114.h"
#include <conio.h>
#include <stdlib.h>

streampos tam_arch(bifstream &a);

void main()
{
    pobjeto C;
    char narch[15];
    streampos i, j;

    cout << "Dame el nombre del archivo: ";
```

```
    cin.getline(narch, 12);

    bifstream A(narch);
    if(!A) {
        cerr << "Error en apertura de archivo: " << narch <<
endl;
        exit(2);
    }

    j = tam_arch(A);
    cout << "Dispones en el archivo de: " << j << " campos" <<
endl;
    cout << "Posición a leer 1 -> 6: ";
    cin >> i;

    i--;
    A.seekg(i*sizeof(pobjeto));
    cout << "Contenido del archivo " << narch << endl;
    A >> C;
    if(!A.eof())
        cout << "Dato: " << i++ << endl << C;
    else
        cout << "Se detecto fin de archivo" << endl;
    A.close();
}


streampos tam_arch(bifstream &a)
{
    streampos a_pos, len, campo;

    a_pos = a.tellg();
    a.seekg(0L, ios::end);
    len = a.tellg();
    a.seekg(a_pos);
    campo = len/sizeof(pobjeto);
    return campo;
}
```

OBSERVACIONES.

Este programa lee un objeto de un archivo que contenga varios objetos, como por ejemplo, el archivo creado por el programa PLCP113.CPP

- Nos pide el nombre del archivo.

- Mediante la función **tam_arch()** calcula el tamaño del archivo, lee el número de bytes que ocupa el objeto, divide el número de bytes totales del archivo, entre el tamaño del objeto, lo que nos reporta el número de objetos almacenados en el archivo, este valor nos lo regresa y lo mostramos en pantalla.
  Note que la función **tam_arch()**, para calcular el tamaño del archivo, primero almacena la posición actual del *apuntador a archivo*, luego mueve el *apuntador a archivo* al final del archivo, llama a tellg(), el valor regresado lo almacena en la variable *len* y regresa el *apuntador a archivo* a su posición original.
- El valor dado es considerado entre 1,...n, sin embargo el archivo es manejado entre 0,....n - 1, por lo que es necesario hacer el ajuste respectivo.

- El valor resultante es multiplicado por el número de bytes del objeto, con ello el *apuntador a archivo* se posiciona al inicio del objeto.

- Leemos el archivo, verificamos si no hubo un error, si lo hubo, salimos del programa, si no lo hubo, mostramos la información leida.

7.- Leyendo y escribiendo desde el mismo archivo.

Una manera de realizar operaciones de escritura y lectura en el mismo archivo binario sin tener que cerrarlo estando en modo de escritura y luego abrirlo en modo de lectura, es utilizar la clase *fstream* definida en el paquete de Borland C++. Esta clase se hereda en una clase llamada *bfstream* la cual es muy parecida a las clases *bifstream* y *bofstream* definidas con anterioridad. Veamos la definición de la clase:

```
// Copyright (c) 1993 by Tom Swan. All rights reserved
// bfstream.h

#ifndef __BFSTREAM_H
#define __BFSTREAM_H     // Prevent multiple #includes

#include <iostream.h>
#include <fstream.h>

// Binary input and output file stream

class bfstream: public fstream {
  friend bfstream & operator>> (bfstream &, signed char &);
  friend bfstream & operator>> (bfstream &, unsigned char &);
  friend bfstream & operator>> (bfstream &, signed short &);
  friend bfstream &operator>> (bfstream &, unsigned short &);
  friend bfstream & operator>> (bfstream &, signed int &);
  friend bfstream & operator>> (bfstream &, unsigned int &);
  friend bfstream & operator>> (bfstream &, signed long &);
  friend bfstream & operator>> (bfstream &, unsigned long &);
  friend bfstream & operator>> (bfstream &, float &);
  friend bfstream & operator>> (bfstream &, double &);
  friend bfstream & operator>> (bfstream &, long double &);
```

```
  friend bfstream & operator<< (bfstream &, signed char);
  friend bfstream & operator<< (bfstream &, unsigned char);
  friend bfstream & operator<< (bfstream &, signed short);
  friend bfstream & operator<< (bfstream &, unsigned short);
  friend bfstream & operator<< (bfstream &, signed int);
  friend bfstream & operator<< (bfstream &, unsigned int);
  friend bfstream & operator<< (bfstream &, signed long);
  friend bfstream & operator<< (bfstream &, unsigned long);
  friend bfstream & operator<< (bfstream &, float);
  friend bfstream & operator<< (bfstream &, double);
  friend bfstream & operator<< (bfstream &, long double);
public:
  bfstream(const char *fn)
    : fstream(fn, ios::in | ios::out | ios::binary) { }
  void writeBytes(const void *, int);
  void readBytes(void *, int);
};


inline bfstream & operator<< (bfstream &bofs, signed char q)
{
  bofs.writeBytes(&q, sizeof(signed char));
  return bofs;
}


inline bfstream &operator<< (bfstream &bofs, unsigned char q)
{
  bofs.writeBytes(&q, sizeof(unsigned char));
  return bofs;
}


inline bfstream & operator<< (bfstream &bofs, signed short q)
{
  bofs.writeBytes(&q, sizeof(signed short));
  return bofs;
}


inline bfstream &operator<<(bfstream &bofs, unsigned short q)
{
  bofs.writeBytes(&q, sizeof(unsigned short));
  return bofs;
}

inline bfstream & operator<< (bfstream &bofs, signed int q)
{
  bofs.writeBytes(&q, sizeof(signed int));
  return bofs;
}

inline bfstream & operator<< (bfstream &bofs, unsigned int q)
{
  bofs.writeBytes(&q, sizeof(unsigned int));
  return bofs;
}
```

```
inline bfstream & operator<< (bfstream &bofs, signed long q)
{
  bofs.writeBytes(&q, sizeof(signed long));
  return bofs;
}

inline bfstream &operator<< (bfstream &bofs, unsigned long q)
{
  bofs.writeBytes(&q, sizeof(unsigned long));
  return bofs;
}

inline bfstream & operator<< (bfstream &bofs, float q)
{
  bofs.writeBytes(&q, sizeof(float));
  return bofs;
}

inline bfstream & operator<< (bfstream &bofs, double q)
{
  bofs.writeBytes(&q, sizeof(double));
  return bofs;
}

inline bfstream & operator<< (bfstream &bofs, long double q)
{
  bofs.writeBytes(&q, sizeof(long double));
  return bofs;
}



inline bfstream & operator>> (bfstream &bifs, signed char &q)
{
  bifs.readBytes(&q, sizeof(signed char));
  return bifs;
}

inline bfstream &operator>>(bfstream &bifs, unsigned char &q)
{
  bifs.readBytes(&q, sizeof(unsigned char));
  return bifs;
}

inline bfstream &operator>> (bfstream &bifs, signed short &q)
{
  bifs.readBytes(&q, sizeof(signed short));
  return bifs;
}




inline bfstream &operator>>(bfstream &bifs,unsigned short &q)
{
  bifs.readBytes(&q, sizeof(unsigned short));
  return bifs;
```

```
}


inline bfstream & operator>> (bfstream &bifs, signed int &q)
{
  bifs.readBytes(&q, sizeof(signed int));
  return bifs;
}


inline bfstream &operator>> (bfstream &bifs, unsigned int &q)
{
  bifs.readBytes(&q, sizeof(unsigned int));
  return bifs;
}


inline bfstream & operator>> (bfstream &bifs, signed long &q)
{
  bifs.readBytes(&q, sizeof(signed long));
  return bifs;
}


inline bfstream &operator>>(bfstream &bifs, unsigned long &q)
{
  bifs.readBytes(&q, sizeof(unsigned long));
  return bifs;
}


inline bfstream & operator>> (bfstream &bifs, float &q)
{
  bifs.readBytes(&q, sizeof(float));
  return bifs;
}


inline bfstream & operator>> (bfstream &bifs, double &q)
{
  bifs.readBytes(&q, sizeof(double));
  return bifs;
}


inline bfstream & operator>> (bfstream &bifs, long double &q)
{
  bifs.readBytes(&q, sizeof(long double));
  return bifs;
}
```

```
#endif  // __BFSTREAM_H

// Copyright (c) 1993 by Tom Swan. All rights reserved.
// Revision 1.00    Date: 02/12/1993   Time: 10:15 am
```

La implementación resulta:

```
// Bfstream.cpp -- Implement bfstream class
// Copyright (c) 1993 by Tom Swan. All rights reserved

#include "bfstream.h"


void bfstream::writeBytes(const void *p, int len)
{
  if (!p) return;
  if (len <= 0) return;
  write((char *)p, len);
}


void bfstream::readBytes(void *p, int len)
{
  if (!p) return;
  if (len <= 0) return;
  read((char *)p, len);
}


// Copyright (c) 1993 by Tom Swan. All rights reserved.
// Revision 1.00    Date: 02/12/1993   Time: 10:04 am
```

OBSERVACIONES

Si observamos el constructor, vemos que es muy parecido al utilizado con anterioridad, solo que en este caso hacemos un OR con *ios::in e ios::out* y con *ios::binary*, como se muestra en seguida:

```
bfstream(const char *fn)
    : fstream(fn, ios::in | ios::out | ios::binary) { }
```

Lo anterior nos permite leer y escribir en el archivo sin cerrarlo, lo único que hay que hacer es el cambio de escritura a lectura, lo cual se hace con la función miembro de *istream* **seekg()**, se vamos de lectura a escritura, utilizamos la función miembro de *ostream* **seekp()**. En caso de que se llegue al fin de archivo y queremos continuar dentro del archivo es necesario limpiar el EOF, ésto se realiza mediante la función miembro de *ios*, **clear()**.

Si analizamos el código nos damos cuenta que es el mismo, que el de las clases *bifstream* y *bofstream*, con la modificación de que ahora se hace referencia a la clase *bfstream*.

Veamos el siguiente código, él cual es una mezcla de programas anteriores, en él cual se utiliza la clase *bfstream*.

La compilación y enlazado se realiza utilizando la siguiente sentencia:

```
C:\USER\BCC PLCP116 BFSTREAM
```

La interfase resulta:

```
// PLCP118.h Almacenamiento de Objetos
// en un archivo binario.

#ifndef _POBJETO
#define _POBJETO

#include <iostream.h>
#include "bfstream.h"




class pobjeto {
    int x;
    float y;
public:
    pobjeto() : x(0), y(0) {}
    pobjeto(int a, float b) {
        x = a;
        y = b;
    }
    friend ostream   &operator<<(ostream   &a, pobjeto &b);
    friend istream   &operator>>(istream   &a, pobjeto &b);
    friend bfstream &operator<<(bfstream &a, pobjeto &b);
```

```
    friend bfstream &operator>>(bfstream &a, pobjeto &b);
};



inline ostream   &operator<<(ostream   &a, pobjeto &b)
{
    a << "Valor de x = " << b.x << endl;
    a << "Valor de y = " << b.y << endl;
    return a;
}



inline istream   &operator>>(istream   &a, pobjeto &b)
{
    cout << "Introdusca el valor de x (entero): ";
    a >> b.x;
    cout << "Introdusca el valor de y (real)  : ";
    a >> b.y;
    return a;
}



inline bfstream &operator<<(bfstream &a, pobjeto &b)
{
    a << b.x;
    a << b.y;
    return a;
}



inline bfstream &operator>>(bfstream &a, pobjeto &b)
{
    a >> b.x;
    a >> b.y;
    return a;
}

#endif  // _POBJETO
```

La aplicación resulta:

```
// Programa PLCP118.CPP Escritura y lectura de varios
// objetos almacenados en el archivo, usando la clase
// BFSTREAM
```

```
// Aplicación de la clase pobjeto
#include "plcp118.h"
#include <conio.h>
#include <stdlib.h>

streampos tam_arch(bfstream &a);



void main()
{
    pobjeto C(10, 20.5F), D(12, 56.78), E(20, 50.55);
    pobjeto F(1, 1.5),  G(100, 100.10), H(2, 222.22), J;
    char narch[15];
    int i = 0;
    streampos j;

    cout << "Dame el nombre del archivo: ";
    cin.getline(narch, 12);

    bfstream A(narch);
    if(!A) {
        cerr << "Error en apertura de: " << narch << endl;
        exit(1);
    }

    cout << "Escribiendo al archivo: " << narch << endl;
    cout << C << D << E << F << G << H;
    A << C << D << E << F << G << H;

    cout << "Oprima una tecla para continuar...";
    getch();
    cout << endl;


    A.seekg(0);
    cout << "Contenido del archivo " << narch << endl;
    while(!A.eof()) {
        A >> J;
        if(A.eof()) break;
        cout << "Dato: " << i++ << endl << J;
    }
    A.clear();


    j = tam_arch(A);
    cout << "Dispones en el archivo de: "
         << j << " campos" << endl;
```

```
    cout << "Posición a leer 1 -> 6: ";
    cin >> i;
    i--;
    A.seekg(i*sizeof(pobjeto));
    A.clear();

    cout << "Contenido del archivo " << narch << endl;
    A >> J;
    if(!A.eof())
        cout << "Dato: " << i << endl << J;
    else
        cout << "Se detecto fin de archivo" << endl;
    A.close();
}
streampos tam_arch(bfstream &a)
{
    streampos a_pos, len, campo;

    a_pos = a.tellg();
    a.seekg(0L, ios::end);
    len = a.tellg();
    a.seekg(a_pos);
    campo = len/sizeof(pobjeto);
    return campo;
}
```

OBSERVACIONES.

En este programa primero se abre el archivo para escritura / lectura, se muestran 6 objetos los cuales son almacenados en disco posteriormente. Se detiene el programa, continúa al oprimir cualquier tecla, se posiciona el *apuntador a archivo* en el inicio del archivo para lectura y mediante un ciclo *while* se leen todos los elementos en el archivo. A continuación se limpia el EOF para poder continuar manipulando el archivo, se llama a la función **tam_arch()** para mostrar el número de objetos almacenados, y permitirle al usuario seleccionar uno a mostrar, una vez que el usuario teclea el número de objeto a mostrar, se posiciona el *apuntador a archivo* al inicio del objeto seleccionado, se lee y por último se muestra en pantalla.

NOTA: El EOF mencionado no es el (-1), definido para archivos de texto, ya que este carácter se puede encontrar en medio en archivos binarios.

Para salir se cierra el archivo.

Los ejemplos mostrados en esta sección ilustran el manejo de archivos binarios, sin embargo, son simples para resaltar el tema, de manejo de archivos y no perderse

en la maraña de la manipulación de objetos, manejo de menus, altas bajas, modificaciones y estrategias de búsqueda.

En el último ejemplo, tal vez convendría utilizar funciones separadas para abrir el archivo, escribir en el archivo y leer del archivo, sin embargo, no se hizo ésto para mantener la simplicidad.

La utilización de archivos en el campo de la computación es básico para el desarrollo de casi cualquier sistema, por lo que un ingeniero en sistemas debe de manejar perfectamente el tópico de archivos.

Es necesario también hacer notar que todavía no hay un estándar bien definido en las librerías de C++ proporcionadas por los fabricantes de compiladores, ni el naciente estándar ANSI C++, ha definido un estándar, por lo que es necesario consultar la información proporcionada por cada fabricante y tener los debidos cuidados, cuando se transporta código de un compilador a otro.