Objetivos de la práctica:
- Hacer un programa completo que permita poner en práctica los conceptos aprendidos en las sesiones anteriores.

Uso de Ficheros

Todas las estructuras de datos vistas hasta ahora utilizan la memoria principal. Esto hace que los datos se pierdan cuando se termina un programa. Los ficheros son una estructura especial que utiliza la memoria secundaria. Es decir se guardan en memoria no volátil y por ese motivo no se pierde su contenido cuando se apaga el programa.

Para cualquier científico este tipo de datos es de mucha importancia ya que una gran cantidad de medidas o pruebas científicas se guardarán en ficheros que posteriormente serán analizados o procesados en buscando conclusiones. En este sentido mientras anteriormente el resultado de nuestros ejercicios era un solo fichero (*.cpp), ahora tendremos nuestro fichero de programa mas uno o varios ficheros de datos (normalmente *.dat o *.txt).

Tipos de acceso a la información y ficheros de texto

Para acceder a los datos disponibles en un fichero se suele disponer de 2 tipos de acceso: secuencial y directo.

- Acceso secuencial: el acceso a los datos del fichero sólo puede hacerse hacia delante a partir del primer elemento y siempre de uno en uno.
- Acceso directo: el acceso a los datos del fichero se puede hacer directamente en cualquier posición del fichero. Es un acceso similar al utilizado en los arrays usando un índice.

Nosotros solamente trabajaremos con acceso secuencial. Es más, solamente trabajaremos con ficheros de texto. Es decir ficheros que contienen una secuencia de caracteres separados por saltos de línea.

Ficheros lógicos y ficheros físicos

En C++ los ficheros son un tipo más de datos, y un fichero concreto es referenciado a través de una variable de tipo fichero. Es lo que se suele llamar fichero lógico. En C++ existen 2 tipos de datos para declarar ficheros:

ifstream	para declarar ficheros de entrada (de donde vamos a leer)
ofstream	para declarar ficheros de salida (donde vamos a escribir)

Ejemplo de declaración de una variable tipo fichero de salida:

```
ofstream f;      //f variable tipo fichero de salida
```

Para utilizar estos tipos habrá que incluir una cabecera nueva de la siguiente forma:

```
#include <fstream>
```

La variable *f* anterior, para que sea de utilidad debe estar asociada a un fichero “real” (por ejemplo datos.txt). Este fichero es llamado fichero físico.

Operaciones con ficheros

Para relacionar los ficheros lógicos y físicos anteriores necesitamos una operación, que llamamos de apertura del fichero:

```
f.open ("datos.txt");   //ahora ya podemos usar el fichero
```

Así siempre que queramos hacer alguna operación con un fichero, primero lo abrimos, luego trabajamos con él y finalmente lo cerramos:

```
f.close();              //cierre del archivo f
```

Cuando se abre un archivo nos estamos posicionando sobre el primer elemento del archivo. Si el archivo que se ha abierto es para leer datos de él (ifstream), debe existir con anterioridad, o sino esta operación generará un error (no se puede abrir un fichero que no existe para leer su contenido).

Si el fichero es de salida (ofstream) se crea un fichero nuevo, y si ya existía su contenido queda borrado.

El modo de apertura de un archivo se puede modificar de la siguiente forma:

```
f.open ("datos.txt", ios:app);
```

consiguiéndose abrir el fichero para añadir datos en él, de manera que no se borra su contenido y los datos que se añadan se pondrán a partir del último elemento. Si el fichero no existe al intentar abrirlo, genera un error.

Como ya se ha comentado hay varios casos en los que la tratar de abrir un archivo se puede generar un error. Para controlar este error, se deberá siempre utilizar estas sentencias:

```
if  (!f )
        cout << "Error abriendo fichero" << endl;
```

En el caso de que no haya error (else) se pondrá el resto de nuestro programa.

Operaciones entrada y salida de datos

Son exactamente las mismas que se puede realizar con la entrada (cin) y salida (cout) estándar.

Es decir para poner el contenido de una variable x (de cualquier tipo simple) en un fichero:

```
f << x;
```

Mientras que para leer algo de un fichero y almacenarlo en un variable x:

```
f >> x;
```

Ejemplo: Programa que escribe los números del 1 al 10 en el fichero datos.txt

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

int main()
{
	ofstream f; //fichero de salida
	int i;
	f.open("datos.txt");
	if(!f)
		cout << "Error abriendo el fichero" << endl;
	else
	{
		for(i = 1; i <= 10; i++)
			f << i << endl; //escribe el contenido de i y salta de línea
		f.close();
	}
	system (“pause”);
	return 0;
}
```

Ejemplo programa para leer los 10 números enteros y mostrarlos por pantalla:

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

int main()
{
	ifstream f;
	int i, dato;
	f.open("datos.txt");

	if(!f)
		cout << "Error abriendo el fichero" << endl;
	else
	{
		for(i = 1; i <= 10; i++)
		{
			f >> dato;
			cout << dato << endl;
		}
		f.close();
	}
	system (“pause”);
	return 0;
}
```

Sin embargo, lo normal es que no sepamos cuantos elementos vamos a leer, sino que queremos leer hasta que lleguemos al final del fichero. Para ello se puede utilizar un bucle while de la siguiente forma:

```
	while(f >> dato)
		cout << dato << endl;
```

De esta forma, la instrucción anterior leerá, mientras sea posible, todos los datos del fichero, quedando el ejemplo de la siguiente forma:

```
#include<iostream>
#include<stdlib>
#include<fstream>
int main()
{
      ifstream f;
      int dato;

      f.open("datos.txt");
      if(!f)
            cout << "Error abriendo el fichero" << endl;
      else
      {
            while(f >> dato)
                  cout << dato << endl;
            f.close();
      }
      system (“pause”);
      return 0;
}
```

Si queremos leer el fichero carácter a carácter, lo más normal será utilizar el método **get( )**, sin embargo éste no devuelve cierto o falso para saber si se ha podido leer con éxito, puesto que tiene que devolver el carácter que ha leído. La forma de leer un fichero con get() será por tanto ligeramente distinta a la que hemos visto.

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

int main()
{
      ifstream f;
      char dato;

      f.open("datos.txt");
      if(!f)
            cout << "Error abriendo el fichero" << endl;
      else
      {
            dato = f.get();
            while(! f.eof())
            {
                  cout << dato << endl;
                  dato = f.get();
            }
            f.close();
      }
      system (“pause”);
      return 0;
}
```

El método eof() es cierto si el dato que hemos leído era un final de fichero y falso en caso contrario. Por esta razón hay que leer primero el carácter y después comprobar si hemos llegado a final de fichero.

Lectura de estructuras

Cuando queremos leer datos simples de un fichero, se puede realizar fácilmente introduciendo la lectura dentro de la condición de un bucle while. Sin embargo, para leer una estructura se deben leer primero cada uno de sus campos antes de considerar la lectura correcta, por lo que para poder efectuar la lectura en la condición del while, habrá que definir una función que realice dicha lectura y devuelva **true** si se ha podido realizar sin errores y **false** en caso contrario.

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

struct Telefono
{
 string nombre;
 string telefono;
};

bool F_IntroTel(ifstream & f, Telefono & tel); //prototipo

int main(void)
{
   Telefono tel;
   ifstream guia;

   guia.open("guia.dat");
   if(!guia)
       cout << "Error abriendo el fichero" << endl;
   else
   {
       while(F_IntroTel(guia, tel))
       {
             cout << tel.nombre << endl;
             cout << tel.telefono << endl;
             cout << endl;
       }
       guia.close();
   }
   system (“pause”);
   return 0;
 }

/*************************************************************/
/*F_IntroTel lee una estructura compuesta por un nombre y */
/* un telefono */
/*f fichero del cual quiero leer */
/*tel estructura donde voy a colocar lo leido */
/*************************************************************/

bool F_IntroTel(ifstream & f, Telefono & tel)
{
 getline(f, tel.nombre);
 getline(f, tel.telefono);
 return !(f.eof());
}
```

Para terminar conviene observar que los ficheros se han de pasar siempre como parámetros por referencia, no importa si los vamos a modificar o no. Además para este caso hemos usado la función **getline** , que nos permite leer una línea de fichero completa. Es evidente que para que esto funciones el fichero de datos debe incluir en cada línea un dato distinto (en una línea un nombre, en la siguiente el teléfono asociado y así sucesivamente).

Nota importante: el fichero que se lee debe tener un último salto de línea después del último dato o no se leerá la última estructura.

Resumen:

1. No olvidar incluir la cabecera: `#include <fstream>`
2. Solamente se practicará con ficheros secuenciales de tipo texto:

- ficheros que guardan únicamente caracteres
- el acceso a la información es secuencial: para llegar a un lugar de fichero hay que pasar por todos los datos anteriores

3. Los ficheros se pueden abrir para leer su contenido o para escribir en ellos. Nunca leeremos y escribiremos simultáneamente (nota: *fichero* es solamente el nombre de la variable):

Tipo para definición de un objeto fichero de escritura: `ofstream fichero;`
Tipo para definición de un objeto fichero de lectura: `ifstream fichero;`
Apertura del fichero:

```
fichero.open (“nombre_fichero”);
```

Cierre del fichero:

```
fichero.close ();
```

Nota: por defecto el Dev C++ deja el fichero creado en el mismo directorio donde está el compilador:

```
fichero.open(“c:\\tmp\\hola.txt”);  // deja en fichero hola.txt en el directorio
                                    //c:\tmp (notad la doble \)
```

4. Escritura y lectura. La lectura y escritura de información de se realiza a través los operadores << y >> (de forma equivalente al uso de cin y cout):

```
fichero << dato;  //donde dato es cualquier tipo de dato convertido a carácter. Escritura en fichero
fichero >> dato;  // lectura desde fichero
```

5. Fin de fichero. Para leer un fichero hasta su fin se empleará:

```
while (fichero >> dato)  // esta condición proporcionará FALSE cuando se llegue al final.
```

6. Estructuras. No se puede leer ni almacenar una estructura directamente en un fichero, sino que debe procederse por separado con cada uno de sus campos:

ejemplo:

```
struct complejo{
            int real;
            int imag;
       };
```

```
Bien: salidafich << complejo.real << “ “ << complejo.imag;
```

```
Mal: salidafich << complejo
```

PROBLEMAS

0.- Dado un fichero que contiene nombre de alumnos y su nota

Juan Serrano 5
Luis Marqués 7.5
Pedro López 3.5
…

Realiza un programa en C++ que lea los nombres de los alumnos y sus notas del fichero y los visualice por pantalla.

1ª PARTE

Supongamos el siguiente formato de fichero que sirve para almacenar información sobre torneos de ajedrez. El fichero será de tipo texto plano y en él se almacena la siguiente información:

- En la primera línea, el número de participantes (n).
- En la segunda línea, el número de jornadas transcurridas (ya jugadas, es decir, con resultados).
- En las siguientes n líneas, los nombres de los n participantes.
- En las siguientes líneas, se almacenarán los emparejamientos con los resultados. Hay n/2 partidas por jornada (y n-1 jornadas). Para cada emparejamiento/partida se utilizarán números naturales que corresponderán a cada participante para indicar el emparejamiento. En cada línea aparecerán 2 números naturales separados por un espacio en blanco, correspondientes a los participantes que se enfrentan, y a continuación (tras un espacio en blanco) dos números separados por un espacio que indican el resultado del enfrentamiento (las posibles variantes son 0.5 0.5 (empate), 1 0 (victoria de las blancas), 0 1 (victoria de las negras) y -1 -1 (el partido aun no se ha jugado)).

Por ejemplo, un fichero de un torneo para 4 participantes puede ser el siguiente:

```
4
2
Andres Lopez Cuesta
Juan Martinez Fernandez
Pablo Casanova Perez
Julian Andujar Carrasco
1 4 0.5 0.5
2 3 1 0
1 3 0 1
2 4 0 1
1 2 -1 -1
3 4 -1 -1
```

En este ejemplo se puede observar que hay 4 participantes (3 jornadas de 2 enfrentamientos cada una), y se han introducido los resultados de las 2 primeras jornadas. Por ejemplo, en la primera jornada se han enfrentado Andrés López Cuesta con Julián Andujar Carrasco, con un resultado de empate, y Juan Martínez Fernández con Pablo Casanova Pérez, con resultado de victoria del primero.

1. Hacer una función que lea el fichero del torneo y lo cargue en memoria. Para ello, utiliza la siguiente estructura definida para el torneo y para sus participantes. Vamos a suponer que nunca habrán más de 10 participantes, ni más de 45 jornadas.

```
const int TAM = 10
struct enfrentamiento{
      int jug1 , jug2; //para guardar quien juega contra quien
      float res1 , res2; //para guardar el resultado
};

struct torneo
{
  int n; //numero de participantes
  int j;  //numero de jornadas jugadas
  string nombres[TAM]; // nombre de los participantes
  enfrentamiento enfrenta[(TAM-1)*(TAM/2)]; // Para 10 jugadores, hay un máx de 45 enfrentam.
};


void LeerFichero( ifstream & f , torneo & t);
```

Nota: Para poder usar la función de apertura de un fichero pasándole como argumento una variable string se debe escribir utilizando c_str(). Ejemplo:

```
string nombre;
fichero.open( nombre.c_str() );
```

Llamar a la función LeerFichero desde el programa principal.

2. Haz una función que muestre por pantalla toda la información del torneo y llámala desde el programa principal.

   ```
   void MostrarTorneo ( torneo t);
   ```

3. Haz una función que escriba en un fichero “de tipo torneo” toda la información almacenada en memoria.
4. Haz una función que, dado un nombre de persona que le pasaremos como argumento, busque en el torneo todas las partidas que ha jugado. Se deberá además mostrar contra quién ha jugado la partida e indicar si empató, ganó o perdió.
5. Haz una función que cree un nuevo torneo. Para ello, deberá pedir el número de participantes y comprobar que es par y menor de 10, a continuación los nombres de los participantes y finalmente generará de forma automática los distintos enfrentamientos de las distintas jornadas, rellenando la estructura y poniendo por defecto a -1 los resultados de los enfrentamientos (número de jornadas jugadas igual a 0).

   ```
   void crear_torneo(torneo & t);
   ```

Algoritmo de cálculo de emparejamientos

Para determinar los emparejamientos en cada jornada, tomamos el requisito de que todos los participantes (un número par) se enfrentan entre sí, lo que supone que ante un número de n participantes se generarán n-1 jornadas, con n/2 enfrentamientos en cada jornada, de manera que no queden participantes sin emparejar en una misma jornada, y no se repitan los emparejamientos en diferentes jornadas.

Con todo, el algoritmo que se puede utilizar es el siguiente:

En la ronda k (1 <= k < n) el equipo i<n jugará con el j<n si (i+j) mod (n-1) = k mod (n-1) donde j es distinto de i. Cuando i=j, es decir que 2i mod (n-1) = k mod (n-1), el equipo i juega contra el n.

2ª PARTE

En la segunda parte de esta práctica vamos a desarrollar una aplicación que, usando las funciones realizadas en la primera parte de la práctica y creando nuevas funciones, nos permita gestionar un torneo de ajedrez.

Para las competiciones de ajedrez que vamos a gestionar podemos decir que existen una serie de participantes y un conjunto de enfrentamientos que terminan con un resultado particular. Podemos considerar que los enfrentamientos se realizan en rondas o jornadas, y que tras cada jornada existe una clasificación de los competidores atendiendo a los resultados obtenidos hasta ese momento.

La aplicación que se desarrolle deberá ser capaz de almacenar información sobre participantes y resultados en la competición, así como de presentar clasificaciones de los participantes según el estado de la competición y generar los enfrentamientos para cada jornada.

DESARROLLO

Para simplificar el desarrollo de la aplicación, vamos a determinar una serie de restricciones que se deben satisfacer en todo momento:

- El número de participantes en la competición será par, y no podrá exceder de 10.
- Cada participante deberá enfrentarse con todos los demás participantes. Así, por ejemplo, si hubiesen 8 participantes, la competición constaría de 7 jornadas en las que cada participante se enfrenta con todos los demás.
- Los resultados de un enfrentamiento pueden ser victoria de un participante frente a otro, o empate entre los dos participantes, para lo cual se asignará una puntuación de 0 puntos para una derrota, 1 punto para una victoria y 0,5 puntos para un empate (para los dos participantes).
- Si no se ha realizado todavía el enfrentamiento, se pondrá -1 a cada participante como resultado provisional.
- De la clasificación de los participantes solo nos va a interesar saber quien es el máximo ganador o quién es el participante con menor puntuación, en un momento dado. Por tanto, esto se calcula como la suma de los puntos obtenidos por los participantes en sus enfrentamientos hasta el momento en que deseemos saber la clasificación, de modo que el jugador que más puntos tenga será el primer clasificado, y el jugador que menos puntos tenga será el último clasificado. No existe ningún criterio de desempate en caso de que varios participantes coincidan en el número de puntos obtenidos.

- Toda la información de participantes, jornadas y resultados se debe almacenar en un fichero de texto, pero no quién es el mejor o peor clasificado, que se deberán calcular en cada momento cuando lo solicite el usuario a la aplicación.

Funcionamiento

El programa debe poder trabajar con un torneo ya creado (cuya información se almacena en un fichero y se carga en el programa) o bien crear un torneo nuevo. Cada torneo se guarda en un fichero diferente.

Cuando se cree un nuevo torneo, se deberán introducir el número de participantes (un número par) y los nombres de todos los participantes del torneo, tras lo cual, se generarán todas las jornadas con todos los enfrentamientos.

El programa debe permitir introducir resultados de cada jornada, de forma que no se podrán introducir resultados de una jornada si no se han llenado los resultados de la jornada anterior.

El programa también permitirá visualizar los resultados de cualquier jornada, así como quién es el primero y el último en la clasificación al terminar una jornada concreta, así como, quién es el primero y el último en ese momento del campeonato.

Una de las opciones que el programa debe permitir almacenar en un fichero el estado actual del torneo, lo que implica guardar el número de participantes, sus nombres, los emparejamientos de cada ronda y los resultados de los encuentros en cada jornada. Este fichero será el que se utilice para cargar un torneo y trabajar sobre él.

Opciones

El programa funcionará a través de un menú con las siguientes opciones:

1. Cargar torneo.
2. Crear un torneo nuevo.
3. Introducir resultados de jornada.
4. Ver resultados de una jornada.
5. Ver quién es el primer clasificado y el último en una jornada.
6. Ver quién es el primer clasificado y el último en el momento actual del campeonato.
7. Ver todos los resultados del torneo (dando los nombres de los competidores y el resultado).
8. Guardar torneo.
9. Salir del programa.

Estas opciones se implementarán a través de un menú, que será lo primero que se muestre al ejecutar la aplicación, y será de tipo cíclico, es decir, tras cada operación se volverá al menú principal.

El programa finalizará cuando el usuario elija la opción de Salir del programa.

Nota: Hay que tener en cuenta que se deberá controlar en cada momento las opciones tales como: “salir del programa sin haber guardado”, “crear un torneo nuevo cuando ya hay cargado otro torneo”, “no se puede visualizar resultados ni introducirlos si no estamos en un torneo”, etc.