PROGRAMACION ESTRUCTURADA BASICA

UNIDAD 3. ARCHIVOS SECUENCIALES

INDICE

UNIDAD 3 – ARCHIVOS SECUENCIALES

OBJETIVOS: *Realizar programas que puedan almacenar datos y resultados en una memoria no volátil a fin de respaldar la información generada permitiendo construir programas que no pierdan los datos cargados al finalizar su ejecución.*

1. Que es un archivo

Los datos con los que se han trabajado hasta el momento han residido en la memoria principal. Sin embargo, las grandes cantidades de datos se almacenan normalmente en un dispositivo de memoria secundaria. Estas colecciones de datos se conocen como archivos (en ingles FILE).

Hay dos tipos de archivos, archivos de texto y archivos binarios. Un archivo de texto es una secuencia de caracteres organizadas en líneas terminadas por un carácter de nueva línea. En estos archivos se pueden almacenar textos en general como ser canciones, un cuento, etc. Los archivos de texto se caracterizan por ser planos, es decir, todas las letras tienen el mismo formato y no hay palabras subrayadas, en negrita, o letras de distinto tamaño o ancho. Los archivos de texto también pueden contener información estructurada (organizada) de determinada forma por ejemplo con una separación entre valores que puede ser coma o punto y coma (archivos csv) que pueden ser interpretados por planillas de cálculo. Otros formatos de texto muy utilizados en sistemas son los archivos xml, o JSON que permiten especificar claramente valores de entidades y objetos facilitando compartir datos entre distintos sistemas.

Un archivo binario es una secuencia de bytes que tienen una correspondencia directa con lo almacenado en memoria. Así que no tendrá lugar ninguna traducción de caracteres. Ejemplos de estos archivos son Fotografías, imágenes, texto con formatos, archivos ejecutables (aplicaciones), etc. Los archivos binarios en general se manejan almacenando bloques de información del mismo tipo. Un archivo binario es un conjunto de datos estructurados en una colección de entidades elementales o básicas denominadas registros que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajos denominadas campos. El formato del registro de un archivo binario estará dado por una estructura en el lenguaje C.

En el **Lenguaje C**, un archivo es un concepto lógico que puede aplicarse a muchas cosas desde archivos de disco hasta terminales o una impresora. Se asocia una sentencia con un archivo específico realizando una operación de apertura. Una vez que el archivo está abierto, la información puede ser intercambiada entre este y el programa.

Se puede conseguir la entrada y la salida de datos a un archivo a través del uso de la biblioteca de funciones; el **Lenguaje C** no tiene palabras claves que realicen las operaciones de E/S. La siguiente tabla da un breve resumen de las funciones que se pueden utilizar. Se debe incluir la biblioteca **<stdio.h >**. Observe que la mayoría de las funciones comienzan con la letra “F”.

fopen ()	Abre un archivo.
fclose ()	Cierra un archivo.
fread ()	Lee un registro del archivo.
fwrite ()	Guarda un registro en el archivo.
feof()	Devuelve un número distinto de 0 si se llega al final del archivo.

2. Como se accede a un archivo

Desde el programa lo que se busca es poder grabar y recuperar información de un dispositivo externo conectado a la computadora que puede ser un disco rígido, un pen drive, una tarjeta de memoria, etc. Es decir, el acceso se hará sobre un componente físico (hardware) que puede tener distinta tecnología (óptica, magnética, etc) y por lo tanto la forma de acceso será diferente.

Como el lenguaje C es de alto nivel no es necesario preocuparse por la forma específica de acceso ya que existe una "abstracción" del lenguaje C que resuelve ese acceso mediante lo que se conoce como flujos o corrientes (stream).

No importa el dispositivo físico donde se encuentre el archivo ya que todos los dispositivos se pueden manejar de la misma forma. Se crea un intermediario entre el programa y el hardware y es finalmente el sistema operativo mediante el uso de drivers (controladores) el que se encarga de acceder a la información a nivel físico.

Para cada archivo que se quiera utilizar se creará un flujo que comunica el programa con el dispositivo externo.

Si existen varias formas para trabajar con archivos la forma de trabajo será mediante el uso de un buffer intermedio. El buffer es un área en la memoria donde se guarda información temporalmente para luego ser llevada al dispositivo externo. Es decir que el programa no graba directamente en el hardware, sino que graba en esa memoria intermedia y luego los datos se toman de esa memoria y a medida que se graban en el dispositivo externo se van borrando de dicha área de memoria.

El buffer se utiliza debido a que las velocidades de acceso son muy diferentes según el hardware conectado. Además, el trabajar directamente en un espacio de memoria es mucho más rápido haciendo que nuestro programa no tenga que esperar los tiempos de acceso y pueda seguir con otros procesos mientras se van grabando los datos físicamente.

Desde el programa se va grabando en el buffer y cuando el buffer se llena o llega a determinada capacidad un proceso se encarga de tomar esos datos y volcarlos al dispositivo físico. Pero este procedimiento es transparente para el programador.

Sin embargo para que se pueda lleva a cabo dicho procedimiento para cada archivo que se quiera utilizar desde el programa se deben guardar bastantes datos. Para el buffer se debe saber, en que posición de la memoria se encuentra, que tamaño tiene y hasta donde está lleno. Ver Figura 1.

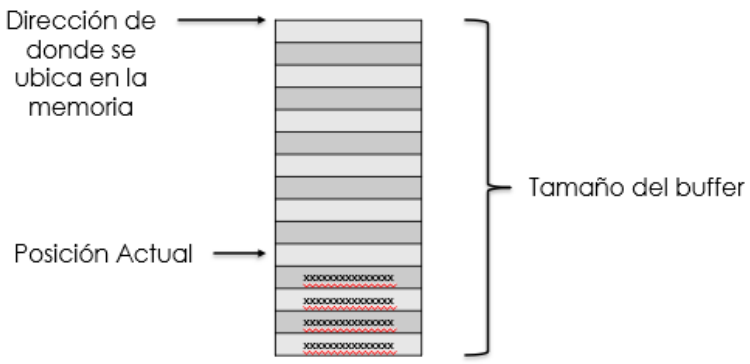


Figura 1. Componentes necesarios para manejar un buffer

Pero ademas de la información del buffer dicho proceso necesita más datos para poder acceder al arhivo. Todos los datos necesarios para el manejo de un archivo están definidos con una estructura definida en la biblioteca stdio.h denominada FILE.

Esta estructura contiene entre otras cosas (ver figura 2):

- La dirección de memoria del buffer
- El tamaño de buffer
- Hasta donde está lleno el buffer
- El desciptor del archivo (código interno con el cual el sistema operativo identifica los archivos abiertos)
- Indicador de posición dentro del archivo físico (desde que punto se puede comenzar a leer o escribir)
- Una serie de señales, banderas o flags, que dan información del resultado de algunas operaciones.

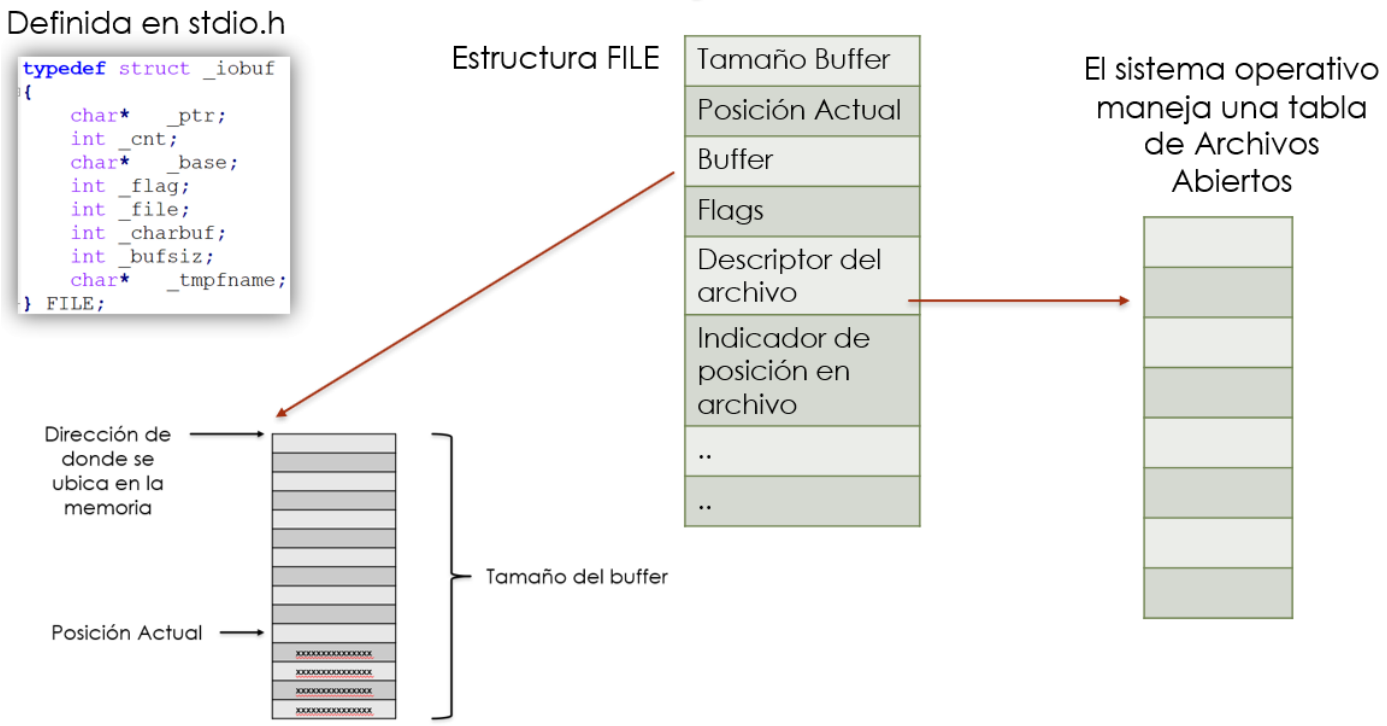


Figura 2: Estructura necesaria para guardar los datos para manejar el flujo de archivos

Todos estos componentes serán administrados de forma transparente al programador ya que solo utilizará algunas funciones que internamente utilizan esta estructura para administrar los archivos.

3. Apertura de un archivo (función fopen ())

Antes de utilizar un Archivo en un programa, ya sea para escribir datos en él y/o para leer datos de él, el Archivo debe ser abierto, con lo que se establece un canal de comunicación entre el programa y el Archivo. Y la función para abrir un Archivo se denomina **fopen ()** y su prototipo es:

FILE * **fopen** (char nombreArchivo [ ] , char modoApertura [ ]);

El fopen se encargará de crear el flujo, decirle al sistema operativo que abra y prepare para trabajar el archivo y de generar toda la estructura necesaria en la memoria para administrar el buffer y la información del archivo como se vio anteriormente. Dicha función retorna una dirección de memoria de donde se encuentra la estructura para el manejo de archivo.

FILE es el nombre de la estructura definida en stdio.h pero la función retorna un FILE *. El asterisco sígnica que no retorna una variable del tipo FILE sino que retorna la dirección de memoria donde creó dicha variable. Es decir retorna un puntero a donde se encuentra la variable FILE. En el lenguaje C una variable que guarda un dirección de memoria es un puntero, ya que apunta (o referencia) a otra variable.

Todas las operaciones sobre el Archivo se realizarán a través de ese puntero. Para indicar el Archivo que se desea abrir se usa la cadena de caracteres nombreArchivo la cual puede incluir, además del nombre del Archivo, la ruta completa donde está ubicado, es decir la unidad de almacenamiento y la carpeta. No olvidar que la barra invertida (\) de la ruta debe escribirse dos veces (\\), porque si se escribe sólo una vez es interpretada como secuencia de escape. Sino se coloca la ruta el archivo se busca en la misma carpeta donde se encuentra el ejecutable. Para que el programa pueda ser ejecutado en distintas computadoras sin errores se recomienda no poner una ruta fija.

Con la cadena modoApertura se indicará si se va a usar el archivo en modo texto o en modo binario, además servirá para especificar si se va a leer del archivo, se va a escribir en él o ambas cosas.

En esta unidad se trabajara con archivos secuenciales es decir que solo se puede abrir un archivo para una cosa a la vez y no se puede volver atrás en el archivo. Los modos disponibles para el uso de archivos secuenciales son los siguientes:

- “r” , “rt” , “rb“ se abre para lectura sino existe error
- “w”, “wt” “ wb” se crea para escritura, si existe lo destruye (ojo no avisa)
- “a”, “at” , “ab” modo de escritura donde se abre para agregar a continuación del último dato que está en archivo, sino existe lo crea

La t corresponde a archivos de texto y la b a archivos binarios. Sino se especifica por defecto se refiere a archivos de texto, es decir que r y rt , w y wt, a y at son lo mismo. En esta materia solo utilizaremos los archivos de texto para exportar datos y por lo tanto se utilizará el modo w o wt. En cambio, para los archivos binario sí se utilizarán los tres modos mencionados rb, wb y ab. La Siguiente tabla resume los modos a utilizar en esta unidad:

Modo	**Acción**	**Archivo ya existe**	**Archivo no existe**	**Lectura**	**Escritura**	**Posición en el archivo**
rb	Lectura Binaria	Correcto	*** Error ***	Correcto	*** Error ***	Principio
wb	Escritura Binaria	Borra contenido	Se crea	*** Error ***	Correcto	Principio
ab	Añadir Binario	Correcto	Se crea	*** Error ***	Correcto	Final
wt	Escritura de texto	Borra contenido	Se crea	*** Error ***	Correcto	Principio

En la tabla se observa que la apertura de un Archivo con el modo “wb” puede ser peligrosa, ya que si el Archivo existe se perderán todos sus datos. Aunque hay situaciones en las que es esa acción precisamente la que se quiere realizar, destruir todos los datos previos del Archivo. Por otra parte, debe tenerse en cuenta que si se utiliza el modo de apertura “ab” los datos nuevos que se escriban en el Archivo nunca sobrescriben otros datos, sino que siempre se añaden al final**.**

Si la función fopen () tiene éxito, es decir abre el Archivo sin ningún problema, devolverá el puntero al Archivo. Por el contrario, si se ha producido algún error al intentar la apertura devuelve el valor NULL. Por tanto, después de ejecutar la función fopen (), el programa deberá comprobar siempre si el puntero devuelto vale NULL.

Ejemplos:

```
/* Especificando solo el nombre del archivo lo busca en la misma carpeta que el
ejecutable */
      FILE *fp;
      fp = fopen("C:\\archivos\\ventas.dat", "rb");
      if(fp == NULL)
      {
            printf ("ERROR al abrir el archivo de ventas");
            getch();
            exit(1);
      }

     //Especificando unidad, carpeta y nombre de Archivo
      FILE *fp;
      fp = fopen("C:\\archivos\\ventas.dat", "rb");
      if(fp == NULL)
      {
            printf ("ERROR al abrir el archivo de ventas");
            getch();
            exit(1);
      }
```

4. Como Cerrar un Archivo. Cierre de un archivo (función fclose ())

Cuando un programa deja de necesitar un Archivo, éste debe cerrarse, cortándose por tanto el canal de comunicación entre el programa y el Archivo. Para cerrar un Archivo se utiliza la función **fclose ()**.

La función **fclose ()** cierra un Archivo que fue abierto mediante una llamada a **fopen ()**. Escribe toda la información que todavía se encuentre en el buffer en el disco y realiza un cierre formal del archivo a nivel del sistema operativo. Libera toda la estructura creada en memoria para el manejo del flujo del archivo. Un error en el cierre de un Archivo puede generar todo tipo de problemas, incluyendo la pérdida de datos, destrucción de archivos y posibles errores intermitentes en el programa, cuya sintaxis es:

```
fclose (pf);
```

Donde "pf" es el puntero al archivo devuelto por la llamada a **fopen ()**. Generalmente, esta función solo falla cuando un disco se ha retirado antes de tiempo o cuando no queda espacio libre en el mismo.

5. Como leer un registro de un archivo. Función de Entrada (función fread ())

La función **fread ()** trabaja con registros de longitud constante. Es capaz de leer desde un Archivo, uno o varios registros de la misma longitud y a partir de una dirección de memoria determinada. Se debe asegurar de que haya espacio suficiente para contener la información leída. La sintaxis de esta función es:

En lenguaje C:

```
fread( <Dir_Variable> , < n°_bytes>,  <cantidadDeRegistros>,  <variable_file>);
```

La función **fread ()** significa que se leerá del Archivo que apunta la <variable_file> el número de bytes <nº_bytes> tantas veces como indique <cantidadDeRegistros >, dejando dichos bytes grabados en memoria a partir de la dirección <Dir_Variable>. Esta dirección será la de una variable con el tamaño suficiente para guardar los datos leídos. El número de bytes que se leen será por tanto el resultado del producto:

<nº_bytes> * < cantidadDeRegistros >

Ejemplos:

```
int num;
fread(&num, sizeof(int), 1, fp);
//Se leen 4*1=4 bytes del archivo, que caben en num.
float Notas[5];
fread(Notas, sizeof(float), 5, fp);
//Se leen 4*5=20 bytes del fichero, que caben en Notas
```

En el Ejemplo N°1, en el segundo caso se leerán 20 bytes, ya que se leerán 5 veces el número de bytes indicado por sizeof(float), o sea 4*5 = 20. Por tanto, se leen 5 números float desde el archivo y se guardan en el array Notas, cuyo tamaño es de 20 bytes.

6. Como escribir un registro de un archivo. Función de Salida *(función fwrite())*

La función **fwrite()** permite escribir datos en un archivo como una sucesión de bytes. La sintaxis de esta función es:

```
fwrite( <Dir_Variable> , < n°_bytes>,  <cantidadDeRegistros>,  <variable_file>);
```

A partir de la dirección de memoria <Dir_Variable> se leerán tantos bytes como se indique en <nº_bytes> tantas veces como se especifique en <cantidadDeRegistros>y se escribirán en el Archivo que apunta la < variable_file >. El número de bytes escritos será por tanto el resultado del producto: <nº_bytes> * <cantidadDeRegistros>.

Ejemplos:

```
int num=1067;
fwrite(&num, sizeof(int), 1, fp);
//Graba 4 bytes en el archivo, desde la dirección num, por tanto el valor 1067
queda escrito en el archivo.
float Notas[5]={5.5, 7.25, 8.5, 4.75, 9.5};
fwrite(Notas, sizeof(float), 5, fp);
//Se escriben 4*5=20 bytes en el archivo, desde la dirección Notas, quedando las
5 notas grabadas en el archivo.
```

La función **fwrite()** devuelve el número de veces que ha escrito el <nº_bytes>, que no siempre coincide con <cantidadDeRegistros>, ya que puede producirse algún error. Por tanto, si el valor que devuelve **fwrite()** no coincide con <cantidadDeRegistros> es que ha ocurrido un error.

Ejemplo . Escribir un número real en un Archivo. Leerlo en otra variable real.

```
FILE *fp;
float var;
float num = 12.23;
if ((fp = fopen("prueba.dat", "wb")) == NULL )
{
   printf("No se puede abrir.\n");
   getch(); exit(1);
}
fwrite( &num, sizeof(float), 1, fp);
fclose(fp);
if ((fp = fopen("prueba.dat", "rb")) == NULL )
{
   printf("No se puede abrir.\n");
   getch(); exit(1);
}
```

```
      fread (&var, sizeof(float), 1, fp );
      fclose(fp);
```

Uno de los usos más comunes de **fread()** y **fwrite()** es el manejo de Archivo en forma de conjunto de registros, donde cada registro está dividido en campos. Para ello en el programa debe definirse un tipo de estructura de datos con el mismo formato que el registro del Archivo, con el objeto de leer y escribir registros completos con las funciones **fread()** y **fwrite()** respectivamente, en lugar de leer y escribir campos concretos.

Ejemplo. Teniendo un vector de estructuras ya cargado con datos, deberá grabarse en un Archivo y después dejarlo en otro vector de estructuras.

```
struct datos
{
   char nombre[20];
   char apellido[40];
};

int main ()
{
      FILE *fp;
      int i;
      struct datos lista1[30], lista2[30];
      //Suponemos que lista1 ya contiene datos.

      if ( (fp = fopen("fich.dat", "wb")) == NULL )
      {
                  printf("No se puede abrir.\n");
                  getch(); exit(1);
      }
      //Escribe 30 registros desde el array lista1.
      for ( i = 0; i < 30; i++)
                  fwrite(&lista1[i],sizeof(struct datos),1,fp)
      fclose(fp);
      if ( (fp = fopen("fich.dat", "rb")) == NULL )
      {
                  printf("No se puede abrir.\n");
                  getch(); exit(1);
      }
      //Lee 30 registros, dejándolos en el array lista2.
      for ( i = 0; i < 30; i++)
                  fread(&lista2[i], sizeof(struct datos),1,fp)
      fclose(fp);
      return 0;
}
```

7. Como leer un archivo sin conocer la cantidad de registros. Función feof().

Cada vez que se lee de un archivo con la función fread, los datos son transferidos a la memoria y el indicador de posición del archivo queda listo para la lectura del siguiente registros justo a continuación del último dato leído. Si se intenta leer y se encuentra que ya no hay datos en el archivo, entonces dentro de la estructura FILE se activa una bandera (flag) que indica que se llegó al final del archivo y por lo tanto no se leyeron datos. Para consultar el estado de dicha bandera se utiliza la siguiente función:

```
      int feof(FILE *);
```

Su prototipo se encuentra en <stdio.h>. Devuelve un número distinto de 0 si se ha alcanzado el final del archivo, si aún hay datos devuelve un 0.

Cuando no se conoce la cantidad de registros que posee un Archivo, se puede utilizar la función **feof()**, la cual permitirá leer la información hasta el fin de Archivo.

Esta función debe utilizarse **luego** de hacer una lectura sobre el archivo con fread.

Ejemplo:

```
struct datos
{
   char nombre[20];
   char apellido[40];
};

int main ( )
{
      FILE *fp;
      int i;
      struct datos lista[30];

      // Se supone que el archivo no va a contener más de 30 registros.

      if ( (fp = fopen("fich.dat", "rb")) == NULL )
      {
         printf("No se puede abrir.\n");
         getch(); exit(1);
      }
      //Lee los registros del archivo hasta el final del mismo.
      i = 0;
      fread(&lista[i], sizeof(struct datos),1,fp)
      while (  !feof (fp)  )
      {
         i++;
         fread(&lista[i], sizeof(struct datos),1,fp)
      }
      fclose(fp);
      return 0;
}
```

8. Archivos relacionados

Al trabajar con archivos muchas veces la información esta separada en distintos archivos y relacionadas por algún campo clave, por ejemplo, se puede tener un archivo con productos que tenga código, precio y descripción de cada producto, en ese caso el código es un campo clave que identifica unívocamente a cada producto. Por otro lado, si se registran las ventas de la empresa cuando se refiera a que producto se vendió el archivo de ventas solamente va a contener el código de producto vendido ya que el resto de la información como el precio y la descripción se recupera del archivo relacionado. Al trabajar con archivos relacionados en general se supone que el archivo de ventas no va a contener códigos de productos que no existan ya que la integridad de los datos fue chequeada al momento de creación de los archivos. Sin embargo, como cada archivo se puede manejar por separado en algún caso puede ocurrir que se borre algún registro y se pierda la relación haciendo imposible recuperar los datos. Estos problemas en sistemas comerciales se solucionan usando entornos que manejan los datos de forma integral (sistemas gestores de base de datos) y hacen que no se permita borrar un registro si se está utilizando en algún lado relacionado.

En esta materia al trabajar con archivos separados, si el planteo del problema a resolver no lo aclara se supone que el dato relacionado siempre se va a encontrar. En otros problemas planteados se pide chequear la integridad mostrando algún mensaje de error por pantalla o grabando un archivo con los registros no encontrados.

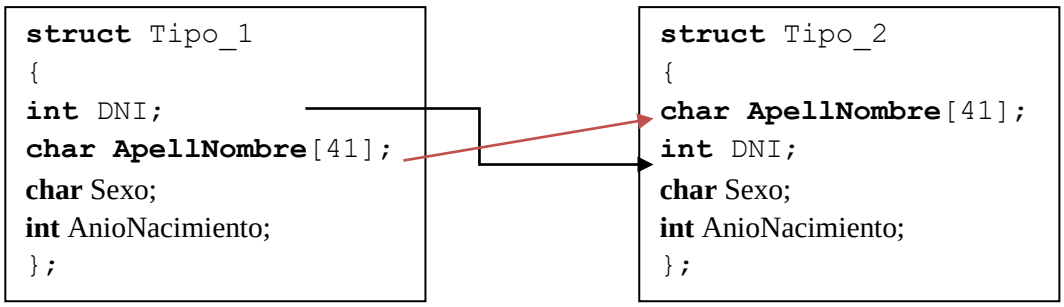
9. Ejemplos del uso de archivos con estructuras de datos

El uso de estructuras de datos en el manejo de "archivos binarios", resulta imprescindible pues la esencia de estos archivos binarios es la grabación y recuperación de registros es decir estructuras de datos (**struct**).

Tanto para la recuperación (lectura) o grabación (escritura) de un registro del archivo el mismo se realiza a través de una variable tipo estructura (**struct**) que obedezca al diseño del registro del archivo.

Es importante tener en cuenta que para poder recuperar (lectura) o grabar (escritura) la información que se encuentra en un archivo se debe conocer exactamente como está formada la estructura, porque será la única manera que se pueda trabajar con archivos. En el ejemplo que se desarrolla a continuación se corresponden a dos estructuras distintas, aunque ambas posean los mismos datos.

La estructura "**struct** Tipo_1" y la estructura "**struct** Tipo_2" contienen los mismos datos y la misma cantidad de bytes (50 bytes), pero son al mismo momento diferentes porque la información dentro de ambas se encuentra en órdenes diferentes. Para que dos estructuras sean exactamente iguales deben coincidir totalmente y no es el caso anterior por lo cual son distintas e incompatibles en su tratamiento.



Ejemplo 1:

Es un programa que permite grabar los datos de 10 alumnos en un archivo llamado "AlumnosUNLaM.dat".

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct FECHA
{
   int Dia;
   int Mes;
   int Anio;
};
struct PERSONA
{
   long int DNI;
   char ApellNombre[41];
   char Sexo;
   struct FECHA Nacimiento;
};
int main()
{
   struct PERSONA alumno;
   int i;
   FILE  *pf;
   pf  = fopen("AlumnosUNLaM.dat" , "wb");
   if (pf==NULL)
   {
      printf("No se pudo crear el archivo.");
      getch();
      exit (1);
   }
   for(i=0 ; i < 10; i++)
   {
     printf("Ingrese Numero de DNI");
```

```
      scanf("%d", &alumno.DNI);
      printf("Ingrese Numero de Apellido y nombre");
      fflush(stdin);
      gets(alumno.ApellNombre);
      printf("Ingrese Sexo");
      fflush(stdin);
      scanf("%c", &alumno.Sexo);
      printf("Ingrese Numero del Dia de Nacimiento");
      scanf("%d", &alumno.Nacimiento.Dia);
      printf("Ingrese Numero del Mes de Nacimiento");
      scanf("%d", &alumno.Nacimiento.Mes);
      printf("Ingrese Numero del Año de Nacimiento");
      scanf("%d",  &alumno.Nacimiento.Anio);
      fwrite(&alumno, sizeof(struct PERSONA), 1, pf);
      /*Aquí la función que permite graba un registro en un archivo binario. Se indica el nombre
      de la variable estructura  " alumno " (registro) que se va a grabar pf que es el puntero
      del archivo a grabar. La funcion sizeof determina el largo del registro por eso esta
      invocada la estructura PERSONA (58 bytes). */
   }
   return 0;
}
```

Ejemplo 2:

Es un programa que permite grabar los datos que se encuentra en un array (vector) de estructura de datos (los cuales fueron ingresados por teclado dentro de una función llamada “CargaDatosAlumnos”), en un archivo llamado “DatosAlumnosUNLaM.dat”

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct FECHA
{
   int Dia;
   int Mes;
   int Anio;
};
struct PERSONA
{
   long int DNI;
   char ApellNombre[41];
   char Sexo;
   struct FECHA Nacimiento;
} ;
void CargaDatosAlumnos (struct PERSONA []); //Prototipo de la función
int main()
{
   struct PERSONA Alumnos[50];
   int i;
   FILE *pf;

   CargaDatosAlumnos( Alumnos );// Llamada de la función
   pf = fopen("DatosAlumnosUNLaM.dat" , "wb");
   if (pf==NULL)
   {
      printf(“No se pudo crear el archivo.”);
      getch();
      exit (1);
   }
   for(i=0 ;  i < 50; i++)
   {
      fwrite(&Alumnos[i], sizeof(struct PERSONA), 1, pf);
      //Aquí la función que permite graba un registro a la vez en un archivo binario.
   }
   fclose(pf);
   return 0;
```

```
}
void CargaDatosAlumnos (struct PERSONA Alum[]) //Declaración de la función
{
   int i;
   for(i=0 ;i < 50; i++)
   {
      printf("Ingrese Numero de DNI");
      scanf("%d", & Alum[i].DNI);
      printf("Ingrese Apellido y nombre");
      fflush(stdin);
      gets(Alum[i].ApellNombre);
      printf("Ingrese Sexo");
      fflush(stdin);
      scanf("%c", & Alum[i].Sexo);
      printf("Ingrese Numero del Dia de Nacimiento");
      scanf("%d", & Alum[i].Nacimiento.Dia);
      printf("Ingrese Numero del Mes de Nacimiento");
      scanf("%d", & Alum[i].Nacimiento.Mes);
      printf("Ingrese Numero del Año de Nacimiento");
      scanf("%d",  & Alum[i].Nacimiento.Anio);
   }
}
```

Ejemplo 3 :

Es un programa que recupera (lee) los datos que se encuentran en un archivo llamado “DatosAlumnosUNLaM.dat”, y luego guardarlos en un arreglo (vector) de estructura de datos, los cuales se muestran por pantalla por medio de una función llamada “DatosAlumnos”. Para realizar este ejemplo se debe compilar siempre y cuando exista el archivo llamado “DatosAlumnosUNLaM.dat” y que contenga 50 registros.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

struct FECHA
{
   int Dia;
   int Mes;
   int Anio;
};
struct PERSONA
{
   long int DNI;
   char ApellNombre[41];
   char Sexo;
   struct FECHA  Nacimiento;
};
void DatosAlumnos (struct PERSONA []);//Prototipo de la función
int main()
{
   struct PERSONA Alumnos[50];
   int i;
   FILE *pf;
   pf = fopen("DatosAlumnosUNLaM.dat", "rb");
   if (pf==NULL)
   {
      printf(“No se pudo abrir el archivo.”);
      getch();
      exit (1);
   }
   for(i=0 ;  i < 50; i++)
   {
      fread(&Alumnos[i], sizeof(struct PERSONA), 1, pf);
      //Aquí la función que permite recuperar (leer) un registro a la vez en un archivo binario.
   }
   DatosAlumnos ( Alumnos );// Llamada de la función
   fclose(pf);
```

```
   return 0;
}

/*Declaración de la función. Permite recibir como parámetro un arreglo (array) de estructura de
datos y luego mostrarlos por patalla.*/
void DatosAlumnos (struct PERSONA Alum[])
{
   int i;
   for(i=0 ;  i < 50; i++)
   {
      printf("\nFecha de Nacimiento: %d-%d-%d Nro DNI: %d nombre: %s sexo %c",
      Alum[i].Nacimiento.Dia, Alum[i].Nacimiento.Mes, Alum[i].Nacimiento.Anio, Alum[i].DNI,
      Alum[i].ApellNombre, Alum[i].Sexo);
   }
}
```

10.Crear archivos de pruebas con datos prefijados

Muchas veces se desarrolla un algoritmo considerando que ya existe determinado archivo con datos previamente cargados. Pero sino se dispone de un archivo de pruebas no se puede comprobar su correcto funcionamiento. Además, es bueno siempre tener distintos conjuntos de datos de prueba para corroborar el correcto funcionamiento de un programa abarcando distintas alternativas.

Para ello se deben crear en forma manual los archivos de prueba. Una forma de crearlos en permitiendo el ingreso de datos por teclado y grabando los archivos, pero este proceso requiere de validaciones y carga de datos por teclado lo que puede llevar a errores y bastante tiempo de desarrollo. Un método más efectivo para las pruebas es definir el conjunto de datos que se requiere que tenga el archivo y armar un programa que genere el archivo con esos datos prefijados. A continuación, se muestra un código de ejemplo para generar un archivo de prueba de productos, pero el código esta diseñado para que rápidamente pueda modificar para generar cualquier otro archivo binario.

El nombre del archivo a generar se declara como una constante con define y luego se declara la estructura que define el formato del registro del archivo:

```
#include <stdio.h>
#define ARCHIVO "productos.dat"

typedef struct
{
    int codigo;
    char descripcion[31];
    float precioUnitario;
    char tipo;
}T_Registro;
```

En el main se declara un vector de estructuras del tipo del formato del registro y se lo inicializa con los valores preestablecidos deseados. Según sea el formato del registro del archivo a general serán los datos y el órden de los campos que debe establecerse para armar los registros de prueba. El siguiente ejemplo inicializa un vector con 6 registros de productos, donde entre llaves se especifican los campos de cada registro en el orden declarado en la estructura, en este caso Codigo, Descripción, Precio y Tipo. Pueden definirse todos los registros que se deseen ya que el tamaño del vector será calculado automáticamente ya que realiza en la inicializacióndel mismo.

```
int main()
{
    int i, cant;
    T_Registro vec[]={
            {100,"PINZA CURVA",3250.38,'I'},
            {111,"PINZA RECTA",1250.50,'N'},
            {200,"CABLE X100",5000.00,'N'},
            {210,"CABLE COBRE X50",9023.00,'I'},
            {915,"TUERCA 24",123.00,'N'},
            {999,"SELLADOR",1020.76,'I'}
    };
```

Luego se procede a abrir el archivo en modo escritura y se graban los registros del vector en el archivo. El cálculo de la cantidad de registros se realiza en forma automática para no tener que modificar nada adicional en el código.

```
    FILE *fp;
    cant = sizeof(vec) / sizeof(T_Registro);
    fp = fopen(ARCHIVO, "wb");
    for (i=0;i<cant;i++)
        fwrite(&vec[i],sizeof(T_Registro),1, fp);
    fclose(fp);

    return 0;
}
```

De esta forma con este simple programa se pueden generar los archivos de prueba deseados, simplemente se deben definir los campos del formato del registro y establecer los valores deseados en la inicialización del vector de estructuras, además de defnir el nombre del archivo a generar en la constante.

11.Exportar datos para otros programas

Los archivos binarios son dependientes del tamaño del tipo de dato del lenguaje y de la arquitectura del procesador con el cual fueron creados. Por ejemplo, en los entornos Windows o Linux actuales una variable int ocupa 4 bytes mientras que en sistemas operativos anteriores como D.O.S. un entero ocupaba 2 bytes en memoria. Adicionalmente no se puede leer un archivo binario sino se conoce exactamente como es el formato del registro de datos con el cual fue generado. Es por ello que cuando se desea intercambiar datos entre distintos sistemas en general se utilizan archivos de texto plano ya que pueden ser interpretados por cualquier lenguaje y bajo cualquier arquitectura sin problemas.

El uso de archivos de texto para exportar datos se puede realizar de dos formas distintas:

- Utilizar algún formato estandarizado que permita grabar los datos de una forma estructurada para que puedan ser interpretados. Algunos ejemplos son los formatos XML o JSON este último creado para representar objetos en lenguajes de programación no estructurados.

- Grabar los registros en el archivo de texto con algún separador de campo, por ejemplo, grabando cada campo de la estructura separado por coma o por punto y coma y haciendo que cada registro se grabe en una línea distinta del archivo de texto.

De las dos estrategias utilizaremos la segunda por su sencillez y además permitirá rápidamente poder consultar los datos exportados mediante programas de hojas de cálculo como por ejemplo Excel.

El primer paso es crear un archivo de texto. Para ello se debe modificar el modo de apertura del archivo cambiando la letra b en los modos indicados previamente por la letra t que representa un archivo de texto. Para exportar los datos utilizaremos dos modos de apertura de archivos de texto:

- wt: cuando se quiera crear un archivo de texto únicamente para escritura tomando en consideración de que si ya existía los datos se van a sobrescribir

- at: cuando se quieran exportar datos en un archivo de texto, agregando registros si ya existe o creando un nuevo archivo si el mismo no existía.

Para escribir los datos en el archivo de texto utilizaremos una nueva función **fprintf**. Esta función es similar al printf para mostrar los datos por pantalla con formato solo que en lugar de por pantalla se le indica el archivo en el cual se guardaran los datos. El formato de la función fprintf es el siguiente:

```
fprintf(puntero FILE, texto con formato, lista de variables separadas por coma);
```

A continuación, se muestra un programa que permite ingresar datos de alumnos por teclado y generar un archivo de texto en formato csv (del inglés comma separated values, valores separados por coma) con los datos ingresados. Como separador de campo en lugar de coma se recomienda utilizar punto y coma ya que de esta forma programas como el Excel al abrirlo automáticamente reconocen los distintos campos y ya nos muestran los datos en distintas columnas.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct
{
   char nombre[40];
   int dni;
   float promedio;
}snombres;
int main()
{
   snombres reg;
   FILE * fp;
   //se usa el modo de apertur wt que crea o reemplaza el archivo de texto
   fp=fopen ("alumnos.csv", "wt");
   if (fp==NULL)
   {
      printf("Error al abrir el archivo.");
      exit(1);
   }
   /*Graba una fila de encabezados para identificar los campos, el \n al final hace que se grabe una lína de texto completa y luego baje  a la fila siguiente   */
   fprintf(fp, "Nombre;DNI;Promedio\n");

   printf ("ingrese dni 0 para terminar:");
   scanf("%d", &reg.dni);
   while (reg.dni>0)
   {
      printf ("ingrese el nombre:");
      getchar();
      gets(reg.nombre);
```

```
        printf ("Ingrese el promedio:");
        scanf("%f", &reg.promedio);
        //El formato es identico al printf solo se agrega el ;como separadador entre dato y dato
        fprintf(fp, "%s;%d;%.2f\n", reg.nombre, reg.dni, reg.promedio);

        printf ("ingrese dni 0 para terminar:");
        scanf("%d", &reg.dni);
    }
    fclose(fp);
    return 0;
}
```