Mantenimiento de Archivos

Todos los sistemas de procesamiento de transacciones, aquellos que utilizan un conjunto de archivos relacionados como los vistos en el apartado anterior, realizan un conjunto de tareas de mantenimiento sobre sus archivos. Estas tareas se conocen por sus siglas **ABM** (de **A**lta, **B**aja y **M**odificación), y generalmente se presentan juntas en un menú; aparte de estas tareas es muy útil agregar **Listados**. Veamos que son cada una de estas tareas:

Altas: consiste en agregar registros a los archivos. En su forma más elemental consiste en la apertura del archivo para escritura ("wb" o "ab"), la asignación de los datos del registro que se quiere escribir en la variable de memoria correspondiente, la escritura del registro en el archivo, y luego su cierre. Podríamos repetir el ejemplo visto en el apunte de archivod para fwrite( ), pero ahora convertido en la función alta().

```
bool alta()
{
  FILE *part;
  Articulo reg;
  bool escribió;
   part=fopen("articulo.dat","ab"); //Apertura en modo agregar
  if(part==NULL)// Se comprueba que la apertura fue correcta
    { cout<<"Error de archivo";
       return false;    }// si no se pudo hacer devuelve false
  reg.Cargar();  //Ingreso de los datos
  escribió=fwrite(&reg, sizeof reg,1,part); //Escritura en el
                                            //    archivo
  fclose(part);// Cierre del archivo
  return 1;
}
```

En la práctica, el Alta suele incluir la validación de los datos (chequear que sean del tipo correcto y que no se repita) antes de su escritura en el archivo.

Bajas: la baja consiste en la eliminación de registros dentro del archivo. Es una de las operaciones más críticas, ya que al borrar información de un sistema, ésta puede perderse para siempre y generar graves problemas. Además, como trabajamos con archivos relacionados, borrar un registro de un archivo puede ocasionar que se pierdan las referencias que hacían otros archivos a ese registro. Teniendo en cuenta las características señaladas, suele implementarse 2 tipos de bajas: lógica y física.

Baja lógica: consiste en "marcar" como borrado un registro. El registro seguirá existiendo dentro del archivo pero el sistema lo ignorará. Esto se consigue agregando un campo más a los registros (una propiedad en la clase) . Por ejemplo:

```
class Articulo
  {
private:
  char cod[5];
  char desc[30];
  float pu;
  int stock;
  bool activo;  //Nuevo campo para baja lógica.
public: ///métodos
  };
```

A la clase Articulo se le agregó la propiedad activo. Este campo se pone en true, por ejemplo, al agregar registros, y se cambia su valor a false al dar la baja lógica. Luego toda la estructura del sistema debe preveer que los registros de artículo deben tener valor 1 (true) en su campo activo para ser procesados.

La baja lógica permite que un registro borrado lógicamente pueda recuperarse, ya que físicamente existe. Luego una baja lógica consiste en la modificación del valor de un campo. Veremos como hacerla en el título Modificación.

Baja Física: consiste en eliminar físicamente registros de un archivo. No existen funciones que permitan eliminar uno o más registros directamente: para eliminar registros hay que volver a generar el archivo, descartando aquellos registros que se desee borrar.

Por lo general se procede de la siguiente manera:
- Se hace una copia registro a registro del archivo, con una extensión distinta (p.e. .bak).
- Se abre el archivo con “wb” (se eliminan los registros existentes), y luego se copian del archivo .bak aquellos registros que se quieran conservar.

Al copiar el archivo original con una extensión distinta, se crea una copia de seguridad que eventualmente permitirá recuperar registros borrados por error.

Modificación: consiste en cambiar el valor de uno o más campos de un registro. Como vimos, la baja lógica consistía en modificar el valor del campo activo; también podría ser necesario modificar el precio de un artículo, disminuir su stock con las ventas, etc.

La modificación podría describirse como la ejecución de las siguientes acciones:
- encontrar la posición en el archivo del registro que se quiera modificar
- copiar el registro en una variable de memoria
- asignar el/los nuevo/s valores en los campos
- escribir otra vez el registro, ahora con los valores nuevos, en la misma posición que ocupaba el registro antes de ser modificado.

Ahora bien, supongamos que deseamos cambiar el precio unitario del artículo de código "abcd", y que el archivo tiene los siguientes registros:

Código de articulo	Descripción	Precio Unitario	Stock	Activo
"aaaa"	"papa"	3.8	100	1
"bbbb"	"uva"	8.6	50	1
"abcd"	"naranja"	5.6	150	1
"bbbd"	"sandía"	3.3	200	1

Para encontrar el registro con el código "abcd" el programa tuvo que haber leído los dos registros anteriores; al realizar la 3º lectura encuentra el código buscado, pero el indicador de posición del archivo del puntero FILE estará ahora al inicio del registro siguiente. Si en esa posición escribiríamos en el archivo el registro con el nuevo valor de precio (p.e. 6.2) el archivo nos quedaría:

Código de producto	Descripción	Precio Unitario	Stock	Activo
"aaaa"	"papa"	3.8	100	1
"bbbb"	"uva"	8.6	50	1
"abcd"	"naranja"	5.6	150	1
"abcd"	"naranja"	6.2	150	1

ya que la escritura en el archivo siempre se realiza en la posición que indica el puntero FILE. Como resultado quedan 2 registros de código "abcd" con diferente precio, y se sobreescribe el registro "bbbd".

Para evitar este problema necesitamos **mover el puntero FILE un registro para atrás**. La función que realiza esta tarea es **fseek().** Sus parámetros son:

fseek(punteroFile, desplazamiento, desde_donde_desplazar)

donde

punteroFile: puntero FILE que se utilizó para abrir el archivo.

desplazamiento: cantidad de bytes que se desea desplazar el puntero

desde_donde_desplazar: posición a partir de la que se hará el desplazamiento. Las opciones son

- 0: comienzo del archivo (o la constante SEEK_SET)
- 1: posición actual (o la constante SEEK_CUR)

- 2: final del archivo (o la constante SEEK_END).

así entonces, si se quisiera ubicar el puntero al principio del archivo:

fseek(p,0.0) se desplaza 0 bytes desde el principio del archivo;

al final del archivo:

fseek(p,0,2) se desplaza 0 bytes desde el final del archivo;

10 bytes desde la posición actual:

fseek(p, 10, 1)

Volviendo a nuestro ejemplo, una vez encontrado el registro hay que volver un registro para atrás. Las opciones son:

fseek(pFILE, -sizeof registro, 1), esto es, retroceder un registro desde la posición actual.

Si bien esta instrucción es válida, algunos compiladores no admiten valores negativos de desplazamiento. Otra posibilidad sería hacer entonces un desplazamiento desde el inicio del archivo equivalente a la posición actual menos 1 registro.

fseek(pFILE, (posición actual –tamaño registro), 0)

Como posición actual es la cantidad de bytes que hay desde el principio del archivo hasta la posición actual, y esa posición es un registro más allá del que necesitamos, al restarle el tamaño de 1 registro logramos ubicarnos en la posición deseada. Bastaría saber cuál es la cantidad de bytes entre el principio del archivo y la posición actual, y eso lo obtenemos utilizando la función **ftell().** Su prototipo es:

long ftell(FILE *)

La función devuelve la cantidad de bytes entre el inicio del archivo y la posición que se encuentra el puntero. Así entonces, la llamada anterior a fseek() podría escribirse como

fseek(pFILE, (ftell(pFILE) –tamaño registro), 0)

Veamos entonces una función para modificar el precio unitario de un registro de articulos

```
int modificacion()
{
  FILE *part;
  Articulo reg;
  float pu;
  int escribio;
  char codigo[5];
  part=fopen("articulo.dat","rb+"); //Se abre el archivo en modo
                                    //lectura/escritura
  if(part==NULL)// Se comprueba que la apertura fue correcta
    { cout<<"Error de archivo";
       return -1;    }// si la apertura no se pudo hacer la
                      //función devuelve un -1
  //Ingreso del código del registro a modificar
  cout<<"Ingrese el código: ";
  cin>>codigo;
  //Búsqueda del registro en el archivo
  while(fread(&reg, sizeof reg, 1, part)==1)
     {
        if(strcmp(codigo, reg.getCodigo())==0)
            {
           cout<<"Ingrese el nuevo precio unitario: ";
           cin>> pu;
           reg.setPu(pu);
           fseek(part, ftell(part)-sizeof reg, 0);
//desplazamiento de part un registro hacia atrás
           escribió=fwrite(&reg, sizeof reg, 1, part);
//escritura del registro modificado
           fclose(part); // cierre del archivo
           return escribio;
//como ya se hizo la modificación se termina la función
           }
       }
  fclose(part);// Cierre del archivo en caso de que no se
//encuentre el código ingresado
  return -2; //se devuelve un -2 si no se encuentra el código en
//el archivo
}
```

Por su parte, una función para baja lógica podría ser

```
int bajaLogica()
{
  FILE *part;
  Articulo reg;
  int escribio;
  char codigo[5];
  part=fopen("articulo.dat","rb+");
//Se abre el archivo en modo lectura/escritura
  if(part==NULL)// Se comprueba que la apertura fue correcta
    { cout<<"Error de archivo";
       return -1;    }
// si la apertura no se pudo hacer la función devuelve un -1
  //Ingreso del código del registro a borrar
  cout<<"Ingrese el código: ";
  cin>>codigo;
  //Búsqueda del registro en el archivo
  while(fread(&reg, sizeof reg, 1, part)==1)
     {
        if(strcmp(codigo, reg.getCodigo()==0)
            {
            reg.getActivo(0);
// se cambia el valor del campo estado
           fseek(part, ftell(part)-sizeof reg, 0);
//desplazamiento de part un registro hacia atrás
           escribió=fwrite(&reg, sizeof reg, 1, part);
// escritura del registro modificado
           fclose(part); // cierre del archivo
           return escribio;
//como ya se hizo la modificación se termina la función
           }
       }
  fclose(part);// Cierre del archivo en caso que no se encuentre
//el código ingresado
  return -2; //se devuelve un -2 si no se encuentra el código en
el archivo
}
```

Ambas funciones utilizan el modo de apertura “rb+”. Este modo posibilita que en una misma sesión pueda leerse y escribirse, y permite que se manipule la posición del puntero con fseek().

Otra alternativa, más atenta a las “buenas prácticas de la programación”, para las funciones anteriores sería descomponerlas en otras funciones que cumplan parte de las tareas necesarias.

Estas tareas eran:

- encontrar la posición en el archivo del registro que se quiera modificar (hacerla con una **función de búsqueda**)
- copiar el registro en una variable de memoria (hacerla con una **función que lea** el registro de esa posición)
- asignar el/los nuevo/s valores en los campos (usar el/los **sets()** que corresponda/n)
- escribir otra vez el registro, ahora con los valores nuevos, en la misma posición que ocupaba el registro antes de ser modificado. (hacer una **función que reciba el registro y la posición y lo escriba en el archivo**)

Veamos cómo quedaría para el caso de la baja lógica

```
int bajaLogica()
{
  Articulo reg;
  int escribió, pos;
  char codigo[5];

  cout<<"Ingrese el código: ";
  cin>>codigo;
  //Búsqueda del registro en el archivo
  pos=buscarRegistroArticulos(cod);
  if(pos==-1) return -2; //no existe el código en un registro
  //lectura del registro que contiene el código buscado
  reg=leerRegistroArticulo(pos);
  reg.getActivo(0);
  // se cambia el valor del campo estado
  escribio=modificarRegistroArticulo(reg, pos);
  return escribió;
 }
```

Siempre es conveniente la distribución de las tareas a realizar entre funciones específicas. Nos permite construir un código más sencillo de entender, modificar y reutilizar.

Listados: es siempre conveniente que los sistemas dispongan de alternativas para presentar la información que manejan de distintas maneras. Una de las posibilidades de cubrir este objetivo es mediante la confección de listados de los contenidos de los archivos.

Un listado elemental sería la lectura e impresión por pantalla –secuencialmente- de cada uno de los registros de un archivo; otros listados podrían ser registros ordenados de distintas maneras, o "filtrados" de acuerdo a distintas características (todos los artículos de precio mayor a 20, todos los artículos de stock menor a 100, etc).

Por el momento podemos hacer listados secuenciales, y filtrados. En la próxima clase veremos alternativas para ordenar todos los registros contenidos en un archivo.