Archivos

2

Las estructuras de datos, a diferencia de los datos simples tienen un único nombre para más de un dato, son divisibles en miembros más elementales y dispones de operadores de acceso.
Por su forma de creación y permanencia en memoria pueden ser estáticas (creadas en tiempo de declaración, ejemplos registro, array) o dinámicas (creadas en tiempo de ejecución, ejemplos estructuras enlazadas con asignación dinámica en memoria)
Por su persistencia pueden ser de almacenamiento físico (archivos) o temporal (array, registros).
Nos ocuparemos aquí de estructuras de almacenamiento físico, archivos, estructura de dato que se utiliza para la conservación permanente de los datos. Desde el punto de vista de la jerarquía de los datos, una computadora maneja bits, que para poder manipularlos como caracteres (digitos, letras o caracteres especiales), se agrupan en bytes. Asi como los caracteres se componen de bits, los campos pueden componerse como un conjunto de bytes. Asi pueden conformarse los registros, o srtuct en C. Asi como un registro es un conjunto de datos relacionados, un archivo es un conjunto de registros relacionados. La organización de estos registros en un archivo de acceso secuebcial o en un archivo de acceso directo.

Estructura tipo registro:

Posiciones contiguas de memoria de datos no homogéneos, cada miembro se llama campo. Para su implementación en pascal se debe:

a) Declaración y definición de un registro

```
struct  NombreDelTipo {
            tipo de dato Identificador;
            tipo de dato Identificador;
} Nombre del identificador;
```

b) Operador de Acceso.
NombreDelIdentificador.Campo1 {Acceso al miembro campo1}

c) Asignación
   i) Interna: puede ser
      (1) Estructura completa Registro1←Registro2
      (2) Campo a campo Registro.campo ← Valor
   ii) Externa
      (1) Entrada
         (a) Teclado: campo a campo Leer(Registro.campo)
         (b) Archivo Binario: por registro completo Leer(Archivo, Registro)
      (2) Salida

(a) Monitor: Campo a campo Imprimir(Registro.campo)
(b) Archivo Binario: por registro completo Imprimir(Archivo, Registro)

Estructura tipo Archivo

Estructura de datos con almacenamiento físico en disco, persiste mas allá de la aplicación y su procesamiento es lento. Según el tipo de dato se puede diferenciar en archivo de texto (conjunto de líneas de texto, compuestas por un conjunto de caracteres, que finalizan con una marca de fin de línea y una marca de fin de la estructura, son fácilmente transportables) archivos binarios (secuencia de bytes, en general mas compactos y menos transportables).
Para trabajar con archivos en C es necesario:

- Definir el tipo de la struct en caso de corresponder
- Declarar la variable (es un puntero a un FILE -definida en stdio.h- que contiene un descriptor que vincula con un índice a la tabla de archivo abierto, este descriptor ubica el FCB -File Control Blocx- de la tabla de archivos abiertos). Consideraremos este nombre como nombre interno o lógico del archivo.
- Vincular el nombre interno con el nombre físico o externo del archivo, y abrir el archivo. En la modalidad de lectura o escritura, según corresponda.

En algoritmos y estructura de datos trabajamos con

- Archivos
  - De texto
  - Binarios
    - De tipo
      - De tipo registro
      - Con registros de tamaño fijo
    - Con acceso directo
    - En la implementación en C utilizamos
      - FILE *
      - fopen
      - fread
      - fwrite
      - feof
      - fseek
      - ftell

Archivos y flujos

Al comenzar la ejecución de un programa, se abren, automáticamente, tres flujos, stdin (estándar de entrada), stdout (estándar de salida), stderr (estándar de error).
Cuando un archivo se abre, se asocia a un flujo, que proporcionan canales de comunicación, entre al archivo y el programa.
FILE * F; asocia al identificador F que contiene información para procesar un archivo.
Cada archivo que se abre debe tener un apuntador por separado declarado de tipo FILE. Para luego ser abierto, la secuencia es:
FILE * Identificador;
Identificador = fopen("nombre externo ", "modo de apertuta)"; se establece una línea de comunicación con el archivo.

Los modos de apertura son:

Modo	Descripción
R	Reset Abre archivo de texto para lectura
Rt	Idem anterior,explicitando t:texto
W	Write Abre archivo de texto para escritura, si el archivo existe se descarta el contenido sin advertencia
Wt	Idem anterior,explicitando t:texto
Rb	Reset abre archivo binario para lectura
Wb	Write Abre archivo binario para escritura, si el archivo existe se descarta el contenido sin advertencia
+	Agrega la otra modalidad a la de apertura

Asocia al flujo
FILE * F;
F = fopen("Alumnos", wb+); abre para escritura (crea) el arcivo binario alumnos, y agrega lectura.
Para controlar que la apertura haya sido correcta se puede:
If ((F = fopen("Alumnos", wb+)) == NULL) {error(1); return 0};
Si el apuntador es NULL, el archive no se pudo abrir.

Archivos de texto:

Secuencia de líneas compuestas por cero o mas caracteres, con un fin de línea y una marca de final de archivo.

Función	Descripción
FILE *f;	Define f como puntero a FILE
f = fopen ("archivo", "w");	Asocia f a un flujo
f = fropen("archivo", "w");	Similar anterior, si esta abierto antes lo cierra
fclose(f);	Vacía el flujo y cierra el archivo asociado
fflush(f);	Produce el vaciado de los flujos
remove("archivo");	El archivo ya no queda accesible
rename("viejo", "nuevo");	Renombra con nuevo el viejo nombre
fprintf(f, "%d", valor);	Escritura con formato en un flujo
fscanf(f, "%d", &valor);	Lectura con formato desde un flujo
c = getchar();	Lee un carácter desde stdin
c = getc(f);	Lee un carácter desde el flujo
c = fgetc(f);	Igual que el anterior
ungetc (c, f);	Retorna el carácter al flujo y retrocede
putchar(c) ;	Escribe un carácter en stdin
putc(c, f);	Escribe un carácter en el flujo
fputc(c,f);	Igual al anterior
gets(s);	Lee una cadena de stdin
fgets(s, n, f);	Lee hasta n-1 carácter del flujo en s
puts(s);	Escribe una cadena en stdin
fputs(s, f);	Escribe la cadena s en el flujo
feof(f)	Retorna no cero si el indicador de fin esta activo
ferror(f);	Retorna no cero si el indicador de error esta activo
clearerr(f);	Desactiva los indicadores de error

Operaciones simples

```
FILE * AbrirArchTextoLectura( FILE * f, char nombre[]){
// Asocia el flujo f  con el archivo nombre, lo abre en modo lectura
      return fopen(nombre, "r");
}
```

```
FILE * AbrirArchTextoEscritura( FILE * f, char nombre[]){
// Asocia el flujo f  con el archivo nombre, lo abre en modo escritura
      return fopen(nombre, "w");
}
```

Equivalencias que pueden utilizarse entre C y la representación algorítmica para flujos de texto Siendo int c; char s[n]; float r; FILE * f;	
Representacion:	Equivalente a:
LeerCaracter(f,c)	c = fgetc(f)
LeerCadena(f,s)	fgets(s,n,f)
LeerConFormato(f,c,r,s)	fscanf(f,"%c%f%s",& c,&r,s)
GrabarCaracter(f,c)	fputc(c,f)
GrabarCadena(f,s)	fputs(f,s)
GrabarConFormato(f,c,r,s)	fprintf(f,"%c %f %s \n", c,r,s)

Patrones algorítmicos con archivos de texto:

Dado un archivo de texto leerlo carácter a carácter y mostrar su contenido por pantalla.

C

```
main()
{
FILE *f1, f2;
int c;
f1 = fopen("entrada", "r");
f2 = fopen("salida","w");
c = fgetc(f1);
while (!feof(f1)) {
   fputc(c, f2);
  c = fgetc(f1);
}
fclose(f1);
fclose(f2);
return 0;
}
```

Solución algorítmica

```
AbrirArchTextoLectura(f1, "entrada")
AbrirArchTextoEscritura(f2,"salida")
LeerCaracter(f1,c)
Mientras(!feof(f1))
   GrabarCaracter(f2,c)
   LeerCaracter(f1,c)
FinMientras
Cerrar(f1)
Cerrar(f2)
```

Dado un archivo de texto con líneas de no más de 40 caracteres leerlo por línea y mostrar su contenido por pantalla.

C

```
main() {
FILE *f1;
char s[10 + 1]c;
f1 = fopen("entrada", "r");
fgets(s, 10 + 1, f1);
while (!feof(f1)) {
   printf("%s/n",s);
  fgets(s, 10 + 1,f1);
}
fclose(f1);
return 0;
}
```

Solución Algorítmica

```
AbrirArchTextoLectura(f1, "entrada")
LeerCadena(f1,s)
Mientras(!feof(f1))
   Imprimir(s)
   LeerCadena(f1,s)
FinMientras
Cerrar(f1)
```

Dado un archivo de texto con valores encolumnados como indica el ejemplo mostrar su contenido por pantalla. 10 123.45 Juan

C

```
main(){
FILE *f1;
int a;
float f;
char s[10 + 1]c;
f1 = fopen("entrada", "r");
fscanf(f1,"%  %f %s",&a,&f,s);
while (!feof(f1)) {
   printf("%10d%7.2f%s\n",a,f,s);
  fscanf(f1,"%d %f %s",&a,&f,s);
}
fclose(f1);
return 0;
}
```

Solución algorítmica

```
AbrirArchTextoLectura(f1, "entrada")
LeerConFormato(f1,a,f,s)          //lectura con formato de un archivo de texto
Mientras(!feof(f1))
   Imprimir(a,f,s)
   LeerConFormato(f1,a,f,s)
FinMientras
Cerrar(f1)
```

Archivos binarios:

Para trabajar en forma sistematizada con archivos binarios (trabajamos con archivos binarios, de acceso directo, de tipo, en particular de tipo registro) se requiere definir el tipo de registro y definir un flujo.

Tenga en cuenta que en algoritmos y estructura de datos trabajamos con distintos tipos de estructuras, muchas de ellas manejan conjunto de datos del mismo tipo a los efectos de resolver los procesos batch que nos presentan distintas situaciones problemáticas de la materia. Así estudiamos Arreglos, en este caso archivos (en particular de acceso directo) y luego veremos estructuras enlazadas.

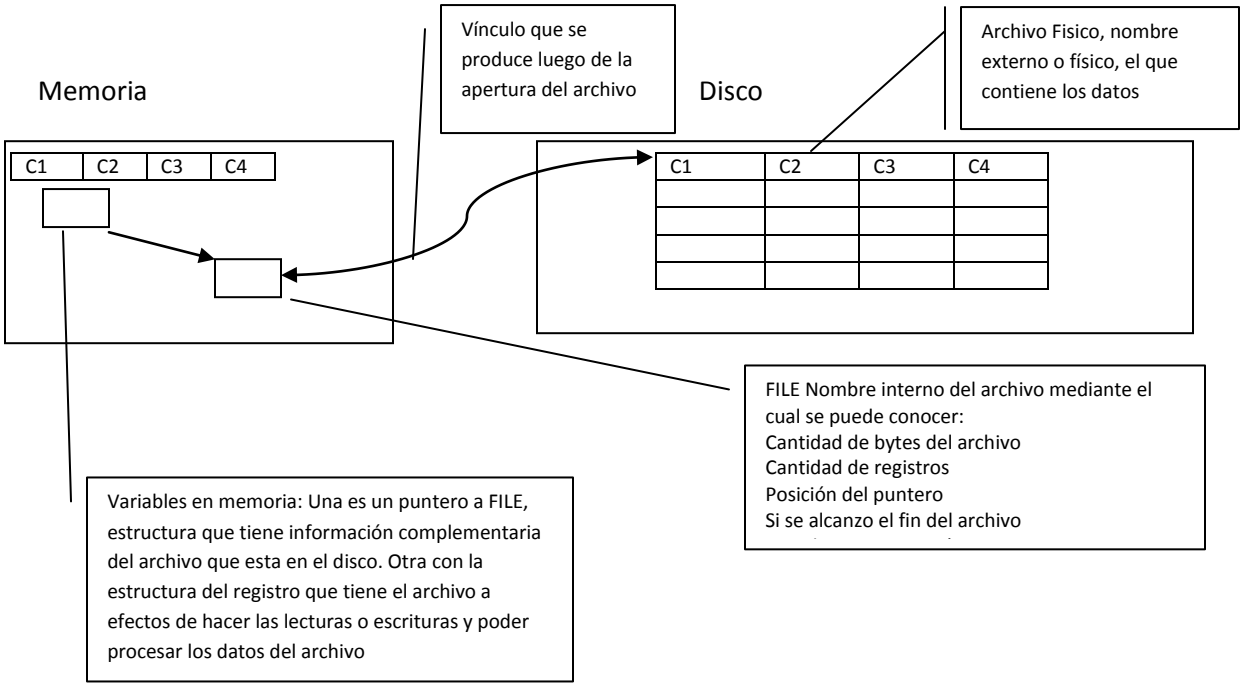
Cada una de estas estructuras tienen sus particularidades y uno de los problemas que debemos afrontar es la elección adecuada de las mismas teniendo en cuenta la situación particular que cada problema nos presente. A los efectos de hacer un análisis comparativo de las ya estudiadas, les muestro una tabla comparativa de arreglos y archivos, señalando algunas propiedades distintivas de cada una de ellas.

Análisis comparativo entre arreglos y archivos de registro de tamaño fijo		
Propiedad	Arreglo	Archivo
Tamaño Físico en Tiempo Ejecución	Fijo	Variable
Almacenamiento	Electrónico	Físico
- Persistencia	No	Si
- Procesamiento	Rápido	Lento
Búsquedas		
- Directa	Si	Si
- Binaria	Si (si esta ordenado)	Si
- Secuencial	Es posible	No recomendada
Carga		
- Secuencial	Si	Si (al final de la misma)
- Directa	Si	Solo con PUP
- Sin repetir clave	Si	No recomendada
Recorrido		
- 0..N	Si	Si
- N..0	Si	Si
- Con corte de control	Si	Si
- Con Apareo	Si	Si
- Cargando los N mejores	Si	No recomendada
Ordenamientos		
- Con PUP	Si	Si
- Método de ordenamiento	Si	No recomendado

En función de estas y otras características particulares de las estructuras de datos se deberá tomar la decisión de seleccionar la más adecuada según las características propias de la situación a resolver.

Como concepto general deberá priorizarse la eficiencia en el procesamiento, por lo que estructuras en memoria y con accesos directos ofrecen la mejor alternativa, lo cual hace a la estructura arreglo muy adecuada para este fin. Por razones varias (disponibilidad del recurso de

memoria, desconocimiento a priori del tamaño fijo, necesidad que el dato persista mas allá de la aplicación, entre otras) nos vemos en la necesidad de seleccionar estructuras diferentes para adaptarnos a la solución que buscamos. El problema, aunque puede presentarse como complejo, se resuelve con ciertas facilidad ya que la decisión no es entre un conjunto muy grande de alternativas, simplemente son tres. Una ya la estudiamos, los arreglos, otra es el objeto de este apunte, los archivos, las que quedan las veremos en poco tiempo y son las estructuras enlazadas.
Observen que se pueden hacer cosas bastante similares a las que hacíamos con arreglos, recorrerlos, buscar, cargar, por lo que la lógica del procedimiento en general la tenemos, solo cambia la forma de acceso a cada miembro de la estructura. En realidad las acciones son mas limitadas dado que muchas cosas que si hacíamos con arreglos como por ejemplo búsquedas secuenciales, carga sin repetición, métodos de ordenamiento, los desestimaremos en esta estructura por lo costoso del procesamiento cuando el almacenamiento es físico (en el disco)



Definiciones y declaraciones:

En los ejemplos de archivos, dado que la estructura del registro no es relevante, se utilizará el modelo propuesto para archivos binarios y de texto, si algún patrón requiere modificación se aclarará en el mismo:

Archivo binario

Numero	Cadena	Caracter
int	char cadena[N]	char

```
struct   TipoRegistro {
         int Numero;
         char Cadena[30];
         char C;
} Registro;
FILE * F;
```

Operaciones simples

```
FILE * abrirBinLectura( FILE * f, char nombre[]){
// Asocia el flujo f  con el archivo nombre, lo abre en modo lectura
      return fopen(nombre, "rb");
}

FILE *abrirBinEscritura( FILE * f, char nombre[]){
// Asocia el flujo f  con el archivo nombre, lo abre en modo escritura
      return fopen(nombre, "wb");
}


int  cantidadRegistros( FILE * f){
// retorna la cantidad de registros de un archivo
      TipoRegistro r;
      fseek(f, 0, SEEK_END); //pone al puntero al final del archivo
      return ftell(f)/sizeof(r); //cantidad de bytes desde el principio/tamaño del registro
}

int  posicionPuntero( FILE * f){
// retorna el desplazamiento en registros desde el inicio
      TipoRegistro r;
      return ftell(f)/sizeof(r); //cantidad de bytes desde el principio/tamaño del registro
}

int  seek( FILE * f, int pos){
// Ubica el puntero en el registro pos (pos registros desplazados del inicio)
      TipoRegistro r;
      return fseek(f, pos * sizeof(r), SEEK_SET);
}

int leer( FILE * f, TipoRegistro *r){
      //lee un bloque de un registro, retorna 1 si lee o EOF en caso de no poder leer.
      return fread(&r, sizeof(r) , 1, f);
}

int grabar( FILE * f, TipoRegistro r){
      //Graba un bloque de un registro.
      return fwrite(&r, sizeof(r) , 1, f);
}
```

```
int LeerArchivoCompleto( FILE * f, TipoVector v, int N){
      //lee un archivo y los guarda en un vector (el tamaño debe conocerse a priori).
      return fread(v, sizeof(v[0]) , CantidadRegistros(f), f);
}
```

Función	Descripción
FILE *f;	Define f como puntero a FILE
f = fopen (“archivo”, “wb”);	Asocia f a un flujo
f = fropen(“archivo”, “wb”);	Similar anterior, si esta abierto antes lo cierra
fclose(f);	Vacía el flujo y cierra el archivo asociado
fflush(f);	Produce el vaciado de los flujos
remove(“archivo”);	El archivo ya no queda accesible
rename(“viejo”, “nuevo”);	Renombra con nuevo el viejo nombre
sizeof(tipo)	Retorna el tamaño de un tipo o identificador
SEEK_CUR	Constante asociada a fseek (lugar actual)
SEEK_END	Constante asociada a fseek (desde el final)
SEEK_SET	Constante asociada a fseek (desde el inicio)
size_t fread(&r, tam,cant, f)	Lee cant bloques de tamaño tam del flujo f
size_t fwrite(&r,tam,cant,f)	Graba cant bloques de tamaño tam del flujo f
fgetpos(f, pos)	Almacena el valor actual del indicador de posicion
fsetpos(f,pos)	Define el indicador de posicion del archive en pos
ftell(f)	El valor actual del indicador de posición del archivo
fseek(f, cant, desde)	Define indicador de posicion a partir de una posicion.

ConvertirBinario_Texto(Dato_Resultado B: TipoArchivo; Dato_resultado T: Texto): una acción

Usar este algoritmo para convertir un archivo binario a uno de texto, teniendo en cuenta la transportabilidad de los archivos de texto estas acciones pueden ser necesarias

```
PRE:    B: Archivo binario EXISTENTE
        T: Archivo de texto a crear
        R: TipoRegistro {Variable local para las lecturas de los registros}
POS:    Crea el archivo de texto con el formato requerido y encolumnado correctamente.
LEXICO
 TipoRegistro R;
ALGORITMO
        AbrirArcBinarioLectura(B);
        AbrirArchTextoEscritura(T);
        MIENTRAS (! feof(B) ) HACER
                LeerRegistro(B,R); {lee de B un registro completo y lo almacena en memoria en R}
                GrabarConFormato(T,r.numero, r.cadena,r.caracter);
                {graba en el archivo B datos del registro que está en memoria respetando la
                máscara de salida, vea que en texto lo hace campo a campo y en binario por
                registro completo}
        FIN_MIENTRAS
        Cerrar(B);
```

```
        Cerrar(T);
FIN. // Convertir archivo binario a texto
```

ConvertirTexto_Binario(Dato_Resultado B: TipoArchivo; Dato_resultado T: Texto): una acción

```
PRE:    B: Archivo binario a crear
        T: Archivo de texto Existente
        R: TipoRegistro {Variable local para las lecturas de los registros}
POS:    Crea el archivo de binario a partir de uno de texto con formato conocido.
LEXICO
R : TipoRegistro;
ALGORITMO
        AbrirArchBinarioEscritura(B);
        AbrirArchTextoLectura(T);
        MIENTRAS (! feof(T) ) HACER
                LeerConFormato(T,r.numero, r.cadena:30,r.caracter);
                {lee desde el archivo B  los datos y los lleva a un registro que está en memoria, vea
                que en texto lo hace campo}
                GrabarRegistro(B,R); {Graba en el archivo B el registro R completo}
        FIN_MIENTRAS
        Cerrar(T);
        Cerrar(B);
FIN. // Convertir archivo binario a texto
```

AgregarRegistrosABinarioNuevo(Dato_Resultado B: TipoArchivo): una acción

```
PRE:    B: Archivo binario a crear al que se le deben agregar registros
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Crea el archivo de binario y le agrega reagistros, r.numero>0 es la condición del ciclo.
LEXICO
R : TipoRegistro;
ALGORITMO
        AbrirArchBinarioEscritura(B);
        Leer(r.numero); {lee desde el teclado el valor del campo numero del registro}
        MIENTRAS (r.numero > 0) HACER {analiza la expresión lógica propuesta}
                Leer(r.cadena,r.caracter); {lee del teclado el resto de los campos}
                GrabarRegistro(B,R); {Graba en el archivo B el registro R completo}
                {vea que  la lectura de un registro la hicimos campo a campo al leer  desde un flujo
                de text. En cambio leimos el registro completo al hacerlo desde un archivo binario.
                Para el caso de la escritura el concepto es el mismo}
                Leer(r.numero); {lee desde el teclado el valor del proximo registro}
        FIN_MIENTRAS
        Cerrar(B);
FIN. // Agregar Registros
```

AgregarRegistrosABinarioExistente(Dato_Resultado B: TipoArchivo): una acción

```
PRE:    B: Archivo binario existente al que se le deben agregar registros
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Crea el archivo de binario y le agrega reagistros, r.numero>0 es la condición del ciclo.
LEXICO
```

```
R : TipoRegistro;
ALGORITMO
        AbrirArchBinarioLectura(B);
        Leer(r.numero);
        seek(B, filesize(B));{pone el puntero al final del archivo, posición donde se agregan}
        //se usaron las funciones de biblioteca desarrolladas anteriormente
        MIENTRAS (r.numero > 0) HACER {todo lo siguiente es similar al patrón anterior}
                Leer(r.cadena,r.caracter);
                GrabarRegistro(B,R);
                Leer(r.numero);
        FIN_MIENTRAS
        Cerrar(B);
FIN. // Agregar Registros
```

RecorrerBinarioV1(Dato_Resultado B: TipoArchivo): una acción

Recorre un archivo completo con un ciclo de repetición exacto, dado que filesize del archivo indica la cantidad de registros, si se lee con un ciclo exacto una cantidad de veces igual a filesize y por cada iterracion se lee un registro se leerían todos los registros del archivo

```
PRE:    B: Archivo binario existente que se desea imprimir
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Muestra el contenido completo del archivo por pantalla.
LEXICO
R : TipoRegistro; {para la lectura}
I  : Entero;
N: Entero
ALGORITMO
        AbrirArchBinarioLectura(B);
        N = filesize(B); {Contiene en N la cantidad de registros }

        PARA [I = 1 .. N] HACER {Itera tantas veces como cantidad de registros}
                LeerRegistro(B,R); {por cada iteración lee un registro diferente}
                Imprimir(R.numero:8,R,cadena:30, R.caracter);{muestra por pantalla}
        FIN_PARA
        Cerrar(B);
FIN. // Recorrido con ciclo exacto
```

RecorrerBinarioV2(Dato_Resultado B: TipoArchivo): una acción

Recorre un archivo completo con un ciclo de repetición mientras haya datos, es decir mientras fin de archivo sea falso, con una lectura anticipada.

```
PRE:    B: Archivo binario existente que se desea imprimir
        R: TipoRegistro {Vriable local para las lecturas de los registros}
POS:    Muestra el contenido completo del archivo por pantalla.

LEXICO
R : TipoRegistro; {para la lectura}
ALGORITMO
        AbrirArchBinarioLectura(B);
        LeerRegistro(B,R);
```

```
        MIENTRAS (! feof(B)) HACER {Itera mientras haya datos}
                Imprimir(R.numero,R,cadena, R.caracter);{muestra por pantalla}
                Leer(B,R); {por cada iteración lee un registro diferente}

        FIN_MIENTRAS
        Cerrar(B);
FIN. // Recorrido con ciclo no exacto
```

RecorrerBinarioV3(Dato_Resultado B: TipoArchivo): una acción

Recorre un archivo completo con una lectura como condición del ciclo mientras.

PRE: B: Archivo binario existente que se desea imprimir
R: TipoRegistro {Vriable local para las lecturas de los registros}

POS: Muestra el contenido completo del archivo por pantalla.

```
LEXICO
R : TipoRegistro; {para la lectura}
F : Boolean {valor centinela que indica si se pudo leer el registro}
ALGORITMO

        MIENTRAS (LeerRegistro(B,R) != feof(B)}
                Imprimir(R.numero:8,R,cadena:30, R.caracter);{muestra por pantalla}
        FIN_MIENTRAS
        Cerrar(B);
FIN. //
```

CorteControlBinario(Dato_Resultado B: TipoArchivo): una acción

Recorre un archivo completo con corte de Control.

PRE: B: Archivo binario existente, ordenado, con una clave que se repite
R: TipoRegistro {Variable local para las lecturas de los registros}

POS: Muestra el contenido completo del archivo por pantalla agrupado por clave común.

Se hace una lectura anticipada para ver si hay datos. Se ingresa en un ciclo de repetición mientras haya datos, eso lo controla la variable F. Aquí se toma en una variable la clave de control. Se ingresa a un ciclo interno mientras haya datos y la clave leída sea igual al valor que se está controlando. Cuando se lee un registro cuyo valor no es igual al que se venía leyendo se produce un corte en el control que se estaba ejerciendo. Al salir del ciclo interno se verifica si sigue habiendo datos, si aún quedan, se toma la nueva clave como elemento de control y se sigue recorriendo, si no hubiera más se termina el proceso completo

```
LEXICO
R : TipoRegistro; {para la lectura}
Anterior : Dato del tipo de la clave por la que se debe agrupar
ALGORITMO
        AbrirArchBinarioLectura(B);
        LeerRegistro (B,R)
        {Inicializacion variables generales si corresponde}
        MIENTRAS (! feof (F)) HACER {Itera mientras haya datos, es decir F = Falso}
                Anterior ← r.clave {conserva en anterior la clave por la cual agrupar}
                {inicialización de variables de cada grupo}
                MIENTRAS((! feof(F)) Y (Anterior = R.Clave) HACER
                {recorre en ciclo interno mientras haya datos y la clave sea la misma}
                        {EJECUTAR LAS ACCIONES DE CADA REGISTRO}
```

```
                LeerRegistro (B,R)
            FIN_MIENTRAS
            {Acciones generales, si corresponden, de cada grupo}
        FIN_MIENTRAS
        {Acciones generales, si corresponde del total de los datos}
        Cerrar(B);
FIN. // Recorrido con corte de control
```

ApareoBinarioV1(Dato_Resultado A, B, C: TipoArchivo): una acción

Requiere al menos dos estructuras, con al menos un campo en común y se requiere procesarlos simultáneamente, intercalándolos por la clave común.

Se suponen los archivos ordenados crecientes, en caso de que el orden sea decreciente solo habrá que modificar los operadores de relación.

PRE: A, B: Archivos binarios existente, ordenado
C: Archivo Binario a crear
Ra,Rb: TipoRegistro {Variables locales para las lecturas de los registros}

POS: Intercala dos archivos manteniendo el orden de las claves

La primera versión presenta tres ciclos de repetición. Un primer ciclo mientras haya datos en ambos archivos y a continuación, cuando uno de ellos se termina, hacer un ciclo de repetición hasta agotar el que no se agoto. Como no se sabe a priori cual de los dos se agotara, se hacen los ciclos para ambos, desde luego que uno de ellos no se ejecutara. Mientras hay datos en ambos se requiere saber cual tiene la clave menor para procesarlo primero. Al agotarse uno de los archivos el otro archivo se lo procesa directamente sin necesidad de hacer ninguna comparación.

```
LEXICO
R a. Rb: TipoRegistro; {para la lectura}
F a, Fb: Boolean {valor centinela que indica si se pudo leer el registro}
ALGORITMO
        AbrirArchBinarioLectura(A);
        AbrirArchBinarioLectura(B);
        AbrirArchBinarioEscritura(C);
        LeerRegistro (A,Ra)
        LeerRegistro (A,Ra)
        {Inicializacion de variables generales}

        MIENTRAS ((! feof (A)) Y (!feof(B))) HACER { mientras haya datos en ambos archivos}
            SI (Ra.clave < Rb.clave) {si la clave del registro a es menor lo procesa y avanza}
            ENTONCES
                GrabarRegistro(C, Ra) {Procesa el registro de A y avanza}
                LeerRegistro (A,Ra)
            SI_NO
                Grabar(C, Rb) {Procesa el registro de B y avanza}
                LeerRegistro (B,Rb)
            FIN_SI
        FIN_MIENTRAS
        {agotar los que no se agotaron}
        MIENTRAS (!feof (A) ) HACER { agota A si es el que no termino}
            GrabarRegistro(C, Ra) {Procesa el registro de A y avanza}
            LeerRegistro (A,Ra)
        FIN_MIENTRAS
```

```
        MIENTRAS (!feof(B) ) HACER { agota B si es el que no termino}
                GrabarRegistro(C, Rb) {Procesa el registro de A y avanza}
                LeerRegistro (A,Rb)
        FIN_MIENTRAS
        Cerrar(A); Cerrar(B);     Cerrar(C);
FIN. // Recorrido apareo
```

ApareoBinarioV2(Dato_Resultado A, B, C: TipoArchivo): una acción

Requiere al menos dos estructuras, con al menos un campo en común y se requiere procesarlos simultaneamnte, intercalándolos por la clave común.

PRE: A, B: Archivos binarios existente, ordenado
C: Archivo Binario a crear
Ra,Rb: TipoRegistro {Variables locales para las lecturas de los registros}

POS: Intercala dos archivos manteniendo el orden de las claves

La segunda versión presenta un solo ciclo de repetició, mientras haya datos en alguno de los archivos. La expresión lógica de selección es un poco mas compleja, se coloca de a cuando b no tiene (Fb es verdadera) o cuando teniendo en ambos (Fa y Fb deben ser falsos) la clave de a es menor

```
LEXICO
R a. Rb: TipoRegistro; {para la lectura}
ALGORITMO
        AbrirBinarioLectura(A);
        AbrirBinarioLectura(B);
        AbrirBinarioEscritura(C);
        LeerRegistro (A,Ra)
        LeerRegistro (B,Rb)
        MIENTRAS ((! feof (A)) O (!feof(B))) HACER { mientras haya datos en algún  archivo}
                SI ((feof(B) O (( ! feof(A) Y(Ra.clave < Rb.clave)))
                {no hay datos en B, del otro lado del O se sabe que B tiene datos, como no se
                conoce si A los tiene,  se verifica, y si  hay también datos en a, habiendo en ambos
                se pregunta si el de a es menor en ese caso se procesa, es conceptualmente igual a
                la versión anterior aunque la expresión lógica parece un poco más compleja}
                ENTONCES
                        GrabarRegistro(C, Ra) {Procesa el registro de A y avanza}
                        LeerRegistro (A,Ra)
                SI_NO
                        GrabarRegistro(C, Rb) {Procesa el registro de B y avanza}
                FIN_SI
FIN_MIENTRAS
```

ApareoBinarioPorDosClaves(Dato_Resultado A, B, C: TipoArchivo): una acción

Requiere al menos dos estructuras, con al menos un campo en común y se requiere procesarlos simultaneamnte, intercalándolos por la clave común.

Se suponen los archivos ordenados crecientes, en este caso por dos campos de ordenamiento

PRE: A, B, : Archivos binarios existente, ordenado
C: Archivo Binario a crear
Ra,Rb: TipoRegistro {Variables locales para las lecturas de los registros}

POS: Intercala dos archivos manteniendo el orden de las claves
Se utiliza el criterio de la primera versión presentada
LEXICO
R a. Rb: TipoRegistro; {para la lectura}
ALGORITMO
AbrirArchBinarioLectura(A);
AbrirArchBinarioLectura(B);
AbrirArchBinarioEscritura(C);
LeerRegistro (A,Ra)
LeerRegistro (B,Rb)

MIENTRAS ((!feof(A)) Y (!feof(B))) HACER { *mientras haya datos en ambos archivos*}
SI ((Ra.clave1 < Rb.clave1)O((Ra.clave1 0 Rb.clave1)Y(Ra.clave2 <Rb.clave2)))
{*si la primera clave del registro a es menor lo procesa y avanza, también procesa de a si la primera clave es igual y la segunda es menor*}
ENTONCES
GrabarRegistro(C, Ra) {*Procesa el registro de A y avanza*}
LeerRegistro (A,Ra)
SI_NO
GrabarRegistro(C, Rb) {*Procesa el registro de B y avanza*}
LeerRegistro (B,Rb)
FIN_SI
FIN_MIENTRAS
{*agotar los que no se agotaron*}
MIENTRAS (! Feof(A)) HACER { *agota A si es el que no termino*}
GrabarRegistro(C, Ra) {*Procesa el registro de A y avanza*}
LeerRegistro (A,Ra)
FIN_MIENTRAS

MIENTRAS (!feof(B)) HACER { *agota B si es el que no termino*}
GrabarRegistro(C, Rb) {*Procesa el registro de A y avanza*}
LeerRegistro (A,Rb)
FIN_MIENTRAS
Cerrar(A); Cerrar(B);Cerrar(C);
FIN. // *Recorrido apareo*

BusquedaDirectaArchivo(Dato_Resultado B: TipoArchivo; Posic: Entero, Dato_Resultado R: TipoRegistro): una accion
Utilizar este algoritmo si los datos en el archivo están ordenados por un campo clave y la posición es conocida o es una Posición Única y Predecible
PRE: B: Archivo binario en el que se debe buscar con clave sin repetir
Posic: Posición donde se encuentra la clave buscada que puede ser PUP
POS: R: el registro completo de la clave buscada

LEXICO
Seek (B, Posic);{*ubica el puntero en la posición conocida o en la PUP*}
LeerRegistro (B,R);{*lee el registro en la posición definida*}
End.

BusquedaBinariaArchivo(Dato_Resultado B: TipoArchivo; Dato Clave:Tinfo; Dato_resultado Posic: Entero): una accion

Utilizar este algoritmo si los datos en el archivo están ordenados por un campo clave y se busca por ese campo. Debe tenerse en cuenta que si la clave es posicional se deberá utilizar búsqueda directa ya que la diferencia en eficiencia está dada entre 1, para la búsqueda directa y log_2N para la binaria

PRE: B: Archivo binario en el que se debe buscar con clave sin repetir
Clave : Valor Buscado

POS: Posic: Posición donde se encuentra la clave, ó (-1) si no esta

```
LEXICO
j : Entero;
u,m : Entero;
ALGORITMO
        Posic = -1;
        Pri = 0;
        U = filesize(B); //la función predefinida
        MIENTRAS (Pri < = U  y  Posic = -1) HACER
                M = (Pri + U ) div 2
                Seek(B, M); // la función predefinida
                LeerRegistro(B,R)
                SI R.clave = Clave
                ENTONCES
                        Posic = M;
                SI_NO
                        SI Clave > R.clave
                        ENTONCES
                                Pri = M+1
                        SI_NO
                                U = M – 1
                        FIN_SI
                FIN_SI
        FIN_MIENTRAS;

FIN. // Búsqueda binaria en archivo
```

BusquedaBinariaArchivoV2(Dato_Resultado B: TipoArchivo; Dato Clave:Tinfo; Dato_resultado Posic: Entero): una accion

Utilizar este algoritmo si los datos en el archivo están ordenados por un campo clave, la clave se repite y se desea encontrar la primera ocurrencia de la misma en el archivo

PRE: B: Archivo binario en el que se debe buscar con clave sin repetir
Clave : Valor Buscado

POS: Posic: Posición donde se encuentra la clave, ó (-1) si no esta

```
LEXICO
j : Entero;
```

```
u,m : Entero;
ALGORITMO
        Posic = -1;
        Pri = 0;
        U = filesize(B);
        MIENTRAS (Pri <  U  ) HACER
                M = (Pri + U ) div 2
                Seek(B, M);
                LeerRegistro(B,R)
                SI R.clave = Clave
                ENTONCES
                        Posic = M;
                        U = M{la encontró, verifica en otra iteración si también esta mas arriba}
                SI_NO
                        SI Clave > R.clave
                        ENTONCES
                                Pri = M+1
                        SI_NO
                                U = M – 1
                        FIN_SI
                FIN_SI
        FIN_MIENTRAS;

FIN. // Búsqueda binaria en archivo
```

El archivo como estructura auxiliar:

Como vimos en general es necesario la utilización de estructuras auxiliares con el propósito de acumular información según alguna clave particular, ordenar alguna estructura desordenada, buscar según una clave. Las estructuras más idóneas para este tipo de procesamiento son estructuras de almacenamiento electrónico por la eficiencia que brindan por la rápida velocidad de procesamiento, aquí la elección debe orientarse hacia estructuras tipo array, que permiten búsquedas directas, binarias y hasta secuencial, o estructuras enlazadas, listas, que permiten búsqueda secuencial, eficientes todas por la eficiencia que brinda el almacenamiento electrónico. De cualquier modo, es posible utilizar estructura tipo archivo como estructura auxiliar o paralela, en algunos casos particulares que se abordan a continuación.

Los casos pueden ser:

- ✓ Generar un archivo ordenado a partir de una estructura desordenada cuando es posible hacerlo por tener una clave de Posición Única Predecible.
- ✓ Generar un archivo paralelo a un archivo ordenado, con clave de PUP para acumular.
- ✓ Generar un archivo paralelo a un archivo ordenado, sin clave de PUP para acumular.
- ✓ Generar un archivo índice, que contenga la dirección en la que se encuentra una clave para poder hacer búsqueda directa.

Como se comentó, estas mismas opciones se pueden utilizar con estructuras en memoria, son mas eficientes, solo se agregan estas alternativas como variantes conceptuales para profundizar el conocimiento de las estructuras de datos.

Para estos ejemplos se toman archivos con las siguientes estructuras

Datos.dat: Archivo de datos personales: TipoArchivoDatosPersonales

Numero	DatosPersonales	OtrosDatos
Entero	Cadena	Cadena

Transacciones.dat: Archivo de transacciones: TipoArchivoTransacciones

Numero	Compras
Entero	Real

GenerarBinarioOrdenadoPUP(Dato_Resultado A,B: TipoArchivoDatosPersonales): una accion

Utilizar este algoritmo sólo si la clave es numérica, con valor inicial y final conocido y están todos los valores intermedios. En este caso es posible generar un archivo con una posición que es única y se la puede conocer a priori. La posición que ocupará esa clave en el archivo ordenad es:

PUP = valor de la clave – Valor de la clave inicial. Es decir si decimos que los valores de la clave en los registros están entre 1 y 999, por ejemplo, la PUP = Registro.clave – 1, si los valores estuvieran comprendidos entre 30001 y 31000 la PUP = Registro.clave – 30000.

PRE: A: Archivo Binario desordenado existente. Las condiciones de la clave permiten una PUP
B : Archivo Binario a crear ordenado

POS: Genera un archivo ordenado con PUP, con el mismo registro que el sin orden

```
LEXICO
R : TipoRegistro;
PUP: Entero;
ALGORITMO
        AbrirArchBinarioLectura(A);
        AbrirArchBinarioEscritura(B);
        MIENTRAS (!feof(A) ) HACER
                LeerRegistro(A,R) {leer el registro del archivo sin orden}
                PUP = R.clave – 1 {Calcular la PUP, recordar que se debe restar el valor de la primera de las claves posibles, para el ejemplo se toma 1}
                Seek(B, PUP);{acceder a la PUP calculada, usando la función que creamos}
                GrabarRegistro(B,R);{graba el registro en la PUP, vea que no es necesario leer para conocer cuál es el contenido previo ya que sea cual fuera será reemplazado por el nuevo}
        FIN_MIENTRAS;
        Close(A); Close(B);
FIN. // Generar archivo con PUP
```

Notas:

- ✓ Paralelo a archivo ordenado para agrupar: es posible que se plantee la situación problemática siguiente: Se dispone de dos archivos, uno con los datos personales, y otro con las transacciones y se busca informar cuanto compro cada cliente. Esto requeriría agrupar por clase. Si el archivo de datos personales esta ordenado, la clave numérica y con valores comprendidos entre 1 y 1000 y están todos. Estas características, por lo visto hasta aquí, supone que en ese archivo se puede hacer una búsqueda directa por la clave ya que puede ser, por las características una PUP. Por otro lado en el archivo de transacciones pueden ocurrir varias cosas:

- Que el archivo de transacciones tenga un único registro por compra y este ordenado. Ambos ordenados, esto puede hacerse utilizando como patrón el apareo. Te invito a que lo hagas.
- Que el archivo de transacciones tenga varios registros por compra y este ordenado. Ambos ordenados, esto puede hacerse utilizando como patrón el apareo (o similar), en combinación con el corte de control. Te invito a que lo hagas
- Que el archivo de transacciones tenga un único registro por compra y este desordenado. Para ordenar las compras se puede generar un archivo paralelo al de datos personales con una PUP, en cada posición un dato de tipo real. Esto se puede realizar adaptando el patrón analizado, en este caso no es necesario leer pues no se debe acumular. Te invito también a que lo hagas.
- Que el archivo de transacciones tenga varios registros por compra y este desordenado. Para ordenar las compras se puede generar un archivo paralelo al de datos personales con una PUP, en cada posición un dato de tipo real. Esto se puede realizar adaptando el patrón analizado, en este caso, como se debe acumular es necesario *APUNTAR (a la PUP con Seek), LEER (el registro completo para llevarlo a memoria, recuerde que en este caso el puntero en el archivo avanza) MODIFICAR (el dato a acumular que está en memoria, APUNTAR (como el puntero avanzo se lo debe volver a la posición anterior, la tiene en la PUP o la puede calcular con filepos(Archivo) – 1, y finalmente GRABAR(llevar el registro modificado en memoria al disco para efectivizar la modificación. Hacelo!!!*
- *Que el archivo ordenado, este ordenado pero no pueda garantizarse la PUP, en este caso la búsqueda no puede ser directa con PUP. El criterio a utilizar es el mismo, con la salvedad que para poder determinar con precisión la posocion que la clave ocupa en el archivo auxiliar se lo debe buscar previamente con búsqueda binaria en el archivo de datos personales. Se vinculan por posición, por tanto cada posición del ordenado se supone la misma en el auxiliar.*
- *Si los datos están ordenados, como vimos, no es necesario generar estructuras auxiliares. Los datos tal como están deben ser mostrados, por tanto no se los debe retener para modificar el orden, que es esto lo que hacen las estructuras auxiliares. En caso de tener que generar una estructura auxiliar, primero debe generarse esta y luego se deben recorrer los dos archivos, el de datos y el auxiliar en forma paralela, uno proveerá información personal y el otro el valor comprado.*

- ✓ Creación de archivo índice: Si se tiene un archivo con una clave que puede ser PUP y esta desordenado y se requiere, por ejemplo, mostrarlo en forma ordenada por la clave, es posible generar un archivo ordenado con una PUP y mostrarlo. Puede ocurrir que no disponga de espacio en disco como para duplicar el archivo. Solo tiene espacio para las distintas posiciones de las claves y la referencia al archivo de datos. Esto permite generar un archivo con PUP que contenga la posición que efectivamente esa clave tiene en el archivo de datos. Esto es el concepto de estructura que estamos mencionando.
- ✓ Si tenemos un archivo desordenado y se quiere generar un archivo ordenado las alternativas son múltiples
    - Si el tamaño del archivo es conocido a priori y se dispone de memoria suficiente, es posible llevar el archivo a memoria, es decir, cargarlo en un vector, ordenar el vector y luego reconstruir el archivo reordenándolo según los datos cargados en el vector.

- Si el tamaño es conocido y el recurso total no alcanza para llevarlo a memoria, se puede cargar la clave por la cual ordenar y una referencia al archivo para poder hacer luego un acceso directo al recorrer el vector ordenado.

GenerarIndicePUP(Dato_Resultado A: TipoArchivoDatosPersonales; B:TipoArchivoEnteros): una accion

Utilizar este algoritmo sólo si la clave es numérica, con valor inicial y final conocido y están todos los valores intermedios .No se tiene espacio para duplicar el archivo .

PRE: A: Archivo Binario desordenado existente. Las condiciones de la clave permiten una PUP
B : Archivo Binario a crear ordenado con la referencia al archivo sin orden

POS: Genera un archivo ordenado con PUP, con la posición del registro en el sin orden

```
LEXICO
R : TipoRegistro;
PUP: Entero;
ALGORITMO
        AbrirBinarioLectura(A);
        AbrirBinarioEscritura(B);
        MIENTRAS (Not feof(A) ) HACER
                LeerRegistro(A,R) {leer el registro del archivo sin orden}
                PUP = R.clave – 1 {Calcular la PUP, recordar que se debe restar el valor de la primera de las claves posibles, para el ejemplo se toma 1}
                Seek(B, PUP);{acceder a la PUP calculada}
                GrabarRegistro(B,filepos(A)-1);{graba la posición del registro en el archivo de datos en la PUP del archivo indice, vea que no es necesario leer para conocer  cuál es el contenido previo ya que sea cual fuera será reemplazado por el nuevo}
        FIN_MIENTRAS;
        Close(A); Close(B);
FIN. // Generar archivo indice con PUP
```

Implementaciones C C++

2-a

Operaciones sobre archivos

El subconjunto de funciones de C, declaradas en el archico cabecera de entradad y salida que utilizaremos son:

`fopen` - Abre un archivo.
`fwrite` - Graba datos en el archivo.
`fread` - Lee datos desde el archivo.
`feof` - Indica si quedan o no más datos para ser leidos desde el archivo.
`fseek` - Permite reubicar el indicador de posición del archivo.
`ftell` - Indica el número de byte al que está apuntando el indicador de posición del archivo.
`fclose` - Cierra el archivo.

Utilizando las funciones anteriores y con el propósito de facilitar las implementaciones, en hojas anteriores se han desarrollado las siguientes funciones propias, que utilizaremos, en algunos casos, a efectos de facilitar la comprensión, estas son:

`seek` – Posiciona el puntero en una posición dada.
`cantidadRegistros` – Indica cuantos registros tiene un archivo.
`posicionPuntero` – Retorna el desplazamiento en registros desde el inicio.
`leer` – Lee un registro del archivo.
`grabar` – Graba un registro en el archivo.

Grabar un archivo de caracteres

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{
      // abro el archivo; si no existe => lo creo vacio
      FILE* arch = fopen("DEMO.DAT","wb+");
      char c = 'A';
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'A' contenido en c
      c = 'B'; // C provee también fputc(c, arch);
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'B' contenido en c
```

```
      c = 'C';
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'C' contenido en c
      fclose(arch);
  return 0;}
```

Leer un archivo caracter a caracter

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{
// abro el archivo para lectura
      FILE* arch = fopen("DEMO.DAT","rb+");
      char c;
      // leo el primer caracter grabado en el archivo
      fread(&c,sizeof(char),1,arch);
      // mientras no llegue el fin del archivo...
      while( !feof(arch) ){
        // muestro el caracter que lei
        cout << c << endl;
        // leo el siguiente caracter
        fread(&c,sizeof(char),1,arch);
      }
      fclose(arch);
  return 0;
}
```

Archivos de registros

Trabajaremos con archivos de registro de tamaño fijo, esto obliga a definir las cadenas como array de caracteres, como lo hace CComo trabajamos con archivos de registro de tamaño fijo, la única consideración que debe tenerse en cuenta es que la estructura a grabar o leer en o desde el archivo no debe tener campos de tipos `string`. En su lugar se utilizan arrays de caracteres, manejando las cadenas con el estili C. C++ provee, y utilizaremos el método `c_str`, que permite obtener una cadena tipo C, es decir char*. Se recuerda además que C no permite asignación de cadenas, es por ello que para copiarlas es necesario utilizar la función strcpy(receptora, acopiar).

Supongamos tener un aechivo de registros con la siguiente estructura:

```
struct Alumno
{
      int dni;
      char nombre[25];
};
```

Grabar un archivo de registros

Lectura de datos por teclado y se graban los mismos en un archivo.

```
#include <iostream> //la inclusion de las cabeceras necesarias
#include <stdio.h>
#include <string.h>
using namespace std; //declaracion del espacio de nombre
int main()
{
      FILE* f = fopen("Alumnos.DAT","w+b");
      int dni;
      string nom;
      Alumno a;
      // ingreso de datos
      cout << "Ingrese dni";
      cin >> dni;
      while( dni>0 ){
        cout << "Ingrese nombre";
        cin >> dni;
        a.dni = dni;
        strcpy(a.nombre,nom.c_str()); //
        fwrite(&a,sizeof(Alumno),1,f); // grabo la estructura en el archivo
        cout << "Ingrese dni: ";
        cin >> dni;
       }
      fclose(f);
  return 0;
}
```

Leer un archivo de registros

Mostrar el contenido del archivo cargado en el punto anteriorA continuación veremos un programa que muestra por consola todos los registros del archivo `PERSONAS.DAT`.

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int main()
{
      FILE* f = fopen("Alumnos.DAT","rb+");
      Alumno a;

      while(fread(&a,sizeof(Alumno),1,f)){
        cout << p.dni << ", " << p.nombre <<", " << p.altura << endl;
            }
      fclose(f);
 return 0; }
```

Acceso directo a los registros de un archivo

Acceder al registro de la quinta posición y mostrar su contenido

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int main()
{
      FILE* f = fopen("Alumnos.DAT","r+b");
      Alumno a;

      fseek(f,sizeof(Alumno)*(5-1), SEEK_SET);
      // se ubica el puntero al comienzo del quinto registro
      fread(&a,sizeof(a),1,f); //sizeof puede ser de tipo o de variable
      // muestro cada campo de la estructura leida
      cout << p.dni << ", " << p.nombre <<endl;
  return 0;
}
```

Acceder al ultimo registro mostrar su contenido

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int main()
{
      FILE* f = fopen("Alumnos.DAT","r+b");
      Alumno a;

      fseek(f,-sizeof(Alumno), SEEK_END);
      // se ubica el puntero al comienzo del ultimo registro
      fread(&a,sizeof(a),1,f); //sizeof puede ser de tipo o de variable
      // muestro cada campo de la estructura leida
      cout << p.dni << ", " << p.nombre <<endl;
  return 0;
}
```

Templates

Template: leer

```
template <typename T> T read(FILE* f)
{
	T buff;
	fread(&buff,sizeof(T),1,f);
	return buff;
}
```

Template: grabar

```
template <typename T> void write(FILE* f, T v)
{
	fwrite(&v,sizeof(T),1,f);
	return;
}
```

Template: seek

```
template <typename T> void seek(FILE* arch, int n)
{
	// SEEK_SET indica que la posicion n es absoluta respecto del inicio del archivo
	fseek(arch, n*sizeof(T),SEEK_SET);
}
```

Ejemplos

Leer un archivo de registros usando el template `read`.

```
	f = fopen("Alumnos.DAT","rb+");
	// leo el primer registro
	Alumno a; = read<Persona>(f);
	while( leer<Alumno> (f) ){

	  cout << p.dni<<", "<<p.nombre<<", "<<p.altura << endl;

	}
	fclose(f);
```

Sintesis Archivos

En algoritmos y estructura de datos trabajamos con

- Archivos
    - De texto
    - Binarios
        - De tipo
            - De tipo registro
            - Con registros de tamaño fijo
        - Con acceso directo
        - En la implementación en C utilizamos
            - FILE *
            - fopen
            - fread
            - fwrite
            - feof
            - fseek
            - ftell

En algoritmos trabajamos, en general con archivos binarios, en particular archivos de tipos, es decir, posiciones contiguas de datos del mismo tipo. En general el tipo utilizada para cada posición es el tipo registro, por lo que trebajamos con archivos de registros. Desde el punto de vista del acceso, trabajamos con archivos de acceso directo, es decir, para acceder a un determinado registro no tenemos necesidad de ir a la primera posición e ir recorriéndolos todos hasta llegar a el, podemos acceder a el directamente. En definitiva, y por todo lo dicho trabajaremos con: **Archivos binarios, de registros de tamaño fijo, con acceso directo.**

Para trabajar con archivos Se debe:

1. Declarar una variable tipo archivo: Sera el nombre interno del archivo y nos permitita vincularnos con el nombre externo o físico del archivo del disco, en definitiva el que tiene los datos.
   Esto se hace declarando, en C una variable de ctipo FILE *, es un puntero a file, que es una estructura en memoria que dispondrá de toda la información del archivo físico cuando se lo vincule.
   FILE* f; declara f como puntero a FILE para vincularse con un achivo físico.
2. Abrir el archivo. Para esto C dispone de la función fopen, que vincula el nombre lógico con el físico según el modo de apertura asignándole asi ese valor al puntero a FILE declarado.
   f=fopen(“Alumnos”,”rb+”); ya vimos que el nombre del archivo y el modo de apertura son cadenas y ya analizamos como se forman.
3. Con esto ya tenemos el archivo para trabajar con el, en el mismo se puede:
    a. Leer un registro, nosotros haremos en el subconjunto seleccionado lectura por bloques con **fread**, recordamos que en C se puede leer de varias formas (por carácter, por cadena, con formato), formas que no abordamos en algoritmos. Leer significa que el bloque que indicamos que lea a partir de la posición del indicador es llevado a memoria para que podamos trabajar con el.
    b. Grabar un registro. Esto es que el bloque al que hacemos referencia y que esta en memoria se graba a partir de la posición en la que esta el puntero.
    c. Cerrar un archivo. Es imprescindible hacerlo cuando se modifica un archivo ya que cerrar el archivo no solo lo cierra sino que, eventualmente, vacia el buffer donde se van almacenando las modificaciones que aun no se materializaron en el archivo.
    d. Verificar o controlar la condición de fin de archivo.
    e. Acceder en forma directa a una posición determinada con fseek

El formato es fseek(flujo, desplazamiento en bytes, desde donde)
El desde donde se establece mediante la utilización de las constantes: SEEK_SET, SEEK_CUR, SEEK_END, que representan: desde el inicio, desde el lugar donde esta el puntero, desde el final,
Alfunos ejemplos:

```
fseek(f, 0, SEEK_SET);
fseek(f, 0, SEEK_END);
fseek(f, 0, SEEK_CUR);
fseek(f, 3*sizeof(r) , SEEK_SET);
fseek(f, -sizeof(r) , SEEK_CUR);
fseek(f, -sizeof(r) , SEEK_CUR);
```

Los archives son estructuras de tamaño variable en tiempo de ejecución, con almacenamiento físico, lo que hace imprescindible su uso si el propósito es hacer que el dato persista pero su procesamiento, por ser físico es lento. Es por ello que debemos tener en cuenta algunas consideraciones antes de su utilización, en particular com estructura auxiliar o para recorridos y búsquedas frecuentes.
Los patrones de recorridos, búsquedas, inserciones, modificaciones y ordenamientos los veremos en otro tutorial para que sean utilizados con las precauciones que la estructura requiere.

1.1. Definiciones Comunes <stddef.h>

Define, entre otros elementos, el tipo `size_t` y la macro `NULL`. Ambas son definidas, también en otros encabezados, como en `<stdio.h>`.

`size_t`

Tipo entero sin signo que retorna el operador `sizeof`. Generalmente `unsigned int`.

`NULL`

Macro que se expande a un puntero nulo, definido por la implementación. Generalmente el entero cero `0` ó `0L`, ó la expresión constante `(void*)0` .

1.2. Manejo de Caracteres <ctype.h>

`int isalnum (int);`

Determina si el carácter dado `isalpha` o `isdigit` Retorna (ok ? ≠0 : 0).

`int isalpha (int);`

Determina si el carácter dado es una letra (entre las 26 minúsculas y mayúsculas del alfabeto inglés). Retorna (ok ? ≠0 : 0).

`int isdigit (int);`

Determina si el carácter dado es un dígito decimal (entre '0' y '9'). Retorna (ok ? ≠0 : 0).

`int islower (int);`

Determina si el carácter dado es una letra minúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

`int isprint (int);`

Determina si el carácter dado es imprimible, incluye al espacio. Retorna (ok ? ≠0 : 0)

`int isspace (int);`

Determina si el carácter dado es alguno de estos: espacio (`' '`), `'\n'`, `'\t'`, `'\r'`, `'\f'`, `'\v'`. Retorna (ok ? ≠0 : 0)

`int isupper (int);`

Determina si el carácter dado es una letra mayúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

`int isxdigit (int);`

Determina si el carácter dado es un dígito hexadecimal (`'0'`..`'9'`, `'a'`..`'f'` o `'A'`..`'F'`). Retorna (ok ? ≠0 : 0)

`int tolower (int c);`

Si `c` es una letra mayúscula (entre las 26 del alfabeto inglés), la convierte a minúscula.

Retorna (mayúscula ? minúscula : `c`)

`int toupper (int c);`

Si `c` es una letra minúscula (entre las 26 del alfabeto inglés), la convierte a mayúscula.

Retorna (minúscula ? mayúscula : `c`)

1.3. Manejo de Cadenas <string.h>

Define el tipo **size_t** y la macro **NULL**, ver *Definiciones Comunes*.

```
unsigned strlen (const char*);
```

Cuenta los caracteres que forman la cadena dada hasta el 1er carácter **'\0'**, excluido. Retorna (longitud de la cadena).

1.3.1. Concatenación

```
char* strcat (char* s, const char* t);
```

Concatena la cadena **t** a la cadena **s** sobre **s**. Retorna (**s**).

```
char* strncat (char* s, const char* t, size_t n);
```

Concatena hasta **n** caracteres de **t**, previos al carácter nulo, a la cadena **s**; agrega siempre un **'\0'**. Retorna (**s**).

1.3.2. Copia

```
char* strncpy (char* s, const char* t, size_t n);
```

Copia hasta **n** caracteres de **t** en **s**; si la longitud de la cadena **t** es < **n**, agrega caracteres nulos en **s** hasta completar **n** caracteres en total; atención: no agrega automáticamente el carácter nulo. Retorna (**s**).

```
char* strcpy (char* s, const char* t);
```

Copia la cadena **t** en **s** (es la asignación entre cadenas). Retorna (**s**).

1.3.3. Búsqueda y Comparación

```
char* strchr (const char* s, int c);
```

Ubica la 1ra. aparición de **c** (convertido a **char**) en la cadena **s**; el **'\0'** es considerado como parte de la cadena. Retorna (ok ? puntero al carácter localizado : **NULL**)

```
char* strstr (const char* s, const char* t);
```

Ubica la 1ra. ocurrencia de la cadena t (excluyendo al '\0') en la cadena s. Retorna (ok ? puntero a la subcadena localizada : **NULL**).

```
int strcmp (const char*, const char*);
```

Compara "lexicográficamente" ambas cadenas. Retorna (0 si las cadenas son iguales; < 0 si la 1ra. es "menor" que la 2da.; > 0 si la 1ra. es "mayor" que la 2da.)

```
int strncmp (const char* s, const char* t, size_t n);
```

Compara hasta **n** caracteres de **s** y de **t**. Retorna (como **strcmp**).

```
char* strtok (char*, const char*);
```

Separa en "tokens" a la cadena dada como 1er. argumento; altera la cadena original; el 1er. argumento es la cadena que contiene a los "tokens"; el 2do. argumento es una cadena con caracteres separadores de "tokens". Retorna (ok ? puntero al 1er. carácter del "token" detectado : **NULL**).

1.3.4. Manejo de Memoria

```
void* memchr(const void* s, int c, size_t n);
```

Localiza la primer ocurrencia de **c** (convertido a un **unsigned char**) en los **n** iniciales caracteres (cada uno interpretado como **unsigned char**) del objeto apuntado por **s**. Retorna (ok ? puntero al carácter localizado : **NULL**).

```
int memcmp (const void* p, const void* q, unsigned n);
```

Compara los primeros **n** bytes del objeto apuntado por **p** con los del objeto apuntado por **q**. Retorna (0 si son iguales; < 0 si el 1ero. es "menor" que el 2do.; > 0 si el 1ero. es "mayor" que el 2do.)

```
void* memcpy (void* p, const void* q, unsigned n);
```

Copia **n** bytes del objeto apuntado por **q** en el objeto apuntado por **p**; si la copia tiene lugar entre objetos que se superponen, el resultado es indefinido. Retorna (**p**).

```
void* memmove (void* p, const void* q, unsigned n);
```

Igual que **memcpy**, pero actúa correctamente si los objetos se superponen. Retorna (**p**).

```
void* memset (void* p, int c, unsigned n);
```

Inicializa los primeros **n** bytes del objeto apuntado por **p** con el valor de **c** (convertido a **unsigned char**). Retorna (**p**).

1.4. Utilidades Generales <stdlib.h>

1.4.1. Tips y Macros

`size_t`
`NULL`
Ver *Definiciones Comunes*.
`EXIT_FAILURE`
`EXIT_SUCCESS`
Macros que se expanden a expresiones constantes enteras que pueden ser utilizadas como argumentos de `exit` ó valores de retorno de `main` para retornar al entorno de ejecución un estado de terminación no exitosa o exitosa, respectivamente.
`RAND_MAX`
Macro que se expande a una expresión constante entera que es el máximo valor retornado por la función `rand`, como mínimo su valor debe ser 32767.

1.4.2. Conversión

```
double atof (const char*);
```
Convierte una cadena que representa un real `double` a número `double`. Retorna (número obtenido, no necesariamente correcto).
```
int atoi (const char*);
```
Convierte una cadena que representa un entero `int` a número `int`. Retorna (número obtenido, no necesariamente correcto) .
```
long atol (const char*);
```
Convierte una cadena que representa un entero `long` a número `long`. Retorna (número obtenido, no necesariamente correcto).
```
double strtod (const char* p, char** end);
```
Convierte como `atof` y, si el 2do. argumento no es `NULL`, un puntero al primer carácter no convertible es colocado en el objeto apuntado por `end`. Retorna como `atof`.
```
long strtol (const char* p, char** end, int base);
```
Similar a `atol` pero para cualquier base entre 2 y 36; si la base es 0, admite la representación decimal, hexadecimal u octal; ver `strtod` por el parámetro `end`. Retorna como `atol`.
```
unsigned long strtoul (const char* p, char** end, int base);
```
Igual que `strtol` pero convierte a `unsigned long`. Retorna como `atol`, pero `unsigned long`.

1.4.3. Administración de Memoria

```
void* malloc (size_t t);
```
Reserva espacio en memoria para almacenar un objeto de tamaño `t`. Retorna (ok ? puntero al espacio reservado : `NULL`)
```
void* calloc (size_t n, size_t t);
```
Reserva espacio en memoria para almacenar un objeto de `n` elementos, cada uno de tamaño `t`. El espacio es inicializado con todos sus bits en cero. Retorna (ok ? puntero al espacio reservado : `NULL`)
```
void free (void* p);
```
Libera el espacio de memoria apuntado por `p`. No retorna valor.
```
void* realloc (void* p, size_t t);
```
Reubica el objeto apuntado por `p` en un nuevo espacio de memoria de tamaño `t` bytes. Retorna (ok ? puntero al posible nuevo espacio : `NULL`).

1.4.4. Números Pseudo-Aleatorios

```
int rand (void);
```
Determina un entero pseudo-aleatorio entre 0 y `RAND_MAX`. Retorna (entero pseudo-aleatorio).
```
void srand (unsigned x);
```
Inicia una secuencia de números pseudo-aleatorios, utilizando a `x` como semilla. No retorna valor.

1.4.5. Comunicación con el Entorno

`void exit (int estado);`

Produce una terminación normal del programa. Todos los flujos con *buffers* con datos no escritos son escritos, y todos los flujos asociados a archivos son cerrados. Si el valor de **estado** es **EXIT_SUCCESS** se informa al ambiente de ejecución que el programa terminó exitosamente, si es **EXIT_FAILURE** se informa lo contrario. Equivalente a la sentencia **return estado**; desde la llamada inicial de **main**. Esta función *no retorna a su función llamante*.

`void abort (void);`

Produce una terminación anormal del programa. Se informa al ambiente de ejecución que se produjo una terminación no exitosa. Esta función *no retorna a su función llamante*.

`int system (const char* lineadecomando);`

Si **lineadecomando** es **NULL**, informa si el sistema posee un procesador de comandos. Si **lineadecomando** no es **NULL**, se lo pasa al procesador de comandos para que lo ejecute.
Retorna (**lineacomando** ? valor definido por la implementación, generalmente el nivel de error del programa ejecutado : (sistema posee procesador de comandos ? ≠0 : 0)).

1.4.6. Búsqueda y Ordenamiento

```
void* bsearch (
     const void* k,
     const void* b,
     unsigned n,
     unsigned t,
     int (*fc) (const void*, const void*)
);
```

Realiza una búsqueda binaria del objeto ***k** en un arreglo apuntado por **b**, de **n** elementos, cada uno de tamaño **t** bytes, ordenado ascendentemente. La función de comparación **fc** debe retornar un entero < 0, 0 o > 0 según la ubicación de ***k** con respecto al elemento del arreglo con el cual se compara. Retorna (encontrado ? puntero al objeto : **NULL**).

```
void qsort (
     const void* b,
     unsigned n,
     unsigned t,
     int (*fc) (const void*, const void*)
);
```

Ordena ascendentemente un arreglo apuntado por **b**, de **n** elementos de tamaño **t** cada uno; la función de comparación **fc** debe retornar un entero < 0, 0 o > 0 según su 1er. argumento sea, respectivamente, menor, igual o mayor que el 2do. No retorna valor.

1.5. Entrada / Salida <stdio.h>

1.5.1. Tipos

`size_t`

Ver *Definiciones Comunes*.

`FILE`

Registra toda la información necesitada para controlar un *flujo*, incluyendo su *indicador de posición en el archivo*, puntero asociado a un *buffer* (si se utiliza), un *indicador de error* que registra sin un error de lectura/escritura ha ocurrido, y un *indicador de fin de archivo* que registra si el fin del archivo ha sido alcanzado.

`fpos_t`

Posibilita registrar la información que especifica unívocamente cada posición dentro de un archivo.

1.5.2. Macros

`NULL`

Ver *Definiciones Comunes*.

EOF

Expresión constante entera con tipo **int** y valor negativo que es retornada por varias funciones para indicar *fin de archivo*; es decir, no hay mas datos entrantes que puedan ser leídos desde un *flujo*, esta situación puede ser porque se llegó al fin del archivo o porque ocurrió algún error. Contrastar con **feof** y **ferror**.

SEEK_CUR
SEEK_END
SEEK_SET

Argumentos para la función **fseek**.

stderr
stdin
stdout

Expresiones del tipo **FILE*** que apuntan a objetos asociados con los flujos estándar de error, entrada y salida respectivamente.

1.5.3. Operaciones sobre Archivos

```
int remove(const char* nombrearchivo);
```

Elimina al archivo cuyo nombre es el apuntado por **nombrearchivo**. Retorna (ok ? 0 : ≠0)

```
int rename(const char* viejo, const char* nuevo);
```

Renombra al archivo cuyo nombre es la cadena apuntada por **viejo** con el nombre dado por la cadena apuntada por **nuevo**. Retorna (ok ? 0 : ≠0).

1.5.4. Acceso

```
FILE* fopen (
     const char* nombrearchivo,
     const char* modo
);
```

Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** asociando un flujo con este según el **modo** de apertura. Retorna (ok ? puntero al objeto que controla el flujo : **NULL**).

```
FILE* freopen(
     const char* nombrearchivo,
     const char* modo,
     FILE* flujo
);
```

Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** y lo asocia con el flujo apuntado por **flujo**. La cadena apuntada por **modo** cumple la misma función que en **fopen**. Uso más común es para el redireccionamiento de **stderr**, **stdin** y **stdout** ya que estos son del tipo **FILE*** pero no necesariamente *lvalues* utilizables junto con **fopen**. Retorna (ok ? flujo : **NULL**).

```
int fflush (FILE* flujo);
```

Escribe todos los datos que aún se encuentran en el buffer del flujo apuntado por **flujo**. Su uso es imprescindible si se mezcla **scanf** con **gets** o **scanf** con **getchar**, si se usan varios **fgets**, etc. Retorna (ok ? 0 : **EOF**).

```
int fclose (FILE* flujo);
```

Vacía el *buffer* del flujo apuntado por **flujo** y cierra el archivo asociado. Retorna (ok ? 0 : **EOF**)

1.5.5. Entrada / Salida Formateada

Flujos en General

```
int fprintf (FILE* f, const char* s, ...);
```

Escritura formateada en un archivo ASCII. Retorna (ok ? cantidad de caracteres escritos : < 0).

```
int fscanf (FILE* f, const char*, ...);
```

Lectura formateada desde un archivo ASCII. Retorna (cantidad de campos almacenados) o retorna (**EOF** si detecta fin de archivo).

Flujos stdin y stdout

`int scanf (const char*, ...);`
Lectura formateada desde **stdin**. Retorna (ok ? cantidad de ítems almacenados : **EOF**).
`int printf (const char*, ...);`
Escritura formateada sobre **stdout**. Retorna (ok ? cantidad de caracteres transmitidos : < 0).

Cadenas

`int sprintf (char* s, const char*, ...);`
Escritura formateada en memoria, construyendo la cadena **s**. Retorna (cantidad de caracteres escritos).
`int sscanf (const char* s, const char*, ...);`
Lectura formateada desde una cadena **s**. Retorna (ok ? cantidad de datos almacenados : **EOF**).

1.5.6. Entrada / Salida de a Caracteres

`int fgetc (FILE*);` ó
`int getc (FILE*);`
Lee un carácter (de un archivo ASCII) o un byte (de un archivo binario). Retorna (ok ? carácter/byte leído : **EOF**).
`int getchar (void);`
Lectura por carácter desde **stdin**. Retorna (ok ? próximo carácter del buffer : **EOF**).
`int fputc (int c, FILE* f);` ó
`int putc (int c, FILE* f);`
Escribe un carácter (en un archivo ASCII) o un byte (en un archivo binario). Retorna (ok ? **c** : **EOF**).
`int putchar (int);`
Eescritura por carácter sobre stdout. Retorna (ok ? carácter transmitido : **EOF**).
`int ungetc (int c, FILE* f);`
"Devuelve" el carácter o byte **c** para una próxima lectura. Retorna (ok ? **c** : **EOF**).

1.5.7. Entrada / Salida de a Cadenas

`char* fgets (char* s, int n, FILE* f);`
Lee, desde el flujo apuntado **f**, una secuencia de a lo sumo **n**-1 caracteres y la almacena en el objeto apuntado por **s**. No se leen más caracteres luego del carácter nueva línea o del fin del archivo. Un carácter nulo es escrito inmediatamente después del último carácter almacenado; de esta forma, s queda apuntando a una cadena. Importante su uso con **stdin**. Si leyó correctamente, **s** apunta a los caracteres leídos y retorna **s**. Sí leyó sólo el fin del archivo, el objeto apuntado por **s** no es modificado y retorna **NULL**. Si hubo un error, contenido del objeto es indeterminado y retorna **NULL**. Retorna (ok ? **s** : **NULL**).
`char* gets (char* s);`
Lectura por cadena desde **stdin**; es mejor usar **fgets**() con **stdin** . Retorna (ok ? **s** : **NULL**).
`int fputs (const char* s, FILE* f);`
Escribe la cadena apuntada por **s** en el flujo **f**. Retorna (ok ? último carácter escrito : **EOF**).
`int puts (const char* s);`
Escribe la cadena apuntada por **s** en **stdout**. Retorna (ok ? ≥ 0 : **EOF**).

1.5.8. Entrada / Salida de a Bloques

`unsigned fread (void* p, unsigned t, unsigned n, FILE* f);`
Lee hasta **n** bloques contiguos de **t** bytes cada uno desde el flujo **f** y los almacena en el objeto apuntado por **p**. Retorna (ok ? **n** : < **n**).
`unsigned fwrite (void* p, unsigned t, unsigned n, FILE* f);`
Escribe n bloques de t bytes cada uno, siendo el primero el apuntado por p y los siguientes, sus contiguos, en el flujo apuntado por f. Retorna (ok ? **n** : < **n**).

1.5.9. Posicionamiento

`int fseek (`

```
    FILE* flujo,
    long desplazamiento,
    int desde
);
```
Ubica el *indicador de posición de archivo* del flujo binario apuntado por **flujo**, **desplazamiento** caracteres a partir de **desde**. **desde** puede ser **SEEK_SET**, **SEEK_CUR** ó **SEEK_END**, comienzo, posición actual y final del archivo respectivamente. Para flujos de texto, **desplazamiento** deber ser cero o un valor retornado por **ftell** y **desde** debe ser **SEEK_SET**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fsetpos (FILE* flujo, const fpos_t* posicion);
```
Ubica el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** según el valor del objeto apuntado por **posicion**, el cual debe ser un valor obtenido por una llamada exitosa a **fgetpos**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fgetpos (FILE* flujo, fpos_t* posicion);
```
Almacena el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** en el objeto apuntado por **posicion**, cuyo valor tiene significado sólo para la función **fsetpos** para el restablecimiento del *indicador de posición de archivo* al momento de la llamada a **fgetpos**. Retorna (ok ? 0 : ≠ 0).

```
long ftell (FILE* flujo);
```
Obtiene el valor actual del *indicador de posición de archivo* para el flujo apuntado por **flujo**. Para flujos binarios es el número de caracteres (bytes ó posición) desde el comienzo del archivo. Para flujos de texto la valor retornado es sólo útil como argumento de **fseek** para reubicar el indicador al momento del llamado a **ftell**. Retorna (ok ? indicador de posición de archivo : –1L).

```
void rewind(FILE *stream);
```
Establece el indicador de posición de archivo del flujo apuntado por **flujo** al principio del archivo. Semánticamente equivalente a **(void)fseek(stream, 0L, SEEK_SET)**, salvo que el indicador de error del flujo es desactivado. No retorna valor.

ANEXO 3 Flujos de texto y binario C++

8

C++ ifstream, ofstream y fstream

C++ para el acceso a ficheros de texto ofrece las clases ifstream, ofstream y fstream. (**i** input, **f** file y **stream**). (**o** output). fstream es para i/o.

Abrir los ficheros

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
/* pasando parámetros en la declaración de la variable */
   ifstream f("fichero.txt", ifstream::in);

/* se abre después de declararlo, llamando a open() */
   ofstream f2;
   f2.open("fichero2.txt", ofstream::out);
```

El primer parámetro es el nombre del fichero, el segundo el modo de apertura.

- **app** (**app**end) Para añadir al final del fichero. Todas las escrituras se hacen al final independiente mente de la posición del puntero.
- **ate** (**at e**nd). Para añadir al final del fichero. En caso de mover el puntero, la escritura se hace donde esta el mismo.
- **binary** (**binary**) Se abre el fichero como fichero binario. Por defecto se abre como fichero de texto.
- **in** (**in**put) El fichero se abre para lectura.
- **out** (**out**put) El fichero se abre para escritura
- **trunc** (**trunc**ate) Si el fichero existe, se ignora su contenido y se empieza como si estuviera vacio. Posiblemente perdamos el contenido anterior si escribimos en él.

Se puede abrir con varias opciones con el operador OR o el caracter **|** .

```
f2.open("fichero2.txt", ofstream::out | ofstream::trunc);
```

Hay varias formas de comprobar si ha habido o no un error en la apertura del fichero. La más cómoda es usar el operator ! que tienen definidas estas clases. Sería de esta manera

```
if (f)
{
   cout << "fallo" << endl;
```

```
   return -1;
}
```

!f (no f) retorna true si ha habido algún problema de apertura del fichero.

Leer y escribir en el fichero

Existen metodos específicos para leer y escribir bytes o texto: get(), getline(), read(), put(), write(). Tambien los operadores **<<** y **>>**.

```
/* Declaramos un cadena para leer las líneas */
char cadena[100];
...
/* Leemos */
f >> cadena;
...
/* y escribimos */
f2 << cadena;
/*Copiar un archivo en otro */
/* Hacemos una primera lectura */
f >> cadena;/*Lectura anticipada controla si es fin de archivo*/
while (!f.feof()){
   /* Escribimos el resultado */
   f2 << cadena << endl;
   /* Leemos la siguiente línea */
   f >> cadena;
}
```

Cerrar los ficheros

```
f.close(); f2.close();
```

Ejemplos Archivos de texto

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
   char cadena[128];
   // Crea un fichero de salida
   ofstream fs("nombre.txt");
   // Enviamos una cadena al fichero de salida:
   fs << "Hola, mundo" << endl;
   // Cerrar el fichero para luego poder abrirlo para lectura:
   fs.close();

   // Abre un fichero de entrada
   ifstream fe("nombre.txt");
   // Lectura mediante getline
   fe.getline(cadena, 128);
   // mostrar contenido por pantalla
   cout << cadena << endl;

   return 0;
}

int main() {
   char cadena[128];
   ifstream fe("streams.cpp");
   while(!fe.eof()) {
```

```
      fe >> cadena;
      cout << cadena << endl;
   }
   fe.close();
   return 0;}
```

Ejemplo Archivo binario

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
struct tipoReg {
   char nombre[32];
   int edad;
   float altura;
};

int main() {
   tipoReg r1;
   tipoReg r2;
   ofstream fsalida("prueba.dat",ios::out | ios::binary);
   strcpy(r1.nombre, "Juan");
   r1.edad = 32;
   r1..altura = 1.78;

   fsalida.write(reinterpret_cast<char *>(&r1),    sizeof (tipoReg));

fsalida.close();// lo cerramos para abrirlo para lectura

   ifstream fentrada("prueba.dat", ios::in | ios::binary);

   fentrada.read(reinterpret_cast<char *>(&r2), sizeof(tipoReg));
   cout << r2.nombre << endl;
   cout << r2.edad << endl;
   cout << r2.altura << endl;
   fentrada.close();

   return 0;
}
```

Acceso directo

```
#include <fstream>
using namespace std;
int main() {
   int i;
   char mes[][20] = {"Enero", "Febrero", "Marzo","Abril", "Mayo",
"Junio", "Julio", "Agosto", "Septiembre", "Octubre", "Noviembre",
      "Diciembre"};
   char cad[20];
```

```
   ofstream fsalida("meses.dat",ios::out | ios::binary);

   // Crear fichero con los nombres de los meses:
   cout << "Crear archivo de nombres de meses:" << endl;
   for(i = 0; i < 12; i++)
      fsalida.write(mes[i], 20);
   fsalida.close();

   ifstream fentrada("meses.dat", ios::in | ios::binary);

   // Acceso secuencial:
   cout << "\nAcceso secuencial:" << endl;
   fentrada.read(cad, 20);
   do {
      cout << cad << endl;
      fentrada.read(cad, 20);
   } while(!fentrada.eof());

   fentrada.clear();
   // Acceso aleatorio:
   cout << "\nAcceso aleatorio:" << endl;
   for(i = 11; i >= 0; i--) {
      fentrada.seekg(20*i, ios::beg);
      fentrada.read(cad, 20);
      cout << cad << endl;
   }

   // Calcular el número de elementos
   // almacenados en un fichero:
   // ir al final del fichero
   fentrada.seekg(0, ios::end);
   // leer la posición actual
   pos = fentrada.tellg();
   // El número de registros es el tamaño en
   // bytes dividido entre el tamaño del registro:
   cout << "\nNúmero de registros: " << pos/20 << endl;
   fentrada.close();

   return 0;
}
```

Para el acceso aleatorio seekp y seekg, que permiten cambiar la posición del fichero en la que se hará la siguiente escritura o lectura. La 'p' es de put y la 'g' de get, es decir escritura y lectura, respectivamente. Otras funciones relacionadas con el acceso aleatorio son tellp y tellg, que indican la posición del puntero en el fichero.

La función seekg permite acceder a cualquier punto del fichero, no tiene por qué ser exactamente al principio de un registro, la resolución de la funciones seek es de un byte.

La función seekp nos permite sobrescribir o modificar registros en un fichero de acceso aleatorio de salida. La función tellp es análoga a tellg, pero para ficheros de salida.