Algoritmos y estructura de datos

Asignatura anual, código 082021

MODULO 3 Struct y FILE

Departamento de Ingeniería en Sistemas de Información
Universidad Tecnológica Nacional FRBA

Tabla de contenido

Estructura de datos: Flujos y Registros

1

Objetivos de aprendizaje
Dominando los temas del presente trabajo Usted podrá.

1. Entender el concepto de estructura de dato.
2. Profundizar en el conocimiento de estructuras con posiciones contiguas de memoria de datos no homogeneos.
3. Conocer estructuras que permiten la persistencia del dato mas alla de la aplicacion.
4. Comenzar con las combinaciones de estructuras.

Introduccion

Hasta ahora trabajamos con datos simples, cuyas características son: tienen un único nombre para un único dato, y son indivisibles en datos más elementales. Por otro lado las estructuras de datos tiene la característica de tener un único nombre para más de un dato, es divisible en miembros más elementales y existen medios u operadores de acceso a cada uno de esos miembros. Pueden alojarse en memoria principal o secundaria, por lo que pueden o no persistir a la aplicación y ser más o menos eficiente en su acceso o recorrido. En este apartado, después de una breve descripción de los tipos de datos se abordara las estructuras tipos registro y los flujos de datos.
Las estructuras de datos, 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 secuencial o en un archivo de acceso directo.

Tipos de Datos

Identifica o determina un dominio de valores y el conjunto de operaciones aplicables sobre esos valores.

1. Primitivos.
2. Derivados.
3. Abstractos.

Los algoritmos operan sobre datos de distinta naturaleza, por lo tanto los programas que implementan dichos algoritmos necesitan una forma de representarlos.
Tipo de dato es una clase de objeto ligado a un conjunto de operaciones para crearlos y manipularlos, un tipo de dato se caracteriza por

1. Un rango de valores posibles.
2. Un conjunto de operaciones realizadas sobre ese tipo.
3. Su representación interna.

Al definir un tipo de dato se esta indicando los valores que pueden tomar sus elementos y las operaciones que pueden hacerse sobre ellos.
Al definir un identificador de un determinado tipo el nombre del identificador indica la localización en memoria, el tipo los valores y operaciones permitidas, y como cada tipo se representa de forma distinta en la computadora los lenguajes de alto nivel hacen abstracción de la representación interna e ignoran los detalles pero interpretan la representación según el tipo.

Como ya vimos, los tipos de datos pueden ser.

1. **Estáticos:** Ocupan una posición de memoria en el momento de la definición, no la liberan durante el proceso solamente la liberan al finalizar la aplicación.
    a. **Simples**: Son indivisibles en datos mas elementales, ocupan una única posición para un único dato de un único tipo por vez.
        i. **Ordinales**: Un tipo de dato es ordinal o esta ordenado discretamente si cada elemento que es parte del tipo tiene un único elemento anterior (salvo el primero) y un único elemento siguiente (salvo el ultimo).
            1. **Enteros**: Es el tipo de dato numérico mas simple.
            2. **Lógico** o booleano: puede tomar valores entre dos posibles: verdadero o falso.
            3. **Carácter**: Proporcionan objetos de la clase de datos que contienen un solo elemento como valor. Este conjunto de elementos esta establecido y normatizado por el estándar ASCII.
        ii. **No ordinales**: No están ordenados discretamente, la implementación es por aproximación
            1. Reales: Es una clase de dato numérico que permite representar números decimales.
    b. **Cadenas**: Contienen N caracteres tratados como una única variable.
    c. **Estructuras**: Tienen un único nombre para mas de un dato que puede ser del mismo tipo o de tipo distinto. Permiten acceso a cada dato particular y son divisibles en datos mas elementales.
    Una estructura es, en definitiva, un conjunto de variables no necesariamente del mismo tipo relacionadas entre si de diversas formas.
    Si los datos que la componen son todas del mismo tipo son homogéneas, heterogéneas en caso contrario.
    Una estructura es estática si la cantidad de elementos que contiene es fija, es decir no cambia durante la ejecución del programa
        i. **Registro**: Es un conjunto de valores que tiene las siguientes características:
        Los valores pueden ser de tipo distinto. Es una estructura heterogénea.
        Los valores almacenados se llaman campos, cada uno de ellos tiene un identificador y pueden ser accedidos individualmente.

El operador de acceso a cada miembro de un registro es l operador punto (**.**)

El almacenamiento es fijo.

ii. **Arreglo**: Colección ordenada e indexada de elementos con las siguientes características:

Todos los elementos son del mismo tipo, un arreglo es una estructura homogénea.

Los elementos pueden recuperarse en cualquier orden, simplemente indicando la posición que ocupa dentro de la estructura, esto indica que el arreglo es una estructura indexada.

El operador de acceso es el operador []

La memoria ocupada a lo largo de la ejecución del programa es fija, por esto es una estructura estática.

El nombre del arreglo se socia a un área de memoria fija y consecutiva del tamaño especificado en la declaración.

El índice debe ser de tipo ordinal. El valor del índice puede verse como el desplazamiento respecto de la posición inicial del arreglo.

Los arreglos pueden ser de varias dimensiones. Esta dimensión indica la cantidad de índices necesarias para acceder a un elemento del arreglo.

El arreglo lineal, con un índice, o una dimensión se llama vector.

El arreglo con 2 o mas índices o dimensiones es una matriz. Un grupo de elementos homogéneo con un orden interno en el que se necesitan 2 o mas índices para referenciar a un elemento de la estructura.

iii. **Archivos**: Estructura de datos con almacenamiento físico en memoria secundaria o disco.

Las acciones generales vinculadas con archivos son

Asignar, abrir, crear, cerrar, leer, grabar, Cantidad de elementos, Posición del puntero, Acceder a una posición determinada, marca de final del archivo, definiciones y declaraciones de variables.

Según su organización pueden ser secuenciales, indexados.

1. **Archivos de texto**: Secuencia de líneas compuestas por cero uno o mas caracteres que finalizan con un carácter especial que indica el final de la línea. Los datos internos son representados en caracteres, son mas portables y en general mas extensos.
2. **Archivos de tipo o binarios**: secuencia de bytes en su representación interna sin interpretar. Son reconocidos como iguales si son leídos de la forma en que fueron escritos. Son menos portables y menos extensos.

2. **Dinámicos:** Ocupan direcciones de memoria en tiempo de ejecución y se instancian a través de punteros. Esta s instancias pueden también liberarse en tiempo de ejecución. El tema de puntadores y estructuras enlazadas (estructuras relacionadas con este tipo de dato se analizan en detalle en capítulos siguentes)
    a. **Listas simplemente enlazadas**: cada elemento sólo dispone de un puntero, que apuntará al siguiente elemento de la lista o valdrá NULL si es el último elemento.
    b. Pilas: son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out: el último en entrar es el primero en salir). Los elementos se "amontonan" o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila**.**

c. **Colas**: otro tipo de listas, conocidas como listas FIFO (First In, First Out: El primero en entrar es el primero en salir). Los elementos se almacenan en fila, pero sólo pueden añadirse por un extremo y leerse por el otro.
d. **Listas circulares**: o listas cerradas, son parecidas a las listas abiertas, pero el último elemento apunta al primero. De hecho, en las listas circulares no puede hablarse de "primero" ni de "último". Cualquier nodo puede ser el nodo de entrada y salida.
e. **Listas doblemente enlazad**as: cada elemento dispone de dos punteros, uno a punta al siguiente elemento y el otro al elemento anterior. Al contrario que las listas abiertas anteriores, estas listas pueden recorrerse en los dos sentidos.
f. **Árboles**: cada elemento dispone de dos o más punteros, pero las referencias nunca son a elementos anteriores, de modo que la estructura se ramifica y crece igual que un árbol.
g. **Árboles binarios**: son árboles donde cada nodo sólo puede apuntar a dos nodos.
h. **Árboles binarios de búsqueda** (ABB): son árboles binarios ordenados. Desde cada nodo todos los nodos de una rama serán mayores, según la norma que se haya seguido para ordenar el árbol, y los de la otra rama serán menores.
i. **Árboles AVL**: son también árboles de búsqueda, pero su estructura está más optimizada para reducir los tiempos de búsqueda.
j. **Árboles B**: son estructuras más complejas, aunque también se trata de árboles de búsqueda, están mucho más optimizados que los anteriores.
k. **Tablas HASH**: son estructuras auxiliares para ordenar listas.
l. **Grafos**: es el siguiente nivel de complejidad, podemos considerar estas estructuras como árboles no jerarquizados.
m. **Diccionarios**.

Estructura tipo Registro

Registro: Es un conjunto de valores que tiene las siguientes características:
Los valores pueden ser de tipo distinto. Es una estructura heterogénea.
Los valores almacenados se llaman campos, cada uno de ellos tiene un identificador y pueden ser accedidos individualmente.
El operador de acceso a cada miembro de un registro es l operador punto (.)
El almacenamiento es fijo.
Declaracion
Genérica
NombreDelTipo = TIPO < $TipoDato_1$ $Identificador_1$; …; $TipoDato_N$ $Identificador_N$>

TipoRegistro = TIPO <Entero N; Real Y> //declara un tipo
TipoRegistro Registro; // define una variable del tipo declarado
En C

```
 struct NombreTipo {
        Tipo Identificador;
        Tipo Identificador;
}
struct TipoRegistro {
        int N;
        double Y;
};                      //      declara un tipo
```

```
TipoRegistro Registro; //        define una variable
```

Ejemplo de estructuras anidadas en C

```
struct TipoFecha {
        int     D;
        int     M;
        int     A;
};                      //      declara un tipo fecha

struct TipoAlumno {
        int             Legajo;
        string          Nombre;
        TipoFecha       Fecha
};                      //      declara un tipo Alumno con un campo de tipo Fecha
TipoAlumno Alumno;
```

Alumno es un registro con tres miembros (campos) uno de los cuales es un registro de TipoFecha. El acceso es:

Nombre	Tipo dato	
Alumno	Registro	Registro total del alumno
Alumno.Legajo	Entero	Campo legajo del registro alumno que es un entero
Alumno.Nombre	Cadena	Campo nombre del registro alumno que es una cadena
Alumno.Fecha	Registro	Campo fecha del registro alumno que es un registro
Alumno.Fecha.D	Entero	Campo dia del registro fecha que es un entero
Alumno.Fecha.M	Entero	Campo mes del registro fecha que es un entero
Alumno.fecha.A	Entero	Campo anio del registro alumno que es un entero

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[1]

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.

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.

[1] En Algoritmos este año trabajaremos con estructuras FILE *

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 array y 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 se muestra una tabla comparativa de arreglos y archivos, señalando algunas propiedades distintivas de cada una de ellas.

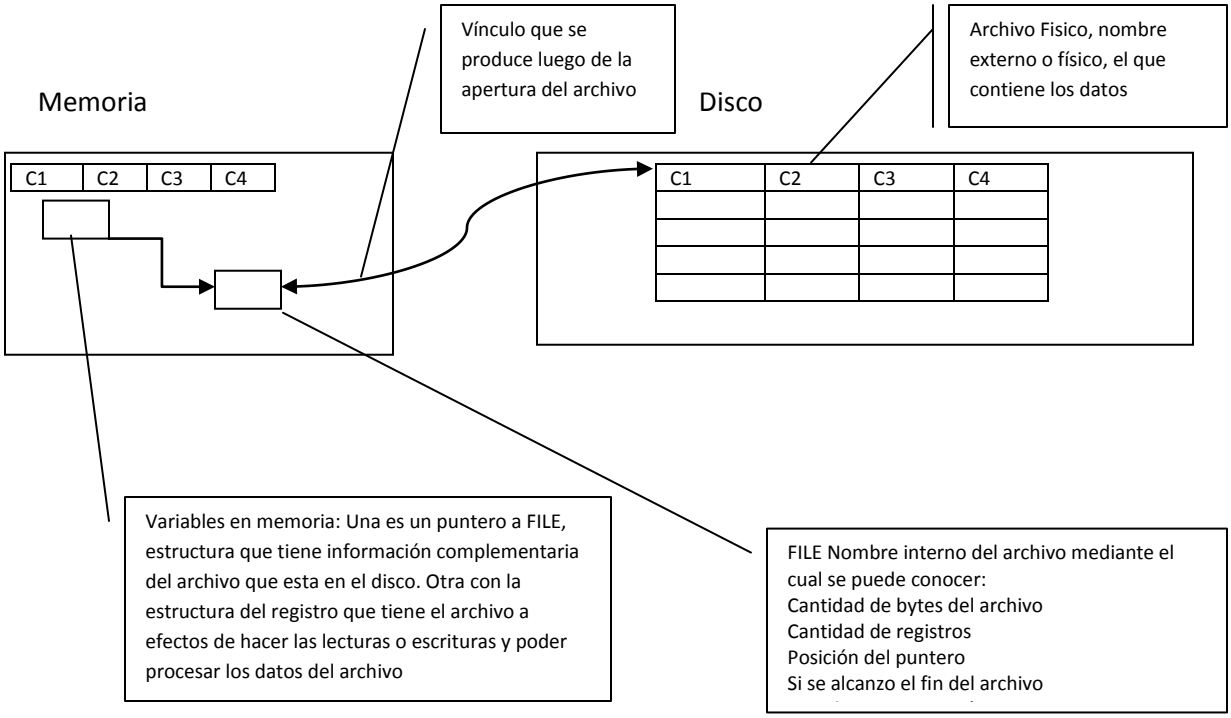
Análisis comparativo entre arreglos y archivos de registro de tamaño fijo		
Propiedad	Arreglo	Archivo
Tamaño Físico en Tiempo Ejecución	Fijo	Variable
Almacenamiento	Electrónico	Físico
- Persistencia	No	Si
- Procesamiento	Rápido	Lento
Búsquedas		
- Directa	Si	Si
- Binaria	Si (si esta ordenado)	Si
- Secuencial	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, esta estructura las veremos en el próximo modulo. 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. Archivos, arreglos y estructuras enlazadas.

La estructura de tipo archivo, como los arreglos y estructuras enlazadas permiten recorridos, búsquedas, carga, eliminación, ordenamientos. Solo cambia la forma de acceso a cada miembro de la estructura. En el caso de los archivos, por distintas razones, las acciones son mas limitadas dado que cosas patrones como búsquedas secuenciales, carga sin repetición, métodos de ordenamiento, se desestiman en esta estructura por lo costoso del procesamiento cuando el almacenamiento es físico (en el disco)

Esquematicamente se puede representar el concepto de archivos de registros como sigue:



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

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
```

Implementacione en C/C++

2

Operaciones sobre archivos

Este documento resume las principales operaciones que son generalmente utilizadas para la manipulación de archivos. Cubre el uso de todas las funciones primitivas que provee el lenguaje C y explica también como desarrollar *templates* que faciliten el uso de dichas funciones.

Las funciones que analizaremos en este documento 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.

Todas estas funciones están declaradas en el archivo `stdio.h` por lo que para utilizarlas debemos agregar la siguiente línea a nuestros programas:

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

Además, basándonos en las anteriores desarrollaremos las siguientes funciones que nos permitirán operar con archivos de registros:

- `seek` – Mueve el indicador de posición de un archivo al inicio de un determinado registro.
- `fileSize` – Indica cuantos registros tiene un archivo.
- `filePos` – Retorna el número de registro que está siendo apuntado por el indicador de posición.
- `read` – Lee un registro del archivo.
- `write` – 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","w+b");

   char c = 'A';
   fwrite(&c,sizeof(char),1,arch);
   // grabo el caracter 'A' contenido en c

   c = 'B';
   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 // cierro el archivo
   fclose(arch);

   return 0;
}
```

La función `fwrite` recibe los siguientes parámetros:

- Un puntero al *buffer* (variable) que contiene el datos que se van a grabar en el archivo.
- El tamaño (en bytes) del tipo de dato de dicho *buffer*; lo obtenemos con la función: `sizeof`.
- La cantidad de unidades del tamaño indicado más arriba que queremos escribir; en nuestro caso: 1.
- Finalmente, el archivo en donde grabará el contenido almacenado en el *buffer*.

La función `fopen` recibe los siguientes parámetros:

- Una cadena indicando el nombre físico del archivo que se quiere abrir.
- Una cadena indicando la modalidad de apertura; `"w+b"` indica que queremos crear el archivo si es que aún no existe o bien, si existe, que queremos dejarlo vacío.

La función `fclose` cierra el archivo que recibe cómo parámetro.

Leer un archivo de caracteres

```
#include <iostream>
#include <stdio.h>

using namespace std;

int main()
{ // abro el archivo para lectura
   FILE* arch = fopen("DEMO.DAT","r+b");

   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;
}
```

La función `fread` recibe exactamente los mismos parámetros que `fwrite`. Respecto de la función `fopen`, en este caso la modalidad de apertura que utilizamos es: `"r+b"` para indicarle que el archivo ya existe y que no queremos vaciar su contenido; solo queremos leerlo.

Respecto de la función `feof` retorna `true` o `false` según si se llegó al final del archivo o no.

Archivo de registros

Las mismas funciones que analizamos en el apartado anterior nos permitirán operar con archivos de estructuras o archivos de registros. La única consideración que debemos tener en cuenta es que la estructura que vamos a grabar en el archivo no debe tener campos de tipos `string`. En su lugar debemos utilizar *arrays* de caracteres, al más puro estilo C.

Veamos un ejemplo:

```
struct Persona
{
   int dni;
   char nombre[25];
   double altura;
};
```

Grabar un archivo de registros

El siguiente programa lee por consola los datos de diferentes personas y los graba en un archivo de estructuras `Persona`.

```
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

int main()
{
   FILE* f = fopen("PERSONAS.DAT","w+b");

   int dni;
   string nom;
   double altura;
   // el usuario ingresa los datos de una persona
   cout << "Ingrese dni, nombre, altura: ";
   cin >> dni;
   cin >> nom;
   cin >> altura;
```

```
   while( dni>0 )
   {
      // armo una estructura para grabar en el archivo
      Persona p;
      p.dni = dni;
      strcpy(p.nombre,nom.c_str()); // la cadena hay que copiarla con strcpy
      p.altura = altura;

      fwrite(&p,sizeof(Persona),1,f); // grabo la estructura en el archivo
      cout << "Ingrese dni, nombre, altura: ";
      cin >> dni;
      cin >> nom;
      cin >> altura;
   }

   fclose(f);

   return 0;
}
```

En este programa utilizamos el método `c_str` que proveen los objetos `string` de C++ para obtener una cadena de tipo `char*` (de C) tal que podamos pasarla cómo parámetro a la función `strcpy` y así copiar el nombre que ingresó el usuario en la variable `nom` al campo `nombre` de la estructura `p`.

Leer un archivo de registros

A 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("PERSONAS.DAT","r+b");
   Persona p;
   // leo el primer registro
   fread(&p,sizeof(Persona),1,f);
   while( !feof(f) )
   {
      // muestro cada campo de la estructura
      cout << p.dni << ", " << p.nombre <<", " << p.altura << endl;
      // leo el siguiente registro
      fread(&p,sizeof(Persona),1,f);
   }

   fclose(f);

   return 0;
}
```

Acceso directo a los registros de un archivo

La función `fseek` que provee C/C++ permite mover el indicador de posición del archivo hacia un determinado byte. El problema surge cuando queremos que dicho indicador se desplace hacia el

primer byte del registro ubicado en una determinada posición. En este caso la responsabilidad de calcular el número de byte que corresponde a dicha posición será nuestra. Lo podemos calcular de la siguiente manera:

```
void seek(FILE* arch, int recSize, int n)
{

   // SEEK_SET indica que la posicion n es absoluta respecto del inicio del archivo
   fseek(arch, n*recSize,SEEK_SET);
}
```

El parámetro `recSize` será el `sizeof` del tipo de dato de los registros que contiene el archivo. En el siguiente ejemplo accedemos directamente al tercer registro del archivo `PERSONAS.DAT` y mostramos su contenido.

```
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

int main()
{
   FILE* f = fopen("PERSONAS.DAT","r+b");
   Persona p;
   // muevo el indicador de posicion hacia el 3er registro (contando desde 0)
   seek(f,sizeof(Persona), 2);
   // leo el registro apuntado por el indicador de posicion
   fread(&p,sizeof(Persona),1,f);
   // muestro cada campo de la estructura leida
   cout << p.dni << ", " << p.nombre <<", " << p.altura << endl;

   return 0;
}
```

Cantidad de registros de un archivo

En C/C++ no existe una función comparable a `fileSize` de Pascal que nos permita conocer cuántos registros tiene un archivo. Sin embargo podemos programarla nosotros mismos utilizando las funciones `fseek` y `ftell`.

```
long fileSize(FILE* f, int recSize)
{
   // tomo la posicion actual
   long curr=ftell(f);

   // muevo el puntero al final del archivo
   fseek(f,0,SEEK_END);  // SEEK_END hace referencia al final del archivo
   // tomo la posicion actual (ubicado al final)
   long ultimo=ftell(f);
   // vuelvo a donde estaba al principio
   fseek(f,curr,SEEK_SET);

   return ultimo/recSize;
}
```

Probamos ahora las funciones `seek` y `fileSize`.

```
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

int main()
{


   FILE* f = fopen("PERSONAS.DAT","r+b");
   Persona p;
   // cantidad de registros del archivo
   long cant = fileSize(f,sizeof(Persona));

   for(int i=cant-1; i>=0; i--)
   {
      // acceso directo al i-esimo registro del archivo
      seek(f,sizeof(Persona),i);
      // leo el registro apuntado por el indicador de posicion
      fread(&p,sizeof(Persona),1,f);
      // muestro el registro leido
      cout << p.dni <<", "<<p.nombre<<", "<< p.altura << endl;
   }

   fclose(f);

   return 0;
}
```

Identificar el registro que está siendo apuntando por el identificador de posición del archivo

Dado un archivo y el tamaño de sus registros podemos escribir una función que indique cual será el próximo registro será afectado luego de realizar la próxima lectura o escritura. A esta función la llamaremos: `filePos`.

```
long filePos(FILE* arch, int recSize)
{
   return ftell(arch)/recSize;
}
```

Plantillas para archivos

3

Templates

Como podemos ver, las funciones `fread` y `fwrite`, y las funciones `seek` y `fileSize` que desa-rrollamos más arriba realizan su tarea en función del `sizeof` del tipo de dato del valor que vamos a leer o a escribir en el archivo. Por esto, podemos parametrizar dicho tipo de dato mediante un *template* lo que nos permitirá simplificar dramáticamente el uso de todas estas funciones.

Template: read

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

Template: write

```
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);
}
```

Template: fileSize

```
template <typename T> long fileSize(FILE* f)
{
   // tomo la posicion actual
     long curr=ftell(f);
   // muevo el puntero al final del archivo
     fseek(f,0,SEEK_END);  // SEEK_END hace referencia al final del archivo
  // tomo la posicion actual (ubicado al final)
```

```
  long ultimo=ftell(f);
// vuelvo a donde estaba al principio
  fseek(f,curr,SEEK SET);

  return ultimo/sizeof(T);
}
```

Template: filePos

```
template <typename T> long filePos(FILE* arch)
{
   return ftell(arch)/sizeof(T);
}
```

Template: busquedaBinaria

El algoritmo de la búsqueda binaria puede aplicarse perfectamente para emprender búsquedas sobre los registros de un archivo siempre y cuando estos se encuentren ordenados.

Recordemos que en cada iteración este algoritmo permite descartar el 50% de los datos; por esto, en el peor de los casos, buscar un valor dentro de un archivo puede insumir $\log_2(n)$ accesos a disco siendo n la cantidad de registros del archivo.

```
template <typename T, typename K>
int busquedaBinaria(FILE* f, K v, int (*criterio)(T,K))
{  // indice que apunta al primer registro
   int i = 0;
   // indice que apunta al ultimo registro
   int j = fileSize<T>(f)-1;
   // calculo el indice promedio y posiciono el indicador de posicion
   int k = (i+j)/2;
   seek<T>(f,k);
   // leo el registro que se ubica en el medio, entre i y j
   T r = leerArchivo<T>(f);

   while( i<=j && criterio(r,v)!=0 )
   {
     // si lo que encontre es mayor que lo que busco...
      if( criterio(r,v)>0 )
      {
         j = k-1;
      }
      else
      {
       // si lo que encontre es menor que lo que busco...
         if( criterio(r,v)<0 )
         {
            i=k+1;
         }
      }
      // vuelvo a calcular el indice promedio entre i y j
      k = (i+j)/2;
     // posiciono y leo el registro indicado por k
      seek<T>(f,k);
// leo el registro que se ubica en la posicion k
      r = leerArchivo<T>(f);
   }
// si no se cruzaron los indices => encontre lo que busco en la posicion k
   return i<=j?k:-1;
}
```

Ejemplos

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

```
f = fopen("PERSONAS.DAT","r+b");
// leo el primer registro
Persona p = read<Persona>(f);
while( !feof(f) )
{
   cout << p.dni<<", "<<p.nombre<<", "<<p.altura << endl;
   p = read<Persona>(f);
}
fclose(f);
```

Escribir registros en un archivo usando el *template* `write`.

```
f = fopen("PERSONAS.DAT","w+b");
// armo el registro
Persona p;
p.dni = 10;
strcpy(p.nombre,"Juan");
p.altura = 1.70;
// escribo el registro
write<Persona>(f,p);
fclose(f);
```

Acceso directo a los registros de un archivo usando los templates `fileSize`, `seek` y `read`.

```
f = fopen("PERSONAS.DAT","r+b");
// cantidad de registros del archivo
long cant = fileSize<Persona>(f);

for(int i=cant-1; i>=0; i--)
{
  // acceso directo al i-esimo registro del archivo
   seek<Persona>(f,i);

   Persona p = read<Persona>(f);
   cout << p.dni<<", "<<r.nombre<<", "<< r.altura << endl;
}

fclose(f);
```

4 Introducción a Streams en C++

Lectura y Escritura en Archivos de Bloques a través de Flujos

El lenguaje C++ y su biblioteca estándar proveen abstracciones para el manejo de flujos (streams) que se conectan a archivos (files). El encabezado que declara estas abstracciones es <fstream>, de file-stream.

Esta sección presenta como

- Crear un flujo desde o hacia un archivo.
- Leer y escribir a través de un flujo
- Manejar los indicadores de posición de lectura y escritura de un flujo
- Cerrar un flujo

Archivos de Bloques de Tamaño Constante

Una de las formas de almacenar datos en archivos es mediante una secuencia de bloques, donde todos los bloques tienen la misma cantidad de bytes. Un bloque de bytes puede almacenar cualquier valor que necesitemos, pero en esta sección asumiremos que un bloque tiene un solo registro representado por un struct.

Por ejemplo, para almacenar en un archivo las temperaturas registradas en distintos puntos de una superficie, podemos diseñar un bloque que contenga la abscisa, la ordenada, y la temperatura registrada. Así, el archivo contendría una secuencia bloques, todos con tres datos reales, y, por lo tanto, del mismo tamaño.

Los bloques los declaramos como structs, y para conocer el tamaño en bytes de un struct aplicamos el operador sizeof.

Para presentar el procesamiento de un archivo de bloques de tamaño constantes, vamos a ver un programa simple que mediante flujos escribe datos, lee datos, y reestablece el indicador de posición de lectura para manejar los datos de curso universitarios.

La directiva

```
#include <fstream>
```

incluye las declaraciones necesarias para manejar las abstracciones de flujo de archivos.

Luego declaramos una estructura que establece la forma de los bloques que vamos a leer y escribir, y creamos una variable en base a esa estructura.

```
struct Curso {
      char especialidad;
      int codigo;
      int nivel;
      int alumnos;
      double promedio;
} curso;
```

El programa crea un nuevo archivo cursos, donde escribiremos cuatro cursos ejemplos. Para eso, necesitamos declarar una variable del tipo ofstream (output-file-stream) que nos va a permitir escribir bloques.

```
ofstream out("cursos", ios::binary);
```

El nombre de la variable es out, el nombre del archivo es cursos, y el modo binary asegura que la cantidad de bytes escritos sea constante. La variable out queda inicializada es el flujo que conecta el programa al archivo cursos.

A continuación, asignamos a los miembros de curso valores ejemplos.

```
curso.especialidad = 'K',
curso.codigo = 1051,
curso.nivel = 1;
curso.alumnos = 29;
curso.promedio = 7.8;
```

Para escribir en el flujo out el bloque curso invocamos a la función writeblock

```
writeblock(out, curso);
```

Luego escribimos otros tres cursos ejemplo. Por último, cerramos el flujo mediante

```
out.close();
```

Ahora volvemos a conectarnos al archivo, esta vez para lectura.

```
ifstream in("cursos", ios::binary);
```

La variable in es del tipo ifstream (intput-file-stream), el cual nos permite leer bloques.

La lectura la realizamos mediante un ciclo que mientras haya bloques en in, lea un bloque y envíe su contenido por cout; en este ejemplo, solo muestra los cursos con más de 25 alumnos. El psueudocódigo es el siguiente:

```
while( haya otro curso en in) )
      if( curso.alumnos > 25 )
            mostrar el curso por cout
```

La expresión

```
readblock(in, curso)
```

lee un bloque desde in y lo almacena en curso, el valor de retorno es in, el cual puede ser utilizado como un valor boolean en la expresión de control de while. El ciclo completo es el siguiente:

```
while( readblock(in, curso) )//al llegar al fin la llamada devuelve el apuntador nulo
          if( curso.alumnos > 25 )
                cout
                      << "Especialidad: " << curso.especialidad << ", "
                      << "      Código: " << curso.codigo       << ", "
                      << "       Nivel: " << curso.nivel        << ", "
                      << "     Alumnos: " << curso.alumnos      << ", "
                      << "    Promedio: " << curso.promedio     << endl;
```

Por último, cerramos el flujo con la sentencia

```
    in.close();
```

Las funciones template writeblock y readblock abstraen y facilitan la lectura de bloques.

Este es el programa completo, incluyendo la definición de writeblock y readblock.

```
/* Escribe y lee archivo de registros
 * Genera un archivo con registros de cursos, luego abre ese archivo y
muestra los cursos de segundo año con cantidad de alumnos mayor a 25.
 * 20130424
 * JMS
 */

#include <iostream>
#include <fstream>

template<typename T>
std::ostream& writeblock(std::ostream& out, const T& block){
    return out.write(
                reinterpret_cast<const char*>(&block),
                sizeof block
          );
}

Conversión entre tipos con reinterpret_cast
Escritura de bytes mediante la función miembro write de ostream: Envia un
número fijo de bytes empezando en una ubicación especifica de memoria al
flujo especificado. La función read de istream recibe un número fijo de
bytes del flujo especificado y los coloca en un area de memoria que
empieza en una dirección especificada. Al escribir el entero numero en un
archivo en lugar de escribir archivoSalida << numero; se puede escribir la
versión binaria
archivoSalida.write(reinterpret_cast<const char *>(&numero), sizeof (numero)

template<typename T>
std::istream& readblock(std::istream& in, T& block){
    return in.read(
                reinterpret_cast<char*>(&block),
                sizeof block
          );
}
```

```
int main(){
    using namespace std;

    struct Curso {
          char especialidad;
          int codigo;
          int nivel;
          int alumnos;
          double promedio;
    } curso;

    ofstream out("cursos", ios::binary);

    curso.especialidad = 'K',
    curso.codigo = 1051,
    curso.nivel = 1;
    curso.alumnos = 29;
    curso.promedio = 7.8;
    writeblock(out, curso);

    curso.especialidad = 'R',
    curso.codigo = 4152,
    curso.nivel = 4;
    curso.alumnos = 41;
    curso.promedio = 6.9;
    writeblock(out, curso);

    curso.especialidad = 'K',
    curso.codigo = 2051,
    curso.nivel = 1;
    curso.alumnos = 22;
    curso.promedio = 6.7;
    writeblock(out, curso);

    curso.especialidad = 'K',
    curso.codigo = 2011,
    curso.nivel = 1;
    curso.alumnos = 26;
    curso.promedio = 7.9;
    writeblock(out, curso);

    out.close();

    ifstream in("cursos", ios::binary);


    while( readblock(in, curso) )
          if( curso.alumnos > 25 )
                cout
                      << "Especialidad: " << curso.especialidad << ", "
                      << "      Código: " << curso.codigo       << ", "
                      << "       Nivel: " << curso.nivel        << ", "
                      << "     Alumnos: " << curso.alumnos      << ", "
                      << "    Promedio: " << curso.promedio     << endl;
```

```
    in.close();
}

Salida:
Especialidad: K,       Codigo: 1051,        Nivel: 1,      Alumnos: 29,
Promedio: 7.8
Especialidad: R,       Codigo: 4152,        Nivel: 4,      Alumnos: 41,
Promedio: 6.9
Especialidad: K,       Codigo: 2011,        Nivel: 1,      Alumnos: 26,
Promedio: 7.9
```

Funciones miembro para posicionar el puntero
seekg (obtener desde donde se hara la próxima entrada, para istream), seekp (colocar, el byte donde se colocara la próxima salida para ostream), tellg, tellp (para determinar las actuales posiciones del puntero obtener y colocar). Se puede indicar un segundo argumento con la dirección de búsqueda que puede ser ios::beg (opción predeterminada desde el inicio) ios::cur para un posicionamiento relativo a la posición actual. ios::end para un posicionamiento relativo al final.
archivo.seekg(n); // desplaza n bytes desde el inicio
archivo.seekg(n, ios::beg); // idem anterior
archivo.seekg(n, ios::end); //se posiciona n bytes hacia atras desde el fin
archivo.seekg(n, ios::cur); se desplaza n bytes desde la posición actual

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;
}
```

Funciones miembros de la clase stream

Funcion	Descripcion
bad	true si ha ocurrido un error
clear	limpia las banderas de estado (status flags)
close	cierra un stream
eof	true si se alcanzó el fin de archivo
fail	true si ha ocurrido un error
fill	establecer manipulador de carácter de relleno
flags	accesa o manipula las banderas de formato de un stream
flush	vaciar el buffer de un stream
gcount	número de caracteres leidos durante la última operación de entrada
get	lectura de caracteres
getline	lectura de una línea de caracteres
good	true si no ha ocurrido un error
ignore	leer y descartar caracteres
open	abrir un stream de entrada y/o salida
peek	verifica la siguiente entrada de carácter
precision	manipula la precisión del stream
put	escritura de caracteres
putback	regresar caracteres al stream
rdstate	regresa la bandera de estado de stream
read	lee datos de un stream hacia un buffer
seekg	realiza acceso aleatorio sobre un stream de entrada
seekp	realiza acceso aleatorio sobre un stream de salida
setf	cambiar las banderas de formato
tellg	lee el puntero del stream de entrada
tellp	lee el puntero del stream de salida
unsetf	limpiar las banderas de formato
width	accesa y manipula la longitud minima del campo
write	escritura datos desde un buffer hacia un stream

Consideraciones sobre stream.

El ciclo

```
while (!arch1.eof () && !arch2.eof ()) {… };
```

En forma mas compacta se puede escribir:

```
while(arch1 and  arch2){…}
```

eof no es la única condición de fin, además, la notación anterior induce a que se asuma que hay una marca eof. Por otro lado, C++ permite usar streams como booleans, el stream es verdadero si se puede seguir leyendo o falso en caso contrario. Lo que nos conduce a la segunda forma de escritura más compacta y comprensible.

Ejemplo de corte de control

Ejemplo de control break sobre un campo de control.
Entrada 3-uplas (pedido, cliente, importe), ordenadas por cliente. Ejemplo:
20 78 10.5; 51 78 9.5; 12 78 5.5; 26 80 1.5; 63 80 20.5
Salida 2-uplas (cliente, total), ordenandas por cliente; y total general. Ejemplo:
78 25.5; 80 22

```
// programa que opera sobre stdin
#include <iostream>
int main() {
struct { unsigned id, cliente; double importe;} pedido;

double totalGeneral=0;
std::cin >> pedido.id >> pedido.cliente >> pedido.importe;
while (std::cin) {
  unsigned previo = pedido.cliente;
  double total = pedido.importe;
   while (std::cin >> pedido.id >> pedido.cliente >> pedido.importe and pedido.cliente==previo) {
     total+=pedido.importe;
   }
  std::cout << previo << '\t' << total <<'\n';
  totalGeneral+=total;
}
std::cout << totalGeneral <<'\n';
return 0;
}
```

```
// programa que opera sobre un flujo binario
#include <iostream>
#include <fstream>
int main() {
struct { unsigned id, cliente; double importe;} pedido;
ifstream flectura("prueba.dat",ios::in | ios::binary);
double totalGeneral=0;
readblock(flectura, pedido);
while (not flectura) {
  unsigned previo = pedido.cliente;
  double total = pedido.importe;
   while (readblock(flectura, pedido) and pedido.cliente==previo) {
     total+=pedido.importe;
   }
   std::cout << previo << '\t' << total <<'\n';
   totalGeneral+=total;
 }
std::cout << totalGeneral <<'\n';
flectura.close();
return 0;
}
```

Ejemplo de corte de Apareo

Ejemplo de apareo sobre un campo, sin repetición, por el que esta ordenado.
Entrada 3-uplas (pedido, cliente, importe), ordenadas por cliente.

```
// programa que opera sobre flujos binarios
#include <iostream>
#include <fstream>
int main() {
struct { unsigned id, cliente; double importe;} pedido1,pedido2;
ifstream flectura1("prueba1.dat",ios::in | ios::binary);
ifstream flectura2("prueba2.dat",ios::in | ios::binary);
ofstream fmezcla("prueba.dat",ios::out | ios::binary);
readblock(flectura1, pedido1);
readblock(flectura2, pedido2);
while (flectura1 or flectura2) {
   if(!flectura2 or (flectura1 and pedido1.cliente < pedido2.cliente))
   //si el segundo no tiene  o teniendo ambos el primero es menos
    {
      writeblock(mezcla, pedido1);
      readblock(flectura1, pedido1);
   } else
       {
          writeblock(mezcla, pedido2);
          readblock(flectura2, pedido2);
       }
 }
std::cout << totalGeneral <<'\n';
flectura1.close();
flectura2.close();
fmezcla.close();
return 0;
}
```

<table>
<tr><th>FILE *</th><th>streams</th></tr>
<tr><td>Archivos cabecera
#include <stdio.h></td><td>#include <iostream>
#include <fstream></td></tr>
<tr><td>Abrir un archivo
Texto lectura
FILE * f = fopen (“archivo”, “r”);
Texto escritura
FILE * f = fopen (“archivo”, “w”);
Binario lectura
FILE * f = fopen (“archivo”, “rb”);
Binario escritura
FILE * f = fopen (“archivo”, “wb”);</td><td>Ifstream f("archivo",ios::in);

ofstream f("archivo",ios::out);

Ifstream f("archivo",ios::in|| ios::binary);

ofstream f("archivo",ios::out|| ios::binary);</td></tr>
<tr><td>Leer por bloque(restringido a 1 bloque de 1 registro)
fread(&r, sizeof(r), 1, f);

Plantilla Lectura
<pre>template <typename T> T read(FILE* f)
{
   T buff;
   fread(&buff,sizeof(T),1,f);
   return buff;
}</pre></td><td>f.read(reinterpret_cast<char*>(&r), sizeof r;

<pre>template<typename T>
std::istream& readblock(std::istream&
in, T& block){
    return in.read(

    reinterpret_cast<char*>(&block),
                sizeof block
          );
}</pre></td></tr>
<tr><td>Grabar por bloque
fwrite(r, sizeof(r),1,f)

Plantilla gabar
<pre>template <typename T> void write(FILE*
f, T v)
{
   fwrite(&v,sizeof(T),1,f);
   return;
}</pre></td><td>out.write(reinterpret_cast<const char*>(&r),
sizeof r)

<pre>template<typename T>
std::ostream& writeblock(std::ostream&
out, const T& block){
    return out.write(

    reinterpret_cast<const
char*>(&block),
                sizeof block
          );
}</pre></td></tr>
</table>