Universidad
Inca Garcilaso de la Vega
Nuevos Tiempos. Nuevas Ideas

Facultad de Ingeniería de Sistemas, Cómputo y Telecomunicaciones
Sistema a Distancia

ESTRUCTURA DE INFORMACIÓN

CARLOS RUIZ DE LA CRUZ MELO
FRANKLIN ARRIOLA RAMÍREZ
YULIANA JÁUREGUI ROSAS

2010

Introducción

El presente libro abarca los temas que comprende el curso de Estructura de Información correspondiente al cuarto ciclo de la carrera de Ingeniería de Sistemas y Cómputo, donde se desarrolla las diferentes estructuras de datos tanto estáticas como dinámicas usadas en los programas computaciones; dichas estructuras van a permitir acceder y representar de forma más organizada los conjuntos de datos.

Los ejemplos prácticos de estas estructuras se han realizado mediante pseudocódigos e implementado en el lenguaje C++, elegido por su practicidad y debido a que el alumno tiene noción sobre este lenguaje empleado en el curso de Lenguaje de programación I.

Cabe mencionar que el uso adecuado de estas estructuras va a aumentar considerablemente la productividad del programa reduciendo el tiempo requerido para escribir, verificar, depurar y mantener los datos que manejan los programas.

Orientación Metodológica

La organización del libro se inicia con una introducción en el que se indica el ciclo al que pertenece el curso, la herramienta a utilizar y la importancia de elaborar programas empleando estructuras de organización de datos.

A continuación se desarrollan las unidades temáticas que de acuerdo al sílabo se constituyen en cinco, cada una de estas unidades a su vez se dividen en lecciones, en la que en cada lección se desarrollan bases teóricas necesarias para entender el tema, así como ejercicios resueltos, de esta manera, la primera unidad introduce a la descripción de tipos abstractos de datos y estructuras secuenciales; en la segunda unidad se desarrolla las diferentes operaciones con estructuras de datos lineales estáticas como son los arreglos; en la tercera unidad se aborda el manejo de las estructuras de datos lineales dinámicas; en la cuarta unidad, se desarrolla lo referente a pilas y colas; finalmente en la quinta unidad se trabaja con estructuras de datos no lineales.

Al término de cada unidad se presenta un resumen, una lectura, una autoevaluación así como los enlaces web y la bibliografía correspondiente que servirá para que el alumno amplié sus conocimientos.

El curso de Estructura de Información tiene como objetivo dar a conocer los mecanismos abstracción para describir las diferentes formas de almacenar los datos en los dispositivos de almacenamiento, de tal forma que puedan ser almacenados y posteriormente recuperados.

Para esto, como estrategias de aprendizaje se presenta el desarrollo de varios ejercicios tipos que sirven de referencia para el desarrollo de otros ejercicios.

Las tutorías son realizadas de forma presencial y virtual, en este último caso por medio de la plataforma se brinda al alumno apoyo para la resolución de ejercicios, así como algunos videos en el que se explica el proceso de cada estructura que se desarrolla en el curso. En este sentido la evaluación se constituye en base a pseudocódigos resueltos mediante los cuales el alumno identificará el proceso de obtención de un resultado.

Primera Unidad

Tipos abstractos de datos y estructuras secuenciales

Sumario

Comprende y sintetiza los conceptos de estructura secuencial, en el cual los datos se registran consecutivamente. Desarrolla los criterios de acceso en una organización secuencial, así como el uso y aplicación de tales conceptos en una situación real.

- Estructura de datos
- Filas

Lección 1

Estructura de datos

Cuando programamos hacemos uso de datos elementales como enteros, reales, caracteres, etc., los cuales tienen una serie de operaciones ligadas a su tipo de dato, tal es así que sobre los datos de tipo entero podemos aplicar operaciones de suma, resta, multiplicación, etc., sin embargo cuando hablamos de estructura de datos nos extendemos un poco más y nos referimos a la manera de disponer de un conjunto de datos de distinto tipo, sobre los cuales se define y aplica un conjunto también de operaciones o métodos con los cuales podemos acceder a ese conjunto de datos que hemos asociado e interrelacionado con los métodos. "Las estructuras de datos son las formas de representación interna de datos de la computadora, mediante las que se representa cualquier situación en la computadora, es decir, son los tipos de datos que maneja la maquina." (López, 2006, p. 36).

1.1. Nociones preliminares para la construcción de algoritmos

Un algoritmo es definido como una secuencia finita de acciones para llevar a cabo una tarea específica, como la solución de un problema en un número finito de pasos. Esto es, que la ejecución del algoritmo concluye en algún momento.

Existen varias maneras para representar algoritmos, así, podemos considerar: diagramas de flujo, Diagramas Nassi-Schneiderman, pseudocódigo.

1.1.1. Pseudocódigo

Es una mezcla de los lenguajes naturales (español, inglés, etc.) y los lenguajes formales (lenguajes de programación), que nos permite representar la secuencia lógica de las acciones de un algoritmo.

La ventaja del pseudocódigo es que en la solución de un programa, el programador se concentra en la lógica sin preocuparse de la rigidez sintáctica que un lenguaje de programación exige.

```
Inicio
        Acción1
        Acción2
          .
          .
          .
        AcciónN
Fin
```

En el pseudocódigo las instrucciones están delimitados por las palabras: Inicio y Fin.

1.1.2. Expresiones

Las expresiones son combinaciones de variables, constantes, operadores, funciones especiales, paréntesis, entre otros, que se forman para representar las operaciones aritméticas, relacionales y lógicas. Por ejemplo:

a + 2 * b – c / 3

Una expresión es una secuencia o concatenación de símbolos tales como operadores y operandos y se pueden clasificar las expresiones como:

- Aritméticas (+, - ,*, /, div, mod, ^)
- Relaciónales (<, >, =, <>,<=, >=)
- Lógicas (and, or, not)

La mayoría de los operadores son binarios, es decir utilizan dos operandos mientras los operadores unarios necesitan solo un operando.

Los operadores binarios utilizan la forma algebraica que conocemos (por ejemplo, a+b), los operadores unarios siempre preceden a sus operandos (por ejemplo, -b).

1.1.3. Identificadores

Un identificador es una secuencia de caracteres con la que se nombran en un programa las entidades definidas por el programador como variables, nombres de constantes, procedimientos, funciones, etc. Para definir un identificador se debe seguir las siguientes reglas:

- Como primer carácter estamos obligados a usar un carácter alfabético, mientras los demás caracteres que podamos usar pueden ser tanto dígitos como caracteres alfabéticos o en tal caso el símbolo de subrayado (_).
- No debe de comenzar con un número, ni debe contener espacios en blanco.
- Se aconseja elegir nombres de identificadores que describan la información que ellos representan, esto facilita la lectura, comprensión y escritura del algoritmo.

1.1.4. Asignación

Reemplaza el valor actual asociado a un identificador con un nuevo valor especificado por un operando, expresión o una nueva expresión cuyo valor lo retorna un método. Se representa por una flecha que apunta hacia la izquierda (←); El formato general de asignación es:

nombreVariable ← Expresión

El antiguo valor de la variable es perdido después de la asignación.

Ejemplo:

```
x ← x+y
y ←  (i > j) && (i <=10)
b ← factorial(a)
```

x ← 3 significa que el valor 3 (a la derecha de la flecha) es colocado en la variable identificador x.

1.1.5. Entrada y Salida de datos

La entrada de datos, permite recibir un valor o dato del exterior (por ejemplo el teclado). Se representa por la primitiva *Leer* la cual, está seguida por el nombre de la variable delimitado entre paréntesis:

Leer (nombre de variables)

Cuando se lee más de una variable debe de ir separado por comas. Por ejemplo para leer las variables a, b, c, se tiene:

Leer (a, b, c)

La salida de datos, nos proporciona un mecanismo para transferir el valor de una variable o un mensaje a un dispositivo de salida (por ejemplo la pantalla, impresora). Se representa por la primitiva *Escribir,* la cual está seguida por el nombre de la variable delimitado entre paréntesis:

Escribir (nombre de variables)

Los mensajes se escriben entre comillas dobles " ". Si una variable es escrita entre comillas, se mostrará el nombre de la variable correspondiente, y si la variable es escrita sin comillas se mostrara el contenido de la variable. Por ejemplo para imprimir el M.C.D. de dos números, se tiene:

Escribir ("El M.C.D es:", m)

1.2. Estructuras básicas de programación

Un programa puede ser escrito utilizando algunas de las siguientes estructuras: secuencial, selectivas y repetitivas.

1.2.1. Estructura secuencial

Se representa por una sentencia ó un conjunto de sentencias, especifica que sus sentencias componentes se ejecutan en la misma secuencia que se escribe. Su sintaxis es:

```
Sentencia1
Sentencia2
:
SentenciaN
```

1.2.2. Estructura selectiva

Las estructuras selectivas se utilizan para tomar decisiones, el tipo de resultado de una estructura selectiva es lógico (booleano), es decir, verdadero o falso**.** Las estructuras selectivas se clasifican en: simples, dobles y múltiples.

1.2.2.1. Estructura selectiva simple: Si - Fin_si

Evalúa una expresión lógica y si su resultado es verdadero (true), se ejecuta una sentencia o un grupo de sentencias. Su sintaxis es la siguiente:

```
Si (expresión_lógica) entonces
      sentencia(s)
Fin_si
```

Por ejemplo si tenemos:

```
Si (promedio>=10.5) entonces
   aprobado ← aprobado +1
Fin_si
total ← total +1
```

- Si el resultado de la expresión lógica (promedio>=10.5) es verdadera, se ejecuta aprobado ← aprobado +1, finalizando la estructura *Si*, luego se ejecuta total ← total+1
- Si el resultado de la expresión lógica (promedio>=10.5) es falsa, no se ejecuta la estructura *Si* y solo se ejecuta total ← total +1

1.2.2.2. Estructura selectiva doble: Si – Sino – Fin_si

Se evalúa la expresión lógica, si este resultado es verdadero se ejecuta la sentencia1, si el resultado es falso se ejecuta la sentencia2. Nunca se ejecutan ambas sentencias, si se ejecuta la sentencia1 ya no se ejecuta la sentencia2 y viceversa. Su sintaxis es la siguiente:

Si (expresión_lógica) **entonces**
sentencia1
Sino
sentencia2
Fin_si

Por ejemplo si tenemos:

```
Si (a>b) entonces
  m ← a
Sino
  m ← b
Fin_si
Escribir (m )
```

- Si el resultado de la expresión lógica (a>b) es verdadera, se ejecuta m←a. finalizando la estructura *Si-Sino,* luego se ejecuta Escibir(m).
- Si el resultado de la expresión lógica (a>b) es falsa, se ejecuta m←b. finalizando la estructura *Si-Sino,* luego se ejecuta Escibir(m).

1.2.2.3. Estructura selectiva múltiple: En caso sea – Fin_caso

Se presenta como una alternativa a las sentencias Si múltiples, ya que en ocasiones resulta ser mas comprensibles y ordenados. Nos permite seleccionar una opción entre un conjunto de opciones de acuerdo al valor de una variable llamada selector. Las sentencias *Si* anidadas pueden aplicarse a cualquier tipo de dato, en contrasté con la sentencia *En caso sea-Fin caso*, en donde el selector solo puede aplicarse a variables: enteras o caracteres. Su sintaxis es la siguiente:

En caso sea (selector) **hacer**
c1: sentencia1
c2: sentencia2
c3: sentencia3

.
.
.
Sino: sentenciaN
Fin_caso

Por ejemplo si tenemos:

```
En caso sea (vocal)
        'a': x←1
        'e': x←2
        'i': x←3
        'o': x←4
        'u': x←5
        Sino:  x←0
Fin_caso
Escribir(x)
```

- Se compara el valor de la variable *vocal* con cada una de las alternativas, según esta comparación, la variable *x* toma un valor específico.
- Así pues, si la variable *vocal* es igual al carácter 'u', se ejecuta x←5, y finaliza la estructura *En-caso sea*, luego el programa ejecuta la sentencia escribir (x)
- Si el valor de la variable *vocal* no coincide con ninguna de las alternativas, se ejecuta la sentencia x ← 0; correspondiente al *sino*, luego el pseudocódigo ejecuta la sentencia Escribir(x).

1.2.2.4. Estructura repetitiva Mientras – Fin_mientras

Repite una o más sentencias mientras el resultado de una expresión lógica es verdadero. Si el resultado de la expresión lógica es falsa, este proceso de repetición termina, y la ejecución del pseudocódigo continúa con la siguiente sentencia que sigue al *Fin_mientras*. Su sintaxis es la siguiente:

```
Mientras (expresión_lógica) hacer
        Sentencia(s)
Fin_mientras
```

Por ejemplo si tenemos:

```
i ←1
Mientras (i<=5) Hacer
        fact ← fact * i
        i ← i+1
Fin_mientras
Escribir(fact)
```

- Se inicializa i ←1, luego se evalúa la expresión lógica (i<=5), mientras el contenido de la variable i es menor o igual a 5 se ejecutan las sentencias que se encuentran entre *Mientras-Fin_mientras*

- Cuando la variable i toma el valor de 6, la expresión lógica (i<=5) se hace falsa terminando la sentencia *Mientras* y ha continuación se ejecuta la sentencia Escribir (fact).

1.2.2.5. Estructura repetitiva Repetir – Hasta_que

Esta sentencia repite una sentencia o más sentencias *hasta* que el resultado de una expresión lógica sea verdadero, momento en que termina la sentencia *Repetir* y se continúa con el resto del pesudocódigo. Su sintaxis es la siguiente:

Repetir
Sentencia(s)
Hasta_que (expresión_lógica)

Por ejemplo si tenemos:

```
i ←1
Repetir
    fact ← fact * i
    i ← i +1
Hasta-que (i =5)
Escribir(fact)
```

- Se inicializa i←1, luego se ejecuta las sentencias que están entre Repetir-Hasta_que, mientras la variable i sea diferente de 5.
- Cuando la variable i toma el valor de 5, la expresión lógica (i=5) es verdadera terminando la sentencia Repetir, y ha continuación se ejecuta la sentencia Escribir (fact).

1.2.2.6. Estructura repetitiva Desde – Fin_desde

La sentencia *Desde*, repite una o más sentencias un número determinado de veces. Se utiliza cuando se conoce por anticipado el número de veces que se van a repetir estas sentencias. Su sintaxis es la siguiente:

Desde (i ←Valor_Inicial) **hasta** (expresión_Lógica) **con** incremento N **hacer**
Sentencia(s)
Fin_desde

Por ejemplo si tenemos:

Desde i ←1 Hasta i<=5 con incremento 1 Hacer

```
        fact ← fact * i
    Fin_desde
    Escribir( fact)
```

- Al ejecutar el *Desde-Fin_desde* se inicializa i←1, luego se evalúa la expresión lógica (i<=5), como es verdadero se ejecuta la sentencia fact ← fact * i, luego la ejecución retorna al Desde, incrementándose el valor de la variable i (i=2), y se evalúa nuevamente la expresión lógica (i<=5).
- Cuando la variable i toma el valor de 6, la expresión lógica (i<=5) se hace falsa terminando la sentencia *Desde-Fin_desde* y ha continuación se ejecuta la sentencia Escribir(fact).

1.3. Función

Se caracteriza porque devuelve un único valor al lugar desde donde se efectuó la llamada, y es necesario especificar el tipo de dato que devuelve (tipo de resultado). Además se debe incluir la palabra retornar seguido del valor devuelto. Así tenemos:

Función nombreDeLaFunción (Lista de parámetros): tipo de resultado
Sentencia(s)
Retornar valor
Fin_función

La llamada a una función se realiza mediante un nombre y una lista de parámetros:

x ← nombreDeLaFunción (lista de parámetros)

1.4. Procedimientos

Se caracteriza porque no se devuelve ningún valor al lugar desde donde se efectuó la llamada, y es por ello que no se especifica ningún tipo de resultado. Como no se devuelve ningún valor, la palabra *retornar* no es necesaria. Así tenemos:

Procedimiento NombreDelProcedimiento (Lista de parámetros)
Sentencia(s)
Fin_procedimiento

Para llamar a un procedimiento basta con escribir su nombre y los correspondientes parámetros si los hubiera. El tratamiento de los parámetros es idéntico al ya mencionado para la función.

Durante la construcción de algunos pseudocódigos se utilizarán las siguientes funciones:

- igual (valor1, valor2) será verdadero si valor1 y valor2 son iguales
- super (valor1, valor2) será verdadero si valor1 es mayor que valor2
- infer (valor1, valor2) será verdadero si valor1 es menor que valor2

Para estas funciones se utiliza las expresiones booleanas

1.5. Tipos de datos

"*Un tipo de datos es una colección de valores*"(Hernandez,2006,3). Es la especificación de un conjunto de valores y de un conjunto válido de operaciones que se asignan sobre una variable. Se distinguen dos tipos de datos:

Tipos De Datos Simples: Son los tipos de datos comunes o básicos como enteros, reales, cadenas alfanuméricas, etc.

Tipos compuestos o tipos abstractos de datos (TAD): Un tipo compuesto permite describir una estructura de datos como un conjunto de variables junto con las operaciones que sobre el se pueden aplicar. "Una herramienta útil para especificar las propiedades lógicas de los tipos de datos es el tipo de datos abstracto o ADT (por sus siglas en ingles), el cual es fundamentalmente una colección de valores y un conjunto de operaciones con esos valores"(Tenenbaum, 1993, p.14).

1.5.1. Caracteristicas de un TAD

- Un TAD es un tipo de dato compuesto definido por el programador que se puede manipular de un modo similar a los tipos de datos simples o elementales definidos por el creador o fabricante en los lenguajes de programación como C++, Pascal, Java, etc.
- Con los TAD se quiere plasmar lo esencial de la realidad sin comprometerse con detalles de implementación (su desarrollo es independiente del lenguaje de programación). De hecho, es posible considerar diferentes implementaciones
- Un TAD es una estructura algebraica, o sea, un conjunto de variables con ciertas operaciones definidas sobre ellos. Piense, por ejemplo, en la calculadora: los elementos que maneja son cantidades numéricas y las operaciones que tiene definidas sobre éstas son las operaciones aritméticas. Otro ejemplo posible es el TAD matriz; los elementos que maneja son las matrices y las operaciones son aquellas que nos permiten crearlas, sumarlas, invertirlas, etc.
- A la hora de utilizar el TAD, la representación debe permanecer oculta. Solo podremos utilizar las operaciones del TAD para trabajar con sus elementos.

1.5.2. Tipos básicos de operaciones en un TAD

Es posible observar que las operaciones de un TAD son de diferentes clases: algunas de ellas nos deben permitir crear entidades nuevas, otras determinar su estado o valor en un instante determinado, unas construir otras entidades a partir de algunas ya existentes, etc. (Tenenbaum, 1993).

Aquí los tipos básicos:

- **Constructores**: Crean una nueva instancia del tipo.
- **Transformación**: Cambian el valor de uno o más elementos de una instancia del tipo.
- **Observación**: Nos permiten observar el valor de uno o varios elementos de una instancia sin modificarlos.
- **Iteradores**: Nos permiten procesar todos los componentes en un TDA de forma secuencial.

1.5.3. Especificación e implementación del TAD

"Una abstracción es una simplificación de un proceso de la realidad en la cual solamente se consideran los aspectos más significativos o relevantes" (Joyanes, 2005, 151). En toda abstracción y en especial cuando hablamos de TAD en los que hay que asociar datos y un conjunto de operaciones, es adecuado diferenciar niveles en su diseño como observamos a continuación:

a) Especificación o definición *¿qué es*? o *¿qué hace*? la abstracción. Esta definición puede ser formal o informal. Por ejemplo: Especificación de una abstracción de operaciones mediante el detalle de los datos de entrada que requieren las operaciones así como el resultado que se espera de ellas (pre o post-condición) expresadas mediante un lenguaje lógico.

b) Implementación o *¿Cómo es*? o *¿Cómo lo hace*? en un determinado entorno informático. Por ejemplo: Implementación de una abstracción de operaciones mediante un programa en C++, Java, u otro lenguaje de programación.

1.5.4. Especificación de un tipo de dato TAD

Una especificación algebraica consta de las siguientes componentes**:**

- **Nombre del tipo:** especifica el nombre del TAD.
- **Usar**: declara si se utiliza otros TAD's previamente definidos.
- **Variable:** son las variables (atributo) que contiene el valor (estado) del TAD.
- **Lista de métodos**: especifica la sintaxis o perfil de los métodos. Se darán como cabecera de procedimiento o función, en la cual se declara sus parámetros y el tipo de valor devuelto.
- **Significado**: es el significado o semántica de los métodos, es decir una función o procedimiento puede tener cualquier comportamiento con tal que respete los parámetros, sus tipos y el tipo de resultado. Con el *significado* se describe el comportamiento de dichas operaciones sobre los datos.

Por ejemplo, podemos describir el TAD de los alumnos, como código, nombre, edad de la siguiente forma:

Especificación ALUMNO
variable
codigo : cadena
nombre : cadena
edad : entero
métodos:
ingresar() : no retorna valor
*visualizar() :*no retorna valor
significado:
ingresar, ingresa los datos del alumno
visualizar, visualiza los datos del *alumno*
Fin_especificación

1.5.5. Ventajas de usar TAD's:

- Independencia de la implementación del tipo.
- Programas más portables, legibles, reutilizables.
- Mayor seguridad de la información, menos efectos colaterales.
- Mayor facilidad para comprobar la corrección.

1.6. Clases y objetos

La idea básica de Programación Orientada a Objetos (POO), es agrupar un conjunto de datos y operaciones para manejarlos como una única entidad que denominamos objeto. El paradigma de la POO conceptúa un programa como un conjunto de objetos que se interrelacionan unos con otros.

Los objetos no son tipos de datos básicos, sino que están formados por otros tipos de datos básicos o simples e inclusive otros objetos y que están correctamente organizados.

1.6.1. Objeto

"Un objeto es simplemente una entidad lógica que contiene datos y un código que manipula estos datos."(Schildt, 1994, p. 617). Un objeto es una entidad que engloba en sí mismo atributos (datos) y métodos (procedimientos o funciones) necesarios para el tratamiento de esos datos. Hasta ahora habíamos hecho programas en los que los datos y las funciones estaban perfectamente separadas, sin embargo cuando se programa con objetos esto no es así, ya que cada objeto contiene atributos y métodos, de tal modo que un programa en la POO esta formado por uno o más objetos que se comunican entre si.

1.6.2. Clase

Una clase se puede considerar como un patrón para construir objetos. Los atributos y los métodos comunes a un conjunto de objetos forman un conjunto que se conoce como clase. "Para crear un objeto

en C++, primero debe definir su forma general usando la palabra reservada class(clase).” (Schildt, 1994, p. 620).

Sintaxis general de una clase en C++

```
class nombreDeLaClase
{
        private:
                Atributos
        public:
                Métodos
};
```

Al conjunto de los métodos y las variables definidas dentro de una clase se le llama *miembros de la clase*, el código está contenido en los métodos.

- Nombre de la clase: sirve para identificar a todos los objetos que tengan una determinada característica
- Atributos: es el dato que se denomina variable miembro de clase. El valor de los atributos, representa el estado de cada objeto.
- Métodos: es la función o procedimiento que se denomina función o procedimiento miembro de clase. Permite que los objetos cambien de estado al ser los únicos que pueden operar sobre los atributos.
- Niveles de acceso: para proteger ciertos miembros de clase. Normalmente, se definirán como ocultos (privados) los atributos y visibles (públicos) los métodos.

Los métodos tienen la misma estructura interna que una función tradicional, con la diferencia que estas funciones estarán asociadas a la clase en la cual se definen, pudiéndose especificar en la clase el prototipo de la función y fuera de la clase su implementación. Se utiliza el operador de “resolución de ámbito” (::)

```
tipo NombreDeLaClase::nombreDelMetodo(Lista de parámetros)
{
   Cuerpo del método;
}
```

- Tipo: Indica el tipo de dato que devuelve el método.
- nombreDelMétodo: Cualquier identificador válido que sea distinto de los que ya están siendo utilizados por otros elementos del programa.
- Lista de parámetros: Conjunto de pares de tipos e identificadores separados por comas. Si el método no tiene parámetros la lista estará vacía.

Por ejemplo se desea tener los datos de los estudiantes con los siguientes datos como código, nombre, edad, y luego visualizarlos en pantalla.

Especificación del TAD

```
Especificación ALUMNO
    variable
        codigo          : cadena
        nombre      : cadena
        edad        : entero
    métodos:
        ingresarDatos()   : no retorna valor
        mostrarDatos()    :no retorna valor
    significado:
        ingresarDatos, ingresa los datos del alumno
        mostrarDatos, visualiza los datos del alumno
Fin_especificación
```

La implementación puede hacerse de la siguiente manera:

Implementación del TAD

```
#include <iostream>
#include <cstdlib>
using namespace std;
class ALUMNO
{
    char codigo[10];
    char nombre[40];
    int  edad;
    public:
        void ingresarDatos()
        {
            cout<<"ingrese codigo:";
            cin>>codigo; fflush(stdin);
            cout<<"ingrese nombre : ";
            cin.get(nombre,40); fflush(stdin);
            cout<<"ingrese edad : ";
            cin>>edad;
        }
        void mostrarDatos()
        {
            cout<<codigo<<"\n";
            cout<<nombre<<"\n";
            cout<<edad;
            cout<<endl;
        }
```

```
};
int main()
{
	ALUMNO miAlumno;
	miAlumno.ingresarDatos();
	miAlumno.mostrarDatos();
	system("PAUSE");
	return 0;
}
```

Lección 2

Filas

El concepto de archivos permite almacenar datos en forma permanente, normalmente en los dispositivos de almacenamiento secundario (discos, cintas, disquetes, usb, etc.).

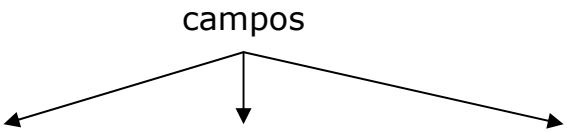
En C/C++ un archivo puede ser un archivo en disco, una impresora, una cinta magnética, una consola, o un teclado. En esta lección nos interesan los archivos en disco como una colección de información que almacenamos en un soporte magnético para poder manipular en cualquier momento. El almacenamiento de datos puede ser de datos simples como de datos compuestos. Cuando almacenamos datos compuestos hablamos de registros en los cuales se aplican dos conceptos:

- **Campo:** es un ítem o elemento de datos elementales, caracterizado por su tamaño o longitud y su tipo de datos (cadena, entero, lógico, etc.)

- **Registro:** es una colección de campos lógicamente relacionados que se almacenan en un archivo en disco en forma de renglones consecutivos.

Ejemplo:

Archivo: alumnos



Código	Nombre	Edad
0950080	Juan carlos, Agreda sanchez	22
0950100	Julio cesar, Castillo rodriguez	23
0951200	Manuel Alejandro, cuba miranda	21
0951233	Ana maria, Montalvo rios	22
0951343	Jose luis, Solis villafani	20

registro

2.1. Tipos de archivos

En general existen dos tipos de archivos:

- **Archivos Secuenciales:** Los datos se almacenan de forma consecutiva y no es posible leer un registro directamente, es decir para leer el registro *n* hay que leer los *n-1* registros anteriores. En este tipo de archivos se utiliza una marca invisible que el sistema operativo coloca al final de los archivos: EOF (End of File), la cual sirve para identificar dónde termina el archivo. Los archivos secuenciales se utilizan con mayor frecuencia en aplicaciones de procesos en lotes.
- **Archivos Directos o Aleatorios o Random:** Permite una mayor rapidez de acceso, al leer o escribir un registro concreto, sin necesidad de leer todos los registros anteriores. Los archivos de acceso directo permiten operaciones tanto de lectura o escritura a cualquier parte del archivo en cualquier momento, como si fueran vectores en memoria. En C/C++ la forma de hacer posible el acceso directo es usando la función *fseek* para situarnos en cualquier punto del archivo y poder efectuar una operación de lectura o escritura.

2.2. Operaciones con archivos

Las operaciones que se pueden realizar con un archivo durante la resolución de un problema son:

- Abrir
- Lectura /escritura
- Cerrar
- Recorrido
- Consulta
- Actualización
- Borrado
- Ordenación
- Búsqueda

2.3. Almacenamiento en archivos

Puede ser de dos maneras:

- **Modo Texto:** Los datos son almacenados usando ASCII y por tanto, podemos abrirlos, consultarlos y modificarlos con cualquier editor de texto.
- **Modo Binario**: Los datos son almacenados en notación hexadecimal y a diferencia de los archivos de texto, estos no pueden visualizarse con un editor o procesador de texto

cualquiera. Un archivo binario es más compacto que un archivo texto y es el indicado para acceso directo o aleatorio.

2.4. Archivo lógico y archivo físico

- **Archivo lógico:** Permite manejar archivos independientemente de su representación, almacenamiento, etc.
- **Archivo físico:** Es donde esta contenida la información que ocupa un espacio exacto y concreto en el disco, y que tiene asociado el nombre del archivo, el dueño, el tamaño, el número de registros.

2.5. Ubicación lógica y física de los registros en un archivo secuencial

En los archivos secuenciales distinguimos dos tipos de ubicación.

- **Ubicación lógica:** cuando hablamos de ubicación logica nos referimos a la abstraccion que hacemos mentalmente sobre como se organiza la información o como conceptuamos la estructura que toman los registros en un archivo, sin considerar de una manera precisa el registro fisico de los datos en un almacenamiento secundario.

- **Ubicación física:** es el concepto en el cual el contenido de los campos del registro fija y determina su posición real o física dentro del dispositivo de almacenamiento secundario. Fisicamente la organización del archivo en el almacenamiento secundario se iguala con la ubicación lógica.

En este archivo los registros tienen la siguiente estructura:

```
class producto
{
        int codigo;                 // 2 bytes
        char descripcion[30];       // 30 bytes
        int cantidad;                       // 2 bytes
        float precio;                       // 4 bytes
};
```

Haciendo la suma encontramos que cada registro del archivo tiene un tamaño de 38 bytes. La función *sizeof()* calcula el tamaño en bytes de una variable.

dir_fisica = dir_logica *sizeof(registro)

Dirección lógica

Dirección física

		código	descripción	cantidad	precio
0	0	1	Camisa de vestir	3	50
1	38	2	Polo	2	30
2	76	3	Pantalón para caballero	1	80

2.6. Manejos de flujo (archivo)

En C y C++, el ingreso y salida de información se realiza usando el concepto de flujos. Un flujo es una corriente de información (de caracteres u otro tipo de datos), en un solo sentido a partir de una fuente y que tiene un destino que puede ser el teclado, discos, pantalla, etc. a los cuales se les denomina archivos, y que nos permite efectuar tanto operaciones de entrada como de salida. Para manejar los flujos C y C++ lo hacen de dos maneras:

- Una manera es a través de las librerías del Standar Input Output de C.
- Otra forma es encapsular las funciones de E/S en una serie de clases conocidas con el nombre genérico de streams.

En la segunda manera las clases proporcionan mayor seguridad, dado que utilizan la técnica RAII ("Resource Acquisition Is Initialization") donde la idea básica es que los constructores obtengan recursos, permitan crear un flujo y asociarlo a un archivo, y los destructores los liberen. Usando estas clases para manipular archivos, una vez realizadas las operaciones E/S, no hay que preocuparse de que el archivo quede abierto, porque es cerrado automáticamente por el destructor.

Si queremos conectar un flujo a un archivo tendremos que declararlo igual que declaramos cualquier otra variable. Podemos declarar *escribe* como un flujo de salida para un archivo, y lee como un flujo de entrada para otro archivo como sigue:

```
ofstream escribe;
ifstream lee;
```

Para realizar programas de manejo de archivos, se necesita incluir la siguiente librería:

```
#include <fstream>
```

El siguiente código es un programa que crea un archivo de texto al cual denominamos texto.txt y que tiene como única finalidad insertar una línea de texto en un archivo.

```
/* inserta una línea en el archivo */
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int  main () {
   ofstream objtex;
   objtex.open ("texto.txt");
   objtex<< " Esta linea sera escrita en un archivo.\n";
   objtex.close();
   system("PAUSE");
   return 0;
}
```

a) Abrir archivo - open()

Abrir un archivo significa primero crear una instancia de una clase ofstream, ifstream o fstream. En el ejemplo anterior la instancia es objtex. Segundo la función miembro open, se utiliza para abrir un archivo físico (en el ejemplo texto.txt) y asociarlo a la instancia objtex. La función miembro open determina el modo de apertura y establece la vía de comunicación entre el flujo de entrada y el archivo correspondiente. Los argumentos de la función open:

open(char *nombreArchivo, char *modo)

nombreArchivo es una cadena de caracteres donde se especifica principalmente el nombre del archivo fisico, además se puede considerar la ruta o path en el cual se encuentra el archivo

modo a través de una cadena de caracteres se indica el modo que se abrira el archivo.

Valor devuelto

- Devuelve un puntero al archivo abierto
- o -1 si se ha producido un error.

El siguiente cuadro muestra los modos de apertura de archivos de texto y binarios.

Modo de apertura Archivo texto	Operación
ios::in	Se abre el archivo para lectura.
ios::out	Se abre el archivo para escritura.
ios::app	Permite al fichero añadir datos al final del archivo
ios::binary	Abre el archivo para operar en modo binario
ios::trunc	Si el archivo abierto para operaciones de salida ya existe físicamente, el contenido se borra y será reemplazado por el nuevo que se introduzca.
ios::ate	Coloca el apuntador de archivo al final. Si se omite, la posición inicial queda al inicio del archivo.

Como ejemplo se abre para lectura el archivo producto.dat, en modo binario:

```
ifstream lee;   // Declara lee como un flujo de entrada
lee.open("producto.dat",ios::in |ios:binary); // abre para
lectura
```

también se puede utilizar de una manera más directa:

```
ifstream lee("producto.dat",ios::in | ios::binary);
```

Otro ejemplo consiste en abrir para escritura el archivo archivo.dat, para adicionar y en modo binario:

```
ofstream escribe1; // Declara escribe1 como un flujo de salida
escribe1.open("archivo.dat",ios::out|ios::app|ios::binary);
```

también se puede utilizar de una manera más directa:

```
ofstream escribe1("archivo.dat",ios::out|ios::app|ios::binary);
```

b) Validar la apertura de un archivo

Al utilizar la función open esta puede fallar por varias razones, entre ellas tenemos dos muy comunes:

a) Intentar abrir un fichero que no existe
b) Intentar abrir un fichero de salida y no tener permiso de escritura sobre dicho archivo.

Cuando sucede no se despliegara un mensaje de error y el programa continuara. Por ello, es conveniente después de una llamada a open comprobar si el fichero se ha abierto correctamente y si no es así, tener conocimiento del error como se observa a continuación:

```
 ifstream entrada("datos.dat",ios::in | ios:binary);
if (!entrada)
{
      cout << "No se pudo abrir el fichero de entrada\n";
}
else
{
   //  procesar el archivo
}
```

c) Método write()

La función write permite registrar una variable simple, un registro o un objeto en un archivo. Los argumentos de la función write:

```
write(char *buffer, int tamaño);
```

donde

- **buffer** es un puntero a los datos que se escribirán
- **tamaño** es la cantidad de espacio que se necesita para registrar el buffer en byte.

Valor devuelto

- Devuelve el número de elementos completos que estén realmente escritos.

Para grabar usamos el operador:

```
reinterpret_cast <char*> (&x)
sizeof(tamaño)
```

En lo que respecta al buffer se usa el operador **reinterpret_cast <char*> (&variable)** en donde variable puede ser de tipo simple como entero, punto flotante, char, cadena o un tipo abstracto de dato (TAD)

En el *tamaño* se usa el operador **sizeof**, que devuelve el número de bytes de una variable:

```
sizeof(tamaño)
```

Lo que hace la función es escribir en el archivo los datos contenidos en la variable x (reinterpret_cast<char *>(&x)) y para ello necesita conocer el tamaño de lo que se va a escribir (sizeof(tamaño)).

Por ejemplos para la variable entera:

```
int nro;
```

La función write se escribe

```
write(interpret_cast <char*> (&nro), sizeof(int));
```

Para la variable de tipo ALUMNO:

```
ALUMNO alum;
```

La función write se escribe

```
write(reinterpret_cast <char*> (&alum), sizeof(ALUMNO));
```

d) Método close()

Una vez que se ha terminado de trabajar con un archivo debe cerrarse, un archivo se cierra utilizando el método close. Al cerrar un archivo desconectamos su flujo. Sintaxis:

```
objeto.close ()
```

Valor devuelto

- Devuelve cero si el flujo se cierra con éxito
- Devuelve EOF para indicar un error.

Cuando un archivo no se utiliza debe de cerrarse.

Por ejemplo se abre un archivo entero.dat, para escritura, en modo adicionar y en modo binario y se registrara una variable entera.

```
ofstream escribe("entero.dat", ios::out|ios::app| ios::binary );
if (!escribe)
   cout << "ERROR: No se puedo realizar la apertura del
archivo";
else
{
   escribe.write(reinterpret_cast<char*>(&nro),sizeof(int));
   escribe.close();
}
```

e) Método read()

La función read posibilita tener conocimiento de todos los campos de un registro en un archivo, es decir, toma lectura de un registro y

carga en la memoria, en una variable de tipo simple o TAD. Esta función tiene los mismos argumentos que el método write

```
read(char *buffer, int tamaño)
```

donde

- **buffer** es un puntero en el cual los datos se cargaran.
- **tamaño** es la cantidad de espacio que requiere el buffer en byte.

Valor devuelto

- Devuelve el número de elementos completos que estén realmente leídos.

f) Metodo eof()

La función eof (end of file), se utiliza para detectar el final de un archivo. Su sintaxis se muestra a continuación:

```
int eof (objeto.eof())
```

Valor devuelto

- 0 si no encontrado el final del archivo
- 1 en caso contrario

Usando una estructura while, el programa puede ejecutar un ciclo de lectura hasta que encuentre el final de un archivo, como se observa en el ejemplo siguiente en el cual el archivo contiene datos enteros:

```
// Primera lectura
lee.read(reinterpret_cast <char *> (&x),sizeof(int));
//Mientras que no sea fin de archivo
while (!lee.eof() )
{
        cout << x<<"\n ";
        // Segunda y demas lecturas
        lee.read(reinterpret_cast <char *> (&x),sizeof(int));
}
```

Ahora solo falta añadirle el código que nos haga saber si el archivo se abrio correctamente para lo cua haremos:

```
ifstream lee("datos.dat",ios::in| ios::binary);
if (!lee)
{
        cout<<"\n no se pudo abrir el archivo correctamente: ";
}
else
{
        // Primera lectura
```

```
        lee.read(reinterpret_cast <char *> (&x),sizeof(int));
        //Mientras que no sea fin de archivo
        while (!lee.eof() )
        {
                cout <<x<<"\n ";
                // Segunda y demas lecturas
                lee.read(reinterpret_cast <char *> (&x),sizeof(int));
        }
}
```

2.7. Filas Secuenciales

Mediante esta estructura es posible definir el concepto de acceso secuencial a una información, es usado principalmente para almacenamiento secundario (disco, disquete, cinta, usb).

a) Características de una fila secuencial

- Para acceder a un elemento de una fila secuencial es preciso recorrer los elementos anteriores a él.
- Después del último elemento se encuentra la marca de fin de fila secuencial. Existe al final una marca de fin.

b) Primitivas básicas

primitiva	descripción
Inicio (F)	Posiciona la fila F antes del primer elemento.
Leer (F, V)	Lee el elemento de la fila F en una posición y lo almacena en la variable V, luego avanza una posición
Escribir (F, V)	Escribe el valor de V en la fila F, siempre en la ultima posición, luego avanza una posición.
Ultimo (F)	Retorna verdadero si encuentra la marca de fin de la fila y falso en otro caso.
Cerrar (F)	Cierra la fila secuencial

2.8. Ejercicios resueltos

Ejemplo 01

Escriba la especificación de los TAD's, el algoritmo y su implementación en C++, para registrar en una tercera fila, la concatenación de dos filas. Por ejemplo:

Entrada:

Fila1: 4, 5, 8, 10
Fila2: 7, 13, 15

Salida:

Fila3: 4, 5, 8, 10, 7, 13, 15

Solución:

Especificación de los TAD's y los algoritmos

Especificación ENTERO

variable

num : entero

métodos

menu() : no retorna valor
ingresarEntero() : no retornar valor
mostrarEntero() : no retornar valor
retornarEntero() : retornar valor entero

significado

menu muestra las opciones a escoger
ingresarEntero permite ingresar números enteros
mostrarEntero muestra los datos del valor entero
retornarEntero retorna valor entero

Fin_especificacion

Especificación FILA

Usa ENTERO

variable

e : ENTERO

métodos

registrarFila(fila, x) : no retornar valor
concatenarFila(fila1,fila2, fila3) : no retornar valor
mostrarFila(fila) : no retornar valor

significado

menu muestra las opciones a escoger
registrarFila añade x del tipo FILA a fila secuencial
concatenar tiene como precondición a la fila1 y a la fila2, y como postcondición, a la fila3
mostrarFila muestra el contenido de la fila secuencial

Fin_especificacion

```
Procedimiento concatenacion (fila1, fila2, fila3)
    Inicio (fila1)      // Lectura
    Inicio (fila2)      // Lectura
    Inicio (fila3)      // Escritura
    Leer (fila1, val1)
    Mientras (no ultima fila1) Hacer
        Escribir (fila3, val1)
        Leer (fila1, val1)
    Fin_mientras
    Cerrar (fila1)
    Leer (fila2,val2)
    Mientras (no ultima fila2) Hacer
```

```
            Escribir (fila3, val2)
            Leer (fila2, val2)
      Fin_mientras
      Cerrar (fila2)
      Cerrar (fila3)
Fin_procedimiento
```

Implementación del TAD

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
using namespace std;
class ENTERO
{
      int num;
      public:
            void menu()
            {
                  cout<< "\nMENU DE OPCIONES\n";
                  cout<< "---------------\n" ;
                  cout<< "<1> Adiccionar fila1 \n";
                  cout<< "<2> Adicionar fila2 \n";
                  cout<< "<3> Concatenar filas \n";
                  cout<< "<4> Mostrar filas \n";
                  cout<< "<5> Salir\n";
            }

            void ingresarEntero()
            {
                  fflush(stdin);
                  cout<<"\n Ingresar entero: ";cin>>num;
            }
            void mostrarEntero()
            {
                  cout<<num<<setw(4);
            }
};

class FILA
{
      ENTERO e;
      public:
      void registrarFila(char fila[], ENTERO x)
      {
            ofstream salida(fila,ios::out |ios::app|ios::binary);
            if (!salida)
```

```
                cout << "ERROR: No se puedo realizar la apertura del
archivo";
        else
        {
                salida.write(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
                salida.close();
        }
    }

    void concatenarFila(char fila1[],char fila2[],char fila3[])
    {
        ENTERO x;
        ifstream F1 (fila1,ios::in|ios::binary);
        ifstream F2 (fila2,ios::in|ios::binary);
        ofstream F3 (fila3,ios::out|ios::binary);
        if (!F3)
                cout<<"ERROR al abrir "<<fila3;
        else  if (!F1 && !F2)
                cout << "ERROR al abrir "<<fila1 <<" o "<<fila2;
        else
        {
                // Primera lectura
                F1.read(reinterpret_cast          <char          *>
(&x),sizeof(ENTERO));
                //Mientras que no sea fin de archivo
                while (!F1.eof() )
                {
                        F3.write(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
                        // Segunda y demas lecturas
                        F1.read(reinterpret_cast          <char          *>
(&x),sizeof(ENTERO));
                }
                F1.close();
                // Primera lectura
                F2.read(reinterpret_cast          <char          *>
(&x),sizeof(ENTERO));
                //Mientras que no sea fin de archivo
                while (!F2.eof() )
                {
                        F3.write(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
                        // Segunda y demas lecturas
                        F2.read(reinterpret_cast          <char          *>
(&x),sizeof(ENTERO));
                }
                F2.close();
```

```
                F3.close();
            }
        }
        void mostrarFila(char fila[])
        {
            ENTERO x;
            ifstream lee(fila,ios::in|ios::binary);
            if (!lee)
                cout << "No se pudo abrir el archivo.";
            else
            {
                // Primera lectura
                lee.read(reinterpret_cast <char *>
(&x),sizeof(ENTERO));
                //Mientras que no sea fin de archivo
                while (!lee.eof() )
                {
                    x.mostrarEntero();
                    // Segunda y demas lecturas
                    lee.read(reinterpret_cast <char *>
(&x),sizeof(ENTERO));
                }
                lee.close();
            }
        }
};
int main()
{
        char opcion;
        FILA x;
        ENTERO e;
        do
        {
            e.menu();
            cout<<"ingrese opcion : ";
            opcion=cin.get();
            switch (opcion)
            {
                case '1':
                    e.ingresarEntero();
                    x.registrarFila("fila1",e);
                    break;
                case '2':
                    e.ingresarEntero();
                    x.registrarFila("fila2",e);
                    break;
                case '3':
                    x.concatenarFila("fila1","fila2", "fila3");
```

```
                    cout<<"\n Se concateno exitosamente\n";
                    break;
                case '4':
                    cout<<"fila 1:\n ";
                    x.mostrarFila("fila1");cout<<"\n";
                    cout<<"fila 2:\n ";
                    x.mostrarFila("fila2");cout<<"\n";
                    cout<<"fila 3:\n ";
                    x.mostrarFila("fila3");cout<<"\n";
                    break;
            }
            cin.ignore();
        }
        while (opcion !='5');
         system("PAUSE");
         return 0;
}
```

Ejemplo 02

Escriba la especificación de los TAD's, el algoritmo y su implementación en C++, para ingresar N números enteros a una fila y obtenga dos nuevas filas, una fila que contenga los números pares y otra fila que contenga los números impares. Por ejemplo:

Entrada
FILA 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Salida
FILA 2: 2, 4, 6, 8, 10
FILA 3: 1, 3, 5, 7, 9

Solución

Especificación de los TAD's y los algoritmos

En esta parte la especificación es muy parecida al del ejemplo 01 con la diferencia que no es necesario contar con el método concatenar, sino mas bien con el metodo separarpiFila tiene como precondición a la fila1, y como postcondición, a la fila2 y a la fila3. Aquí el algoritmo de este método:

```
Procedimiento separarpiFila(fila1, fila2, fila3)
    Inicio (fila1)          // Lectura
    Inicio (fila2)          // Escritura
    Inicio (fila3)          // Escritura
    Leer (fila1, x)
    Mientras (no ultimo fila1) Hacer
```

```
                Si (x.retornarEntero( ) Mod 2 = 0 ) entonces
                        Escribir (fila2, x)
                Sino
                        Escribir (fila3, x)
                Fin_si
                Leer (fila1, x)
        Fin_mientras
        Cerrar (fila1)
        Cerrar (fila2)
        Cerrar (fila3)
Fin_procedimiento
```

Implementación del TAD

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
using namespace std;
class ENTERO
{
        int num;
        public:
        void menu()
        {
                cout<< "\nMENU DE OPCIONES\n";
                cout<< "----------------\n" ;
                cout<< "1. Adiccionar fila\n";
                cout<< "2. Separar fila \n";
                cout<< "3. Mostrar filas\n";
                cout<< "4. Salir\n";
        }
        void ingresarEntero()
        {
                fflush(stdin);
                cout<<"\n Ingresar entero: ";cin>>num;
        }
        void mostrarEntero()
        {
                cout<<num<<setw(4);
        }
        int retornarEntero()
        {
                return num;
        }
};

class FILA
```

```
{
	ENTERO e;
	public:
	void registrarFila(char fila[50],ENTERO x)
	{
		ofstream salida(fila,ios::out |ios::app|ios::binary);
		if (!salida)
		cout << "ERROR: No se puedo realizar la apertura del
archivo";
		else
		{
			salida.write(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
			salida.close();
		}
	}

	void separarpiFila(char fila1[],char fila2[], char fila3[])
	{
		ENTERO x;
		ifstream F1 (fila1,ios::in|ios::binary);
		ofstream F2 (fila2, ios::out|ios::app|ios::binary);
		ofstream F3 (fila3, ios::out|ios::app|ios::binary);
		if(!F3)
			cout<<"\n ERROR al abrir "<<fila3;
		else if (!F1  && !F2)
				cout<< "error al abrir "<<fila1<< " o
"<<fila2;
		else
		{
			F1.read(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
			while(!F1.eof())
			{
				if(x.retornarEntero() % 2 == 0)
					F2.write(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
				else
					F3.write(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
				F1.read(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
			}
			F1.close();
			F2.close();
			F3.close();
		}
	}
```

```
	void mostrarFila(char fila[])
	{
		ENTERO x;
		ifstream lee(fila,ios::in|ios::binary);
		if (!lee)
			cout << "No se pudo abrir el archivo.";
		else
		{
			// Primera lectura
			lee.read(reinterpret_cast <char *>
(&x),sizeof(ENTERO));
			//Mientras que no sea fin de archivo
			while (!lee.eof() )
			{
				x.mostrarEntero();
				// Segunda y demas lecturas
				lee.read(reinterpret_cast <char *>
(&x),sizeof(ENTERO));
			}
			lee.close();
		}
	}
};
int main()
{
	char opcion;
	FILA x;
	ENTERO e;
	do
	{
		e.menu();
		cout<<"ingrese opcion : ";
		opcion=cin.get();
		switch (opcion)
		{
			case '1':
				e.ingresarEntero();
				x.registrarFila("fila1",e);
				break;
			case '2':
				x.separarpiFila("fila1","fila2", "fila3");
				cout<<"\n Se separo exitosamente \n";
				break;
			case '3':
				cout<<"fila 1:\n ";
				x.mostrarFila("fila1");cout<<"\n";
				cout<<"fila 2:\n ";
```

```
                        x.mostrarFila("fila2");cout<<"\n";
                        cout<<"fila 3:\n ";
                        x.mostrarFila("fila3");cout<<"\n";
                        break;
            }
            cin.ignore();
        }
        while (opcion !='4');
         system("PAUSE");
         return 0;
}
```

Ejemplo 03

Escriba la especificación de los TAD's, el algoritmo y su implementación en C++, para registrar en una tercera fila, la intersección de dos filas. Por ejemplo:

Entrada
FILA 1: 3, 7, 11, 15, 19
FILA 2: 4, 7, 13, 15, 18, 19, 20
Salida
FILA 3: 7, 15, 19

Solución

Especificación de los TAD's y los algoritmos

En esta parte la especificación es muy parecida al del ejemplo 01 con la diferencia que no es necesario contar con el método *concatenarFila*, sino mas bien con el metodo *intersectarFila* que tiene como precondición a la fila1 y a la fila2, y como postcondición, a la fila3:

```
Procedimiento intersectarFila(fila1, fila2, fila3)
    // Definir variables
    Lógico: salir
    Inicio (fila1)              // Lectura
    Inicio (fila3)              // Escritura
    // Primera lectura
    Leer (fila1, x)
    Mientras (no ultima fila1) Hacer
        salir ← false
         Inicio (fila2)              // Lectura
        Leer (Fila2, y)
        Mientras(no ultima fila2 y no salir) Hacer
            Si (x.retornarEntero( ) = y.retornarEntero())
        entonces
                    salir ← true
```

```
                    Fin_si
                    Leer (fila2, y)
                Fin_mientras
                 Cerrar (fila2)
                Si  (salir = true) entonces
                    Escribir (fila3, x)
                Fin_si
                Leer (fila1, x)
            Fin_mientras
            Cerrar (fila1)
            Cerrar (fila3)
        Fin_procedimiento
```

Implementación del TAD

De manera similar a la especificación, en el programa del ejemplo 01 habria que reemplazar la función concatenarFila por la función intersectarFila que se observa abajo:

```
void intersectarFila(char fila1[],char fila2[], char fila3[])
{
        bool salir;
        ENTERO x,y;
        ifstream F1(fila1,ios::in|ios::binary);
        ofstream F3(fila3,ios::out|ios::app|ios::binary);
        if(!F3)
            cout<<"\n ERROR al abrir "<<fila3;
        else if (!F1)
                cout<< "error al abrir "<<fila1;
        else
        {
            F1.read(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
            while(!F1.eof() )
            {
                salir=false;
                ifstream F2(fila2,ios::in|ios::binary);
                    F2.read(reinterpret_cast<char
            *>(&y),sizeof(ENTERO));
                while(!F2.eof() && !salir)
                {

    if(x.retornarEntero()==y.retornarEntero())
                            salir=true;
                        F2.read(reinterpret_cast<char
*>(&y),sizeof(ENTERO));
                }
                F2.close();
```

```
                        if (salir ==true)
                                F3.write(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
                        F1.read(reinterpret_cast<char
*>(&x),sizeof(ENTERO));
                }
                F1.close();
                F3.close();
        }
}
```

luego modificar el código del main para que quede como se observa a continuación:

```
int main()
{
        char opcion;
        FILA x;
        ENTERO e;
        do
        {
                e.menu();
                cout<<"Ingrese opcion : ";
                opcion=cin.get();
                switch (opcion)
                {
                        case '1':
                                e.ingresarEntero();
                                x.registrarFila("fila1",e);
                                break;
                        case '2':
                                e.ingresarEntero();
                                x.registrarFila("fila2",e);
                                break;
                        case '3':
                                x.intersectarFila("fila1","fila2", "fila3");
                                cout<<"\n          Se          intersecto
exitosamente\n";
                                break;
                        case '4':cout<<"fila 1:\n ";
                                x.mostrarFila("fila1");cout<<"\n";
                                cout<<"fila 2:\n ";
                                x.mostrarFila("fila2");cout<<"\n";
                                cout<<"fila 3:\n ";
                                x.mostrarFila("fila3");cout<<"\n";
                                break;
                }
```

```
            cin.ignore();
        }
        while (opcion !='5');
        system("PAUSE");
         return 0;
}
```

2.9. Ejercicios propuestos.

1. Escriba la especificación de los TAD'S, el algoritmo y su implementación en C++, para registrar en una tercera fila, la ordenación de dos filas secuenciales. Por ejemplo:

   Entrada
   FILA1: 2, 12, 25, 30, 80
   FILA2: 3, 26, 28
   Salida
   FILA3: 2, 3, 12, 25, 26, 28, 30, 80

2. Escriba la especificación de los TAD's, el algoritmo y su implementación en C++, para registrar en una tercera fila, la intercalacion de dos filas. Por ejemplo:

   Entrada
   FILA1: 20, 3, 7, 9
   FILA2: 5, 6
   Salida
   FILA3: 20, 5, 3, 6, 7, 9

3. Escriba la especificación de los TAD'S, el algoritmo y su implementación en C++, para registrar en una tercera fila, los elementos impares de la fila1 y de la fila2. Por ejemplo:

   Entrada
   FILA 1: 7, 3, 14, 26, 9
   FILA 2: 11, 18, 17, 8, 5, 20,21
   Salida
   FILA 3: 7, 3, 9, 11, 5, 21

4. Escriba la especificación de los TAD's, el algoritmo y su implementación en C++, para registrar en una segunda fila, los datos pares al final de la fila1. Por ejemplo:

Entrada
FILA1: 20, 12, 9, 10, 5, 3
Salida
FILA2: 9, 5, 3, 20, 12, 10

Observación
Si desea puede usar otras filas auxiliares (no arreglos)

5. Escriba la especificación de los TAD's, el algoritmo y su implementación en C++, para registrar en una segunda fila, se puedan invertir las ubicaciones de los datos impares y pares de la fila1

Por ejemplo:

Entrada
FILA1: 20, 12, 9, 5
Salida
FILA2: 9, 5, 20, 12

Observación
Si desea puede usar otras filas auxiliares (no arreglos)

Resumen

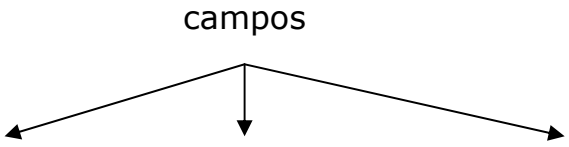
Cuando programamos hacemos uso de datos elementales (tipos simples) como enteros, reales, caracteres, etc, al que se asocian operaciones segun su tipo de dato, por ejemplo sumas y restas para los tipos enteros o concatenacion para las cadenas, sin embargo una estructura de datos dispone sobre un conjunto de datos elementales que pueden ser de distinto tipo, un conjunto de operaciones para acceder a esos datos, y que se conoce como tipo abstracto de dato(TAD).

Las estructuras de datos pueden dividirse en dos grandes grupos: Estructuras de datos lineales y no lineales. Ambos grupos explican la manera de construir sus operaciones a través del uso de algoritmos. Un algoritmo es una secuencia finita de acciones para llevar a cabo una tarea especifica, y puede representarse como diagramas de flujo, Diagramas Nassi-Schneiderman y pseudocódigo.

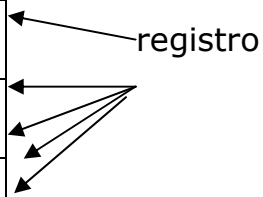
El pseudocódigo es una técnica mezcla de lenguaje natural y formal que nos permite representar la secuencia lógica de las acciones de un algoritmo. Para su comprensión y aplicación es necesario conocer que es una expresion, un identificador, definición de variables, operaciones de asignación, entrada y salida de datos asi como de estructuras básicas de programación(secuencial, iterativa y selectivas) enmarcadas dentro de funciónes y procedimientos.

Volviendo a los TAD's es posible describirlos a través de una especificación en la que se asocian datos y operaciones (algoritmos) e implementarlos en un lenguaje de programación. En este libro los TAD's se implementan en C++ usando la programación orientada a objetos(POO). En una POO, el concepto de objeto se refiere a una entidad que engloba datos(atributos) y operaciones(metodos), y que se construyen en base a un patrón denominado clase.

En esta unidad se tratan las estructuras de filas secuenciales, utiles para conservar los datos de un programa en memoria secundaria mediante el uso de archivos en modo texto(se guarda la información en caracteres) o en modo binario(se guarda la información en forma hexadecimal). En un archivo se distingue el concepto de campo(dato simple) y el de registro(colección de campos) como se observa en la representacion grafica de abajo de un archivo de alumnos:



CODIGO	NOMBRE	EDAD
0950080	Juan carlos, Agreda sanchez	22
0950100	Julio cesar, Castillo rodriguez	23
0951200	Manuel Alejandro, cuba miranda	21
0951233	Ana maria, Montalvo rios	22
0951343	Jose luis, Solis villafani	20



Lectura

Organizacion de los datos para su proceso en computadora

Todo lo que las computadoras hacen es procesar datos. Únicamente aceptan, procesan datos y comunican resultados. No pueden llevar a cabo actividades físicas. No pueden doblar metal, por ejemplo, pero pueden dar la información necesaria para controlar las maquinas dobladoras de metal.

Normalmente, los datos que las computadoras manejan están organizados en agrupamientos lógicos, para que los procesos sean efectivos y los resultados obtenidos sean útiles. La entidad lógica mas pequeña es un campo que consiste en grupo de caracteres unidos tratados como una sola unidad. Algunos campos fueron ilustrados en el ejemplo de la nomina de la Universidad de Longhorn, en el capitulo anterior. Había un campo nombre ("Rob Brooks"), un campo de horas trabajadas, un campo de cuotas por hora y un campo de tasa de impuesto. Durante el proceso, los caracteres de cada uno serán usados como una unidad.

Por lo regular los campos son agrupados juntos para formar un registro. Un registro es, entonces, una colección de campos unidos o grupos de datos, que son tratados como una sola unidad. De ahí que habría un registro integrado por cuatro campos por cada uno de los estudiantes empleados de la Universidad de Longhorn. Los registros son agrupados para formar un archivo . Un archivo es un numero de registros relacionados que son tratados como una unidad. El archivo de la nomina de estudiantes de Longhorn consiste en el registro de todos los estudiantes empleados . Finalmente, una base de datos es una colección de datos relacionados que pueden ser estructurados en diferentes formas para cumplir con las necesidades de proceso y recuperación de las organizaciones e individuos.

Donald H. Sanders., Alan Freedman (1985). *Biblioteca McGraw-Hill de informática. Tomo I: Antecedentes*. Madrid, McGraw-Hill., pp. 42-43.

Autoevaluación

1. Determine que alternativa contiene identificadores correctos

   a) suma, 90dato b) suma, dato90 c) su%ma, 90dato90 d) ninguna

2. Cuál es la diferencia entre una clase y un objeto

   a) Clase es el patrón y el objeto es equivalente a una variable.
   b) El objeto es un concepto matemático y la clase una definición.
   c) El objeto es el patrón y la clase es equivalente a una variable.
   d) ninguna.

3. ¿Cómo se avanza en una fila secuencial?

   a) con cada lectura
   b) con cada escritura
   c) cuando se verifica la marca EOF.
   d) ninguna.

4. Escoja la mejor alternativa a la pregunta de saber cuánto ocupa cada objeto o registro que almacena una fila secuencial, donde cada objeto tiene la siguiente definición de clase:

```
class ALUMNO
{
        char dni[12];
        int código;
        char nombre[50];
        bool condicion;
        char tipodematricula;
        float promPonderado;
        char sexo;
};
```

   a) 72 bytes b) 74 bytes c) 71 byte d) ninguna

Claves: 1:b; 2:a; 3:a; 4:c

Enlaces

http://www.di-mare.com/adolfo/p/oop-adt.htm
http://eupt2.unizar.es/eda/temario/Especificaciones.html
http://www.davidparedes.es/2009/01/02/tipos-abstractos-de-datos-tad/

Bibliografía

Tenenbaum A. M., Langsam Y., Augenstein,M.A., (1993) *Estructura de Datos en C*, 2ª Ed. México D.F., Prentice Hall Hispanoamericana S.A. Cap. 1: Introducción a la estructura de datos, pp. 1-76.

Joyanes Aguilar, L., Sanchez Garcia, L., Fernadez Azuela, M. Zahonero Martinez, I.,(2005) *Estructura de Datos en C,* Madrid, Mc Graw-Hill - Interoamericana. Cap. 1: Algoritmos, estructuras de datos y programas, pp.9-53.

Hernandez, R., Lazaro, J.C., Dormido, R., Ros, S. (2006) Estructura de Datos y Algoritmos, 2ª Ed. México D.F., Prentice Hall Hispanoamericana S.A. Cap. 2: Análisis de Algoritmos avanzados, pp. 53-89.

Schildt H., (1994) Turbo C/C++ Manual de Referencia, Una información completa ideal para todo usuario de Turbo C/C++, Madrid, Mc Graw-Hill/Interamericana. Cap. 1: Introducción al lenguaje C, pp. 10-55.

López R. L., (2006) *Metodología de la Programación Orientada a Objetos*, México D.F., Alfaomega, Grupo Editor S.A. Cap. 2: Estructuras de datos, pp. 43-77.