Algoritmos y estructuras de datos – curso de verano 2023

Curso a cargo de: Ing.Roxana Leituz con la colaboración del Dr.Oscar Bruno

El curso tendrá modalidad virtual con una clase en línea semanal destinada a orientar sobre los temas, resolver dudas y hacer ejercicios, se recomienda participar activamente de las mismas.

En el link del repositorio del material hay 4 carpetas con un documento para escribir las consultas y dudas que vayan teniendo, las mismas serán respondidas por ese medio o en la clase de consulta.

Considerando que el curso se desarrolla en muy poco tiempo se tomará un único parcial integrador de todos los temas con dos recuperatorios con modalidad presencial.

Adjunto se encuentra todo el material teórico obligatorio realizado por el director de cátedra Dr.Oscar Bruno con todos los temas necesarios para aprobar la materia.

En el TEMARIO se indican los temas que se tratarán cada semana para que los estudiantes puedan organizar su estudio.

1. De los trabajos prácticos:
    - **a.** Los trabajos prácticos acompañan la materia y tienen fecha de entrega. **La fecha debe respetarse estrictamente para aquellos que quieran promocionar la materia.**
    - **b.** Los que no promocionen **TAMBIÉN TIENEN OBLIGACIÓN DE ENTREGAR EL/LOS TP un día antes antes de finalizar la cursada.**
    - c. TODOS los trabajos deben contener una carátula con la presentación de los alumnos , los enunciados a resolver y una explicación con la estrategia de solución. (grupos de 3 a 5 personas)
2. De las comunicaciones:
    - a. Las comunicaciones con los docentes son por mail institucional (**rleituz@frba.utn.edu.ar)**.

CRONOGRAMA DE CLASES (31 de enero al 10 de marzo)

L	M	M	J	V
	31	1	2	3
6	7	8	9	10
13	14	15	16	17
~~20~~	~~21~~	22	23	24
27	28	1	2	3
6	7	8	9	10

CLASES ONLINE

PARCIALES

NOTA DE PROMOCIÓN MÍNIMA: 8 (OCHO)

NOTA DE APROBACIÓN MÍNIMA: 6 (SEIS)

Clases en línea:

- **Martes 31 de enero – 18,30hs**
- **Martes 7 de febrero – 18,30hs**
- **Miércoles 15 de febrero – 18,30hs**
- **Miércoles 22 de febrero – 18,30hs**
- **Martes 28 de febrero – 18,30hs**

Parciales (campus 19 hs.)

- **Integrador 2 de marzo**
- **Recuperatorio 1 lunes 6**
- **Recuperatorio 2 miércoles 8**

- **6 de marzo Entrega de TP para promoción (9 para no promoción)**
- **Viernes 10 de marzo Cierre de notas**

Link: https://meet.google.com/nnd-izig-vas

Repositorio de material:
https://drive.google.com/drive/folders/1GQn1wipBBsfWL0xXWl6TfAMIpypV6VHg?usp=share_link

TEMARIO

SEMANA 1

Primeros pasos

Tema	
Introducción a la algoritmia	
Entorno de desarrollo C-C++	
Tipos de datos	
Estructuras de control	
Asignación	
Análisis de caso	
Ciclos	
Patrones algorítmicos simples	
Intercambios	
Máximos y mínimos: distintos lotes, distintos criterios	
Seguidillas	
Ejercicios integradores	
Práctica	

Funciones

Tema	
Definiciones y declaraciones	
Declaración, definición, prototipos, invocación, bibliotecas	
Intercambio de información	
Argumentos, parámetros: por valor, por referencia	
Reusabilidad – Generalidad	
Concepto de reusabilidad	
Concepto de generalidad	
Ejercicios integradores	

Práctica	

Struct

Tema	
Definiciones y declaraciones	
Combinación de estructuras: struct con un campo struct	
Asignación interna y externa	

SEMANA 2

Array

Tema	
Necesidad de su uso. Definiciones y declaraciones	
Vectores y matices	
Datos simples, estructuras	
Combinación de estructuras: array de struct, struct con array	
Recorridos	
Secuencial	
En un rango	
Con Corte de control (criterio único/criterio múltiple)	
Apareo (criterio único/criterio múltiple)	
Búsqueda	
Directa	
Secuencial	
Dicotómica (criterio único/criterio múltiple)	
Carga	
Directa	
Secuencial	
Ordenada	

Ordenamiento	
Con Posición Única Predecible	
Método de ordenamiento	
Ejercitación	
Ejercicios integradores	
Práctica	

SEMANA 3

Flujos

Tema	
Necesidad de su uso. Definiciones y declaraciones	
Funciones de C FILE*, fopen, fclose, frear, fwrite, feof, ftell, fseek	
Creación y carga	
Agregar datos a estructura existente	
Recorridos	
Secuencial	
En un rango	
Con Corte de control (criterio único/criterio múltiple)	
Apareo (criterio único/ criterio múltiple)	
Búsqueda	
Directa	
Secuencial	
Dicotómica (criterio único/criterio múltiple)	
Carga	
Directa	
Secuencial	
Ordenada PUP	
Ordenamiento	

Con Posición Única Predecible	
Estructuras auxiliares	
Archivos, vectores	
Ejercitación	
Ejercicios integradores	
Práctica	

SEMANA 4

Punteros y estructuras enlazadas

Tema	
Necesidad de su uso. Definiciones y declaraciones Est. Indexada Vs Enlazada	
Concepto de punteros, operadores, asignación dinámica de memoria	
Estructuras autoreferenciadas	
Pilas	
Definición, push, pop	
Ejemplos de aplicación	
Colas	
Definicion, queue, unqueue	
Ejemplos de aplicación	
Listas simplemente ordenadas	
Definición, insertar ordenado	
Buscar, insertar sin repetir	
Carga sin repetir la clave	
Vectores de listas, Lista de listas	
Listas doblemente enlazadas, listas circulares	
Ejemplos de aplicación	
Ejercitación	
Ejercicios integradores, combinaciones complejas de estructuras	

Práctica	
Criterio de selección de estructuras	
Práctica	
Estructuras arbóreas	
Inserción y recorridos	

Combinaciones complejas de estructuras - Ejercitación

Tema	
Vectores/matrices de estructuras estáticas y/o enlazadas	
Estructuras enlazadas de struct	
struct de punteros	
Estructuras enlazadas de struct con punteros a estructuras enlazadas	
Estructuras enlazadas con struct con un campo con un array	
Estructuras enlazadas de struct	
Ejercitación y práctica	
Ejercicios con archivos	
Ejercicios con vectores y matrices	
Ejercicios con estructuras enlazadas lineales	
Ejercicios con estructuras arbóreas	
Ejercicios con combinaciones complejas de estructuras	

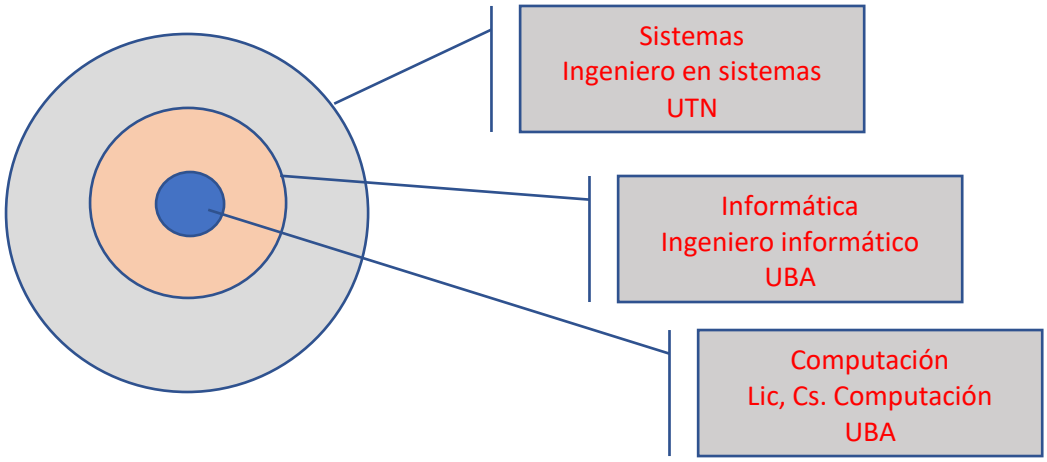
SEMANA 5

EXÁMENES Y RECUPERATORIOS

1

Conceptos preliminares

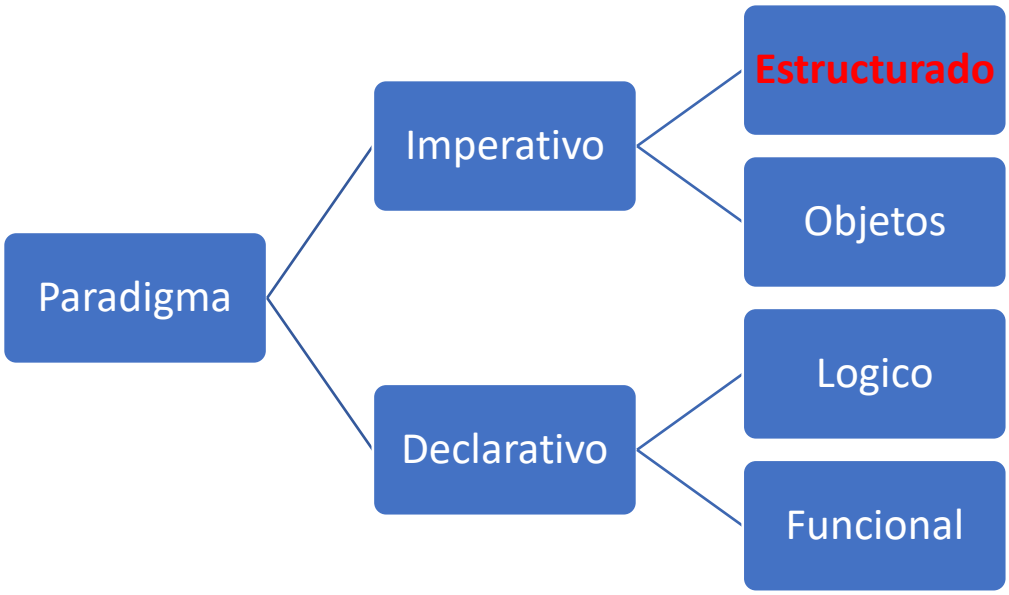
Introducción:



El área de programacion

- **Matemática discreta**
- **Algoritmos y estructura de datos**
- **Sintaxis y semántica de los lenguajes**
- **Paradigmas de programación**

Paradigmas de programación



Matriz Transversal de Contenidos propuesta:

	Generales			Tipos de datos									
	LI	RP	TD	DS	ER	AT	AB	EI	ED	EA	EG	CC	Bib
MD	ALTA	BAJA	ALTA	BAJA	BAJA	BAJA	BAJA	BAJA	BAJA	BAJA	BAJA	NO	NO
AyE	**ALTA**	**ALTA**	**ALTA**	**ALTA**	**ALTA**	**ALTA**	**ALTA**	**ALTA**	**ALTA**	**MED.**	**BAJA**	**ALTA**	**MED.**
SSL	ALTA	MED.	ALTA	BAJA	MED.	MED.	MED.	MED.	BAJA	MED.	ALTA	MED.	ALTA
PP	ALTA	MED.	ALTA	BAJA	BAJA	MED.	MED.	MED.	BAJA	MED.	BAJA	BAJA	ALTA

Programación

La programación es una actividad transversal asociada a cualquier área de la informática, aunque es la ingeniería del software el área específica que se ocupa de la creación del software. En principio la programación se veía como un arte, solo era cuestión de dominar un lenguaje de

Programa:

Programa: conjunto de instrucciones no activas almacenadas en un computador, se vuelve **tarea** a partir de que se selecciona para su ejecución y permite cumplir una función específica. Un **proceso** es un programa en ejecución.

Dato

Dato representación de un objeto el mundo real mediante el cual se pueden modelizar aspectos de un problema que se desea resolver con un programa en una computadora.
<dato> -> <objeto><atributo><valor>

Lenguaje de programación

Conjunto de instrucciones permitidas y definidas por sus reglas sintácticas y su valor semántico para la expresión de soluciones de problemas.

Problema: Enunciado con una incognita

Tipo de problema: no computables; **Computables** (tratables, intratables)

Etapas de resolución de problemas con computadoras.

1. Análisis del problema: en su contexto del mundo real.
2. Diseño de la solución: Lo primero es la modularización del problema, es decir la descomposición en partes con funciones bien definidas y datos propios estableciendo la comunicación entre los módulos.
3. Especificación del algoritmo: La elección adecuada del algoritmo para la función de cada modulo es vital para la eficiencia posterior.
4. Escritura del programa: Un algoritmo es una especificación simbólica que debe convertirse en un programa real sobre un lenguaje de programación concreto.
5. Verificación: una vez escrito el programa en un lenguaje real y depurado los errores sintácticos se debe verificar que su ejecución conduzca al resultado deseado con datos representativos del problema real.

Algoritmo

Algoritmo
Especificación rigurosa (debe expresarse en forma univoca) de la secuencia de pasos, instrucciones, a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito. Esto último supone que el algoritmo empieza y termina, en el caso de los que no son de tiempo finito (ej. Sistemas en tiempo real) deben ser de número finito de instrucciones.

Un algoritmo debe tener al menos las siguientes características:

1. **Ser preciso**:
2. **Ser definido**.
3. **Ser finito**:
4. **Presentación formal**:
5. **Corrección**:
6. **Eficiencia**:

Propiedades de los algoritmos

1. Especificación precisa de la entrada:
2. Especificación precisa de cada instrucción:
3. Un algoritmo debe ser exacto y correcto, tener etapas bien definidas y concretas.
4. Debe ser fácil de entender, codificar y depurar.
5. Debe hacer uso eficiente de los recursos de la computadora

Identificadores

Nombre simbolico que define quien programa para denotar “identificar” ciertos elementos en la aplicación o programa.
Estos elementos pueden ser:

1. Variables →asociado a un espacio de almacenamiento en la memoria de un ordenador que contiene un valor que puede ser modificado (variable) durante el proceso. Ejemplo **int i = 0**; sentencia que identifica con el nombre simpolico i a un espacio de memoria que inicialmente tiene el valor entero o pero que puede ser modificado asignándole el resultado de una expresión entera.

2. Constante → asociado a un espacio de memoria que contiene un valor FIJO que no cambia ni puede ser cambiado durante el proceso o la ejecución del programa. Solo puede ser modiicado editando el programa, modificando el valor y a través de una nueva compilacion. Se pueden declarar de dos formas:
    a. Mediante la utilización de la palabra reservada *const* → **const float PI = 3.1416**;
    b. Con la directiva de preprocesador → **# define PI 3.1416**. En este caso cada aparición de el identificador PI es reemplazado por el valor 3.1416
3. Funciones → asociado a un espacio de memoria que contiene líneas de código que se especializan en realizar una acción determinada **int miFuncion(int, int**);

ValorL

Siendo int a=10, b=5; c;

La sentencia c = a+b; le asigna a c la suma de los valores de a y b. el identificador c esta situado a la izquierda (left) de la asignación, por esta condicion recibe el nombre de valor. Solo pueden ser ValorL las variables, las constantes no cumplen con esto dado que su valor, como se menciono, no se puede modificar durante la ejecución del programa, solo puede ser cambiado modificando el código y compilando nuevamente.

Declaraciones y definiciones

Una declaración es una construcción que especifica las propiedades de un identificador : declara lo que una palabra (identificador) "significa". Especifica la existencia y la semántica de los identificadores al compilador ya sea el tipo de dato para variables y constantes, o la firma para funciones). declaración asocia un nombre a un determinado tipo de dato, lo que brinda información fundamental al compilador. No hace nada más que la asociación: relaciona una etiqueta con un tipo. Cuando esta etiqueta se “asociada” con una entidad concreta para hacer uso de las operaciones permitidas sobre esta entidad, ya sea un dato (variable o constante) o un algoritmo (una función).

El término "declaración", entonces, se contrasta con el término "definición". C es fuertemente tipado por lo que las declaraciones son imprescindibles. Si una declaración de una constante o variable especifica el valor permanente de la constante o el valor inicial de una variable, algunos lenguajes lo llaman definición, si solo especifica el tipo declaración. De forma similar se puede pensar las funciones, la declaración es la firma y la definición la implementación.

La declaración se completa con la definición, es aquí donde se concreta la creación de la entidad. Las declaraciones pueden definir y/o referenciar, si además reserva almacenamiento a un objeto o función es una definición.

Expresiones y sentencias

Por definición una expresión es un conjunto de operadores y operandos que reducen a un valor, por ejemplo siendo a un entero con valor 5 y b otro entero con valor 10; a+b es una expresión que tiene operadores (+) y operados (a,b) en este caso reducen al valor 5 + 10, es decir 15. Por lo que a+b es una expresión, en este caso aritmética. Siguiendo con a y b y los valores detallados entonces a>b también es una expresión, en este caso el operador es de relación y es “>”.el valor a que reduce esta expresión es uno de dos posibles, puede ser cierto (verdadero) o no cierto (falso) dependiendo de los valores de a y b. Por extensión a también es una expresión, el valor es 5, lo que ocurre es que no es completa ya que arece de operadores.

Una sentencia es una acción que se ejecuta efectivamente para obtener una sentencia en c se debe finalizar con “;”. Ejemplo c=a+b; es una sentencia que le asigna a c el valor de la expresión a+b y termina con un ;

Las sentencias pueden ser:

1. Simples → una única acción → asignación
2. Estructurada → responden a un formato o estructura determinada → análisis de caso, repeticiones.

3. Compuestas→ una o mas sentencias simples, estructuradas ocombinaciones que se tratan como una unidad **{**sentencia;… ;sentencia**}**

Fundamentos de C/C++

2

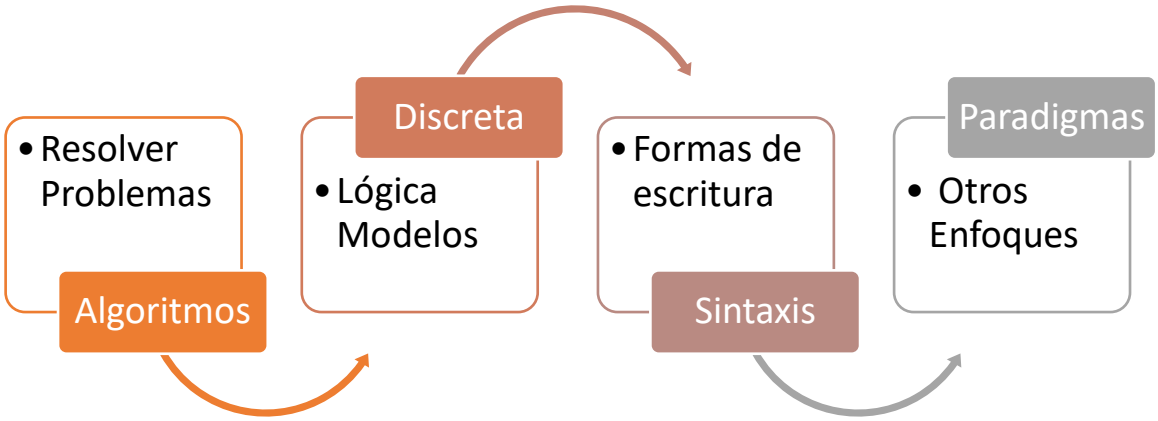
CHE...

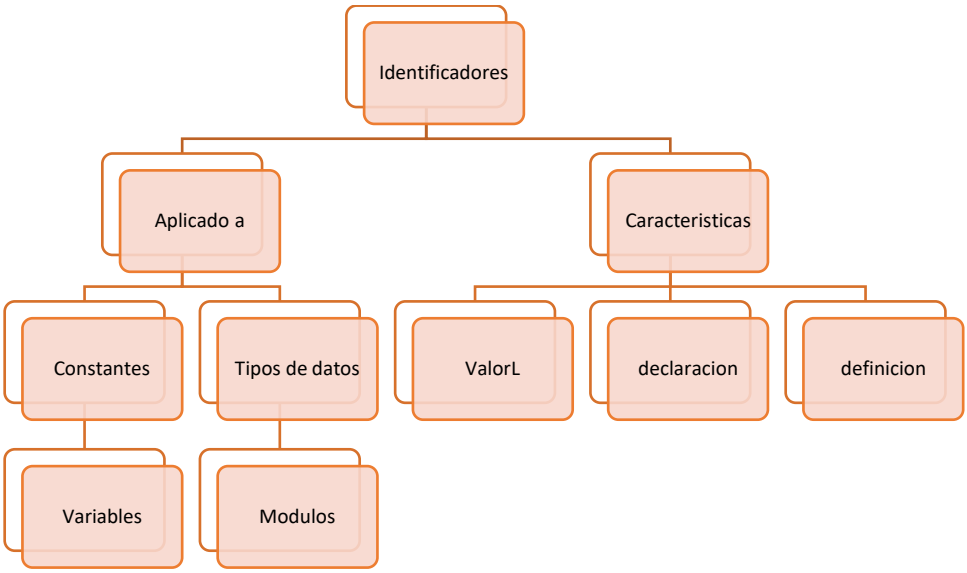
Para aprender a ReconoceR el problema

Y Alcanzar la solución mediante

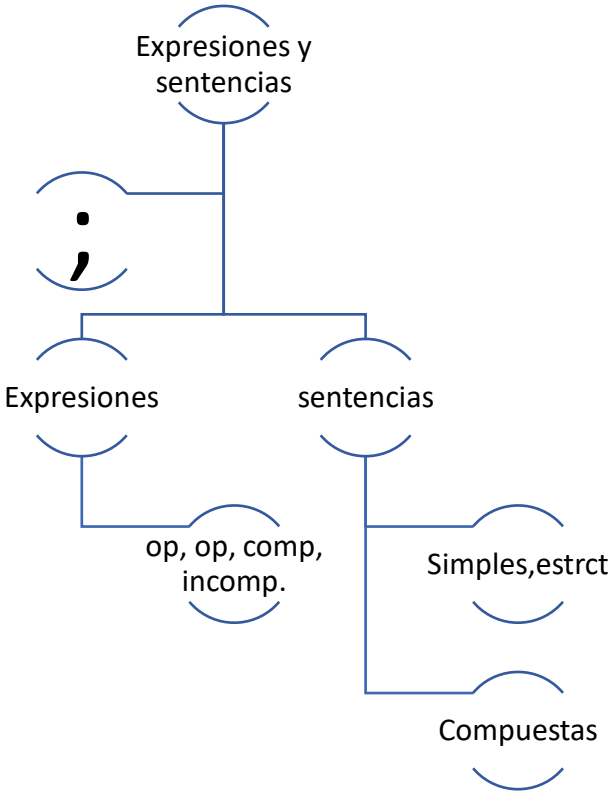
- Habilidad, Esfuerzo --> Ejercitacion
- Creatividad --> redefinicion
- Habilidad instrumental
- Conocimiento heuristico

Area programación





Expresiones y sentencias



Fundamentos C++

Sin declaración using	Con declaración using

//programa para imprimir texto
#include <iostream>
 	
int main()
{
  std::cout << “Hola\n”;
 	
  return 0;
}
	//programa para imprimir texto
	#include <iostream>
	using std::cout; //
	using std::cin;
	using std::endl;
	
	int main()
	{
	  cout << “Hola” << endl;
	
	  return 0;

Instruccion	Descripcion
#include	Directiva del preprocesadpr
<iostream>	Componente de entrada/salida (objetos cin, cout,cerr)
using	Declaración que elimina necesidad de repetir el prefijo std.
int main()	Funcion principal que retorna un entero
{ }	Definición de un bloque de programa
std::cout	Uso del nombre cout del espacio de nombres std, dispositivo std de salida
::	Operador binario de resolución de alcance
<<	Operador de inserción en flujo
“Hola\n”	Literal Hola + salto de línea (también << std::endl;
;	Finalización de una sentencia
return 0	Punto de finalización correcta de la función

Si la función, como en este caso tiene un encabezado int main() debe tener al menos un return de un valor entero. Una función void nombre() puede finalizar con la instrucción return o sin ella.

Programa que muestra la suma de dos enteros

```
# include <iostream>
int main()
{
   // declaracion de variables
   int numero1;
   int numero2;
   int suma;

   std::cout << “Escriba el primer entero”;
   std::cin >>numero1;

   std::cout << “Escriba el segundo entero”;
   std::cin >>numero2;

   suma = numero1 + numero2;

   std::cout << “La suma de “ <<numero1 << “ + ” <<numero2 << “ es: “ << suma << std::endl;
   return 0;
}
```

Instrucción	Descripcion
cin	Dispositivo std de entrada
>>	Operador de extracción de flujo
+	Operador de suma
-	Operador de resta
*	Operador multiplicativo
/	Operador de división
%	Operador de modulo o resto
( )	Operador para agrupar expresiones ej: a * (b+c)
==	Operador de igualdad
>	Mayor
>=	Mayor igual
<	Menor
>=	Menor igual
!=	Operador de desigualdad
=	Operador de asignación
+=	Asignación y suma x+=3; equivale a x = x + 3;
-=	Resta y asignación
*=	Multiplicación y asignación
/=	División y asignación
++	Operador de incremento
--	Operador de decremento

Programa que comprara dos enteros, utiliza la declaración using

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
   int numero1;
   int numero2;
   cout << “Escriba dos enteros para comparar”;
   cin >>numero1 >> numero2;

   if (numero1 > numero2)
   cout <<  numero1 << “ > ” <<numero2 << std::endl;

   if (numero1 == numero2)
   cout <<  numero1 << “ == ” <<numero2 << std::endl;

   if (numero1 < numero2)
   cout <<  numero1 << “ < ” <<numero2 << std::endl;
   return 0;
}
```

Palabras Reservadas

C y C++

Auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while

Solo C++

and and_eq asm bitand bitor bool catch class compl const_cast delete dynamic_cast explicit export false friend inline mutable namespace new not not_eq operator or or_eq private protected public reinterpret_cast static_cast template this throw true try typeid typename using virtual wchar_t xor xor_eq

Tipos de datos fundamentals y jerarquia de promoción

Tipos de Datos
long double
double
float
unsigned long int
long int
unsigned int
int
unsigned short int
short int
unsigned char
char
bool

Espacios de nombres

include <iostream>

Sin incluir directiva ni declaracion using

std::cout << “Hola” << std::endl;

#include <iostream>
Con declaracion using
//la declaración using de cada elemento solo incorpora los nombres especificados
using std::cin; usa el objeto cin del espacio std
using std::endl;
cout << “Hola” <<endl;
#include <iostream>
Con directiva using
//la directiva using incorpora la totalidad de los nombres del espacio de nombre
using namespace std; usa el espacio de nombre std
cout << “Hola” <<endl;
……..
Espacios de nombres
Un programa incluye muchos identificadores definidos con distintos alcances. Una variable de un alcance se “traslapa” si entra en conflicto con una del mismo nombre y distinto alcance.para evitar esto se utilizan los espacios de nombres.
Para acceder a un miembro de un espacio de nombre se lo califica con el nombre del espacio, el operador de resolución de alcance y el nombre del miembro std::cout ….
Otras forma de uso de los espacios de nombre es mediante declaración using o directiva using.

//demostración de espacios de nombre

```
#include <iostream>
using namespace std; //usa el espacio de nombre std
int entero1 = 98 //variable global
//crea espacio de nombres ejemplo
namespace Ejemplo//declara dos constantes y una variable{
 const doublé PI = 3.14;
 const doublé E  = 2.71;
 int entero1 = 8;
 void imprimirValores(); //prototipo
      namespace interno// espacio de nombres anidado       {
        enum Anios {FISCAL1 = 1990, FISCAL2, FISCAL3};
       } // fin espacio de nombre interno
} //fin espacio de nombre Ejemplo

namespace //crea espacio de nombre sin nombre{
    Doublé doubleSinNombre = 88.22;
}
Int main ()
   cout << doubleSinNombre;
   cout << entero1; //imprime la variable global
   cout << Ejemplo::entero1; // imprime entero1 del espacio de nombre Ejemplo
   Ejemplo::imprimirValores(); //invoca a la función
   Cout << Ejemplo::Interno::FISCAL1; //imprime el valor del espacio de nombre interno
```

Resumen:

1. Los comentarios de una linea comienzan con //, esta línea es omitida por el compilador.
2. Las directivas del preprocesador comienzan con #, estas líneas no terminan con ; ya que no son parte de C. Permiten incorporar archivos de encabezado. <iostream> contiene información necesaria para utilizar cin y cout-
3. Los programas en C ejecutan la función main.
4. Toda sentencia termina con ;

5. La declaración using std::cout informa al compilador que puede encontrar a cout en el espacio de nombre std y elimina la necesidad de repetir el prefijo std.

Algunas funciones de biblioteca

Definiciones Comunes <stddef.h>

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

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

Manejo de Cadenas <string.h>

```
char* strcpy (char* s, const char* t);
```
Copia la cadena **t** en **s** (es la asignación entre cadenas). Retorna (**s**).

Búsqueda y Comparación

```
int strcmp (const char*, const char*);
```
Compara "lexicográficamente" ambas cadenas. Retorna (0 si las cadenas son iguales; < 0 si la 1ra. es "menor" que la 2da.; > 0 si la 1ra. es "mayor" que la 2da.)

Tips y Macros

```
EXIT_FAILURE
EXIT_SUCCESS
```

Sentencias

```
<sentencia> ->
     <sentencia expresión> |
     <sentencia compuesta> |
     <sentencia de selección> |
     <sentencia de iteración> |
     <sentencia etiquetada> |
     <sentencia de salto>

<sentencia expresión> ->
     <expresión>? ;

<sentencia compuesta> ->
     {<lista de declaraciones>? <lista de sentencias>?}
<lista de declaraciones> ->
     <declaración> |
     <lista de declaraciones> <declaración>
<lista de sentencias> ->
     <sentencia> |
     <lista de sentencias> <sentencia>
```

- La sentencia compuesta también se denomina *bloque*.

```
<sentencia de selección> ->
     if (<expresión>) <sentencia> |
     if (<expresión>) <sentencia> else <sentencia> |
     switch (<expresión>) <sentencia>
La expresión que controla un switch debe ser de tipo entero.

<sentencia de iteración> ->
     while (<expresión>) <sentencia> |
     do <sentencia> while (<expresión>) ;  |
     for (<expresión>? ; <expresión>? ; <expresión>?) <sentencia>

<sentencia etiquetada> ->
     case <expresión constante> : <sentencia> |
     default : <sentencia> |
     <identificador> : <sentencia>
Las sentencias case y default se utilizan solo dentro de una
     sentencia switch.

<sentencia de salto> ->
     continue ; |
     break ; |
     return <expresión>? ; |
     goto <identificador> ;
```

- La sentencia **continue** solo debe aparecer dentro del cuerpo de un ciclo. La sentencia **break** solo debe aparecer dentro de un **switch** o en el cuerpo de un ciclo. La sentencia **return** con una expresión no puede aparecer en una función **void**.

Estilos de Indentación

A continuación se muestran diferentes estilos, la elección es puramente subjetiva, pero su aplicación al largo de un mismo desarrollo debe ser consistente.

Estilo K&R – También conocido como "The One True Brace Style"

El libro **[K&R1988]** usa siempre este estilo, salvo para las definiciones de las funciones, que usa el estilo **BSD/Allman**. Las Secciones principales de **Java** también usan este estilo. Este es el estilo que recomienda y que usa la Cátedra para todas las construcciones.

```
while( SeaVerdad() ) {
      HacerUnaCosa();
      HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo BSD/Allman

Microsoft Visual Studio 2005 impone este estilo por defecto. Nuevas secciones de Java usan este estilo. Es un estilo recomendable.

```
while( SeaVerdad() )
{
      HacerUnaCosa();
      HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo Whitesmiths

```
while( SeaVerdad() )
  {
       HacerUnaCosa();
       HacerOtraCosa();
  }
HacerUnaUltimaCosaMas();
```

Estilo GNU

```
while( SeaVerdad() )
{
      HacerUnaCosa();
      HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo Pico

```
while( SeaVerdad()
{ HacerUnaCosa();
HacerOtraCosa(); }
HacerUnaUltimaCosaMas();
```

Tipos de Datos

3

Dato

Dato representación de un objeto el mundo real mediante el cual se pueden modelizar aspectos de un problema que se desea resolver con un programa en una computadora.
<dato> -> <objeto><atributo><valor>

Algoritmo

Algoritmo
Especificación rigurosa (debe expresarse en forma univoca) de la secuencia de pasos, instrucciones, a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito. Esto último supone que el algoritmo empieza y termina, en el caso de los que no son de tiempo finito (ej. Sistemas en tiempo real) deben ser de número finito de instrucciones.

Identificadores

Nombre simbolico que define quien programa para denotar “identificar” ciertos elementos en la aplicación o programa.
Estos elementos pueden ser:

4. Variables →asociado a un espacio de almacenamiento en la memoria de un ordenador que contiene un valor que puede ser modificado (variable) durante el proceso.
5. Constante → asociado a un espacio de memoria que contiene un valor FIJO que no cambia ni puede ser cambiado durante el proceso o la ejecución del programa. Solo puede ser modiicado editando el programa, modificando el valor y a través de una nueva compilacion.
6. Funciones → asociado a un espacio de memoria que contiene líneas de código que se especializan en realizar una acción determinada **int miFuncion(int, int**);

ValorL
Solo pueden ser ValorL las variables, las constantes no cumplen con esto dado que su valor no se menciono, no se puede modificar durante la ejecución del programa.

Declaraciones y definiciones

Una Declaracion es una construcción que especifica las propiedades de un identificador, declara lo que una palabra (identificador) "significa". Si una declaración de una constante o variable especifica el valor permanente de la constante o el valor inicial de una variable, algunos

lenguajes lo llaman definición, si solo especifica el tipo declaración. De forma similar se puede pensar las funciones, la declaración es la firma y la definición la implementación.
La declaración se completa con la definición, es aquí donde se concreta la creación de la entidad. Las declaraciones pueden definir y/o referenciar, si además reserva almacenamiento a un objeto o función es una definición.

Tipos de datos

Los tipos de datos Identifican o determinan un dominio de valores y un 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.

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**.

Criterios de selección

1. Priorizar de ser posible acceso directo y velocidad de procesamiento.
    a. Vector en primer lugar si se cumple tamaño fijo, razonable y conocido a priori, y sin necesidad de persistencia→ acceso directo, búsqueda binaria, búsqueda secuencial
2. Si no se conoce el tamaño y no se requiere persistencia
    a. Estructuras enlazadas
        i. Pila si se debe invertir el orden o si es irrelevante
        ii. Colas si se debe mantener
        iii. Listas si se debe generar
3. Archivo si se requiere persistencia: directamente o estructuras auxiliares después cargar al archivo

Toma de decisiones

1. Los datos se ingresan desde el teclado

a. Se deben mostrar en la misma secuencia de entrada: No es necesario guardarlos en una estructura auxiliar. Así como los recibimos se deben mostrar, no se requiere conservarlos en memoria para ningún procesamiento posterior.
b. Se requiere procesamiento posterior. Supongamos que debemos mostrarlos ordenados por un criterio diferente al ingreso: en este caso se debe generar una estructura auxiliar, esta puede ser un vector o lista si el ordenamiento es por un campo¸ matriz, vector de listas o lista de listas si el ordenamiento es por dos criterios

2. Los datos se ingresan desde un archivo físico
   a. Similar al ingreso por teclado → solo cambia el origen del dato
   b. Similar al ingreso por teclado → solo cambia el origen del dato
3. Secuencia de decisiones
   a. Origen del dato
      i. Teclado
      ii. Archivo
   b. Elección de estructura auxiliar
      i. Solo se muestra con el criterio de ingreso; no se requiere estructura auxiiiar
      ii. Se requiere reordenar o conservar para buscar: Requiere auxiliar
         1. Orden por un campo el tamaño es conocido a priori
            a. Vector
               i. Carga directa
               ii. Carga secuencial
                  1. Ordenar posteriormente
                  2. Cargar ordenada
                  3. Dejar sin orden, búsqueda secuencial
         2. Orden por un campo tamaño no conocido
            a. Lista ordenada
               i. Insertar ordenado
               ii. CargarSinRepetir
               iii. InsertarOrdenado
         3. Orden por dos campos
            a. Ambos conocidos y acotados
               i. Vector de vector → matriz
            b. Uno acotado y definido, el otro no
               i. Vector de punteros
            c. Ambos No acotados
               i. Lista de listas
   c. Que guardar en la estructura auxiliar
      i. Todos los datos si se requieren
      ii. Selo los que se requieren como salida, no hacer substancia de lo superfluo
      iii. La clave de ordenamiento y la referencia al dato

Tipos de datos simples – Tamaño y Rango

		Implementaciones					
Nombre del Tipo	Otros Nombres	**Borland Turbo C++ 3.0** *16 bits*		**Borland C++ 4.02** *32 bits*		**Microsoft Visual Studio .NET** *32 bits*	
		Bytes	Rango	Bytes	Rango	Bytes	Rango

Char	signed char	1	-128 .. 127 [-2^7 .. (2^7-1)] (ASCII Standard)		
unsigned char	–	1	0 .. 255 [0 .. (2^8-1)] (ASCII Extendido)		
Short	short int, signed short, signed short int	2	–32,768 .. 32,767 [-2^{15} .. (2^{15}-1)]		
unsigned short	unsigned short int	2	0 .. 65,535 [0 .. (2^{16}-1)]		
Int	Signed, signed int	2	–32,768 .. 32,767 [-2^{15} .. (2^{15}-1)]	4	–2,147,483,648 .. 2,147,483,647 [-2^{31} .. (2^{31}-1)]
unsigned int	unsigned	2	0 .. 65,535 [0 .. (2^{16}-1)]	4	0 .. 4,294,967,295 [0.. (2^{32}1)]
Long	long int, signed long, signed long int	4	–2,147,483,648 .. 2,147,483,647		
unsigned long	unsigned long int	4	0 .. 4,294,967,295		
Enum	–	*igual a int*			
Float	–	4	3.4 x 10^{-38} .. 3.4 x 10^{38} (7 dígitos de precisión)		
Double	–	8	1.7 x 10^{-308} .. 1.7 x 10^{308} (15 dígitos)		
long double	–	10	3.4 x 10^{-4932} .. 1.1 x 10^{4932} (19 dígitos)		*igual a double*

Precedencia y Asociatividad de los 45 Operadores

ID Asociatividad Izquierda-Derecha, DI Asociatividad Derecha-Izquierda

1	ID	operadores de acceso	`( )`			invocación a función
			`[ ]`			subíndice de arreglo
			`.`			acceso a struct y a union
			`->`			acceso a struct y a union
2	DI	operadores unarios (operan sobre un solo operando)	`+`	`-`		signos positivo y negativo
			`~`			complemento por bit
			`!`			NOT lógico
			`&`			dirección de
			`*`			"indirección"
			`++`			pre-incremento
			`--`			pre-decremento
			`sizeof`			tamaño de
			`(`*tipo*`)`			conversión explícita
3	ID	operadores multiplicativos	`*`			multiplicación
			`/`			cociente
			`%`			módulo o resto
4	ID	operadores aditivos	`+`	`-`		suma y resta
5	ID	operadores de desplazamiento a izquierda	`<<`			desplazamiento de bits
			`>>`			desplazamiento de bits a derecha
6	ID	operadores relacionales	`<`	`>`		
			`<=`	`>=`		
7	ID	operadores de igualdad	`==`	`!=`		igual a y distinto de
8	ID	operadores binarios por bit	`&`			AND
9	ID		`^`			OR exclusivo
10	ID		`|`			OR
11	ID	operadores binarios lógicos	`&&`			AND
12	ID		`||`			OR
13	DI	operador condicional (único que opera sobre 3 operandos)	`? :`			
14	DI	operadores de asignación	`=`			
			`*=`	`/=`	`%=`	`+=` `-=`
			`<<=`	`>>=`	`&=`	`^=` `|=`
15	ID	operador concatenación expresiones	`,`			"coma"

- Los operadores **`&&`** , **`||`** y **`,`** "coma" son los únicos que garantizan que los operandos sean evaluados en un orden determinado (de izquierda a derecha).

- El operador condicional (**`? :`**) evalúa solo un operando, entre el 2do. y el 3ro., según corresponda.

Algunas consideraciones

El resultado de la división de dos enteros es un entero, la división de enteros es, entonces una división entera, por tanto siendo:

Int a=10, b=2, c=3;
float d = 3.0, e=10.0;

Determine los resultados de

a/b
a/c
a/3.0
a/d
e/d

Justifique la respuesta→ Ayuda: determine que es jerarquía de datos, cual es la secuencia (char, int, long, float, double), que pasa en una expresión con datos de diferentes jerarquías, busque la definición casteo explicito, implicito

Es posible asignar a=b=c=15; que valores asigna a que identificadores, porque, es la signacion = un operador o una sentencia?

Siendo a=10; que efecto produce a++; y si hubiera sido ++a? que hubiera pasado con –

Siendo int a = 10, b;
¿Que valores toman a y b luego de b=a++;? ¿Por qué?

Siendo int a = 10, b;
¿Que valores toman a y b luego de b = ++a?; ¿Por qué?

Que valor toman a y b luego de a+=b; y a*=b, Por qué?

Sea int a;
Es posible la asignación a = 'A'+'B'; esta sentencia le asigna valor al identificador a?, ¿que valor? ¿Por que?

Ejercicios

1. Cuál de las siguientes sentencias son correctas para la ecuación algebraica $y=ax^3 + 7$.
    a. y = a * x * x * x + 7
    b. y = a * x * x * (x + 7)
    c. y = (a * x) * x *(x + 7)
    d. y = (a * x) * x * x + 7
    e. y = a * (x * x * x) + 7
    f. y = a * (x * x * x + 7)
2. Escriba un programa que pida al ususrio dos números e informe la suma, la resta, el producto y el cociente de los mismos
3. Imprima un programa que imprima los números del 1 al 4 en una misma línea, hágalo de las formas siguientes:
    a. Utilizando un solo operador de inserción de flujo
    b. Una única sentencia con 4 operadores de inserción de flujo
    c. Utilizando cuatro sentencias
4. Escriba un programa que reciba tres números por el teclado e imprima la suma, el promedio, el producto, el mayor y el menor de esos números. Escriba un adecuado dialogo en pantalla.
5. Escriba un programa que reciba un numero que represente el radio de un circulo e imprima el diámetro, circunferencia y área.
6. Que imprime el siguiente código
    a. std::cout << "*\n**\n***\n****" <<std::endl;
    b. std::cout << 'A';
    c. std::cout << static_cast< int > 'A'; (que es static_cast? Investigue.)
7. Utilizando solo lo que hemos desarrollado en esta introduccion escriba un programa que calcule los cuadrados y los cubos de los números de 0 a 10 y los muestre por pantalla.
8. Escriba un programa que reciba un numero entero de 5 digitos, que separe el numero en sus digitoe y los muestre por pantalla, uno por línea comenzando por elmas significacivo en la primera línea.

9. Dados dos valores enteros y positivos determinar y mostrar por el dispositivo estándar de salida: la suma, la resta, el producto y la división de los mismos. Analice precondiciones adecuadas y utilice leyendas adecuadas. Resuelva teniendo en cuenta las precondiciones y da una solución alternativa sin considerarlas.
10. Responda: que pasa si el conjunto de datos es float, si es entero y sin la restricción de ser positivo.
11. Dada una terna de números naturales <dia, mes, año> que representan al día, al mes y al año de una fecha informarla como un solo número natural de 8 dígitos (AAAAMMDD). Establezca precondiciones.
12. Dada un número natural de 8 dígitos, con formato (AAAAMMDD) descompóngalo en sus elementos lógicos y muéstrelos por el dispositivo estándar de salida.
13. Dado un valor entero determinar y mostrar: la quinta parte del mismo, el resto de la división por 5 y la tercera parte del valor del primer punto. Resuelva sin precondiciones, informando por la salida estándar los resultados.
14. Dado una terna de valores determine e imprima: El mayor y el menor del conjunto. Muestre por salida estándar con las aclaraciones que considere.
15. Que cambios estratégicos produciría si el conjunto de valores ingresados fueran 20? Y si el lote fuera de una cantidad no conocida a priori?
16. Dadas dos ternas de valores que representan las fechas validas de nacimiento de dos personas, indique cual de las dos corresponde al mayor. Utilice las leyendas que crea corresponden.

17. Dado un par de valores que representa una el mes y año de una fecha valida del siglo XXI determinar e imprimir la cantidad de días de ese mes.
18. Dado una terna de valores enteros y positivos determinar si los mismos forman un triangulo.
19. Dado una terna de valores que representan los lados de un triangulo determinar si el tipo de triangulo que forman.
20. Dados dos números naturales M y N, determinar e imprimir cuantos múltiplos de M hay el el conjunto 1 a N.
21. Dados dos números enteros, M y N informar su producto por sumas sucesivas.
22. Dados un conjunto de valores enteros informar el promedio de los mayores que45 y la suma de los menores que –10.
23. Dado un valor M determinar e imprimir los M primeros múltiplos de 3 que no lo sean de 5, dentro del conjunto de los números naturales.
24. Dados un conjunto de valores enteros y positivos determinar e informar el mayor
25. Dado un conjunto de N valores informar el mayor, el menor y en qué posición del conjunto fueron ingresados.
26. Dado un conjunto de valores reales, que finaliza con un valor nulo, determinar e imprimir (si hubieron valores): el máximo de los negativos y el minimo de los positivos

Algunas funciones de Biblioteca de C

Manejo de Caracteres <ctype.h>

`int isalnum (int);`
Determina si el carácter dado `isalpha` o `isdigit` Retorna (ok ? ≠0 : 0).
`int isalpha (int);`
Determina si el carácter dado es una letra (entre las 26 minúsculas y mayúsculas del alfabeto inglés). Retorna (ok ? ≠0 : 0).
`int isdigit (int);`
Determina si el carácter dado es un dígito decimal (entre '0' y '9'). Retorna (ok ? ≠0 : 0).
`int islower (int);`
Determina si el carácter dado es una letra minúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)
`int isprint (int);`
Determina si el carácter dado es imprimible, incluye al espacio. Retorna (ok ? ≠0 : 0)
`int isspace (int);`
Determina si el carácter dado es alguno de estos: espacio (`' '`), `'\n'`, `'\t'`, `'\r'`, `'\f'`, `'\v'`. Retorna (ok ? ≠0 : 0)
`int isupper (int);`
Determina si el carácter dado es una letra mayúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)
`int isxdigit (int);`
Determina si el carácter dado es un dígito hexadecimal (`'0'..'9'`, `'a'..'f'` o `'A'..'F'`). Retorna (ok ? ≠0 : 0)
`int tolower (int c);`
Si `c` es una letra mayúscula (entre las 26 del alfabeto inglés), la convierte a minúscula.
Retorna (mayúscula ? minúscula : `c`)
`int toupper (int c);`
Si `c` es una letra minúscula (entre las 26 del alfabeto inglés), la convierte a mayúscula.
Retorna (minúscula ? mayúscula : `c`)

Manejo de Cadenas <string.h>

Define el tipo `size_t` y la macro `NULL`
`unsigned strlen (const char*);`
Cuenta los caracteres que forman la cadena dada hasta el 1er carácter `'\0'`, excluido. Retorna (longitud de la cadena).

Concatenación

`char* strcat (char* s, const char* t);`
Concatena la cadena `t` a la cadena `s` sobre `s`. Retorna (`s`).
`char* strncat (char* s, const char* t, size_t n);`
Concatena hasta `n` caracteres de `t`, previos al carácter nulo, a la cadena `s`; agrega siempre un `'\0'`. Retorna (`s`).

Copia

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

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

```
char* strcpy (char* s, const char* t);
```
Copia la cadena **t** en **s** (es la asignación entre cadenas). Retorna (**s**).

Búsqueda y Comparación

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

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

```
int strcmp (const char*, const char*);
```
Compara "lexicográficamente" ambas cadenas. Retorna (0 si las cadenas son iguales; < 0 si la 1ra. es "menor" que la 2da.; > 0 si la 1ra. es "mayor" que la 2da.)

```
int strncmp (const char* s, const char* t, size_t n);
```
Compara hasta **n** caracteres de **s** y de **t**. Retorna (como **strcmp**).

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

Conversión

```
double atof (const char*);
```
Convierte una cadena que representa un real **double** a número **double**. Retorna (número obtenido, no necesariamente correcto).

```
int atoi (const char*);
```
Convierte una cadena que representa un entero **int** a número **int**. Retorna (número obtenido, no necesariamente correcto) .

```
long atol (const char*);
```
Convierte una cadena que representa un entero **long** a número **long**. Retorna (número obtenido, no necesariamente correcto).

```
double strtod (const char* p, char** end);
```
Convierte como **atof** y, si el 2do. argumento no es **NULL**, un puntero al primer carácter no convertible es colocado en el objeto apuntado por **end**. Retorna como **atof**.

```
long strtol (const char* p, char** end, int base);
```
Similar a **atol** pero para cualquier base entre 2 y 36; si la base es 0, admite la representación decimal, hexadecimal u octal; ver **strtod** por el parámetro **end**. Retorna como **atol**.

```
unsigned long strtoul (const char* p, char** end, int base);
```
Igual que **strtol** pero convierte a **unsigned long**. Retorna como **atol**, pero **unsigned long**.

Números Pseudo-Aleatorios

```
int rand (void);
```
Determina un entero pseudo-aleatorio entre 0 y **RAND_MAX**. Retorna (entero pseudo-aleatorio).

```
void srand (unsigned x);
```
Inicia una secuencia de números pseudo-aleatorios, utilizando a **x** como semilla. No retorna valor.

Matemática

`int abs(int i);`
`long int labs(long int i);`
`<stdlib.h>` Calcula el valore del entero `i`. Retorna (valor absoluto de `i`).
`double ceil (double x);`
`<math.h>` Calcula el entero más próximo, no menor que `x`. Retorna (entero calculado, expresado como `double`).
`double floor (double x);`
`<math.h>` Calcula el entero más próximo, no mayor que `x`. Retorna (entero calculado, expresado como `double`).
`double pow (double x, double z);`
`<math.h>` Calcula x^z; hay error de dominio si $x < 0$ y z no es un valor entero, o si x es 0 y $z \neq 0$. Retorna (ok ? x^z : error de dominio o de rango).
`double sqrt (double x);`
`<math.h>` Calcula la raíz cuadrada no negativa de `x`. Retorna ($x \geq 0.0$? raíz cuadrada : error de dominio).

Asignación - Análisis de caso - Repeticion

4

Asignación

Asignar significa dar valor a cierta dirección de memoria. Se asocia a un espacio de almacenamiento en la memoria de un ordenador que contiene un valor que puede ser modificado (variable) durante el proceso. La asignación puede ser:

1) Interna

a) Nombre del identificador = expresión;

2) Externa

a) De entrada → lleva a memoria un valor:
   i) Desde el teclado → cin>>
   ii) Desde otro flujo de datos → archivo

b) Sa salida muestra o impreme una dirección de memoria:
   i) Por pantalla → cout<<
   ii) Por otro flujo → archivo

Asignación interna

Dar valor al espacio almacenamiento en memoria asociado a un identificador que refiere a una variable de un determinado tipo de dato, mediante el operador de asignación "=".
La asignación interna es destructiva, esto es reemplaza el valor contenido previamente por el valor de la expresión que se asigna.

La forma genérica de la asignación es:

Nombre del identificador = expresión;

Se debe cumplir con:

- El identificador debe ser valorL
- La expresion debe ser del mismo tipo que el identificador
- La asignación es un operador por lo que **a=b=d=expresión**; le asigna el valor de la expresión a todos los identificadores, asociando de derecha a izquierda
- La expresión puede ser completa o incompleta.

int a = 15; declara a como entera e inicializa asignándole el valor de 15.
a = 3*4; destruye el valor anterior de a reemplazandolo por el resultado de la nueva expresión.
En el ejemplo anterior se asigna al identificador a el valor de la expresión aritmética 3*4.

Tambien es posible asignar expresones lógicas o llamadas booleanas, con dos únicos valores posibles. En C toda expresión verdadera tiene el valor de 1 y expresiones falsas el valor es 0.
int a = 5>4; le asigna al identificador el valor de 1 por ser una expresión verdadesra. Que cree que asigna, en caso de considerar que sea una expresión valida a = 3>2<35. Justifique su respuesta.
C++ tiene como tipo de dato primitivo el tipo bool, en cuyo caso el identificador solo puede tener dos valores posibles.

La asignación a = a+1; le asigna al identificador el valor que tenia incrementado, en este caso en 1. Observese que el identificador a tiene significado diferente sedun se encuentre a la izquierda de la expresión o a la derecha. A la izquierda es el valorl que se modificara, a la derecha refiere al contenido antes de la modificación.

Por ultimo recuerde que se pueden combinar los operadores arimeticos con el de asignación y que existen los operadores ++ y – que pueden ser post o pre ya sea para incrementar o disminuir

Asignacion Externa
Se puede dar valor a un espacio de almacenamiento asociado a un identificador obteniéndolo de un dispositivo externo (asignación externa de entrada) o derivarla hacia un dispositivo externo (asignacion externa de salida.

Asignacion externa de entrada
En C++ el dispositivo externo de entrada es el teclado, esta asociado el identidicador **cin** definido en el espacio de nombre STD. Otros flujos dispositivos para obtener datos son los flujos que pueden ser de texto o binarios. Estos serán objeto de estudio mas adelante.
La asignación externa de entrada con el dispositivo estándar es **cin>>identificador**; donde:

- **cin** es el dispositivo definido en el espacio de nombre std.
- **>>** operador de extracción de flujo
- **Identificador** una variable valorL

La asignación externa de entrada es destructiva.
Asignacion externa de salida

En C++ el dispositivo externo de salida es la pantalla, esta asociado el identidicador **cout** definido en el espacio de nombre STD. Otros flujos dispositivos para obtener datos son los flujos que pueden ser de texto o binarios. Estos serán objeto de estudio mas adelante.
La asignación externa de salida con el dispositivo estándar es **cout<<expresión<<expresión;** donde:

cout es el dispositivo definido en el espacio de nombre std.
<< operador de inserción de flujo
expresion puede ser completa, incompleta, literales

La asignación externa de salida no es destructiva.
Ejemplos
Sea int a=10, b=5;
cout<<a;
muestra 10 por pantalla y la próxima asignación la hace inmediatamente después
cout<<a<<endl;
muestra 10 por pantalla y la próxima asignación la hace en la línea siguiente endl produce un salto de linea
cout<<a<<"\n";
muestra 10 por pantalla y la próxima asignación la hace en la línea siguiente "\n" produce un salto de línea.
cout<<" el valor de a es : "<<a<<" el de b " <<b<<" el producto es "<<a*b<<endl;
muestra →el valor de a es 10 el de b 5 el producto 50, la próxima asignación es en la línea siguiente.

Estructuras de selección o análisis de caso

Si bien las acciones en un algoritmo deben realizarse una después de otra, muchas veces existe la necesidad de realizar acciones diferenciadas según una determinada condición. En este caso los lenguajes ofrecen sentencias estructuradas llamada **estructuras de selección o análisis de casos.** Se debe decidir entre un conjunto de alternativas y hacer en consecuencia. Tambien es muy común que un conjunto de acciones, un bloque de sentencias debe repetirse una determinada cantidad de veces. Para esto los lenguajes ofrecen ciclos o estructuras de repetición.
En este apartado el objeto de estudio serán las estructuras de selección y los ciclos-

Estructuras de selección o análisis de caso

1) Estructura de selección simple
   a) incompleta
   b) completa
2) Estructura de selección multiple
   a) incompleta
   b) completa

Estructura de selección simple

Sentencia de selección if incompleta
forma

```
if (expresión) sentencia;
```

if palabra reservada
(expresion) una expresión encerrada entre parentesis, en general una expresión booleana y la sentencia se ejecuta si esa expresión es verdadera. Si la expresión es aritmética y no booleana considera verdadero si la expresión es diferente de cero y falso en caso de ser cero.
La sentencia solo se cumple bajo la condición de verdadero de la expresión. En caso de ser falso se ejecuta la acción inmediata siguiente al condicional en caso de existir. Sentencia puede ser simple, estructurada o compuesta, en este caso se hace necesario crear el bloque de sentencias entre llaves.

Ejemplos
Sea int a=10, b=5;

If (a > b) cout <<”hola”;
Como a es mayor que b la expresión es verdadera por lo que muestra Hola

If (a < b) cout <<”hola”;
Como a es mayor que b la expresión es falsa por lo que no muestra nada

If (a == b) cout <<”hola”;
Como a es mayor que b la expresión es falsa, no son iguales, por lo que no muestra nada

If (a = b) cout <<”hola”;
Aquí no se utiliza el operador de relación, es el operador de asignación, se le asigna a a el valor de b, 5. Por eso esa expresión vale 5, 5 es distinto de cero, por lo que se entiende como verdadero en cuyo caso muestra Hola

If(a>b) {cout<<”hola”; cout <<”Chau”};

Como a es mayor que b ejecuta las dos sentencias simples compuestas en una sentencia mediante las llaves
If(a>b) {cout<<"hola"; cout <<"Chau"};
Como a es mayor que b ejecuta las dos sentencias simples compuestas en una sentencia mediante las llaves

If(a>b) cout<<"hola"; cout <<"Chau";
Como a esmayor que b muestra Hola, al no tener llaves ejecuta la sentencia simple, Chau lo muestra independientemente del valor de verdad de la expresión

Ejenplo
Ingresar un valor entero e informar si es positivo, negativo o cero
int a;
cin>> a;
if(a>0) cout>>"positivo";
if(a<0) cout>>"negativo";
if(a==0) cout>>"cero";
se analiza cada caso y se responde en consecuencia, de las tres sentencias dos de ellas serán falsas

Sentencia de selección if completa
forma

if (expresión) sentencia1; else sentencia2;

if palabra reservada
(expresion) una expresión encerrada entre parentesis, en general una expresión booleana y dos sentencias. La sentencia1 se ejecuta si esa expresión es verdadera. Si la expresión es aritmética y no booleana considera verdadero si la expresión es diferente de cero y falso en caso de ser cero.
En caso de ser falsa la expresion se ejecuta la sentencia 2.

Ejemplos
Sea int a=10, b=5;

If (a > b) cout <<"hola"; else cout<<"chau";
Como a es mayor que b la expresión es verdadera por lo que muestra Hola

If (a < b) cout <<"hola"; else cout<<"chau";
Como a es mayor que b la expresión es falsa por lo que muestra Chau

```
If(a>b)  {
         cout << "Hola";
         cout << " que tal"
         };
else     {
         cout<<"chau";
         cout<<"hasta luego"
         }
```
Muestra hola que tal por ser una sentencia completa

```
If(a>b)
         cout << "Hola";
```

```
        cout << " que tal"
        ;
else    {
        cout<<"chau";
        cout<<"hasta luego"
        }
```

Al haberse omitido las llaves ejecutara solo hola y allí da por finalizado el condicional, al encontrar luego un else no puede determinar a que in corresponde por lo que generara un errol sintactico ya que la estructura de la freas del condicional es sintácticamente incorrecta.

If anidados

```
int a;
cin>>a;
if (a>0)
        cout << "positivo";
else
        if(a<b)
                cout<<"negativo";
        else
                cout<<" es cero";
```

Analisis de caso multiple completo (con clausula default)

```
switch( ordinal){
  case valor1: accion1;
  case valor2: accion2;
  case valor3: accion3;
  ...
  case valorN: accionN;

  default: accionD;
}
```

default es usada para los valores que no correspondieron en casos anteriores
En general se utiliza un break para salir del switch que permitan comportamientos diferentes según el valor del ordinal

```
Switch( ordinal )
{
  case 1: sentencia1; break;
  case 2: sentencia2; break;
  default: sentenciaD; /* break; */
}
```

El analisis de caso multiple incomplete es similar pero sin causa default, por lo que si la entrada no se corresponde con algun valor no se ejecuta ninguna accion. Cada caso debe responder a un valor unico o una lista no permitiendose subrangos.

Estructura de repeticion, ciclos

La necesidad de iterar

Estas estructuras permiten repetir sentencias una determinada cantidad de veces, en C existent res sentencias que lo permiten y estas son: **for; while; do…while.** Cada una de estas estructuras tienen una sintaxis diferente, cada una con sus particularidades aun cuando tienen muchos puntos en comun. Tanto es asi que si bien cada una tiene su aplicacion particular en general todas pueden adaptarse simulando el comportamiento de las otras. Solo existe la excepcin de la condicional poscondicional que es la unica que al menos se ejecuta una vez. Una primera descripcion de las mismas puede ser:

1) Exactas controlan cantidades
2) No exaxtas evaluan expresiones, se ejecuta siempre que estas sean verdaderas
   a) Precondicional → evalua condicion al niicio y require
      i) Asignacion previa del dato de la expression logica
      ii) Evaluacion de la expresion, si es verdadera se ejecuta, de lo contrario finaliza
      iii) Acciones que correspondan
      iv) Nueva asignacion del dato de la expression logica para ver si continua o termina
   b) Poscondicional
      i) Acciones que correspondan
      ii) Asignacion del dato de la expresion para evaluar si continua o termina

Para la utilizacion de estas estructuras se require controlar:

- Cual es el tamaño del bloque o, dicho de otro modo cuantas veces se debe repetir.
  - Esto se determina con una expresion de control y se ejecuta siempre que esa expression sea verdadera
    - for(;expresion de control;) [0..N]
    - while(expresion de control) [0..N]
    - do{ } while(expresion de control); [1..N]
- Que acciones se repiten
  - Una sentencia, que puede ser:
    - Vacia
    - Simple
    - Estructurada
    - compuesta

Ciclo de repeticion for

for(expresion de inicializacion; expresion de control; expresion de salto)

Esta compuesta por tres sentencias cada una de las cuales finaliza con ; y, desde el punto de vista sintactico pueden no estar.

Ejemplo:

for(i=0; i<5; i++)

la primera vez el iterador I parte de 0, en cada paso se incrementa en 1 i++ y se hace mientras la expression i<5 sea verdadero

for(i=5; i>0; i--)

la primera vez el iterador I parte de 5, en cada paso disminuye su valor en 1 i-- y se hace mientras la expression i>0 sea verdadero

for(i=0; i<10; i+=2)

la primera vez el iterador I parte de 0, en cada paso se incrementa en 2 i+=2 y se hace mientras la expression i<10 sea verdadero

```
i=0;
for(; i<5; )
 i++;
```
la primera vez el iterador I parte de 0 que esta inicializado fuera del cuerpo del for, en cada paso se incrementa en 1 i++, con una accion en el for y se hace mientras la expression i<5 sea verdadero. Esto es un ejemplo de lo dicho que de las tres sentencias no son obligatorias dentro del encabezado del ciclo for

Otros ejmplos
Cuantas veces se repite y por que?

```
for(i=0,j=0; i<10&&j<6; i+=2,j++)

for(i=0,j=0; i<10||j<6; i++,j++)

for(i=0;i<10;i++)
        for(j=0;j<5;j++)
```

Dados los siguientes ejemplos determiner que imprime

La sentencia vacia
```
for(i=0; i<5; i++);
        cout<< "hola";
```
repite 5 veces la sentencia vacia, vea el ; despues del). Al final imprime hola una vez

Una sentencia simple
```
for(i=0; i<5; i++)
        cout<< "hola";
```
repite 5 veces la sentencia de asignacion externa de salida, vea que no hay; ; despues del).

Una sentencia compuesta
```
for(i=0; i<5; i++){
        cout<< "hola";
        cout<< " buenas tardes";
}
```
repite 5 veces la sentencia compuesta por dos asignaciones externas de salida, vea las llaves que componen el bloque

Se busca una sentencia compuesta pero se omiten las llaves
```
for(i=0; i<5; i++)
        cout<< "hola";
        cout<< " buenas tardes";
```

repite 5 veces la sentencia simple y al final una vez buenas tardes, vea que no hay llaves que componen el bloque

cuantas veces se repiten estas sentencias
```
cin>>numero;
for(i=0;i<10&&numero>0;i++){
```

```
        cout<< numero;
        cin>>numero;
};
```

Ciclo de repeticion while

Se ejecuta [0..N] veces, evalua una expresion antes del inicio del ciclo y repite mientras esta sea verdadera, en general necesita una asignacion previa del dato de la expression, una evaluacion de la expression y una nueva asignacion del dato de la expression para determiner si continua o termina.

```
while(expresion) sentencia;
```

se ejecuta la sentencia (vacia, simple, estructuada o compuesta) mientras la expression sea verdadera

```
cin>>numero;
while(numero>0){
        cout<< numero;
        cin>>numero;
};
```

Lo siguiente es correcto? Es equivalente a lo anterior, justifique

```
while(cin>>numero)
        cout<<numero;
```

El resto de los ejemplos son similares a lo visto con sa sentencia for

do while

```
do{
lista de sentencias;
} while (expresion);
```

A diferencia del ciclo while este ciclo se ejecuta al menos una vez ya que la evaluacion de la expression se hace al final.

5

Patrones algorítmicos simples:

Intercambio – Máximos/Mínimos - Seguidillas

PATRONES ALGORITMICOS

Como ya se ha visto “el infinito no es implementable” para los informáticos el infinito es un concepto que desde el punto de vista de la implementación estamos restringidos al menos hasta ahora.

Vimos que entre las características de un algoritmo aparece el concepto de “finitud”, el algoritmo es finito en cantidad de instrucciones y en tiempo de ejecución, por ejemplo. Las estructuras de datos también son finitas, en caso particular de la materia hablamos de struct, array, flujos y estructuras enlazadas. Las sentencias necesarias para la implementación de los programas en el paradigma estructurado también son finitas, particularmente tres, con algunas particularidades, pero son finalmente tres: asignación, análisis de caso y repetición. La implementación de los algoritmos, luego de la interpretación precisa de cual es el problema a resolver, en la mayoría de los casos responde a combinación de patrones algorítmicos particulares que, en general están restringidos a un conjunto no mayor a 25 patrones distintos.

Estos patrones requieren algunas adaptaciones mínimas según cada caso según, por ejemplo el acceso al dato según la estructura a utilizar o el criterio de búsqueda y/o ordenamiento. Salvo estos matices, se puede considerar que responden a lógicas similares y fácilmente adaptables.

Es por esto que abordaremos el desarrollo de algunos patrones algorítmicos simples de utilidad especifica en muchas situaciones problemáticas precisas

Veamos primero la estructura conceptual de un algoritmo

LEXICO {Léxico Global del algoritmo}

{Declaración de tipos, constantes, variables y acciones}

Acción 1

PRE {Precondición de la acción 1}

POS {Poscondición de la acción 1}

LEXICO {Léxico local, propio de la acción 1}

Declaraciones locales

ALGORITMO {Implementación de la acción 1}

{Secuencia de instrucciones de la acción 1}

FIN {Fin implementación algoritmo de la acción 1}

ALGORITMO

PRE {Precondición del algoritmo principal}

POS {Poscondición del algoritmo principal}

{Secuencia de instrucciones del algoritmo principal}

FIN {Fin del algoritmo principal}

Algoritmos que usan asignaciones

Intercambio

En ocasiones se hace necesario intercambiar los valores entre dos identificadores dados dos identificadores con valores particulares desarrollar un modulo que le asigne a cada uno de los identificadores el valor del otro

int a = 10, b = 5, auxiliar;

// como hemos visto la asignación interna es destructiva por lo que si se hace a = b; se perdería el valor inicial de a por lo que no se le podrá asignar a b, es por ello que se requiere de una variable auxiliar para contener el valor antes de perderlo, esto queda:

```
auxiliar = a;     //asignarle 5 a auxiliar para no perder el valor original de a
a = b;            //asignarle a a el valor de b haciendo el primer intercambio
b = auxiliar      // asignarle a b el valor original de a contenido temporalmente en auxiliar
```

Ejemplo de uso: dados tres valores enteros int mayor = 5, medio = 7; menor = 6; desarrollar un programa que le asigne el mayor valor a mayor, el menor a menor y el restante a medio

1. Poner en mayor el mayor entre medio y mayor
   if(medio > mayor) intercambio(medio, mayor)

2. Poner en mayor el mayor entre mayor y menor, aquí esta el mayor de los tres
   if(menor > mayor) intercambio(menor, mayor)

3. Poner en medio el mayor entre menor y medio, aquí quedan los tres ordenados
   if(menor>medio) intercambio(menor, medio)

Algoritmos que utilizan secuencias

Máximos y mínimos

En lo que hace a la información relativa a un conjunto de datos puede ser necesario encontrar cual o cuales de los elementos de ese conjunto cumplen con ciertas características. Por ejemplo si se tiene información de los nombres de los estudiantes y sus calificaciones puede ser posible conocer el nombre del mejor estudiante, según el valor de sus notas, o conociendo el nombre de los atletas y el tiempo empleado en una carrera de velocidad puede requerirse conocer el ganador. En el primer caso se busca al estudiante cuyo promedio es el mayor, en el otro caso se busca el atleta cuyo tiempo es el menor. Existen algoritmos puntuales que permiten buscar máximos y mínimos.

Para ello es necesario poder determinar el tamaño del bloque, de modo de seleccionar la estructura de repetición mas adecuada y la condición de finalización que permita cumplir con precisión lo que se desea evaluar

Buscar un máximo

En forma genérica se puede expresar

Datos e entrada L_1 ... L_n Lista de N elementos

Datos de salida M identificador que contendrá al valor máximo

N pertenece a los números Naturales

N > 0 Por lo menos hay un elemento en la lista

L_i pertenece a los números racionales para todo 1<= i <= N

Poscondición

M pertenece a los números racionales

$m>=l_i$ para todo 1<=i<=n

LÉXICO

Valor : Entero // Identificador que contendrá las lecturas

Máximo : Entero // Identificador que contendrá el máximo del conjunto

Esta acción le asigna el primer valor al máximo que puede ser el primer valor leído, si es que se tiene o un valor arbitrario. El valor arbitrario puede ser un valor razonablemente bajo para que cualquier dato valido lo pueda desplazar o un valor utilizado como valor centinela para indicar que corresponde a la primer lectura.

FIN

Para inicializar el identificador que contendra el maximo se puede hacer con el primer valor leido o con un valor razonablemente alto dentro del contexto delproblema

ALGORITMO

Leer(Valor) ;

Máximo←,valor ;

MIENTRAS haya datos HACER

SI Valor > Máximo

ENTONCES

Máximo ← valor

FIN_SI

FIN_MIENTRAS

SI hubodatos

ENTONCES

Escribir(¨El máximo del conjunto es ¨, máximo)

```
        SI_NO
                Escribir(¨No hubo datos para procesar¨)
        FIN_SI
FIN
```

La búsqueda de un máximo requiere:

1. Determinar el conjunto de datos para poder definir que tipo de composición iterativa es la mas adecuada para el problema planteado
2. Definir al menos dos identificadores del mismo tipo de dato, uno para las sucesivas lecturas y el otro para contener el máximo del conjunto.
3. Se debe inicializar el máximo esto puede ser con un valor arbitrario, razonablemente bajo o un valor particular que pueda ser utilizado como valor centinela o con el valor de la primer lectura.
4. A continuación se compara la nueva entrada con el valor máximo, cada vez que la nueva entrada lo supera, se conserva ese valor como el nuevo máximo.

Ejemplo 1: dado N valores enteros y positivos desarrollar un programa que imprima el máximo del conjunto. Solo hay uno en todos los casos tenemos valor y maximo

<table>
<tr><td colspan="2">Inicializando máximo con un valor arbitrario razonablemente chico</td></tr>
<tr><td>máximo = 0;
for(i=0; i<N;i++){
  cin>>valor;
  if (valor>máximo) máximo = valor;
}</td><td>maximo = 0; i = 0;
while(i<N){
  cin>>valor;
  if (valor>máximo) máximo = valor;
  i++
}</td></tr>
<tr><td colspan="2">Inicializando máximo con la primer lectura</td></tr>
<tr><td>cin>>valor;
máximo = valor;
for(i=1; i<N;i++)
  cin>>valor;
  if (valor>máximo) máximo = valor;

Otra alternativa
<s>máximo = 0;</s>
for(i=0; i<N;i++)
 cin>>valor;
 if (i==0||valor>máximo) máximo = valor;
// en este caso se utiliza al identificador i para definir cual es la primera de las lecturas. i tiene el valor 0 solo una vez, justamente la primera</td><td>Pueden desarrollarse equivalencias entre for y while siguiendo los lineamientos de lo ya expuesto</td></tr>
</table>

Ejemplo de uso de un identificador como bandera

Dado un conjunto de valores float, diferentes de cero determinar el máximo de los negativos y el mínimo de los positivos

```
maximoNegativo = 0.0;
minimoPositivo = 0.0;
cin >> valor;
while(valor!=0){//esta condición hace que el valor cero no pertenezca al conjunto
 if(valor>0)       // si es positivo
    if (minimoPositivo==0.0 || valor<minimoPositivo) //si es la primer lectura o es menor
        minimoPositivo = valor;
     else // si no es positivo es negativo porque el cero esta excluido
          if (maximoNegativo==0.0 || valor>maximoNegativo)
             maximoNegativo = valor;
  cin>>valor;
}
```

Aquí se utiliza el valor inicial de máximo y mínimo como valor centinela, se pregunta por el valor cero para determinar si es la primera de las lecturas. Además la expresión que se evalúa p||q se utiliza esa propiedad que si p es verdadero no importa el valor de verdad de q para determinar el valor de p||q, por lo que la primera vez la proposición q no es evaluada.

La búsqueda de un mínimo utiliza el mismo criterio solamente que requiere :

1. La inicialización del mínimo si se hace con un valor arbitrario este debe ser lo suficientemente alto coma para que cualquier dato del conjunto lo reemplace
2. Las comparaciones posteriores requieren simplemente cambia el operador de relación. Si en el máximo se compra por mayor, en el mínimo debe hacerse por menor.

Estos patrones, en ocasiones, requieren conocer el máximo y el siguiente y/o el máximo y su posición relativa, es decir que posición ocupa ese máximo o mínimo en el lote evaluado. Estas patrones, que requieren poca modificación a lo expuesto lo dejo para que encuentren ustedes la solución.

Seguidillas

La utilización de este algoritmo puntual permite resolver el análisis de una secuencia de datos que cumplen con la precondición de tener una clave que se repite, están ordenados o agrupados por esta clave y se requiere información de cada subconjunto formado por todos los valores de la misma clave y además información sobre la totalidad de los datos. Debe garantizarse que se evaluaran todos los datos y que los que corresponden al mismo subgrupo serán evaluados agrupados.

Dado un conjunto de datos correspondiente a los diferentes clientes (numero de cliente, importe de la compra. Como precondición se establece que los datos de cada cliente están agrupados, y el lote termina con un numero de cliente menor o igual a cero. Se busca informar: cantidad de clientes evaluados y el importe total de cada uno de ellos

Soluicion conceptual

LÉXICO

Importe,Suma : Entero;

NumeroCliente, Anterior : Entero;

AGORITMO

Leer(NumeroCliente)

// hace una lectura anticipada del dato de la expresión lógica//

MIENTRAS (NumeroCliente > 0) HACER

//garantiza la secuencia de lectura de todos los datos//

Suma = 0; //inicializa acumuladores//

Anterior = NumeroCliente; //guarda e valor a controlar//

MIENTRAS (NumeroCliente > 0 y Anterior = NumeroCliente;) HACER

//ciclo que garantiza estar en el mismo subgrupo y que aun haya datos//

Leer(Importe); // lectura del resto de los datos//

Suma = Suma + Importe;

Leer(NumeroCliente)//lectura del nuevo s es igual al anterior

Continua, sino sale del ciclo.

FIN_MIENTRAS

Escribir("El Cliente : ",Anterior, " Compro : ",Suma);

FIN_MIENTRAS

FIN.

Acciones y funciones

6

Introducción

SI L = L 1 + L 2
Entonces
Esfuerzo$_{(L)}$ > Esfuerzo$_{(L1)}$ + Esfuerzo $_{(L2)}$

Modularizacion

Es conveniente, e importante descomponer por varias razones:

- Favorece la comprensión.
- Favorece el trabajo en equipo.
- Favorece el mantenimiento.
- Permite la reusabilidad del código.
- Permite la generalidad de tipos y de criterios
- Permite además separar la lógica de la algoritmia

Datos locales y globales

Unos se declaran en la sección de declaración del programa principal, los otros en la sección de declaración de cada modulo. Otros pueden ser declarados en un bloque determinado, por lo que solo tendrá visibisilidad en el mismo.

- El alcance de un identificador es el bloque del programa donde se lo declara.
- Si un identificador declarado en un bloque es declarado nuevamente en un bloque interno al primero el segundo bloque es excluido del alcance de la primera sección. Algunos lenguajes permiten la incorporación de operadores de ámbito.

Ocultamiento y protección de datos

Todo lo relevante para un modulo debe ocultarse a los otros módulos. De este modo se evita que en el programa principal se declaren datos que solo son relevantes para un modulo en particular y se protege la integridad de los datos.

Parámetros

Son variables cuya característica principal es que se utilizan para transferir información entre módulos.

Hay dos tipos de parámetros, los pasados por valor y los pasados por referencia o dirección.

Los parámetros pasados por referencia o dirección no envían una copia del dato sino envían la dirección de memoria donde el dato se encuentra por lo que tanto el proceso que lo llama como el proceso llamado pueden acceder a dicho dato para modificarlo.

Integridad de los datos

Es necesario conocer que datos utiliza con exclusividad cada modulo para declararlos como locales al modulo ocultándolo de los otros, si los datos pueden ser compartidos por ambos módulos debería conocerse cuales son, si el modulo los utiliza solo de lectura o puede modificarlos y es aquí donde se utilizan los parámetros.

Protección de datos

Si una variable es local a un modulo se asegura que cualquier otro modulo fuera del alcance de la variable no la pueda ver y por lo tanto no la pueda modificar.

Uso de parámetros para retornar valores

Si bien el pasaje por valor es útil para proteger a los datos, existen situaciones en las que se requiere hacer modificaciones sobre los datos y se necesitan conocer esas modificaciones. Para esto se deben utilizar parámetros pasados por referencia.

Utilidad del uso de parámetros

El uso de parámetros independiza a cada modulo del nombre de los identificadores que utilizan los demás. En el caso de lenguajes fuertemente tipiados solo importa la correspondencia en cantidad tipo y orden entre los actuales del llamado y los formales de la implementación con independencia del nombre del identificador.

Reusabilidad

El uso de parámetros permite separar el nombre del dato, del dato en si mismo, lo que permite que el mismo código sea utilizado en distintas partes del programa simplemente cambiando la lista de parámetros actuales o argumentos.

Ejemplo en C++.

```
int suma1(int a, int b){
   return a+b;
}
```

```
C=suma1(10*4,15*2);
```
Los argumentos vinculados con parametros valor pueden ser expresiones.

```
void suma2(int a, int b, int& c){//&c por referencia, por dirección la modificacion dec altera x
   c = a + b;
   return;
}
Suma2(25, 82, x);
```
Argumento vincula parámetro pasado por referencia debe se valorL

```
void Intercambio ( int &a, int &b)
{// comienzo del bloque de declaración de la funcion
        int auxiliar;       //declaración de una variable auxiliar t
        auxiliar = *a;      //asignación a t del valor de a, para contenerlo
        a = b;    //asignación a a del valor de b para intercambiarlo
        b = auxiliar;       //asignación a b del valor que contenía originalmente a.
} // fin del bloque de la funcion
```

Beneficios del uso de acciones y funciones
Una acción o funcion tiene cuatro propiedades esenciales. Ellas son:

1. Generalidad
2. Ocultamiento de información
3. reusabilidad
4. Modularidad

De estas propiedades, se deducen una serie de beneficios muy importantes para el desarrollo de algoritmos.

1. Dominar la complejidad
2. Evitar repetir código
3. Mejorar la legibilidad
4. Facilitar el mantenimiento
5. Favorecer la corrección
6. Favorecer la reutilización

C++. agrega una característica de reutilización de código realmente poderosa que es el concepto de generalización de tipos mediante plantillas y criterios mediante punteros a funciones.
Las definiciones de las plantillas comienzan con la palabra template seguida de una lista de parámetros entre **<** y >, a cada parámetro que representa un tipo se debe anteponer la palabra

La función genérica según el tipo es:

```
//definicion de la plantilla de la función intercambio
template < typename T>
void Intercambio ( T &a, T &b) //con dos parámetros de tipo generico
{// comienzo del bloque de declaración de la funcion
        T t;      //declaración de una variable auxiliar t
        t = a;    //asignación a t del valor de a, para contenerlo
        a = b;    //asignación a a del valor de b para intercambiarlo
        b = t;    //asignación a b del valor que contenía originalmente a.
    return;
} // fin del bloque de la funcion
```

Invocación intercambio
Sea int a=10, b=5;
float c=3.0, d=5.0;
intercambio<int>(a,b);
intercambio<float>(c,d);

Concepto de recursividad:
Es un proceso que se basa en su propia definición. Una función puede invocarse a sí misma como parte de los tratamientos de cálculo que necesita para hacer su tarea
Parte de instancias complejas y las define en términos de instancias más simples del mismo problema, llegando a un punto donde las instancias más simples son definidas explícitamente.
Define el problema en términos de un problema más simple de la misma naturaleza.
Debe disminuir el espacio del problema en cada llamada recursiva
Hay una instancia particular que se conoce como caso base o caso degenerado
Divide el problema original en subproblemas más pequeños. Cuando es lo suficientemente chico se resuelve directamente y se combinan soluciones del subproblema hasta que queda resuelto el problema

Tiene:

- ✓ Una ecuación de recurrencia, en función de términos anteriores $T_n = F(T_{n-1}, T_{n-2}, T_0)$.
- ✓ Uno o varios términos particulares que no dependen de los anteriores. $Ti = G_{(i)}$ (base)

Funcion Factorial

- ✓ Ecuación de recurrencia : $n! = n * (n-1)!$
- ✓ Condiciones particulares: $0! = 1$

Instancias que permanecen en memoria:

Funcion PotenciaNatural

- ✓ Ecuación de recurrencia : $a^n = a^{(n-1)} * a$ si $n > 1$
- ✓ Condiciones particulares: $a^0 = 1$

Funcion DivisionNatural

Dados dos valores num y den, con den <> 0 se puede definir el cálculo del cociente y el resto del siguiente modo:

- ✓ Si num < den → el cociente es = y el resto num.
- ✓ Si num <= den y si c y r son el cociente y resto entre num-den y den → cociente = c+1 y resto r.

Resumen:

En este trabajo se avanza sobre la necesidad de mayor abstracción procedural introduciendo conceptos claves de programación como acciones y funciones como la forma mas adecuada de estructurar problemas en el paradigma procedural. Es una introducción a la abstracción procedural y de datos que servirá de base para sustentar futuros conocimientos de estructuración de programas en clases cuando se aborde, en otra instancia, la programación orientada a objetos.
Se dio valor a términos como reusabilidad, ocultamiento de datos, polimorfismo y cohesión. Se introdujeron conceptos como parámetros actuales, argumentos, parámetros formales, pasaje por valor, pasaje por referencia. Si introdujo un concepto desde el punto de vista de la algoritmia importante como la programación genérica que permite el lenguaje que estamos abordando.

Resumen de funciones

Proposito de las funciones

1. Permite la descomposición como forma de alcanzar la solución
    a. Si L = L1 + L2 → Esf(L) > Esf(L1) + Esf(L2)
2. Promueve la modularidad
3. Favorece
    a. Comprension
    b. Trabajo en equipo
4. Facilita el codigo
    a. Evita repeticiones
5. Perimte
    a. Integridad y protección del dato
    b. Reusabilidad → uso de parametros
    c. Separar la lógica de la algoritmia → funciones de criterio
    d. Separar la lógica del tipo de dato → plantillas

Programacion Modular

1. Propone dividir el problema en modulos → funciones
2. Cada módulo permite aislar con mas precision el problema
3. En C/C++ todo programa ejecuta una función → main()
4. Dentro de la función principal se invocan otras funciones
5. Cada función invocada tiene asignada una tarea especifica.
6. La función, luego de ejecutarse vuelve al punto de invocación

Invocacion/declaración → distintos esquemas

Definicion anticipada	Declaracion definición	Bibliotecas propias

Definición de la funcion
int f(int a, int b){
return a + b;
};
int main()
{ ……..
Invocación
x = f(y,z);
f(x,y)…
w = f (3,4);
….
}
	Declaración de la firma
	int f(int, int);
	
	int main()
	{ ……..
	invocación
	x = f(y,z);;
	…
	w = f (3,4);
	….
	}
	Definicion de la funcion
	int f(int a, int b){
	return a + b;
	};
		Inclusión de la biblioteca
		#include “miBiblio.h”
		
		int main()
		{ ……..
		
		invocación
		x = f(y,z);;
		…
		w = f (3,4);
		….
		}
		La biblioteca separa los prototipos en un archivo .h y el código en un archivo .c

Declaracion de una funcion

Firma o prototipo

Tipo de retorno nombre (lista de tipo de dato del parametro y tipo de parametro)

Int suma(int, int);

1. Tipos de retorno
    a. Escalares
    b. Struct
    c. Punteros
    d. Valor ausente (void)
2. Nombre → regla identificadores Letra(Letra+Digito)*
3. Argumentos
    a. Por valor tipo
    b. Por referencia o dirección tipo&

Definicion de una función

Tipo de retorno nombre (lista parámetros su tipo y tipo de dato){
Int suma(int a, int b)
Cuerpo de la función
}

1. Tipos de retorno
    a. Escalares
    b. Struct
    c. Punteros
    d. Valor ausente (void)
2. Nombre → regla identificadores L(L+D)*
3. Parametros
    e. Por valor tipo identificador
    f. Por referencia o dirección tipo& identificador
2. Cuerpo de la función
    {
    Declaraciones locales;
    Acciones;
    Retorno;
    }

Invocaciones

1. Con valor ausente una invocación a si misma
    a. Nombre(lista de argumentos)
2. Retornando escalar, puntero o struct en una expresión
    a. Identificador = Nombre(lista de argumentos)
3. Los argumentos vinculados con parámetros variables deben ser valorL
4. Los argumentos vinculados con parámetro valor son expresiones, completas o incompletas
5. C++ pasa parámetros por valor o referencia, eso se evidencia en la declaración y definición pero no en la invocacion

Ejemplos

Funcion	Detalle
Firmas o prototipos	
void F1(int)	Firma de función que no retorna valor y recibe un parámetro por valor

void F2(int&)	No retorna valor pero evalua y modifica el argumento vinculado al parámetro
int F3(int)	Similar a F1 retornando un escalar
int F4(int&)	Similar a F2 retornando un escalar
int F5(int, int)	Evaluando dos parametros valor
int F6(int, int&)	Evaluando un parámetro por valor y otro por referencia
int F7(void)	Declaración explicita sin parámetros
Invocacionei	
F1(3*6) F1(a) F1(a+b)	Invocado con una expresión completa vinculada al parámetro valor Imvocado con un identificador (expresión incompleta)
F2 (a)	Invocado con un valor por el ásaje por referencia
A = F3(2*4) A = F3(b)	Un parámetro valor
A = F4(X)	Un parámetro por referencia
A = F5(x, 1+2)	Dos parametros valor
A = F6(14, w)	Un parámetro valor y otro por referencia
A = F7()	Sin parámetros

Definiciones: Ejemplos

```
int suma1 (int a, int b){
return a + b;
}
void suma2 (const int a, const int  b, int &c){
 c = a+b;
return;
}

void intercambio (int &a, int &b){
int c = a;
a = b;
b =  c;
return;
}
```

```
int suma1(int a , int b)
{
return a+b;
}

void suma2( int a, int b, int& c){
 c = a +b;
return;
}

Int main(){
Int h=10, i=15, j, k=20, l=30, m, n;
J=suma1(h,i);
m=suma1(suma1(h,i),3*4);h,i)

suma2(h,i,j);
cout<< j;
```

Pasaje de cadenas como parametros

int F (char* s) o int F (char s[])
Pasaje de vectores y matrices
int F (int v[])
int F (int m[][columnas]

Reusabilidad ejemplos

A = suma1(B, C);
D = suma1(D, E);
F = suma1 (2*3, 4-2);
Se invoca en distintos puntos del programa con conjunto de argumentos diferentes respetando orden yipo y características. Recordando que los que se vinculan con parámetros variables deben ser valor, los que se vinculan con parámetros valor spn expresiones

Generalidad

1. Según el tipo de dato → plantillas
2. Según el criterio de selección → punteros a funciones

Generalidad por tipo de dato	
Sin generalizar	**Generalidad con plantilla**
	template < typename T>

int suma1 (int a, int b){
int c = a+b;
return c;
}
 	
float suma1 (float a, float b){
float c = a+b;
return c;
}
 	
Invocación
X = suma1 (y,z)
	T suma1 (T a, T b){
	return a + b;
	}
	
	Invocación
	X = suma1<int> (y,z)
	X = suma1<float>(y,z)

Estructuras, vectores y matrices

7

Array

Necesidad de su uso. Definiciones y declaraciones

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, en principio, 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.
- Al elemento de posición genérica i le sigue el de posición i+1 (salvo al ultimo) y lo antecedede el de posición i-1 (salvo al primeo).
- El índice debe ser de tipo ordinal. El valor del índice puede verse como el desplazamiento respecto de la posición inicial del arreglo.
- La posición del primer elemento en el caso particular de C es 0(cero), indica como se dijo, el desplazamiento respecto del primer elemento. En un arreglo de N elemento, la posición del ultimo es N – 1, por la misma causa.
- 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 lista o vector.
- El arreglo con 2 o más índices o dimensiones es una tabla o matriz. Un grupo de elementos homogéneo con un orden interno en el que se necesitan 2 o más índices para referenciar a un elemento de la estructura.
- En el caso de C, no hay control interno m para evitar acceder a un índice superior al tamaño físico de la estructura, esta situación si tiene control en C++, mediante la utilización de at para el acceso (se vera mas adelante).

	Caracteristicas de los array
Almacenamiento	Electronico
Procesamiento	Rapido
Tamaño T.E.	Fijo en tiempo de ejecución
Persistencia	Sin persistencia
Proposito	Procesar con rapidez conjunto de datos homogéneos Uso como estructura auxiliar para optimizar proceso
Declaracion	Tipo de dato Nombre[índice][índice]----
Acceso	Nonbre[índice][índice]….
Carga	En la declaración Secuencial

	Ordenada Sin repetición de clave Directa → según cantidad de índices definidos se selecciona vector o matriz
Busquedas	Directa Secuencial Binaria
Ordenamiento	PUP Metodos
Recorridos	Totales Parciales Con Corte de control Apareando

Indexada→ índice acceso a cada miembro s[5]

char s[5] = “abc”

a	b	c		

s[0]

1 indice (1 dim) → vector → int v[5]

V[0]	V[1]	V[2]		

2º mas (+1 dim)→ matriz → int M[3][4]

M[0][0]	M[0][1]	M[0][2]	M[0][3]
M[1][0]	M[1][1]	M[1][2]	M[1][3]
M[2][0]	M[2][1]	M[2][2]	M[2][3]

Acceso directo
Tamaño fijo tiempo de ejecución
int vector[10];

V[0]	V[1]								

Carga
define TOPE 5

int main(){

int v[TOPE]

¿?	¿?	¿?	¿?	¿?

int v[TOPE]={1,2,3,4,5};

1	2	3	4	5

int v[TOPE] = {1,2,3};

1	2	3	0	0

int v[TOPE] = {0}:

0	0	0	0	0

```
for(i=0; i<TOPE; i++)
        v[i]=0;

for(i=0; i<TOPE; i++)
        cin>>v[i];

for(i=0; i<TOPE; i++)
        cout<<v[i];
```

Carga directa
v[j] = expresion; j debe estar en el rango [0..TOPE-1]

int m[2][3]={ {1,4,5 },{4,8,32 } };

1	4	5
4	8	32

```
for (i=0;i<2;i++)
        for(j=0;j<3;j++)
                cin>>m[i][j];
```

STRUCT: Definiciones y declaraciones

```
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
```

Combinación de estructuras: struct con un campo struct

TipoAlumno Vector[4];

legajo	nombre	Fecha		
		d	m	A

Vector
Vector[0]
Vecto[0].legajo
Vector[0].nombre
Vector[0].fecha
Vector[0].fecha.d
Vector[0].fecha.m
Vector[0].fecha.a

Declaración y uso

Genérica
En C:
TipoDeDato Identificador[Tamaño];
int VectorEnteros[10]// declara un vector de 10 enteros
TipoAlumno VectorAlum[10] // declara un vector de 10 registros.
TipoAlumno MatrizAlumno[10][5] //declara matriz o tabla de 10 filas y y columnas

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

TipoAlumno VectorAlum[10];
```

Accesos

Nombre	Tipo dato	
VectorAlum	Vector	Vector de 10 registros de alumnos
VectorAlum[0]	Registro	El registro que esta en la posición 0 del vector
VectorAlum[0].Legajo	Entero	El campo legajo del registro de la posición 0
VectorAlum[0].Fecha	Registro	El campo fecha de ese registro, que es un registro
VectorAlum[0].Fecha.D	Entero	El campo dia del registro anterior

Vector de 5 componentes
int V[5]

V[0]
V[1]
V[2]
V[3]
V[4]

Matriz de 5 filas y tres columnas
int M[5] [3]

M[0][0]	M[0][1]	M[0][2]
M[1][0]	M[1][1]	M[1][2]
M[2][0]	M[2][1]	M[2][2]
M[3][0]	M[3][1]	M[3][2]
M[4][0]	M[4][1]	M[4][2]

Búsqueda

Busqueda secuencial

int tope = 8;
int N=0;

Ve[0]	6
1	3
2	9
3	14
4	1
5	11
6	25
7	96

Buscado	i	v[i]	buscado != v[i]
cuando el buscado esta índice [0..N-1]			
14	0	6	v
	1	3	v
	2	9	v
	3	14	f
cuando no está el buscado → índice toma el valor de N			
19	0	6	v
	1	3	v
	2	9	v
	3	14	v
	4	1	v
	5	11	v
	6	25	v
	7	96	v
	8[N]	???	

```
#define TOPE 8
int busquedaSecuencial(int  buscado, int v[], int N)
{
        int i = 0;
        while(i<N&&v[i]!=buscado)
                i++;
                if(i<N) return i;
        return -1;
};
```

12	35	8	0	0	0	0	0

```
int main()
{
        int Vector[TOPE] = {12, 35, 8};       //TOPE tamaño fisico del vector
        int N = 3;                            // N tamaño lógico
        int aBuscar, esta;
        cin>>aBuscar;
        esta = busquedaSecuencial(aBuscar, Vector, N)//
        if(esta == -1){Vector[N]=aBuscar; N++ };

return 0;
}
```

Interrogantes
Como controlar para no superar el tope?
Que hacer si el buscado no esta?
Como se pasa un vector como argumento?
Como se declara como parámetro? Se protege de modificación? Y en una matriz?

Busquedas
Busqueda directa PUP, clave posicional
saber nombre del equipo Numero 101 → V[101-101].nombre →V[0].nombre
saber nombre del equipo Numero 104 → V[104-101].nombre →V[3].nombre

Pos	Numero	Nombre
0	101	Huracán
1	102	Boca
2	103	River
3	104	Racing
4	105	sl
5	106	Chaca
6	107	Platense
7	108	Indep

V[clave-valor - pos inicial]
Carga

1. **Declaración**
    a. int v[8] = {1,4,2,6,8,9,2,0};→ inicializa todas las posiciones con la lista
    b. int v[8] = {2,6,8};→ inicializa todas las posiciones, tres con la lista el resto con cero.
    c. int v[8] = {0} → inicializa todas las posiciones con cero
    d. equipo equipos[8] = ???? {{101 , “huracán”} , { , } , { , } }

2. **Secuencial → elemento por elemento**

a. **asignación interna** → for(i=0;i<8;i++) V[i] = valor;
b. **asignación externa** → for(i=0;i<8;i++) cin>> V[i];

3. **Directa** → Clave numérica, PUP→ pos V[Clave-valor posición inicial]
4. **ordenada o no sin repetición de clave**

búsqueda

1. **Directa** Clave numérica, PUP→ Ef(1)→ V[Clave – valor 1ra posición]
2. **Secuencial** {} valores sin orden → recorrer desde la primera posición hasta que lo encuentre o hasta que se termine el vector→ Ef(N)

```
int búsquedasecuencial(int v[], int buscado, int n){
vector, valor buscado, tope del vector
i = 0; → índice para recorrer desde el principio
while(i<n&&buscado!= v[i])
        i++; → para acceder a la sgte posicion

if(i<n) return i;
else return -1;//arbitrario indica dato buscado no esta
retorna la posicion donde el buscado esta o -1 si el buscado no esta
}
```

3. **Binaria → deben estar ordenados**→ Ef(logarítmica)

512 256 128 64 32 16 8 4 2 1

2^{10}

Condiciones de los datos para la búsqueda	
Directa PUP	Ef(1)
Binaria	ordenados Ef(log)
Secuencial	datos están sin orden Ef (N)

Busqueda Binaria

0	14
1	25
2	29
3	32
4	45
5	54
6	68
7	92

N 8
Pos 1er 0
pos ultimo 7→N-1

buscado	pos 1ro	pos ultimo	mitad(p+u)/2	V[mitad]
32	0	7	3	32
54	0	7	3	32
	4	7	5	54
25	0	7	3	32
	0	2	1	25
34	0	7	3	32
	4	7	5	54
	4	4	4	45
	4	3	¿??	¿???

cuando el valor de la posición del primero supera el valor de la posicion del ultimo es indicador que el dato no está.

```
int busquedaBinaria(int v[], int buscado, int N, int primero&){
//retorna la posición donde está el dato o menos uno si no lo encuentra
        int ultimo = N-1;
        int medio;
        primero = 0;
        while (primero <= ultimo){
                medio = (primero + ultimo)/2;
                if(v[medio] == buscado) return medio; //la posición donde lo encontró
                if(buscado > v[medio]) primero = medio + 1;
                else ultimo = medio – 1;
        };
        return -1;// si sale del while es que el primero es mayor que ultimo ese es el indicio que el dato
buscado no esta por eso retorna -1
}
```

Desafios

Algunas variaciones sobre la búsqueda binaria:

1. Array de registros ordenado en forma creciente por un campo de tipo entero
2. Idem anterior con campo de tipo float
3. Idem anterior con cadena de caracteres
4. Array de registros ordenado en forma decreciente por un campo de tipo entero
5. Array de registros ordenado por dos campos, encontrar correspondencia de ambos
6. Idem anterior apuntando al primer registro del subconjunto
7. Array de registros ordenado por dos campos, el primero creciente y el segundo posicional en el subgrupo.
8. Es posible resolver con una única funcion los puntos 1,2 y 3? Justifique,
9. Es Es posible resolver con una única funcion los puntos 5, 6, 7? Justifique
10. Es posible resolver con una única funcion los puntos 1 a 7? Justifique

Ejemplos

Caso 5 (4,15)		
C1	C2	C3
4	6	
4	9	
4	15	
7	4	
7	12	
14	5	
14	14	

Caso 6 (7)		
C1	C2	C3
4	6	
4	9	
4	15	
7	4	
7	12	
14	5	
14	14	

Caso 7 (14,2)		
C1	C2	C3
4	1	
4	2	
4	3	
7	1	
7	2	
14	1	
14	2	

Carga

carga ordenada sin repetir la clave

v

0	20
1	25
2	31
3	34
4	54
5	82
6	
7	

pos=busquedaBinaria(V,34,5,primero)
pos=-1; primero=3

Cantidad física → declaración → 8
cantidad lógica → actual → 5

```
N = 0;
while(N<8){
        cin >> valor;
        pos = busquedaBinaria(V, valor, N, primero);
        if(pos == -1) {cargarvalor(V,valor,N,primero); N++};
};
```

0	20	20
1	25	25
2	31	31
3	54	34
4	82	54
5		82
6		
7		

valor 34 N=5 primero=3

desplazamiento
V[5]=V[4]
v[4]=V[3]

i
4 → V[5]=V[4]
3 → v[4]=V[3]

```
void cargavalor(int V[].int valor, int N, int &primero){
for(i = N-1 ; i >= primero ;i--)
        V[i+1]=v[i];
v[primero]= valor;
return;
}
```

Ordenamiento

Ordenamiento → mas simple → menos eficiente → burbuja

0	5									
1	2								2	
2	3							3	3	
3	4				4		4		4	
4	1		5		5		5		5	
Paso	→ i	1		2		3		4		
Comp.	→j	4		3		2		1		
P+C	→N	5		5		5		5		

i	j	v[j-1]	v[j]	variación de j
1	1	v[0]	v[1]	[j-1][j] → índices que compara en cada paso
	2	v[1]	v[2]	
	3	v[2]	v[3]	
	4	v[3]	v[4]	1..N-i →1..5-1 →1..4
2	1	v[0]	v[1]	
	2	v[1]	v[2]	
	3	v[2]	v[3]	1..N-i →1..5-2 →1..3
3	1	v[0]	v[1]	
	2	v[1]	v[2]	1..N-i →1..5-3 →1..2
4	1	v[j]	v[j-1]	1..N-i →1..5-4 →1..1

```
void ordenarVector(int v[], int N) {// v el vector a ordenar N cantidad de componentes
        int i,j, aux;
        for(i = 1; i < N; i++){// pasos → 1..N-1
                for( j = 1; j <= N – i;j++){//comparaciones en cada paso → 1..N-i
                        if(v[j-1] > v[j]{
                                aux = v[j];
                                v[j] = v[j -1];
                                v[j – 1] = aux;
                        }
                } // fin ciclo interno
        }// fin ciclo externo
return;
} // fin de la funcion
```

Ordenar por más de un campo → solo cambia el criterio de selección

Cambiar la lógica Vs cambiar la algoritmia

Cambiar la algoritmia

```
struct tr{
        int c1;
        int c2;
};
prueba v[8];
```

```
void ordenarVector(tr v[], int N) {// v el vector a ordenar N cantidad de componentes
        int i,j, aux;
        for(i = 1; i < N; i++){
                for( j = 1; j <= N – i;j++){
```

```
                        if(v[j-1].c1 > v[j].c1|| (v[j-1].c1== v[j].c1&& v[j-1].c2> v[j].c2)){
                                aux = v[j];
                                v[j] = v[j -1];
                                v[j – 1] = aux;
                        }
                        else{
                            if(v[j-1].c1== v[j].c1)
                                if(v[j-1].c2> v[j].c2){
                                    aux = v[j];
                                    v[j] = v[j -1];
                                    v[j – 1] = aux;
                        }
                } // fin ciclo interno
        }// fin ciclo externo
return;} // fin de la función
```

Cambiando la logica

```
void ordenarVector(tr v[], int N) {// v el vector a ordenar N cantidad de componentes
        int i,j, aux;
        for(i = 1; i < N; i++){
                for( j = 1; j <= N – i;j++){
                        if(v[j-1].c1 > v[j].c1|| (v[j-1].c1== v[j].c1&& v[j-1].c2> v[j].c2)){
                                aux = v[j];
                                v[j] = v[j -1];
                                v[j – 1] = aux;
                        }
                } // fin ciclo interno
        }// fin ciclo externo
return;
} // fin de la función
```

Consultas

- Como se resuelve si cambia el criterio de ordenamiento o el tipo de dato
    - Ambos numéricos creciente
    - Uno numérico y el otro una cadena
    - Ambos decrecientes
    - Uno creciente y otro decreciente
- Es posible una misma algoritmia para todos los casos?
- Y si los campos de ordenamiento fueran multiplas?

Recorrido

Recorridos

1. Completo
   fo(i = 0; i<N, i++)
   cout << v[i];

2. Con corte de control
   Precondición → Al menos un campo que se repite, agrupado por ese campo, que conforman un subconjunto
   Pos condición → Información de cada subconjunto y/o del conjunto general
3. Apareo
   Precondición → Al menos dos estructuras con al menos un campo en común y ordenados.

Pos condición → manejo conjunto intercalando y manteniendo el orden

Preguntas

Si se dispone de un vector y la clave de identificación es posicional:

1. Cual es la búsqueda adecuada?
2. Cual es la eficiencia de la búsqueda?
3. Como se accede a la posición de búsqueda?

Si la clave fuera numérica y los datos están ordenados cuales serian sus respuestas?

Si la clave fuera una cadena de caracteres?

En caso de estar sin orden, como respondería? Existe diferencia si la clave es numérica o cadena?- Justifique-

En caso de estar desordenado, puede plantear alternativas a la búsqueda;

Que estrategias conoce para buscar u ordenar par mas de un campo?

Apareo (conceptual)

Concepto: Aplicable a dos o más estructuras son al menos un campo en común ordenadas por ese campo que se procesan paralelamente, intercalando los valores para conservar el orden

Seudocódigo:

```
Situarse al principio de ambas estructuras
Hacer mientras (haya datos en ambas)
 Si la primera cumple criterio de ordenamiento
        Procesarla;
        Avanzar con ella
  Sino
        Procesar la otra;
        Avanzar con ella
Fin del mientras
Agotar la estructura que no se termino
    Procesarla;
    Avanzar
 Fin del proceso
```

Vectores

```
int i = 0;
int j = 0;

while((i<N) && (j<M)){
            if(v1[i].c1<v2[j].c1){
                //procesar v1[i];
                i++;}
            else{
                //procesar v2[j];
                j++;}
        }

while(i<N){
  //procesar v1[i];
        i++;      }

while(j<M){
  //procesar v2[j];
        j++;}
```

Indice para recorrer V1
Indice para recorrer V2

Mientras hay datos en ambos vectores
Si el de v1 es menor
Lo procesa
Avanza
Sino
Procesa el otro
Avanza

En caso de que el primero no termin
Lo procesa
Avanza

Si el que no termino es el otro

```
int j = 0; int i = 0;
// j==M || i<N && v1[i].c < v2[j].c
while((i<N) || (j<M)){
            if(j==M || i<N && v1[i].c < v2[j].c){
                //procesar v1[i];
                i++;}
            else{
                //procesar v2[j];
                j++;}
  }
```

1
8
14
22
25

2
3
11

Con plantilla y función criterio

Implementación de la función

```
template <typename T>
void Apareo(T v1[], T v2[], int N, int M, int (*criterio)(T,T)){
   int i = 0;
   int j = 0;
   while((i<N) && (j<M)){
```

programa principal e Invocación

```
struct tr{ int c1; intc2};
int criterioCampo1Asc(tr a,tr b){
  if(a.c1<b.c1) return 1;
  return 0;
}

int criterioCampo1Des(tr a,tr b){
```

```
      if(criterio(v1[i],v2[j])==1){
         //procesar v1<T>[i];
         cout << v1[i].c1 << endl;
         i++;} else{
         //procesar v2<T>[j];
         cout << v2[j].c1 << endl;
         j++;}   };
while(i<N){
      //procesar v1<T>[i];
      cout << v1[i].c1 << endl;
      i++; }
while(j<M){
     //procesar v2<T>[j];
     cout << v2[j].c1 << endl;
     j++;  } }
```

```
  if(a.c1>b.c1) return 1;
  return 0;
}
int criterioCampo1yCampo2Des(tr a,tr b){
if(a.c1>b.c1||( a.c1==b.c1&& a.c2>b.c2) return 1;
  return 0;
}
int main (){
……
Apareo<tr>(v1,v2,N,M,criterioCampo1Asc)

Apareo<tr>(v1,v2,N,M,criterioCampo1Desc)

Apareo<tr>(v1,v2N,M,,criterioCampo1yCampo2Desc)
```

Otros patrones Apareo

Apareo por criterios diferentes

- Por más de un campo de ordenamiento
  - conservar algoritmia modificando la lógica → agrupar criterios →&&
- Modificar criterio
  - Cambiando algoritmia
  - Utilización de funciones de criterio

Apareo de N estructuras

- Ir apareando por pares y luego aparear loe resultados
- Utilizar una estructura auxiliar →V1

A1
4
6
12
20

A2
3
7
19
25

A3
1
2
5
8

V1

Clave	Control
20	0
25	0
8	0

```
fread(&r1,sizeof(r1), 1, f1)
V1[0].clave = r1.clave
fread(&r2,sizeof(r2), 1, f2)
V1[1].clave = r2.clave
fread(&r3,sizeof(r3), 1, f3)
V1[2].clave = r3.clave
archivosActivos = 3
while(archivosActivos>0){
  p = buscarMinimoNoCero(V)
switch(p)
{
   case 0: //procesar archivo1, si es feof poner 0 en el control y disminuir archivosActivos
   break;
   case 1: // procesar archivo2, si es feof poner 0 en el control y disminuir archivosActivos;
   break;
   case 2: // procesar archivo3, si es feof poner 0 en el control y disminuir archivosActivos
   break;
}
```

```
int buscarMinimoNoCero( tr V[], int N){
int i, minimo;
for( i = 0;V[i].control!=0; i++);
minimo = V[i].clave; i++;
for( ; i<N ; i++){
 if(V[i].clave < minimo) minimo = V[i].clave;
……..}
return posición del minimo
```

Corte de Control (conceptual)

Concepto: Aplicable a una estructura con al menos un campo que se repite, agrupados por ese campo.
Propósito: Mostrar los datos en forma ordenada sin repetir ese campo común.

Como se requieren los datos

Materia	Legajo	Nota
AyED	145234	5
AyED	234169	3
AyED	135321	8
SSL	242132	9
SSL	125328	7

Ejemplo de muestra de resultados

Materia AyED
Orden Legajo Nota
1 145234 5
2 234169 3
3 135321 8
Cantidad Alumnos 3
Materia SSL
Orden Legajo Nota
1 242132 9
2 125328 7
Cantidad Alumnos 2
Total Alumnos 5

Seudocódigo:

Situarse al principio de la estructura
Inicializar contadores generales
Mostrar títulos generales
Hacer mientras (haya datos)
  Inicializar contadores de cada subgrupo
  Mostrar títulos subgrupos
  Identificar grupo a analizar→subgrupo

  Hacer mientras (haya datos Y mismo subgrupo)
    Procesar el registro
    Avanzar al siguiente
  Fin ciclo interno
 Informar datos de cada subgrupo
Fin ciclo externo
Informar sobre generalidades

Observar

El ciclo interno tiene la conjunción de la condición del ciclo externo y la propia.
Las lecturas deben hacerse al final del ciclo interno menor.
Este Patrón algorítmico NO ORDENA, solo muestra agrupados los datos que ya están ordenados evitando repeticiones innecesarias.

Vectores

```
int i = 0;
totalAlumnos = 0;
<TITULOS GENERALES>
while(i<N){
  alumnosCurso = 0
  Control = V[i].curso
  <TITULOS DEL SUB GRUPO>
  while(i>N&&V[i].curso==control){
    alumnosCurso++
    totalAlumnos ++
    Mostrar Datos del V[i]
    i++ }// fin ciclo interno
    Mostrar Datos del Subgrupo
} // fin ciclo externo
Resultados Finales
```

Inicializar indice de recorrido
Inicializar variables generales
Colocar titulos generales
Recorrer mientra haya datos
Inicializaciones del subgrupo
Tomar la clave de control
Titulus de los subgrupos
Hay datos y son del mismo subgrpo
Modificación variables subgrupo
Modificación variables generals
Mostrar ese elemento si corresp.
Avanzar al siguiente
Si corresponde mostrar

Si corresponde mostrar

Plantillas para vectores

agregar → Agrega un nodo al final del vector, el control de no superar el valor físico debe hacerse antes de invocar a la función, actualiza el tamaño lógico

```
void agregar(int arr[], int& n,
int x){
   arr[n]=x;
   n++;

   return;
}


agregar(V, 10, 123)
```

```
template <typename T> void
agregar(T arr[], int& n, T v){
   arr[n]=v;
   n++;

   return;
}


agregar<tipoReg>(V, 10, reg)
agregar<int> (V, 10, 123)
```

mostrar → muestra el contenido de un vector, se agrega el concepto de puntero a función.

```
void mostrar(int v[], int n){

   for(int i=0; i<n; i++){
      cout <<(v[i]<<endln;
   }

   return;
}
```

```
template <typename T> void
mostrar(T v[], int n),void
(*ver(T))){
   for(int i=0; i<n; i++){
      ver(v[i];
   }

   return;
}

void ver_reg(TipoReg r){
 cout << r.c1…….<<endl;
 return;
}
mostrar<tipoReg>(v, 10,ver_reg);
```

ordenar → ordena los elementos de un vector, creciente en enteros, según el criterio de ordenamiento

```
void ordenar(int v[], int n,){

   int aux;
   int i,j,ord=0;
   for(i=0,i<n-1&&ord==0,i++){
      ord=1;
      for(int i=0; i<n-1; i++){

      if(v[j]>v[j+1]){
            aux = v[j];
            v[j] = v[j+1];
            v[j+1] = aux;
            ord = 1;
         }
      }
   }
   return;
}



Ordenar(v,n);
```

```
template <typename T> void
ordenar(T v[], int n, int
(*criterio)(T,T)){
   T aux;
   int i,j,ord=0;
   for(i=0,i<n-1&&ord==0,i++){
      ord=1;
      for(int i=0; i<n-1; i++){

if(criterio(v[j],v[i+1])>0){
            aux = v[j];
            v[j] = v[j+1];
            v[j+1] = aux;
            ord = 1;
         }
      }
   }
   return;
}
int enteroCrec(int e1,inte2){
    return e1>e2?1:0;
 }
ordenar<int>(v, n, enterocrec);
```

buscar→ busca secuencialmente un elemento en un vector retornando el índice donde lo encuentra, si el dato buscado esta o el valor -1 silo buscado no esta en el vector

```
int buscar(int v[], int n, int x)
{
   int i=0;
   while( i<n && (v[i]!= x)){
      i++;
   }
   return i<n?i:-1;
}



buscar(v,n, 123);
```

```
template <typename T, typename K>
int buscar(T v[], int n, K x, int
(*criterio)(T,K)){
   int i=0;
   while( i<n &&
criterio(v[i],x)>0 ){
      i++;
   }
   return i<n?i:-1;
}

int Legajo(tr r, int leg){
return r.leg!=leg?1:0;

buscar<tr,int>(v,n, 123, legajo);
```

busquedaBinaria→ busca dicotomicamente un elemento en un vector retornando el índice donde lo encuentra, si el dato buscado esta o el valor -1 silo buscado no esta en el vector

```
int busquedaBinaria(int a[], int
n, int v){
   int i=0;
   int j=n-1;
   int k;

  while( i<=j ){
      k=(i+j)/2;
      if( v > a[k] ){
         i=k+1;
      }
      else
      {
         if(v<a[k]){
            j=k-1;
         }
         else{
            return k;
         }
      }
   }
   return -1;
}
```

```
template<typename T, typename K>
int busquedaBinaria(T v[], int n,
K v, int (*criterio)(T, K)){
   int i=0;
   int j=n-1;
   int k;

  while( i<=j ){
      k=(i+j)/2;
      if( criterio(v,a[k])>0 ){
         i=k+1;
      }
      else
      {
         if(criterio(v,a[k])<0){
            j=k-1;
         }
         else{
            return k;
         }
      }
   }
   return -1;
}
```

Implementaciones C C++

Agregar un elemento al final de un array

```
void agregar(int arr[], int& len, int v)
{
      arr[len]=v;
      len++;
      return;
}
```

Recorrer y mostrar el contenido de un array

```
void mostrar(int arr[], int len)
{
      for(int i=0; i<len; i++){
        cout << arr[i] << endl;
      }
  return;
}
```

Buscar en un vector

```
int buscar(int arr[], int len, int v)
{
      int i=0;
      while( i<len && arr[i]!=v ){
       i++;
      }
  return i<len?i:-1; }
```

Eliminar el valor que se ubica en una determinada posición del array

```
void eliminar(int arr[], int& len, int pos)
{
      for(int i=pos; i<len-1; i++ ){
       arr[i]=arr[i+1];
      }

      len--;
  return;
}
```

Insertar un valor en una determinada posición del array

```
void insertar(int arr[], int& len, int v, int pos)
{
      for(int i=len-1; i>=pos; i--){
        arr[i+1]=arr[i];
      }
      // insertar el elemento e incremento la longitud del array
      arr[pos]=v;
      len++;
  return;
}
```

Insertar un valor respetando el orden del array

```
int insertarOrdenado(int arr[], int& len, int v)
{
      int i=0;
      while( i<len && arr[i]<=v ){
       i++;
      }
      // insertar el elemento en la i-esima posicion del array
      insertar(arr,len,v,i);
return i;}
```

Insetar un valor respetando el orden del array, sólo si aún no lo contiene

La siguiente función busca el valor `v` en el array `arr`; si lo encuentra entonces asigna `true` a `enc` y retorna la posición que `v` ocupa dentro de `arr`. De lo contrario asigna `false` a `enc`, inserta a `v` en `arr` respetando el orden de los números enteros y retorna la posición en la que finalmente `v` quedó ubicado.

```
int buscaEInserta(int arr[], int& len, int v, bool& enc)
{
      // busco el valor
      int pos = buscar(arr,len,v);
      // determino si lo encontre o no
      enc = pos>=0;
      // si no lo encontre entonces lo inserto ordenado
      if( !enc ){
        pos = insertarOrdenado(arr,len,v);
      }
      // retorno la posicion en donde se encontro el elemento o en
donde se inserto
return pos;}
```

Analisis comparativo de estructuras

Los vectores permiten manejar conjunto de datos del mismo tipo. Otra estructura, el archivo, también permite manejar conjunto de datos del mismo tipo, asi como también las estructuras enlazadas con asignación dinámica en memoria. Las estructuras tienen sus particularidades y, para la resolución de los problemas, debemos decidir cual de ellas es la más adecuada a los efector de resolver los problemas planteados. Por ejemplo, si el dato debe persistir más alla de la aplicación no hay duda que el almacenamiento debe ser físico, en un archivo. Pero si se trata de buscar, reordenar o priorizar la velocidad de procesamiento, allí la elección debe darse hacia estructuras de almacenamiento electrónico, en este caso los vectores o matrices. El cuadro que sigue busca caracterizar estas estructuras para poder tomar la mejor decisión en el momento de la selección.

Caracteristica	Archivo	Vector
Almacenamiento	Fisico	Electronico
Procesamirnto	Lento	Rapido
Persistencia al fin de la aplicacion	SI	NO
Tamaño en tiempo ejecucion	Variable	Fijo
Busqueda	Directa Binaria Secuencial (Ineficiente)	Directa Binaria Secuencial
Carga	Directa Agregando al final	Directa Agregando al final Agregando en orden
Ordenamiento	Con Pos. Unica Predecible	Con Pos. Unica Predecible Moetodos de ordenamiento
Recorridos	Completo Con corte de control Apareando	Completo Con corte de control Apareando
Carga sin repetir la clave	NO (salvo el caso de PUP)	SI
Busqueda de los N Mejores	NO	SI
Utilizacion como est. auxiliar	NO (salvo el caso de PUP)	Es su uso mas frecuente

Desde el punto de vista de la algoritmia se puede sostener si es necesario buscar o modificar el orden entre los datos de entrada y salida lo mas adecuado es tratar con estructuras auxiliares con almacenamiento electrónico. Esto es priorizar ARRAY sodre ARCHIVO. Desde la eficiencia priorizar para la carga o la busquda DIRECTA, BINARIA, SECUENCIAL (esta ultima solo en array).

Ejercicios con vectores y matrices

1. Ingresar un valor N (< 25). Generar un arreglo de N componentes en el cual las mismas contengan los primeros números naturales pares e imprimirlo.
2. Ingresar un valor entero N (< 30) y a continuación un conjunto de N elementos. Si el último elemento del conjunto tiene un valor menor que 10 imprimir los negativos y en caso contrario los demás.
3. Ingresar un valor entero N (< 20). A continuación ingresar un conjunto VEC de N componentes. A partir de este conjunto generar otro FACT en el que cada elemento sea el factorial del elemento homólogo de VEC. Finalmente imprimir ambos vectores a razón de un valor de cada uno por renglón
4. Ingresar un valor entero N (< 25). A continuación ingresar un conjunto VEC de N componentes. Si la suma de las componentes resulta mayor que cero imprimir las de índice impar, sino los otros elementos.
5. Ingresar un valor entero N (< 30). A continuación ingresar un conjunto UNO y luego otro conjunto DOS, ambos de N componentes. Generar e imprimir otro conjunto TRES intercalando los valores de posición impar de DOS y los valores de posición par de UNO.
6. Ingresar un valor entero N (< 40). A continuación ingresar un conjunto VALOR de N elementos. Determinar e imprimir el valor máximo y la posición del mismo dentro del conjunto. Si el máximo no es único, imprimir todas las posiciones en que se encuentra.
7. Ingresar un valor entero N (< 15). A continuación ingresar un conjunto DATO de N elementos. Generar otro conjunto de dos componentes MEJORDATO donde el primer elemento sea el mayor valor de DATO y el segundo el siguiente mayor (puede ser el mismo si está repetido). Imprimir el conjunto MEJORDATO con identificación.
8. Ingresar un valor entero N (< 25). A continuación ingresar un conjunto GG de N elementos. Imprimir el arreglo en orden inverso generando tres estrategias para imprimir los elementos a razón de: a) Uno por línea, b) Diez por línea, c) Cinco por línea con identificación
9. Ingresar un valor entero N (< 40). A continuación ingresar un conjunto A y luego otro conjunto B ambos de N elementos. Generar un arreglo C donde cada elemento se forme de la siguiente forma: C[1] = A[1]+B[N] C[2] = A[2]+B[N-1]
10. Ingresar dos valores enteros M (< 10) y N (< 15). A continuación ingresar un conjunto A de M elementos y luego otro B de N elementos. Generar e imprimir:
    a) Un conjunto C resultante de la anexión de A y B.
    b) Un conjunto D resultante de la anexión de los elementos distintos de cero de A y B.
11. Ingresar un valor N (< 25). Generar un arreglo de N componentes en el cual las mismas contengan los primeros números naturales pares e imprimirlo.
12. Ingresar un valor entero N (< 30) y a continuación un conjunto de N elementos. Si el último elemento del conjunto tiene un valor menor que 10 imprimir los negativos y en caso contrario los demás.
13. Ingresar un valor entero N (< 20). A continuación ingresar un conjunto VEC de N componentes. A partir de este conjunto generar otro FACT en el que cada elemento sea el factorial del elemento homólogo de VEC. Finalmente imprimir ambos vectores a razón de un valor de cada uno por renglón
14. Ingresar un valor entero N (< 25). A continuación ingresar un conjunto VEC de N componentes. Si la suma de las componentes resulta mayor que cero imprimir las de índice impar, sino los otros elementos.
15. Ingresar un valor entero N (< 30). A continuación ingresar un conjunto UNO y luego otro conjunto DOS, ambos de N componentes. Generar e imprimir otro conjunto TRES intercalando los valores de posición impar de DOS y los valores de posición par de UNO.
16. Ingresar un valor entero N (< 40). A continuación ingresar un conjunto VALOR de N elementos. Determinar e imprimir el valor máximo y la posición del mismo dentro del conjunto. Si el máximo no es único, imprimir todas las posiciones en que se encuentra.

17. Ingresar un valor entero N (< 15). A continuación ingresar un conjunto DATO de N elementos. Generar otro conjunto de dos componentes MEJORDATO donde el primer elemento sea el mayor valor de DATO y el segundo el siguiente mayor (puede ser el mismo si está repetido). Imprimir el conjunto MEJORDATO con identificación.
18. Ingresar un valor entero N (< 25). A continuación ingresar un conjunto GG de N elementos. Imprimir el arreglo en orden inverso generando tres estrategias para imprimir los elementos a razón de: a) Uno por línea, b) Diez por línea, c) Cinco por línea con identificación
19. Ingresar un valor entero N (< 40). A continuación ingresar un conjunto A y luego otro conjunto B ambos de N elementos. Generar un arreglo C donde cada elemento se forme de la siguiente forma: C[1] = A[1]+B[N] C[2] = A[2]+B[N-1]
20. Ingresar dos valores enteros M (< 10) y N (< 15). A continuación ingresar un conjunto A de M elementos y luego otro B de N elementos. Generar e imprimir:
    a) Un conjunto C resultante de la anexión de A y B.
    b) Un conjunto D resultante de la anexión de los elementos distintos de cero de A y B.
21. Ingresar dos valores enteros M (< 25) y N (< 10) A continuación ingresar un conjunto A de M elementos y luego otro B de N elementos, ambos ordenados en forma creciente por magnitud. Generar e imprimir el conjunto TOTAL resultante del apareo por magnitud de los conjuntos A y B.
22. Ingresar un valor entero N (< 40). Luego ingresar un conjunto REFER de N elementos reales (ingresan ordenados por magnitud creciente). Finalmente ingresar un valor pesquisa X. Desarrollar el programa que determine e imprima:
    a) Con cual elemento (posición) del conjunto coincide, o
    b) Entre cuales dos elementos (posiciones) se encuentra, o
    c) Si es menor que el primero o mayor que el último.
23. Ingresar un valor entero CANT (< 50) y a continuación un conjunto SINOR de CANT elementos. Desarrollar un programa que determine e imprima:
    a) El conjunto SINOR en el que cada elemento original se intercambie por su simétrico: A[1] con A[CANT], A[2] con A[N-1], ... etc.
    b) El conjunto SINOR ordenado de menor a mayor sobre si mismo indicando la posición que ocupaba cada elemento en el conjunto original.

24. Una empresa que distribuye mercadería hacia distintas localidades del interior dispone de dos archivos de registros: Uno denominado DESTINOS con información de la distancia a cada uno de los destinos: a) Nro. de destino (3 dígitos) b) Distancia en kilómetros (NNN.NNN). Otro denominado VIAJES con los viajes realizados por cada camión (< 200), donde cada registro contiene: a) Patente del camión (6 caracteres) b) Nro. de destino c) Nro. de chofer (1 a 150). Desarrollar estrategia, algoritmo y codificación del programa que determine e imprima:
    1) Cantidad de viajes realizados a cada destino (solo si > 0).
    2) Nro. de chofer con menor cantidad de Km (entre los que viajaron).
    3) Patente de los camiones que viajaron al destino 116 sin repeticiones de las mismas.

25. Ejercicio Nro. 54:

26. Ingresar dos valores, M (< 30) y N (< 25) y a continuación por filas todos los componentes de una matriz MATRIZA de M filas y N columnas. Desarrollar un programa que:
    a) Imprima la matriz MATRIZA por columnas.
    b) Calcule e imprima el valor promedio de los componentes de la matriz.
    c) Genere e imprima un vector VECSUMCOL donde cada componente sea la suma de la columna homóloga.
    d) Genere e imprima un vector VECMAXFIL donde cada componente sea el valor máximo de cada fila.

27. Ejercicio Nro. 55:

28. Ingresar un valor N (< 25) y luego por filas una matriz cuadrada CUADRA de N filas y columnas. Desarrollar un programa que determine e imprima:
    a) Todos los elementos de la diagonal principal o secundaria según de cual resulte mayor la sumatoria de elementos.
    b) Los elementos del cuarto (N/2 filas y N/2 columnas) cuya sumatoria resulte mayor (considerando que N fuera par).
    c) Los elementos de la triangular superior o inferior dependiendo de cual tenga mayor sumatoria de elementos.
29. Ingresar dos valores, M (< 20) y N (< 25) y a continuación por columnas todos los componentes de una matriz DESORDE de M filas y N columnas. Desarrollar un programa que:
    a) Ordene (creciente) cada columna de la matriz sobre si misma y la imprima a razón de una columna por renglón.
    b) Ordene (creciente) la matriz sobre si misma por fila desde el elemento 1,1 al M,N y la imprima a razón de una fila por renglón.
30. Ingresar por plano, fila y columna todos los elementos de una matriz MATRIDIM de M planos, filas y columnas. Desarrollar un programa que:
    a) Imprima la matriz MATRIDIM por columnas, fila, plano.
    b) Calcule e imprima el valor promedio de la matriz.
    c) Determine e imprima el mayor valor y en que lugar del la matriz se encuentra.
    d) Genere e imprima una matriz MATCSUMCOL donde cada elemento sea la suma de la columna homóloga.
    e) Genere e imprima una matriz MATMAXFIL donde cada elemento sea el valor máximo de cada fila.

Templates

Los templates permiten parametrizar los tipos de datos con los que trabajan las funciones, generando de este modo verdaderas funciones genéricas.

Generalización de las funciones agregar y mostrar

```
template <typename T> void agregar(T arr[], int& len, T v)
{
      arr[len]=v;
      len++;
      return;
}
```

```
template <typename T> void mostrar(T arr[], int len)
{
      for(int i=0; i<len; i++){
        cout << arr[i];
        cout << endl;
      }
  return;
}
```

Veamos como invocar a estas funciones genéricas.

```
int main()
{
      string aStr[10];
      int lens =0;
      agregar<string>(aStr,lens,"uno");
      agregar<string>(aStr,lens,"dos");
      agregar<string>(aStr,lens,"tres");
      mostrar<string>(aStr,lens);
      int aInt[10];
      int leni =0;
      agregar<int>(aInt,leni,1);
      agregar<int>(aInt,leni,2);
      agregar<int>(aInt,leni,3);
      mostrar<int>(aInt,leni);
  return 0;
}
```

Ordenamiento

La siguiente función ordena el array `arr` de tipo `T` siempre y cuando dicho tipo especifique el criterio de precedencia de sus elementos mediante los operadores relacionales `>` y `<`. Algunos tipos (y/o clases) válidos son: `int`, `long`, `short`, `float`, `double`, `char` y `string`.

```
template <typename T> void ordenar(T arr[], int len)
{
      bool ordenado=false;
      while(!ordenado){
        ordenado = true;
        for(int i=0; i<len-1; i++){
            if( arr[i]>arr[i+1] ){
              T aux = arr[i];
              arr[i] = arr[i+1];
              arr[i+1] = aux;
              ordenado = false;
            }
        }
      }
return;
}
```

Punteros a funciones

Las funciones pueden ser pasadas cómo parámetros a otras funciones para que éstas las invoquen. Se pueden crear apuntadores a funciones, en lugar de direccionar datos los punteros a funciones apuntan a código ejecutable. Un puntero a una función es un apuntador cuyo valor es el nombre de la función.

Sintaxis
TipoRetorno (*PunteroFuncion)(ListaParametros)

```
 int f(int);        //declara la funcion f
int (*pf)(int);   // define pf a funcion int con argumento int
pf = f;        // asigna la direccon de f a pf
```

Los apuntadores permiten pasar una función como argumento a otra función. Utilizaremos esta carácteristica de los lenguajes de programación para parametrizar el criterio de precedencia que queremos que la función ordenar aplique al momento de comparar cada par de elementos del array arr.
Observemos con atención el tercer parámetro que recibe la función ordenar desarrollada mas abajo. Corresponde a una función que retorna un valor de tipo int y recibe dos parámetros de tipo T, siendo T un tipo de datos genérico parametrizado por el template.
La función criterio, que debemos desarrollar por separado, debe comparar dos elementos e1 y e2, ambos de tipo T, y retornar un valor: negativo, positivo o cero según se sea: e1<e2, e1>e2 o e1=e2 respectivamente.
Las funciones pueden ser pasadas cómo parámetros a otras funciones para que éstas las invoquen. Utilizaremos esta carácteristica de los lenguajes de programación para parametrizar el criterio de precedencia que queremos que la función `ordenar` aplique al momento de comparar cada par de elementos del array `arr`.

```
template <typename T>void ordenar(T arr[], int len, int
(*criterio)(T,T))
{
      bool ordenado=false;
      while(!ordenado){
        ordenado=true;
        for(int i=0; i<len-1; i++){
        // invocamos a la funcion para determinar si corresponde o no permutar
            if( criterio(arr[i],arr[i+1])>0 ){
              T aux = arr[i];
              arr[i] = arr[i+1];
              arr[i+1] = aux;
              ordenado = false;
            } // cierra bloque if
        } // cierra bloque for
      }// cierra bloque while
return; }
```

Ordenar arrays de diferentes tipos de datos con diferentes criterios de ordenamiento

A continuación analizaremos algunas funciones que comparan pares de valores (ambos del mismo tipo) y determinan cual de esos valores debe preceder al otro.

Comparar cadenas, criterio alfabético ascendente:

```
int criterioAZ(string e1, string e2)
{
      return e1>e2?1:e1<e2?-1:0;
}
```

//Comparar cadenas, criterio alfabético descendente:

```
int criterioZA(string e1, string e2)
{
      return e2>e1?1:e2<e1?-1:0;
}
```

//Comparar enteros, criterio numérico ascendente:

```
      int criterio09(int e1, int e2)
{
      return e1-e2;
}
```

//Comparar enteros, criterio numérico descendente:

```
      int criterio90(int e1, int e2)
{
      return e2-e1;
}
```

Probamos lo anterior:

```
int main()
{
	int len = 4;
	// un array con 6 cadenas
	string x[] = {"Pablo", "Pedro", "Andres", "Juan"};
	// ordeno ascendentemente pasando como parametro la funcion criterioAZ
	ordenar<string>(x,len,criterioAZ);
	mostrar<string>(x,len);
	// ordeno descendentemente pasando como parametro la funcion criterioZA
	ordenar<string>(x,len,criterioZA);
	mostrar<string>(x,len);
	// un array con 6 enteros
	int y[] = {4, 1, 7, 2};
	// ordeno ascendentemente pasando como parametro la funcion criterio09
	ordenar<int>(y,len,criterio09);
	mostrar<int>(y,len);
	// ordeno ascendentemente pasando como parametro la funcion criterio90
	ordenar<int>(y,len,criterio90);
	mostrar<int>(y,len);
  return 0;
}
```

Arrays de estructuras

Trabajaremos con la siguiente estructura:

```
struct Alumno
{
	int legajo;
	string nombre;
	int nota;
};
// esta funcion nos permitira "crear alumnos" facilmente
Alumno crearAlumno(int le, string nom, int nota)
{
	Alumno a;
	a.legajo = le;
	a.nombre = nom;
	a.nota = nota;
  return a; }
```

Mostrar arrays de estructuras

La función `mostrar` que analizamos más arriba no puede operar con arrays de estructuras porque el objeto `cout` no sabe cómo mostrar elementos cuyos tipos de datos fueron definidos por el programador. Entonces recibiremos cómo parámetro una función que será la encargada de mostrar dichos elementos por consola.

```
template <typename T>
void mostrar(T arr[], int len, void (*mostrarFila)(T))
{
      for(int i=0; i<len; i++){
        mostrarFila(arr[i]);
      }
return;
}
```

Probemos la función anterior:

```
// desarrollamos una funcion que muestre por consola los valores de una estructura
void mostrarAlumno(Alumno a)
{
      cout << a.legajo << ", " << a.nombre << ", "<< a.nota << endl;
}

int main()
{
      Alumno arr[6];
      arr[0] = crearAlumno(30,"Juan",5);
      arr[1] = crearAlumno(10,"Pedro",8);
      arr[2] = crearAlumno(20,"Carlos",7);
      arr[3] = crearAlumno(60,"Pedro",10);
      arr[4] = crearAlumno(40,"Alberto",2);
      arr[5] = crearAlumno(50,"Carlos",4);
      int len = 6;
      // invoco a la funcion que muestra el array
      mostrar<Alumno>(arr,len,mostrarAlumno);
  return 0;
}
```

Ordenar arrays de estructuras por diferentes criterios

Recordemos la función `ordenar`:

```
template <typename T>void ordenar(T arr[], int len, int
(*criterio)(T,T))
{
      bool ordenado=false;
      while(!ordenado){
        ordenado=true;
        for(int i=0; i<len-1; i++){
            if( criterio(arr[i],arr[i+1])>0 ){
              T aux = arr[i];
              arr[i] = arr[i+1];
              arr[i+1] = aux;
              ordenado = false;
            }
        }
      }
 return;
}
```

Definimos diferentes criterios de precedencia de alumnos:

```
//a1 precede a a2 si a1.legajo<a2.legajo:
int criterioAlumnoLegajo(Alumno a1, Alumno a2)
{
```

```
      return a1.legajo-a2.legajo;
}
//a1 precede a a2 si a1.nombre<a2.nombre:
int criterioAlumnoNombre(Alumno a1, Alumno a2)
{
      return a1.nombre<a2.nombre?-1:a1.nombre>a2.nombre?1:0;
}
```

a1 precede a a2 si a1.nombre<a2.nombre. A igualdad de nombres entonces precederá el alumno que tenga menor número de legajo:

```
int criterioAlumnoNomYLeg(Alumno a1, Alumno a2)
{
      if( a1.nombre == a2.nombre ){
        return a1.legajo-a2.legajo;
      }
      else{
        return a1.nombre<a2.nombre?-1:a1.nombre>a2.nombre?1:0;
      }
}
```

Ahora sí, probemos los criterios anteriores con la función ordenar.

```
int main()
{
      Alumno arr[6];
      arr[0] = crearAlumno(30,"Juan",5);
      arr[1] = crearAlumno(10,"Pedro",8);
      arr[2] = crearAlumno(20,"Carlos",7);
      arr[3] = crearAlumno(60,"Pedro",10);
      arr[4] = crearAlumno(40,"Alberto",2);
      arr[5] = crearAlumno(50,"Carlos",4);
      int len = 6;
      // ordeno por legajo
      ordenar<Alumno>(arr,len,criterioAlumnoLegajo);
      mostrar<Alumno>(arr,len,mostrarAlumno);
      // ordeno por nombre
      ordenar<Alumno>(arr,len,criterioAlumnoNombre);
      mostrar<Alumno>(arr,len,mostrarAlumno);
      // ordeno por nombre+legajo
      ordenar<Alumno>(arr,len,criterioAlumnoNomYLeg);
      mostrar<Alumno>(arr,len,mostrarAlumno);
  return 0;
}
```

Archivos

8

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

Estructura tipo registro:

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

a) Declaración y definición de un registro

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

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

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

Estructura tipo Archivo

Estructura de datos con almacenamiento físico en disco, persiste mas allá de la aplicación y su procesamiento es lento. Según el tipo de dato se puede diferenciar en archivo de texto (conjunto de líneas de texto, compuestas por un conjunto de caracteres, que finalizan con

una marca de fin de línea y una marca de fin de la estructura, son fácilmente transportables) archivos binarios (secuencia de bytes, en general mas compactos y menos transportables).
Para trabajar con archivos en C es necesario:

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

En algoritmos y estructura de datos trabajamos con

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

Archivos y flujos

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

Los modos de apertura son:

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

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

Archivos de texto:

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

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

Operaciones simples

FILE * AbrirArchTextoLectura(FILE * f, char nombre[]){
// Asocia el flujo f con el archivo nombre, lo abre en modo lectura

```
    return fopen(nombre, “r”);
}

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

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

Patrones algorítmicos con archivos de texto:

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

C

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

Solución algorítmica

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

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

C

```
main() {
FILE *f1;
char s[10 + 1]c;
```

```
 f1 = fopen(“entrada”, “r”);
 fgets(s, 10 + 1, f1);
 while (!feof(f1)) {
   printf(“%s/n”,s);
  fgets(s, 10 + 1,f1);}
 fclose(f1);
 return 0;
}
```

Solución Algorítmica

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

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

C

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

Solución algorítmica

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

Archivos binarios:

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

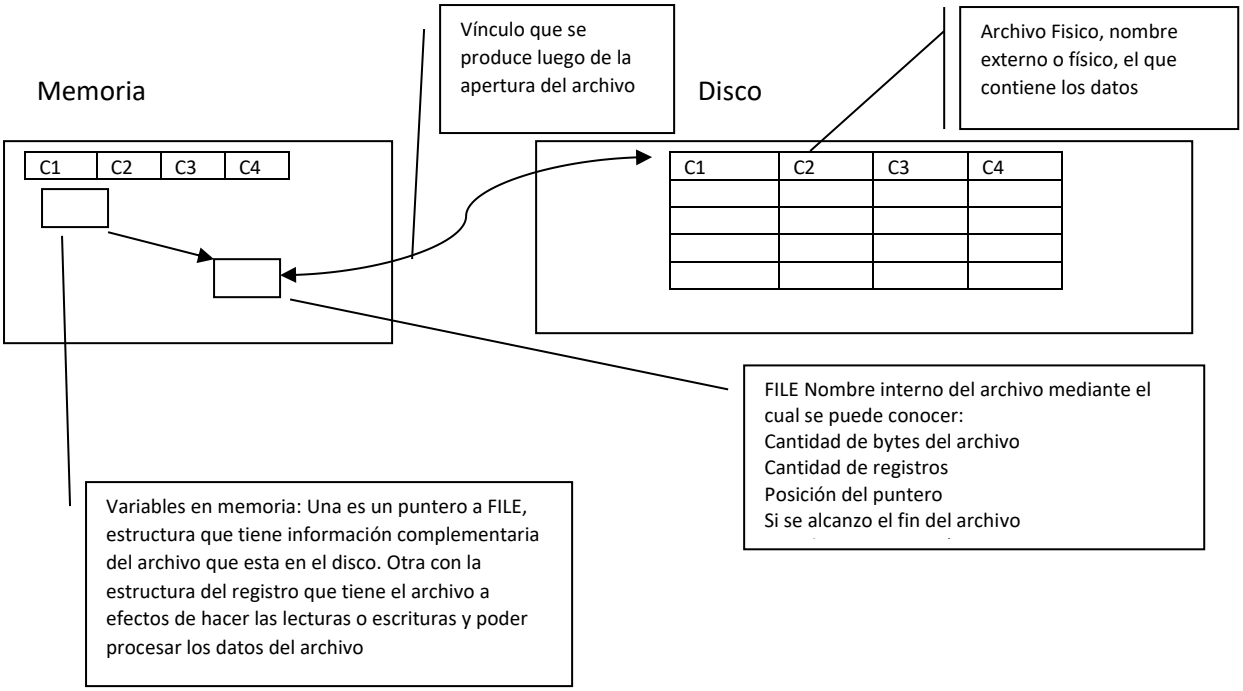
Tenga en cuenta que en algoritmos y estructura de datos trabajamos con distintos tipos de estructuras, muchas de ellas manejan conjunto de datos del mismo tipo a los efectos de resolver los procesos batch que nos presentan distintas situaciones problemáticas de la materia. Así

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

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

En función de estas y otras características particulares de las estructuras de datos se deberá tomar la decisión de seleccionar la más adecuada según las características propias de la situación a resolver.
Como concepto general deberá priorizarse la eficiencia en el procesamiento, por lo que estructuras en memoria y con accesos directos ofrecen la mejor alternativa, lo cual hace a la estructura arreglo muy adecuada para este fin. Por razones varias (disponibilidad del recurso de memoria, desconocimiento a priori del tamaño fijo, necesidad que el dato persista mas allá de la aplicación, entre otras) nos vemos en la necesidad de seleccionar estructuras diferentes para adaptarnos a la solución que buscamos. El problema, aunque puede presentarse como complejo, se resuelve con ciertas facilidad ya que la decisión no es entre un conjunto muy grande de alternativas, simplemente son tres. Una ya la estudiamos, los arreglos, otra es el objeto de este apunte, los archivos, las que quedan las veremos en poco tiempo y son las estructuras enlazadas.
Observen que se pueden hacer cosas bastante similares a las que hacíamos con arreglos, recorrerlos, buscar, cargar, por lo que la lógica del procedimiento en general la tenemos, solo cambia la forma de acceso a cada miembro de la estructura. En realidad las acciones son mas limitadas dado que muchas cosas que si hacíamos con arreglos como por ejemplo búsquedas

secuenciales, carga sin repetición, métodos de ordenamiento, los desestimaremos en esta estructura por lo costoso del procesamiento cuando el almacenamiento es físico (en el disco)
FILE* f=fopen("SL"," ")



Definiciones y declaraciones:

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

Archivo binario

Numero	Cadena	Caracter
int	char cadena[N]	char

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

Operaciones simples

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

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


int  cantidadRegistros( FILE * f){
```

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

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

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

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

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

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

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

ftell(f)	El valor actual del indicador de posición del archivo
fseek(f, cant, desde)	Define indicador de posicion a partir de una posicion.

Implementaciones C C++

Operaciones sobre archivos

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

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

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

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

Grabar un archivo de caracteres

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{
      // abro el archivo; si no existe => lo creo vacio
      FILE* arch = fopen("DEMO.DAT","wb+");
      char c = 'A';
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'A' contenido en c
      c = 'B'; // C provee también fputc(c, arch);
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'B' contenido en c
      c = 'C';
      fwrite(&c,sizeof(char),1,arch); // grabo el caracter 'C' contenido en c
      fclose(arch);
  return 0;}
```

Leer un archivo caracter a caracter

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

Archivos de registros

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

Supongamos tener un aechivo de registros con la siguiente estructura:

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

Grabar un archivo de registros

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

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

```
        cout << "Ingrese dni: ";
        cin >> dni;
       }
      fclose(f);
  return 0;
}
```

Leer un archivo de registros

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

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

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

Acceso directo a los registros de un archivo

Acceder al registro de la quinta posición y mostrar su contenido

```
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int main()
{
      int N=10;
      FILE* f = fopen("Alumnos.DAT","r+b");
      Alumno a;

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

Acceder al ultimo registro mostrar su contenido

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

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

Templates

Template: leer

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

Template: grabar

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

Template: seek

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

Ejemplos

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

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

	  cout << R.dni<<", "<<R.nombre << endl;

	}
	fclose(f);
```

Biblioteca Standard C

1.1. Definiciones Comunes <stddef.h>

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

size_t

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

NULL

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

1.2. Manejo de Caracteres <ctype.h>

int isalnum (int);

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

int isalpha (int);

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

int isdigit (int);

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

int islower (int);

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

int isprint (int);

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

int isspace (int);

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

int isupper (int);

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

int isxdigit (int);

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

int tolower (int c);

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

Retorna (mayúscula ? minúscula : **c**)

int toupper (int c);

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

1.3. Manejo de Cadenas <string.h>

Define el tipo **size_t** y la macro **NULL**, ver *Definiciones Comunes*.
unsigned strlen (const char*);
Cuenta los caracteres que forman la cadena dada hasta el 1er carácter **'\0'**, excluido. Retorna (longitud de la cadena).

1.3.1. Concatenación

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

1.3.2. Copia

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

1.3.3. Búsqueda y Comparación

char* strchr (const char* s, int c);
Ubica la 1ra. aparición de **c** (convertido a **char**) en la cadena **s**; el **'\0'** es considerado como parte de la cadena. Retorna (ok ? puntero al carácter localizado : **NULL**)
char* strstr (const char* s, const char* t);
Ubica la 1ra. ocurrencia de la cadena t (excluyendo al '\0') en la cadena s. Retorna (ok ? puntero a la subcadena localizada : **NULL**).
int strcmp (const char*, const char*);
Compara "lexicográficamente" ambas cadenas. Retorna (0 si las cadenas son iguales; < 0 si la 1ra. es "menor" que la 2da.; > 0 si la 1ra. es "mayor" que la 2da.)
int strncmp (const char* s, const char* t, size_t n);
Compara hasta **n** caracteres de **s** y de **t**. Retorna (como **strcmp**).
char* strtok (char*, const char*);
Separa en "tokens" a la cadena dada como 1er. argumento; altera la cadena original; el 1er. argumento es la cadena que contiene a los "tokens"; el 2do. argumento es una cadena con caracteres separadores de "tokens". Retorna (ok ? puntero al 1er. carácter del "token" detectado : **NULL**).

1.3.4. Manejo de Memoria

void* memchr(const void* s, int c, size_t n);
Localiza la primer ocurrencia de **c** (convertido a un **unsigned char**) en los **n** iniciales caracteres (cada uno interpretado como **unsigned char**) del objeto apuntado por **s**. Retorna (ok ? puntero al carácter localizado : **NULL**).
int memcmp (const void* p, const void* q, unsigned n);
Compara los primeros **n** bytes del objeto apuntado por **p** con los del objeto apuntado por **q**. Retorna (0 si son iguales; < 0 si el 1ero. es "menor" que el 2do.; > 0 si el 1ero. es "mayor" que el 2do.)
void* memcpy (void* p, const void* q, unsigned n);
Copia **n** bytes del objeto apuntado por **q** en el objeto apuntado por **p**; si la copia tiene lugar entre objetos que se superponen, el resultado es indefinido. Retorna (**p**).
void* memmove (void* p, const void* q, unsigned n);

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

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

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

1.4. Utilidades Generales <stdlib.h>

1.4.1. Tips y Macros

size_t

NULL

Ver *Definiciones Comunes*.

EXIT_FAILURE

EXIT_SUCCESS

Macros que se expanden a expresiones constantes enteras que pueden ser utilizadas como argumentos de **exit** ó valores de retorno de **main** para retornar al entorno de ejecución un estado de terminación no exitosa o exitosa, respectivamente.

RAND_MAX

Macro que se expande a una expresión constante entera que es el máximo valor retornado por la función **rand**, como mínimo su valor debe ser 32767.

1.4.2. Conversión

```
double atof (const char*);
```

Convierte una cadena que representa un real **double** a número **double**. Retorna (número obtenido, no necesariamente correcto).

```
int atoi (const char*);
```

Convierte una cadena que representa un entero **int** a número **int**. Retorna (número obtenido, no necesariamente correcto) .

```
long atol (const char*);
```

Convierte una cadena que representa un entero **long** a número **long**. Retorna (número obtenido, no necesariamente correcto).

```
double strtod (const char* p, char** end);
```

Convierte como **atof** y, si el 2do. argumento no es **NULL**, un puntero al primer carácter no convertible es colocado en el objeto apuntado por **end**. Retorna como **atof**.

```
long strtol (const char* p, char** end, int base);
```

Similar a **atol** pero para cualquier base entre 2 y 36; si la base es 0, admite la representación decimal, hexadecimal u octal; ver **strtod** por el parámetro **end**. Retorna como **atol**.

```
unsigned long strtoul (const char* p, char** end, int base);
```

Igual que **strtol** pero convierte a **unsigned long**. Retorna como **atol**, pero **unsigned long**.

1.4.3. Administración de Memoria

```
void* malloc (size_t t);
```

Reserva espacio en memoria para almacenar un objeto de tamaño **t**. Retorna (ok ? puntero al espacio reservado : **NULL**)

```
void* calloc (size_t n, size_t t);
```

Reserva espacio en memoria para almacenar un objeto de **n** elementos, cada uno de tamaño **t**. El espacio es inicializado con todos sus bits en cero. Retorna (ok ? puntero al espacio reservado : **NULL**)

```
void free (void* p);
```

Libera el espacio de memoria apuntado por **p**. No retorna valor.

```
void* realloc (void* p, size_t t);
```

Reubica el objeto apuntado por **p** en un nuevo espacio de memoria de tamaño **t** bytes. Retorna (ok ? puntero al posible nuevo espacio : **NULL**).

1.4.4. Números Pseudo-Aleatorios

```
int rand (void);
```

Determina un entero pseudo-aleatorio entre 0 y **RAND_MAX**. Retorna (entero pseudo-aleatorio).

```
void srand (unsigned x);
```

Inicia una secuencia de números pseudo-aleatorios, utilizando a **x** como semilla. No retorna valor.

1.4.5. Comunicación con el Entorno

```
void exit (int estado);
```

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

```
void abort (void);
```

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

```
int system (const char* lineadecomando);
```

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

1.4.6. Búsqueda y Ordenamiento

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

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

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

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

1.5. Entrada / Salida <stdio.h>

1.5.1. Tipos

size_t

Ver *Definiciones Comunes*.

FILE

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

fpos_t

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

1.5.2. Macros

NULL
Ver *Definiciones Comunes*.
EOF
Expresión constante entera con tipo **int** y valor negativo que es retornada por varias funciones para indicar *fin de archivo*; es decir, no hay mas datos entrantes que puedan ser leídos desde un *flujo*, esta situación puede ser porque se llegó al fin del archivo o porque ocurrió algún error. Contrastar con **feof** y **ferror**.
SEEK_CUR
SEEK_END
SEEK_SET
Argumentos para la función **fseek**.
stderr
stdin
stdout
Expresiones del tipo **FILE*** que apuntan a objetos asociados con los flujos estándar de error, entrada y salida respectivamente.

1.5.3. Operaciones sobre Archivos

```
int remove(const char* nombrearchivo);
```
Elimina al archivo cuyo nombre es el apuntado por **nombrearchivo**. Retorna (ok ? 0 : ≠0)
```
int rename(const char* viejo, const char* nuevo);
```
Renombra al archivo cuyo nombre es la cadena apuntada por **viejo** con el nombre dado por la cadena apuntada por **nuevo**. Retorna (ok ? 0 : ≠0).

1.5.4. Acceso

```
FILE* fopen (
     const char* nombrearchivo,
     const char* modo
);
```
Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** asociando un flujo con este según el **modo** de apertura. Retorna (ok ? puntero al objeto que controla el flujo : **NULL**).
```
FILE* freopen(
     const char* nombrearchivo,
     const char* modo,
     FILE* flujo
);
```
Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** y lo asocia con el flujo apuntado por **flujo**. La cadena apuntada por **modo** cumple la misma función que en **fopen**. Uso más común es para el redireccionamiento de **stderr**, **stdin** y **stdout** ya que estos son del tipo **FILE*** pero no necesariamente *lvalues* utilizables junto con **fopen**. Retorna (ok ? flujo : **NULL**).
```
int fflush (FILE* flujo);
```
Escribe todos los datos que aún se encuentran en el buffer del flujo apuntado por **flujo**. Su uso es imprescindible si se mezcla **scanf** con **gets** o **scanf** con **getchar**, si se usan varios **fgets**, etc. Retorna (ok ? 0 : **EOF**).
```
int fclose (FILE* flujo);
```
Vacía el *buffer* del flujo apuntado por **flujo** y cierra el archivo asociado. Retorna (ok ? 0 : **EOF**)

1.5.5. Entrada / Salida Formateada

Flujos en General

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

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

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

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

Flujos stdin y stdout

```
int scanf (const char*, ...);
```

Lectura formateada desde **stdin**. Retorna (ok ? cantidad de ítems almacenados : **EOF**).

```
int printf (const char*, ...);
```

Escritura formateada sobre **stdout**. Retorna (ok ? cantidad de caracteres transmitidos : < 0).

Cadenas

```
int sprintf (char* s, const char*, ...);
```

Escritura formateada en memoria, construyendo la cadena **s**. Retorna (cantidad de caracteres escritos).

```
int sscanf (const char* s, const char*, ...);
```

Lectura formateada desde una cadena **s**. Retorna (ok ? cantidad de datos almacenados : **EOF**).

1.5.6. Entrada / Salida de a Caracteres

```
int fgetc (FILE*); ó
int getc (FILE*);
```

Lee un carácter (de un archivo ASCII) o un byte (de un archivo binario). Retorna (ok ? carácter/byte leído : **EOF**).

```
int getchar (void);
```

Lectura por carácter desde **stdin**. Retorna (ok ? próximo carácter del buffer : **EOF**).

```
int fputc (int c, FILE* f); ó
int putc (int c, FILE* f);
```

Escribe un carácter (en un archivo ASCII) o un byte (en un archivo binario). Retorna (ok ? **c** : **EOF**).

```
int putchar (int);
```

Eescritura por carácter sobre stdout. Retorna (ok ? carácter transmitido : **EOF**).

```
int ungetc (int c, FILE* f);
```

"Devuelve" el carácter o byte **c** para una próxima lectura. Retorna (ok ? **c** : **EOF**).

1.5.7. Entrada / Salida de a Cadenas

```
char* fgets (char* s, int n, FILE* f);
```

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

```
char* gets (char* s);
```

Lectura por cadena desde **stdin**; es mejor usar **fgets**() con **stdin** . Retorna (ok ? **s** : **NULL**).

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

Escribe la cadena apuntada por **s** en el flujo **f**. Retorna (ok ? último carácter escrito : **EOF**).

```
int puts (const char* s);
```

Escribe la cadena apuntada por **s** en **stdout**. Retorna (ok ? ≥ 0 : **EOF**).

1.5.8. Entrada / Salida de a Bloques

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

1.5.9. Posicionamiento

```
int fseek (
     FILE* flujo,
     long desplazamiento,
     int desde
);
```

Ubica el *indicador de posición de archivo* del flujo binario apuntado por **flujo**, **desplazamiento** caracteres a partir de **desde**. **desde** puede ser **SEEK_SET**, **SEEK_CUR** ó **SEEK_END**, comienzo, posición actual y final del archivo respectivamente. Para flujos de texto, **desplazamiento** deber ser cero o un valor retornado por **ftell** y **desde** debe ser **SEEK_SET**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).
`int fsetpos (FILE* flujo, const fpos_t* posicion);`
Ubica el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** según el valor del objeto apuntado por **posicion**, el cual debe ser un valor obtenido por una llamada exitosa a **fgetpos**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).
`int fgetpos (FILE* flujo, fpos_t* posicion);`
Almacena el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** en el objeto apuntado por **posicion**, cuyo valor tiene significado sólo para la función **fsetpos** para el restablecimiento del *indicador de posición de archivo* al momento de la llamada a **fgetpos**. Retorna (ok ? 0 : ≠ 0).
`long ftell (FILE* flujo);`
Obtiene el valor actual del *indicador de posición de archivo* para el flujo apuntado por **flujo**. Para flujos binarios es el número de caracteres (bytes ó posición) desde el comienzo del archivo. Para flujos de texto la valor retornado es sólo útil como argumento de **fseek** para reubicar el indicador al momento del llamado a **ftell**. Retorna (ok ? indicador de posición de archivo : –1L).
`void rewind(FILE *stream);`
Establece el indicador de posición de archivo del flujo apuntado por **flujo** al principio del archivo. Semánticamente equivalente a **(void)fseek(stream, 0L, SEEK_SET)**, salvo que el indicador de error del flujo es desactivado. No retorna valor.

Archivos binarios: Analisis, síntesis y comparaciones

En la implementación en C utilizamos
FILE* se asocia a un flujo para controlarlo→ FILE* f;
fopen asocia el nombre lógico al físico→ f=fopen(“NombreFisico”,”rb+”);
fread lectura por bloques→fread(&r, sizeof(r), 1, f);
fwrite escritura por bloques→fwite(&r, sizeof(r), 1, f);
feof indicador de final del archivo→feof(f)
fseek permite acceso directo→ fseek(f,2*sizeof(r),SEEK_SET)
ftell retorna de bytes de desplazamiento desde el inicio del flujo→ftell(f);
fclose cierra el archivo, vacía el buffer y coloca marca de EOF→flose(f)
struct tr{ int c1; int c2;} → supongo entero de 8 bytes

8		16		24		32		40	
3	5	4	1	2	8	14	5	5	11

ftell(f) →8 24 36
fseek(f, 16, SEEK_SET) →
fseek(f, 2*8, SEEK_CUR) →
fseek(f, -sizeof(r), SEEK_END) ←
ftell(f)/sizeof(r) → 2 → 4
fseek(f,0,SEEK_END); p = ftell(f)/sizeof(r);→calcula cantidad de registros
tr v[10]
FILE* f = fopen(“ma”,”rb+”)

a=fread(&reg, sizeof(reg),1,f)

3	5

fread(&a, sizeof(int),1,f)

3

fread(v, sizeof(reg),4,f) v → vector de struct; v[0] → struct pos 0


```
fread(v, sizeof(reg),4,f)
fread(&v[1], sizeof(reg),4,f)


fread(     );
while(! feof(f) ){
 insertarordenado(lista, reg);
fread(     );
}

while(fread(   )){
insertarordenado(lista, reg);
}
```

Análisis comparativo de Archivos Vectores→Declaración, Acceso

Vector (de registros)	Archivo (de registros)
Consideraciones generales	
Almacenamiento lógico (memoria) Tamaño Fijo (T. E.). Procesamiento rápido Sin persistencia después aplicación Prioriza velocidad procesamiento	Almacenamiento físico (disco) Tamaño variable (T.E.). Procesamiento lento Con persistencia después aplicación Garantiza persistencia
Definiciones y declaraciones	
Tr V[X]; Note que determina a priori el tamaño (X) y especifica el tipo de dato de cada posición(Tr).	FILE* f = fopen ("XXXX", "rb+"); Solo indica como lo abre sin especificar particularidad del dato, solo "b o t" y no el tamaño
Acceder al registro de posición N	

V[N];
Accede al registro en memoria con esa posición.
	fseek(f, N*sizeof(r), SEEK_SET);
	posiciona el puntero en el registro indicado
	fread(&r, sizeof(r), 1, f);
	Lleva a memoria el registro N.

Modificar el registro de la posición N	

V[N].campo = valor
Modifica el campo especifico del registro N que está en memoria
	fseek(f, N*sizeof(r), SEEK_SET);
	APUNTAR al registro a modificar
	fread(&r, sizeof(r), 1, f);
	LEER Lo lleva a memoria.
	r.campo = valor
	MODIFICAR el dato en memoria
	fseek(f, N*sizeof(r), SEEK_SET);
	Vuelve APUNTAR al registro.
	fwrite(&r, N*sizeof(r), 1, f);
	GRABAR en el disco.

Acceder al registro siguiente al de la posición N y tenerlo a disposición	

V[N+1]	fseek(f, (N+1)*sizeof(r), SEEK_SET);
	APUNTAR al registro a modificar
	fread(&r, sizeof(r), 1, f);
	LEER Lo lleva a memoria.

Acceder al primer registro y tenerlo a disposición	
V[0]	fseek(f, 0, SEEK_SET); fread(&r, sizeof(r), 1, f);

<table>
<tr><th colspan="2">Acceder al último registro y tenerlo a disposición</th></tr>
<tr><td>V[X-1]</td><td>fseek(f, -sizeof(r), SEEK_END);
fread(&r, sizeof(r), 1, f);</td></tr>
<tr><th colspan="2">Búsqueda binaria</th></tr>
<tr><td>int bb(Tr V[], int X, int N){
int p = 0;
int u = N-1;
int m;

while(p<=u){
m = (p + u)/2;

if(V[m]. campo == X) return m;
else if(X>V[m].campo p= m++;
else u = m--;
}
return -1;
}</td><td>int bb(FILE* f,int X){
int p = 0;
int u = cantRegistros(f)-1;
int m;
Tr r;
Int tReg = sizeof(r);

while(p<=u){
m = (p + u)/2;

fseek(f,m*tReg, SEEK_SET);
fread(&r, sizeof(r), 1, f);

if(r. campo == X) return m;
else if(X>r.campo p= m++;
else u = m--;
}
return -1;
}</td></tr>
<tr><th colspan="2">Búsqueda directa (PUP)</th></tr>
<tr><td>V[Clave-vInicial]</td><td>fseek(f,sizeof(r)*(Clave-vInicial),SEEK_SET)
APUNTAR al registro buscado
fread(&r, sizeof(r), 1, f);
LEER el registro apuntado</td></tr>
</table>

Apareo (conceptual)

Concepto: Aplicable a dos o más estructuras son al menos un campo en común ordenadas por ese campo que se procesan paralelamente, intercalando los valores para conservar el orden

Seudocódigo:

```
Situarse al principio de ambas estructuras
Hacer mientras (haya datos en ambas)
 Si la primera cumple criterio de ordenamiento
        Procesarla;
        Avanzar con ella
  Sino
        Procesar la otra;
        Avanzar con ella
Fin del mientras
Agotar la estructura que no se termino
    Procesarla;
    Avanzar
 Fin del proceso
```

Vectores

```
int i = 0;
int j = 0;

while((i<N) && (j<M)){
          if(v1[i].c1<v2[j].c1){
             //procesar v1[i];
             i++;}
          else{
             //procesar v2[j];
             j++;}
       }

while(i<N){
   //procesar v1[i];
        i++;    }

while(j<M){
   //procesar v2[j];
        j++;}
```

Archivos

```
fread(&r1,sizeof(r1), 1, f1);
fread(&r2,sizeof(r2), 1, f2);

while(!feof(f1)&& !feof(f2)){
  if(r1.c<r2.c){
     // procesar r1;
      fread(&r1,sizeof(r1), 1, f1)};
   else{
     // procesar r2;
      fread(&r2,sizeof(r2), 1, f2) ;}
 };

while(!feof(f1)){
      procesar r1;
      fread(&r1,sizeof(r1), 1, f1)};

while(!feof(f2)){
      procesar r2;
      fread(&r2,sizeof(r2), 1, f2)};
```

1
8
14
22
25

2
3
11

```
int j = 0; int i = 0;
// j==M || i<N && v1[i].c < v2[j].c
while((i<N) || (j<M)){
          if(j==M || i<N && v1[i].c < v2[j].c){
             //procesar v1[i];
             i++;}
          else{
             //procesar v2[j];
             j++;}
  }
```

Con plantilla y función criterio

Implementación de la función

```
template <typename T>
void Apareo(T v1[], T v2[], int N, int M, int
(*criterio)(T,T)){
   int i = 0;
   int j = 0;
   while((i<N) && (j<M)){
      if(criterio(v1[i],v2[j])==1){
         //procesar v1<T>[i];
         cout << v1[i].c1 << endl;
         i++;} else{
         //procesar v2<T>[j];
         cout << v2[j].c1 << endl;
         j++;}    };
while(i<N){
      //procesar v1<T>[i];
      cout << v1[i].c1 << endl;
      i++; }
while(j<M){
     //procesar v2<T>[j];
     cout << v2[j].c1 << endl;
     j++;  } }
```

programa principal e Invocación

```
struct tr{ int c1; intc2};
int criterioCampo1Asc(tr a,tr b){
  if(a.c1<b.c1) return 1;
  return 0;
}

int criterioCampo1Des(tr a,tr b){
  if(a.c1>b.c1) return 1;
  return 0;
}
int criterioCampo1yCampo2Des(tr a,tr b){
if(a.c1>b.c1||( a.c1==b.c1&& a.c2>b.c2) return 1;
  return 0;
}
int main (){
……….
Apareo<tr>(v1,v2,N,M,criterioCampo1Asc)

Apareo<tr>(v1,v2,N,M,criterioCampo1Desc)

Apareo<tr>(v1,v2N,M,,criterioCampo1yCampo2Desc)
```

EJEMPLO COMPLETO EN C++ con plantillas y criterio de selección

Función completa

Programa que lo usa

```
int main()
{
int N=5;
int M=6;
//PARA CRITERIO ASCENDENTE
tr v1[N]={{1,23},{22,18},{25,40},{70,120},{103,99}};
tr
v2[M]={{4,23},{11,18},{12,40},{88,120},{102,99},{500,99}};

//PARA CRITERIO DESCENDENTE
//tr v1[N]={{103,23},{70,18},{25,40},{22,120},{1,99}};
//tr
v2[M]={{500,23},{102,18},{88,40},{12,120},{11,99},{4,99}};


Apareo<tr>(v1,v2,N,M,criterioCampo1Asc);

//Apareo<tr>(v1,v2,N,M,criterioCampo1Des);

//Apareo<tr>(v1,v2,N,M,criterioCampo1yCampo2Desc);

}
```

```
// ejemplo complete de aparep con funciones de ctiterio
nclude <iostream>
using namespace std;
struct tr{
        int c1;
        int c2;
};
void apareoVectores(int ,int ,tr[],tr[]);
```

```
void apareoArchivos(FILE*,FILE*);
int criterioCampo1Asc(tr ,tr );
int criterioCampo1Des(tr ,tr );
int criterioCampo1yCampo2Des(tr ,tr );

template <typename T>
void Apareo(T [], T [], int , int , int (*)(T,T));

int main()
{
            tr datos;
            int N=5;
            int M=6;
            //PARA CRITERIO ASCENDENTE
            tr v1[N]={{1,23},{22,18},{25,40},{70,120},{103,99}};
            tr v2[M]={{4,23},{11,18},{12,40},{88,120},{102,99},{500,99}};

//          apareoVectores(N,M,v1,v2);

            //PARA CRITERIO DESCENDENTE
//          tr v1[N]={{103,23},{70,18},{25,40},{22,120},{1,99}};
//          tr v2[M]={{500,23},{102,18},{88,40},{12,120},{11,99},{4,99}};


//          Apareo<tr>(v1,v2,N,M,criterioCampo1Asc);

//          Apareo<tr>(v1,v2,N,M,criterioCampo1Des);

//          Apareo<tr>(v1,v2,N,M,criterioCampo1yCampo2Desc);

}
template <typename T>
void Apareo(T v1[], T v2[], int N, int M, int (*criterio)(T,T)){
   int i = 0;
   int j = 0;
   while((i<N) && (j<M)){
      if(criterio(v1[i],v2[j])==1){
         //procesar v1<T>[i];
         cout << v1[i].c1 << endl;
         i++;
                  }
      else{
         //procesar v2<T>[j];
         cout << v2[j].c1 << endl;
         j++;
                  }
   };
   //RECORRO PARA VER SI ME QUEDARON DATOS EN LAS ESTRUCTURAS
   while(i<N){
      //procesar v1<T>[i];
      cout << v1[i].c1 << endl;
      i++;
          }
   while(j<M){
     //procesar v2<T>[j];
     cout << v2[j].c1 << endl;
     j++;
```

```
        }
}
int criterioCampo1Asc(tr a,tr b){
  if(a.c1<b.c1)
                return 1;
  return 0;
}

int criterioCampo1Des(tr a,tr b){
  if(a.c1>b.c1)
                return 1;
  return 0;
}

int criterioCampo1yCampo2Des(tr a,tr b){

        if ((a.c1>b.c1)||(a.c1==b.c1&& a.c2>b.c2))
                return 1;
        return 0;

}


void apareoVectores(int N,int M,tr v1[],tr v2[])
{
        int i = 0;
        int j = 0;

        while((i<N) && (j<M)){
          if(v1[i].c1<v2[j].c1){
              //procesar v1[i];
              cout << v1[i].c1 << endl;
              i++;
                  }
          else{
              //procesar v2[j];
               cout << v2[j].c1 << endl;
              j++;
                  }
        }
        //RECORRO PARA VER SI ME QUEDARON DATOS EN LAS ESTRUCTURAS
        while(i<N){
   //procesar v1[i];
        cout << v1[i].c1 << endl;
        i++;
        }

        while(j<M){
   //procesar v2[j];
        cout << v2[j].c1 << endl;
        j++;
        }
}

void apareoArchivos(FILE* f1,FILE* f2)
{
        tr r1,r2;
```

```
        fread(&r1,sizeof(r1), 1, f1);
        fread(&r2,sizeof(r2), 1, f2);

        while(!feof(f1) && !feof(f2)){
                if(r1.c1<r2.c1){
        //procesar r1;
        cout << r1.c1 << endl;
        fread(&r1,sizeof(r1), 1, f1);

                }
        else{
        //procesar r2;
        cout << r2.c1 << endl;
        fread(&r2,sizeof(r2), 1, f2);

                }
        }
        //RECORRO PARA VER SI ME QUEDARON DATOS EN LAS ESTRUCTURAS
        while(!feof(f1)){
    //procesar r1;
    cout << r1.c1 << endl;
    fread(&r1,sizeof(r1), 1, f1);
        }

        while(!feof(f2)){
    //procesar r2;
    cout << r2.c1 << endl;
    fread(&r2,sizeof(r2), 1, f2);
        }

}
```

Corte de Control (conceptual)

Concepto: Aplicable a una estructura con al menos un campo que se repite, agrupados por ese campo.
Propósito: Mostrar los datos en forma ordenada sin repetir ese campo común.

Como se requieren los datos

Materia	Legajo	Nota
AyED	145234	5
AyED	234169	3
AyED	135321	8
SSL	242132	9
SSL	125328	7
SSL	146342	2

Ejemplo de muestra de resultados

Materia AyED

Orden	Legajo	Nota
1	145234	5
2	234169	3
3	135321	8

Cantidad Alumnos 3

Materia SSL

Orden	Legajo	Nota
1	242132	9
2	125328	7
3	135321	2

Cantidad Alumnos 3

Total Alumnos 6

Seudocódigo:

Situarse al principio de la estructura
Inicializar contadores generales
Mostrar títulos generales

Hacer mientras (haya datos)
  Inicializar contadores de cada subgrupo
  Mostrar títulos subgrupos
  Identificar grupo a analizar→subgrupo

  Hacer mientras (haya datos Y mismo subgrupo)
    Procesar el registro
    Avanzar al siguiente
  Fin ciclo interno
 Informar datos de cada subgrupo
Fin ciclo externo
Informar sobre generalidades

Observar

El ciclo interno tiene la conjunción de la condición del ciclo externo y la propia.
Las lecturas deben hacerse al final del ciclo interno menor.
Este Patrón algorítmico NO ORDENA, solo muestra agrupados los datos que ya están ordenados evitando repeticiones innecesarias. En síntesis solo presenta los datos en una forma mas adecuada

Vectores

```
int i = 0;
totalAlumnos = 0;
<TITULOS>
while(i<N){
  alumnosCurso = 0
  Control = V[i].curso
  <TITULOS>
while(i>N&&V[i].curso==control)
   alumnosCurso++
   totalAlumnos ++
   Mostrar Datos del V[i]
   i++ }// fin ciclo interno
   Mostrar Datos del Subgrupo
} // fin ciclo externo
Resultados Finales
```

Archivos

```
fread(&r1,sizeof(r1), 1, f1);
totalAlumnos = 0;
<TITULOS>
while(!feof(f){
  alumnosCurso = 0
  Control = r.curso
  <TITULOS>
 while(!feof(f)&&control==r.curso)
   alumnosCurso++
   totalAlumnos ++
   Mostrar Datos del registro
   fread(&r1,sizeof(r1), 1, f1);}
   Mostrar Datos del Subgrupo
} // fin ciclo externo
Resultados Finales
```

Otros patrones Apareo

Apareo por criterios diferentes

- Por más de un campo de ordenamiento
  - conservar algoritmia modificando la lógica → agrupar criterios →&&
- Modificar criterio
  - Cambiando algoritmia
  - Utilización de funciones de criterio

Apareo de N estructuras

- Ir apareando por pares y luego aparear loe resultados
- Utilizar una estructura auxiliar →V1

A1
4
6
12
20

A2
3
7
19
25

A3
1
2
5
8

V1

Clave	Control
20	0
25	0
8	0

```
fread(&r1,sizeof(r1), 1, f1)
V1[0].clave = r1.clave
fread(&r2,sizeof(r2), 1, f2)
V1[1].clave = r2.clave
fread(&r3,sizeof(r3), 1, f3)
V1[2].clave = r3.clave
archivosActivos = 3
while(archivosActivos>0){
  p = buscarMinimoNoCero(V)
switch(p)
{
   case 0: //procesar archivo1, si es feof poner 0 en el control y disminuir archivosActivos
   break;
   case 1: // procesar archivo2, si es feof poner 0 en el control y disminuir archivosActivos;
   break;
   case 2: // procesar archivo3, si es feof poner 0 en el control y disminuir archivosActivos
   break;
}
```

```
int buscarMinimoNoCero( tr V[], int N){
int i, minimo;
for( i = 0;V[i].control!=0; i++);
minimo = V[i].clave; i++;
for( ; i<N ; i++){
 if(V[i].clave < minimo) minimo = V[i].clave;
……..}
return posición del minimo
```

4	Juan
2	Jose
6	Pedro
8	Pablo
9	Ana
1	maria

Como conocemos el tamaño guardamos en un vector

```
struct tr {
int NL;
char Nombre[20];
};
Tr R; int i;
Tr Vector[10];
FILE* f = fopen (“Alumnos”,  “rb+);
fread(Vector,sizeof(R),6,f);//archivo y vector ambos desordenados
OrdenarVector(Vector,6);//archivo desordenado, vector ordenado
```

1	maria
2	Jose
4	Juan
6	Pedro
8	Pablo
9	Ana

```
// mostrar los datos por pantalla orenados→ datos que están en el vector
for( i=0;i<6;i++)
        cout<<Vector[i].NL << Vector[i].Nombre;


//ordenar el archivo → usando vector estructura auxiliar
fseek(f,0,SEEKSET);
fwrite(Vector,sizeof(R),6,f);//guarda todos los datos del vector en el archovo
```

4	Juan
2	Jose
6	Pedro
8	Pablo
9	Ana
1	maria

Como NO conocemos el tamaño guardamos en una lista

```
struct tr {
int NL;
char Nombre[20];
};
struc Nodo{
 tr info;
Nodo* sgte
};
Nodo* lista = NULL;
Nodo * P = NULL
```

Info		sgte
NL	Nombre	Nodo *

```
Tr R; int i;
FILE* f = fopen ("Alumnos",  "rb+);


while(fread(&R,sizeof(R),1,f)//cargar archivo en lista. Archivo desordenado, lista ordenada
        InsertarOrdenado(lista,R);

// mostrar los datos por pantalla ordenados→ datos que están en la lista
// recorro la lista sin vaciarla
P = lista
for( ;P!=NULL;){
        cout<<P->info.NL <<P->info,Nombre;
        P= P->sgte;
};

//ordenar el archivo → usando lista estructura auxiliar y vacindola
fseek(f,0,SEEKSET);

while(lista!=NULL)
        R = pop(lista);
        fwrite(&R,sizeof(R),1,f);//guarda todos los datos del vector en el archivo
```

NL Nombre

1	R
4	D
5	W

NL Nombre

2	J
3	A
7	M
9	G
14	P
25	Q

Archivo Leer(A1, R1)

R1

Fin de archive	W

R2

9	G

```
fread(&R1, sizeof(R1), 1, A1);
fread(&R2, sizeof(R1), 1, A2);
while(!feof(A1)&&!feof(A2)){ // hay datos en ambos
        if(R1.NL < R2.NL){
                cout<<R1.NL<<R1.nombre;
                fread(&R1, sizeof(R1), 1, A1);
                }
        else{
                cout<<R2.NL<<R2.nombre;
                fread(&R2, sizeof(R2), 1, A2);
                }
};

while(!feof(A2)){ // agota el 2do si no se termino
        cout<<R2.NL<<R2.nombre;
        fread(&R2, sizeof(R2), 1, A2);
}

while(!feof(A1)){ //agota el 1ro I no se termino
        cout<<R1.NL<<R1.nombre;
        fread(&R1, sizeof(R1), 1, A1);
}
```

Apareo entre archivo y vector

```
int i = 0;
fread(&r, sizeof(r), 1, f);


while((i<N) && (!feof(f))){
            if(v1[i].c1<r.c1){
                //procesar v1[i];
                i++;}
            else{
                //procesar r;
               fread(&r, sizeof(r), 1, f) ;}
         }

while(i<N){
  //procesar v1[i];
         i++;      }

while(!feof(f)){
  //procesar r;
  fread(&r, sizeof(r), 1, f) ;
}
```

Apareo de listas

```
Nodo* p = lista1;
Nodo* q = lista2;
Nodo* lista3= NULL;
Nodo* fin = NULL;
while(p!=NULL&&q!=NULL){
    if(p->info.c1<q->info.c1){
       queue(lista3, fin,p->info);
       p = p->sgte;
    else
      queue(lista3, fin, q->info);
      q = q->sgte;
     }

while(p!= NULL){
   //procesar p->info
   p = p->sgte;

while(q!= NULL){
   //procesar q->info
   q = q->sgte;}
   fin = NULL;}
```

Como pensar los problemas de Algoritmos

Estructuras de datos: Características

	Vector	Archivo	Listas ordenadas
Almacenamiento	logico	fisico	Logico
Procesamiento	rapido	Lento	Rapido
Tamaño en T.E.	Fijo	Variable	Variable
Tamaño cada posicion	Informacion	Informacion	Info + sgte
Busqueda directa	SI→V[N]	fseek	Buscar secuencialmente
Persistencia	NO	SI	NO
Busqueda binaria	SI	SI	NO→arbol
Busqueda secuencial	SI	NO se recomienda	SI→unica posible
Carga directa	SI →V[N] = valor	Acceder → fseek Grabar → frwrte	Utilizando diferentes funciones
Carga secuencial	SI→v[ultimaPos] = valor	Acces. a ultima pos grabar	SI Insertar ordenado
Carga sin repetir	Buscar No esta agregar	NO se recomienda Est. Auxiliares	Buscar No esta agregar
Ordenamiento	PUP Metodos de ordenam.	PUP Solo usando est. aux	Genera insertarOrdenado
Definicion	Td Nombre[tamaño]	FILE*f; f= fopen(…..)	Definir Nodo → struct Punt Nodo* X = NULL
Acceso	V[N]	Acceder → fseek leerr → fread	Buscar secuencialmente
Corte de control	SI	SI	SI
Apareo	SI	SI	SI
Modificar la pos N	V[N] = valor	Apuntar → fseek Leer→fread Modificar en memoria Apuntar → fseek Grabar → fwrite	Buscar secuencialmente modificar

Estructuras de datos: Criterios de selección

1. Priorizar de ser posible acceso directo y velocidad de procesamiento.
    a. Vector en primer lugar si se cumple tamaño fijo, razonable y conocido a priori, y sin necesidad de persistencia→ acceso directo, búsqueda binaria, búsqueda secuencial
2. Si no se conoce el tamaño y no se requiere persistencia
    a. Estructuras enlazadas
        i. Pila si se debe invertir el orden o si es irrelevante
        ii. Colas si se debe mantener
        iii. Listas si se debe generar
3. Archivo si se requiere persistencia: directamente o estructuras auxiliares después cargar al archivo

Estructuras → muchos datos del mismo tipo

Archivos

Vectores → memoria → mejorar tiempos de procesamiento → tamaño fijo

Listas ordenadas → memoria →mejorar tiempos de procesamiento → tamaño variable

Observen como vienen los datos:

1. Es posible que los datos de partida estén en la forma en que nos piden los resultados→ procesamos directamente el archivo

2. Es posible que los datos de partida NO estén en la forma en que nos piden los resultados→ Debemos cargar en estructuras auxiliares y procesar esta estructura, reordenar los datos del archivo y procesar el archivo

Datos provienen de archivos

1. Tal como están se muestran

Consigna →Mostrar todos los datos del archivo
No requiere guardarlo en otra estructura, leer cada registro y mostrarlo

Dato Archivo ordenado

Consigna →Mostrar todos los datos del archivo ordenados por el mismo campo
No requiere guardarlo en otra estructura, leer cada registro y mostrarlo
Dado un archivo de registros mostrar su contenido por pantalla
Los resultados están en el orden que están en el archivo **No Necesitan estructura auxiliar**

2. Cambiar el orden, u ordenarlo

Consigna es mostrarlo ordenado, si esta desordenado, o el el campo no se corresponde con el resultado
Se requiere una estructura auxiliar memoria para facilitar el procesamiento
Vector → conozca tamaño a priori
Cargo el archivo al vector
Ordeno el vector
Muestro los datos del vector
Lista ordenada → No conozca tamaño a priori
Cargo el archivo a la lista ordenada
Muestro los datos de la lista

3. Ordenar un archivo desordenado

Dado un archivo de registros desordenado mostrar su contenido por pantalla ordenado por el campo1
Dado un archivo de registros desordenado ordenarlo por el campo1
Dado un archivo de registros desordenado mostrar su contenido por pantalla ordenado por el campo1 y a igualdad el campo 1 por el campo 2

Se requiere una estructura auxiliar memoria para facilitar el procesamiento
Vector → conozca tamaño a priori
Cargo el archivo al vector
Ordeno el vector
Me posiciono en la primera posición del archivo
Grabo los datos del vector → archivo se reemplaza y queda ordenado
Lista ordenada → No conozca tamaño a priori
Cargo el archivo a la lista ordenada
Me posiciono en la primera posición del archivo
Grabo los datos del vector → archivo se reemplaza y queda ordenado

---

Buscar en un archivo

1. Archivo ordenada tiene en la clave de búsqueda una PUP

No requiere estructura auxiliar → **búsqueda directa** Pos = Clave-valor 1ra pos
Si decido usarla puedo → vector No Lista→ solo permite secuencial
Clave 101 … 150 → buscar 135 → fseek(f, (135-101)*sizeof(r), SEEK_SET)

2. Archivo ordenado → la posicion no es predecible a priori

Una solución posible → **Busqueda binaria** en el archivo
Otra solución → cargarlo en estructura auxiliar y buscar en ella

3. Archivo desordenado

Necesariamente debemos cargarlo estructura auxiliar
Vector → si se da la condición del tamaño
Búsqueda secuencial
Ordenar el vector → búsqueda binaria → más eficiente que la secuencial
Cada posición solo necesitamos la información
Lista → si no se conoce el tamaño a priori
Búsqueda secuencial
Cada posición necesitamos la información + referencia al siguiente

Dados dos archivos uno con los datos personales de los alumnos, sin orden, el otro ordenado por legajo y materia con las materias cursadas Desarrollar un programa que muestre por pantalla los resultados de las materias y el nombre del estudiante. El campo que vincula ambos archivos es el numero de legajo

---

Datos están en dos estructuras y las quiero agrupar según un determinado orden

1. Dos archivos ordenados → intercalar conservando el orden

Apareo de los archivos sin necesidad de cargarlos en una estructura auxiliar

2. 1 archivo ordenado y el otro no

El ordenado lo dejo como esta
El desordenado lo cargo en una estructura auxiliar → vector ordenado o una lista ordenada
Una solución →apareo archivo ordenado con la estructura auxiliar
Otra solución → ordenar el archivo desordenado→ aparear los archivos

3. Ambos desordenados

Cargar ambos en una estructura auxiliar
Una solución → aparear las estructura auxiliares
Otra solución → ordenar los archivos → aparearlos
Otra solución → si alcanza → cargar ambos archivos misma estructura auxiliar y ordeno

Dados dos archivos ambos ordenados por numero de legajo, uno con los datos personales de los alumnos, el otro ordenado por legajo y materia con las materias cursadas Desarrollar un programa que muestre por pantalla los resultados de las materias y el nombre del estudiante ordenado por numero de legajo. El campo que vincula ambos archivos es el numero de legajo

---

Porque y cuando corte de control

1. Datos en un archivo ordenado por dos campos (legajo y materia) con repetición del legajo

Mostrar los datos agrupados por el primer campo → no necesito estructura auxiliar

nl	mat	nota
4	Algo	1
4	Ssl	2
7	Algo	2
7	Md	10

7	syo	9

Mostrar los datos agrupados por nl → corte de control

NL 4
Mat nota
Algo 1
Ssl 2

Nl 7
Mat nota
Algo 2
Md 10
Syo 9

2. Archivo esta sin orden→ idéntica consigna

Cargarlo estructura auxiliar, ordenada por dos campos
Una solución→ hacer el corte de control en la estructura auxiliar
Otra solución → Ordenar el archivo, y corte de control en el archivo

Dni	Nl	cm	Nota

archivo

Dni	nl	mat	nota
6	14	Alg	2
9	25	Sint	4
3	18	Syo	3
2	30	md	1
5	35	alg	3
14	24	md	1

Son 6 registros

```
struct tr{int dni; int nl; char cm[20]; int nota};
tr vector[6];
tr r;
```

vector

Dni	nl	mat	nota

```
FILE* f = fopen("nombre","rb+");
I = 0;
1 fread(&r,sizeof(tr),1,f);
while(!feof(f)){
 vector[i]= r;
i++;
```

```
 fread(&r,sizeof(r),1,f);
}

2 while(fread(&r,sizeof(r),1,f)){
 vector[i]= r;
i++;
}

3 for(i=0;i<6; i++){
fread(&r,sizeof(r),1,f);
vector[i]= r;
}

4 fread(&r,sizeof(r),1,f);
for(i=0;!feof(f); i++){
vector[i]= r;
fread(&r,sizeof(r),1,f);
}

5 fread(vector,sizeof(r),6,f);→ 5.1 fread(&vector[0],sizeof(r),6,f);


fread(V
```

vector

Dni	nl	mat	nota
6	14	Alg	2
9	25	Sint	4
3	18	Syo	3
2	30	md	1
5	35	Alg	3
14	24	md	1

EN ESTE PUNTO VECTOR Y ARCHIVO DESORDENADO

```
ordenarVector(vector,6);// archivo sin orden, vector ordenado
```

Mostrar por pantalla sin modificar el archivo

```
for(i=0;i<6; i++)
cout <<vector[i].dni<< vector[i].nl……;
```

Reordenar el archivo

```
 fseek(f,0,SEEK_SET);
1 for(i=0;i<6; i++){
r = vector[i];
fwrite(&r,sizeof(r),1,f);
}

2 for(i=0;i<6; i++){
fwrite(&vector[i],sizeof(r),1,f);
}

3 fwrite(vector,sizeof(r),6,f);
```

Archivo ordenado

Volvemos al principio archivo sin orden y no se conoce el tamaño

Estructura adecuada lista

info				Sgte NODO*
Dni	nl	mat	nota	

```
struct Nodo{
        tr info;
        Nodo* sgte;
};
Nodo* Lista = NULL;

while(fread(&r,sizeof(r),1,f))
 insertarOrdenado(lista,r);
```

archivo sin orden. Pero la lista ordenada

mostrar fseek(f,0,SEEK_SET);

```
while(lista!=NULL){     ==
r = pop(lista);         ==
cout<<r.dni…….          fwrite(&r,sizeof(r),1,f)
}
```

Ordenar por dos campos

Ambos acotados → matriz

Dia 1..7	Ciudad 1..100	pasajes

```
int M[7][100];

while( fread(     )){

 M[r.dia – 1][r.ciudad -1] = r. pasajes;

}
```

+Clase 1 → Introducción a punteros, estructuras enlazadas y pilas

Punteros

Conceptos preliminares

Un puntero es un tipo de dato que como valor contiene una dirección de memoria en la que puede alojarse un dato simple una estructura o un determinado código (puntero a una función)

Puntero

Valores → Direcciones de memoria donde un dato o código

Operaciones → asignación →&, new(tipo) NULL

Ejemplo

int a = 10; // declaración de una variable de tipo entero inicializada en 10

int *p=NULL,*p2;// declaración de puntero a

entero podrá tener la dirección de memoria donde se aloja un int

a → 0x123 struct tr{int a; int b};tr r; tr *q = &r; r →=0x456

10	10 15

*p p = new int()

q = &r;

0x123	0x456

Asigna la dirección de a()

p = &a; *p

Nombre	tipo de dato	valor	q=new tr();
a	int	10	delete (p)
p	puntero a int	0x123	
*p	int	10 → valor contenido en el lugar de memoria donde apunta p	
r	tr	10, 15	r.a r.b
q	puntero a tr	0x456	
*q	tr	10, 15	(*q).a q->a (*q).b q->b
(*q).b	int	15	(*q).a
q->b	int ->	15 →operador de acceso a un miembro de struct apuntada por puntero.	

Operadores

& → de dirección → se aplica sobre un identificador

* → de indireccion o desreferenciacion → se aplica sobre un puntero

-> operador de acceso a un campo de un registro apuntado por un puntero equivale a (*puntero).campo

Asignaciones

p = NULL → asigna el valor nulo a un puntero

p2 = p; le asigna a p2 el valor de p, ambos punteros son del mismo tipoy ahora apuntan a la misma dirección

p = q; INCORRECTO los punteros con tipo y p y q apuntan a tipos de datos distintos

*p = q->b le asigna al entero apuntado por p el campo b del registro apuntado por r

p = new int() crea un nuevo espacio de memoria en TE para almacenar un entero → asignación dinámica

q= new tr(); crea un nuevo espacio de memoria en TE para almacenar un tr → asignación dinámica

Los espacios de memoria creados con new son INSTANCIAS de los punteros, y se referencian mediante el mismo. Al no tener nombre propio se los llama variables anónimas.

Los punteros son variables estáticas, las instancias son dinámicas, se pueden crear y liberar en TE

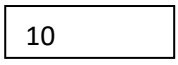
delete (p); libera la instancia a la que apunta p, similar seria delete(q);

(*q).c1 q->c1

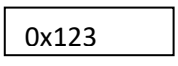
Prioridad asociatividad operadores

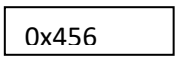
PRIORIDAD AS.	**OPERADOR/ES**
G1 S/A	::
G2 I→D	. -> [ ] ( ) ++ -- posfijo
G3 D→I	sizeof ++ -- prefijo ! + - unario &direccion * indirección
G4 I→D	* / %
G5 I→D	+ - suma resta

int a = 10, v[4] = {6,7,8,9}, v1[4],*p1,*p2;
a 0x123 *(p2+2) v[2] *(v+2) *(p2+2) v[2] p2++



p2 = &v[0], o p2 = v





p1 = & a;
0XABC

?			

V1 = V → INCORRECTA PORQUE TRATA DE ASIGNAR DIRECCIONES Y EN LOS VECTORES SON CONSTANTES

0x456

v[0] 6	**v[1] 7**	**v[2] 8**	**v[3] 9**

v equivalente a &v[0]
(*p1)++ *p1 *(v+2) v[2]
*p2 6 p2++ avanza el puntero v1++ produce error porque v es puntero constante
*(p2+2) = 8 = v[2] p2[2]
*(2+p2) = 8 = 2[v]
p2++;
*p2, *p2++, (*p2)++; ++*p2; ++(*p2)

int a, v[10]

Suponiendo las siguientes declaraciones con las correspondientes definiciones

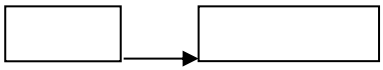
struct tr{int a, int b}; tr r;

int cantRegistros(FILE *);

void ordenarVector(tr [], int);

Interrogantes

int *p = new int(); genera en tiempo de ejecución lo que sigue?

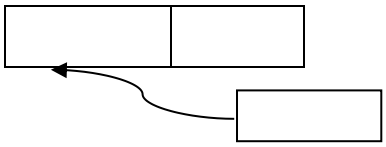


p *p

p = new int[10] es correcto? de ser asi se puede acceder a cada miembro? justifique

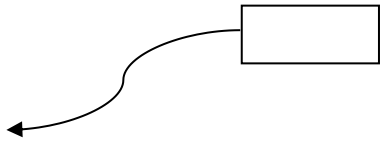
?									

tr *q = new tr(); como se accede a la estructura, y a cada miembro? tr v4[4]



q

q = new tr[4]; es correcto, genera lo que sigue? se puede acceder a la estructura o a cada miembro? como?



con un flujo de registros como el descripto, abierto para lectura, es posible la siguiente secuencia de código?

```
int n = cantRegistros(f);

q = new tr[n];

fread(q, sizeof (tr), n,f);

ordenarVector(q, n);

mostrarVector( tr q);
```

```
struct articulo{
        char s[10];
        int precio;
};

struct vendedor{
        char nombre[20];
        articulo productos[10]
};

vendedor vendedores[30];

vendedor matrizComplicada[5][10][10];
```

vendedor * punteroDificil = new vendedor[5][10][10]; es posible? como se accede a cada miembro? se puede hacer lo mismo que con la estructura anterior? cual es, si la hay, la diferencia conceptual?

Punteros a funciones

Los punteros contienen direcciones de memoria, las que pueden ser a un tipo de dato, o a código, en este caso se los denomina punteros a funciones

```
int  *fptr (int);   // E.2
```

es el prototipo de una función que recibe un int y devuelve puntero-a-int

```
int (*fptr)(int);   // E.1
```

es la declaración de un puntero-a-función que recibe un int y devuelve un int.

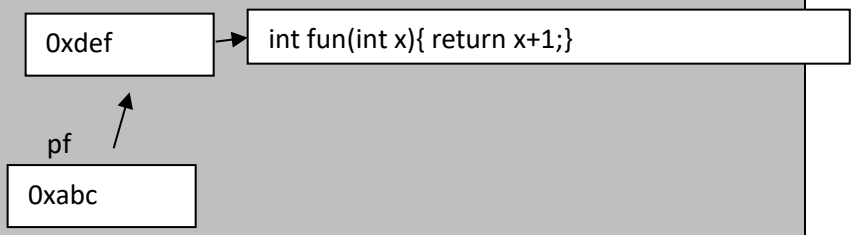
```
void func(int (*fptr)(int));   // E.3: Ok.
```

La expresión E.3 es el prototipo de una función que no devuelve ningún valor y recibe como argumento un puntero-a-función.

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

int* fun0(int i) {           //
 cout << i << endl;
 return &i;}
//recibe un int y retorna un puntero a int

int fun1(int i) {            //
 cout << i << endl;
 return 10*i;} // recibe un int y retorna un int

void fun3(int (*pf)(int)) { pf(20); }      // int* pf es distinto a int (*pf)
```

Función que retorna valor ausente, recibe un puntero-a-función que evalua un int y devuelve un int. Ejecutan la función señalada por su argumento

```
int main(void) {
 int x = 10;
 int y = fun1(x);            //asigna a y el resultado de fun1
 fun1(fun1(y));              // invoca fun1 recursivamente
 int (*pf1)(int) = &fun1;     // pf1 es el puntero a la función fun1
 pf1(x);                     // se invoca a la fun1 mediante el puntero
 fun3(pf1);                  // se invoca fun3 con el puntero pf1 que ejecuta fun1
 int* (*pf2)(int) = &fun0;    // pf2 es puntero-a-función que recibe un int y devuelve un
puntero-a-int.  Lo iniciamos con la dirección de la función fun0
 pf2(y);                  // Invocamos fun0 utilizando su puntero


int fun(int x){return x+1;}              fun → 0xabc      0xdef
```




```
int (*pf)(int) = &fun;
```

Sintesis

Puntero es un tipo de dato (tipado) que apunta a direcciones de memoria en la que se aloja dato o código

Apuntar, metafóricamente, significa que contiene la dirección de memoria en la que puede haber un dato o una porción de código (puntero a función)

Se puede asignar dirección de memoria de datos estáticos o, mediante new esta asignación puede ser dinámica.

Asignacion dinámica crea instancias que son, justamente, dinámicas, es decir direcciones que pueden crearse o liberarse en tiempo de ejecución.

Estructuras enlazadas

Una estructura de tipo vector, si se considera que el primer elemento lógico coincide con el primer elemento físico, por ejemplo, en v[o], y el siguiente elemento lógico es el siguiente físico a v[i] le sigue v[i+1] es una estructura secuencial. pero para recorrer un vector, por ejemplo, no es imprescindible recorrerlo secuencialmente.

2	
3	
5	
1	
4	

esta estructura puede ser mostrada ordenada, por ejemplo, sin necesidad de ordenarla.

Si se tiene referencia al primer elemento lógico índice 3 en este caso se muestra 1. Si luego se accede a la posición indicada por el otro campo, cero en este caso se puede acceder al siguiente elemento lógico. Esto es una estructura enlazada. Se

1
2
3
4
5

2	**1**
3	4
5	-1
1	0
4	2

POSICION DEL PRIMERO 3

Nombre	Cargan	sacan
pilas	Delante del primero	De la primera posición
colas	Después del ultimo Prioridad	Primera posición Según la prioridad
Listas simplemente enlazadas Listas circulares Listas doblemente enlazadas Listas doblemente enlazadas	Según criterio	Según ese criterio

Las estructuras enlazadas pueden ser lineales, vinculan un elemento con uno, pueden ser arbóreas, vinculan un elemento con mas de uno, estos son los arboles, o pueden vincularse entre varios estos son grafos

en algoritmos veremos estructuras lineales de según como se carguen los elementos y como se los obtengan pueden ser pilas, colas o listas.

Diferentes estructuras lineales

Pilas: el nuevo elemento se agrega delante del primero, en el momento de sacar se extrae el que esta en la primera posición (LIFO) → pila de platos →recursividad

Colas: el nuevo elemento se agrega despues del ultimo, en el momento de sacar se extrae el que esta en la primera posición (FIFO) → cola de colectivo utilización de un recurso

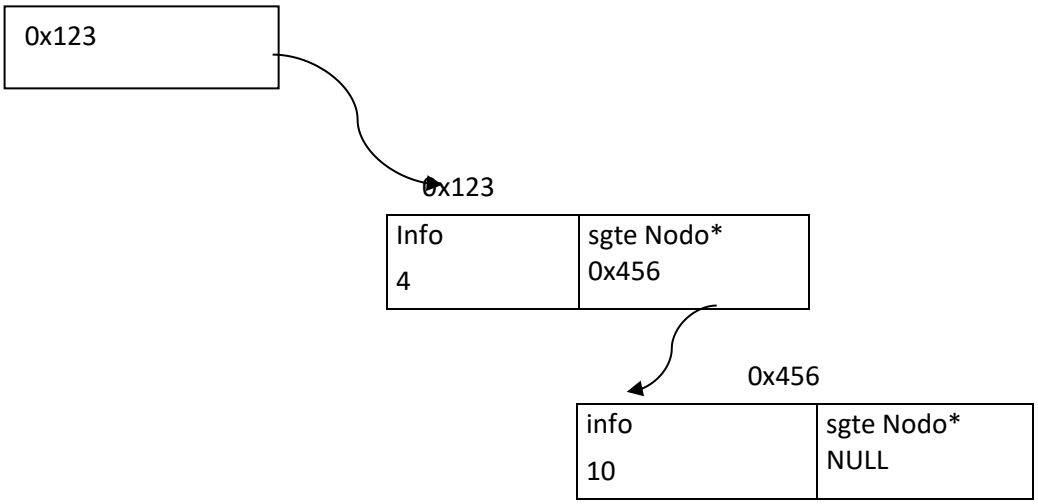
Listas: el nuevo elemento se agrega siguiendo un determinado criterio, en general de orden. Para sacar se lo hace buscando el elemento y se lo extrae del lugar donde esta, si es que esta. esta estructura a diferencia de las otras dos se la puede recorrer.

Las pilas y colas solo pueden ser accedidas par poner un nuevo dato o sacarlo, esto de los lugares fijos donde es posible hacerlo.

las listas pueden ser: Simplemente enlazadas, doblemente enlazadas, circulares, circulares doblemente enlazadas. En AyED profundizaremos el estudio de las simple y doblemente enlazadas.

Las estructuras enlazadas se pueden implementar con vectores como hemos visto, pero esto no resolvería el problema del tamaño fijo en tiempo de ejecución. Es por ello que las implementaremos con punteros y asignación dinámica en memoria para ir generando los registros (al que llamaremos NODO) que vayamos necesitando en tiempo de ejecución. Estos nodos generaran lo que se conoce como estructuras autoreferenciadas. Tendran un campo con la información y otro que indicara la siguiente dirección de memoria donde habrá un nodo del mismo tipo o la constante NULL que indicara donde finaliza la estructura. Veamos

PUNTERO CONTROL DE LA ESTRUCTURA Nodo *

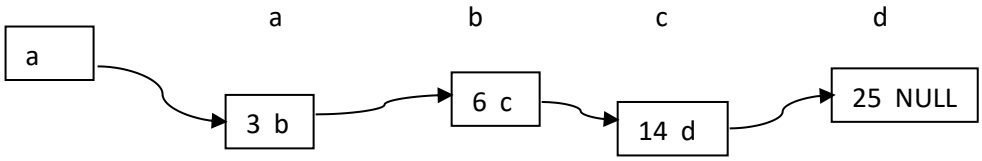



```
struct Nodo{
int info;
Nodo* sgte;
}
Nodo *pila;
```

int v[4]

3
6
14
25

Lista



Nombre	Tipo	valor
v	vector	3, 6, 14, 25
lista	puntero	Direccion de memoria a
v[0]	int	3
lista→info	int	3
lista→sgte	puntero	Direccion de memoria b
v[1]	int	6
lista→sgte→info	int	6
lista→sgte→sgte	puntero	Direccion de memoria C
v[2]	int	14
lista→sgte→sgte→info	int	14
lista→sgte→sgte→sgte	puntero	Direccion de memoria D
v[3]	int	25
lista→sgte→sgte→sgte→info	int	25

Pasos para la creación y carga de estas estructuras:

1. Definir la estructura autorreferenciada→ el registro que llamaremos NODO

Este tendrá dos campos: Uno con la información y el otro con un puntero a una estructura momo la que se esta definiendo.

Ejemplo para una estructura de enteros → **struct Nodo{ int info; Nodo* sgte};**

2. declarar un identificador que contenga la dirección de memoria del inicio de la estructura, un puntero

Nodo* p = NULL; //p tendrá la referencia a la dirección de memoria del primer elemento, para crearlo es necesario hacerlos apuntar a NULL
3. tomar el dato para cargar en la estructura
4. pedir memoria
5. guardar el nuevo dato en la instancia creada
6 actualizar los punteros

Los pasos para todas las estructuras son los mismos, solo se diferencian en la forma en que se actualizan los punteros.

Se necesita un puntero para conocer donde comienza la estructura. El resto de los nodos se van vinculando con el campo siguiente de cada nodo intermedio. El valor del campo siguiente del ultimo nodo es NULL lo que significa que alli termino la estructura. Si el puntero de control de la estructura, cualquiera sea esta apunta a NULL significa que la estructura esta vacia

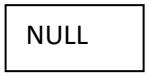
Cargar nodos en una pila

definir el Nodo

struct Nodo{ int info, Nodo* sgte}; *// crear el nodo con los dos campos*

crear la estructura

Nodo* Pila = NULL; //declarar el puntero que apuntara al inicio de la estructura. Se debe inicializar con NULL



Pila

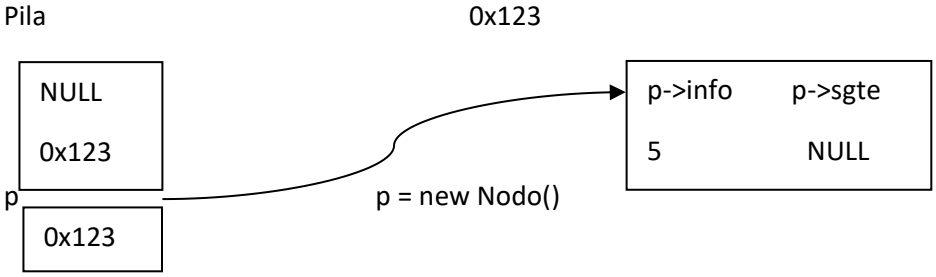
En esta estructura pueden darse dos situaciones diferentes

1 Que la pila este vacia, es decir que apunte a NULL. Al insertar un nodo este va a ser el primero y único

2 que la pila ya tenga nodos por lo que el que se genera va a ser el rimero pero no será el único

Insertar valor 5 en una Pila de enteros que esta vacia

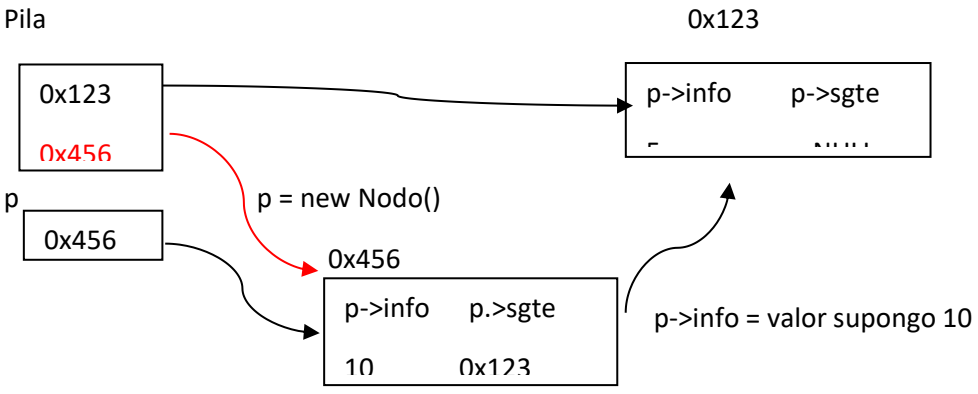
Pila si la pila esta vacia el puntero apunta a NULL



Nodo *p = new Nodo(), el nodo tiene dos campos p->info= 5; p->sgte = NULL/Pila

Ahora que se agrego el nodo la pila deja de estar vacia, el nodo creado es el primero por lo que el puntero Pila debe apuntar a esa instancia a la que apunta p, entonces Pila = p

Si la pila ya tiene nodos el procedimiento es similar



p->sgte = pila, el ex primer nodo

y ahora el puntero a pila debe apuntar a p ya que pasa a ser el primero

Ambas son idénticas excepto el la línea que se asigna p->sgte pero asignar NULL o pila cuando se inserta el primer nodo tienen idéntica semantica

```
void push(Nodo* & Pila, int x){          se invoca con la pila, que siempre se modifica, y el valor a guardar x
 Nodo*p = new Nodo();                    pedir memoria
  p→info = x;                            guardar la informacion
  p→sgte =Pila;                          enlazar el nuevo nodo con la estructura existente delante de Pila
  Pila = p;                              el nodo creado pasa a ser el primero
 return;
}
```

Esto vale para las dos situaciones, la sentencia p→sgte = pila; hace que el nodo siguiente al creado sea el que estaba en primer lugar, el apuntado por pila. si la pila esta vacia su valor es nul, por lo que asignaría NULL al campo siguiente del nodo creado indicando con esto que la estructura finaliza en ese nodo.

En el caso de necesitar eliminar un nodo de la pila y retorna el valor allí guardado la secuencia será:

```
int pop(Nodo* &pila){   recibe la pila como parámetro y retorna el valor contenido en el primer nodo
  int x;                el tipo de dato de la información para retornarlo
  Nodo* p = Pila;       un puntero al comienzo de la estructura para luego eliminar ese nodo
  x = pila→info;        conservar el valor del primer nodo para retornarlo
  pila = p→sgte;        Avanzar con la pila un nodo para eliminar el nodo que estaba en el tope
  delete p;             eliminar el que era primero, la pila esta apuntando al que era segundo o a NULL
  return x;             retornar la información que estaba en el primer nodo
}
```

Tenga en cuenta que para utilizar esta función la pila debe tener al menos un nodo. La pila solo permite push o pop, no admite recorrido. En push y en pop el puntero a pila SIEMPRE cambia. La pila invierte el orden

1
2
3
4
5

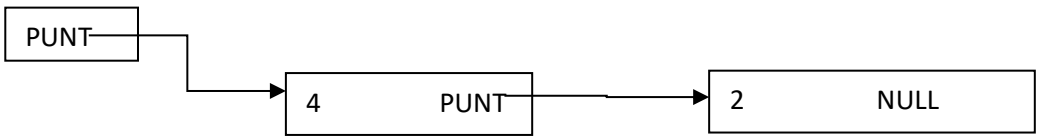
Secuencial

- **1er elemento lógico → primero físico**
- **sgte elemento lógico → siguiente físico**

0	2	1
1	3	4
2	5	-1
3	1	0
4	4	2

Enlazadas

- **posición 1er elemento logico**
- **c/elemento registro → nodo → info, siguente elemento logico**



Dado una pila de enteros desordenda, ordenarla utilizando push y pop

6
9
3
14
2

2
3
6
9
|4

Repasamos y avanzamos con colas
Puntero

Tipo de dato que contiene dirección de memoria a un dato o a un segmento de codigo

Operadores de dirección & indireccion * y →

```
int a = 10;
int* p = &a;
cout << *p << endl;

tr*q=new tr();
*q refiere al registro apuntado por q
(*q).c1 o su equivalente q→c1 acceso al camp1 del registro apuntadp
por q
```

Asignación dinámica de memoria

```
int* p = new int();

*p = 10;

delete p;
```

Estructuras autoreferenciadas

Las estructuras enlazadas tienen, para cada elemento, una struct, que se llama nodo con un campo con la información y el otro con una referencia a una estructura del mismo tipo. En el caso de la implementación con asignación dinámica en memoria un campo con la información y otro campo, un punyero, con la dirección de memoria del siguiente nod, o el valor NUU en caso de no haber siguiente.

```
struct Nodo
{
   int info;  // valor que contiene el nodo
   Nodo* sig; // puntero al siguiente nodo
};
```

Punteros a estructuras: operadores de acceso a un miembro

```
Nodo* p = new(Nodo);

(*p). info; equivale a p->info

Lo dicho vale para el campo sig
```

Punteros a vectores dinámicos

```
tr* q = new tr[10]
```

Punteros y funciones: Punteros a funciones

int *fptr (int); // E.2
es el prototipo de una función que recibe un int y devuelve puntero-a-int
int (*fptr)(int); // E.1
es la declaración de un puntero-a-función que recibe un int y devuelve un int.
void func(int (*fptr)(int)); // E.3: Ok.

La expresión E.3 es el prototipo de una función que no devuelve ningún valor y recibe como argumento un puntero-a-función.

Estructuras enlazadas

A diferencia de las estructuras secuenciales en las que el primer elemento lógico coincide con el primero físico y el siguiente lógico con el siguiente físico, por ejemplo si se quiere recorrer un vector secuencialmente, una estructura enlazada es una estructura tal que en cada posición tiene un registro al que se llama NODO, con al menos dos campos → uno con la información y el otro con la referencia al siguiente nodo. Estas estructuras además del nodo con esas características deben tener la referencia al primer elemento lógico qu puede o no ser el primero físico. Se pueden implementar con vextores pero lo implementaremos con asignación dinámica en memoria.

Los pasos son:

1. crear el nodo
2. declarar un puntero para controlar la estructura.
3. tomar la información
4. pedir memoria
5. guardar la información
6. actualizar los punteros

Estas estructuras pueden ser lineales, arbóreas o grafos, en EyED abordamos lineales que a su vez pueden ser Pilas, Colas y Listas, se diferencian, además del uso conceptual, por la forma en que se carga un nuevo dato o se extrae uno en particular.

Pilas → con un dato simple en el campo info

struct Nodo{ int info, Nodo* sgte}; →crear el nodo

Nodo* Pila = NULL; → "inicializar la estructura" hacer apuntar a NULL el puntero de control

```
void push(Nodo* &Pila, int x){
   Nodo * p = new Nodo();
   p→info = x;
   p→sgte = Pila;
   Pila = p;
   return;
}

int pop(Nodo* & Pila){
   int x;
   Nodo* p = Pila;
   x = p→info;
   Pila = p→sgte;
   detete p;
```

```
  return;
}
```

Pila → con una struct en el campo info

Crear la struct para la información: **struct tipoInfo{int c1, int c2}**

crear el nodo con una struct tipoInfo en el campo de la información: **struct Nodo{tipoInfo info; Nodo* sgte};**

los procedimientos push y pop solo difieren en el tipo de dato:

1. **push** solo en el prototipo cambiando el tipo de dato de x a tipoInfo
2. **pop** en el valor de retorno de la función debe retornar tipoInfo en lugar de int y, por supuesto, "x" debe ser de tipoInfo.

Esto requeriría desarrollar estos procedimientos para cada tipo de dato que querramos utilizar salvo que se desarrollen con plantillas utilizando plantillas o templates en C++

Para generalizar el tipo de dato con plantillas se requiere:

Nodo

```
template <typename T> struct Nodo
{
  T info;      // valor que contiene el nodo
  Nodo<T>* sig; // puntero al siguiente nodo
};
```

push

```
template <typename T> void push(Nodo<T>*& Pila, T x)
{
  Nodo<T>* p = new Nodo<T>();
  p ->info = x;
  p->sgte = Pila;
  Pila = p;
  return;
}
```

pop

```
template <typename T> T pop(Nodo<T>*&)
{
  T x;
  Nodo<T>* p = Pila;
  x = p ->info;
  Pila = p->sgte;
  delete p;
```

```
  return x;
}
```

Estructura tipo cola

Esta estructura desde el punto de vista conceptual esta compuesta por un conjunto de nodos. Difiere de la pila en la forma en que se agrega un nuevo elemento o se extrae uno. El agregado se hace después del ultimo elemento, la razón de este nombre de la estructura es justamente por eso, como cualquier cola en cualquier contexto lo razonable es que el ingreso sea ubicándose después del ultimo. para extraer un elemento, siguiendo el criterio también de las colas se hace desde el tope, es decir el primero que se libera es el primero que ingreso por eso se las identifica como FIFO por las siglas en ingles. a los efectos de la implementación se dividirá en las dos situaciones diferentes que pueden presentarse para la carga, uniéndose luego en una función que contemple ambos casos. Lo mismo será tara eliminar nodos. La estructura pila requiere un puntero al principio ya que desde allí se inserta un elemento nuevo o se extrae uno. en el caso de la cola como se inserta después del ultimo es conveniente tener un puntero también a esta dirección.

Implementacion de una cola con datos simples

Declaracion del Nodo → sin diferencias a lo realizado para pilas

```
struct Nodo{
   int x;
   Nodo* sgte};
}
```

definicion de los punteros que controlan la estructura

Nodo* frente = NULL; Puntero al comienzo de la estruntura inicializada en NULL

Nodo* fin = NULL; Puntero al final de la estructura también inicializado en NULL

El procedimiento para agregar un dato en una cola lo llamaremos **queue** por su nombre en ingles. Para extraer un nodo lo llamaremos unqueue.

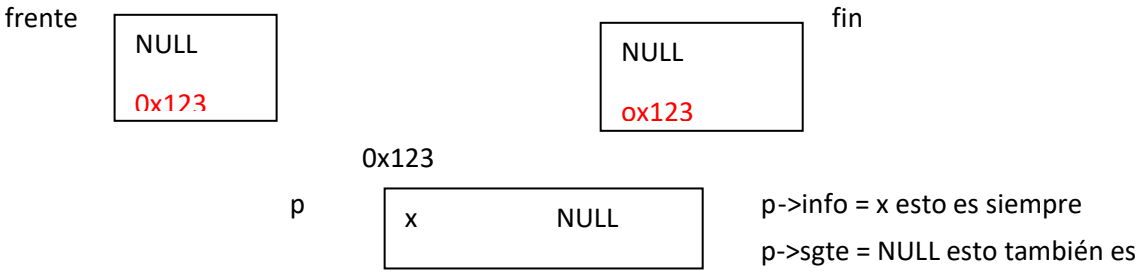
Nodo* frente=NULL

Nodo * fin = NULL

Agregar a una cola vacia donde frente y fin apuntan a NULL

```
   Nodo * p = new Nodo();
   p→info = x;
   p→sgte = NULL; como p va a ser el ultimo nodo no tendra siguiente por lo que apunta a NULL
   frente = p; como no había nodos el frente debe apuntar a este recine creado
   fin = p; el puntero al fin siempre apuntara a este nodo ya que fin apunta, al final que es donde se agrega
```

en este caso se modifican el inicio y el final, no había nodos, ahora hay uno que es primero y ultimo



siempre

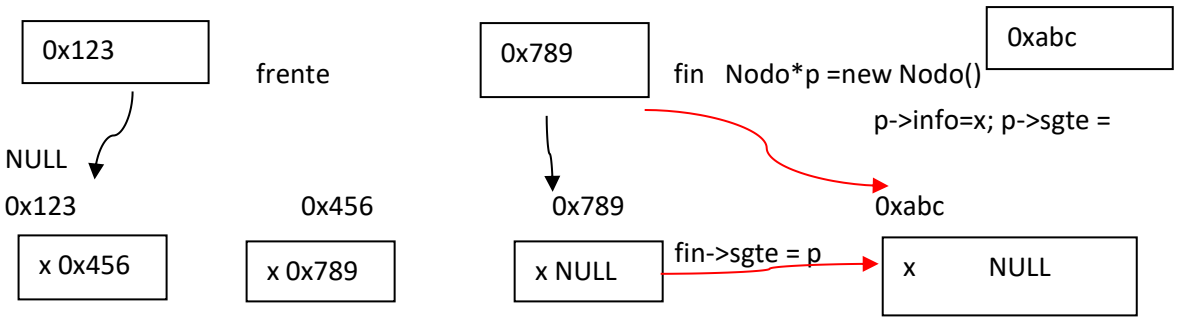
Nodo * p = new Nodo()

ahora el frente debe apuntar al nodo creado p. Solo en esta situación se modifica el frente

frente = p;

el puntero fin debe apuntar al nodo creado, esto ocurre siempre que se agrega un nodo pues se agrega al final

De este modo queda actualizada la cola con el primer nodo



Agregar a una cola con datos donde frente apunta al primer nodo y fin al ultimo

```
Nodo * p = new Nodo();
p→info = x;
p→sgte = NULL;
fin→sgte = p; se enlaza el nuevo nodo a continuación del que era el ultimo
fin = p; se redefine el nuevo fin apuntando a este nuevo nodo
```

Las cargas según lo visto solo cambian en la línea 4, en caso que la cola estuviera vacia y frente apuntara a NULL ahora, con un nodo debería apuntar a el. En el segundo caso el frente no cambia pero se hace necesario enlazar el nuevo nodo con el ex ultimo nodo que estaba apuntado por fin. Luego de esto debe cambiarse la dirección a la que este puntero apunta para que lo siga haciendo al ultimo

con dato de tipo simple

```
void queue(Nodo*&frente, Nodo* &fin, int X){
   Nodo * p = new Nodo();
   p→info = x;
```

```
   p→sgte = NULL;
   if (frente==NULL) frente = p; else fin→sgte = p;
   fin = p; se redefine el nuevo fin apuntando a este nuevo nodo
   return;
}
```

con plantilla

```
Nodo
template <typename T> struct Nodo {
  T info;      // valor que contiene el nodo
  Nodo<T>* sig; // puntero al siguiente nodo
};

template <typename T> void queue(Nodo<T>*& frente, Nodo<T>*&fin, T x)
{
  Nodo<T>* p = new Nodo<T>();
  p ->info = x;
  p->sgte = NULL;
  if (frente==NULL) frente = p; else fin→sgte = p;
  return;
}
```

Un análisis similar se puede hacer para eliminar nodos de una cola, la eliminación es similar al procediento pop de una pila, con la iferencia que al eliminar el ultimo nodo se debe hacer apuntar a NULL sl pintero al final

con dato de tipo simple

```
int unqueue(Nodo*&frente, Nodo* &fin){
   int x;
   Nodo * p = frente;
   x = p→info;
   frente = p→sgte;
   if (frente==NULL) fin = NULL;
   delete p;
   return x;
}
```

con plantilla

Nodo

```
template <typename T> struct Nodo {
  T info;      // valor que contiene el nodo
  Nodo<T>* sig; // puntero al siguiente nodo
};

template <typename T> T unqueue(Nodo<T>*& frente, Nodo<T>*&fin)
{
  T x;
  Nodo<T>* p = frente;
  x = p ->info;
  frente = p->sgte;
  if (fte==NULL) fin = NULL;
  delete p;
  return x;
}
```

```
Ejemplos de uso con datos simples
int main()
{
   Nodo<int>* pila = NULL;
   Nodo<int>* frente = NULL;
   Nodo<int>* fin = NULL;

   push<int>(pila,10);
   push<int>(pila,20);
   push<int>(pila,30);
   while( pila!=NULL )cout << pop<int>(pila) << endl;

   queue<int>(frente, fin ,20);
   queue<int>(frente, fin ,30);
   while( frente!=ULL )cout << unqueue<int>(frente, fin) << endl;

   return 0;
}



Ejemplo con estructuras
struct Alumno
{
   int legajo;
   string nombre;
};

// función auxiliar para cargar los datos de la struct
Alumno cargarRegistro(int leg, string nom)
{
   Alumno a;
   a.leg = leg;
   a.nom = nom;
   return a;
}
```

```
int main()
{
   Nodo<Alumno>* pila = NULL;
   Alumno a = cargarRegistro(1,"Roxana");
   push<Alumno>(pila a;

   Alumno a = cargarRegistro(2,"Oscar");
   push<Alumno>(pila a;


   while( pila )
   {
      a = pop<Alumno>(pila);
      cout << a.leg << ", " << a.nom << endl;
   }

   return 0;
}
```

Clase 3 → listas ordenadas simplemente enlazadas

Como hemos visto las estructuras enlazadas pueden ser lineales (pilas, colas, listas) orboreas o grafos.

En las estructuras lineales lo que las diferencia es donde se agrega un dato nuevo y/o de qué lugar se lo extrae. en el caso de las pilas y las colas tienen lugares específicos para ello: En pilas se agrega delante del primer nodo (la metáfora de la pila de platos), en las colas detrás del ultimo (metáfora de cola en los colectivos), en ambas estructuras también se extraen los datos de un lugar determinado: el tope.

Las listas no tienen esta restricción; se puede insertar un valor delante del primero, al final de la estructura o en medio de dos valores particulares. En general se las inserta siguiendo un determinado criterio de ordenamiento. Esta estructura puede ser recorrida con propósitos diversos, se puede modificar un dato particular, se puede eliminar un nodo que este al principio, o al final, o en medio de dos particulares.

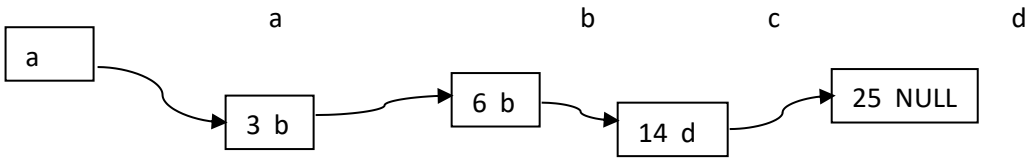
Esta estructura esta en memoria, pero los datos no necesariamente están en posiciones contiguas como ocurre con los vectores.

Veamos

int v[4]

3
6
14
25

Lista



Nombre	Tipo	valor
v	vector	3, 6, 14, 25
lista	puntero	Direccion de memoria a
v[0]	int	3
lista→info	int	3
lista→sgte	puntero	Direccion de memoria b
v[1]	int	6
lista→sgte→info	int	6
lista→sgte→sgte	puntero	Direccion de memoria C
v[2]	int	14
lista→sgte→sgte→info	int	14
lista→sgte→sgte→sgte	puntero	Direccion de memoria D
v[3]	int	25
lista→sgte→sgte→sgte→info	int	25

Abordaremos distintos patrones para insertar nodos en una lista, supondremos ordenadas en forma creciente. Abordaremos la implementación con datos simples y luego con plantillas Los patrones a abordar serán:

- ✓ Insertar el primer nodo de una lista o delante del primero
- ✓ insertar después del ultimo
- ✓ Insertar en una posición determinada
- ✓ Insertar ordenado → donde corresponda según el criterio de ordenamiento
- ✓ buscar un valor en la lista
- ✓ insertar ordenado sin repetir la clave
- ✓ cambiar el criterio de ordenamiento
- ✓ Eliminar nodos
  - el primero
  - el ultimo
  - uno determinado según su valor
  - todos los nodos de la estructura
- ✓ Mostrar todos los valores de la lista
  - eliminando el mostrado
  - sin eliminar el mostrado

Con datos simples	Con plantillas y/o estructuras

Insertar delante del primero: similar a push retorna la dirección del nodo creado
 	
Nodo * **insertaPrimero**(int*& l, int x)
{
int* p = new Nodoint();
p ->info = x;
p->sgte = l;
l = p;
return; p
}
	Nodo
	template <typename T> struct Nodo{
	T info; // valor que contiene el nodo
	Nodo<T>* sig; // puntero al siguiente nodo
	};
	
	template <typename T>
	Nodo<T>* **insertaPrimero**(Nodo<T>*& l, T x)
	{
	Nodo<T>* p = new Nodo<T>();
	p ->info = x;
	p->sgte = l;
	l = p;
	return; p
	}
	
	Ejemplo de uso
	struct tr{int c1; int c2};
	Nodo<int>* l1 = NULL;
	Nodo<tr>* l2 = NULL;
	tr r; r.c1 = 2; r.c2 = 15;
	insertaPrimero<int>(l1, 25);
	insertaPrimero<tr>(ls, a);

Agregar al final: Se debe recorrer la lista hasta llegar a apuntar al ultimo nodo, de este se debe apuntar al nuevo Nodo* **insertarAlFinal**(int*& l, int x){	template<typename T>

```
  Nodo* nuevo = new Nodo();
  nuevo->info = x;
  nuevo->sig = NULL;

  if( l==NULL )  {
    l = nuevo; }
  else {
    Nodo* aux = l;
    while(aux->sig!=NULL )
      aux = aux->sig;

    aux->sig = nuevo;
  }
return nuevo;
}
```

```
Nodo<T>* insertarAlFinal(Nodo<T>*& l, T
x){
  Nodo<T>* nuevo = new Nodo<T>();
  nuevo->info = x;
  nuevo->sig = NULL;

  if( l==NULL )   {
    l = nuevo;   }
  else    {
    Nodo* aux = l;
    while(aux->sig!=NULL )
      aux = aux->sig;

    aux->sig = nuevo;
  }
return nuevo;
}
```

Buscar: si un valor determinado se encuentra en la lista retorna el puntero a ese nodo; de lo contrario retorna NULL

Version 1: se supone No Ordenada y dato simple

```
Nodo* buscar(Nodo* l, int v){
  Nodo* aux = l;

while( aux!=NULL && aux->info!=v )   {
    aux=aux->sig;
  }
  return aux;
}
```

Versión 2 si esta ordenada puede salir antes del final

```
Nodo* buscar(Nodo* l, int v){
  Nodo* aux = l;

while( aux!=NULL && aux->info!=v )   {
    aux=aux->sig;
  }
  return (uux!=NULL&&aux-
>info==v)?aux:NULL);
}
```

Ejemplo completo de busqueda

```
int main(){
  Nodo<Alumno>* p = NULL;

agregar<Alumno>(p,CargarRegistro(1,"Roxana
"));
```

```
template <typename T>
Nodo<T>* buscar(Nodo<T>*l, T v){
Nodo<T>* aux = l;

while( aux!=NULL && aux->info!=v )   {
    aux=aux->sig;
  }
  return aux;
}
```

```
template <typename T>
Nodo<T>* buscar(Nodo<T>*l, T v){
Nodo<T>* aux = l;

while( aux!=NULL && aux->info!=v )   {
    aux=aux->sig;
  }
return (uux!=NULL&&aux-
>info==v)?aux:NULL);
}
```

Version 3 con un criterio diferente de ordenamiento utilizando punteros a funciones

```
template <typename T, typename K>
Nodo<T>*buscar(Nodo<T>*l, K v, int
(*criterio)(T,K) )
{
  Nodo<T>* aux = p;
```

```
agregar<Alumno>(p,cargarRegistro(2,"Oscar")
);

  int leg;
  cout << "Ingrese el legajo a buscar: ";
  cin >> leg;

   // busca por legajo
  Nodo<Alumno>* r =

buscar<Alumno,int>(p,leg,criterioAlumnoLeg);
  if( r!=NULL )
    cout << r->info.leg << ", " << r->info.nom
<< endl;

  string nom;
  cout << "Ingrese el nombre a buscar: ";
  cin >> nom;

   // busc por nombre
  r =

buscar<Alumno,string>(p,nom,criterioAlumno
Nom);

  if( r!=NULL )
    cout << r->info.leg << ", " << r->info.nom
<< endl;

  return 0;
}
```

```
  while( aux!=NULL && criterio(aux-
>info,v)!=0 )   {
    aux = aux->sig;
  }
  return aux;
}
```

Ejemplo de uso

// funciones con distintos criterios de busqueda

```
int criterioAlumnoLeg(Alumno a,int leg){
  return a.leg-leg;
}

int criterioAlumnoNom(Alumno a,string
nom){
  return strcmp(a.nom.c_str(),nom);
}
```

La invocación

Se pasa la lista, el valor buscado y el puntero a la función que lo evaluara

```
Nodo<Alumno>* r =

buscar<Alumno,int>(l,leg,criterioAlumnoLe
g);

Nodo<Alumno>* r =

buscar<Alumno,int>(l,nom,criterioAlumno
Nom);
```

Mostrar el contenido de una lista,

Version 1 conservando los valores

```
void mostrar(Nodo* p)
{
  Nodo* aux = p;
  while( aux!=NULL )
  {
    cout << aux->info << endl;
    aux = aux->sig;
  }
}
```

Version 2 Eliminando el nodo luego de mostrar

```
void mostrar(Nodo* & l) {

 while( l )
    cout << pop(l) << endl;
```

return;
}
Liberar la memoria de todos los nodos de una lista
Version 1 avanzando el puntero manualmente
void **suprimir**(Nodo*& l)
{
Nodo* ant;
while(l!=NULL) {
ant = l;
l = l->sig;
delete ant;
}
return
}
Version 2 usando funciones ya desarrolladas
void **liberar**(Nodo*& p){
while(l)
pop(l);
return;
}	

eliminarNodo elimina un nodo con un valor dado si lo encuentra
void **eliminarNodo**(Nodo*& l, int v){
Nodo* actual = l;
Nodo* ant = NULL;
 	
while(actual!=NULL && aux->info!=v) {
ant = actual;
actual = actual->sig;
}
If(actual->info!=v return; //no lo encontro
if(ant!=NULL)//si no es el primer nodo
ant->sig = actual->sig;
else//si se elimina el primero
l = actual->sig;//actualiza puntero al inicio
 	
delete actual;
return;
}
	template <typename T, typename K>
	void eliminar(Nodo<T>*& p, K v, int (*criterio)(T,K)){
	Nodo<T>* actual = l;
	Nodo<T>* ant = NULL;
	
	while(actual!=NULL && criterio(aux->info,v)!=0) {
	ant = aux;
	aux = aux->sig;
	}
	If(actual->info!=v return;
	if(ant!=NULL)
	ant->sig = aux->sig;
	else
	p = aux->sig;
	
	delete aux;
	return;
	}

Insertar ordenado
Nodo* insertarOrdenado(Nodo*& l, int v){
 	
Nodo* nuevo = new Nodo();
nuevo->info = v;
nuevo->sig = NULL;
 	
Nodo* ant = NULL;
Nodo* actual = l;
	template <typename T>
	Nodo<T>* insertarOrdenado(Nodo<T>*& p, T v, int (*criterio)(T,T)){
	Nodo<T>* nuevo = new Nodo<T>();
	nuevo->info = v;
	nuevo->sig = NULL;
	
	Nodo<T>* ant = NULL;
	Nodo<T>* actual = p;

```
  while( actual!=NULL && aux->info<v ) {
    ant = actual;
    actual = actual->sig;
  }

  if( ant==NULL )
     l = nuevo;
  else
    ant->sig = nuevo;

  nuevo->sig = actual;

  return nuevo;
}
```

```
 while( aux!=NULL && criterio(aux->info,v)<=0 ) {
    ant = actual;
    actual = actual->sig;
  }

  if( ant==NULL )
    l = nuevo;
  else
    ant->sig = nuevo;

  nuevo->sig = aux;

  return nuevo;
}
```

Insertar sin repetir la clave: Inserta solo en caso que el valor a agregar no se encuentre en la lista retorna
(encontró el valor? puntero a ese nodo:puntero al nuevo que inserta)

```
Nodo* cargarSinRepetir(Nodo*& l,tr v,
int (*criterio)(T,T)){
  Nodo* x = buscar(l,v,);

  if( x == NULL )
  {
    x = insertarOrdenado(p,v,);
  }

  return x;
}
```

```
template <typename T>
Nodo<T>* cargarSinRepetir(Nodo<T>*& l,T v,
int (*criterio)(T,T)){
  Nodo<T>* x = buscar<T,T>(p,v,criterio);

  if( x == NULL )
  {
    x = insertarOrdenado<T>(p,v,criterio);
  }

  return x;
}
```

Una aplicación completa

```
struct TipoRegistro // para un nodo con una strucct en el campo info
{
  int c1;
  string c2;
}

template <typename T> struct Nodo
{
  T info;      // valor que contiene el nodo
  Nodo<T>* sig; // puntero al siguiente nodo
};

// para dato simple será:          Nodo<int>* l1 = NULL;
// para nodo con struct será:      Nodo<TipoRegistro>* l2 = NULL;
```

//Insertar ordenado con plantilla y criterio

template <typename T>

```
Nodo<T>* insertarOrdenado(Nodo<T>*& p, T v, int (*criterio)(T,T) ){
  Nodo<T>* nuevo = new Nodo<T>();
  nuevo->info = v;
  nuevo->sig = NULL;
  Nodo<T>* actual = p;
  Nodo<T>* ant = NULL;
  while( actual!=NULL && criterio(actual->info,v)<=0 ) {
    ant = actual;
    actual = actual->sig;
  }
  if( ant==NULL )  p = nuevo;
   else ant->sig = nuevo;

  nuevo->sig = actual;
  return nuevo;
}
```

```
Nodo* insertarOrdenado(Nodo*& p, TipoRegistro v) {
  Nodo* nuevo = new Nodo();
  nuevo->info = v;
  nuevo->sig = NULL;
  Nodo<T>* actual = p;
  Nodo<T>* ant = NULL;
  while( actual!=NULL && criterio(actual->info,v)<=0 ) {
    ant = actual;
    actual = actual->sig;
  }
  if( ant==NULL )  p = nuevo;
   else ant->sig = nuevo;

  nuevo->sig = actual;
  return nuevo;
}
```

```
// funciones auxiliares de criterio
int criterioCampo1Asc(TipoRegistro a,int c1) {
  return a.c1-c1;      // si el dato a cargar es mayor retorna negativo y sigue en el while
}

int criterioCampo1Desc(TipoRegistro a,int c1) {
  return c1 - a.c1;  // si el dato a cargar es menor retorna positivo  y sale del while
}

int criterioCampo2(TipoRegistro a,string c2){
  return strcmp(a.c2.c_str(),c2);
}
```

```
TipoRegistro cargarRegistro(int c1, string c2)
{
  Alumno a;
  a.leg = leg;
```

```
  a.nom = nom;
  return a;
}
```

```
int main()
{
  Nodo<TipoRegistro>* l1 = NULL;        // lista con plantilla por c1 asc
  Nodo* l2 = NULL;                      // lista sin plantilla
  Nodo<TipoRegistro>* l3 = NULL;        // lista con plantilla por c1 desc
  Nodo<TipoRegistro>* l4 = NULL;        // lista con plantilla por c2


 // carga ordenado por c1 ascendente
  insertarOrdenado<TipoRegistro>(l1,cargarRegistro(1,"Roxana")criterioCampo1Asc);
  insertarordenado<TipoRegistro>(l1,cargarRegistro(2,"Oscar")criterioCampo1Asc);

// carga ordenado por c1 descendente
  insertarOrdenado<TipoRegistro>(l3,cargarRegistro(1,"Roxana")criterioCampo1Desc);
  insertarordenado<TipoRegistro>(l3,cargarRegistro(2,"Oscar")criterioCampo1Desc);

// carga ordenado por c2
  insertarOrdenado<TipoRegistro>(l4,cargarRegistro(1,"Roxana")criterioCampo2);
  insertarordenado<TipoRegistro>(l4,cargarRegistro(2,"Oscar")criterioCampo2);

 // carga sin plantilla
  insertarOrdenado(l2,cargarRegistro(1,"Roxana"));
  insertarordenado(l4,cargarRegistro(2,"Oscar"));


……..
```

Ejercicio Nro. 1:

Idem ejercicio 60 pero retornando un parámetro con valor **'S'** o **'N'** según haya sido exitoso o no el requerimiento. (Definir parámetros y codificar).

Ejercicio Nro. 2:

Dada una pila y dos valores **X** e **I**, desarrollar un procedimiento que inserte el valor **X** en la posición **I** de la pila si es posible. (Definir parámetros y codificar).

Ejercicio Nro. 3:

Dada una pila y un valor **X**, desarrollar un procedimiento que inserte el valor **X** en la última posición de la pila y la retorne. (Definir parámetros y codificar).

Ejercicio Nro. 4:

Dada una pila y dos valores **X** e **Y**, desarrollar un procedimiento que reemplace cada valor igual a **X** que se encuentre en la pila por el valor **Y** retornando la pila modificada. En caso de no haber ningún valor igual a **X** retornar la pila sin cambio. (Definir parámetros y codificar).

Ejercicio Nro. 5:

Definir una función **INVERSA** que evalúe dos conjuntos de caracteres separados por un punto y retorne True si los conjuntos son inversos (ej: ABcDe.eDcBA) o False si no lo son. Los conjuntos deben ingresarse por teclado. (Definir parámetros y codificar).

Ejercicio Nro. 6:

Desarrollar un procedimiento que ingrese por teclado un conjunto de Apellidos y Nombre de alumnos y los imprima en orden inverso al de ingreso. (Definir parámetros y codificar).

Ejercicio Nro. 7:

Dada una pila desarrollar un procedimiento que ordene la misma de acuerdo al valor de sus nodos y la retorne. Solo se deben usar pilas. (Definir parámetros y codificar).

Ejercicio Nro. 8:

Dada una cola (nodo = registro + puntero), desarrollar y codificar un procedimiento que elimine 2 nodos de la misma (indicar con un parámetro 'S'/'N' si ello fue, o no posible)

Ejercicio Nro. 9:

Dada una cola (nodo = registro + puntero), desarrollar y codificar una función que devuelva la cantidad de nodos que tiene.

Ejercicio Nro. 10:

Dadas dos colas **COLA** y **COLB** (nodo = registro + puntero), desarrollar y codificar un procedimiento que genere una única cola **COLAB** a partir de ellas. (Primero los nodos de **COLA** y luego los de **COLB**).

Ejercicio Nro. 11:

Dada una cola (nodo = registro + puntero), imprimirla en orden natural si tiene más de 100 nodos, caso contrario imprimirla en orden inverso.

Ejercicio Nro. 12:

Dadas dos colas **COLA** y **COLB** (nodo = registro + puntero), desarrollar y codificar un procedimiento que genere otra cola **COLAB** por apareo del campo **ARRIBO** del registro (define orden creciente en ambas).
Nota: **COLA** y **COLB** dejan de ser útiles después del apareo.

Ejercicio Nro. 13:

Dado un archivo de registros de alumnos, donde cada registro contiene:
a) Apellido y Nombre del alumno (35 caracteres) b) Número de legajo (7 dígitos)
c) División asignada (1 a 100)
ordenado por número de legajo, desarrollar el algoritmo y codificación del programa que imprima el listado de alumnos por división, ordenado por división y número de legajo crecientes, a razón de 55 alumnos por hoja.

Ejercicio Nro. 14:

Idem Ejercicio Nro. 13, pero el listado de alumnos por división debe realizarse ordenado creciente por división y decreciente por número de legajo.

Ejercicio Nro. 15:

Idem Ejercicio Nro. 13 pero considerando que las divisiones asignadas son 100 y se identifican con un código de 4 caracteres.

Ejercicio Nro. 16:1

Dado un arreglo de **N** (< 30) colas (nodo = registro + puntero), desarrollar y codificar un procedimiento que aparee las colas del arreglo en las mismas condiciones que las definidas en el Ejercicio Nro. 12.
Nota: Retornar la cola resultante y no mantener las anteriores.

Ejercicio Nro. 17:

Dada una lista (nodo = registro + puntero), desarrollar y codificar una función que devuelva la cantidad de nodos que tiene.

Ejercicio Nro. 18:

Dadas dos listas **LISTA** y **LISTB** (nodo = registro + puntero), desarrollar y codificar un procedimiento que genere una única lista **LISTC** a partir de ellas. (Primero los nodos de **LISTA** y luego los de **LISTB**).

Ejercicio Nro. 19:

Dada una **LISTA** (nodo = registro + puntero), imprimirla en orden natural si tiene más de 100 nodos, caso contrario imprimiría en orden inverso.

Ejercicio Nro. 20:

Dadas dos listas **LISTA** y **LISTB** (nodo = registro + puntero), desarrollar y codificar un procedimiento que genere otra lista **LISTC** por apareo del campo **LEGAJO** del registro (define orden creciente en ambas).
Nota: **LISTA** y **LISTB** dejan de ser útiles después del apareo).

TRABAJO PRÁCTICO

CONTEXTO:

Una situación muy común en los restaurant y bares es que al momento de solicitar la cuenta los comensales gestionen los pagos de cada uno lo que suele ser un problema que demanda mucho tiempo de los mozos para informar el precio de cada plato.

Por ese motivo se solicita un programa que cargue los pedidos identificando: número de mesa, código del plato, cantidad, id de comensal.

Estos datos se iran almacenando en archivos denominados:

MESA999.dat donde 999 será reemplazado por el número de mesa.

Al finalizar y solicitar el cierre se pedirá que emita el listado agrupado por comensal.

Los dueños también le solicitan un informe estadístico de las ventas.

Funciones a desarrollar:

1. Carga de pedidos: se reciben los datos de cada pedido que se deben guardar en un archivo con el nombre MESA999.dato donde "999" será el numero de mesa real: número de mesa, código del plato, cantidad, id de comensal. Si el archivo existe se sigue acumulando datos sino se crea.

2. Solicitud de cierre: se recibe el número de mesa que solicita el cierre se hacen los cálculos para imprimir por pantalla el acumulado de cada comensal y el total general. Para este listado se cuenta con un archivo de precios que tiene la siguiente información: id de producto, nombre, precio.
   Este listado por pantalla deberá tener el siguiente formato:

   Comensal 1
   Nombre del plato Valor Cantidad Totales
   ….
   Total comensal 1

   Comensal 2
   Nombre del plato Valor Cantidad Totales
   ….
   Total comensal 2
   …
   Total de la mesa: $ …..

3. estadísticas: Al cerrar la mesa deberá ir acumulando información para las estadísticas. Para ello deberá utilizar un vector de listas donde deberá acumular los pedidos (id y cantidad) realizados para cada mesa y acumular lo cobrado para cada una con el objetivo de hacer la caja al final del día. Luego de almacenar la información el archivo debe borrarse. (ya sean los datos o el archivo físico)
4. Informes y cierre de caja: Al finalizar el día deberá emitir por pantalla el total acumulado del día y un informe que muestre por mesa el ranking de los 3 platos mas vendidos.

Se recuerda que deben entregar un informe con caratula donde se vean las pantallas , se explique la estrategia y el modo de uso. El trabajo debe compilar y resolver todos los puntos sin

excepción para considerarse aprobado, los trabajos incompletos pierden la promoción pero deberán completarse para aprobar la materia.