MODULO 4

Estructuras de datos estáticas

Breve descripción

En este documento se explica qué son los vectores, cómo se definen y cómo se utilizan.

Versión 1.1

Martin Jerman
Martin.jerman@inspt.utn.edu.ar

Introducción

En una empresa, le piden a uno de sus empleados, que construya una aplicación que, mediante un histograma, represente el total facturado en 7 días. La solicitud, implicaría declarar 7 variables para ingresar los valores y así poder representar el histograma.

Si se desease hacer uso de variables, tendríamos un código como el siguiente:

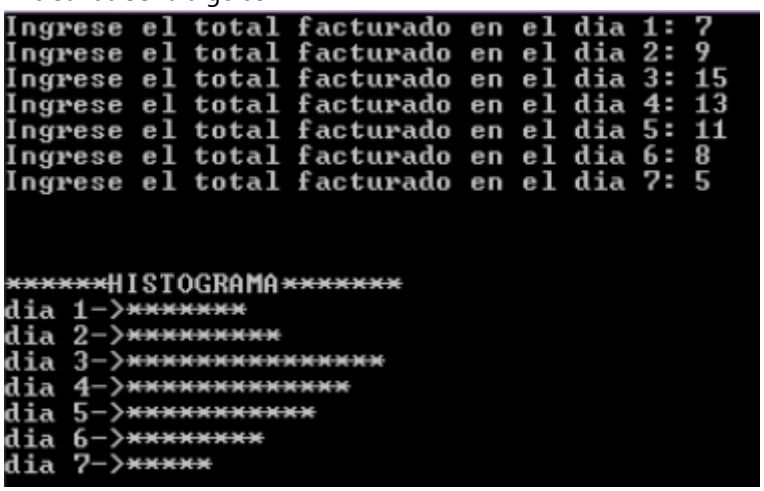
```
main(){
	int dia1=0, dia2=0, dia3=0, dia4=0, dia5=0, dia6=0, dia7=0;
	ingresarDiaria(&dia1, &dia2, &dia3, &dia4, &dia5, &dia6, &dia7);
	emitirEstacionalidadSemanal(dia1, dia2, dia3, dia4, dia5, dia6, dia7);
	printf("\n\n");
	system("pause");
	return 0;
	}

void imprimirHistograma(int dia){
	int i;
	for (i=0; i<dia; i++) printf("*");
	printf("\n");
	}

void emitirEstacionalidadSemanal( int dia1, int dia2, int dia3, int dia4, int
dia5, int dia6, int dia7){
	printf("\n\n\n******HISTOGRAMA*******\n");
	printf("dia 1->"); imprimirHistograma(dia1);
	printf("dia 2->"); imprimirHistograma(dia2);
	printf("dia 3->"); imprimirHistograma(dia3);
	printf("dia 4->"); imprimirHistograma(dia4);
	printf("dia 5->"); imprimirHistograma(dia5);
	printf("dia 6->"); imprimirHistograma(dia6);
	printf("dia 7->"); imprimirHistograma(dia7);
	}

void ingresarDiaria(int *dia1, int *dia2, int *dia3, int *dia4, int *dia5, int
*dia6, int *dia7){
	printf("Ingrese el total facturado en el dia 1: "); scanf("%d", dia1);
	printf("Ingrese el total facturado en el dia 2: "); scanf("%d", dia2);
	printf("Ingrese el total facturado en el dia 3: "); scanf("%d", dia3);
	printf("Ingrese el total facturado en el dia 4: "); scanf("%d", dia4);
	printf("Ingrese el total facturado en el dia 5: "); scanf("%d", dia5);
	printf("Ingrese el total facturado en el dia 6: "); scanf("%d", dia6);
	printf("Ingrese el total facturado en el dia 7: "); scanf("%d", dia7);
	}
```

Y la salida sería algo así:



¿Pero qué ocurriría si en otro momento, tenemos que hacer el histograma para 10 días? ¿Y qué pasaría si fuera para 30 días? Si, el código sería un caos de copy/paste de las mismas líneas. Obviamente hay una forma mejor de hacerlo y es usando vectores.

¿Qué es un vector o array?

Los vectores son una forma de almacenar datos que permiten contener una serie de valores del mismo tipo, cada uno de los valores contenidos tiene una posición asociada que se usará para accederlos. Está posición o índice será siempre un número entero positivo.

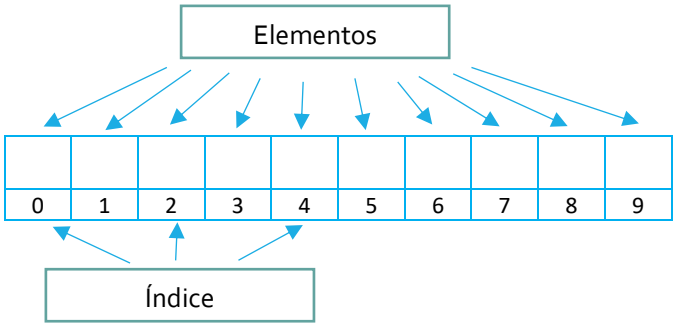
En C la cantidad de elementos que podrá contener un vector es fijo, y en principio se define cuando se declara el vector. Los vectores se pueden declarar de la siguiente forma:

```
tipo_elemento nombre[largo];
```

Esto declara la variable nombre como un vector de `tipo_elementos` que podrá contener largo cantidad de elementos, y cada uno de estos elemento podrá contener un valor de tipo `tipo_elemento`.

Por ejemplo:

```
double valores[10];
```



El índice del primer elemento siempre es cero

En este ejemplo declaramos un vector de 10 elementos del tipo `double`, los índices de los elementos irían entre cero (para el primer elemento) y 9 (para el último).

En el ejemplo anterior el tamaño del vector se especificó explícitamente con un 10 en la declaración de la variable. Si bien es válido, no es lo mas recomendable para cuando pasemos vectores a las funciones. Lo ideal es utilizar la instrucción de pre-compilador `define` para definir el tamaño como una constante. Por ejemplo:

```
#include <stdio.h>
#define VECSIZE 10

main(){
    int i, vec[VECSIZE];
    return 0;
    }
```

En los ejemplos hasta ahora vistos, hemos declarado vectores de `float` e `int`, pero pueden ser de cualquier otro tipo de dato primitivo (`double` o `char`) o compuesto (`struct`).

Inicialización de vectores

De la misma forma que con las otras declaraciones de variables que hemos visto se le puede asignar un valor inicial a los elementos o también se pueden declarar:

```
int nombre[largo]={2, 4, 8};
```

En caso estamos asignándole valores a los primeros 3 elementos del vector nombre. Notar que largo debe ser mayor o igual a la cantidad de valores que le estamos asignando al vector, en el caso de ser la misma cantidad no aporta información, por lo que el lenguaje nos permite escribir:

```
int nombre[]={2, 4, 8};
```

Que declarará nombre como el vector de largo 3, es decir que si **no se especifica el tamaño** se **reservarán tantos espacios como elementos** haya entre llaves.

No se puede inicializar todos los elementos de un array en una línea diferente a la de la declaración:

```
int dias[7];
dias = {7,9,15,13,11,8,5}; //error
```

Ni tampoco se puede inicializar un array con más elementos de los declarados en la dimensión:

```
int dias[7] = {7,9,15,13,11,8,5,25,2}; //error
```

Acceso a los valores

Para acceder a un elemento accederemos a través de un índice que referencia su posición. Por ejemplo:

```
int elemento, vec[10];
elemento = vec[3];
```

Asumiendo que tenemos el vector anterior definido estaríamos guardando el valor de la posición 4 (recordar que se cuenta desde cero) en la variable `elemento`.

Para recorrer el vector completo, podemos utilizar una variable como índice dentro de un ciclo, como se ve a continuación.

```
#include <stdio.h>
double producto_escalar(double v1[], double v2[], int d);

int main(){
	const int largo = 3;
	double vector_1[] = {5,1,0};
	double vector_2[] = {-1,5,3};

	double resultado = producto_escalar(vector_1, vector_2, largo);
	// imprime el resultado
	printf("(%f, %f, %f) . (%f, %f, %f) = %f\n",
		vector_1[0], vector_1[1], vector_1[2],
		vector_2[0], vector_2[1], vector_2[2],
		resultado);
	return 0;
	}

/* producto escalar entre dos vectores */
double producto_escalar(double v1[], double v2[], int d){
	double resultado = 0;
	int i;
	for (i=0; i < d; i++) {
		resultado += v1[i] * v2[i];
	}
	return resultado;
}
```

En el ejemplo anterior usamos los vectores de C para representar vectores matemáticos y calcular el producto escalar entre ellos. Una peculiaridad que se puede notar es que al recibir un arreglo en una función no se especifica el largo. Veremos este punto más adelante.

Otra función clásica es la búsqueda de un máximo o mínimo, que podemos escribirla de la siguiente manera:

```
int buscar_maximo(double valores[], int num_valores){
	int maximo_pos = 0;
	for (int i = 1; i < num_valores; i++) {
		if (valores[i] > valores[maximo_pos]) {
			maximo_pos = i;
		}
	}
	return maximo_pos;
}
```

Otro ejemplo sencillo, calcular el promedio de los valores.

```
double promedio(double valores[], int largo)
{
	double suma=0;
	for (int i=0;i<largo;i++) {
		suma+=valores[i];
	}
	return suma/largo;
}
```

Hasta aquí hemos visto que las funciones reciben un vector por parámetro, pero no se especifica en la propia declaración de la variable en tamaño el vector. Sino que el tamaño es otro parámetro de la misma función. Eso es así porque al ser un vector un espacio de memoria contiguo, el valor de la **propia variable vector es la dirección de memoria de la primera posición**. Es decir que una variable vector es un puntero a la primer posición de memoria de ese vector y por ende, **siempre se pasa por referencia**.

Un vector por ser un puntero al primer elemento, cuando se pasa a una función se pasa siempre por referencia.

Vectores y su relación con los punteros

Como comentamos anteriormente, la variable que se declara como vector es una dirección de memoria al primer elemento y es constante, es decir que no podemos hacer que un vector apunte por defecto a otro elemento que no sea el primero.

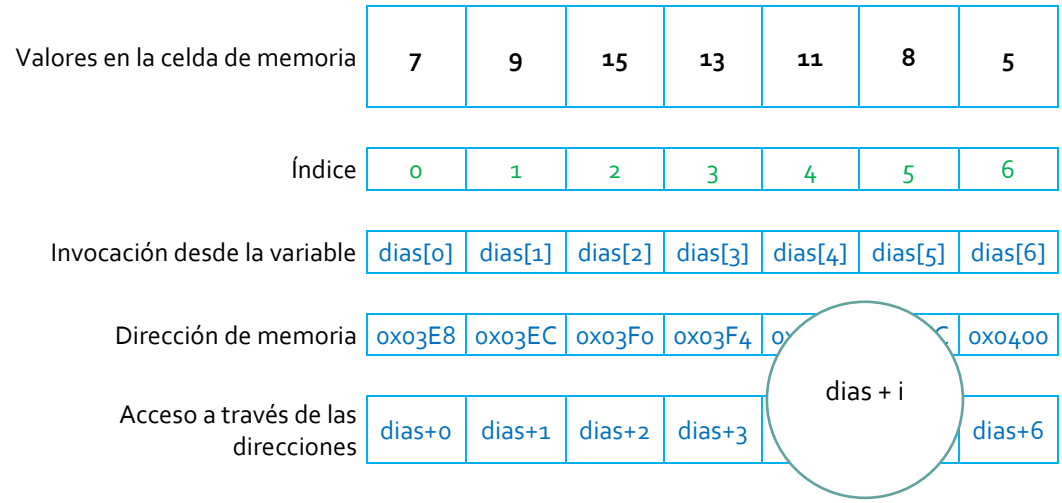
Por ejemplo, si tenemos la declaración:

```
int dias[7] = {7,9,15,13,11,8,5};
```

Valores en la celda de memoria	**7**	**9**	**15**	**13**	**11**	**8**	**5**
Índice	0	1	2	3	4	5	6
Invocación desde la variable	dias[0]	dias[1]	dias[2]	dias[3]	dias[4]	dias[5]	dias[6]
Dirección de memoria	0x03E8	0x03EC	0x03F0	0x03F4	0x03F8	0x03FC	0x0400

Entonces dado que un vector es un puntero al primer elemento y que es un espacio contiguo de memoria, podemos hacer lo siguiente:

```
printf("el elemento %d es %d\n", i, dias+i);
```



Ahora bien, ¿siendo un vector un espacio contiguo de memoria, como puedo acceder a la dirección de memoria del elemento i? Pues bien, es necesario primero llegar al elemento `i` y luego utilizar el operador de dirección `&`. Con esto quedaría:

```
&dias[i];
```

Lo que es lo mismo que decir:

```
dias+i;
```

Por lo tanto: `&dias[i] = dias+i;`

¡Se puede acceder al contenido de los vectores a través de la notación de punteros! Por ejemplo:

```
*(dias+i);
```

De aquí entonces que

```
*(dias+i) = dias[i];
```

Valores en la celda de memoria	7	9	15	13	11	8	5
Índice	0	1	2	3	4	5	6
Invocación desde la variable	dias[0]	dias[1]	dias[2]	dias[3]	dias[4]	dias[5]	dias[6]
Dirección de memoria	0x03E8	0x03EC	0x03F0	0x03F4	0x03F8	0x03FC	0x0400
Acceso a través de direcciones	dias+0	dias+1	dias+2	dias+3	dias+4	dias+5	dias+6
que es igual a decir	&dias[0]	&dias[1]	&dias[2]	&dias[3]	&dias[4]	&dias[5]	&dias[6]
acceso por punteros	*(dias+0)	*(dias+1)	*(dias+2)	*(dias+3)	*(dias+4)	*(dias+5)	*(dias+6)

Vectores y funciones

Como se dijo anteriormente, los vectores son punteros al primer byte del espacio de memoria al que apuntan. Esto nos hace pensar en: ¿cómo puedo hacer para pasar un vector a una función? Pues bien, se pasa como un puntero, por referencia, siempre. Vemos un ejemplo:

```
#define DIM 10
void modificarVector(int vec[]){
    vec[1]=0;
}
```

```
int main() {
    int miVector[DIM], i;

    for (i=0 ; i<DIM ; i++)
        miVector[i]=1;
    modificarVector(miVector);
    for (i=0 ; i<DIM ; i++)
        printf("%d",miVector[i]);
    return 0;
}
```

En el ejemplo anterior, vemos como un vector de enteros se carga con 1s y luego la función `modificarVector()` edita el contenido sin necesidad de pasar el vector por referencia. Esto también funciona si el vector se quiere devolver como resultado de la función:

```
int* modificarVector2(int vec[]){
    vec[1]=0;
    return vec;
}
```

¿Ahora bien, podemos crear un vector en una función y devolverlo? La respuesta es **NO**. Esto es así porque los vectores son un espacio determinado al momento de ejecutar la función, como una variable local. Por ende, al terminar de ejecutar la función, este desaparece. La única forma de lograr esto es **no usar memoria estática** y usar funciones para pedir **memoria dinámicamente**. Estas funciones se verán más adelante.

Vectores y las cadenas de caracteres (strings)

En C, una cadena de caracteres o `string` es una sucesión de caracteres ASCII en un espacio de memoria. Además, debe terminar con el carácter nulo o `'\0'` (contra barra cero) que muchas funciones utilizan para saber que la cadena termino. Finalmente, los elementos de la cadena son del tipo `char`. Entonces un string en C:

- Es una sucesión de caracteres ASCII
- Sus elementos son del tipo `char`
- Termina con el carácter nulo o `\0`

La declaración de un string es igual a la de un vector:

```
char cadena[11];
```

o

```
char cadena[TAM];
```

Además, podemos inicializar la cadena como un vector

```
char cadena[11]={'H', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o', '\0'};
```

Quedando en memoria:

H	o	l	a		m	u	n	d	o	\0

0	1	2	3	4	5	6	7	8	9	10

Aquí `'H'` es un carácter, pero `"Hola mundo"` es una cadena de caracteres.

Otra forma de inicializar una cadena es la siguiente:

```
char cadena[]="Hola mundo";
```

En este caso, cadena es un espacio de memoria de 10 lugares, uno para cada carácter del string incluido el barra cero. Además, `cadena` **es un puntero constante**, es decir que no puede cambiar

su valor dado que apunta al espacio de memoria asignado. En este punto un vector es diferente al puntero (este último si puede ser reapuntado a otro espacio de memoria).

Otra forma de declaración es la siguiente:

```
char * otraCadena;
```

Entonces:

```
otraCadena = "Me encanta programar";
```

Asigna un puntero al array de caracteres. Esta forma se conoce como **literal cadena**. En este caso el puntero `otraCadena` sí puede ser reapuntado a otro espacio de memoria o cadena de caracteres. Además, los literales de cadena se almacenan en el segmento de datos. Es el mismo sitio que se reserva para los valores constantes y variables globales. Un literal es tratado como una constante.

Ahora bien, un literal cadena no es la copia de una cadena, es un puntero. ¿Existe alguna diferencia entre ellas?, sí:

```
char cadena[]="Hola mundo";
```
→ Es un array

```
char * otraCadena="Hola mundo";
```
→ Es un puntero al array

Ingreso de cadenas

Se puede ingresar una cadena carácter por carácter utilizando la función `scanf`:

```
scanf("%c", &caracter[i]);
```

O se puede ingresar el texto completo utilizando la función `scanf`:

```
scanf("%s", cadena);
```

En este caso, el nombre del array no lleva el operador de dirección de memoria `&`, porque lleva implícito la dirección de memoria.

Podemos limitar la entrada de caracteres de esta forma:

```
scanf("%10s", cadena);
```

La función `scanf` guarda lo que pulsemos por teclado hasta el primer espacio en blanco o `enter`, por lo cual, si ingresamos una cadena compuesta, sólo se emitirá la cadena hasta el primer espacio blanco. Podemos evitarlo utilizando:

```
scanf("%[^\n] ", cadena);
```

o

```
gets(cadena);
```

Impresión de cadenas

Por otra parte, las cadenas pueden emitirse carácter a carácter:

```
printf ("%c", caracter[i]);
```

O bien pueden emitirse completas:

```
printf ("%s", cadena);
```

La función `printf` recibe un apuntador al inicio de la cadena, es decir, se tiene acceso a una cadena constante por un puntero a su primer elemento. Emitir la cadena es una característica propia de la función. El formato `%s` indica que interprete esa dirección como el comienzo de una cadena y la función interpreta carácter a carácter hasta encontrar el `'\0'`.

Funciones para cadenas

Para realizar operaciones con caracteres existen funciones de biblioteca definidas en `ctype.h`.

Ejemplos de funciones para tratamiento de caracteres son:

`isalnum(caracter)`: devuelve cierto (un entero cualquiera distinto de cero) si carácter es una letra o dígito, y falso (el valor entero o) en caso contrario.

`isalpha(caracter)`: devuelve cierto si carácter es una letra, y falso en caso contrario.

`isblank(caracter)`: devuelve cierto si carácter es un espacio en blanco o un tabulador.

`isdigit(caracter)`: devuelve cierto si carácter es un digito, y falso en caso contrario.

`isspace(caracter)`: devuelve cierto si carácter es un espacio en blanco, un salto de línea, un retorno de carro, un tabulador, etc., y falso en caso contrario.

`islower(caracter)`: devuelve cierto si carácter es una letra minúscula, y falso en caso contrario.

`isupper(caracter)`: devuelve cierto si carácter es una letra mayúscula, y falso en caso contrario.

`toupper(caracter)`: devuelve la mayúscula asociada a carácter, si la tiene; si no, devuelve el mismo carácter.

`tolower(caracter)`: devuelve la minúscula asociada a carácter, si la tiene; si no, devuelve el mismo carácter.

Para realizar operaciones con cadenas hay que usar funciones de biblioteca. El lenguaje no dispone de operadores para cadenas, pero se utilizarán funciones definidas en `string.h`. Algunos ejemplos de funciones para tratamiento de cadenas son:

```
char *strcpy(char *s1, const char*s2);
```
Copia la cadena apuntada por s2 (incluyendo el carácter nulo) a la cadena apuntada por s1. La función retorna el valor de s1.

```
int strcmp(const char *s1, const char *s2);
```
Compara la cadena apuntada por s1 con la cadena apuntada por s2. La función retorna un número entero mayor, igual, o menor que cero, según si la cadena apuntada por s1 es mayor, igual, o menor que la cadena apuntada por s2.

```
char *strcat(char*s1, const char *s2);
```
Agrega una copia de la cadena apuntada por s2 (incluyendo el carácter nulo) al final de la cadena apuntada por s1. El carácter inicial de s2 sobrescribe el carácter nulo al final de s1 La función retorna el valor de s1.

```
size_t strlen(const char *s);
```
Calcula el número de caracteres de la cadena apuntada por s. La función retorna el número de caracteres hasta el carácter nulo, que no se incluye.

La función strtok

Esta función es muy especial por cómo funciona. Su función es separar una cadena recibida por parámetros en palabras o tokens según una lista de separadores o delimitadores que también se recibe por parámetro. Su encabezado es:

```
char *strtok(char *str, const char *delim);
```
en donde `str` es la cadena que vamos a procesar y `delim` es la cadena de separadores.

¿Cómo función la función? Pues bien, tiene dos formas de llamada. En el primer uso, la función se invoca con los parámetros que se utilizaran, por ejemplo:

```
char texto[] = "Texto de ejemplo. Utiliza, varios delimitadores\n";
char delim[] = " ,.";
char *token;
token = strtok( texto, delim);
```
hasta aquí, la función recibe los parámetros y se guarda en token el puntero a la primer palabra o `token` obtenida. Para tomar la segunda palabra, se debe llamar a la función de la siguiente manera:

```
token=strtok(NULL,delim);
```

donde el parámetro a procesar es `NULL`. ¿Por qué es esto? Porque la función `strtok` guarda internamente de forma estática (`static`) la cadena a procesar, de manera no se pierde el valor de la variable en los siguientes llamados a la función. Es muy importante mantener la misma lista de separadores como segundo parámetro. Caso contrario se cambia el comportamiento de la función.

Veamos un ejemplo de código:

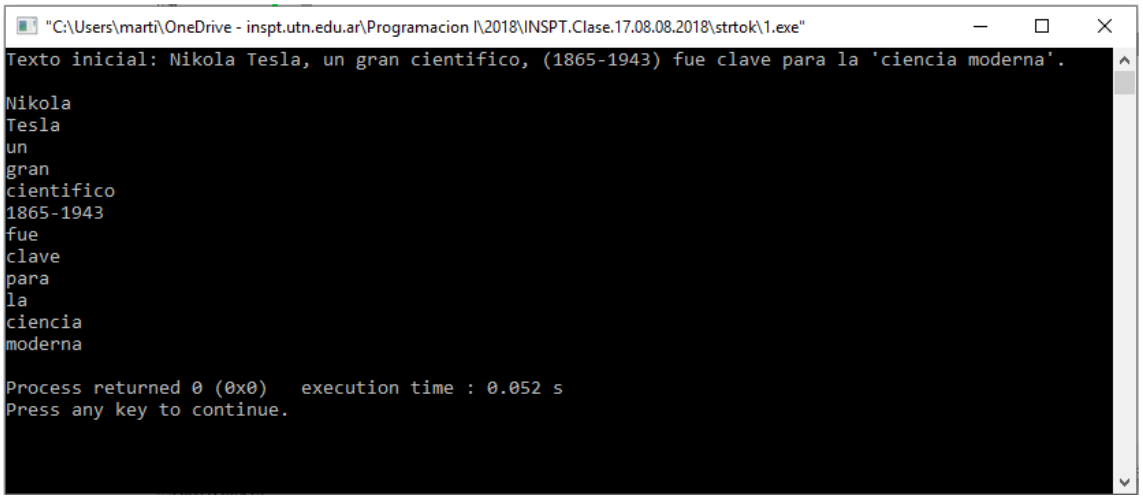
```
#include <stdlib.h>
#include <stdio.h>

int main(){
char texto[] = "Nikola Tesla, un gran cientifico, (1865-1943) fue clave para la
'ciencia moderna'.";
char signosPuntuacion[] = " (,.)'";
char *token;

printf("Texto inicial: %s\n\n", texto);
token = strtok( texto, signosPuntuacion);

do{
    printf("%s\n", token);
       token=strtok(NULL,signosPuntuacion);
   } while(token != NULL );
return 0;
}
```

En este ejemplo se procesa la cadena `texto`, y tras utilizar los separadores `signosPuntuacion` obtenemos la siguiente salida:



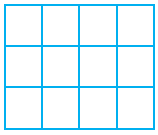
Como se ve en la salida, hay un error: se olvidó agregar el guion medio (-) como separador para separar los años de nacimiento y defunción. Agregando ese carácter al vector de `signosPuntuacion` se resuelve el problema.

Vectores multidimensionales

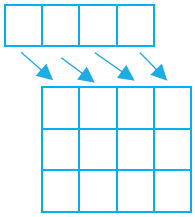
Hasta ahora hemos visto qué son y cómo se trabaja con vectores y en particular las cadenas de caracteres. Si graficamos esto, vemos que un vector es como una grilla de casilleros en una sola línea. A estos vectores se los conoce como vectores unidimensionales.

Ahora bien, ¿qué sería un vector bidimensional? Pues bien, no es mas que un vector por cada posición del vector "raíz", lo que usualmente se le dice **una matriz**.

Si bien pensamos en términos de programación pensamos que la matriz es esto:



En realidad, lo que es es esto:



Aquí, el vector principal contiene direcciones de memoria a otros vectores que forman "la matriz".

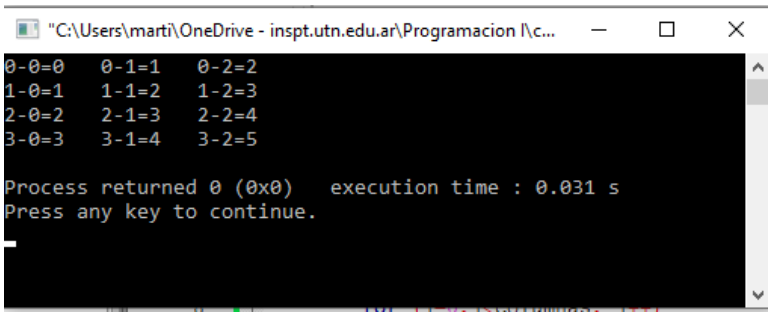
Veamos un ejemplo:

```
#include <stdio.h>
main(){
    int i,j,filas=4,columnas=3;
    int mat[filas][columnas];

    /* cargamos la matriz con datos */
    for (i=0 ; i<filas ; i++)
        for (j=0 ; j<columnas ; j++)
            mat[i][j]=i+j;

    /* mostramos la matriz */
    for (i=0 ; i<filas ; i++){
        for (j=0 ; j<columnas ; j++)
            printf("%d-%d=%d\t", i, j, mat[i][j]);
        printf("\n");
    }
    return 0;
    }
```

La salida de este programa es la siguiente:



En el ejemplo anterior se ve como se declara un vector bidimensional o matriz:

```
int mat[filas][columnas];
```

En este caso, se opto por iniciar los valores utilizando un ciclo y los índices para recorrer la matriz y cargarle un valor. Pero se puede inicializar como los vectores, de la siguiente manera:

```
int mat[4][3]= {7,9,15,13,11,8,5,9,10,2,8,4};
```

o bien

```
int mat[4][3]= {{7,9,15},{13,11,8},{5,9,10},{2,8,4}};
```

En este caso, si se omiten valores, se completa con ceros.

También se puede omitir la primera dimensión si se inicializa en la declaración, es decir que en este caso la inicialización es forzosa. Por ejemplo:

```
int mat[][3] = {7,9,15,13,11,8,5,6,7,1,3,8};
```

De esta forma, el vector toma la dimensión de la cantidad de elementos definidos en la inicialización.

Luego queda recorrer la matriz para trabajar cada posición, en este caso simplemente mostrar su valor por pantalla. Hay que notar que en este caso la matriz es de dos dimensiones, por lo que es necesario dos ciclos `for` para trabajar los datos. Podemos tener matrices de 3 dimensiones o más; a mayor cantidad de dimensiones, mayor cantidad de ciclos para recorrerlas.

Estructuras o registros

Hasta el momento hemos visto los tipos primitivos de variables, como el char, el int o el float. Luego vimos el pseudo tipo de dato cadena o string; que es un vector de char terminado con el carácter `'\0'`.

Ahora bien, ¿qué pasaría si necesitamos trabajar con una lista de legajos de alumnos (`int`), sus 3 notas (`int`) y calcular el promedio de cada legajo (`float`)?

Necesitaríamos 5 vectores: 1 para los legajos, 1 para la nota 1, otro para la nota 2, otro para la nota 3 y otro para el promedio. Además, deberíamos tener en cuenta que los relacionaríamos por posición: el legajo de la posición 1 va con los datos de la posición 1 de los demás vectores.

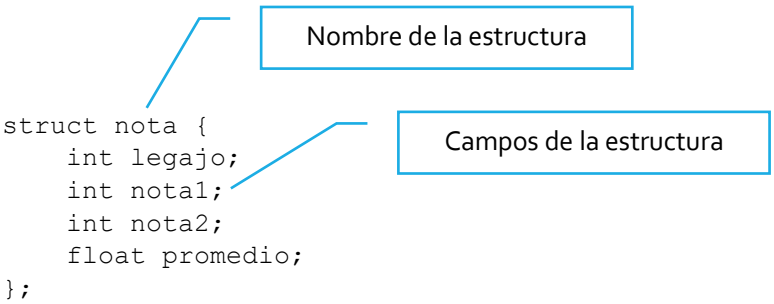
Lo anterior sería una solución, pero es poco práctica. Existe una forma más practica y cómoda para trabajar cuando se tienen este tipo de datos y es a través de las estructuras de datos.

Las estructuras (struct)

Las estructuras nos permiten agrupar valores de diferentes tipos de datos en una misma variable. Diremos que la variable es de un **tipo compuesto**. De aquí se desprende que:

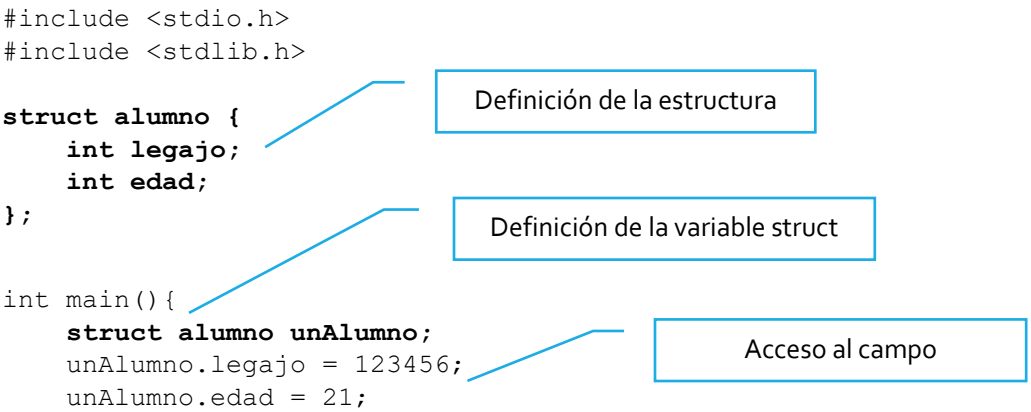
- Una estructura se divide en campos
- Cada campo puede ser de cualquier tipo de dato
- Cada campo se identifica con un nombre

Veamos cómo se declara en C:

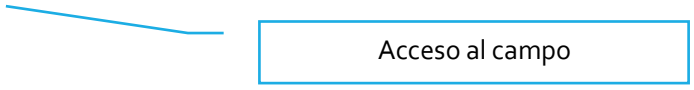


Con esta definición hemos creado un nuevo **tipo de dato** llamado **`notas`**, el cual es una estructura que consta de 4 campos (3 `int` y un `float`).

Dado que las estructuras son tipos de datos que el programador define en el propio código fuente, deben de definirse antes de ser utilizadas. Por ende, se definen fuera de toda función, incluso de `main`. Veamos un ejemplo sencillo:



```
    printf("El legajo %d tiene una edad de %d\n", unAlumno.legajo,
unAlumno.edad);
    return 0;
}
```



Como se ve en el ejemplo anterior, la estructura esta definida fuera del `main`.

En la primera línea del código se utiliza el nuevo tipo de dato alumno para definir la variable `unAlumno` y en las siguientes líneas se les cargan valores a sus campos para luego ser mostrados. En ambos casos, se accede a los campos a través del punto (.). De aquí se desprende que es válido hacer lo siguiente:

```
unAlumno.edad = unAlumno.edad + 25;
```

Otra forma de definir estructuras es al momento de declarar la variable, con lo cual la definición de la estructura está dentro de `main`, por ejemplo:

```
int main(){
    struct miStruct {
        int legajo;
        char genero;
    } otroAlumno;
    printf("No hago nada\n");
    return 0;
}
```

Si bien es correcta la definición del a estructura anterior, no se suele definir la estructura en el propio cuerpo del main dado que el alcance de dicha definición es el propio cuerpo de main. Por lo general necesitamos aquella estructura como tipo de dato en otras funciones; por lo cual es mejor definirla fuera de `main` y que dicha estructura quede disponible como tipo de dato, como en el primer ejemplo.

Al momento de declarar las variables tipo `struct` podemos inicializarle los campos de la siguiente manera:

```
#include <stdio.h>

struct empleado {
    int legajo;
    int edad;
    char nombre[10];
};

int main(){
    struct empleado unEmpleado = {1234, 21,"Martin"};
    printf("El profe %s tiene legajo %d\n",unEmpleado.nombre,
unEmpleado.legajo);
    return 0;
}
```

Es importante destacar que, si se inicializa una variable de esta forma, **es necesario colocarles un valor a todos los campos** de la estructura. También es importante resaltar que, si un campo es un vector, éste debe iniciarse con una sucesión de los valores. En el ejemplo se ve que es una cadena, pero observar el siguiente ejemplo:

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

```
struct alumno {
    int legajo;
    int notas[10];
};

int main(){
    struct alumno unAlumno = {1234, {4,7,9,8,4}};
    return 0;
}
```

Como se ve en el ejemplo, el vector se inicializa "elemento por elemento".

Al igual que ocurre con los tipos de datos primitivos (`int`, `double`, etc.), los tipos de dato creados por el programador o estructuras pueden asignarse entre variables siempre y cuando sean del mismo tipo. Por ejemplo:

```
int main(){
    struct alumno unAlumno = {1234, {4,7,9,8,4}};
    struct alumno otroAlumno;
    otroAlumno = unAlumno;
    return 0;
}
```

Y como los datos primitivos, también pueden pasar a las funciones por valor o ser devueltos por funciones.

Cabe hacer una observación en el caso de que la estructura se pase por referencia a la función. Veamos el ejemplo siguiente:

```
#include <stdio.h>

struct alumno {
    int legajo;
    int notas[10];
    float promedio;
};

void promediar(struct alumno* unAlumno){
    int i, aux = 0;

    for (i=0 ; i<10 ; i++)
        aux += (*unAlumno).notas[i];
    (*unAlumno).promedio = aux / 10.0;
}

int main(){
    struct alumno unAlumno = {1234, {4,7,9,8,4,4,7,10,8,5}};
    promediar(&unAlumno);
    printf("El promedio es %2.2f\n",unAlumno.promedio);
    return 0;
}
```

Como se puede ver, si la estructura se pasa por referencia hay que acceder al contenido del puntero `struct` para luego acceder al campo. Esta forma de escribirlo se ve algo engorrosa. Por ello, se puede acceder a los campos directamente con el operador flecha (**->**). Entonces la función podría definirse de la siguiente manera:

```
void promediar(struct alumno* unAlumno){
    int i, aux = 0;

    for (i=0 ; i<10 ; i++)
        aux += unAlumno->notas[i];
    unAlumno->promedio = aux / 10.0;
}
```

El operador flecha (->) simplifica el trabajo en punteros a estructuras.

Como vemos en el ejemplo, el operador flecha hace que trabajemos el campo como si fuera la variable directamente.

Estructuras en vectores

Como cualquier tipo de dato primitivo, las estructuras se pueden utilizar en vectores. Por ejemplo:

```
#include <stdio.h>

struct empleado {
    int legajo;
    int edad;
};

int main()
{
    struct empleado empleados[10];
    int i;

    for (i=0 ; i<10 ; i++){
        empleados[i].legajo = rand() % 1000;
        empleados[i].edad = 21 + i;
    }
    return 0;
}
```

Lo mismo que aplica a vectores unidimensionales también aplica a los multidimensionales

Observen como se accede a la posición del vector e inmediatamente se utiliza el operador punto para acceder al campo deseado.
Este mismo principio aplica para vectores multidimensionales.

Estructuras anidadas

Como el titulo lo indica, es posible anidar estructuras según se necesite. Veamos un ejemplo:

```
#include <stdio.h>

struct empleado {
    int legajo;
    int edad;
    char nombre[20];
    char apellido[20];
};
struct materia {
    int codigoMateria;
    char nombre[20];
};
```

```
struct profesor {
    struct empleado idEmpleado;
    struct materia idMateria;
};

int main()
{
    struct empleado unEmpleado = {1234, 21, "Axl", "Rose"};
    struct materia unaMateria = {12, "Canto Metalero"};
    struct profesor unProfe;

    unProfe.idEmpleado = unEmpleado;
    unProfe.idMateria = unaMateria;
    printf("El profe %s dicta la materia %s\n",unProfe.idEmpleado.nombre,
unProfe.idMateria.nombre);
    return 0;
}
```

Como se observa en el ejemplo, las estructuras se acceden una a otra a través del operador punto (.) hasta llegar al dato que se desea.

Definición de un tipo de dato

Por definición, para que un tipo de dato se considere un tipo de dato, no es suficiente definir la información que se va a guardar y cómo se va a guardar, sino también el cómo se va a procesar.

De aquí se desprende que un tipo de dato debe definir:

- Un conjunto de valores (que puede tomar el tipo de dato)
- Un conjunto de operaciones (que se puede hacer con los valores del tipo de dato)

Por ejemplo, dada la siguiente estructura:

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
```

Decir que la estructura anterior define el tipo de dato fecha **es incompleto** dado que no se sabe cómo operarla.
Para que realmente sea un tipo de dato, hay que definir las funciones u operaciones para esta estructura. En este caso podríamos definir las siguientes operaciones:

```
struct fecha newFecha(int dia, int mes, in anio);
int getDia(struct fecha);
int getMes(struct fecha);
int getAnio(struct fecha);
int esAnioBisiesto(struct fecha);
int getDiasEntre(struct fecha fechaInicio, struct fecha fechaFin);
```

En este punto sí tenemos un tipo de dato fecha definido, dado que tenemos la estructura definida y las operaciones que procesa dicha estructura.

Completando la definición con typredef

Hasta aquí vimos que la definición de una estructura ejemplo es la siguiente:

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
```

Pero al momento de definir la variable, el tipo de dato es algo engorroso:

```
struct fecha unaFecha;
```

Para mejorar la escritura del tipo de dato y convertirlo realmente en un tipo de dato, podemos definir la estructura como tal utilizando la sentencia **`typedef`**. Con lo cual podemos tener la estructura definida y luego agregando la línea:

```
typedef struct fecha Fecha;
```

Convertimos a la estructura fecha en un tipo de dato Fecha:

```
Fecha unaFecha;
```

Sin embargo, la definición de la estructura se puede combinar con **`typedef`**, con lo que la definición de la estructura nos quedaría:

```
typedef struct {
    int dia;
    int mes;
    int anio;
} Fecha;
```

Con lo cual al momento de utilizar esta definición nos quedaría igual:

```
fecha unaFecha;
```

Con esta forma de definir estructuras, se completa la forma de definir un **tipo de dato**.

Enumeraciones

Existe un tipo especial de variables denominadas variables enumeradas o simplemente enumeraciones. Se caracterizan por poder adoptar valores entre una selección de constantes enteras denominadas enumeradores, cuyos valores son establecidos en el momento de la declaración del nuevo tipo. Como se ha señalado, son enteros y (una vez establecidos) de valor constante, razón por la que se los denomina también constantes de enumeración. Ejemplo:

```
enum estado {MALO=0, REGULAR=1, BUENO=2, EXTRA=3};
```

La sentencia anterior declara `estado` como un tipo de variable de enumeración.

Los miembros de esta clase pueden adoptar los valores indicados y son representados por los mnemónicos: MALO, REGULAR, BUENO y EXTRA. Estas cuatro constantes son los enumeradores del nuevo tipo.

Veamos la declaración:

```
enum dias { DOM, LUN, MAR, MIE, JUE, VIE, SAB } diaX;
```

Lo anterior establece un tipo `enum` al que se identifica por `dias`; las variables de este tipo pueden adoptar un conjunto de seis valores enteros 0, 1, 2, 3, 4, 5, 6 (enumeradores) representados por los mnemónicos `DOM, LUN, . . . , SAB`. Además, se define una variable enumerada `diaX` de este tipo.

Ahora bien, también podemos hacer la declaración siguiente:

```
enum modelo { ULT =-1, BW40=0, C40, BW80, C80, MONO =7 };
```

Aquí se define un tipo `enum` al que identificamos por la etiqueta modelo; las variables de este tipo pueden adoptar 6 valores (-1, 0, 1, 2, 3 y 7) que se identifican con los nemónicos: `ULT`, `BW40`, `C40`, `BW80`, `C80` y `MONO`.

```
#include <stdio.h>
enum color{
    blanco, amarillo, rojo, verde, azul, marron, negro
    };

int main(){
    enum color color_coche;
    printf("Ingrese el color del coche\n");
    scanf("%d", &color_coche);
    while ((color_coche>=0)&&(color_coche<=6)){
        switch(color_coche) {
            case blanco: printf("El coche es blanco\n"); break;
            case amarillo: printf("El coche es amarillo\n"); break;
            case rojo: printf("El coche es rojo\n"); break;
            case verde: printf("El coche es verde\n"); break;
            case azul: printf("El coche es azul\n"); break;
            case marron: printf("El coche es marron\n"); break;
            case negro: printf("El coche es negro\n"); break;
            }
        printf("Ingrese el color del coche\n");
        scanf("%d", &color_coche);
        }
    system("pause");
    return 0;
    }
```

Así como con los structs, también podemos usar typedef con los enum. Por lo que podemos definir la enumeración:

```
typedef enum {
    blanco, amarillo, rojo, verde, azul, marron, negro
    } color;
```

Y al declarar la variable seria:

```
color unColor;
```

Uniones

Una unión es un tipo de datos derivado, como una estructura, con miembros que comparten el mismo espacio de almacenamiento. Una variable de tipo unión puede contener (en momentos diferentes) objetos de diferentes tipos y tamaños.

Las uniones proporcionan una forma de manipular diferentes tipos de datos dentro de una sola área de almacenamiento.

En cualquier momento una unión puede contener un máximo de un objeto debido a que los miembros de una unión comparten el espacio de almacenamiento.

Una unión se declara con el mismo formato de una `struct`. Primero declaramos el tipo unión y luego declaramos variables de ese tipo.

Veamos un ejemplo:

```
#include <stdio.h>
union numero {
    int x;
    double y;
    };

int main(){
    union numero valor;
    valor.x = 100;
    printf( "%s %s\n%s%d\n%s%2.2f\n\n",
        "Coloca un valor en el miembro entero",
        "e imprime ambos miembros.",
        "int:    ", valor.x,
        "double: ", valor.y );

    valor.y = 100.0;
    printf( "%s %s\n%s%d\n%s%f\n\n",
        "Coloca un valor en el miembro flotante",
        "e imprime ambos miembros.",
        "int:    ", valor.x,
        "double: ", valor.y );
    printf( "sizeof(valor.x):%d\n",sizeof(valor.x));
    printf( "sizeof(valor.y):%d\n",sizeof(valor.y));
    printf( "sizeof(valor):%d\n",sizeof(valor));
    return 0;
    }
```

Ejecutando el código anterior podemos ver que, si el campo `x` tiene un valor, el campo `y` no tiene valor. En cambio, si el valor `y` es seteado, se pierde el valor del campo `x`.

Ordenamiento y búsqueda

En las páginas anteriores hemos trabajado con datos que fueron cargados en vectores. Pero ¿qué ocurre si entre los datos que tiene cargados el vector hay datos inválidos que deben removerse? ¿O que ocurre si se necesita saber si determinado dato está o no en el vector? ¿O si los datos están en orden?

Para resolver estas cuestiones es necesario darles un vistazo a los algoritmos de ordenamiento y búsqueda. Ambos algoritmos serán explicados sobre vectores dado que en ellos se puede editar la información con facilidad. Algunos de estos algoritmos se pueden implementar en archivos binarios con algunas adaptaciones.

Algoritmos de ordenamiento

Cuando hablamos de "ordenar" algo, ¿en qué estamos pensando? En darle a ese "algo" un lugar determinado respecto de otro "algo" con el que lo comparamos. Entonces, ordenar un conjunto de elementos consiste en **comparar los elementos** entre sí y **según un criterio** darle un lugar o posición determinado. Por ejemplo, si el criterio de "de menor a mayor", la comparación será tomando de a dos elementos del conjunto evaluando si uno de ellos es "menor" que el otro y ubicando al "menor" en un lugar y al otro en otro.

Veamos a continuación algunos algoritmos de ordenamiento.

Ordenamiento por burbujeo

Este algoritmo es bien sencillo. Se va comparando cada elemento del arreglo con el siguiente; si un elemento es mayor que el que le sigue, entonces se intercambian; esto producirá que en el vector quede como su último elemento, el más grande. Este proceso deberá repetirse recorriendo todo el vector hasta que no ocurra ningún intercambio. Los elementos que van quedando ordenados ya no se comparan.

Si tenemos el siguiente vector desordenado:

1	5	4	2
0	1	2	3

Empezamos a comprar:

- 1 < 5 es verdadero, quedan igual: 1 5 4 2
- 5 < 4 es falso, se intercambian los valores: 1 4 5 2
- 5 < 2 es falso, se intercambian los valores: 1 4 2 **5**

Hasta este punto, el elemento mayor queda cargado en la última posición. Ahora queda volver a comparar todos los elementos nuevamente, excepto el último. Quedando:

- 1 < 4 es verdadero, quedan igual: 1 4 2 5
- 4 < 2 es falso, se intercambian los valores: 1 2 **4 5**

Y finalmente el último ciclo:

- 1 < 2 es verdadero, queda igual: 1 **2 4 5**

En este punto se termina el algoritmo, quedando

1 2 4 5

De este modo, la función de ordenamiento queda de la siguiente manera:

```
void ordenamientoBurbuja(int vec[], int dim){
int i, j, aux, cambio;
/* Ordenamiento*/
for (i = 0; i < dim-1; i++) {
          for (j = 0; j < dim - i; j++)
       if (v[j] > vec[j+1])
           intercambio(&vec[j], & vec[j+1]);
    }
}
```

Ordenamiento por burbujeo mejorado

Esta versión del algoritmo introduce una variable bandera "ordenado" para identificar que, si en un ciclo no se cambió ningún elemento de lugar, entonces el vector ya está ordenado y por ende no es necesario seguir ordenando.

Veamos el código:

```
void ordenamientoBurbujaMejorado(int v[], int d){
    int i,j,aux, ordenado;
    /* Ordenamiento*/
    for (i = 0; i < d-1; i++) {
        ordenado=1;
        for (j = 0; j < d-i; j++){
            if (v[j] > v[j+1]){
                intercambio(&v[j], &v[j+1]);
                ordenado=0;
            }
        }
        if (ordenado==1)
            i=d; /* si en el for interno no es intercambio, esta ordenado */
        }
    }
```

Aquí la variable "ordenado" indicara anticipadamente si el vector esta ordenado antes de terminar todos los ciclos.

Ordenamiento por inserción

El ordenamiento por inserción es una manera muy natural de ordenar. Consiste en tomar uno por uno los elementos de un arreglo y lo coloca en su posición con respecto a los anteriormente ordenados. Así empieza con el segundo elemento y lo ordena con respecto al primero. Luego sigue con el tercero y lo coloca en su posición ordenada con respecto a los dos anteriores, así sucesivamente hasta recorrer todas las posiciones del arreglo.

Veamos el ejemplo:

1	5	4	2
0	1	2	3

Empezamos a comparar:

- 5 > 1 es verdadero, queda igual: 1 5 4 2

Luego seguimos con el 4 de la posición 2:

- 4 > 1 es verdadero, quedan igual: 1 5 4 2
- 4 > 5 es falso, se intercambian: 1 4 5 2

Finalmente, con el 2 de la última posición:

- 2 > 1 es verdadero, queda igual: 1 4 5 2
- 2 > 4 es falso, se intercambian: 1 2 5 4
- 4 > 5 es falso, se intercambian: 1 2 4 5

Con esto, el algoritmo codificado queda así:

```
void ordenarPorInsercion(int v[], int d){
    int i,j, k, cont, aux;
    /* Ordenar y mostrar resultados intermedios*/
    for (cont = 1 ; cont < d ; cont++){
        /* Colocar v[cont] */
        aux = v[cont];
        k = cont-1; /* posicion del elemento a comparar */
        while ((v[k] > aux) && (k>0)){
            /* Desplazar elementos */
            v[k+1] = v[k];
            k--;
            }
        if (v[k] <= aux){
            /* posicion intermedia */
            v[k+1] = aux;
            }
        else { /* colocar el primero */
            v[1] = v[0];
            v[0] = aux;
            }
        }
    }
```

Ordenamiento Quick-sort

El método se basa en dividir los n elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una partición izquierda, un elemento central denominad o pivote o elemento de partición, y una partición derecha.

La partición o división se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha).

Las dos sublistas se ordenan entonces independientemente.

Para dividir la lista en particiones (sublistas) se elige uno de los elementos de la lista y se utiliza como pivote o elemento de partición.

Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede seleccionar cualquier elemento de la lista como pivote, por ejemplo, el primer elemento de la lista.

Si la lista tiene algún orden parcial conocido, se puede tomar otra decisión para el pivote.

Idealmente, el pivote se debe elegir de modo que se divida la lista **exactamente por la mitad**, de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones «pobres» de pivotes.

Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo quicksort.

La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista.

La primera etapa de quicksort es la división o «particionado» recursivo de la lista, hasta que todas las sublistas constan de sólo un elemento.

Veamos un ejemplo.

8	1	4	9	6	3	5	2	7	0
0	1	2	3	4	5	6	7	8	9

La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar el sistema para situar en la sublista izquierda todos los elementos menores que el pivote y en la sublista derecha todos los elementos mayores que el pivote. Supongamos que todos los elementos de la lista son distintos, aunque será preciso tener en cuenta los casos en que existan elementos idénticos. En el Ejemplo se elige como pivote el elemento central de la lista actual.

8	1	4	9	6	3	5	2	7	0

Pivote (elemento central): `6`

La etapa 2 requiere mover todos los elementos menores al pivote a la parte izquierda del array y los elementos mayores a la parte derecha. Para ello se recorre la lista de izquierda a derecha utilizando un índice i que se inicializa en la posición más baja (inferior) buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un índice j inicializado en la posición más alta (superior).

El índice i se detiene en el elemento 8 (mayor que el pivote) y el índice j se detiene en el elemento o (menor que el pivote).

8	1	4	9	**6**	3	5	2	7	0
i →								←	j

ahora se intercambian 8 y o para que estos dos elementos se sitúen correctamente en cada sublista; y se incrementa el índice i, y se decrementa en índice j para seguir los intercambios.

0	1	4	9	**6**	3	5	2	7	8
	i →						←	j	

a medida que el algoritmo continúa, i se detiene em el elemento mayor, 9, y j se detiene en el elemento 2.

0	1	4	9	**6**	3	5	2	7	8
			i →			←	j		

es intercambian los elementos mientras que i y j no se crucen. En caso contrario se detiene este bucle. En el caso anterior se intercambian 9 y 2.

0	1	4	2	**6**	3	5	9	7	8
				i	←	j			

Continúa la exploración y ahora el contador i se detiene en el elemento `6` (que es el pivote) y el índice `j` se detiene en el elemento menor `5`.

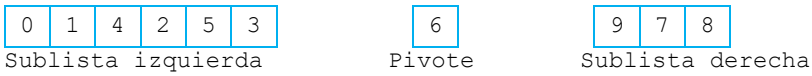
0	1	4	9	5	3	**6**	2	7	8
					ij				

los índices tienen actualmente los valores `i=5` y `j=5`. Continua la exploración hasta que `i>j`, terminando con `i=6` y `j=5`.

0	1	4	9	5	3	**6**	2	7	8
					j	i			

en esta posición los índices `i` y `j` han cruzado posiciones en el array y en este caso se detiene la búsqueda y no se realiza ningún intercambio ya que el elemento al que accede `j` esta ya

correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:



Veamos el código del ejemplo:

```
void quicksort(int a[], int primero, int ultimo){
    int contint=0, pivote, i, j, central, tmp,c;

    central = (primero + ultimo)/2;
    pivote = a[central];
    i = primero;
    j = ultimo;
    do {
        while (a[i] < pivote)
            i++;
        while (a[j] > pivote)
            j--;
        if (i<=j){
            tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
            i++;
            j--;
            }
        }while (i <= j);
    if (primero < j)
        quicksort(a, primero, j);
    if (i < ultimo)
        quicksort(a, i, ultimo);
    }
```

Algoritmos de búsqueda

En este curso veremos dos algoritmos, uno bien sencillo y otro más complejo: la búsqueda secuencial y la búsqueda binaria.

Búsqueda secuencial

Este algoritmo es bien sencillo. Simplemente consta de un ciclo `for` en el que se recorre el vector uno a uno en busca del elemento deseado.
Con esta breve descripción, el algoritmo queda de la siguiente manera:

```
int busquedaSecuencial(int val, int v[], int dim){
    int i;

    for (i=0 ; i<dim ; i++) {
        if (v[i]==val)
            return i;
        else
            return -1;
        }
    }
```

En el código anterior, la función devuelve la posición en la que se encontró el valor por primera vez, es decir, la primera ocurrencia.

Búsqueda binaria

Este algoritmo es bastante más complejo que el anterior. De hecho, es recursivo y tiene un requerimiento importante: **el vector debe estar ordenado**.

La idea del algoritmo es la siguiente:

- La función recibe el `vector`, el `valor` a buscar y dos `limites` (`límite inferior` que será cero, y `límite superior` que será el tamaño del vector).
- Usando los límites, se calcula la `posición central` y se compara dicho valor con el pasado por parámetros:
  - Si es igual, la función devuelve la posición y se termina el algoritmo.
  - Si el valor buscado es menor al de la `posición central`, se llama recursivamente a la función de búsqueda, pero con el `limite superior` actualizado a `posición central-1`.
  - Por el contrario, si el valor buscado es mayor al de la `posición central`, se llama recursivamente a la función de búsqueda, pero con el `limite inferior` actualizado a `posición central+1`.
- Las comparaciones y llamadas recursivas se dan mientras que el `limite superior` sea mayor al `limite inferior`. Cuando esta condición ya no se da más (es decir que se cruzan los limites),

Veamos un ejemplo:

2	6	8	9	14	34	44	66	69	78
0	1	2	3	4	5	6	7	8	9

Teniendo el vector de arriba, busquemos la posición del valor 66.

- `limInf` inicia en `0`, `limSup` en `9` y val en `66`.
- Calculamos la `posMedia`, que nos da 4
- Comparamos su contenido con `val`. Dado que `14 < 66`, actualizamos el `limInf` al valor de la posición `5`.
- Volvemos a iniciar la búsqueda, ahora con `limInf` inicia en `5`, `limSup` en `9` y val en `66`.
- Calculamos la `posMedia`, que nos da `7`.
- Comparamos su contenido con `val`. Dado que `66 = 66`, se devuelve el valor.

En el código, el algoritmo queda así:

```
int busquedaBinaria( const int b[], int claveDeBusqueda, int bajo, int alto, int
d ){
   int central;
   while ( bajo <= alto ) {
     central = ( bajo + alto ) / 2;
     if ( claveDeBusqueda == b[ central ] ) {
         return central;
      }
      else if ( claveDeBusqueda < b[ central ] ) {
         alto = central - 1;
      }
      else {
         bajo = central + 1;
      }
   }
   return -1;
}
```