NOTA AL ALUMNO

Este documento es una recopilación más o menos completa de los apuntes que utilicé para dar clases desde los años 2006 al 2010 en la asignatura Informática II del ciclo común de Ingeniería de la UCA. No tiene el tratamiento formal de los temas que merece, para ello existen varios libros muy buenos. Más bien, es una transcripción prolija del material dictado en aula, con los mismos gráficos horribles (hechos en Paint ahora) que suelo dibujar en el pizarrón.

Espero que les sea de utilidad para la cursada de la materia, y por ahí hasta sirve para aprender a programar!

Ing. Daniel Solmirano

Oct. 2010

TABLA DE CONTENIDOS

1- ARREGLOS MULTIDIMENSIONALES

INTRODUCCIÓN

Sabemos que un arreglo es una estructura de datos compuesta por una secuencia de tamaño fijo de elementos del mismo tipo. Dicha secuencia posee un nombre (el nombre del arreglo), y cada elemento puede ser accedido a través de un subíndice. El rango inferior del subíndice es 0 y el superior, N -1 (siendo N el tamaño del arreglo).

En la memoria de la computadora, los elementos de un arreglo ocupan posiciones contiguas de la memoria. Si tenemos un arreglo de enteros, p. ej.: `int a[100];` y a[0] se encuentra en la posición 0x200, entonces a[1] se encuentra en 0x204, a[2] en 0x208, y así sucesivamente (ya que cada entero ocupa 4 bytes).

Los arreglos multidimensionales son estructuras muy similares a los arreglos, pero que utilizan más de un subíndice para referirse a sus elementos. Cada subíndice agrega una "dimensión" al arreglo. Por ejemplo, podemos declarar un arreglo multidimensional de 3 x 3 (totalizando 9 elementos) de la siguiente manera:

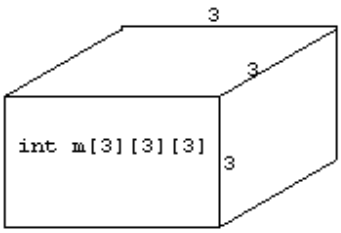
```
int m[3][3];
```

Su representación gráfica será:

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]

En el caso de los arreglos de 2 dimensiones, el nombre más común que se les da es el de *matrices* (dado su parecido con las matrices de álgebra). Podemos pensar que el primer subíndice define las filas, y el segundo subíndice, las columnas dentro de cada fila.

Un arreglo de 3 dimensiones sería similar a un cubo, con una primera dimensión que defina el "alto", una segunda de "ancho", y una tercera de "profundidad".



En un arreglo multidimensional no es posible tener filas de diferente longitud, cada una de ellas será exactamente de la longitud definida por la dimensión en el momento de su declaración.

Los arreglos multidimensionales son utilizados para representar datos que en la vida real se definen en tablas de múltiples entradas. Algunos ejemplos:

- El pago de expensas de un edificio (donde cada fila puede representar un departamento, las columnas los meses del año, y en cada "celda" podemos ingresar un pago mensual).
- Una lista de asistencia al curso (donde cada fila representa un alumno, cada columna la asistencia a una clase en particular, y en cada celda podemos ingresar un 1 si el alumno está presente, o 0 si está ausente).

USO DE ARREGLOS MULTIDIMENSIONALES

Debemos distinguir los diferentes momentos en que hacemos referencias a variables del tipo arreglo multidimensional:

Declaración: se declara el tipo de dato, el nombre del arreglo y luego cada una de las dimensiones utilizando corchetes. Dentro de los corchetes se indica el tamaño de esa dimensión en particular. De la misma manera que con los arreglos unidimensionales, el tamaño de cada dimensión debe ser conocido de antemano (ya sea "hardcodeando" el número en la declaración, o utilizando una constante #define).

Ejemplos:

```
double temperaturas_del_anio [31][12];
char nombres_equipo [40][11];
```

Declaración + inicialización: podemos declarar e inicializar un arreglo, es decir, lograr que el valor de cada una de sus "variables" sean conocidos desde el momento en que la variable se declara. Para ello utilizamos la notación de llaves. Al igual que en el caso de los arreglos unidimensionales, si declaramos menos datos de los que componen el arreglo, el resto se rellenará de ceros.

```
// Esto declara un arreglo de 8 x 8 con las 2 primeras filas
// completas, y las restantes con 0.
int tablero[8][8] = {
{2, 3, 4, 5, 10, 4, 3, 2},
{1, 1, 1, 1,  1, 1, 1, 1}};
```

Acceso a las variables: Se realiza de la misma manera que con los arreglos unidimensionales. Cada variable se referencia por tantos índices como dimensiones tenga el arreglo. Es posible referenciar un "sub-arreglo" dentro del arreglo multidimensional utilizando menos índices que la cantidad de dimensiones del arreglo. Esto puede ser útil, por ejemplo, para referirse a filas completas dentro de una matriz.

Ejemplo:

```
// Este programa escribe una "cruz" en una matriz de 10 x 10 y luego la imprime.
#include <stdio.h>
#include <stdlib.h>

void imprime_linea(char[], int);

int main()
{
    char figura[10][10];
    int f, c;
    for (f = 0; f < 10; f++)
    {
        for (c = 0; c < 10; c++)
        {
            if (f == c || f == 0 || c == 0 || c == 9 || f == 9 || f  + c == 9)
                figura [f][c] = '*'; //Seteamos una variable accediendo con sus 2 indices
            else
                figura [f][c] = ' ';
        }
    }
    for ( f = 0; f < 10; f++)
        imprime_linea(figura[f] , 10); //Referenciamos una fila completa de la matriz
    return 0;
}

void imprime_linea(char a[], int largo)
{
    int i;
    for (i = 0; i < largo; i++)
    {
        printf("%c", a[i]);
    }
    printf("\n");
}
```

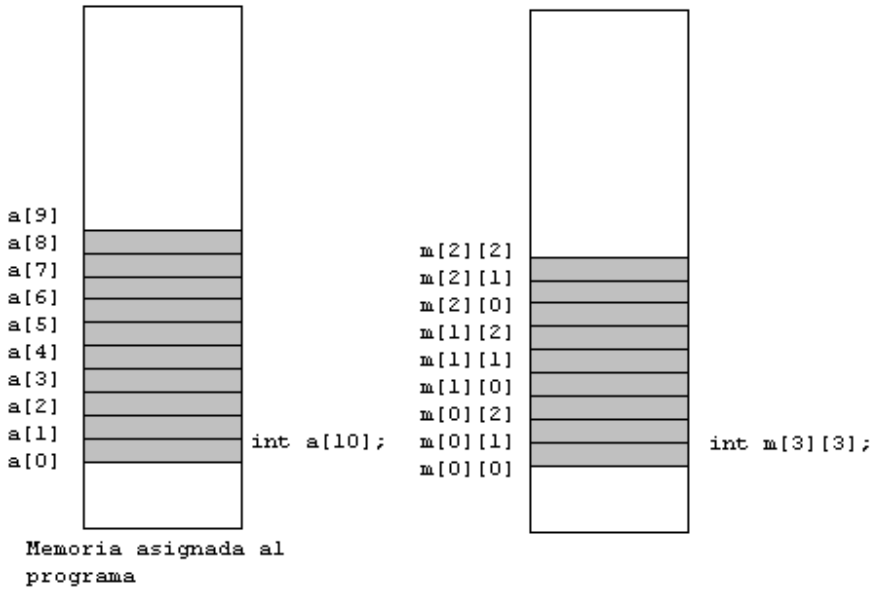
Pasaje por parámetro a funciones: Al momento de utilizar un arreglo multidimensional como parámetro de una función, debe tenerse en cuenta que en la declaración de la función es necesario indicar las primeras N-1 dimensiones del mismo. ¿Qué quiere decir esto? Por ejemplo, si declaramos una función para imprimir un arreglo de 3 x 3 enteros, entonces su prototipo será:

```
void imprimir_matriz(int [][3]);
```

¿Por qué ocurre esto? Para el compilador, un arreglo multidimensional se “ve” en memoria de la misma manera que un arreglo unidimensional:

1) En memoria, un arreglo unidimensional se ve como una secuencia de variables del mismo tipo, una después de la otra. La memoria es lineal, cada posición tiene un número asignado y es seguida por otra. El programa reserva una cantidad X de bytes para cada variable, esa cantidad depende del tipo de dato de la variable. Así, un arreglo de 10 enteros ocupa 10 x 4 bytes = 40 bytes en memoria.

2) ¿Qué ocurre con los arreglos multidimensionales? Dado que la memoria es lineal (1 dimensión) y los mismos tienen más de 1 dimensión, lo que hace el compilador es “aplanar” las dimensiones, colocando una fila inmediatamente luego de la otra, tal como se indica en la figura:



Cuando referenciamos un arreglo multidimensional dentro de una función, el compilador calcula la “posición real” de memoria de cada variable multiplicando la longitud de la fila x la cantidad de filas y sumando el desplazamiento de columna (en el caso de más de 2 dimensiones, el cálculo es análogo, agregando cada una de las dimensiones):

Posición real = longitud de fila x número de fila + columna.

En el caso de la figura, si queremos calcular la posición real de la variable m[2][1], veremos que su posición dentro del arreglo es 2 x 3 + 1 = 7 (lo cual puede comprobarse en la figura contando en 0 desde la posición inicial y llegando al índice 7).

Para poder hacer este cálculo, el compilador necesita conocer de antemano el valor “3”, es decir el tamaño de las dimensiones sobre la cual se está trabajando.

2- OPERACIONES A NIVEL DE BIT

Las operaciones a nivel de bit nos permiten manipular los contenidos de variables de tipo enteras o char, bit por bit. Esto significa que podemos alterar los contenidos de una variable pensando en ella como una secuencia de bits en vez de entenderla como una variable del tipo de dato correspondiente. Por ejemplo:

```
char letra = 'a';
```

La variable letra contiene el caracter 'a', pero también podemos verla como la secuencia de bits

0	1	1	0	0	0	0	1

Podemos pensar esta secuencia como un arreglo, donde sólo podemos ingresar ceros y unos. Pero, al contrario de un arreglo regular, no podemos acceder en forma directa a una posición a través de su subíndice. Para poder manipular sus contenidos es que contamos con los operadores a nivel de bit. Hay dos tipos de operadores: los operadores lógicos, y los operadores de desplazamiento. Los primeros permiten operar una o dos variables del mismo tipo, bit por bit. Los segundos permiten "mover" los bits de una variable hacia la izquierda o la derecha.

OPERADORES LÓGICOS BINARIOS

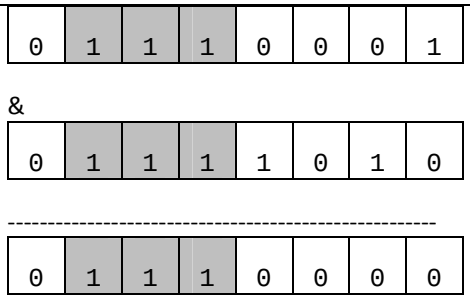
Los operadores lógicos binarios son 3

& : AND a nivel bit **|** : OR inclusivo a nivel bit **^**: OR exclusivo (XOR) a nivel bit

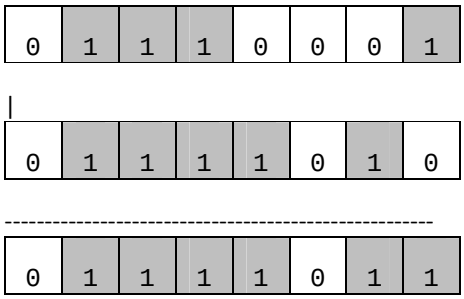
- El operador & recibe dos variables del mismo tipo (enteros o char) y aplica la operación AND bit por bit, retorna otro valor que contiene bit 1 en las columnas donde ambas variables tienen 1, y 0 en otro caso
- El operador | aplica la operación OR bit por bit, retorna otro valor que contiene bit 1 en las columnas donde alguna de las variables tiene un 1, y 0 en aquellas donde ambas variables tengan 0.
- El operador ^ aplica la operación XOR bit por bit, retorna otro valor que contiene bit 1 en aquellas columnas donde una variable presenta un 1, y la otra un 0. Si aparecen dos 1 o dos 2, el resultado de dicha columna será 0.

Veamos un ejemplo:

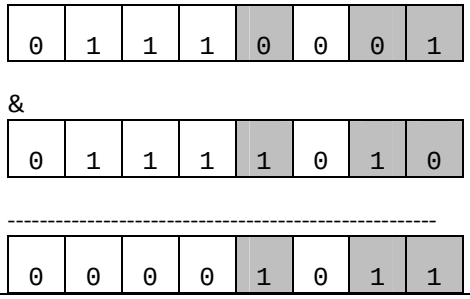
```
char l1 = 113; // letra 'q'
char l2 = 122; // letra 'z'


char l3 = l1 & l2;
```




```
l3 = l1 | l2;
```




```
l3 = l1 ^ l2;
```

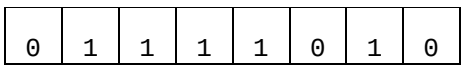


En los ejemplos, pensar “qué letra” representa la variable l3 luego de realizar las operaciones AND, OR y XOR, no tiene sentido, ya que estamos considerando su representación interna bit por bit, y no el valor ASCII resultante.

COMPLEMENTO A NIVEL DE BIT

Finalmente, existe un cuarto operador que no es binario sino unario. Dada una variable, retorna el resultado de negar cada uno de sus bits. Este operador es ~

```
char l1 = 122;
char l2 = ~l1;
```



	---------- 1 0 0 0 0 1 0 1

OPERADORES DE DESPLAZAMIENTO

Los operadores de desplazamiento son 2:

- << : shift izquierdo a nivel de bits
- >> : shift derecho a nivel de bits

Ambos operadores mueven los bits del operando de la izquierda, tantas veces como indique el operando de la derecha (dicho valor debe ser positivo). El operador de shift izquierdo "rellena" con ceros los bits desplazados. El operador de shift derecho "rellena" los bits desplazados con el contenido del primer bit para así preservar el signo de la variable.

`char x = 65;` `char y = x << 3;`	0 1 0 0 0 0 0 1 << 3 ---------- 0 0 0 0 1 **0 0 0**

En este caso los 3 bits de la izquierda se "pierden" debido al desplazamiento hacia la izquierda.

`char x = 97;` `x = x >> 2;`	0 1 1 0 0 0 0 1 >> 2 ---------- X X 0 1 1 **0 0 0**

En este caso los 2 bits de la derecha se "pierden" debido al desplazamiento. Ponemos una "x" en vez de un número en los primeros dos bits, ya que usualmente no podemos afirmar qué valor quedó copiado.

Podemos pensar que el operador de desplazamiento izquierdo *x* << *n* multiplica x por 2^n. Análogamente, *x>>n* , cuando la variable tiene signo positivo, realiza una división entera por 2.

MÁSCARAS A NIVEL BIT

Debido a que los bits de una variable sólo pueden ser accedidos a través de los operadores descriptos anteriormente, es necesario realizar algunos pasos adicionales si queremos modificar un cierto bit en una variable sin cambiar el resto. En este sentido, es normal "encender" bits (ponerlos en 1), siendo la operación contraria un poco menos frecuente.

Usaremos máscaras para "encender" bits en una variable, o para detectar si un bit tiene un valor dado. Una máscara es una constante en la cual sólo algunos bits están "prendidos" o puestos en 1, y los demás se mantienen en 0. Combinando una máscara con una variable, vamos a poder encender o apagar bits, y detectar qué bits están en 1.

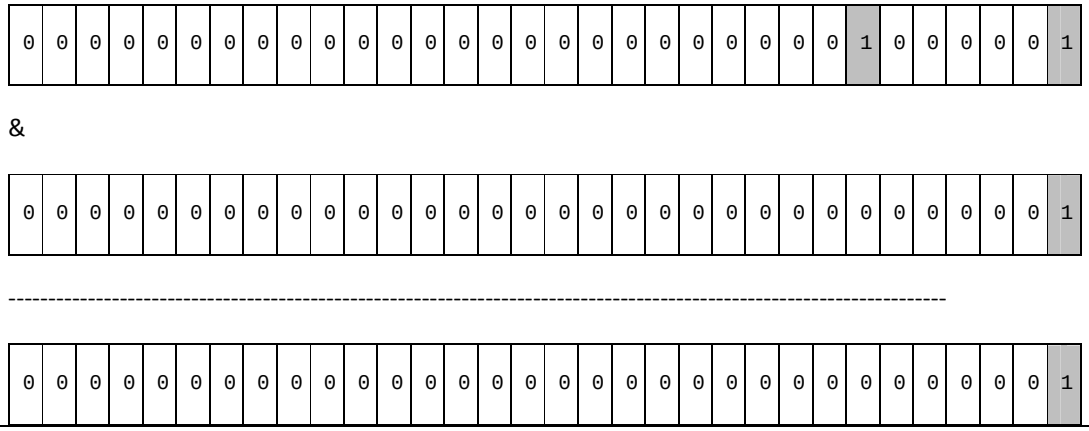
Para poder poner un cierto bit en 1 en una variable, lo que haremos es realizar un OR binario con una máscara que tenga ese bit en 1. Por ejemplo:

```
#define MASK1  1
#define MASK2 (1<<12)

int x = 32;
int y = 123;

x = x | MASK1;

y = y | MASK2;
```

La máscara no tiene por qué estar compuesta únicamente por un solo bit a 1; es posible tener máscaras con mayor cantidad de bits a 1.

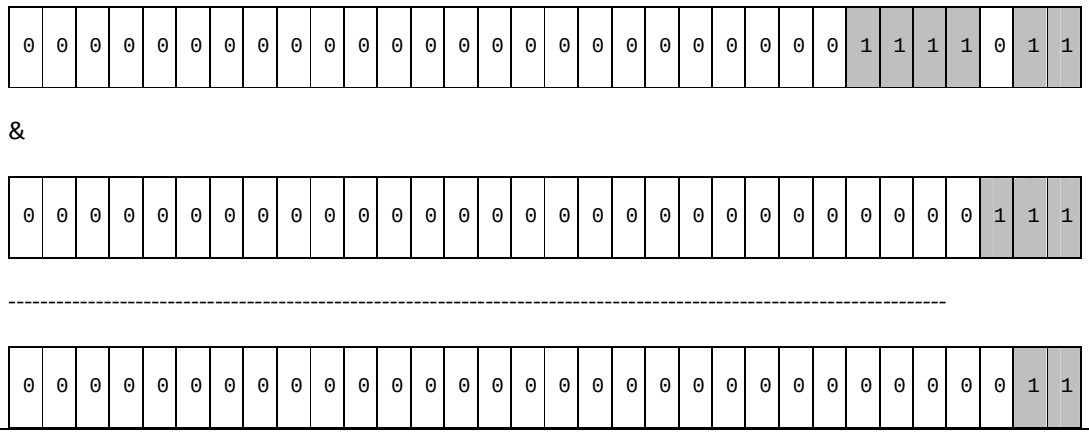
Para poder saber si una variable tiene un cierto bit en 1, lo que haremos es realizar un AND binario con una máscara que tenga ese bit en 1. Por ejemplo:

```
#define MASK1  1
int x = 65;
if (x & MASK1)
   printf("La variable
x contiene un 1 en el
primer bit\n");
```



Análogamente, es posible usar máscaras compuestas para descubrir varios bits al mismo tiempo.

```
#define MASK1  7
int x = 123;
int y = x & MASK1;
printf("%d", y);
```



Finalmente, es posible alterar el contenido de un bit (llevarlo a 1 si está en 0, y viceversa), aplicando un XOR con una máscara que tenga dicho bit a 1. Si en la posición correspondiente a la máscara había un 1, el XOR lo transformará en un 0. Si, al contrario, había un 0, 0 XOR 1= 1.

3- STRUCTS (ESTRUCTURAS DE DATOS PROPIAMENTE DICHAS)

INTRODUCCIÓN

Al hablar de arreglos y matrices introdujimos el concepto de "estructura de datos". Una estructura de datos es un conjunto de datos agrupados por algún motivo, con una cierta organización interna. Hay diferentes motivos por el cual es conveniente trabajar varios valores de datos como si fueran uno solo:

- A nivel operativo, simplificar la manipulación de valores, trabajando con un solo valor compuesto en vez de varios valores simples.
- A nivel conceptual, agrupar en una única variable los diferentes datos que se refieren a una misma idea.

Arreglos y matrices son ejemplos de estructuras de datos incorporadas en el lenguaje C. Con ellas podemos modelar secuencias de datos del mismo tipo (tanto lineales como n-dimensionales). Teóricamente sería posible utilizar arreglos para manipular entidades conceptuales más complejas, acordando, p.ej., que cada posición se refiere a algún dato del cual queremos hablar, y utilizando índices correspondientes en diferentes arreglos para "linkear" datos que están relacionados. Esta manera de trabajar es compleja y tiende a producir errores rápidamente.

El lenguaje C (y la mayoría de los denominados lenguajes "estructurados") nos da la posibilidad de definir nuestras propias estructuras de datos. Al hacerlo podemos representar abstracciones conceptuales de datos (eventualmente definir nuestros propios tipos de datos), que nos permiten trabajar mucho más fácilmente con representaciones complejas de la realidad.

Veamos un ejemplo:

Para representar un punto en el espacio R^3, utilizamos sus 3 coordenadas en los ejes cartesianos:

P1 = (2, 3, 0) P2 = (1,-2, 5)

¿Qué opciones conocemos para representar en el lenguaje C un punto en R^3?

- Utilizar 3 variables (`double x, y, z;`). Si necesitamos definir 2 puntos , utilizamos 6 variables, y así sucesivamente.
- Utilizar un arreglo de longitud 3 y convenir que la posición 0 corresponde a x, la 1 a y y la 2 a z (`double punto[3]`). Si necesitamos definir múltiples puntos, podemos usar o bien una matriz, o podemos usar un arreglo de longitud 3*N, donde las 3 primeras posiciones son para el primer punto, las otras 3 para el siguiente, etc.

Ninguno de las dos soluciones representa fielmente lo que queremos modelar: que un punto en el espacio se representa mediante 3 coordenadas, llamadas x, y y z.

LA PALABRA RESERVADA STRUCT

La palabra reservada **`struct`** es utilizada para definir nuestras propias estructuras de datos. Cuando definimos una estructura de datos, estamos indicándole al lenguaje cómo se organizarán los datos en memoria cuando utilicemos dichas estructuras. La sintaxis es:

```
struct <nombre>
{
   [miembros];
};
```

Retomando el ejemplo anterior:

```
struct s_punto
{
      double x;
      double y;
      double z;
};
```

El código escrito más arriba sirve para definir una estructura de datos llamada `struct s_punto`. Dicha estructura tiene 3 miembros o componentes del tipo `double`, llamados `x`, `y` y `z`. Cuando hagamos referencia a una variable de tipo `struct s_punto`, estaremos hablando de una variable que internamente contiene esos 3 componentes. Análogamente al uso que hacemos de subíndices cuando trabajamos con arreglos, aquí utilizaremos los nombres de los componentes para referirnos a cada parte individual de una variable de tipo `struct s_punto`. Para acceder a cada componente utilizamos el operador `.` (el punto).

```
#include <stdio.h>

struct s_punto
{
      double x;
      double y;
      double z;
};

int main()
{
   struct s_punto p1, p2;
   p1.x = 2.1;
   p1.y = 3;
   p1.z = 0;
```

```
    scanf("%lf", &p2.x);
    scanf("%lf", &p2.y);
    scanf("%lf", &p2.z);

    p1.x = p1.x + p2.x;
    p1.y = p1.y + p2.y;
    p1.z = p1.z + p2.z;

    printf("\n P1(x,y,z) = (%.2lf, %.2lf, %.2lf)\n", p1.x, p1.y, p1.z);
    return 0;
}
```

Las estructuras se definen luego de las directivas de preprocesador pero antes de los prototipos de funciones, de manera que el compilador las reconozca en el resto del programa. Cuando definimos una estructura, no estamos ocupando ningún espacio en memoria, simplemente estamos indicando que desde ese lugar en adelante, cada vez que hagamos referencia a un valor del tipo del struct, en realidad estamos hablando de la totalidad de los datos que representa. Luego, cada vez que declaremos una variable del tipo del struct, ocuparemos tanta memoria como miembros tenga el struct.

En el ejemplo dado anteriormente vemos que para poder "llenar" una variable del `tipo struct s_punto`, debemos referenciar sus componentes uno por uno (ya que la función `scanf()` no sabría como completar los 3 datos simultáneamente). El operador punto nos permite acceder a las componentes internas de una variable struct. En el caso de la impresión utilizando printf(), ocurre lo mismo, debemos trabajar con la variable struct componente por componente.

Al momento de definir una estructura de datos, no estamos limitados a utilizar componentes de tipos de datos simples (los definidos por el lenguaje): podemos utilizar arreglos e incluso otras estructuras.

Por ejemplo, si queremos representar empleados de una empresa, de cada uno de los cuales nos importa su legajo, nombre, su apellido, su fecha de nacimiento y su fecha de ingreso a la compañía, podemos utilizar las siguientes estructuras:

```
struct s_fecha
{
    int dia, mes, anio;
};

struct s_empleado
{
    int legajo;
    char nombre[40];
    char apellido[50];
    struct s_fecha fecha_nac;
    struct s_fecha fecha_ing;
};
```

PASAJE DE PARÁMETROS Y DEVOLUCIÓN EN FUNCIONES

Es posible tanto pasar estructuras como parámetros a funciones, como retornar estructuras como resultado de invocación a una función. Debe tenerse en cuenta que cuando pasamos una estructura como parámetro a una función, la misma se comporta como si fuera una variable de tipo de dato simple; es decir, que se copiarán los datos al espacio de memoria de la función, y la variable original no se verá modificada si llegamos a cambiar algo dentro de la función.

Volviendo al ejemplo del punto, podríamos definir las siguientes funciones:

- Una función que retorne un punto cargado por el usuario.
- Una función que imprima los contenidos de un punto por pantalla.

```
#include <stdio.h>

struct s_punto
{
      double x;
      double y;
      double z;
};

struct s_punto cargar_punto(void);
void imprimir_punto(struct s_punto);

int main()
{
   struct s_punto p1, p2;
   p1 = cargar_punto();
   p2 = cargar_punto();

   p1.x = p1.x + p2.x;
   p1.y = p1.y + p2.y;
   p1.z = p1.z + p2.z;

   printf("\n P1");
   imprimir_punto(p1);
   return 0;
}

struct s_punto cargar_punto(void)
{
      struct s_punto resultado = { 0, 0, 0 };
      scanf("%lf", &resultado.x);
      scanf("%lf", &resultado.y);
      scanf("%lf", &resultado.z);
      return resultado;
}

void imprimir_punto(struct s_punto punto)
{
      printf("(x,y,z) = (%.2lf, %.2lf, %.2lf)\n", punto.x, punto.y, punto.z);
}
```

ALIAS DE TIPOS (TYPEDEF)

El lenguaje C provee un mecanismo que nos permite definir “alias de tipos”. Un alias de tipo no es otra cosa que un sinónimo para referirnos a un tipo ya existente. Podemos utilizarlo para referirnos más brevemente a tipos de datos struct. La sintaxis es

```
typedef <un tipo de datos> <su alias>;
```

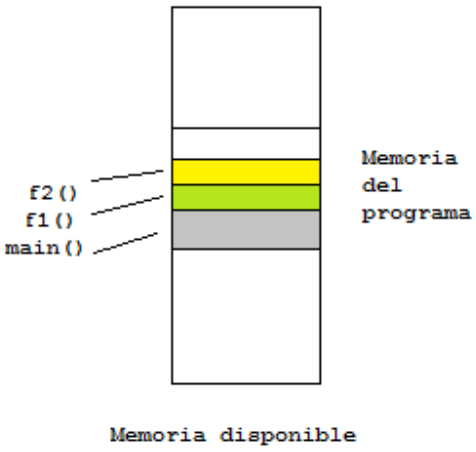
```
#include <stdio.h>

struct s_punto
{
      double x;
      double y;
      double z;
};

typedef struct s_punto Punto;

Punto cargar_punto(void);
void imprimir_punto(Punto);

int main()
{
   Punto p1, p2;
   p1 = cargar_punto();
   p2 = cargar_punto();
  …
```

4- PUNTEROS

Los conceptos de punteros y memoria dinámica están íntimamente relacionados con la estructura en memoria de un programa ejecutándose. Para poder entenderlos, debemos conocer algo respecto a cómo un programa se ejecuta en la computadora.

EJECUCIÓN DE UN PROGRAMA

Una vez que un programa es compilado por el compilador de nuestra elección, se obtiene código máquina, capaz de ser cargado y ejecutado por el Sistema Operativo. Es él quien se encarga de gestionar la forma en que el proceso (proceso = programa en ejecución) va a poder acceder a los diferentes recursos de hardware. Para gestionar el acceso a memoria asignada al proceso, se la divide en 3 zonas: la zona de datos (para variables globales y constantes), el stack o memoria estática, y el heap o memoria dinámica. El stack se denomina así porque su comportamiento es similar al de una pila: cada nuevo requerimiento de memoria se ubica por encima del anterior (se apila), y al finalizar, se quita o desapila. Cuando el programa es cargado, el S.O. busca una zona de memoria libre para este nuevo proceso. Cada función que se ejecuta (empezando por el main) se carga en el stack. Ya que para cada función hemos declarado las variables y su tipo, el S.O. sabe cuánto espacio en memoria se necesita. Cuando desde el main se invoca una función, se reserva memoria en el stack, tanto para los parámetros como para las variables de la función, y también para el valor de retorno de la función. Si esta función llama a otra, se sigue apilando memoria, y así sucesivamente. Cuando la función retorna, el espacio en memoria se desapila (en la práctica, lo único que se hace es “bajar” el cursor que indica el tope de la pila a la posición en que se encontraba antes de ejecutar la función).

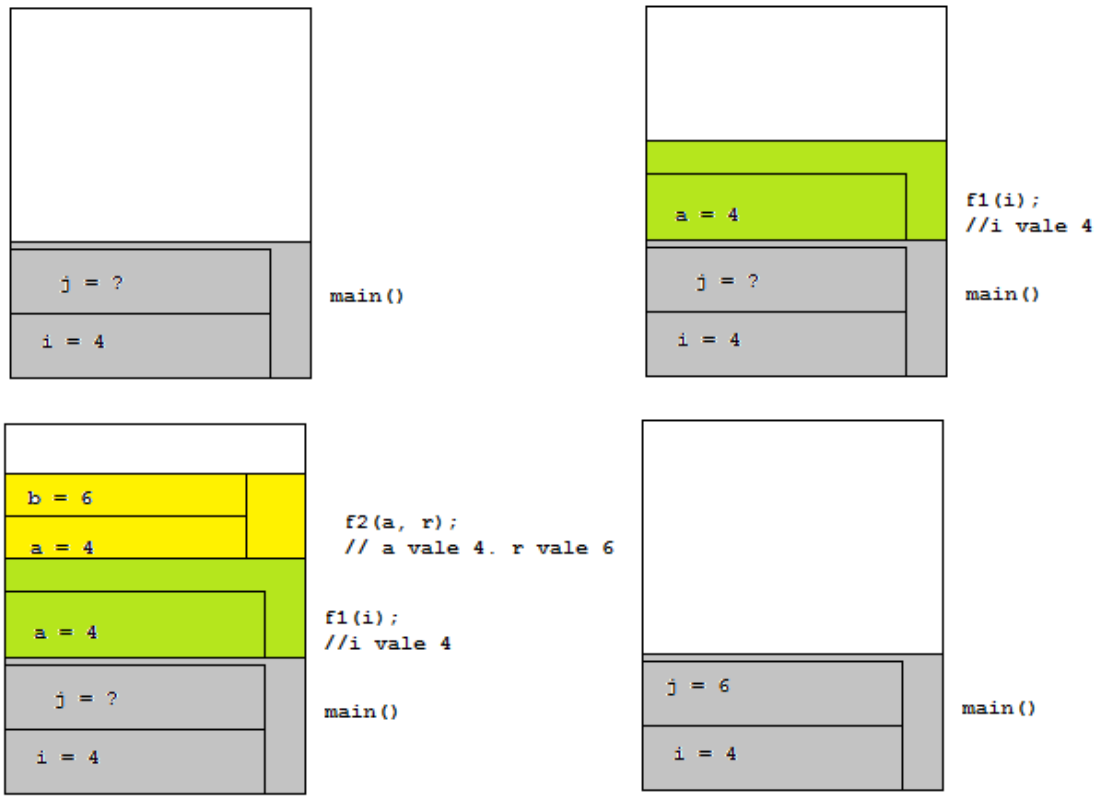



```
int main()
{
   int i = 4, j;
   j = f1(i);
   return 0;
}

int f1(int a)
{
  int r = a + 2;
  f2 (a, r);
  return r;
}

void f2(int a, b)
{
   printf("%d %d", a, b);
}
```

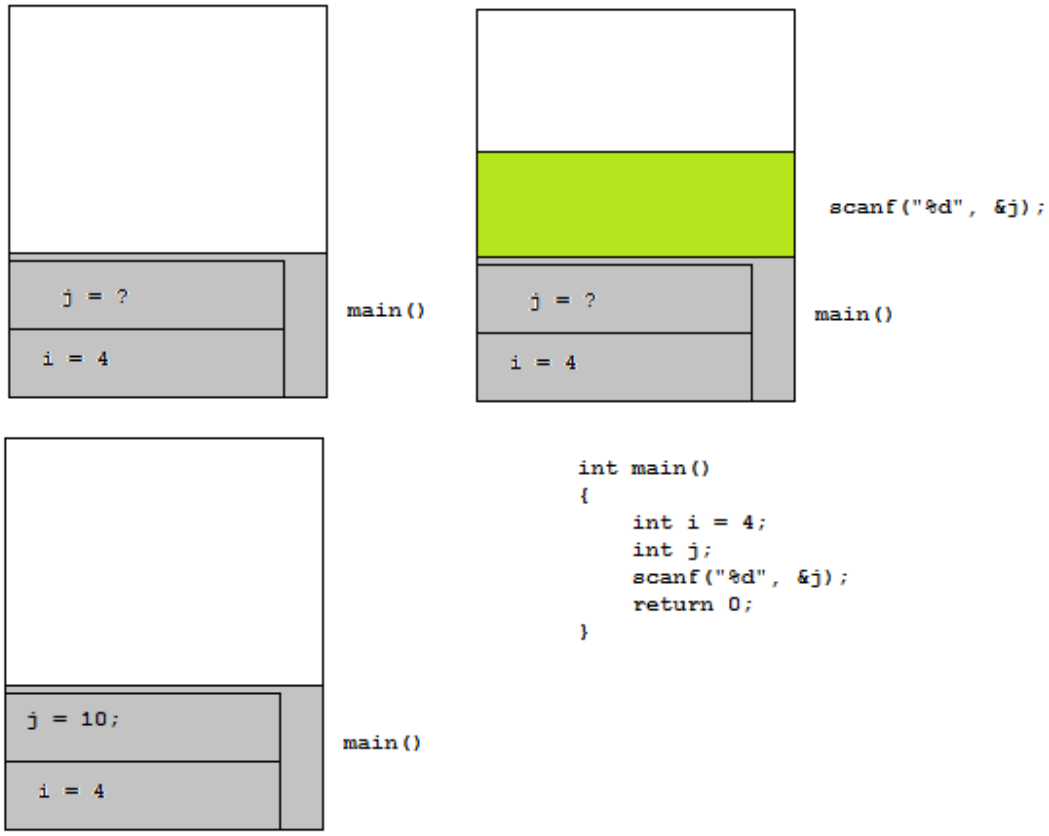
Por supuesto el stack tiene un espacio finito: es posible generar, mediante muchas llamadas sucesivas, una condición denominada “stack overflow” o sobrepasamiento de pila, en la cual el proceso finaliza abruptamente.

PASAJE DE PARÁMETROS A FUNCIONES

Cuando se invoca una función, además del espacio necesario para las variables locales de la función, se reserva espacio para los parámetros con los que la función es invocada. Los valores con los que la función se ejecuta son copiados desde la función llamadora. Dentro del ámbito de ejecución de la función, es posible reasignar valores a los parámetros, pero estos cambios siempre son locales, ya que los parámetros fueron copiados.



Dicho esto, está claro que una función no puede modificar el valor de las variables que recibe. Sin embargo, sabemos que esto no es necesariamente cierto. Aprendimos anteriormente que si pasamos un arreglo por parámetro, sus valores pueden ser cambiados. También es posible invocar la función scanf() y lograr el mismo efecto:



¿Qué ocurre en estos casos? Si nos fijamos en el ejemplo, estamos pasando 2 parámetros a scanf(): una cadena de texto y una variable entera "j". Pero la variable j tiene un signo & delante. Este operador (llamado "operador de referencia") nos permite obtener un **puntero** a la variable j.

PUNTEROS

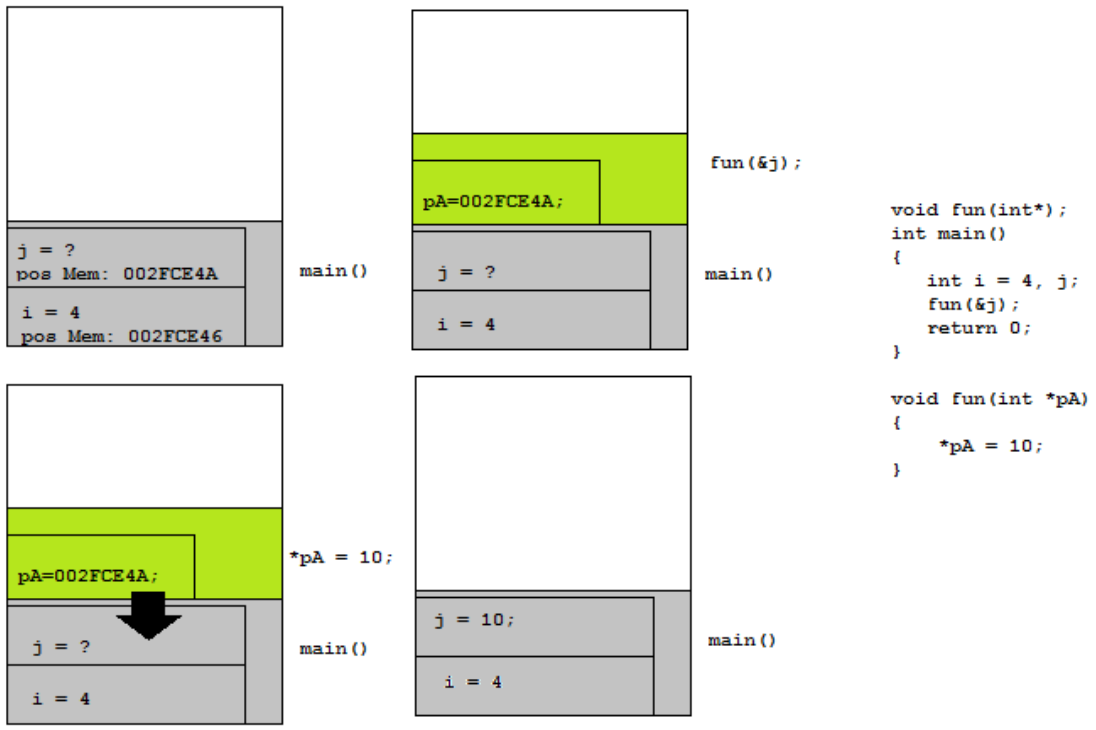
¿Qué es un puntero? Es una referencia a una variable, o dicho de otra manera, una variable que contiene un valor: la dirección de memoria de otra variable.

Si pensamos que una variable es una "caja" donde podemos almacenar información, un puntero puede pensarse como un "vale" que dice "vale por esa caja". El "vale" contiene información, pero su información no es la variable en sí, sino dónde se encuentra ésta.

Al pasar un puntero como parámetro a una función, obtenemos un beneficio: le estamos "dando permiso" para que cambie los contenidos de la variable original. Esto se logra realizando la operación complementaria

a referenciar: desferenciar o indireccionar un puntero[1]. ¿Qué significa desreferenciar un puntero? Básicamente es realizar la operación inversa a referenciar: tomamos un puntero, le aplicamos el operador apropiado, y obtenemos la variable a la que hace referencia. Al enviar un puntero como parámetro a una función, y aún cuando se copia el contenido del puntero, al desreferenciarlo ganamos acceso a la variable original. Luego, podemos modificar sus contenidos, aunque la variable se encuentre en el espacio de memoria que no es el propio de la función.

Veamos un ejemplo:



En este ejemplo podemos identificar lo siguiente:

- Cada variable se encuentra en una posición de memoria.
- Al invocar la función fun(), copiamos el contenido del puntero a un nuevo espacio en memoria, propio de la función. El contenido del puntero es la dirección de memoria donde se encuentra la variable j.
- Declaración de variables: ll tipo de dato para una variable del tipo puntero se escribe *tipo_de_dato_del_puntero * nombre_variable;* , si bien todos los punteros se refieren a posiciones

[1] Ninguna de estas palabras existe en español pero a los efectos prácticos podemos usarlas para indicar esta operación.

de memoria (por lo tanto un puntero a char o un puntero a int "miden" lo mismo), es necesario aclarar el tipo de dato de la variable a la que se hace referencia.

- Para usar un puntero debemos "sacarlo" de una variable ya existente, para eso usamos el operador de referencia. Si declaramos un puntero y no le asignamos nada, ese puntero hará referencia a posiciones de memoria fuera de la memoria que el programa tiene asignado, y probablemente fallará su ejecución.
- Para obtener la variable original a la que hace referencia un puntero, lo desreferenciamos con el operador asterisco. Tanto & como * son operadores unarios; se ponen delante de la variable y de esa manera obtenemos el puntero (&variable -> puntero) o la variable (*puntero -> variable).

Veamos otro ejemplo. En este caso, declaramos un puntero como una variable más:

```
#include <stdio.h>
int main()
{
      int i = 0, j = 0;
      int *p1, *p2;

      p1 = &i;
      p2 = &j;
      scanf("%d", p1);
      scanf("%d", &j);

      printf("i: %2d - j: %2d\n", i, j);
      p2 = p1;
      (*p2)++;
      printf("i: %2d - j: %2d\n", i, j);
      return 0;
}
```

En el ejemplo anterior, hacemos que `p1` apunte a `i`, y que `p2` apunte a `j`. En las líneas en las que recibimos datos por teclado, vemos que es lo mismo enviar `p1/&i`, o `p2/&j`. Luego del `printf()` cambiamos el puntero `p2` de lugar, lo hacemos apuntar al lugar donde está apuntando `p1`. Finalmente desreferenciar `p2` y ejecutar el postincremento, estamos afectando la variable i.

Tener en cuenta las siguientes consideraciones:

- Un puntero sin asignar puede hacer que un programa deje de ejecutarse al tratar de leer/escribir de posiciones de memoria no válidas.
- No es necesario declarar una variable de tipo puntero, podemos utilizar el operador & y "generar" una cuando necesitemos pasar un puntero como parámetro.
- La precedencia de operadores suele complicar las cosas: es recomendable escribir las operaciones de referencia y desreferencia entre paréntesis, hasta saber bien en qué momento hacen falta y cuándo no. Por ejemplo, si quitamos los paréntesis en el código de arriba, se aplicaría primero el postincremento y luego el operador de desreferencia, creando un código que no hace nada.
- Los punteros no son otra cosa que valores enteros. Pero no es aconsejable tratarlos como tales, prestar mucha atención a los mensajes que genera el compilador para verificar que estamos

trabajando con los tipos correctos. Un error de asignación entre punteros y enteros va a compilar con "warnings", pero no da error. El error se produce al momento de ejecutar el programa.

ARITMÉTICA DE PUNTEROS

Es posible utilizar los operadores + y – con punteros. Las siguientes operaciones tienen sentido:

- Sumarle o restarle un número "n" a un puntero: el resultado es otro puntero desplazado ("n" bytes x el tamaño del tipo de dato) respecto al original. Por ejemplo, si tenemos un puntero a `char` y le sumamos 1, obtenemos un puntero a la siguiente posición en memoria desde la ubicación de la variable `char`.
- Restar dos punteros: obtendremos la diferencia de posición entre los 2 punteros.
- Al utilizar el operador de suma y resta de enteros, tiene importancia el tipo de dato con el que estamos trabajando:
    - Si sumamos 1 a un puntero a entero, obtendremos una dirección de memoria 4 bytes mayor a la dirección original.
    - Si sumamos 1 a un puntero a char, obtendremos una dirección de memoria 1 byte mayor a la dirección original.
- No se puede suponer que si hay dos variables del mismo tipo declaradas en forma contigua, al sumar 1 a un puntero a la primera se obtendrá la segunda.
- Para imprimir punteros, podemos usar "%p". Al hacerlo así obtendremos la dirección de memoria a la que referencia el puntero, en código hexadecimal.

Ejemplos:

```
#include <stdio.h>
int main()
{
     int i;
     int a[10] = {1,2,3,4,5,6,7,8,9,10};
     int *p1, *p2;
    char c1 = 'a';
    char c2 = 'b';
    char *pc;

    p1 = &a[0];
    p2 = &a[7];
    printf("Numeros en un arreglo: %p: %d - %p: %d - Dif: %d \n",
p1, *p1, p2, *p2, p2 - p1);

    pc = &c1;
    printf("Caracteres sueltos: %p: %c - %p(%p): %c \n", pc, c1,
pc+1, &c2, c2 );

    *(p1 + 3) = 200;
    *(p2 - 1) = 50;

    for (i = 0; i < 10; i++)
        printf("%p: %d\n", &a[i], a[i]);
```

```
    return 0;
}
```

```
Numeros en un arreglo: 0028FF00: 1 - 0028FF1C: 8 - Dif: 7
Caracteres sueltos: 0028FEF7: a - 0028FEF8(0028FEF6): b
0028FF00: 1
0028FF04: 2
0028FF08: 3
0028FF0C: 200
0028FF10: 5
0028FF14: 6
0028FF18: 50
0028FF1C: 8
0028FF20: 9
0028FF24: 10

Process returned 0 (0x0)   execution time : 0.014 s
Press any key to continue.
```

- En el primer `printf()`, estamos imprimiendo la dirección de memoria del primer puntero, su contenido, la dirección de memoria del segundo puntero, su contenido, y la diferencia en bytes entre ellos.
- En el segundo `printf()`, podemos ver que si bien pc apunta a c1, pc+1 no apunta a c2; al contrario, el programa c2 está "más abajo" que c1. También vemos que las sumas producen aumentos de a 1 byte, en el caso de los punteros a enteros, son de 4.
- El loop de impresión del arreglo muestra que se puede sumar y restar en forma segura dentro de los límites de un arreglo, y modificar los contenidos del mismo.

ARREGLOS ESTÁTICOS

Cuando trabajamos con arreglos, estamos trabajando en forma indirecta con punteros. Al momento de declararlo, el nombre del arreglo puede ser pensado como un puntero a la primera posición del mismo.

- La sintaxis `int arreglo[10];` es una forma de declarar un puntero a entero y al mismo tiempo apuntarlo a la primera de 10 posiciones en memoria de tipo entero que se reservan en el stack.
- Las funciones no pueden declarar el tamaño de los arreglos que reciben, ya que en realidad lo que están recibiendo son punteros. Al momento de declarar un prototipo de función, es indistinto usar [] o * .

Ejemplo:

```
#include <stdio.h>

int fun(int*);
char fun2(char*);

int main()
{
    char frase[25] = "hola mundo";
    //es posible declarar char* frase = "hola mundo"; pero el
arreglo
    //resultante es "estático", es decir que si tratamos de
ejecutar
    //fun2 sobre el mismo, el programa termina con error.
    //Esto solo puede hacerse con chars, los arreglos de otros
tipos
    //se deben declarar con corchetes.
    int arreglo[10] = {1,2,3,4,5,6,7,8,9,10};
    int a;
    char c;

    a = fun(arreglo);
    c = fun2(frase);

    printf("%d %c \n", a, c);
    return 0;

}

int fun(int * puntero)
{
    if (*(puntero + 3) > puntero[4])
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

char fun2(char* s)
{
    s[4] = *(s+1) + (s[2] - s[3]);
    return s[4];
}
```

5 - MANEJO DINÁMICO DE MEMORIA

Sabemos que la gestión de memoria de un programa escrito en el lenguaje C reconoce 3 áreas: el stack o pila, donde se almacena el espacio para las variables declaradas en el main() y en el cuerpo de cada función, la región estática, donde se guardan variables globales y constantes, y el heap o área dinámica. A diferencia del stack, donde se conoce en cada momento cuánta memoria es necesaria para ejecutar cada función, la memoria en el heap es otorgada en forma dinámica: existen instrucciones que nos permiten pedir memoria, y liberarla cuando ya no la necesitamos.

¿Para qué podemos utilizar el área de memoria dinámica? Básicamente para cualquier procedimiento en el cual desconozcamos de antemano la cantidad de datos con los que vamos a estar trabajando: por ejemplo si tenemos que almacenar información cargada por teclado, sin un límite máximo conocido, o si tenemos que procesar un archivo cuyo tamaño desconocemos, etc. De esta manera evitaremos declarar estructuras innecesariamente grandes, y aprovechar mejor los recursos de los cuales disponemos. Como mencionamos anteriormente, la memoria disponible en el stack es limitada; la memoria del heap no sufre de tantas restricciones. Sin embargo, al ser un área que es compartida con otros procesos que pueden correr al mismo tiempo, debemos ser cuidadosos a la hora de solicitarla, y retornarla al sistema lo antes posible, de manera de no ocupar recursos cuando no los necesitamos.

Ejemplo:

Supongamos que queremos realizar un programa que ordene una secuencia de números ingresada por teclado. Hasta este momento la única manera que podemos hacer esto es declarar un arreglo de un tamaño máximo (N = 100 x ejemplo), y marcar con alguna bandera el fin de los valores válidos. Esta manera de resolver el problema tiene dos inconvenientes:

- El programa no puede procesar más de N elementos.
- Si los datos a ingresar son sólo 10, estamos desperdiciando 90 espacios en memoria.

Si bien la pérdida de memoria en el ejemplo es ínfima, en una escala mayor de procesamiento de datos esta decisión puede conducir a fallas por falta de memoria. Para evitar este problema, recurriremos al área de memoria dinámica para almacenar los datos. Sólo usaremos la cantidad de memoria que necesitemos, y luego de usarla la liberaremos para posterior reuso.

SOLICITAR MEMORIA DINÁMICA – FUNCIÓN MALLOC()

Para solicitar un bloque de memoria del heap usamos la función malloc(). La invocación a la función retorna un puntero al nuevo bloque, por lo cual usaremos una variable de tipo puntero para saber dónde se encuentra el bloque.

El prototipo de la función malloc es el siguiente: **`void* malloc(size_t)`**;

¿Qué significa cada parte?

- void* es un tipo de retorno “puntero genérico”: la función malloc() puede retornar un puntero a memoria almacenada de cualquier tipo: esta es la forma que tiene el lenguaje de indicarlo
- malloc es el nombre de la función, recibe como parámetro la cantidad de bytes que queremos reservar.
- size_t es un typedef para valores long unsigned int. Lo que significa que podemos pasar cualquier valor numérico entero positivo.

¿Cómo utilizamos la función malloc()? Lo más común es declarar un puntero a entero y asignar el puntero que retorna malloc() a este puntero.

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

int main()
{
    int i, n;
    int *arreglo = NULL;
    printf("Indique la cantidad de datos a ingresar: ");
    scanf("%d", &n);
    arreglo = (int *) malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
    {
        printf("Ingreso de dato %d", i+1);
        scanf("%d", (arreglo + i));
    }
    return 0;
}
```

Observaciones:

- Las últimas versiones de los compiladores de C detectan automáticamente el tipo de puntero que se está asignando, pero anteriormente era norma “castear” el puntero al tipo correcto. Por ejemplo: `int* arreglo = (int*) malloc(10 * sizeof(int));`
- Los punteros se inicializan a NULL de manera que estén con un valor inicial conocido. NULL equivale habitualmente al valor 0.
- Para calcular la cantidad de bytes que necesitamos reservar en el heap, multiplicamos la cantidad de valores que queremos reservar * sizeof(tipo de dato).
- Si se reserva más de un espacio de memoria, entonces estamos trabajando con un arreglo “dinámico”. Por lo tanto, es posible usar la notación de corchetes [] para acceder a sus miembros.
- Es posible que el pedido de memoria falle (por ejemplo, si solicitamos más memoria que la que el sistema tiene disponible para nuestro proceso). En ese caso, malloc() retorna NULL. En la práctica (sobre todo en esta materia) es poco común que dicha situación ocurra, pero es obligación verificar que el malloc no haya fallado, con un if a continuación del llamado a malloc. En algunos ejemplos de este apunte se omite dicha verificación, con el espíritu de mantener los ejemplos sencillos, pero recuerde que el alumno está obligado a hacerlo en los ejercicios.

LIBERAR MEMORIA SOLICITADA: FREE()

El pedido de memoria realizado con malloc() se acompaña de su operación complementaria: liberar la memoria que el sistema entrega para que el mismo proceso, más adelante, o incluso otros procesos, puedan aprovechar dicha memoria. La norma es liberar la memoria en el momento en el cual deja de necesitarse. Si la memoria no es liberada en forma explícita, el sistema se encarga de ello al finalizar el proceso.

El prototipo de la función es: **`void free(void*);`**

- El parámetro que se le envía a free() es un puntero entregado por malloc() (o realloc(), de la cual hablaremos más adelante).
- Es imposible liberar memoria del stack, sólo puede liberarse memoria del heap. O sea no tiene sentido hacer free() de un arreglo declarado con la notación de corchetes.
- No es posible liberar una "parte" de la memoria que se solicitó, free() recibe como parámetro un valor que ha sido entregado anteriormente por malloc() (internamente, el programa mantiene una tabla de la memoria pedida, entonces cuando le pedimos que libere un cierto puntero, busca ese dato en la tabla; por lo cual sólo esos valores pueden ser liberados).

El ejemplo anterior se completa agregando la línea "free(arreglo);", una vez que ya no necesitemos el mismo.

Observaciones:

- Es posible programar funciones que soliciten un espacio en memoria y que sea responsabilidad de la función llamadora liberar el espacio posteriormente. Muchas de las funciones de la biblioteca <string.h> utilizan esta modalidad.
- Es una mala práctica de programación no "limpiar" la memoria que no requerimos. Debido a la complejidad relacionada con la gestión manual de memoria, lenguajes de programación posteriores a la invención de C han hecho automático dicho manejo (tanto la solicitud como su posterior liberación, a través de algoritmos denominados Garbage Collectors).

CAMBIAR LA CANTIDAD DE MEMORIA RESERVADA: REALLOC()

No siempre necesitamos el bloque de memoria que solicitamos la primera vez: a veces podemos requerir mayor cantidad, y en otras ocasiones, es posible que debamos devolver parte de la memoria pedida porque no la vamos a utilizar. La función realloc() permite modificar la cantidad de memoria pedida.

El prototipo de la función es: **`void* realloc(void*, size_t);`** donde:

- La función retorna un nuevo puntero a la memoria realocada. Es posible que al pedir más (o menos) memoria, el sistema decida ubicar la memoria en otra posición (de manera de aprovechar mejor los espacios que se van generando). Cuando esto ocurre, todos los contenidos originales se copian a la nueva posición, y ésta es retornada.
- El primer parámetro es el puntero original (pedido con malloc() o con un realloc() anterior).
- El segundo parámetro es el nuevo tamaño que ocupa la memoria: podemos pedir más, o menos. En el primer caso nuestra memoria se amplía al nuevo tamaño (p.ej., si originalmente pedimos 10

espacios y luego pedimos 14, se agregarán 4 espacios), y en el segundo, se achica (si teníamos 10 espacios y pedimos 5, se eliminarán los últimos 5).

Ejemplo: si queremos crear un arreglo de enteros donde sólo ocupemos la memoria mínima necesaria, podemos ir pidiendo de a 1 lugar:

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

int main()
{
    int i = 0, n, ocupado = 5;
    int *arreglo = (int *)malloc( 5 * sizeof(int) );
    printf("Agregue datos, finalice con 0: ");
    do
    {
        scanf("%d", &n);
        if (n != 0)
        {
            *(arreglo + i ) = n;
            i++;
            if (i >= ocupado)
            {
                arreglo = realloc(arreglo, (i + 1) * sizeof(int));
                ocupado = i;
            }
        }
    } while( n != 0);

    printf("Memoria ocupada: %d\n", ocupado);
    free(arreglo);
    return 0;
}
```

En el ejemplo anterior, si usamos menos de 5 espacios en memoria, la memoria asignada será mayor a la usada. ¿Cómo podría arreglarse?

6 - RECURSIVIDAD

INTRODUCCIÓN

Imaginemos por un momento que nos encontramos frente a una fila de gente. No podemos ver dónde termina la fila, ni sabemos cuántas personas hay en ella. Tan sólo podemos ver y hablar con la primer persona. Ella, a su vez, puede darse vuelta y hablar con quien tiene inmediatamente detrás. Queremos saber quién es la persona que tiene el número de documento mayor. ¿De qué manera podemos resolver el problema?

Una solución a nuestro problema consiste en preguntarle a la persona que tenemos detrás cuál es su número de documento, y compararlo con el nuestro. De esta comparación podemos obtener el mayor número de documento entre ambo. A continuación, la persona detrás nuestro debe hacer exactamente lo mismo, pero usando este nuevo número. Al finalizar la fila, la última persona tendrá la respuesta y se la devolverá a la anterior, que hará lo mismo, hasta llegar a la primer persona.

Otra posibilidad es preguntarle a la persona que está detrás nuestro, cuál es el mayor número de documento de la fila que empieza en él mismo, comparar su dato con el nuestro y de ahí obtener la respuesta. La segunda persona usa la misma estrategia que nosotros: para saber cuál es el mayor número de documento de su fila, va a preguntarle a la siguiente persona cuál es el mayor documento de la otra fila… y así sucesivamente hasta que lleguemos a la última persona de la fila. Ella, al no tener ninguna detrás, sabe que el mayor número de documento es el suyo. A medida que el resultado va "volviendo", el mayor número es el que obtenemos como resultado final.

Un tercer ejemplo: supongamos que queremos conocer la cantidad de letras que tiene una cadena de caracteres. Tradicionalmente lo que podemos hacer es utilizar un for y contar a medida que avanzamos, hasta llegar al carácter '\0'. Pero también es posible pensarlo de la siguiente manera: la cadena vacía tiene una longitud de 0, y cualquier otra cadena mide 1 más que lo que mide la cadena que empieza en el segundo elemento. Es decir:

```
int longitud(char cad[])
{
    if (cad[0] == '\0')
          return 0;
      else
           return 1 + longitud (cad +1);
}
```

En todos los casos, hemos utilizamos un método de resolución recursivo.

DEFINICIÓN DE RECURSIVIDAD

Decimos que una definición es recursiva cuando involucra aquello que se define en el cuerpo de la definición. En el caso de las funciones recursivas, diremos que una función es recursiva cuando se invoca a sí misma.

Ejemplos de definiciones recursivas:

- Si decimos "los amigos de mis amigos son mis amigos", estamos implicando que dentro de la definición de amigos está involucrada la misma definición.
- Mis antepasados son mis padres, y sus padres, y sus padres, etc. Podemos decir que mis antepasados son mis padres, y los antepasados de mis padres. Entonces los antepasados de mis padres serán sus padres, y los antepasados de ellos, y así sucesivamente.

En una definición recursiva, debemos identificar dos partes:

- El caso base (aquello que no se repite): *la longitud de la cadena vacía es 0, mis antepasados son mis padres,* etc
- El caso recursivo (aquello que se repite): *mis antepasados son los antepasados de mis padres, la persona con documento más grande en la fila es la persona con documento más grande en la fila que comienza luego de mí*; etc.
    - Es necesario notar que la definición recursiva involucra aquello que se define, pero con alguna variación que permite ir "achicando" la definición hasta llegar al caso base. Si una definición recursiva repite aquello que está definiendo sin variación, se cae en un ciclo infinito de repeticiones.

EJEMPLO DE FUNCIONES RECURSIVAS: EL FACTORIAL

La función factorial es el clásico caso de función que se puede definir recursivamente. Matemáticamente, sabemos que:

```
!0 = 1
!n = n (n-1)! (si n > 0)
```

Podemos definir entonces el factorial de la siguiente manera:

```
unsigned long int factorial(unsigned long int n)
{
      if (n == 0)
            return 1;
      else
            return n * factorial(n - 1);
}
```

Utilizamos en este caso valores `unsigned` para no tener problemas con el signo de la variable.

EJEMPLO DE FUNCIÓN RECURSIVA: ENCONTRAR EL MÁXIMO EN UN ARREGLO

Veamos el ejemplo de encontrar la persona con el mayor DNI en una fila. Supongamos que la fila es un arreglo de estructuras personas, y que vamos a retornar un puntero a la posición donde se encuentra la persona con mayor DNI. Un `dni == 0` marca el fin de los valores válidos.

Hay dos variantes recursivas en este ejercicio: la primera consiste en comparar el primer elemento del arreglo con el mayor del resto. La segunda consiste en comparar el primer elemento con el segundo, y así hasta llegar al final del arreglo.

Variante 1:

```
typedef struct
{
	char nombre[20];
	int dni;
} Persona;

Persona* mayor(Persona* fila)
{
	Persona* mayorAux;
	if ( fila[1]->dni == 0)
		return fila;
	else
	{
		mayorAux = mayor(fila + 1);
		if (fila->dni >  mayorAux->dni)
			mayorAux = fila;
		return mayorAux;
	}
}
```

Variante 2:

```
Persona* mayor(Persona* fila, Persona* mayorAux)
{
	if ( fila->dni == 0)
		return mayorAux;
	else
	{
		if (fila->dni >  mayorAux->dni)
			return mayor(fila + 1, fila);
		else
			return mayor(fila + 1, mayorAux);
	}
}
```

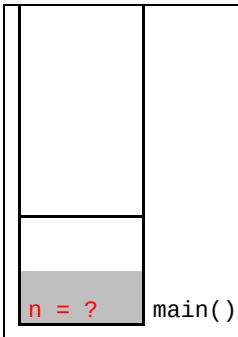
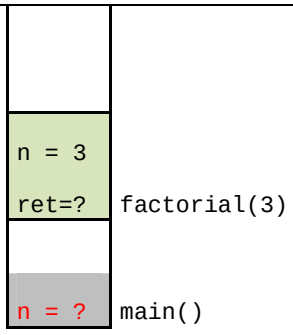
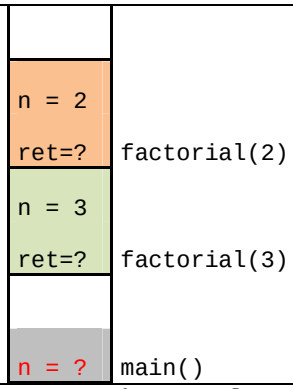
Esta variante se invoca pasando el primer elemento del arreglo como segundo parámetro, para alimentar la función con un valor válido. Al llegar al final del arreglo, lo que nos quedó en el segundo parámetro es el valor que queremos.

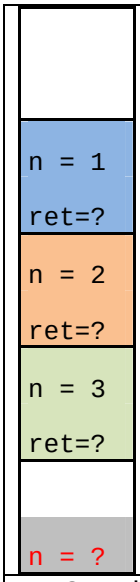
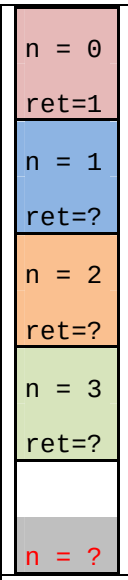
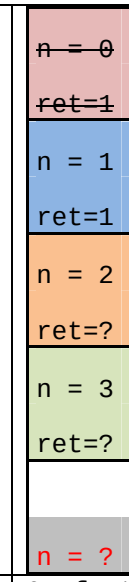
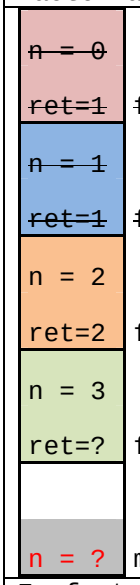
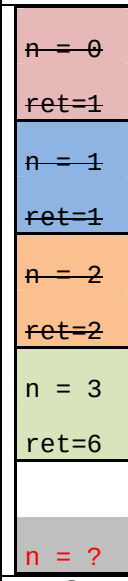
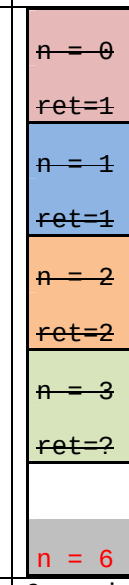
SEGUIMIENTO DE PILA

En ocasiones es conveniente visualizar qué ocurre en la memoria cuando empezamos a resolver problemas con algoritmos recursivos. Cada invocación de función genera un apilamiento de memoria en el stack, y es fácil "perder el hilo" de dónde nos encontramos.

Realicemos el seguimiento de la siguiente porción de código:

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

unsigned long int factorial(unsigned long int n);

int main()
{
      unsigned long int n = factorial(3);
      printf("%lu\n", n);
      return 0;
}

unsigned long int factorial(unsigned long int n)
{
      if (n == 0)
            return 1;
      else
            return n * factorial(n - 1);
}
```

		
1- Inicia main Se invoca factorial(3)	2- Se ejecuta factorial(3) Se copia el valor 3 El valor de retorno de la función se desconoce hasta que se ejecute factorial(2) Se llama a factorial(2)	3- Se ejecuta factorial(2) Se copia el valor 2 El valor de retorno de la función se desconoce hasta que se ejecute factorial(1) Se llama a factorial(1)

<table>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>4- Se ejecuta factorial(1)
Se copia el valor 1
El valor de retorno de la función se desconoce hasta que se ejecute factorial(0)
Se llama a factorial(0)</td>
<td>5- Se ejecuta factorial(0)
Se copia el valor 0
Se retorna 1. Ahora factorial(1) puede finalizar su ejecución</td>
<td>6- factorial(1) retorna 1. Ahora factorial(2) puede finalizar su ejecución</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>7- factorial(2) retorna 2 x 1 = 2. Ahora factorial(3) puede finalizar su ejecución</td>
<td>8- factorial(3) retorna 2 x 3 = 6. Ese valor es devuelto al main()</td>
<td>9- main() finaliza su ejecución</td>
</tr>
</table>

7- ESTRUCTURAS DINÁMICAS – GENERALIDADES Y LISTAS

INTRODUCCIÓN

Una estructura dinámica es una agrupación de datos – que ponemos juntos porque se refieren a una misma idea o concepto – que tiene la facultad de modificar su tamaño en la medida que lo necesitemos. Una estructura dinámica nos sirve para modelar, en la computadora, situaciones problemáticas en las cuales la cantidad de "individuos" (datos individuales) con la que trabajamos es desconocida de antemano, que presentan entre sí una cierta correlación.

Ejemplo: la cola de gente en una caja de supermercado es una estructura dinámica. Si consideramos que cada persona es un individuo (elemento) de nuestra estructura, veremos que ésta crece (cuando llega gente) y decrece (cuando la gente se va). Los elementos están en correlación, cada elemento pertenece a la cola y sabe quién lo antecede y quién lo sigue (excepto claro el primero y el último).

Dependiendo del tipo de relación que haya entre los elementos, y de las operaciones que definimos sobre los mismos, tradicionalmente en Informática se definen las siguientes estructuras:

- Listas: estructura secuencial de datos. Cada elemento conoce al elemento que le sigue (y potencialmente a su predecesor). Conociendo al primer elemento, podemos tener acceso a cualquiera de ellos. Las operaciones que podemos hacer son: agregar, quitar elementos, recorrerla (por ejemplo para imprimirla), y agregar en forma ordenada (si la lista define un orden interno). Ejemplos: lista de personas en un equipo de futbol.

- Pilas: estructura secuencial de datos de modelo LIFO (last in – first out). Esto quiere decir que los elementos se agregan y se eliminan por el mismo extremo. Ejemplo: una torre de CDs, sólo podemos agregar y quitar CDs por el extremo superior. En una pila no tiene sentido hablar de "recorrer" o de "agregar en forma ordenada", ya que las únicas operaciones definidas son agregar (push) y quitar (pop).

- Colas: estructura secuencial de datos de modelo FIFO (first in – first out). Esto quiere decir que los elementos se agregan por un extremo y se quitan por el otro. Por ejemplo: una cola de gente en el supermercado. En una pila no tiene sentido hablar de "recorrer" o de "agregar en forma ordenada", ya que las únicas operaciones definidas son encolar (enqueue) y desencolar (dequeue).

- Árboles: estructura "ramificada" de datos. Cada elemento conoce a otros a los que llama sus hijos. A su vez cada hijo también es un elemento, y puede conocer otros hijos. Por ejemplo: un árbol genealógico.

Podemos pensar que cada estructura está definida de dos maneras: a través de su **especificación** (es decir qué cosas esperamos que nos permita hacer); y a través de su **implementación** (cuál es el código que nos permite cumplir con la especificación).

CONOCIMIENTOS PREVIOS

Las especificaciones de qué hace una estructura dinámica no requieren conocimientos previos de informática, podríamos decir que son formas convencionales de representación de datos (de hecho son tan

convencionales que otros lenguajes de programación las implementan en forma nativa). Sí es necesario estar de acuerdo en dichas operaciones cuando hablamos entre nosotros en clase: independientemente de cómo las implementemos, cuando hablemos de pilas pensaremos sólo en términos de *push* y *pop*, cuando hablemos de colas, en enqueue y dequeue, y así sucesivamente.

La implementación, por otro lado, requiere que tengamos presentes todos los temas vistos anteriormente en la materia:

- Estructuras
- Punteros
- Memoria Dinámica
- Recursividad

¿De qué manera utilizamos estos conocimientos? La implementación de una estructura dinámica se realiza de la siguiente manera:

- Utilizaremos un *struct* para representar cada uno de los individuos de nuestra estructura. Este *struct* va a contener 2 tipos de información: la del individuo que estemos representando por un lado, y por otro lado, información que nos permitan relacionarla con el resto de la estructura.
- Cada vez que necesitemos agregar un elemento a la estructura, lo haremos a través de un *malloc().* Cada vez que necesitemos quitar uno, lo quitaremos con *free().* De esta manera, la estructura ocupa sólo la memoria que necesita.
- En los *structs*, la relación entre un elemento y los demás la realizaremos a través de punteros. Dado entonces que cada struct conoce a otros structs (del mismo tipo), los mismos son *recursivos*.
- La estructura completa la referenciaremos con un puntero a uno de sus elementos. Inicialmente este puntero estará en NULL y diremos que la estructura está vacía, y al momento que agreguemos un elemento, ya no lo estará.

Veamos ahora cómo se ponen en práctica todas estas consideraciones.

LISTAS

Como dijimos anteriormente, una lista es una secuencia de datos. En la implementación básica de listas que utilizaremos, cada elemento conoce al elemento que lo sigue, y nosotros conoceremos (a través de un puntero) al primer elemento de la lista. Las listas pueden estar ordenadas de acuerdo al orden de llegada de los elementos, o pueden tener un cierto orden definido de acuerdo a un criterio.

Ejemplos de listas: una lista de invitados a una fiesta, una lista de asistentes a una reunión, una lista de compras de supermercado, etc.

Similitudes y diferencias con arreglos: una lista permite representar la misma información que podríamos representar en un arreglo. Las similitudes son evidentes: un arreglo también es una secuencia de datos. En

un arreglo, cada elemento tiene un sucesor (excepto el último), y para accederlo debemos simplemente avanzar 1 posición de memoria. Respecto a las diferencias: un arreglo tiene todos sus elementos contiguos en la memoria, en una lista esto no es así. Para acceder al n-ésimo elemento, simplemente hacemos arreglo[n], en cambio en una lista tendremos que posicionarnos en el elemento inicial y avanzar n veces hasta llegar al enésimo.

OPERACIONES

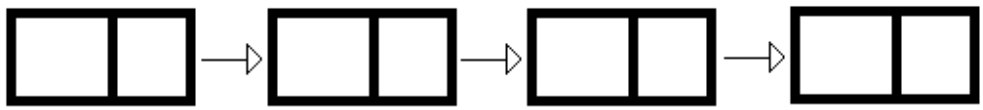
En una lista las siguientes operaciones son válidas:

- Agregar un elemento: si la lista está vacía, esta operación "crea" la lista.
- Quitar un elemento.
- Agregar un elemento en forma ordenada, si la lista tiene un criterio de orden.
- Recorrer la lista.
- Otras (pero las anteriores son las básicas).

IMPLEMENTACIÓN DE LA ESTRUCTURA

Para implementar una lista necesitaremos modelar cada uno de sus elementos. Cada elemento lo representaremos con un struct al que llamaremos "nodo". Así, una lista estará compuesta por nodos. Cada nodo contendrá información del individuo que representa, y tendrá información sobre el siguiente nodo de la lista (es decir, cada nodo conoce al nodo siguiente).

Gráficamente lo representaremos así:



En la parte izquierda del nodo pondremos la información que queremos representar, la parte derecha del nodo es un puntero al siguiente elemento.

El código para representar un nodo es el siguiente:

```
struct s_nodo
{
      int valor;
      struct s_nodo* sig;
};
typedef struct s_nodo* t_nodo;
```

`valor` es una variable de tipo int, pero podría ser de cualquier otro tipo (o podría haber más de una variable). Lo interesante es que la estructura `s_nodo` tiene un puntero a otra estructura `s_nodo` (por eso decimos que es recursiva). Además definimos un tipo `t_nodo`, que es puntero a `s_nodo`.

IMPLEMENTACIÓN DE LAS OPERACIONES

Vamos a definir las siguientes operaciones: agregar(), insertar() (agregar ordenado) y eliminar().

Las funciones anteriores pueden programarse de dos maneras: en forma recursiva y en forma iterativa. Cada una de ellas presenta ventajas y desventajas. Generalmente las formas recursivas se codifican más brevemente, y su funcionamiento es uniforme dentro de toda la lista. Las soluciones iterativas generalmente son más eficientes, pero deben definir funcionamientos diferentes de acuerdo a la posición de la lista sobre la que están operando (el principio, el medio, el final, o la lista vacía).

Agregar un valor a una lista

Para agregar un valor a una lista, vamos a definir el siguiente prototipo de función: `agregar(t_nodo*, int)`. Veamos un ejemplo de uso y aclaremos qué significa el tipo `t_nodo*`.

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

struct s_nodo
{
      int valor;
      struct s_nodo* sig;
};
typedef struct s_nodo* t_nodo;

void agregar(t_nodo*, int);

int main()
{
      t_nodo lista = NULL;
      agregar(&lista, 3);
      agregar(&lista, 7);
}
```

¿Qué pasa en el código de arriba?

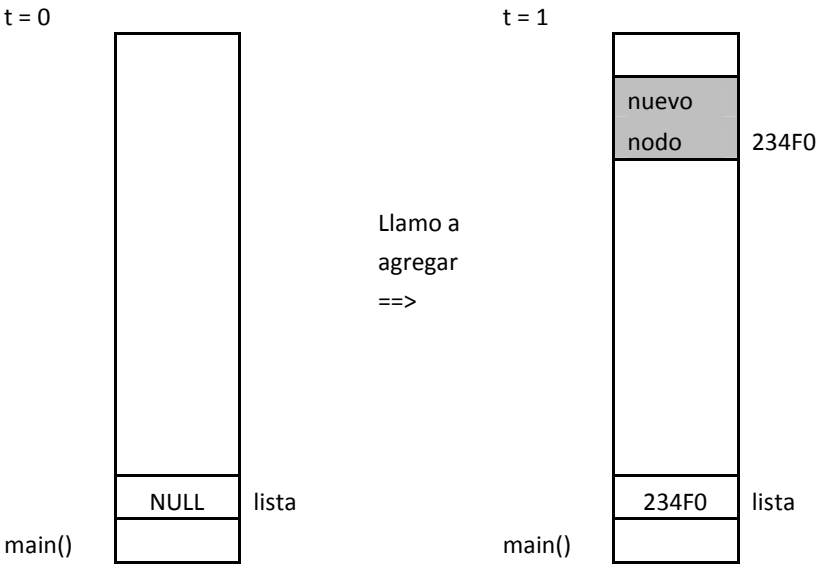
1- Declaramos las estructuras que necesitamos

2- Declaramos el prototipo de una función agregar. Esta función nos sirve para crear la lista, si es que ésta está vacía, o para agregarle un nodo al final, si es que ya contiene elementos.

3- En el main(), utilizamos la lista enviando como parámetro &lista. ¿Por qué? Sabemos que las funciones en C no pueden modificar los argumentos que le enviamos (por ejemplo, si invocamos la función pow(a, b), a la salida de la función a y b no se modifican). Cuando queremos que una función modifique el parámetro que recibe, tenemos que pasarlo como un puntero. Como lista es

una variable de tipo puntero, pero a su vez queremos que se modifique dentro de la función, tenemos que enviar un puntero a ese puntero, o lo que es lo mismo, un doble puntero o doble indirección.

Para verlo gráficamente:



Luego de invocar a agregar(), la variable lista, que era un puntero apuntando a NULL, ahora contiene la dirección de memoria donde se creó el nuevo nodo. La única manera de lograr eso, en el lenguaje C, es si pasamos un puntero a dicha variable (ya que de esa manera, podemos modificarlo en una función).

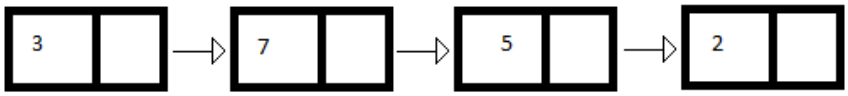
Veamos la codificación (en forma recursiva) de la función agregar:

```
void agregar(t_nodo* nodo, int valor)
{
      if ( *nodo == NULL)
      {
            *nodo = (t_nodo) malloc(sizeof(struct s_nodo));
            (*nodo)->valor = valor;
            (*nodo)->sig   = NULL;
      }
      else
      {
            agregar(& (*nodo)->sig, valor);
      }
}
```

Vemos que la función hace que el puntero a un nodo apunte a una nueva posición de memoria sólo cuando éste está apuntando a NULL: cuando invoquemos agregar() la primera vez, hará que nuestra variable lista

apunte a un nuevo nodo. Cuando llamemos a la función una segunda vez, como lista no está apuntando más a NULL, se invocará recursivamente, pasando como parámetro un doble puntero al miembro "sig" del primer nodo. El efecto práctico es que dicho puntero "sig" va a apuntar a un nuevo nodo que se cree cuando la función sea invocada la segunda vez. Si volvemos a agregar otro nodo, ocurrirá nuevamente lo mismo (el nodo inicial no es vacío y avanza recursivamente, el siguiente nodo no es vacío tampoco y vuelve a avanzar, aquí sí el nodo estará apuntando a NULL así que crea una nueva estructura en memoria y la hace apuntar por ese nodo).

```
int main()
{
      t_nodo lista = NULL;
      agregar(&lista, 3);
      agregar(&lista, 7);
      agregar(&lista, 5);
      agregar(&lista, 2);

}
```



Insertar en forma ordenada un valor en una lista

Si la lista que queremos crear debe respetar un cierto orden, entonces la función `agregar()` no nos sirve, porque ésta agrega sólo al final.

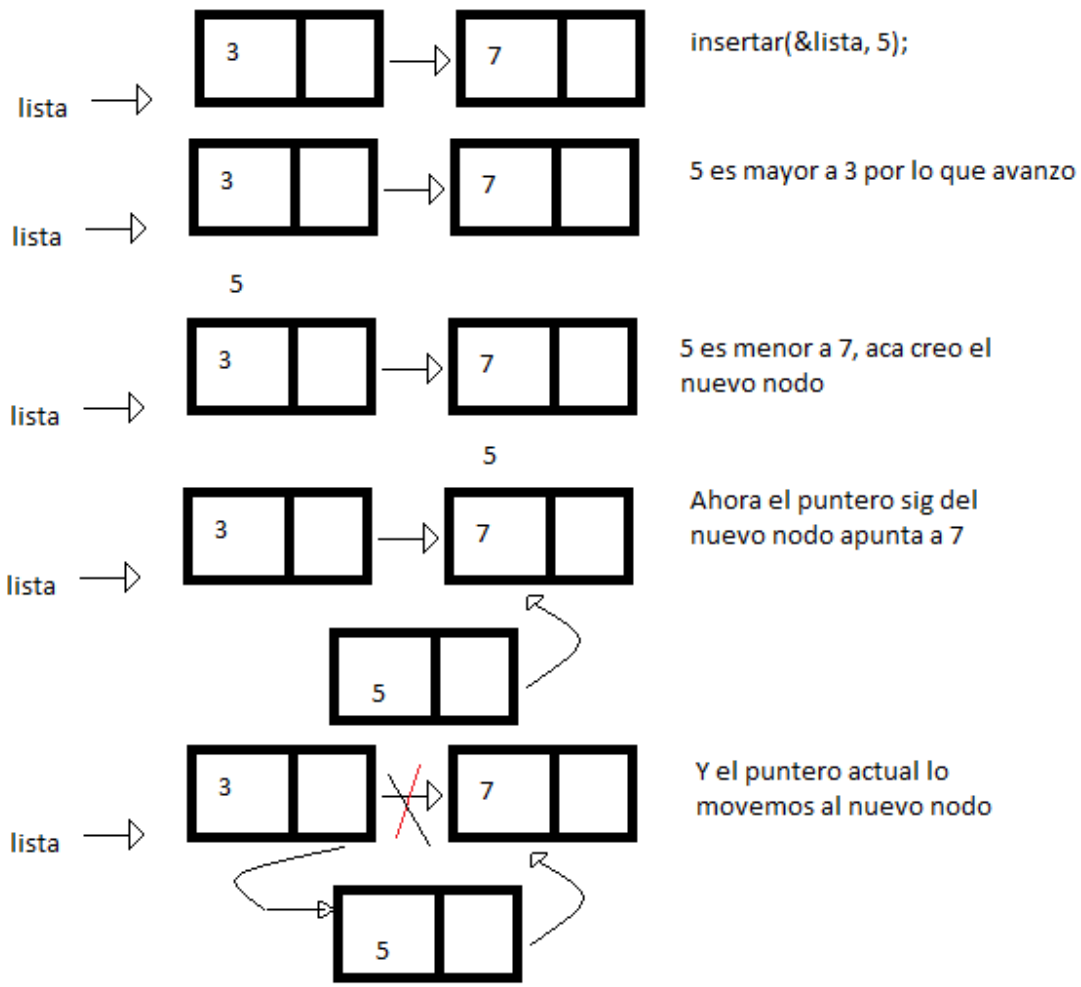
Revisemos la lógica que deberíamos seguir si queremos insertar en una lista en forma ordenada. Supongamos que partimos de la lista `3->7` y queremos agregar el valor 5. La lista que vamos a obtener va a ser `3->5->7`.

Lo que vamos a tener que hacer es, básicamente, avanzar por toda la lista hasta encontrar un nodo cuyo valor sea mayor al valor que queremos insertar. Si nos paramos en `lista` y miramos el nodo con el valor 3, si queremos insertar el 5, vamos a tener que avanzar un paso. Si nos paramos en el puntero "`sig`", y observamos el valor del nodo al que apunta, vemos que es mayor al número que queremos insertar, por lo tanto es aquí donde debemos agregar nuestro nodo. A continuación:

- creamos un nodo y lo hacemos apuntar por un puntero auxiliar.
- Hacemos que el puntero sig del nuevo nodo apunte al nodo que estamos apuntando actualmente (en el ejemplo, el que contiene al 7).
- Ahora hacemos que el puntero al nodo actual en donde estamos, apunte a nuestro nuevo nodo auxiliar (o sea, el que contiene el 5).

Una última consideración es que no siempre vamos a insertar un valor entre 2 nodos: si nos ocurre que llegamos al final de la lista, tendremos que insertar el valor allí.

Gráficamente:



El código nos queda así:

```
void insertar(t_nodo* nodo, int valor)
{
      t_nodo aux = NULL;
      if (*nodo == NULL || valor < (*nodo)->valor)
      {
            aux = (t_nodo) malloc(sizeof(struct s_nodo));
            aux->valor = valor;
            aux->sig   = *nodo;
            *nodo      = aux;
      }
      else
      {
            insertar( & (*nodo)->sig, valor);
      }
}
```

Eliminar valores de una lista

Cuando eliminamos valores de una lista, lo que debemos hacer es avanzar hasta el nodo que queremos borrar, eliminarlo con free, y "reenganchar" los nodos restantes. La lista debe tener al menos un elemento para que podamos borrar, si la lista está vacía podemos imprimir un mensaje de error.

```
void eliminar(t_nodo* nodo, int valor)
{
    t_nodo aux = NULL;
    if (*nodo == NULL)
    {
        printf("Error al borrar valor de una lista vacia\n");
        return;
    }
    else if ( (*nodo)->valor == valor )
    {
        aux = (*nodo);
        *nodo = (*nodo)->sig;
        free(aux);
    }
    else
    {
        eliminar( & (*nodo)->sig, valor);
    }
}
```

Esta función también sirve para eliminar el primer elemento de una lista (ya que el puntero original pasará a apuntar al segundo elemento), como cualquier elemento del medio e inclusive el último (ya que hará que el anteúltimo nodo apunte a NULL, y borrará el último).

8 - ESTRUCTURAS DINÁMICAS –PILAS Y COLAS

INTRODUCCIÓN

Pilas y colas son dos estructuras de datos que se utilizan cuando queremos modelar problemas en los cuales las operaciones de inserción y borrado de elementos en la estructura ocurren por los extremos de la misma. En ambos casos utilizaremos el mismo tipo de nodos que hemos utilizado para programar listas simplemente enlazadas, sin embargo veremos que la implementación de las operaciones es diferente.

PILAS

Las pilas son estructuras de datos que sirven para modelar pilas ("stacks") de la vida real. Por ejemplo: pensemos en una torre de CDs.

Si tenemos una torre de CDs vacía y empezamos a agregar discos en la misma, podemos notar que:

- No hay problemas para agregar discos, siempre el último disco que agreguemos va a estar en el tope de la torre.
- Para quitar un disco, sólo lo podemos hacer por el tope. Si estamos buscando un disco en el medio, debemos quitar todos los anteriores, quitar el que buscamos, y volver a colocar todos.

Una pila se comporta de esa manera. Las operaciones de inserción y eliminación se producen por un extremo, al cual denominamos "tope" de la pila. No es posible acceder a un elemento del medio sin sacar antes los anteriores.

Decimos que una pila es una estructura LIFO (last in first out): el último elemento agregado es el primero en ser quitado.

A las operaciones de inserción y de remoción se las denomina “push” y “pop”.

ESPECIFICACIÓN

Sobre una pila sólo es posible realizar las siguientes operaciones:

- Push: permite agregar un elemento
- Pop: permite quitar un elemento y obtener que dicho elemento almacenaba.

IMPLEMENTACIÓN

La implementación de una pila podemos hacerla sobre una lista simplemente enlazada, usando el inicio de la lista como tope de la pila. Lo hacemos así porque es el único extremo que cambia, y el que tenemos más “a mano”: si utilizáramos el final de la lista como tope, tendríamos que recorrerla en su totalidad tanto para agregar como para eliminar un elemento.

Los prototipos de las operaciones nos quedan definidos así:

```
struct s_nodo
{
      int valor;
      struct s_nodo* sig;
};
typedef struct s_nodo* t_nodo;

void push(t_nodo*, int);
int pop(t_nodo*);
```

Ejemplo de uso:

```
int main()
{
    int val;
    t_nodo pila = NULL;
    do
    {
        printf("Ingrese un valor, con 0 finaliza: ");
        scanf("%d", &val);
        if (val != 0)
            push(&pila, val);
    } while ( val != 0);

    printf("\nLos elementos de la pila, desapilados, son:\n");
    while (pila != NULL)
    {
        val = pop(&pila);
        printf("%d -", val);
    }
    printf("\n");
    return 0;
}
```

PUSH

La función push permite agregar un nuevo elemento a la pila. Dicho elemento se convierte en el nuevo tope. La implementación consiste básicamente en crear un nuevo nodo, ponerlo enfrente de los que ya tenemos, y mover el tope de la pila al nuevo nodo creado.

```
void push(t_nodo* pila, int valor)
{
    t_nodo aux = (t_nodo) malloc( sizeof(struct s_nodo));
    aux->dato = valor;
    aux->sig  = NULL;
    aux->sig = *pila;
   *pila = aux;
}
```

POP

La función pop permite extraer el último elemento de una pila. Al hacerlo, eliminaremos el nodo que guardaba dicho elemento. El tope de la pila se va a mover al nodo siguiente al primero. No es posible ejecutar la función pop sobre una pila vacía, se producirá un error en tiempo de ejecución.

```
int pop(t_nodo* pila)
{
    int valor;
    t_nodo aux = *pila;

    valor = aux->dato;
    *pila = aux->sig;
    free(aux);
    return valor;
}
```

COLAS

Las colas son estructuras de datos que sirven para modelar (sorpresa) colas de atención de la vida real.

En una cola de atención sin prioridad, la primer persona que llega a la cola es la primer persona en ser atendida, y la primera en retirarse. La gente que llega después, se va encolando detrás de la primera, de manera que el último que llegó será el último en retirarse.

Al extremo por el cual se producen las inserciones lo llamamos el “final” de la cola, y al extremo por el cual los elementos se quitan, lo llamamos el “frente”.

Decimos que una cola es una estructura FIFO (First In, First Out). El primer elemento que se agrega es el primero que se quita.

ESPECIFICACIÓN

Sobre una cola sólo dos operaciones son válidas:

- queue o encolar: agrega un elemento al final de la cola.
- dequeue o desencolar: quita un elemento del frente de la cola.

IMPLEMENTACIÓN

Implementaremos una cola sobre una lista simplemente enlazada. Las inserciones se producirán al final de la lista, y las eliminaciones, desde el principio.

Dado que necesitaremos operar por ambos extremos de la lista implementación, es conveniente mantener punteros tanto al frente como al final en forma simultánea. De esta manera, en ningún caso deberemos recorrer la lista en su totalidad para agregar o quitar elementos.

Para poder manipular ambos extremos en forma simultánea, utilizaremos una estructura auxiliar.

```
struct s_nodo
{
      int valor;
      struct s_nodo* sig;
};
typedef struct s_nodo* t_nodo;

struct s_cola
{
      t_nodo frente;
      t_nodo final;
};
typedef struct s_cola t_cola;
```

En el cuadro anterior podemos observar que una estructura `t_cola` contiene dos punteros, uno a cada extremo de la misma. Los prototipos de las funciones, entonces, recibirán como parámetro un puntero a una estructura `t_cola`, lo cual permitirá modificar dichos punteros internamente.

```
void queue(t_cola*, int);
int dequeue(t_cola*);
```

Ejemplo de uso:

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

...

int main()
{
    int val;
    t_cola cola = { NULL, NULL };
    do
    {
        printf("Ingrese valores, con 0 finaliza: ");
        scanf("%d", &val);
        if (val != 0)
            queue(&cola, val);
    } while ( val != 0);

    printf("\nDatos ingresados: ");
    while (cola.frente != NULL)
    {
        val = dequeue(&cola);
        printf("%d -", val);
    }
    printf("\n");
    return 0;
```

```
}
```

QUEUE

La función para encolar datos debe realizar lo siguiente:

- Crear un nuevo nodo en memoria y almacenar el elemento enviado como parámetro
- Hacer que el puntero siguiente del final de la cola apunte al nuevo nodo y mover el final a la nueva posición, o si la cola estaba vacía, hacer que frente y final apunten al nuevo nodo creado.

```
void queue(t_cola* cola, int valor)
{
    t_nodo aux = (t_nodo) malloc(sizeof(struct s_nodo));
    nodo->valor = valor;
    nodo->sig  = NULL;

    if (cola->frente == NULL && cola->final == NULL)
    {
        cola->frente = aux;
        cola->final  = aux;
    }
    else
    {
        cola->final->sig = aux;
        cola->final = aux;
    }
}
```

DEQUEUE

La función para desencolar datos debe realizar lo siguiente:

- Mover el puntero del frente de la cola al siguiente elemento.
- Almacenar el valor del nodo que está al frente de la cola y eliminar dicho nodo.
- Sólo si la cola queda vacía, hacer que el final de la cola también apunte a NULL.

```
int dequeue(t_cola* cola)
{
    int valor;
    t_nodo aux   = cola->frente;
    cola->frente = cola->frente->sig;
    valor = aux->dato;
    free(aux);
    if (cola->frente == NULL)
    {
        cola->final = NULL;
    }
    return valor;
}
```

9 - ESTRUCTURAS DINÁMICAS – LISTAS DOBLEMENTE ENLAZADAS

INTRODUCCIÓN

La primer implementación de la estructura dinámica *lista* la realizamos con nodos simplemente enlazados. Los llamamos así ya que cada nodo sólo conoce al nodo que le sigue, pero no conoce a ningún otro.

Existe una segunda implementación de listas dinámicas: listas doblemente enlazadas. En esta implementación, cada elemento conoce al nodo que le sigue, pero también conoce al nodo que le antecede. Si bien esta implementación consume más recursos de memoria que la primera versión, también ofrece mayor flexibilidad a la hora de recorrerla, agregar y quitar elementos.

ESPECIFICACIÓN

Sobre una lista doblemente enlazada es posible realizar las mismas operaciones que sobre una lista simplemente enlazada:

- Agregar elementos
- Insertar (agregar en forma ordenada)
- Eliminar
- Recorrerla

IMPLEMENTACIÓN

El nodo que utilizaremos para representar los elementos de la lista va a tener ahora 2 punteros, uno que apunte al nodo anterior y otro que apunte al nodo siguiente. En el extremo izquierdo de la lista, el puntero anterior apuntará a NULL; y en el extremo derecho, el puntero siguiente hará lo mismo.

```
struct s_nodo
{
      int valor;
      struct s_nodo* sig;
      struct s_nodo* ant;
};
typedef struct s_nodo* t_nodo;
```

Las funciones se pueden implementar de la misma manera en que se implementan funciones sobre listas simplemente enlazadas. También es posible realizar implementaciones eficientes a través de ciclos iterativos, en vez de recurrir a llamadas recursivas. Las listas doblemente enlazadas presentan una ventaja sobre las simples, ya que a través del puntero al nodo anterior, podemos “ir y venir” en ambos sentidos.

CREACIÓN DE UN NUEVO NODO

Utilizaremos una función auxiliar para crear nuevos nodos. De esta manera, nos ahorraremos las líneas de código de inicialización:

```
t_nodo nuevo_nodo(int valor)
{
      t_nodo aux = (t_nodo) malloc(sizeof(struct s_nodo));
      aux->valor = valor;
      aux->ant   = NULL;
      aux->sig   = NULL;
      return aux;
}
```

AGREGAR ELEMENTOS

Al agregar un elemento al final de la lista, no debemos olvidarnos de "linkear" los nodos anterior y siguiente. Aquí hay un ejemplo de una implementación iterativa en vez de recursiva:

```
void agregar(t_nodo* lista, int valor)
{
      t_nodo aux;
      if (*lista == NULL)
      {
            *lista = nuevo_nodo(valor);
      }
      else
      {
            aux = *lista;
            while (aux->sig != NULL)
                  aux = aux-> sig;

            aux->sig = nuevo_nodo(valor);
            aux->sig->ant = aux;
      }
}
```

En el caso de las implementaciones sin recursividad, habitualmente tendremos que agregar condiciones para distinguir si lo que estamos haciendo lo hacemos al principio, al medio, o al final de la lista. En el cuadro anterior, vemos cómo debemos controlar si la lista está vacía cuando agregamos un nodo.

Nota: es posible reemplazar estas líneas

```
      aux = *lista;

      while (aux->sig != NULL)

            aux = aux-> sig;
```

por esta:

```
      for (aux = *lista; aux->sig != NULL; aux = aux-> sig);
```

Notar el ; al final del for, el ciclo no tiene otras instrucciones internas

INSERTAR ELEMENTOS

Para insertar elementos (es decir, agregarlos de manera que la lista quede ordenada), recorreremos la lista hasta llegar al nodo frente al cual debemos efectuar la operación. Una vez allí, crearemos un nuevo nodo cuyo puntero siguiente apuntará al nodo donde estamos posicionados, y su puntero anterior al nodo anterior correspondiente. Luego debemos "relinkear" los nodos en forma apropiada (el siguiente del nodo anterior y el anterior del nodo donde estamos posicionados apuntarán al nuevo).

En el caso que la inserción se produzca al frente o al final de la lista, deberemos tener cuidado de no realizar asignaciones innecesarias de punteros (ya que el anterior y el siguiente en cada caso apuntarán a NULL).

```
void insertar(t_nodo* lista, int valor)
{
      t_nodo aux;
      t_nodo nuevo = nuevo_nodo(valor);
      if (*lista == NULL || valor < (*lista)->valor)
      {
            nuevo->sig = *lista;
            if (*lista != NULL)
            (*lista)->ant = nuevo;
            *lista = nuevo;
      }
      else
      {
            for (aux = *lista; valor > aux->valor && aux->sig != NULL;
                  aux = aux->sig);

            if (valor < aux->valor)
            {
                  nuevo->sig    = aux;
                  nuevo->ant    = aux->ant;
                  aux->ant->sig = nuevo;
                  aux->ant      = nuevo;
            }
            else
            {
                  nuevo->ant = aux;
                  aux->sig   = nuevo;
            }
      }
}
```

ELIMINAR ELEMENTOS

Para eliminar elementos de una lista doblemente enlazada, aplican las mismas consideracioens que para insertar datos: si estamos eliminando el primer nodo debemos modificar el inicio de la lista, si estamos en el medio debemos "relinkear" los nodos anteriores y siguientes, y si estamos eliminando el último elemento simplemente lo quitamos y apuntamos el último puntero a NULL.

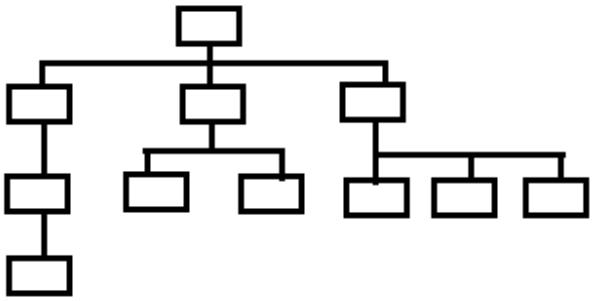
```
void eliminar(t_nodo*lista, int valor)
{
    t_nodo aux;
    for(aux = *lista; aux->valor != valor && aux->sig != NULL;
        aux = aux->sig);

    if (aux == *lista)
    {
        *lista = (*lista)->sig;
        (*lista)->ant = NULL;
        free(aux);
    }
    else if (aux->valor == valor)
    {
        if (aux->sig != NULL)
            aux->sig->ant = aux->ant;
        aux->ant->sig = aux->sig;
        free(aux);
    }
}
```

10 - ESTRUCTURAS DINÁMICAS – ARBOLES BINARIOS

INTRODUCCIÓN

Listas, pilas y colas son estructuras dinámicas “lineales”: para cada elemento de la estructura, conocemos un sucesor y tal vez un antecesor, pero solamente uno. Podríamos definir estructuras dinámicas que sirvieran para representar grafos más complejos, por ejemplo una red de ciudades interconectadas, o una red de subterráneos, o los planos de una casa, etc.

Una de las estructuras más utilizadas computacionalmente es la estructura árbol. El término no se refiere a los árboles del reino vegetal, sino a representaciones del estilo de los árboles genealógicos o los árboles que representan una jerarquía.

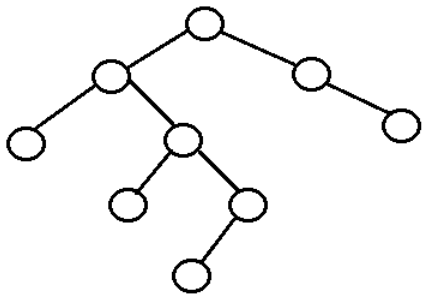


En el ejemplo anterior, el árbol está “de cabeza” y habitualmente lo representamos así. Decimos que la “raíz” del árbol es el elemento original al cual se conectan los demás. Los elementos de los extremos son las “hojas”, y los intermedios, las “ramas”.

Un tipo particular de árbol son los llamados árboles binarios. En él, cada elemento puede tener a lo sumo 2 hijos (el hijo izquierdo y el derecho).

DEFINICIONES VARIAS

Un árbol binario es una estructura en la que cada nodo conoce a lo sumo a otros dos nodos, sus hijos. Los nodos que no tienen hijos son los nodos hoja; el nodo que da origen al árbol es la raíz del mismo. Podemos pensar en un árbol como una estructura recursiva. Un árbol es o bien el árbol vacío (NULL), o bien es un nodo que contiene dos sub-árboles, el izquierdo y el derecho, y estos a su vez también son árboles.



Tanto los árboles como los árboles binarios poseen varias **propiedades** interesantes. Podemos efectuar las siguientes definiciones:

- Un árbol es **estrictamente binario** si cada nodo que no es hoja tiene hijo izquierdo y derecho no nulo (el del ejemplo anterior no sería estrictamente binario).
- El **nivel de un nodo** de un árbol binario se define de la siguiente manera: la raíz tiene nivel 0, y cualquier otro nodo tiene un nivel 1 mayor al nivel de su nodo padre.
- La profundidad de un árbol es el mayor nivel de cualquier hoja.
- Un árbol binario es **completo** si todas sus hojas están al mismo nivel.
- Un árbol binario de nivel N tiene como máximo 2^{N+1}-1 nodos. Por ejemplo, un árbol binario de nivel 3 puede contener hasta 15 nodos (1 en el nivel 0, 2 en el nivel 1, 4 en el nivel 2 y 7 en el nivel 3).
- Un árbol crece en forma exponencial, esto quiere decir que necesitamos relativamente pocos niveles para representar grandes cantidades de datos. Por ejemplo, en un árbol de profundidad 9 se pueden representar hasta 1023 elementos.
- Un árbol binario está ordenado cuando el valor del nodo raíz es mayor que cualquier valor de su subárbol izquierdo, menor que cualquier valor de su subárbol derecho, y a su vez ambos subárboles también están ordenados. Un árbol con estas características se denomina “árbol binario de búsqueda”.

ARBOLES BINARIOS DE BÚSQUEDA

Los árboles binarios de búsqueda o ABB son estructuras que permiten realiza búsquedas de datos veloces. En un ABB de N elementos, hacen falta a lo sumo log2(N) comparaciones para encontrar un dato (la demostración está fuera del alcance de lo que se explica pero es muy sencilla).

OPERACIONES

Las operaciones que podemos definir sobre un ABB son:

- Agregar datos: si el árbol está vacío, podemos insertar valores. Si no lo está, agregaremos el dato a izquierda, si el valor a agregar es menor que el valor del nodo donde estamos parados, o a derecha en caso contrario. La operación es recursiva hasta alcanzar un subárbol (nodo) vacío.
- Eliminar datos: sólo podremos eliminar nodos hojas; en caso de querer eliminar un nodo que no es hoja, deberemos eliminar todo el subárbol correspondiente.
- Recorrer un árbol: al ser una estructura no lineal, existen diferentes maneras de hacerlo. Se desarrollarán más adelante.

IMPLEMENTACIÓN

Para implementar un árbol binario utilizaremos una estructura recursiva:

```
struct s_nodo_bin
{
      int valor;
      struct s_nodo_bin* izq;
      struct s_nodo_b* der;
};
typedef struct s_nodo_bin* t_nodo_bin;
```

El algoritmo de creación/inserción de datos es como sigue:

```
void agregar(t_nodo_bin* arbol, int valor)
{
      if (*arbol == NULL)
      {
            *arbol = (t_nodo_bin)malloc(sizeof(struct s_nodo_bin));
            (*arbol)->valor = valor;
            (*arbol)->izq   = NULL;
            (*arbol)->der   = NULL;
      }
      else
      {
            if (valor < (*arbol)->valor)
                  agregar(& (*arbol)->izq, valor);
            else
                  agregar(& (*arbol)->der, valor);
      }
}
```

Este algoritmo presupone que nunca se insertan valores duplicados (esta presunción la haremos siempre para árboles binarios).

Una función booleana que sirva para determinar si un valor se encuentra en un árbol se indica a continuación:

```
int esta(t_nodo_bin arbol, int valor)
{
      if (arbol == NULL)
            return 0;
      else
      {
            if (valor == arbol-> valor)
                  return 1;
            else
                  if (valor < arbol->valor)
                        return esta(arbol->izq, valor);
                  else
                        return esta(arbol->der, valor);
      }
}
```

La eliminación de un subárbol se realiza encontrando el nodo y luego eliminando en forma recursiva el subárbol (se deja el ejercicio al lector).

RECORRIDO DE UN ÁRBOL

Es posible recorrer un árbol de diferentes maneras. Los algoritmos más usuales son 4:

- Recorrido en pre-order.
- Recorrido en in-order.
- Recorrido en post-order.
- Recorrido por niveles.

En todo caso, recorrer un árbol se trata de visitar todos sus nodos. *Visitar* es un término genérico: en los ejemplos a continuación visitar implica imprimir por pantalla el valor del nodo.

Recorridos en orden: los tres recorridos mencionados implican una operación recursiva: si el árbol está vacío, no debemos hacer nada. Si no lo está, hay que elegir una secuencia en la cual se visite el nodo raíz, y se recorran los subárboles izquierdo y derecho. Dependiendo cómo se haga obtenemos cada uno de los recorridos:

Pre order:

- Visitar el nodo padre
- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho

In Order:

- Recorrer el subárbol izquierdo
- Visitar el nodo padre
- Recorrer el subárbol derecho

Post Order:

- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho
- Visitar el nodo padre

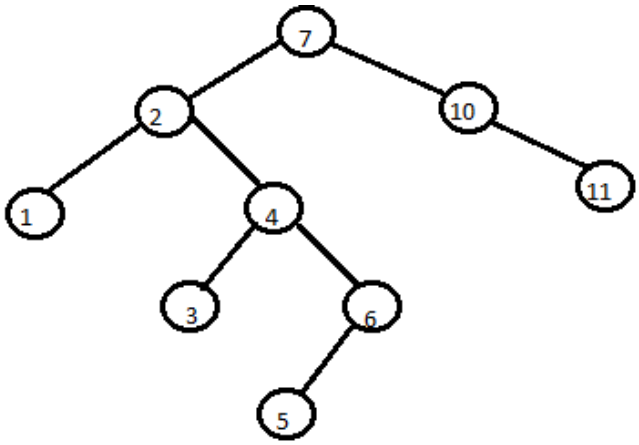
Veamos el ejemplo en código para el recorrido in-order:

```
void inorder(t_nodo_bin arbol)
{
      if (arbol != NULL)
      {
            inorder(arbol->izq);
            printf("%d - ", arbol->valor);
            inoder(arbol->der);
      }
}
```

El recorrido inorder tiene una propiedad interesante: como se recorren primero los nodos menores al nodo padre, y luego los mayores (y esto ocurre en forma recursiva para cada subárbol), al invocar este recorrido, la impresión de los nodos se realiza en forma ordenada.

RECORRIDO DE UN ÁRBOL POR NIVELES

Recorrer un árbol por niveles implica visitar primero todos los nodos del nivel 0, luego todos los nodos del nivel 1, y así sucesivamente hasta visitar las hojas del último nivel. Supongamos que visitamos los nodos de izquierda a derecha.



Si recorremos este árbol por niveles, la impresión nos queda: 7 – 2 – 10 – 1 – 4 – 11 – 3 – 6 – 5 .

No existe un algoritmo recursivo sencillo que nos permita recorrer un árbol por niveles. Sin embargo, existe uno no recursivo que utiliza una estructura auxiliar que nos permite hacerlo.

Imaginemos por un momento que estamos subiendo una gran familia a un bus. El primero en subir es el abuelo. Cuando este sube, avisa a sus hijos que hagan la cola para subirse al bus. Los hijos hacen la fila, y en turno, cuando cada uno de ellos sube, avisa a sus hijos que hagan la cola para subirse al bus. Si repetimos esta situación hasta que estén todos en la fila, veremos que el orden en que ascienden es el orden genealógico: primero el abuelo, luego sus hijos, luego los hijos del primer hijo, luegos los hijos del segundo, etc.

Este algoritmo nos permite recorrer un árbol por niveles, y se realiza utilizando una estructura auxiliar del tipo cola.

```
void por_niveles(t_nodo_bin arbol)
{
	t_cola cola = {NULL, NULL};
	t_nodo_bin nodo_aux = NULL;
	queue(&cola, arbol);

	while (cola.frente != NULL)
	{
		nodo_aux = dequeue(&cola);
		if (nodo_aux->izq != NULL)
			queue(&cola, nodo_aux->izq);
		if (nodo_aux->der != NULL)
			queue(&cola, nodo_aux->der);

		printf("%d -", nodo_aux->valor);
	}
}
```

11 - MANEJO DE ARCHIVOS

INTRODUCCIÓN

La definición tradicional dice que un archivo es el mínimo bloque de información que puede almacenarse en forma permanente en una computadora. Esta definición varía dependiendo del sistema operativo y de la arquitectura de la computadora, pero para los fines prácticos nos es suficiente. Un archivo puede pensarse como una secuencia de bytes escritos en un dispositivo de almacenamiento permanente (disco rígido, CD, diskette...); posee un nombre y una extensión, y se encuentra en una ruta de directorio. Por ejemplo: *C:\Windows\system.ini* en el sistema operativo Windows XP o Windows Vista, o */etc/passwd* en Linux / Unix.

Los sistemas operativos proveen mecanismos y utilidades que permiten crear, borrar y mover archivos, pero la lectura y escritura de sus contenidos suele realizarse con programas especializados. Los programas y aplicaciones utilizan archivos cuando desean persistir información aún cuando no se están ejecutando.

Un programa, durante su ejecución, puede leer y guardar datos en archivos. Cuando el programa se cierra, toda la información que se encuentra en memoria (tanto en variables estáticas o en estructuras dinámicas) se pierde. Pero si esta información se graba en un archivo, la siguiente vez que el programa se ejecute ésta se hallará disponible, tanto para ser leída como para ser actualizada en caso de ser necesario.

Ejemplos de usos de archivos:

- almacenar un conjunto de registros donde cada uno de ellos representa un alumno.
- almacenar una serie de líneas que registran los ingresos por teclado del usuario.
- almacenar una serie de resultados de cuentas.
- almacenar la configuración de visualización de un listado donde la primer línea contiene el número de cuántas filas se mostrarán por vez, la segunda contiene la longitud máxima por línea, y la tercera y cuarta indican las letras de inicio y fin del listado.

TIPOS DE ARCHIVOS

Los contenidos de un archivo dependen enteramente del programa que los crea. La interpretación de los datos dentro del mismo será hecha de acuerdo al formato en que la información es almacenada.

Es habitual distinguir dos tipos de archivos: **binarios** (también llamados de acceso aleatorio) y de **texto**. La principal diferencia entre ellos es que en los archivos de texto los contenidos son interpretados byte por byte como caracteres ASCII; en tanto que en los archivos binarios no puede saberse de antemano qué significa cada uno de los bytes que contiene. Los archivos binarios habitualmente almacenan un conjunto de **registros**, que son secuencias de bytes que pueden interpretarse de cierta manera. Los registros poseen todos el mismo tamaño, y es esta cualidad la que permite acceder a una posición específica del archivo: ya que sabemos cuántos bytes ocupa cada registro, si queremos leer el 10° registro del archivo podemos hacerlo directamente, yendo al byte correspondiente. Es por ello que se los llama de acceso aleatorio. Por el contrario, los archivos de texto son de acceso secuencial: un archivo de texto está constituido por líneas (que finalizan con un retorno de carro o enter), y si queremos leer una línea específica, tendremos que leer todas las anteriores ya que posiblemente la longitud de cada línea no sea fija.

OPERACIONES EN ARCHIVOS DE TEXTO

La operación de lectura se realiza de manera secuencial, empezando por el primer carácter. El lenguaje C provee funciones para leer tanto carácter por carácter como líneas completas. La operación de escritura se realiza análogamente, bien carácter por carácter o escribiendo líneas completas. En general no se agrega el carácter '\0' en ningún lugar, dado que no tiene sentido indicar el fin de una palabra/frase. La operación de actualización de un archivo de texto es engorrosa (por ejemplo si queremos borrar una palabra deberíamos desplazar el resto de los contenidos del archivo para ocupar su espacio), por lo que en la práctica no se realiza (en caso de ser necesario puede escribirse un nuevo archivo con los contenidos actualizados). Sí es posible agregar información a un archivo de texto, al final del mismo.

OPERACIONES EN ARCHIVOS BINARIOS

La operación de lectura puede hacerse en forma secuencial, empezando por el primer registro, o en forma aleatoria, posicionando el "cursor" en un byte determinado y leyendo a partir de esa posición. Para escribir en un archivo binario agregamos registros al final del mismo, y si queremos actualizar un determinado registro, también es posible hacerlo. Si deseamos borrar un registro, tenemos dos opciones: desplazar todos los otros registros de la misma manera en que lo haríamos con un archivo de texto, o "marcar" el registro con valores inválidos (e incorporar una lógica que nos permita reconocer que ese dato no es válido en nuestro programa).

FUNCIONES INCORPORADAS

El lenguaje C nos provee de un conjunto de funciones de manipulación de archivos para realizar todas las tareas mencionadas anteriormente. Algunas de las funciones son específicas para su uso con archivos binarios, otras son específicas de archivos de texto, y un tercer grupo es común a ambas.

Nota Previa: Para trabajar con las funciones de archivos es necesario incluir la biblioteca <stdio.h>. En dicha biblioteca se encuentra la definición de la estructura que utilizaremos para manipular archivos. Las funciones de manejo de archivos reciben como parámetro o retornan un puntero a una de estas estructuras (no es necesario conocer cómo está compuesta sino simplemente usarla). La estructura es llamada FILE y en nuestras funciones y programas declararemos un puntero FILE*.

Los prototipos de funciones descriptos a continuación se modificaron respecto de los originales, a efectos de simplificar la explicación. La lista completa de las funciones de manejo de archivos puede ser encontrada en Internet, un buen lugar para comenzar es aquí.

APERTURA DE ARCHIVOS - FOPEN

La primera operación que debemos realizar antes de leer o escribir en un archivo, es abrirlo. Abrir un archivo significa indicarle al sistema operativo que nuestro programa quiere hacer uso del mismo. Cuando esto ocurre, el archivo queda "bloqueado", imposibilitando que otros programas lo lean/escriban al mismo tiempo. El sistema operativo es el encargado de todo este manejo.

El prototipo de la función de apertura es **FILE* fopen(char*, char*).** fopen retorna un puntero a una estructura FILE que deberemos pasar a las demás funciones de manipulación de archivos. fopen recibe 2 parámetros, el primero es el nombre del archivo que deseamos abrir, incluyendo la ruta completa de acceso, y el segundo es una cadena de texto que indica **el modo de apertura**. Cuando abrimos un archivo, debemos indicar lo siguiente:

- si deseamos abrir el archivo en modo lectura ("r"), en modo escritura ("w"), o en modo agregar ("a").
- si deseamos abrir el archivo en forma binaria ("b")
- si deseamos actualizar el archivo ("+").

En forma práctica, los modos de apertura pueden ser:

- "r" Abre en modo lectura un archivo de texto. Si el archivo con el nombre indicado no existe, no lo crea.
- "w" Abre en modo escritura un archivo de texto. Si el archivo con el nombre indicado existe, es borrado.
- "a" Abre en modo agregar (append) un archivo de texto. Si el archivo existe, no lo borra.
- "r+" Abre en modo lectura+escritura un archivo de texto. Si el archivo no existe, no lo crea.
- "w+" Abre en modo escritura+lectura un archivo de texto. Si el archivo no existe, lo crea. Si existe, lo borra.
- "a+" Abre en modo agregar+lectura un archivo de texto. Si el archivo existe, no lo borra. Si no existe lo crea.
- "rb", "wb", "ab", "r+b", "w+b", "a+b" que hacen lo mismo, pero para archivos binarios.

En todos los casos, si la apertura del archivo falla, la función retorna NULL.

Ejemplos:

```
#include <stdio.h>

int main()
{
    FILE* fd = NULL;
    ...
    fd = fopen("C:\\Users\\Alumno\\Documents\\ejemplo.txt", "r");
    if (fd == NULL)
        printf("El archivo ejemplo.txt no existe");
    ...

#include <stdio.h>
int main()
{
    FILE* fd = NULL;
    ...
    fd = fopen("C:\\Users\\Alumno\\Documents\\ejemplo.dat", "w+b");
    if (fd == NULL)
```

```
        // No se pudo crear el archivo en la ruta indicada
        printf("El archivo ejemplo.dat no pudo ser abierto");
    ...
```

CIERRE DE ARCHIVOS - FCLOSE

Cuando terminamos de hacer nuestras operaciones de lectura / escritura sobre un archivo, debemos indicarle al sistema operativo que no deseamos seguir utilizándolo. Esto lo hacemos para liberar los recursos que el sistema operativo dedica a mantener el archivo abierto, y que de esa manera tanto dichos recursos como el archivo en sí puedan ser usados por otras aplicaciones.

El prototipo de la función de cierre de archivos es **void fclose(FILE*),** y recibe como parámetro el puntero a archivo que oportunamente fue retornado por fopen.

FIN DE ARCHIVO - FEOF

feof nos permite saber si nos encontramos ubicados al final del archivo que estamos leyendo/escribiendo. Los archivos poseen una "marca de fin de archivo" que se encuentra luego del último byte del archivo. Cuando el "cursor" con el que nos desplazamos por el archivo llega a esta marca, feof retorna 1.

El prototipo de la función es **int feof(FILE*).** Debido a que es necesario realizar una lectura adicional para encontrar la marca de fin de archivo, una estrategia muy común para leer un archivo con un ciclo de repetición es la siguiente:

```
Abrir Archivo
Leer Archivo
while (!feof( archivo ) )
{
      Realizar operaciones
      Leer Archivo
}
Cerrar Archivo
```

Otra opción es preguntar si la operación de lectura fue exitosa (las operaciones de lectura retornan cero cuando no se efectuaron correctamente)

```
Abrir Archivo
while (!feof( Archivo ) )
{
      if (Leer Archivo)
      {
          Realizar operaciones
      }
}
Cerrar Archivo
```

ARCHIVOS DE TEXTO - LECTURA

Para leer un archivo de texto nuestras opciones son: leer un caracter, leer un patrón, o leer una línea completa.

La función **fgetc** nos permite leer un carácter del archivo y desplazar el cursor en 1 posición. El prototipo de la función es **int fgetc(FILE*)**. Cuando fgetc encuentra la marca de fin de archivo, retorna la constante EOF. Tenga en cuenta que el tipo de retorno es int, ya que la constante EOF que se devuelve es entera (distinta de cualquier valor de tipo char.) Por lo tanto el uso correcto de esta función es aginar el valor de retorno a una variable de tipo int, luego verififcar que esa variable int sea distinta de EOF, y en caso afirmativo, se puede asignar a una variable de tipo char ese int, para tratarlo como letra/carácter.

La función **fscanf** nos permite leer un patrón determinado de un archivo, y guardar los contenidos leidos en una o más variables, que se pasan como parámetro. El prototipo de la función es **int fscanf(FILE*, char* , ...)** . Se puede utilizar de la misma manera que se utiliza scanf, con la diferencia que la lectura de hace desde el archivo. fscanf retorna EOF si la lectura falla.

La función **fgets** nos permite leer una línea completa (hasta encontrar el caracter '\n') y guardar los contenidos en una variable de tipo char[]. El prototipo de la función es **char* fgets(char*, int, FILE*)**. El primer parámetro es el arreglo donde se almacenará el texto leído; el segundo parámetro es la cantidad máxima de caracteres a leer (si no se encuentra '\n' antes), y el tercer parámetro es el puntero al archivo. El caracter '\n' se copia a la cadena destino. La función agrega automáticamente un '\0' al final de la cadena. Retorna un puntero a la misma cadena, o NULL si hubo algún error o se llegó al final del archivo.

ARCHIVOS DE TEXTO - ESCRITURA

Análogamente a la lectura, al momento de escribir en un archivo de texto podemos: escribir un caracter, escribir un texto formateado, o escribir una cadena completa.

La función **fputc** nos permite escribir un carácter al archivo. Su prototipo es **int fputc(char, FILE*).** El primer parámetro es el archivo a escribir y el segundo es el puntero al archivo. La función retorna EOF si el carácter no se pudo escribir.

La función **fprintf** nos permite escribir un texto formateado. Es idéntica en su funcionamiento a la función printf, pero requiere un parámetro adicional (el archivo donde se quiere escribir el texto). El prototipo es **int fprintf(FILE*, char*, ...)**; el primer parámetro es el puntero al archivo, el segundo es la cadena de formato, y los siguientes parámetros son los que se utilizan para completar la cadena. El valor de retorno, en caso que haya habido un error, es negativo; de lo contrario es positivo y es igual al número de caracteres escrito.

La función **fputs** nos permite escribir una cadena completa. Su prototipo es **int fputs(char*, FILE*)**. Si la función no puede escribir, retorna EOF. Tener en cuenta que si se desea escribir 1 línea, debemos agregar el carácter '\n' a la cadena ya que **fputs** no lo hará automáticamente por nosotros.

RESUMEN DE FUNCIONES PARA ARCHIVOS DE TEXTO

Lectura	**Escritura**
int fgetc(FILE*);	int fputc(char, FILE*);
int fscanf(FILE*, char*, ...);	int fprintf(FILE*, char*, ...);
char* fgets(char*, int, FILE*);	int fputs(char*, FILE*);

ARCHIVOS BINARIOS - LECTURA

Para leer un archivo binario, sólo tenemos una operación disponible: **fread**. El prototipo de la función es **int fread(void*, int, int, FILE*)**; donde: el primer parámetro es un puntero a una estructura o arreglo de estructuras donde se va a guardar la información leída, el segundo parámetro indica el tamaño que ocupa cada registro (generalmente, un sizeof del tipo de datos de la estructura), el tercer parámetro indica cuántos registros queremos leer, y el cuarto parámetro es el puntero al archivo. Una vez producida la lectura, el "cursor" del archivo se desplaza la cantidad de bytes leídos. La función retorna la cantidad de registros leídos, si retorna menos (o cero) es o bien que hubo un error, o que se llegó al fin de archivo.

ARCHIVOS BINARIOS - ESCRITURA

Análogamente a la lectura, para escribir en un archivo binario la operación disponible es **fwrite**. El prototipo de la función es **int fwrite(void*, int, int, FILE*)**, donde: el primer parámetro es un puntero a una estructura o arreglo de estructuras a escribir en el archivo, el segundo es el tamaño que ocupa cada registro (generalmente, un sizeof del tipo de datos de la estructura), el tercero indica cuántos registros queremos escribir, y el cuarto es el puntero al archivo. La función retorna la cantidad de registros escritos, si retorna otro número es que se produjo un error.

RESUMEN DE FUNCIONES PARA ARCHIVOS BINARIOS

Lectura	**Escritura**
int fread(void*, int, int, FILE*);	int fwrite(void*, int, int, FILE*);

OTRAS FUNCIONES DE MANEJO DE ARCHIVOS

- **ftell** nos indica la posición del cursor (en bytes) en el archivo. Sólo tiene sentido utilizarla en archivos binarios (aunque funcione en archivos de texto). Su prototipo es **long int ftell(FILE*)**.
- **fseek** nos permite desplazar el cursor del archivo a una posición específica dentro del mismo. Sólo tiene sentido utilizarla en archivos binarios. Su prototipo es **int fseek(FILE*, long int, int)**, donde:
    - el primer parámetro es el puntero al archivo;
    - el segundo parámetro indica cuántos bytes queremos movernos;
    - el tercer parámetro indica desde dónde deseamos realizar el desplazamiento. Este parámetro tiene 3 valores posibles: SEEK_CUR indica que queremos movernos desde la posición donde el cursor se encuentra, SEEK_SET indica que queremos movernos desde el principio del archivo, y SEEK_END indica que queremos desplazarnos desde el final del archivo. Teniendo en cuenta esto, para SEEK_CUR podemos usar valores positivos o negativos, con SEEK_SET sólo valores positivos o cero, y con SEEK_END, sólo valores negativos o cero.
    - La función retorna 0 en caso de ser exitosa, o un valor distinto de cero en otro caso.
- **rewind** nos permite mover el cursor hacia el principio del archivo. Su prototipo es **void rewind(FILE*).**
    - rewind puede considerarse equivalente a hacer fseek(ptr_archivo, 0, SEEK_SET).
- **fflush** nos permite enviar los datos que se mantienen en buffer hacia el archivo. No tiene sentido utilizarlo si el archivo se abrió en modo lectura. En modo escritura, cualquier dato todavía no escrito al archivo es enviado al mismo (los datos que se graban con fwrite se mantienen en un buffer para reducir la cantidad de operaciones de Entrada/Salida que se ejecutan).