APUNTADORES

Los ejemplos fueron compilados y ejecutados en Linux (Solaris)

Para errores y comentarios escribir a:
Jiménez@banxico.org.mx Jiménez@ipn.mx

¿Qué es un apuntador?

Un apuntador es un objeto que apunta a otro objeto. Es decir, una variable cuyo valor es la dirección de memoria de otra variable.

No hay que confundir una dirección de memoria con el contenido de esa dirección de memoria.

```
int x = 25;
```

Dirección	1502	1504	1506	1508			
		25					

La dirección de la variable x (&x) es 1502
El contenido de la variable x es 25

Las direcciones de memoria dependen de la arquitectura de la computadora y de la gestión que el sistema operativo haga de ella.

En lenguaje ensamblador se debe indicar numéricamente la posición física de memoria en que queremos almacenar un dato. De ahí que este lenguaje dependa de la máquina en la que se aplique.

En **C** no debemos, *ni podemos*, indicar numéricamente la dirección de memoria, si no que utilizamos una etiqueta que conocemos como variable (las variables tienen asociada una dirección de memoria). Lo que nos interesa es almacenar un dato, y no la localización exacta de ese dato en memoria.

Una variable *apuntador* se declara como todas las variables. Debe ser del mismo tipo que la variable *apuntada*. Su identificador va precedido de un asterisco (*):

```
int *punt;
```

Es una variable apuntador que apunta a variable que contiene un dato de tipo entero llamada *punt*.

```
char *car:
```

Es un apuntador a variable de tipo *carácter*.

Un apuntador tiene su **propia dirección** de memoria:

```
&punt
&car
```

Es decir: hay tantos tipos de **APUNTADORES** como tipos de datos, aunque también pueden declararse **APUNTADORES** a estructuras más complejas (funciones, *struct*, objetos (instancias de una clase), ficheros e incluso **APUNTADORES** vacíos (*void*) y **APUNTADORES** nulos (NULL).

Declaración de variables *apuntador*: Sea un fragmento de programa en **C**:

```
char dato;                  //variable que almacenará un carácter.
char *punt;                 //declaración de puntador a carácter.
punt = &dato;               //en la variable punt guardamos la dirección de memoria
                            // de la variable dato; punt apunta a dato. Ambas son del mismo tipo, char.
```

```
int *punt = NULL, var = 14;
punt = &var;
printf("%#x,    %#x", punt, &var)      //la misma salida: misma dirección
printf("\n%d,  %d", *punt, var);       //salida: 14, 14
```

Hay que tener cuidado con las direcciones *apuntadas*:

```
printf("%d,  %d", *(punt+1), var+1);
```

*(punt + 1) representa el valor contenida en la dirección de memoria aumentada en una posición (int=2 bytes), que será un valor no deseado. Sin embargo var+1 representa el valor 15. punt + 1 representa lo mismo que &var + 1 (avance en la dirección de memoria de var).

Al trabajar con APUNTADORES se emplean dos operadores específicos:

- Operador de dirección: **&** Representa la dirección de memoria de la variable que le sigue:

  &fnum representa la dirección de fnum.

- Operador de contenido o in dirección: *****. El operador * aplicado al nombre de un apuntador indica el valor de la variable apuntada:

```
 float  altura = 26.92, *apunta;
 apunta = &altura;              //inicialización del apuntador
.printf("\n%f", altura);        //salida 26.92
.printf("\n%f", *apunta);       //salida 26.92
```

No se debe confundir el operador * en la declaración del apuntador:

```
int *p;          Con el operador * en las instrucciones:

*p = 27;         printf("\nContenido = %d", *p);
```

Veamos con un ejemplo en **C** la diferencia entre todos estos conceptos

```
Es decir:    int x = 25, *pint;
             pint = &x;
```

La variable *pint* contiene la dirección de memoria de la variable x. La expresión: *pint representa el valor de la variable (x) apuntada, es decir 25. La variable *pint* también tiene su propia dirección: &pint

Veamos con otro ejemplo en **C** la diferencia entre todos estos conceptos

```
void main(void) {
int a, b, c, *p1, *p2;
void *p;
p1 = &a;                       // Paso 1. La dirección de a es asignada a p1
*p1 = 1;                       // Paso 2. p1 (a) es igual a 1. Equivale a a = 1;
p2 = &b;                       // Paso 3. La dirección de b es asignada a p2
*p2 = 2;                       // Paso 4. p2 (b) es igual a 2. Equivale a b = 2;
p1 = p2;                       // Paso 5. El valor del p1 = p2
*p1 = 0;                       // Paso 6. b = 0
p2 = &c;                       // Paso 7. La dirección de c es asignada a p2
```

```
*p2 = 3;                        // Paso 8. c = 3
printf("%d %d %d\n", a, b, c);  // Paso 9. ¿Qué se imprime?
p = &p1;                        // Paso 10. p contiene la dirección de p1
p1 = p2;                        // Paso 11. p1= p2;
*p1 = 1;                        // Paso 12. c = 1
printf("%d %d %d\n", a, b, c);  // Paso 13. ¿Qué se imprime?
   }
```

Vamos a hacer un seguimiento de las direcciones de memoria y de los valores de las variables en cada paso. Suponemos que la variable **a** es colocada en la dirección 0000, **b** en la siguiente, es decir 0002, con un *offset* de 2 bytes, por ser valores *integer*.

Se trata de un sistema de posiciones relativas de memoria. Se verá en aritmética de **APUNTADORES**.

Se obtiene el siguiente cuadro. En él reflejamos las direcciones relativas de memoria y los cambios en cada uno de los pasos marcados:

Paso	**a** 0000	**b** 002	**c** 0004	**p1** 0006	**p2** 0008	**p3** 0010
1				**0000**		
2	**1**			0000		
3	1			0000	**0002**	
4	1	2		0000	0002	
5	1	2		**0002**	0000	
6	1	**0**		0002	0002	
7	1	0		002	0004	
8	1	0	**3**	0002	0004	
9	1	0	3	0002	0004	
10	1	0	3	0002	0004	**0006**
11	1	0	3	**0004**	0004	0006
12	1	0	**1**	0004	0006	0006
13	1	0	1	0004	0004	0006

Inicialización de APUNTADORES:

< Almacenamiento > < Tipo > * < Nombre > = < Expresión >

Si <Almacenamiento> es *extern* o *static*, <Expresión> deberá ser una expresión constante del tipo <Tipo> expresado.

Si <Almacenamiento> es auto, entonces <Expresión> puede ser cualquier expresión del <Tipo> especificado.

Ejemplos:

1) La constante entera 0, NULL (cero) proporciona un apuntador nulo a cualquier tipo de dato:

```
int *p;
p = NULL;          //actualización
```

2) El nombre de un arreglo de almacenamiento *static* o *extern* se transforma según la expresión:

a) float mat[12];
   float *punt = mat;
b) float mat[12];
   float *punt = &mat[0];

3) Un "*cast*" apuntador a apuntador:

int *punt = (int *) 123.456;

Inicializa el apuntador con el entero. Esto es, en la dirección a la que apunta la variable punt se almacena el valor 123.

4) Un apuntador a carácter puede inicializarse en la forma:

char *cadena = Esto es una cadena";

5) Se pueden sumar o restar valores enteros a las direcciones de memoria en la forma:

(aritmética de **APUNTADORES**)

static int x;
int *punt = &x+2, *p = &x-1;

6) Equivalencia: Dos tipos definidos como **APUNTADORES** a objeto **P** y apuntador a objeto son equivalentes sólo si **P** y **Q** son del mismo tipo. Aplicado a matrices:

nombre_apuntador = nombre_matriz;

APUNTADORES y ARREGLOS

Sea el arreglo de una dimensión:

int mat[ ] = {2, 16, -4, 29, 234, 12, 0, 3};

En el que cada elemento, por ser tipo *int*, ocupa dos bytes de memoria. Suponemos que la dirección de memoria del primer elemento, es **1500**:

&mat[0] es 1500
&mat[1] será 1502
&mat[7] será 1514

Apuntadores y arreglos:

int mat[ ] = {2, 16, -4, 29, 234, 12, 0, 3};

En total los **8** elementos ocupan 16 bytes.

Podemos representar las direcciones de memoria que ocupan los elementos del arreglo, los datos que contiene y las posiciones del arreglo en la forma:

Dirección	1502	1504	1506	1508	1510	1512	1514
	2	16	-4	29	234	12	0	3
Elemento	mat[1]	mat[2]	mat[3]	mat[4]	mat[5]	mat[6]	mat[7]

El acceso podemos hacerlo mediante el índice:

```
x = mat[3]+mat[5];          // x = 29 + 12
```

para sumar los elementos de la cuarta y sexta posiciones.

Como hemos dicho que podemos acceder por posición y por dirección: ¿Es lo mismo &mat[0] y mat?

Y &mat[1] = mat++ ?

Veamos el código de un ejemplo:

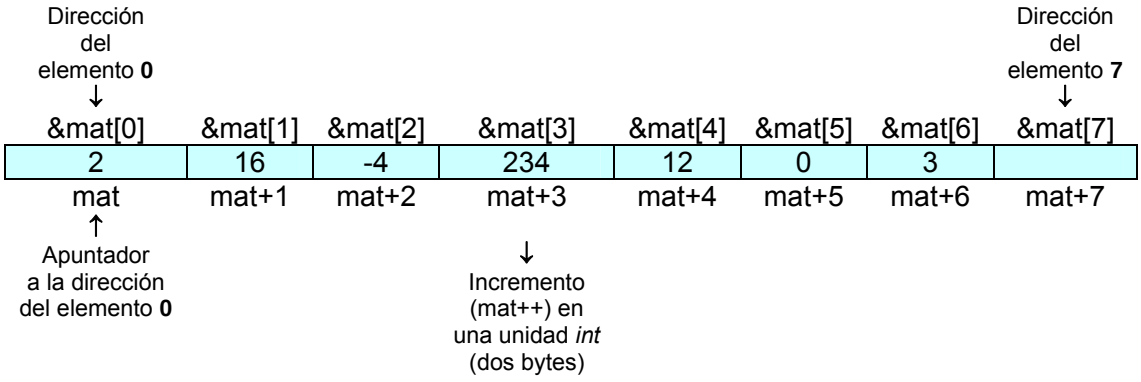
```
#include <stdio.h>
#include <conio.h>
int mat[5]={2, 16, -4, 29, 234, 12, 0, 3}, i=0; //declaradas como globales
void main()   {
        printf("\n%d", &mat[0]);          //resultado: 1500 (dirección de mem)
        printf("\n%p", mat);              //resultado: 1500 ( "  "  "  "  "  )
        i++;                              //i=1
        printf("\n%p", mat+i);            //resultado: 1502 ( "  "  "  "  "  )
        printf("\n%d", *(mat+i));         //resultado: 16 (valor de mat[1] o valor
        getch();                          //en la dirección 1502
        }
```

Parece deducirse que accedemos a los elementos del arreglo de dos formas:

- mediante el subíndice.
- mediante su dirección de memoria.

Elemento	mat[1]	mat[2]	mat[3]	mat[4]	mat[5]	mat[6]	mat[7]
2	16	-4	29	234	12	0	3

Analizando las **direcciones de memoria** del arreglo:



De lo anterior se obtienen varias conclusiones:

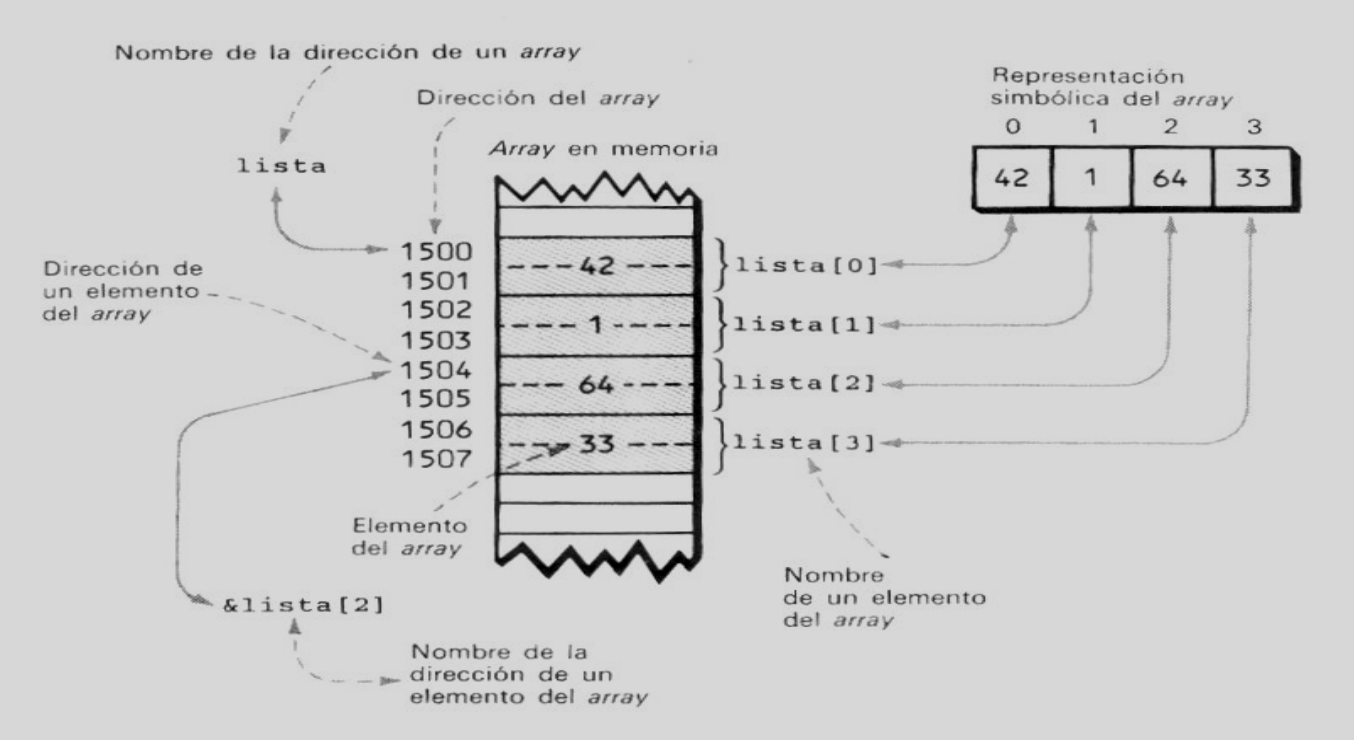
- Es lo mismo &mat[0] que mat, &mat[2] que mat + 2
- Para pasar de un elemento al siguiente, es lo mismo:

```
for(i=0; i<8; i++)
        printf("&mat [%d] = %p", i, &mat[i]);
```

que el código:

```
for(i=0; i<8; i++)
        printf("mat + %d = %p",  i,  mat + = i);
```

A esta forma de desplazarse en memoria se le llama aritmética de **APUNTADORES**



Aritmética de APUNTADORES:

A una variable apuntador se le puede asignar la dirección de cualquier objeto.

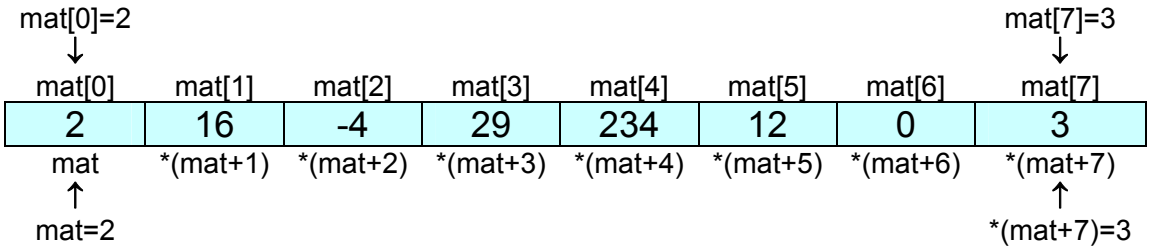
- A una variable apuntador se le puede asignar la dirección de otra variable apuntador (siempre que las dos señalen el mismo objeto)
- A un apuntador se le puede inicializar con el valor NULL
- Una variable apuntador puede ser restada o comparada con otra si ambas apuntan a elementos de un mismo arreglo.
- Se puede sumar o restar valores enteros: p++, pv+3, teniendo en cuenta que el desplazamiento (offset) depende del tipo de dato apuntado:

```
p++;            //p apunta a la siguiente dirección
pv+=3           //pv apunta 3*n bytes del dato apuntado (offset)
```

Si tenemos:

```
float *decimal;     //suponemos que apunta a 0000
decimal++;          //apunta a 0004
```

Utilizando la aritmética de **APUNTADORES** nos desplazamos de unas posiciones de memoria a otras. Pero. ¿Cómo acceder a los contenidos de esas posiciones utilizando notación de **APUNTADORES**?



Empleamos el operador *, *indirección* que nos da el contenido de la dirección de memoria apuntada.

Y ¿cómo se aplica la aritmética de **APUNTADORES** para desplazarnos en un arreglo bidimensional?:

```
float mat[2][4];          //declaración del arreglo
```

Col 0	Col 1	Col 2	Col 3
1.45	-23.5	-14.08	17.3
20	2.95	0.082	6.023

Utilizando **APUNTADORES**, la declaración será:

```
float (*mat)[4];          //arreglo bidimensional
```

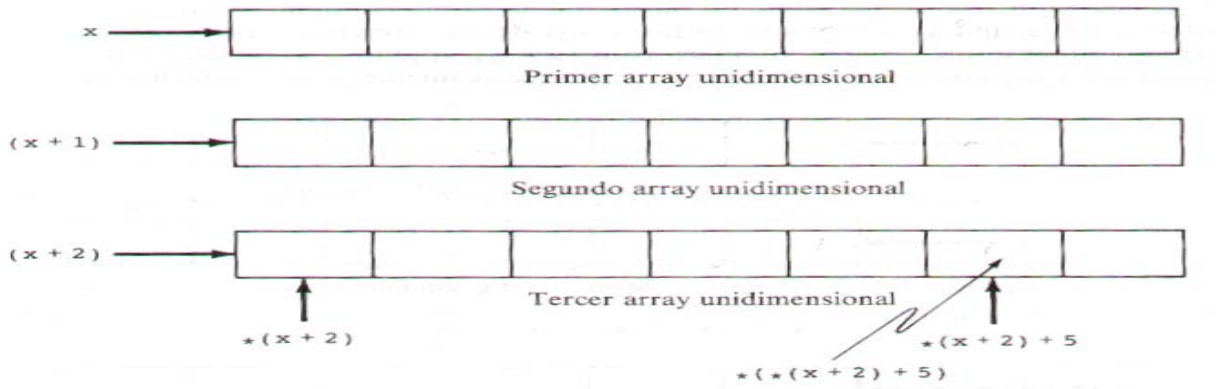
En donde *mat* es un apuntador a un grupo contiguo de arreglos monodimensionales (vectores) de 4 elementos cada uno.

Existe, por tanto una equivalencia:

Con subíndices	Con **APUNTADORES**	Valor
mat[0][0]	*(*(mat+0)+0)	1.45
mat[0][1]	*(*(mat+0)+1)	-23.5
mat[0][2]	*(*(mat+0)+2)	-14.08
mat[0][3]	*(*(mat+0)+3)	17.3
mat[1][0]	*(*(mat+1)+0)	20
mat[1][2]	*(*(mat+1)+1)	2.95
mat[1][3]	*(*(mat+1)+2)	0.082
mat[1][4]	*(*(mat+1)+3)	6.023

Recordemos que *mat representa un apuntador a la primera fila. A la segunda fila nos referimos mediante *(mat+1)+j para las direcciones y con *(*(mat+1)+j) para los contenidos. El segundo subíndice actúa sobre la columna.

Si se tiene un arreglo de x[10][20] y quiero acceder al elemento de la fila 3 y la columna 6, lo hago escribiendo x[2][5]. Con notación de **APUNTADORES**, es equivalente a * (* (x + 2) +5), ya que x + 2 es un apuntador a la fila 3. Por tanto. El contenido de dicho apuntador, *(x+2), es la fila 3. Si me desplazo 5 posiciones en esa fila llego a la posición *(x+2)+5, cuyo contenido es *(*(x+2)+5). Ver dibujo:



Si en x[10][20] quiero acceder al elemento de la fila 3 y la columna 6, lo hago escribiendo x[2][5]. Con notación de **APUNTADORES**, lo que hacemos es considerar que es un arreglo formado por 10 arreglos unidimensionales (vectores) de 20 elementos cada uno, de modo que accedo a x[2][5] mediante la expresión:

(* (x + 2) +5)

Ya que x + 2 es un apuntador a la fila 3. Por tanto. El contenido de dicho apuntador, *(x+2), es la fila 3. Si me desplazo 5 posiciones en esa fila llego a la posición *(x+2)+5, cuyo contenido es *(*(x+2)+5). Las siguientes expresiones con **APUNTADORES** son válidas:

**x → x[0][0] ; *(*(x+1)) → x[1][0]

*(*x+1) → x[0][1]; **(x+1) → x[1][0] **ver apéndice C**

Si se tiene un arreglo estático:

int arreglo[filas][columnas];

Quiero acceder al elemento arreglo[y][z] para asignarle un valor, lo que el compilador hace es:

*(*arreglo +columnas x y + z)) = 129; //asignación

Si fuera int arreglo[2][5] y quisiera asignar 129 al elemento de la fila 1 y columna 2, pondría:

*(arreglo + 5x1 + 1)) = 129;

Es decir, desde el origen del arreglo avanza 6 posiciones de memoria:

	arreglo[1][1]
fila 0 ☺ ☺ ☺ ☺ ☺	*(*(arreglo+5)+1)
fila 1 ☺ 129 ☺ ☺ ☺	*(*arreglo + 6)

APUNTADORES a ARREGLOS

Un arreglo multidimensional es, en realidad, una colección de vectores. Según esto, podemos definir un arreglo bidimensional como un apuntador a un grupo contiguo de arreglos unidimensionales. Las declaraciones siguientes son equivalentes:

int dat[fil][col] int (*dat)[col]

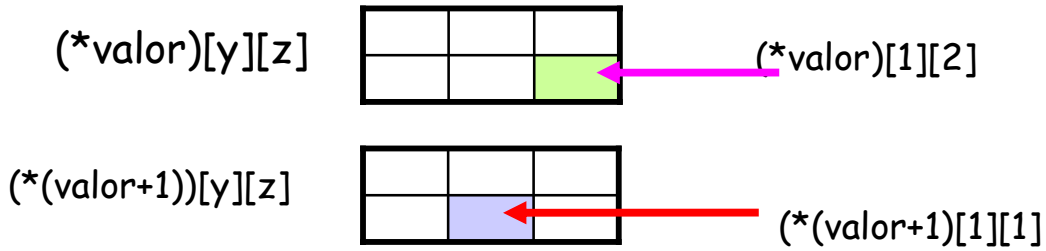
En general:

tipo_dato nombre[dim1][dim2]. . . . .[dimp]; equivale a:
tipo_dato (*nombre)[dim2][dim3]. . . . .[dimp]; -> Apuntador a un grupo de arreglos

El arreglo: int valor[x][y][z]; puede ser representado en la forma:

int (*valor)[y][z]; → Apuntador a un grupo de arreglos bidimensionales .

Sea el arreglo: valor[2][2][3]:



O como un arreglo de **APUNTADORES**:

int *valor[x][y];

sin paréntesis

En su nueva declaración desaparece la última de sus dimensiones.

Veamos más declaraciones de arreglos de **APUNTADORES**:

int x[10][20] → int *x[10];
float p[10][20][30]; → int *p[10][20]; arreglo de 200 **APUNTADORES**, cada uno de los cuales apunta a un arreglo de 30 elementos

APUNTADORES a cadenas de caracteres:

Una cadena de caracteres es un arreglo de caracteres. La forma de definir un apuntador a una cadena de caracteres:

char *cadena;

El identificador del arreglo es la dirección de comienzo del arreglo. Para saber dónde termina la cadena, el compilador añade el carácter '\0' (ASCII 0, NULL):
char *nombre = "PEPE PEREZ";

Si quiero recorrer la cadena con notación de apuntador:

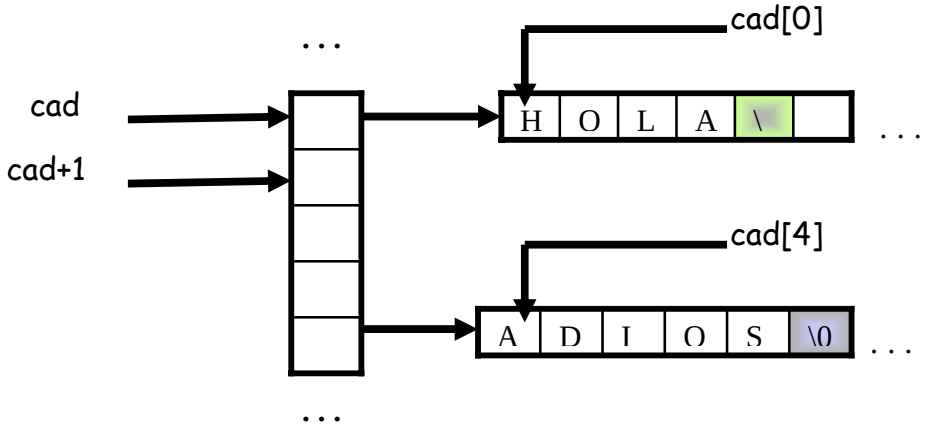
```
i = 0;
do
printf("%c", *(nombre+i);
while(*(nombre+ i ++));        //postincremento
```

↑ Condición de salida (\0)

En un arreglo de **APUNTADORES** a cadenas de caracteres cada elemento apunta a un carácter. La declaración será:

```
char *cad[10];                    //por ejemplo
```

Gráficamente podría ser:



ARREGLOS DE APUNTADORES a cadenas de caracteres:

La declaración:

```
char cad[10][80];
```

Reserva memoria para 10 cadenas de caracteres de 80 caracteres cada una. Pero en la declaración como arreglo de apuntadores:

```
char *cad[10];
```

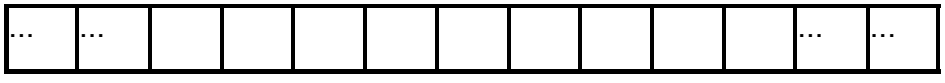
El compilador desconoce el tamaño de las cadenas: ¿cuánta memoria reserva?

- si el arreglo es *static* y se inicializan las cadenas en el propio código, el compilador calcula la dimensión no explicitada (arreglos sin dimensión explícita).
- si las dimensiones no son conocidas, sabremos dónde comienzan las cadenas, pero no dónde terminan. Para ello se efectúa la llamada reserva dinámica de memoria (funciones malloc, calloc(), realloc() y free() de stdlib.h ó alloc.h):

```
char cad[10][80];
```

Equivale a char **cad reservando 800 bytes

Inicio _____↓ _____ reserva _____....._____



Devolución de apuntadores

Cuando se utiliza una función cuyo prototipo es de la forma ***tipo * funcion(lista de parámetro)***, se dice que regresa la dirección que apunta a un objetos. Generalmente se utiliza para regresar direcciones a objetos complejos, esto es, tipos de datos especiales, tales como estructuras, arreglos, y objetos que son instancias de una clase, entre otros.

Escribamos un ejemplo. Queremos crear una estructura y regresarla mediante una función.

```
#include <math.h>
#include <fstream>
#include <iostream>

// constante que hace que se puedan
// utilizar las librerias estándar
// (PARA SOLARIS)
using namespace std;

// definición de la estructura
struct complejo
{
double Im;      //  parte imaginaria del numero complejo
double Re;      //  parte  real del numero complejo
};

// definimos un tipo
typedef struct complejo COMPLEJO;

// función que crea y lee una estructura de tipo COMPLEJO
COMPLEJO * crea_estructura(void)
{
//  declaramos una variable de tipo apuntador
//   que apunta  a una estructura del tipo struct complejo
COMPLEJO *ES_T;

//  damos memoria a la estructura
ES_T = new (COMPLEJO);

// leemos los valores de la estructura
cout<<"Dame la parte real del numero complejo: "<<endl;
cin>>ES_T->Re;

cout<<"Dame la parte imaginaria del numero complejo: "<<endl;
cin>>ES_T->Im;

// regresamos la dirección de la estructura
return(ES_T);

}
```

Una posible llamada a esta función seria:

```
// declaración de una variable de tipo apuntador a estructura COMPLEJO
COMPLEJO * ES;
```

```
// llamado a la funcion
ES =  crea_estructura();        // la variable ES toma la dirección que
                                // se creo en la función

// escribiendo los elementos de la estructura
cout<<" el numero complejo es: "<<ES->Re<<" + "<<ES->Im<<endl;
```

Recuerde que se utilaza → por ser un apuntador a estructura, si fuera una variable de tipo estructura normal se utilizaría **punto**.

Ejemplo 2: Se quiere crear un arreglo de reales del tamaño de las necesidades del usuario

```
#include <math.h>
#include <fstream>
#include <iostream>

// constante que hace que se puedan
// utilizar las librerias estándar
// (PARA SOLARIS)
using namespace std;

double * crea_arreglo(int *ta)
{
// declaramos una variable de tipo apuntador
// que apunta  a un entero
double *arreglo;
int n, i;
double x;

// leemos los valores de la estructura
cout<<"Dame la tamaño del arreglo: "<<endl;
cin>>n;

*ta=n;

// damos memoria a la estructura
arreglo = new double[n];

    for(i=0;i<n;i++)
    {
    // leemos los elementos del arreglo
    cout<<"Dame la parte real del numero complejo: "<<endl;
    cin>>x;
    // note que una vez asignada la memoria le doy el tratamiento
    //  como si fuera un arreglo
    arreglo[i]=x;
    }

    // regresamos la dirección del primer elemento del arreglo de enteros
    return(arreglo);

}
```

La función que llame a la función ***crea_arreglo*** deberá primero definir una variable de tipo apuntador a **double**, y posteriormente al llamado a dicha variable se le puede utilizar como

arreglo de reales o utilizando la aritmética de apuntadores. La sintaxis de lo anterior quedaría como:

```
double *X;          // apuntador a double que hará las veces de arreglo de reales
int ne, i;          // numero de elementos del arreglo

X= crea_arreglo(&ne);
```

Dado que el parámetro de la función espera una dirección, con el operador & le pasamos la dirección asociada a la variable **ne**. Con esto le decimos a la función que nos regrese el numero de elementos que tiene el arreglo.

```
// escritura de los elementos del arreglo
for(i=0;i<ne;i++)
cout<<" X["<<i<<"] = "<<X[i]<<endl;
```

APUNTADORES como parámetros de una función

Se pueden utilizar parámetros en una función del tipo apuntador, pero estas variables, se les puede utilizar e interpretar de diferentes maneras. En el ejemplo de de la función ***crea_arreglo*** nosotros así la creamos para que nos regrese en la dirección que le pasamos un entero, pero otra persona puede interpretar que en el parámetro nos puede regresar el arreglo, esto es, puede bien ser un arreglo.

Para ver esta situación, hagamos el mismo ejemplo de la función ***crea_arreglo*** pero ahora indicándole a la función que el arreglo lo regrese en la lista de parámetros, así como el número de elementos.

```
#include <math.h>
#include <fstream>
#include <iostream>

// constante que hace que se puedan
// utilizar las librerias estándar
// (PARA SOLARIS)
using namespace std;

void crea_arreglo(double *arreglo, int *ta)
{
    //  en los parámetros de la función se declara una variable de tipo
    //  apuntador  que apunta  a un numero real
    int n, i;
    double x;

    // leemos los valores de la estructura
    cout<<"Dame la tamaño del arreglo: "<<endl;
    cin>>n;

    *ta=n;

    // damos memoria a la estructura
    arreglo = new double[n];

    for(i=0;i<n;i++)
    {
```

```
    // leemos los elementos del arreglo
    cout<<"Dame la parte real del numero complejo: "<<endl;
    cin>>x;
    // note que una vez asignada la memoria le doy el tratamiento
    //  como si fuera un arreglo
    arreglo[i]=x;
    }

}   // fin de la función crea_arreglo
```

La función que llame a la función ***crea_arreglo*** deberá primero definir una variable de tipo apuntador a **double**, y posteriormente al llamado a dicha variable se le puede utilizar como arreglo de reales o utilizando la aritmética de apuntadores. La sintaxis de lo anterior quedaría como:

```
    double *X;          // apuntador a double que hará las veces de arreglo de reales
    int ne, i;          // numero de elementos del arreglo

    crea_arreglo(X,&ne);
```

En los parámetros de la función se tienen dos variables de tipo apuntador. Uno de ellos lo tomamos como arreglo y otro como apuntador a una sola variable de tipo entero. El primer parámetro de la función lo tomamos como el arreglo que queremos crear, dado que la variable X es de tipo apuntador y la función espera un parámetro de este tipo solo ponemos el nombre de la variable. El segundo parámetro de la función espera una dirección(una variable de tipo apuntador), con el operador & le pasamos la dirección asociada a la variable **ne**. Con esto le decimos a la función que nos regrese el numero de elementos que tiene el arreglo fuera de la función (puede ser en el main).

```
// escritura de los elementos del arreglo
for(i=0;i<ne;i++)
    cout<<" X["<<i<<"] = "<<X[i]<<endl;
```

Parámetros paso por referencia

El paso por referencia permite pasar una variable a una función y devolver los cambios que la función ha hecho al programa principal. El mismo efecto se consigue pasando un apuntador a la función. En **C++** se puede indicar al compilador que genere automáticamente un llamado por referencia en lugar de un llamado por valor para uno o mas parámetros, esto se hace utilizando la siguiente sintaxis ***tipo funcion(tipo & var);***

Veamos esto con un ejemplo: se quiere escribir una función que intercambie en el contenido de dos variables si una bandera tiene como valor 1, y que sume el contenido de dos variables si la bandera es 0.

```
#include <math.h>
#include <fstream>
#include <iostream>
#include <stdlib.h>

// constante que hace que se puedan
// utilizar las librerias estándar
// (PARA SOLARIS)
using namespace std;
```

```
// funcion que utiliza variables paso por REFERENCIA
void inter(int &a, int &b, int B)
{
     int t;

     if(B)          // si la bandera tiene un valor diferente de cero
     {
     //intercambiamos los valores de la variables
     t = a;
     a = b;
     b = t;
     }
     else
     {
     // si la variable B = 0 sumamos el contenido de las variables
     // antes escribimos el contenido de las variables
     cout<<" a: "<<a <<",  b: "<<b<<endl;
     t=a+b;
     // escribimos el contenido de la suma de las variables
     cout<<"valor de t: "<<t<<endl;
     }
}

main()
{
     //  note que declaramos dos variables normales
     int num, num2;

     // asignamos valores a las variables
     num = 123;
     num2 = 321;

// escribimos el contenido de las variables
     cout<<" num: "<<num;
     cout<<",  num2: "<<num2<<endl;

// llamamos a la funcion con la bandera = 1
     inter(num,num2,1);

// escribimos el contenido de las variables
     cout<<" num: "<<num;
     cout<<",  num2: "<<num2<<endl;

// llamamos a la funcion con la bandera = 1
     inter(num,num2,0);

}         //  fin de la funcion main
```

Observamos que:

- En el prototipo se tienen dos variables que les antecede el operador ***ampersand*** al nombre de las variables. Esto indica al compilador que estas variables deben ser tratadas como si se hubiese pasado un apuntador a las variables, es decir, que en la función se utilizarán las variables reales de la función **main**.

- En la función, las variables ***a*** y ***b*** se utilizan como cualquier otra variable. La diferencia es que se están usando las variables pasadas a la función desde el programa principal, no una copia de esas variables. La otra variable, **B**, es tratada como una variable normal, mientras que ***a*** y **b** es un sinónimo para las variables llamadas ***num*** y ***num2*** en el programa principal.
- Cuando en la funcion ***main*** a la variable **B** se le da el valor de uno, se le esta diciendo que intercambie los valores de las variables num y num2. Esto quiere decir que las variables ***a*** y **b** en la funcion, actúan como variables de tipo apuntador, por lo tanto los cambios que se hagan en la funcion ***inte***r. los resiente la función **main**.
- Cuando en la funcion ***main*** a la variable **B** se le da el valor de cero, se le esta diciendo que a las variables ***a*** y **b** las trate como variables normales (como paso por valor, esto quiere decir que el contenido de las variables, se pueden operar como cualquier otra variable.

Como conclusión se puede decir que: en las variables en las que se usa el paso por referencia, no es necesario utilizar los operadores ***** o **&**. De hecho, se produciría un error si se usan. Las variables paso por referencia, se pueden utilizar indistintamente como si fueran de tipo apuntador o como si fueran valores indistintamente.

Hay algunas restricciones que se aplican a las variables referenciadas:

1. No se puede referenciar una variable referenciada. Esto es, no se puede obtener su dirección.
2. No pueden crearse arreglos de referencia.
3. No se puede crear un apuntador a una referencia.
4. No se están permitidas las referencias en los campos de bits.

Apuntadores a funciones

Una función ocupa una posición física en la memoria, por lo que se puede referenciar por medio de una dirección, por ello se puede usar un apuntador a una función como argumento de llamado a otras funciones. Para comprender este concepto, se tiene que entender los pasos que sigue el compilador y como se hace el llamado de funciones en **C**. El primer paso de lo que llamamos compilador, es el de traducir el programa escrito en lenguaje C a un programa escrito en lenguaje de maquina. El segundo paso, es el de sustituir los llamados de las funciones por las direcciones donde empieza cada función.

En lenguaje **C** se obtiene la dirección de la función utilizando el nombre de la función sin paréntesis y sin argumentos, como si se tratara de una variable de tipo *arreglo*, como cuando se utiliza solo el nombre del arreglo, sin índices.

Veamos un ejemplo: queremos utilizar un apuntador a la función pow de **C**, que devuelve la *base* elevada un numero *exponente* dado, la sintaxis es double *pow*(double *base*, double *exp*); . La dirección de esta función es pow, y construiremos una funcion que utilice como parámetro un apuntador a función, a esta función la llamaremos: *llama_funcion* .

```
#include <math.h>
#include <fstream>
#include <iostream>

// constante que hace que se puedan
// utilizar las librerias estándar
// (PARA SOLARIS)
using namespace std;
```

```
void llama_funion(double b, double e, double (*P)(double, double) );

int main(void)
{
    double bas, pot;

    // definición del apuntador a una funcion
    double (*PO)(double,double);

    PO=std::pow;       // obtenemos la dirección de la funcion

    // lectura de la base y el exponente
    cout<<" Dame el numero base  "<<endl;
    cin>>bas;
    cout<<" Dame el numero exponente  "<<endl;
    cin>>pot;

    // llamamos a la funcion
    llama_funion(bas,pot,PO );

}                          /* fin de main */

void llama_funion(double b, double e, double (*P)(double, double) )
{

    double potencia;
    cout<<"Potencia de un numero "<<endl;

    potencia=(*P)(b,e);          // llamado a la función que entra como parámetro

    cout<<" el numero: "<<b<<" elevado a la potencia: "<<e<<" es: "<<potencia<<endl;

};
```

En el ejemplo anterior el apuntador a (*P) ejecuta la función pow que es a la que apunta. Para mas claridad también en el llamado de la función llama_funcion, se puede poner directamente el nombre de la función pow, de tal manera que el llamado quedaría como, llama_funcion(bas,exp, pow);

En algunos casos puede ser que el usuario tenga la necesidad de llamar a mas de una función, por lo que se vuelve mas complicado el codificar la opción deseada, se pueden utilizar herramientas de bifurcación, tales como el ***switch***, una serie de ***if's*** o una estructura de ***if's*** anidados.

Para tomar la decisión de llamar a una función deseada, se puede utilizar el apuntador a funciones. Para esto, plantearemos el siguiente problema: Escriba un programa que le permita al usuario, elegir entre las siguientes operaciones, calcular el factorial de un número, multiplicar dos números, y elevar un número a una potencia dada, utilizando recursividad.

Para resolver este problema, se utilizara un arreglo de apuntadores a funciones, le llamaremos opciones y su sintaxis es la siguiente:

```
double (*opciones[])(int a, int b) = {potencia ,multiplica, fact};
```

Donde:

double	: cada función regresa un numero real.
*opciones[]	: es un arreglo de apuntadores.
(int a, int b)	: indica que todas las funciones tendrán dos parámetros de tipo entero
{}	: indica cuales son los elementos del arreglo
potencia, multiplica y fac	: son las direcciones de las funciones

Construyamos el programa completo:

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

/*
programa que utilice apuntador a funciones
*/

// función que calcula el factorial de un numero de manera recursiva

double fact(int n, int k=0)
{
    if(n==1)
    return(1);                  // paso final
    else
    return(n*fact(n-1));        // paso base
}

// función que calcula la potencia de un numero
// dando su exponente, de manera recursiva

double potencia(int x, int n)
{
    if(n==0) return(1);
    else
    return(x*potencia(x,n-1));
}

// función que multiplica dos números de manera recursiva

double multiplica(int x, int n)
{
    if(n==1)return(x);
    else
    return(x+multiplica(x,n-1));
}

// función que muestra un menú y le pide al usuario que elija una opción

int menu()
{
    int op;
```

```
    do{
    printf("0.- potencia\n");
    printf("1.- multiplica\n");
    printf("2.- factorial\n");
    printf("Selecciona una opción: ");
    scanf("%d",&op);
    }while((op!=1) && (op!=0) && (op!=02));

    return(op);
}

// Prototipo para seleccionar una funcion
// utilizando apuntadores
/*
    Regresa un real
    es un arreglo de cadenas
    es una funcion que utiliza dos parámetros de tipo entero
    las funciones a las que apunta son :
    potencia
    multiplica
    fact
*/
double (*opciones[])(int a, int b) = {potencia ,multiplica, fact};

void main(void)
{
    double res;         // variable que recibe le resultado de una operación
    int n;              // variable que se utiliza para capturar la opción
    int x,y             // variables para operar

// arreglo que nos permite escribir la operación que el usuario eligió
    char texto[3][18]={"la potencia","la multiplicación","el factorial"};

// limpiamos la pantalla
    system("clear");

// pedimos los números para operar
    printf("dame el valor de x & y \n");
    scanf("%d%d",&x,&y);

// elegimos la opción
    n=menu();

// mandamos ejecutar la opción elegida
    res = (*opciones[n])(x,y);

// mandamos escribir el resultado
    printf("\n  %s   es: %.0lf \n",texto[n],res);

//  forma mas compacta
//  printf("\n  %s   es: %.0lf \n",texto[n],(*opciones[n])(x,y));
}
```

Note que la funcion que calcula el factorial también tiene dos parámetros, siendo que solo se requiere un numero, esta funcion se construyo de esta manera para poder estandarizar las fun-

ciones, también es conveniente notar que esta funcion, su segundo parámetro esta igualado a cero (int k=0), esto nos permite indicarle a la funcion que si solo le pasamos un parámetro, el orto lo tomo como cero, aunque en realidad, no lo utilizamos.

Otras clases de APUNTADORES:

➡**APUNTADORES** genéricos: Son tipo *void*:

```
void *generico;
```

Los **APUNTADORES** tipo *void* pueden apuntar a otro tipo de datos. Es una operación delicada que depende del tipo de compilador. Es conveniente emplear el "*casting*" para la conversión. Aún así, no todas las conversiones están permitidas.

➡**APUNTADOR** nulo: En **C** un apuntador que apunte a un objeto válido nunca tendrá un valor cero. El valor cero se utiliza para indicar que ha ocurrido algún error (es decir, que alguna operación no se ha podido realizar)

```
int *p = NULL;          //int *p=0;
```

➡**APUNTADORES** constantes: Una declaración de apuntador precedida de *const* hace que el objeto apuntado sea una constante (aunque no el apuntador):

```
const char *p = "Valladolid";
p[0] = 'f'              //error. La cadena apuntada por p es constante.
p = "Púlsela"           //OK.  p apunta a otra cadena.
```

Si lo que queremos es declarar un apuntador constante;

```
char *const p = "Medina";

p[0] = 's';             //error: el objeto "Medina", es cte.

p = "Peñafiel";         //error: el apuntador p es constante.
```

➡**APUNTADORES** a **APUNTADORES**:

```
int **apuntador;        //apuntador a apuntador a un objeto int.
```

El tipo de objeto apuntado después de una doble indirección puede ser de cualquier clase.

Permite manejar arreglos de múltiples dimensiones con notaciones del tipo ***mat, de múltiple in dirección que pueden generar problemas si el tratamiento no es el adecuado. Ojo a los "**APUNTADORES** locos".

➡**APUNTADORES** a datos complejos: Se pueden declara **APUNTADORES** a datos definidos por el usuario (*typedef*()), a datos *struct*, a funciones, como argumentos de funciones.

➡**Declaraciones complejas**:

Una declaración compleja es un identificador con más de de un operador. Para interpretar estas declaraciones hace falta saber que los corchetes y paréntesis (operadores a la derecha del identificador tienen prioridad sobre los asteriscos (operadores a la izquierda del identificador. Los paréntesis y corchetes tienen la misma prioridad y se evalúan de izquierda a derecha. A la

izquierda del todo el tipo de dato. Empleando paréntesis se puede cambiar el orden de prioridades. Las expresiones entre paréntesis se evalúan primero, de más internas a más externas.

Para interpretar declaraciones complejas podemos seguir el orden:

1. Empezar con el identificador y ver si hacia la derecha hay corchetes o paréntesis.
2. Interpretar esos corchetes o paréntesis y ver si hacia la izquierda hay asteriscos.
3. Dentro de cada nivel de paréntesis, de más internos a más externos, aplicar puntos 1 y 2.

Veamos un ejemplo:

```
char *(*(*var)( ))[10]
```

Aplicando los puntos anteriores, podemos decir que:

```
char *(*(*var)( ))[10]
  ↑  ↑ ↑ ↑ ↑  ↑   ↑
  7  6 4 2 1  3   5
```

La interpretación es:

1. La variable *var* es declarada como
2. un apuntador a
3. una función que devuelve
4. un apuntador a
5. un arreglo de 10 elementos, los cuales son
6. **APUNTADORES** a
7. objetos de tipo char.

Nota: Ver apéndice B

Fallas comunes con apuntadores.

A continuación se muestran dos errores comunes que se hacen con los apuntadores.

No asignar un apuntador a una dirección de memoria antes de usarlo

```
int *x *x = 100;
```

Lo adecuado será, tener primeramente una localidad física de memoria, digamos

```
int y
int *x, y; x = &y; *x = 100;
```

- In dirección no válida
- Supongamos que se tiene una función llamada *malloc()* la cual trata de asignar memoria dinámicamente (en tiempo de ejecución), la cual regresa un apuntador al bloque de memoria requerida si se pudo o un apuntador a nulo en otro caso.
- *char *malloc()* -- una función de la biblioteca estándar.
- Supongamos que se tiene un apuntador *char *p*
- Considerar:
- *p = (char *) malloc(100): /* pide 100 bytes de la memoria */
- *p = 'y';
- Existe un error en el código anterior. ¿Cuál es?

- El * en la primera línea ya que *malloc* regresa un apuntador y **p* no apunta a ninguna dirección.
- El código correcto deberá ser:
- p = (char *) malloc(100);
- Ahora si *malloc* no puede regresar un bloque de memoria, entonces *p* es nulo, y por lo tanto no se podrá hacer:
- *p = 'y';
- Un buen programa en C debe revisar lo anterior, por lo que el código anterior puede ser reescrito como:
- p = (char *) malloc(100): /* pide 100 bytes de la memoria */
- if (p == NULL) {
- printf("Error: fuera de memoria\n");
- exit(1);
- }
- *p = 'y';

Apéndice A

Listas

Los arreglos son estructuras estáticas en dónde la manipulación de datos es a través de posiciones localizadas secuencialmente. Para declarar estas estructuras se debe definir un tamaño determinado el cual no puede modificarse posteriormente. Esto puede ser un problema en el caso de que:

- No sepamos de antemano el tamaño requerido para nuestra aplicación
- hay una gran cantidad de operaciones y manipulaciones de los datos dentro de las estructuras

En estos casos es generalmente mas conveniente utilizar estructuras dinámicas, es decir, las cuales pueden aumentar o disminuir de tamaño de acuerdo a los requerimientos específicos del procedimiento. Así se resuelve el problema de no saber el tamaño exacto desde un principio y las manipulaciones de datos se pueden hacer de una manera más rápida y eficiente.

Una lista ligada es entonces un grupo de datos organizados secuencialmente, pero a diferencia de los arreglos, la organización no esta dada implícitamente por su posición en el arreglo. En una lista ligada cada elemento es un ***nodo*** que contiene el dato y además una ***liga*** al siguiente dato. Estas ligas son simplemente variables que contienen la(s) dirección(es) de los datos contiguos o relacionados. Estas variables generalmente son estructuras definidas de la siguiente forma:

```
struct nodo{
    tipo1 dato1;
    tipo2 dato2;
    struct nodo *siguiente;          // apuntador a estructura
};
```

C u ya representación grafica es la siguiente:

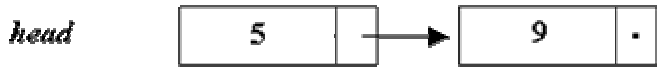
Para manejar una lista es necesario contar con un apuntador al primer elemento de la lista ***"head"***. Las ventajas de las listas ligadas son que:

- Permiten que sus tamaños cambien durante la ejecución del programa
- Tiene mayor flexibilidad en el manejo de los datos.

Este principio de listas ligadas se puede aplicar a cualquiera de los conceptos de estructura de datos: arreglos, colas y pilas, árboles, graficas, etc. Es decir, las operaciones de altas, bajas y cambios, así como búsquedas y ordenamientos se tendrán que adaptar en la cuestión del manejo de localidades únicamente.

Listas ligadas sencillas

Una lista ligada sencilla es un grupo de datos en dónde cada dato contiene además un apuntador hacia el siguiente dato en la lista, es decir, una ***liga*** hacia el siguiente dato.



Para poder dar de alta un dato en una lista ligada sencilla es necesario recorrer la lista nodo por nodo hasta encontrar la posición adecuada. Se crea un nuevo nodo, se inserta el dato y se actualizan las ligas del nodo nuevo y del anterior para intercalar el nuevo nodo en la lista.

Listas ligadas circulares

Una lista ligada circular es una lista en la cual el último nodo es ligado al primer elemento de la lista. La ventaja de este tipo de estructura es que siempre se puede llegar a cualquier nodo siguiendo las ligas. La desventaja es que si no se tiene cuidado una búsqueda puede resultar en un ciclo infinito. Esto se puede evitar al determinar a un nodo como nodo-cabeza o nodo inicial (***head***).

Listas ligadas dobles

Hasta ahora se han manejado listas que se recorren en una sólo dirección. En algunas aplicaciones es práctico o hasta indispensable poder recorrer una lista en ambas direcciones. Para estos casos se tienen las listas doblemente ligadas. Esta propiedad implica que cada nodo debe tener dos apuntadores, uno al nodo predecesor y otro al nodo sucesor.

Apéndice B

Apuntadores de mayor complejidad

int *p;	p es un apuntador a un entero
int *p[10];	p es un arreglo de 10 apuntadores a enteros
int (*p)[10];	p es un apuntador a un arreglo de 10 enteros
int *p(void);	p es una función que devuelve un apuntador a entero
int p(char *a);	p es una función que acepta un argumento que es un apuntador a carácter, devuelve un entero
int *p(char *a);	p es una función que acepta un argumento que es un apuntador a carácter, devuelve un apuntador a entero
int (*p)(char *a);	p es un apuntador a función que acepta un argumento que es un apuntador a carácter, devuelve un apuntador a entero
int (*p(char *a))[10];	p es una función que acepta un argumento que es un apuntador a carácter, devuelve un apuntador a un arreglo de 10 enteros
int p(char (*a)[]);	p es un apuntador a función que acepta un argumento que es un apuntador a

	un arreglo de caracteres, devuelve un apuntador a entero
int p(char *a[]);	p es un apuntador a función que acepta un argumento que es un arreglo de apuntadores a caracteres, devuelve un apuntador a entero
int *p(char a[]);	p es una función que acepta un argumento que es un arreglo de caracteres, devuelve un apuntador a entero
int *p(char (*a)[]);	p es una función que acepta un argumento que es un apuntador a un arreglo de caracteres, devuelve un apuntador a entero
int *p(char *a[]);	p es una función que acepta un argumento que es un apuntador a un arreglo de apuntadores a caracteres, devuelve un apuntador a entero
int (*p)(char (*a)[]);	p es una función que acepta un argumento que es un apuntador a un arreglo de caracteres, devuelve un apuntador a entero
int *(*p)(char (*a)[]);	p es un apuntador a una función que acepta un argumento que es un apuntador a un arreglo de apuntadores a caracteres, devuelve un apuntador a entero
int *(*p)(char *a[]);	p es un apuntador a una función que acepta un argumento que es un arreglo de apuntadores a caracteres, devuelve un apuntador a entero
int(*p[10])(void);	p es una arreglo de 10 apuntadores a función, cada función devuelve un entero
int (*p[10])(char * a);	p es un arreglo de 10 apuntadores a función; cada función acepta un argumento que es un apuntador a carácter y devuelve un entero
int *(*p[10])(char a);	p es un arreglo de 10 apuntadores a función; cada función acepta un argumento que es un carácter, y devuelve un apuntador a entero
char *(*p[10])(char * a);	p es un arreglo de 10 apuntadores a función; cada función acepta un argumento que es un carácter, y devuelve un apuntador a carácter.

Apéndice C

```
// Arreglos y Apuntadores
// construcción de una matriz dinámica con apuntadores

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

void main(void)
{
        int nc, nr;
        int **M;            // definición de un apuntador de apuntadores a entero

        printf("dame el numero de renglones\n");
        scanf("%d",&nr);

        // creamos un vector de apuntadores a entero
        *M=(int *)malloc(sizeof(int*)*nr);

        printf("dame el numero de columnas\n");
        scanf("%d",&nc);

        // creamos un vector de enteros de tamaño nc
        for(int i=0; i<nr; i++)
        M[i]=(int*)malloc(sizeof(int)*nc);


//  Una vez que se aparto memoria para construir el arreglote dos dimensiones,
// la variable apuntador a apuntadores se utiliza como una matriz

        // llenamos la matriz de nr renglones por nc columnas
        for(int i=0; i<nr; i++)
        for(int j=0; j<nc; j++){
        printf("dame un entero\n");
        scanf("%d",&M[i][j]);
```

```
        }

        system("clear");

        // escribimos la matriz de forma rectangular
        for(int i=0; i<nr; i++){
        for(int j=0; j<nc; j++)
        printf("%d\t",M[i][j]);
        printf("\n");
        }

}
```

Espero que haya sido de utilidad para mejorar sus programas.

Atte.

Javier Jiménez Pacheco