UPIIH
Unidad Profesional Interdisciplinaria de Ingeniería Campus Hidalgo

Unidad IV

Manejo de Apuntadores y Estructuras

Dr. Abraham Escobar Pío

Unidad IV. Manejo de Apuntadores y Estructuras

UPIIH
Unidad Profesional Interdisciplinaria de Ingeniería Campus Hidalgo

Competencia específica: Implementa programas en Lenguaje C con apuntadores para estructuras simples y complejas para la ocurrencia de procesos.

Temas

4.1 Apuntadores y variables
4.1.1 Memoria Dinámica
4.1.2 Apuntadores y arreglos
4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales
4.1.4 Direccionamiento de un apuntador a arreglos bidimensionales
4.1.5 Arreglo de apuntadores
4.1.6 Apuntadores y funciones
4.1.7 Parámetros por valor
4.1.8 Parámetros por referencia
4.1.9 Apuntadores y cadenas

4.2 Estructuras
4.2.1 Estructuras simples
4.2.2 Estructuras complejas
4.2.3 Estructura dentro de una estructura
4.2.4 Arreglo de estructuras
4.2.5 Apuntadores a estructuras

4.1 Apuntadores y variables.

4.1.1 Memoria Dinámica

Es memoria que se reserva en tiempo de ejecución. Su principal ventaja frente a la estática, es que su tamaño puede variar durante la ejecución del programa.

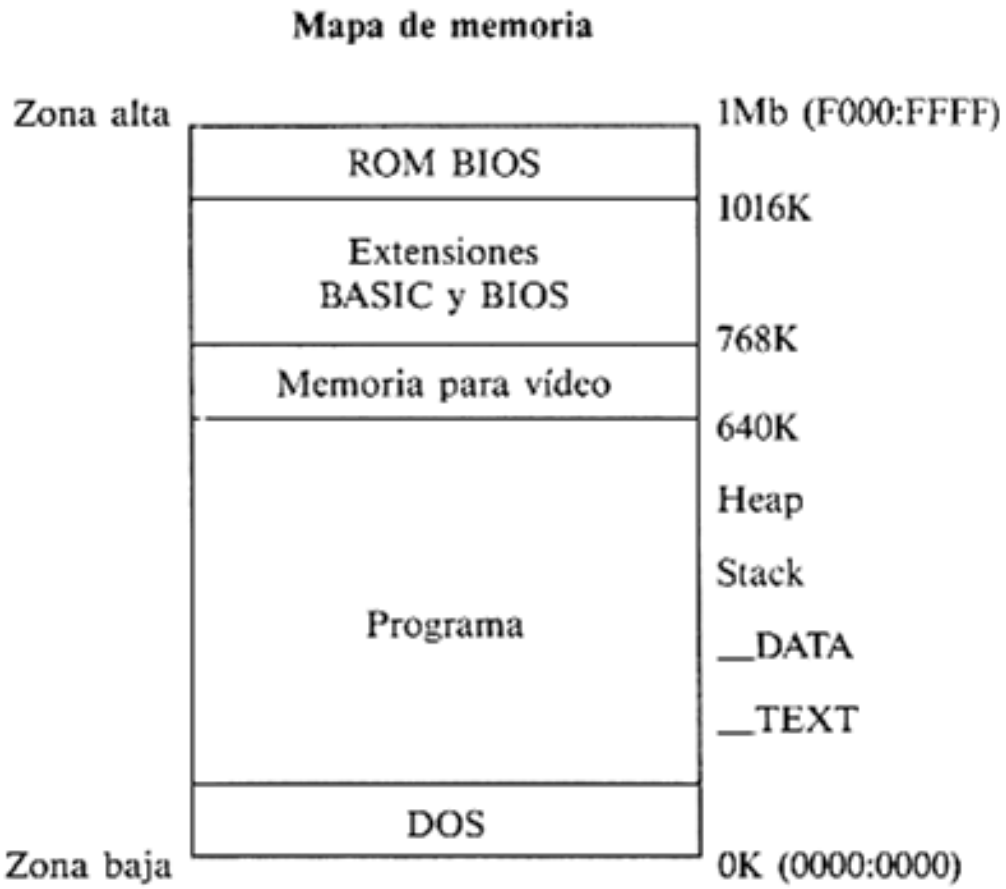
(En C, el programador es encargado de liberar esta memoria cuando no la utilice más). El uso de memoria dinámica es necesario cuando no se sabe el numero exacto de datos/elementos a tratar.

4.1 Apuntadores y variables.

4.1.1 Memoria Dinámica

El método utiliza funciones predefinidas en C, como *malloc()* y *free().*

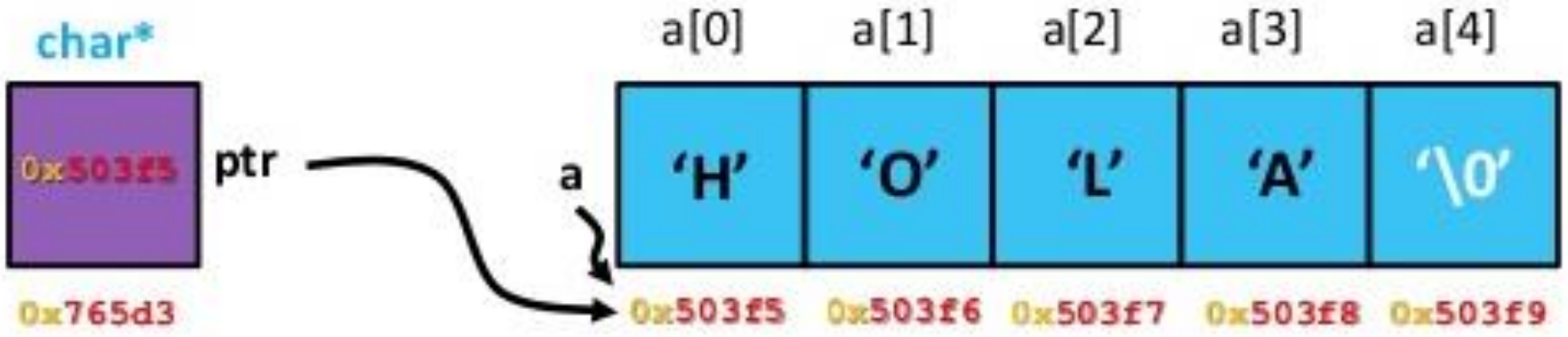
Como es lógico, estas funciones utilizan el área de memoria libre (**Heap**), para realizar las asignaciones de memoria.



4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Un apuntador es una variable que contiene la dirección de memoria de un dato o de otra variable que contiene al dato. Quiere esto decir que el apuntador apunta al espacio físico donde está el dato o la variable.



4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Un apuntador puede apuntar a un objeto de cualquier tipo, como por ejemplo, a una estructura o a una función.

Los apuntadores se pueden utilizar para referenciar y manipular estructuras de datos, para referenciar bloques de memoria asignados dinámicamente y para proveer el paso de argumentos por referencia en las llamadas a funciones.

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Cuando se trabaja con apuntadores son frecuentes los errores por utilizarlos sin haberles asignado una dirección válida; esto es, punteros que por no estar iniciados apuntan no se sabe a dónde, produciéndose accesos a zonas de memoria no permitidas.

Por lo tanto, debe ponerse la máxima atención para que esto no ocurra, iniciando adecuadamente cada uno de los punteros que utilicemos.

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

CREACIÓN DE APUNTADORES

Un apuntador es una variable que guarda la dirección de memoria de otro objeto. Para declarar una variable que sea un apuntador, la sintaxis es la siguiente:

*tipo *var-apuntador;*

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

*tipo *var-apuntador;*

En la declaración se observa que el nombre de la variable apuntador, var-apuntador, va precedido del modificador *, el cual significa “apuntador a”; tipo especifica el tipo del objeto apuntado, puede ser cualquier tipo primitivo o derivado.

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Por ejemplo, si una variable pint contiene la dirección de otra variable a, entonces se dice que pint apunta a a. Esto mismo expresado en código C es así:

```
int a = 0;    // "a" es una variable entera
int *pint;   // pint es un apuntador a un entero
pint = &a; // pint igual a la dirección de a; entonces,
                // pint apunta al entero "a"
```

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

```
int a = 0;
int *pint;
pint = &a;
```

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

```
#include <stdio.h>
int main()
{
int a = 0;
a = 10;
a = 10 - 3;
printf("%d", a);
}
```

```
#include <stdio.h>
int main()
{
int a = 0, *pint = &a;

*pint = 10;
*pint = *pint - 3;
printf("%d", *pint);
}
```

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Suponiendo definida la variable a, la definición:

```
int *pint = &a;
```

es equivalente a:

```
int *pint;
pint = &a;
```

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

En conclusión *pint es un entero que está localizado en la dirección de memoria almacenada en pint.

El espacio de memoria requerido para un apuntador es el número de bytes necesarios para especificar una dirección máquina, que normalmente son 4 bytes. Un apuntador iniciado correctamente siempre apunta a un objeto de un tipo particular. Un apuntador no iniciado no se sabe a dónde apunta.

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Operadores

Los ejemplos que hemos visto hasta ahora ponen de manifiesto que en las operaciones con apuntadores intervienen frecuentemente el operador dirección de (&) y el operador de indirección (*).

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

El operador unitario & devuelve como resultado la dirección de su operando y el operador unitario * interpreta su operando como una dirección y nos da como resultado su contenido.

Por ejemplo:

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

```
#include <stdio.h>
#include <locale.h>
int main()
{
setlocale(LC_CTYPE, “spanish”);
// Las dos líneas siguientes declaran la variable entera a,
// el apuntador q a enteros y la variable real b.
int a = 10, *q;
double b = 0.0;
q = &a; // asigna la dirección de a, a la variable q.
// q apunta a la variable entera a
b = *q; // asigna a b el valor de la variable a
printf("En la dirección %X está el dato %G \n“, q, b);
}
```

```
En la dirección 61FF10 está el dato 10
```

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

OPERACIONES CON PUNTEROS

A las variables de tipo puntero, además de los operadores &, * y el operador de asignación, se les puede aplicar los operadores aritméticos + y – (sólo con enteros), los operadores unitarios ++ y – – y los operadores de relación.

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Operación de asignación

El lenguaje C permite que un puntero pueda ser asignado a otro puntero.

Por ejemplo:

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

```
#include <stdio.h>
#include <locale.h>
int main()
{
setlocale(LC_CTYPE, “spanish”);
int a = 10, *p, *q;
p = &a;
q = p; // la dirección que contiene p se asigna a q
printf("En la dirección %X está el valor %d“, q, *q);
}
```

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Ejecución del programa:

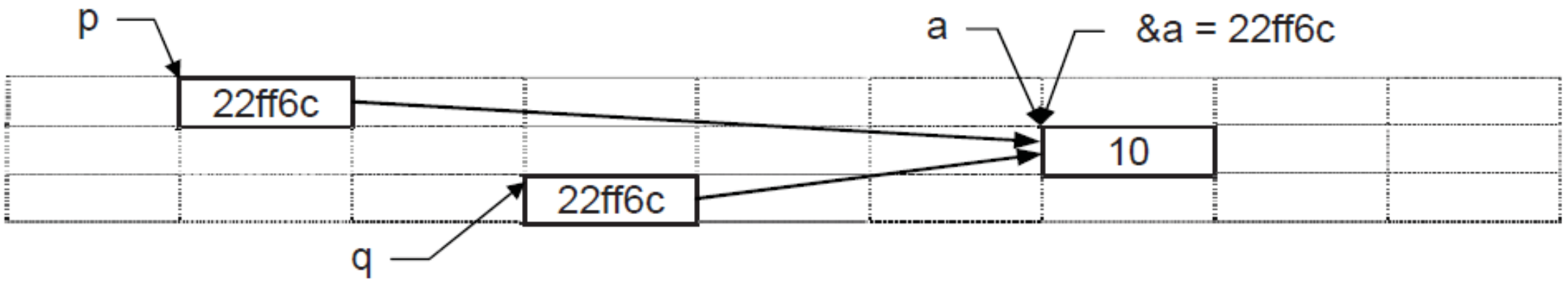
En la dirección 22ff6C

Después de ejecutarse la asignación q = p, p y q apuntan a la misma localización de memoria, a la variable a. Por lo tanto, a, *p y *q son el mismo dato; es decir, 10.

4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Gráficamente puede imaginarse esto así:



4.1 Apuntadores y variables.

4.1.2 Apuntadores y arreglos

Operaciones aritméticas

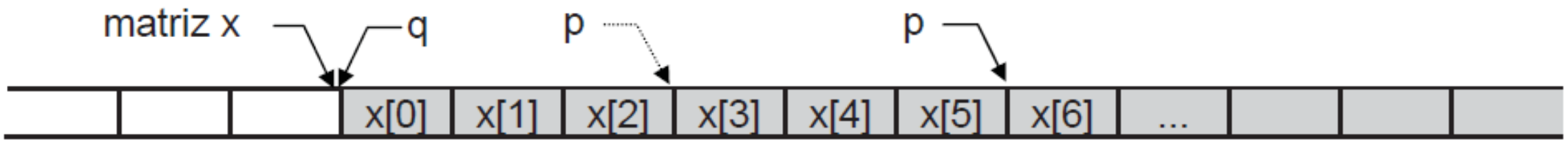
A un apuntador se le puede sumar o restar un entero. La aritmética de apuntadores difiere de la aritmética normal en que aquí la unidad equivale a un objeto del tipo del apuntador; esto es, sumar 1 implica que el apuntador pasará a apuntar al siguiente objeto, del tipo del apuntador, más allá del apuntado actualmente.

4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

Por ejemplo, supongamos que p y q son variables de tipo apuntador que apuntan a elementos de una misma matriz x:

```
int x[100];
int *p, *q; // declara p y q como punteros a enteros
p = &x[3]; // p apunta a x[3]
q = &x[0]; // q apunta a x[0]
p = p + 3; // p avanza tres enteros; ahora apunta a x[6]
```



4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

Así mismo, la operación p - q, después de la operación p = p + 3 anterior, dará como resultado 6 (elementos de tipo **int**).

La operación p - n, siendo n un entero, también es válida; partiendo de que p apunta a x[6], el resultado de la siguiente operación será el comentado.

p = p - 3; // p retrocede tres enteros; ahora apuntará a x[3]

4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

Si p apunta a x[3], p++ hace que p apunte a x[4], y partiendo de esta situación, p– – hace que p apunte de nuevo a x[3]:

```
p++; // hace que p apunte al siguiente entero; a x[4]
p--; // hace que p apunte al entero anterior; a x[3]
```

No se permite sumar, multiplicar, dividir o rotar apuntadores y tampoco se permite sumarles un real.

4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

```
int x[100], b, c, *pa, *pb;
x[50] = 10;
x[51] = 40;
x[10] = 20;
x[0] = 30;
pa = &x[50];
pb = &x[10];
b = *pa + 1;
c = *(pa + 1);
(*pb)--;
x[0] = *pb;
```

4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

```
int x[100], b, c, *pa, *pb;
x[50] = 10;
x[51] = 40;
x[10] = 20;
x[0] = 30;
pa = &x[50];
pb = &x[10];
b = *pa + 1;
c = *(pa + 1);
(*pb)--;
x[0] = *pb;
```

```
 x[50] = 10, x[10] = 19, x[0] = 19, b = 11, c = 40
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

Comparación de apuntadores

Cuando se comparan dos apuntadores, en realidad se están comparando dos enteros, puesto que una dirección es un número entero. Esta operación tiene sentido si ambos apuntadores apuntan a elementos de la misma matriz.

4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

Por ejemplo:

```
int n = 10, *p, *q, x[100];
p = &x[99];
q = &x[0];
if (q + n <= p)
q += n;
```

4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

En C existe una relación entre punteros y matrices tal que cualquier operación que se pueda realizar mediante la indexación de una matriz se puede hacer también con punteros.

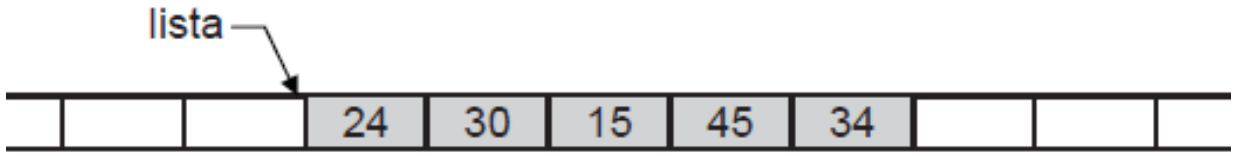
Para clarificar lo expuesto, analicemos el siguiente programa que muestra los valores de una matriz, primero utilizando indexación y después con punteros.

4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

```
#include <stdio.h>
int main()
{
int lista[] = {24, 30, 15, 45, 34};
for (int i = 0; i < 5; i++)
printf("%d ", lista[i]);
}
```

Ejecución del programa:
24 30 15 45 34

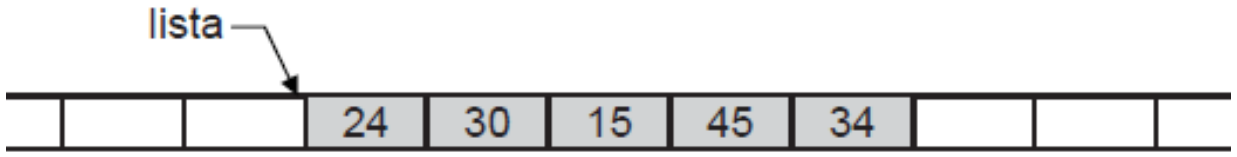


4.1 Apuntadores y variables.

4.1.3 Direccionamiento de un apuntador a arreglos unidimensionales

```
#include <stdio.h>
int main()
{
int lista[] = {24, 30, 15, 45, 34};
int *plista = &lista[0];
for (int i = 0; i < 5; i++)
printf("%d ", *(plista + ind));
}
```

Ejecución del programa:
24 30 15 45 34



4.1 Apuntadores y variables.

4.1.9 Apuntadores y cadenas

Apuntadores a Cadenas de Caracteres

Puesto que una cadena de caracteres es una matriz de caracteres, es correcto pensar que la teoría expuesta anteriormente es perfectamente aplicable a cadenas de caracteres.

4.1 Apuntadores y variables.

4.1.9 Apuntadores y cadenas

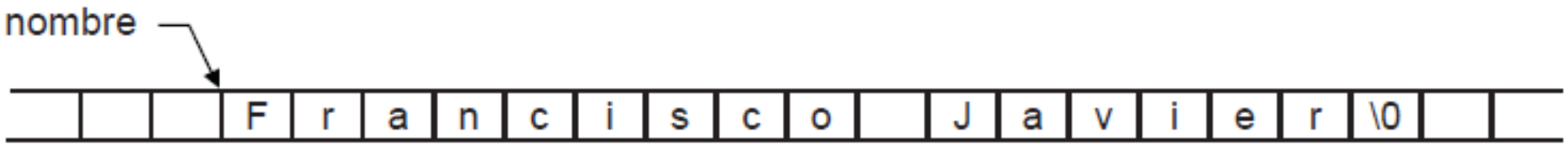
Un apuntador a una cadena de caracteres puede definirse de alguna de las dos formas siguientes:

*char *cadena;*
*unsigned char *cadena;*

4.1 Apuntadores y variables.

4.1.9 Apuntadores y cadenas

```
char cadena[] = "Francisco Javier";
char *nombre = cadena;
nombre[9] = '-'; // se modifica el elemento de índice 9
```



4.1 Apuntadores y variables.

4.1.9 Apuntadores y cadenas

```
#include <stdio.h>
int main()
{
char *nombre = "Francisco Javier";  // apuntador a una cadena de caracteres
char cadena[]= "Marco Antonio";   // un arreglo tipo char con una cadena de caracteres asignada
char *q;   // se declara un apuntador tipo char
q = cadena;  // el apuntador se direcciona al arreglo tipo char
q[7]='u';    // Se cambian letras de cadena
q[8]='r';
q[9]='e';
q[10]='l';

printf("%s \n", nombre);  // Se muestran en pantalla los de caracteres a partir del apuntador nombre
printf("%s", cadena);  // Se muestran en pantalla la cadena modificada
}
```

4.1 Apuntadores y variables.

4.1.4 Direccionamiento de un apuntador a arreglos bidimensionales

```
#include <stdio.h>
int main()
{
int lista[2][3] = {24, 30, 15, 45, 34, 40};
int *plista = &lista[0][0];
for (int i = 0; i<6; i++)
printf("%d ", *(plista + i));
}
```

Ejecución del programa:
24 30 15 45 34 40

lista

	[0]	[1]	[2]
[0]	24	30	15
[1]	45	34	40

lista

[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]
24	30	15	45	34	40

4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

Matrices de punteros.

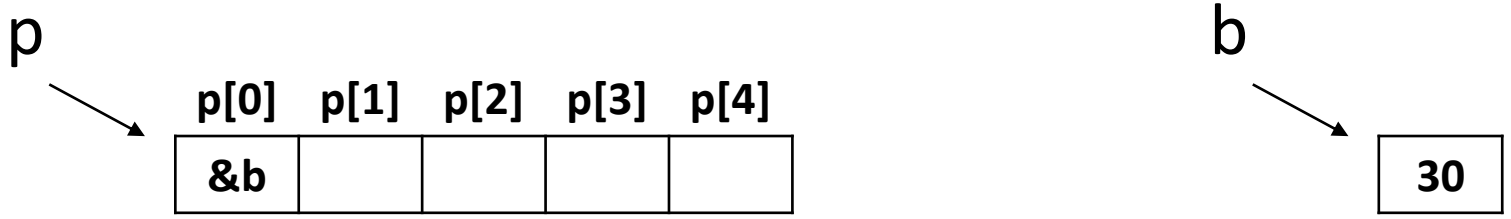
Se puede definir una matriz, para que sus elementos contengan, en lugar de un dato de un tipo primitivo, una dirección o puntero. Por ejemplo:

```
int *p[5]; // matriz de 5 elementos de tipo (int *)
int b = 30; // variable de tipo int
p[0] = &b; // p[0] apunta al entero b
printf("%d", *p[0]); // escribe 30
```

4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

```
int *p[5]; // matriz de 5 elementos de tipo (int *)
int b = 30; // variable de tipo int
p[0] = &b; // p[0] apunta al entero b
printf("%d", *p[0]); // escribe 30
```



4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

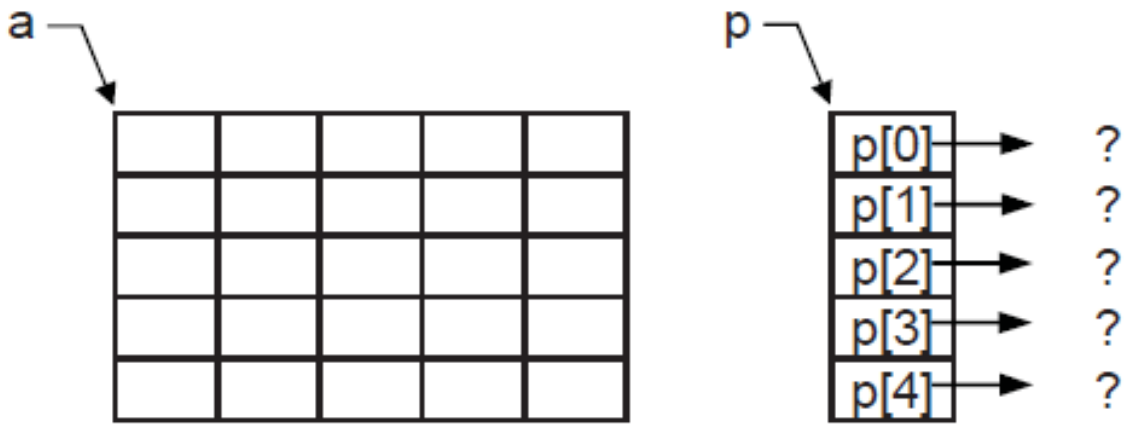
Según lo expuesto, una matriz de dos dimensiones y una matriz de punteros se pueden utilizar de forma parecida, pero no son lo mismo. Por ejemplo:

```
int a[5][5]; // matriz de dos dimensiones
int *p[5]; // matriz de punteros
```

4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

Las declaraciones anteriores dan lugar a que el compilador de C reserve memoria para una matriz a de 25 elementos de tipo entero y para una matriz p de cinco elementos declarados como punteros a objetos de tipo entero (int *).

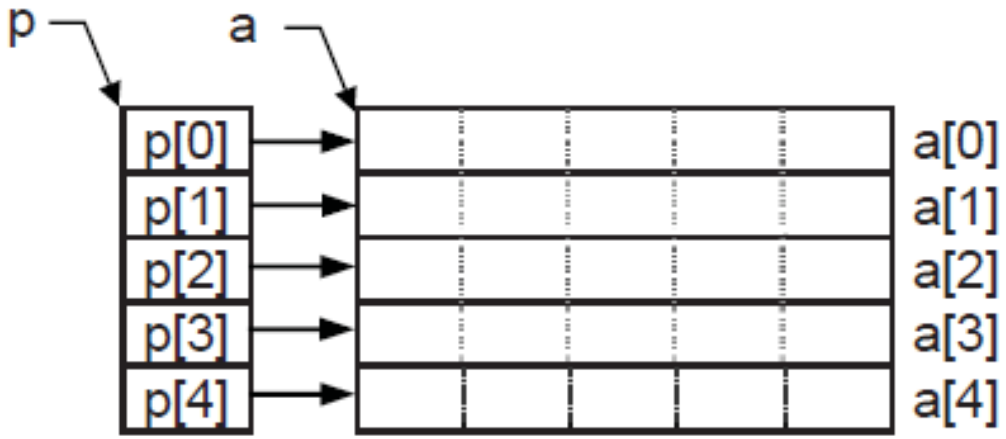


4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

Supongamos ahora que cada uno de los objetos apuntados por los elementos de la matriz p es a su vez una matriz de cinco elementos de tipo entero. Por ejemplo, hagamos que los objetos apuntados sean las filas de a:

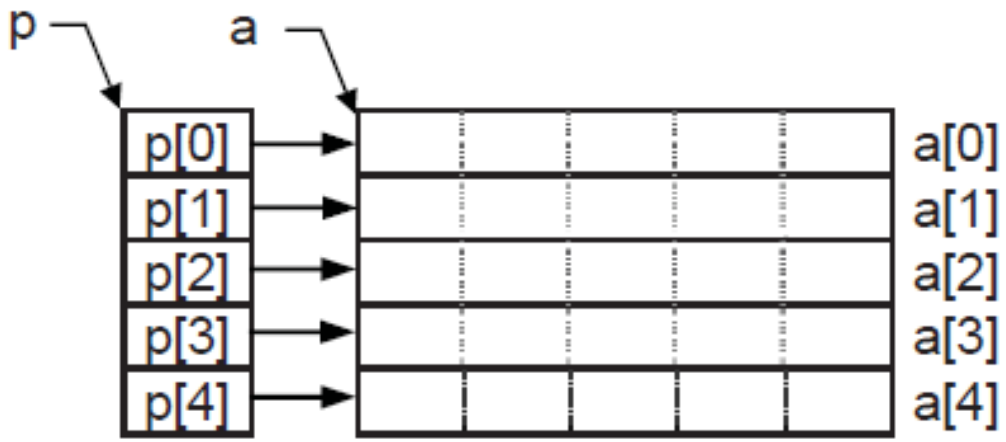
```
int a[5][5]; // matriz de dos dimensiones
int *p[5]; // matriz de punteros a int
for (int i = 0; i < 5; i++)
p[i] = a[i]; // asignar a p las filas de a
```



4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

```
printf("Ingresa valores enteros de una matriz de 5 x 5\n\n");
for (int i = 0; i < 5; i++)
    for (int j = 0; j < 5; j++)
    {
        printf("a[%d][%d] = ",i,j);
        scanf("%d", &p[i][j]);
    }
printf("\n\n");
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 5; j++)
printf("%d \t", p[i][j]);
printf("\n");
}
```



4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

Punteros a punteros

Para especificar que una variable es un puntero a un puntero, la sintaxis utilizada es la siguiente:

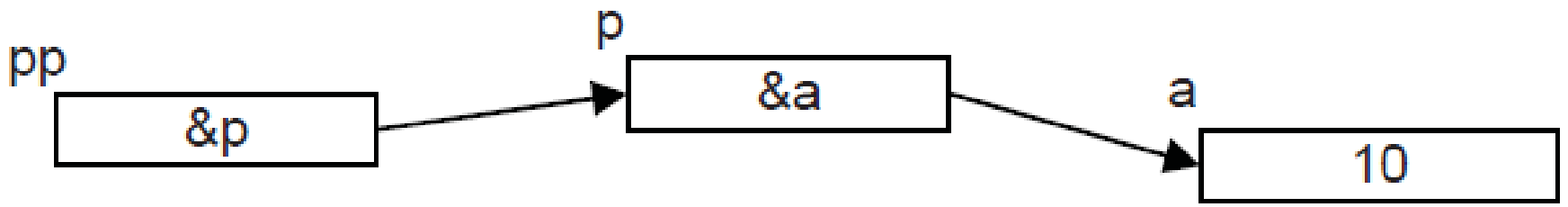
***tipo** **varpp;*

donde tipo especifica el tipo del objeto apuntado después de una doble indirección y varpp es el identificador de la variable puntero a puntero.

4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

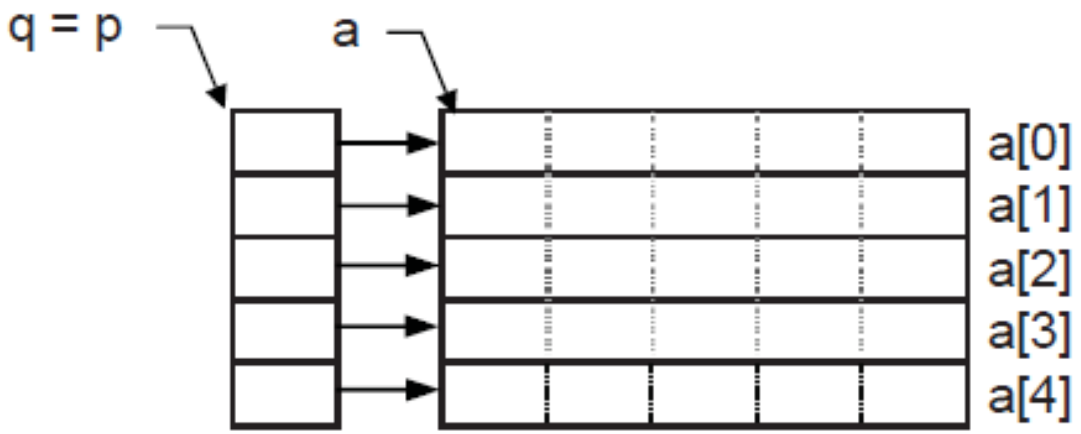
```
int a, *p, **pp;
a = 10; // dato
p = &a; // puntero que apunta al dato
pp = &p; // puntero que apunta al puntero que apunta al dato
```



4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

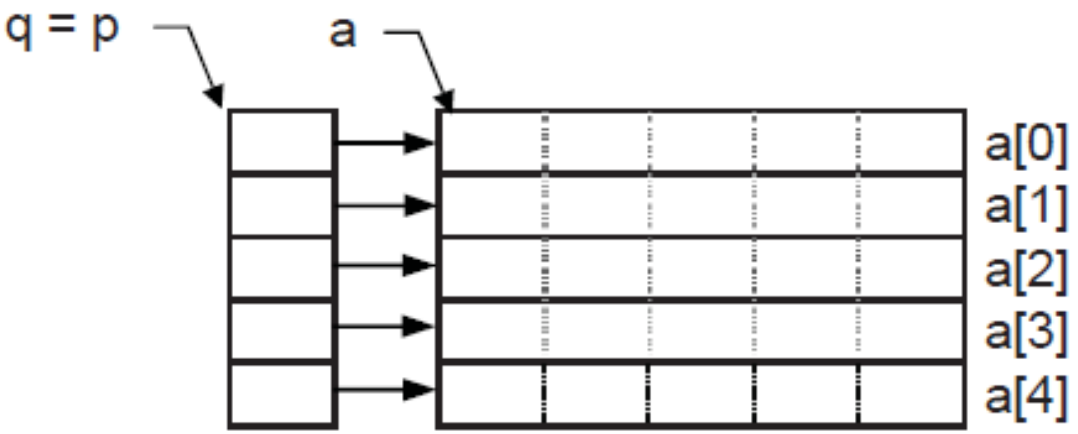
```
int i, j;
int a[5][5]; // matriz de dos dimensiones
int *p[5]; // matriz de punteros
int **q; // puntero a puntero a un entero
for (i = 0; i < 5; i++)
p[i] = a[i]; // asignar a p las filas de a
q = p;
```



4.1 Apuntadores y variables.

4.1.5 Arreglo de apuntadores

```
printf("Ingresa valores enteros de una matriz de 5 x 5\n\n");
for (int i = 0; i < 5; i++)
    for (int j = 0; j < 5; j++)
    {
        printf("a[%d][%d] = ",i,j);
        scanf("%d", &q[i][j]);
    }
printf("\n\n");
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 5; j++)
printf("%d \t", q[i][j]);
printf("\n");
}
```



4.1 Apuntadores y variables.

4.1.6 Apuntadores y funciones

Punteros a funciones

El nombre de una función representa la dirección de comienzo de la misma.

Para declarar un puntero a una función se procede así:

*tipo (*p_identf)();*

4.1 Apuntadores y variables.

4.1.6 Apuntadores y funciones

*tipo (*p_identf)();*

***tipo** = es el tipo del resultado devuelto por la función*

***p_identif** = identifica a una variable tipo puntero. Esta variable recibirá un puntero a una función, dado por el propio nombre de la función.*

4.1 Apuntadores y variables.

4.1.7 Parámetros por valor

```
#include <stdio.h>
void escribir(int a, int b)
{
   printf("%d, %d \n", a, b);
}
int main()
{
     void (*p)(int, int);
     p = escribir;
     (*p)(3,9);
}
```

4.1 Apuntadores y variables.

4.1.8 Parámetros por referencia

```
float sumar(float x, float y)
{
        float suma = 0;
        suma = x + y;
        return suma;
}
int main()
{
    float (*p1)(float,float);
    p1 = sumar;
    float dato1 = 10;
    float dato2 = 20;
    printf("%G + %G = %G", dato1, dato2, p1(dato1, dato2));
}
```

4.1 Apuntadores y variables.

Práctica 12. Apuntadores.

12.1 Ejemplos de Apuntadores.
12.2 Creación de programa para dar solución a problemas propuestos.

4.2 Estructuras

Una estructura es un nuevo tipo de datos que puede ser manipulado de la misma forma que los tipos predefinidos como float, int, char, entre otros.

Una estructura se puede definir como una colección de datos de diferentes tipos, lógicamente relacionados. En C una estructura sólo puede contener declaraciones de variables.

4.2 Estructuras

4.2.1 Estructuras simples

Crear una estructura es definir un nuevo tipo de datos, denominado **tipo estructura** y declarar una variable de este tipo.

En la definición del tipo estructura, se especifican los elementos que la componen así como sus tipos. Cada elemento de la estructura recibe el nombre de **miembro**.

4.2 Estructuras

4.2.1 Estructuras simples

```
struct tipo_estructura
{
        declaraciones de los miembros
};
```

tipo_estructura es un identificador que nombra el nuevo tipo definido.

4.2 Estructuras

4.2.1 Estructuras simples

Después de definir un tipo estructura, podemos declarar una variable de ese tipo, de la forma:

struct tipo_estructura [variable[, variable]…];

Para referirse a un determinado miembro de la estructura, se utiliza la notación:

variable.miembro

4.2 Estructuras

4.2.1 Estructuras simples

```
struct ficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
};

struct ficha var1, var2;
```

```
struct ficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
} var1, var2;
```

4.2 Estructuras

4.2.1 Estructuras simples

```
struct ficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
};
struct ficha var1, var2;
var1.telefono = 771223344;
gets(var2.nombre);
```

4.2 Estructuras

4.2.1 Estructuras simples

Ejercicio:

Crearás una estructura que contenga miembros que puedan captar el nombre de un alumno y su calificación de 4 materias. Después el usuario ingresará el nombre y la calificación por materia.

Mostrarás en pantalla el nombre y promedio del alumno.

4.2 Estructuras

4.2.1 Estructuras simples

```
#include <stdio.h>
#include <locale.h>
int main()
{
   setlocale(LC_CTYPE, "spanish");
   struct alumno
   {
      char nombre[40];
      int matematicas, quimica, fisica, humanidades;
      float promedio;
   };
   struct alumno mateo;
```

4.2 Estructuras

4.2.1 Estructuras simples

```
    printf("Ingresa el nombre del alumno: ");
    gets(mateo.nombre);
    printf("\nIngresa la Calificación de matemáticas = ");
    scanf("%d", &mateo.matematicas);
    printf("\nIngresa la Calificación de química = ");
    scanf("%d", &mateo.quimica);
    printf("\nIngresa la Calificación de física = ");
    scanf("%d", &mateo.fisica);
    printf("\nIngresa la Calificación de humanidades = ");
    scanf("%d", &mateo.humanidades);

    mateo.promedio = (mateo.matematicas + mateo.quimica + mateo.fisica + mateo. humanidades)/4.0;

    printf("\n\nEl alumno %s tiene un promedio = %G \n", mateo.nombre, mateo.promedio);
}
```

4.2 Estructuras

4.2.1 Estructuras complejas

```
typedef struct alumno
{
    char nombre[40];
    int matematicas, quimica, fisica, humanidades;
    float promedio;
}alumno;


alumno mateo;
```

4.2 Estructuras

4.2.3 Estructura dentro de una estructura

Para declarar un miembro como una estructura, es necesario haber declarado previamente ese tipo de estructura.

```
struct fecha
{
  int dia, mes, anyo;
};
struct alumno
    {
        char nombre[40];
        int matematicas, quimica, fisica, humanidades;
        float promedio;
        struct fecha fecha_alta;
    };
```

4.2 Estructuras

4.2.3 Estructura dentro de una estructura

```
struct alumno mateo;

printf("\n¿Cuál es la fecha de ingreso?\n");
printf("Año (4 dígitos) = ");
scanf("%d", &mateo.fecha_alta.anyo);

printf("\nMes (2 dígitos) = ");
scanf("%d", &mateo.fecha_alta.mes);

printf("\nDia (2 dígitos) = ");
scanf("%d", &mateo.fecha_alta.dia);

printf("\n\nFecha de ingreso, Año: %d, Mes: %d, Día: %d \n", mateo.fecha_alta.anyo,
mateo.fecha_alta.mes, mateo.fecha_alta.dia);
```

4.2 Estructuras

4.2.4 Arreglo de estructuras

Cuando los elementos de un arreglo son de tipo estructura, el arreglo recibe el nombre de arreglo de estructuras o arreglo de registros. Esta es una construcción muy útil y potente.

Del ejercicio anterior, vamos a ingresar los datos de un salón entero. (usar número bajos 2 o 3).

4.2 Estructuras

4.2.4 Arreglo de estructuras

```
#include <stdio.h>
#include <locale.h>
int main()
{
   setlocale(LC_CTYPE, "spanish");
   int num_alum;
   printf(“¿Cuántos alumnos deseas ingresar? = “);
   scanf(“%d”,&num_alum);
   struct alumno
   {
      char nombre[40];
      int matematicas, quimica, fisica, humanidades;
      float promedio;
   };
   struct alumno salon1[num_alum];
```

4.2 Estructuras

4.2.4 Arreglo de estructuras

```
for(int i=0; i<num_alum; i++)
   {
      printf("Ingresa el nombre del alumno: ");
      fflush(stdin);
      gets(salon1[i].nombre);
      printf("\n¿Cuál es la fecha de ingreso?\n");
      printf("Año (4 dígitos) = ");
      scanf("%d", &salon1[i].fecha_alta.anyo);
      printf("\nMes (2 dígitos) = ");
      scanf("%d", &salon1[i].fecha_alta.mes);
      printf("\nDia (2 dígitos) = ");
      scanf("%d", &salon1[i].fecha_alta.dia);
```

4.2 Estructuras

4.2.4 Arreglo de estructuras

```
        printf("\nIngresa la Calificación de matemáticas = ");
        scanf("%d", &salon1[i].matematicas);
        printf("\nIngresa la Calificación de química = ");
        scanf("%d", &salon1[i].quimica);
        printf("\nIngresa la Calificación de física = ");
        scanf("%d", &salon1[i].fisica);
        printf("\nIngresa la Calificación de humanidades = ");
        scanf("%d", &salon1[i].humanidades);
salon1[i].promedio = (salon1[i].matematicas + salon1[i].quimica + salon1[i].fisica + salon1[i]. humanidades)/4.0;

printf("\n\nEl alumno %s tiene un promedio = %G \n", salon1[i].nombre, salon1[i].promedio);

printf("Fecha de ingreso, Año: %d, Mes: %d, Día: %d \n\n\n", salon1[i].fecha_alta.anyo, salon1[i].fecha_alta.mes,
salon1[i].fecha_alta.dia);
}
```

4.2 Estructuras

4.2.5 Apuntadores a estructuras

Los punteros a estructuras se declaran igual que los punteros a otros tipos de datos. C utiliza el operador -> para referirse a un miembro de una estructura apuntada por un puntero.

Ejemplo:

El siguiente ejemplo declara un puntero ***hoy*** a una estructura, asigna un valor a cada miembro de la misma y, apoyándose en una función, escribe su contenido.

4.2 Estructuras

4.2.5 Apuntadores a estructuras

```
#include<stdio.h>
#include<stdlib.h>
#include<locale.h>
struct fecha
{
   unsigned int dd;
   unsigned int mm;
   unsigned int aa;
};
void escribir(struct fecha *f)
{
   printf("Día %u del mes %u del año %u \n", f->dd, f->mm, f->aa);
}
```

4.2 Estructuras

4.2.5 Apuntadores a estructuras

```
int main()
{
   setlocale(LC_CTYPE, "spanish");
   struct fecha *hoy;

   hoy = (struct fecha *)malloc(sizeof(struct fecha));

   printf("Introducir fecha (dd-mm-aa): ");
   scanf("%u %u %u", &hoy->dd, &hoy->mm, &hoy->aa);
   escribir(hoy);
}
```

4.2 Estructuras

Práctica 13. Estructuras.

13.1 Ejemplos de Estructuras.
13.2 Creación de programa para dar solución a problemas propuestos.