Licenciatura en Sistemas
Programación de computadoras

Equipo docente: Jorge Golfieri, Natalia Romero, Romina Masilla y Nicolás Perez
Mails: jgolfieri@hotmail.com , nataliab_romero@yahoo.com.ar , romina.e.mansilla@gmail.com, nperez_dcao_smn@outlook.com
Facebook: https://www.facebook.com/groups/171510736842353
Git: ***http://***github.com/UNLASistemasProgramacion/Programacion-de-Computadoras

---

Unidad 6:

Punteros. Concepto. Tipos de punteros. Punteros como argumentos de funciones. Manejo dinámico de memoria (malloc, free). Estructuras dinámicas. Problemas: implementación y testeo.

Bibliografía citada:

Brian W. Kernighan y Dennis M. Ritchie (1991) - El Lenguaje de programación C - Pearson Educación. Capítulo 5.

Luis Joyanes Aguilar, Andrés Castillo Sanz, y Lucas Sánchez García (2005) - C algoritmos, programación y estructuras de datos - McGraw-Hill España. Capítulo 12 y 14.

B. Kernighan and D. Ritchie - The C Programming Language - 2nd Edition - Prentice Hall.

Punteros – Guía Teórica

Si uno quiere ser eficiente escribiendo código en el lenguaje de programación C, se debe tener un profundo y activo conocimiento del uso de los punteros.

¿Qué es un puntero?:

Son variables que contienen la dirección de otra variable.

Los usos principales, que tienen, los punteros, son los siguientes:

-Nos ayuda, para que una función devuelva más de un valor. Por ejemplo, una función que devuelva un vector de enteros, en dicha función mandamos la dirección del primer elemento a la función principal, y a partir de ella, imprimimos todos los valores contenidos en el vector.

-Mejor uso de la **memoria dinámica.** Esto es lo que más nos tiene cuenta, el lector debe tener presente que, el **uso de punteros ayuda a ahorrar memoria y por consiguiente, hace más efectivo el uso y administración de la misma.**

¿Cómo se declara?

int k;

int *ptr;

ptr es el nombre de nuestra variable. El * informa al compilador que lo que queremos es una variable puntera, es decir, que se reserven los bytes necesarios para alojar una dirección en la memoria. **Lo de "int" significa que queremos usar nuestra variable puntero para almacenar la dirección de un entero. Se dice entonces que dicho tipo de puntero "apunta" a un entero.**

Supongamos ahora que queremos almacenar en ptr la dirección de nuestra variable entera k. Para hacerlo hacemos uso del operador unitario & y escribimos:

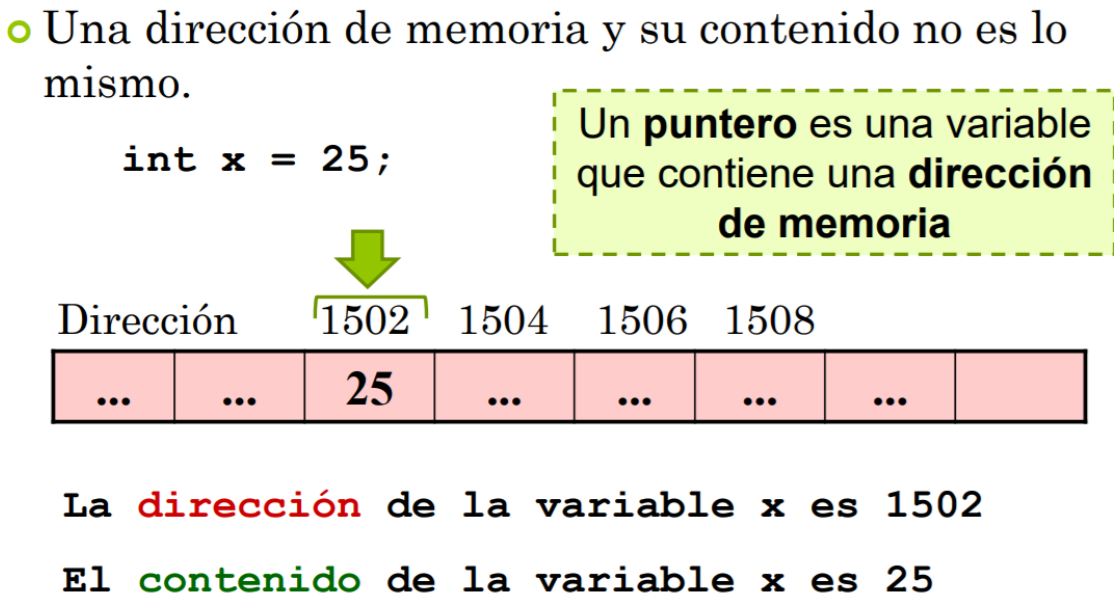
ptr = &k;

Lo que el operador & hace es obtener la dirección de k, aún cuando k está en el lado derecho del operador de asignación '=' y copia esa dirección en el contenido de nuestro puntero ptr. Ahora, ptr es un "puntero a k".

Hay un operador más que discutir**: El operador de "indirección" (o de desreferencia) es el asterisco y se usa como sigue:**

*ptr = 7;

esto copiará el 7 a la dirección a la que apunta ptr. Así que como ptr “apunta a” (contiene la dirección de) k, la instrucción de arriba asignará a k el valor de 7. Esto es, que cuando usemos el '*' hacemos referencia al valor al que ptr está apuntando, no el valor de el puntero en sí.



Ejemplo 1:

```
#include <stdio.h>
int j, k;
int *ptr;
int main (void)
{
 j = 1;
 k = 2;
 ptr = &k;
 printf("\n");
 printf("j tiene el valor: %d y esta alojado en: %p\n", j, (void *)&j);
 printf("k tiene el valor: %d y esta alojado en: %p\n", k, (void *)&k);
 printf("ptr tiene el valor: %p y esta alojado en: %p\n", ptr, (void *)&ptr);
 printf("El valor del entero al que apunta ptr es: %d\n", *ptr);
 return 0;
```

}

```c
#include <stdio.h>
int main()
{   int *ptr;
    int dato=30;

    ptr = &dato;
    *ptr = 50;

    printf("Dato = %d\n", dato);

    return 0;
}
```

& es el **operador de dirección**: permite obtener la dirección de memoria de la variable que le sigue

Tipos de punteros:

Consideremos por qué tenemos que identificar el "tipo" de variable a la que apunta un puntero como en:

int *ptr;

Una de las razones para hacer esto es que una vez que ptr apunta a algo y si escribimos:

*ptr = 2;

El compilador sabrá cuantos bytes va a copiar en la posición de memoria a la que apunta ptr. Si ptr fuera declarado como un puntero a entero, se copiarían 4 bytes. De modo similar para números de punto flotante (float) y enteros dobles (doubles), se copiaría el número apropiado de bytes. Pero definir el tipo al que el puntero apunta permite un cierto número de maneras interesantes en que el compilador puede interpretar el código. Por ejemplo, consideremos un bloque de memoria consistente en 10 números enteros en una fila. Eso es 40 bytes de memoria son reservados para colocar 10 enteros.

Digamos que ahora apuntamos nuestro puntero entero ptr al primero de estos números enteros. Es más, supongamos que este primer entero está almacenado en la posición de memoria 100 (decimal). Entonces que pasa cuando escribimos:

ptr + 1;

Ya que el compilador “sabe” que este es un puntero (que su valor es una dirección de memoria) y que apunta a un entero (su dirección actual: 100, es la dirección donde se aloja un entero), añade 4 a ptr en lugar de 1, así que ptr apunta al siguiente entero.

Punteros y arreglos:

int mi_arreglo[] = {1,23,17,4,-5,100};

Tenemos entonces un arreglo conteniendo seis enteros. Nos referimos a cada uno de estos enteros por medio de un subíndice a mi_arreglo, es decir usando mi_arreglo[0] hasta mi_arreglo[5]. Pero podemos acceder a ellos de un modo alternativo usando un puntero de esta manera:

int *ptr;

ptr = &mi_arreglo[0]; /* apuntamos nuestro puntero al primer entero de nuestro arreglo */

Y entonces podemos imprimir los valores de nuestro arreglo, ya sea usando la notación de arreglos o "desreferenciando" nuestro puntero.

Ejemplo 2:

```
#include <stdio.h>
int mi_arreglo[] = {1,23,17,4,-5,100};
int *ptr;
int main(void)
{
 int i;
 ptr = &mi_arreglo[0]; /* apuntamos nuestro puntero  al primer elemento del arreglo*/
 printf("\n\n");
 for (i = 0; i < 6; i++)
 {
 printf("mi_arreglo[%d] = %d ", i, mi_arreglo[i]);
printf("ptr + %d = %d\n",i, *(ptr + i));
}
 return 0;
}
```

*Void *:*

C nos ofrece un puntero de tipo void (carente de tipo). Podemos declarar un puntero de este tipo al escribir algo como:

void *vptr;

Un puntero void es una especie de puntero genérico. Por ejemplo, mientras C no permite la comparación entre un puntero del tipo entero con uno del tipo carácter, cada uno de estos puede ser comparado con un puntero del tipo void.

Punteros y cadenas (String):

Consideremos, por ejemplo:

char mi_cadena[40];

mi_cadena [0] = 'T';

mi_cadena [1] = 'e';

mi_cadena [2] = 'd':

mi_cadena [3] = '\0';

Que desde ya es lo mismo que:

char mi_cadena [40] = {'T', 'e', 'd', '\0',};

char mi_cadena [40] = "Ted";

Veamos como funcionaria entonces los punteros para entender esto último:

Ejemplo 3:

```
#include <stdio.h>
char strA[80] = "Cadena a usar para el programa de ejemplo";
char strB[80];
int main(void)
{
 char *pA; /* un puntero al tipo caracter */
 char *pB; /* otro puntero al tipo caracter */
 puts(strA); /* muestra la cadena strA */
 pA = strA; /* apunta pA a la cadena strA */
 puts(pA); /* muestra a donde apunta pA */
 pB = strB; /* apunta pB a la cadena strB */
 putchar('\n'); /* dejamos una línea en blanco */
 while(*pA != '\0') /* linea A (ver texto) */
```

```
{
*pB++ = *pA++; /* linea B (ver texto) */
}
*pB = '\0'; /* linea C (ver texto) */
puts(strB); /* muestra strB en la pantalla */
return 0;
}
```

¿Qué realiza este código?:

Lo que hicimos arriba fue comenzar por definir dos arreglos de 80 caracteres cada uno. Ya que estos son definidos globalmente, son inicializados a \0 primeramente. Luego strA tiene sus primeros 42 caracteres inicializados a la cadena que está entre comillas.

Ahora, yendo al código, declaramos dos punteros a caracter y mostramos la cadena en pantalla. A modo ilustrativo utilizamos PUT en vez de PRINTF; después apuntamos con el puntero pA a strA. Esto quiere decir que, por el significado de la operación de asignación, copiamos la dirección de memoria de strA[0] en nuestra variable puntero pA. Usamos entonces puts() para mostrar lo que estamos apuntando con pA en la pantalla. Consideremos aquí que el prototipo de la función puts() es:

int puts(const char *s);

Por el momento ignoremos eso de “const”. El parámetro pasado a puts() es un puntero, esto es, el valor del un puntero (ya que en C todos los parámetros son pasados por valor), y ya que el valor de un puntero es la dirección de memoria a la que apunta, o , para decirlo simple: una dirección. Así que cuando escribimos:

puts(strA);

como hemos visto, estamos pasando la dirección de strA[0].

De modo similar cuando hacemos: puts(pA); estamos pasando la misma dirección, ya que habíamos establecido que pA = strA;

Sigamos examinando el código hasta el while() en la línea A:

- La línea A indica: “Mientras el caracter apuntado por pA (es decir: *pA) no sea un caracter nul (el que es ‘\0’), haz lo siguiente”

- La línea B indica: “copia el caracter apuntado por pA (es decir *pA) al espacio al que apunta pB, luego

incrementa pA de tal manera que apunte al siguiente caracter, de igual modo incrementa pB de manera que apunte al siguiente espacio”

Una vez que hemos copiado el último caracter, pA apunta ahora a un caracter nul de terminación de cadena y el ciclo termina. Sin embargo, no hemos copiado el caracter de

terminación de cadena. Y, por definición: una cadena en C debe terminar en un caracter nul. Así que agregamos nul con la línea C.

Resulta realmente didáctico ejecutar este programa en un depurador, mientras se observa strA, strB, pA y pB e ir recorriendo cada paso del programa, también es bueno probar inicializando strB[] a una cadena en lugar de hacerlo simplemente declarándole; puede ser algo como:

strB[80] = "12345678901234567890123456789012345678901234567890"

Donde el número de dígitos sea mayor que la longitud de strA y luego repite la observación paso a paso mientras observas el contenido de las variables.

Volviendo al prototipo para puts() por un momento, la "const" usada como parámetro informa al usuario que la función no modificará a la cadena apuntada por s, es decir que se tratará a esa cadena como una constante.

Desde luego, lo que hace el programa de arriba es una manera simple de copiar una cadena con otra.

Punteros y estructuras:

Enfoquémonos un poco en las estructuras, que será lo que más útil nos vendrá. Consideremos que tenemos una estructura para guardar datos de un paciente, por ejemplo, la estructura Ficha.

Declaramos dicho puntero con la declaración:

struct ficha *st_ptr;

Y hacemos que apunte a nuestra estructura de ejemplo con:

st_ptr = &mi_ficha;

Ahora podremos acceder a un miembro de la estructura desreferenciando el puntero.

Pero ¿Cómo desreferenciamos un puntero a estructura? Bueno, consideremos el hecho de que queramos usar el puntero para cambiar la edad del paciente. Para esto escribiríamos:

(*st_ptr).edad = 63;

Sin embargo, esta notación no es muy usada y los creadores de C nos han brindado la posibilidad de utilizar una sintaxis alternativa y con el mismo significado, la cual sería:

st_ptr -> edad = 63;

Ejemplo 4:

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


//creamos la estructura que guardara el nombre, apellido, edad y salario
struct ficha{
 char nombre[20]; /* nombre */
 char apellido[20]; /* apellido */
 int edad; /* edad */
 float salario; /* salario */
 };

//luego creamo una variable del tipo struc ficha
struct ficha mi_ficha; /* definimos mi_ficha del tipo estructura ficha */

//declaramos el procedimiento, Mostrar
void mostrar(struct ficha *p); /* prototipo de la función */


int main(void)
{

 struct ficha *st_ptr; /* un puntero a una estructura del tipo ficha */
 st_ptr = &mi_ficha; /* apuntamos el puntero a mi_ficha */

//Agregamos los datos
strcpy(mi_ficha.apellido,"Perez");
strcpy(mi_ficha.nombre,"Nicolas");
mi_ficha.edad = 28;
```

```
mostrar(st_ptr); /* Llamamos a la función pasándole el puntero */

return 0;
}

void mostrar(struct ficha *p)
{
printf("\n%s ", p -> nombre); /* p apunta a una estructura */
printf("%s \n", p -> apellido);
printf("%d\n", p -> edad);
}
```

Matrices y punteros:

Aquí no tengo muchas más cosas nuevas que comentar, pero si me gustaría que vean este programa en funcionamiento.

Ejemplo 6:

```
#include <stdio.h>
#define RENGLONES 5
#define COLUMNAS 10
int multi[RENGLONES][COLUMNAS];

int main(void)
{
 int renglon, columna;
 for (renglon = 0; renglon < RENGLONES; renglon++)
 {
 for (columna = 0; columna < COLUMNAS; columna++)
 {
 multi[renglon][columna] = renglon*columna;
 }
```

```
}
for (renglon = 0; renglon < RENGLONES; renglon++)
{
for (columna = 0; columna < COLUMNAS; columna++)
{
printf("\n%d ",multi[renglon][columna]);
printf("%d ",*(*(multi + renglon) + columna));
}
}
return 0;
}
```

Debido a la doble desreferencia requerida en la versión de puntero, se dice que el nombre de una matriz bidimensional es equivalente a un puntero a puntero. Con arreglos de 3 dimensiones estaríamos hablando de arreglos de arreglos de arreglos y entonces el nombre de tal sería el equivalente de un puntero a puntero a puntero.

Gestión dinámica de la memoria:

Hay veces en que resulta conveniente reservar memoria en tiempo de ejecución usando malloc(), calloc(), o cualquier otra función de reservación de memoria.

Usar este método permite posponer la decisión del tamaño del bloque de memoria necesario para guardar, por ejemplo, un arreglo, hasta el tiempo de ejecución. O permitirnos usar una sección de la memoria para guardar un arreglo de enteros en un tiempo determinado, y posteriormente, cuando esa memoria no sea necesaria, liberarla para otros usos, como para guardar un arreglo de estructuras.

Cuando la memoria es reservada, las funciones de reservación de memoria (como malloc() o calloc(), etc.) regresan un puntero.

Si usas uno de estos compiladores antiguos, y quieres reservar memoria para un arreglo de enteros, tienes que hacer la conversión (cast) del puntero tipo char a un puntero de tipo entero (int). Por ejemplo, para reservar espacio para 10 enteros, escribiríamos:

```
int *iptr;
iptr = (int *)malloc(10 * sizeof(int));
if (iptr == NULL)
{ .. Rutina del manejo de error va aquí .. }
```

Si estamos utilizando un compilador compatible con ANSI, malloc() regresa un puntero del tipo void y ya que un puntero de este tipo puede ser asignado a apuntar a una variable de cualquier tipo de objeto, el cast convertidor (int *) mostrado en el código expuesto arriba no es necesario. La dimensión del arreglo puede ser determinada en tiempo de ejecución por lo que no es necesario conocer este dato en tiempo de compilación.

Esto significa que el 10 de arriba puede ser una variable leída desde un archivo de datos, desde el teclado, o calculada en bas e a una necesidad, en tiempo de ejecución.

Debido a la equivalencia entre la notación de arreglos y la notación de punteros, una vez que el puntero iptr ha sido asignado como arriba, podemos usar la notación de arreglos. Por ejemplo, uno puede escribir:

int k;

for (k = 0; k < 10; k++)

ptr[k] = 2;

para establecer el valor de todos los elementos a 2.

Aún con un buen entendimiento de los punteros y de los arreglos, es usual que algo que hace tropezar a los novatos en C sea la asignación dinámica de memoria para arreglos multidimensionales. En general, nos gustaría ser capaces de acceder a los elementos de dichos arreglos usando notación de arreglos, no notación de punteros, siempre que sea posible. Dependiendo de la aplicación podemos o no conocer las dimensiones de un arreglo en tiempo de compilación. Esto nos conduce a una variedad de caminos a seguir para resolver nuestra tarea.

Como hemos visto, cuando alojamos dinámicamente un arreglo unidimensional, su dimensión puede ser determinada en tiempo de ejecución. Ahora que, para el alojamiento dinámico de arreglos de orden superior, nunca necesitaremos conocer la primera dimensión en tiempo de compilación. Si es que vamos a necesitar conocer las otras dimensiones depende de la forma en que escribamos el código. Vamos a discutir sobre varios métodos de asignarle espacio dinámicamente a arreglos bidimensionales de enteros. Para comenzar consideremos el caso en que la segunda dimensión es conocida en tiempo de compilación:

METODO 1:

Una manera de enfrentar este problema es usando la palabra clave typedef. Para alojar arreglos de 2 dimensiones, recordemos que las siguientes dos notaciones dan como resultado la generación del mismo código objeto:

multi[renglon][columna] = 1;

((multi + renglon) + columna) = 1;

También es cierto que las siguientes dos notaciones dan el mismo resultado:

multi[renglon] *(multi + renglon)

Ya que la que está a la derecha debe evaluarse a un puntero, la notación de arreglos a la izquierda debe hacerlo también. De hecho multi[0] retornará un puntero al primer entero en el primer renglón, multi[1] un puntero al primer entero del segundo renglón, etc. En realidad multi[n] se evalúa como un puntero a ese arreglo de enteros que conforma el n-

ésimo renglón de nuestro arreglo bidimensional. Esto significa que podemos pensar en multi como un arreglo de arreglos y multi[n] como un puntero al n-ésimo arreglo de este arreglo de arreglos. Aquí la palabra puntero (puntero) es usada para representar el valor de una dirección.

Mientras que este uso es común en los libros, al leer instrucciones de este tipo, debemos ser cuidadosos al distinguir entre la dirección constante de un arreglo y una variable puntero que es un objeto que contiene datos en sí misma.

Veamos ahora el:

```
#include <stdio.h>
#include <stdlib.h>
#define COLUMNAS 5
typedef int Arreglo_de_renglones[COLUMNAS];
Arreglo_de_renglones *rptr;
int main(void)
{
 int nrenglones = 10;
 int renglon, columna;
 rptr = malloc(nrenglones * COLUMNAS * sizeof(int));
 for (renglon = 0; renglon < nrenglones; renglon++)
 {
 for (columna = 0; columna < COLUMNAS; columna++)
 {
 rptr[renglon][columna] = 17;
 }
 }
 return 0;
}
```

METODO 2:

En el método 1 de arriba, rptr se volvió un puntero del tipo “arreglo unidimensional de COLUMNAS de enteros”. Es evidente entonces que existe una sintaxis para usar este tipo sin la necesidad de usar la palabra clave typedef. Si escribimos:

```
int (*xptr)[COLUMNAS];
```

las variable xptr tendría las mismas características que la variable rptr del método 1, y no habremos usado la palabra clave typedef. Aquí xptr es un puntero aun arreglo de enteros y el tamaño de ese arreglo está dado por la constante COLUMNAS. Los paréntesis hacen que la notación de punteros predomine, a pesar de que la notación de arreglo tiene una mayor precedencia de evaluación. Es decir que si hubiéramos escrito:

int *xptr[COLUMNAS];

habríamos definido a xptr como un arreglo de punteros consistente en un número de punteros igual a la cantidad definida por COLUMNAS. Es obvio que no se trata de lo mismo. Como sea, los arreglos de punteros tienen utilidad al alojar dinámicamente arreglos bidimensionales, como veremos en los siguientes dos métodos.

METODO 3:

Consideremos el caso en el que no conozcamos el número de elementos por cada renglón en tiempo de compilación, es decir que el número de renglones y el número de columnas será determinado en tiempo de ejecución. Un modo de hacerlo sería crear un arreglo de punteros de tipo entero (int) y luego reservar memoria para cada renglón y apuntar estos punteros a cada renglón. Consideremos:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
 int nrenglones = 5; /* Ambos, nrenglones y ncolumnas pueden ser */
 int ncolumnas = 10; /* evaluados o leídos en tiempo de ejecución */
 int renglon;
 int **renglonptr;
 renglonptr = malloc(nrenglones * sizeof(int *));
 if (renglonptr == NULL)

 {
 puts("\nError al reservar espacio para punteros de renglon.\n");
 exit(0);
 }
 printf("\n\n\nIndice Puntero(hex) Puntero(dec) Dif.(dec)");
 for (renglon = 0; renglon < nrenglones; renglon++)
 {
```

```
 renglonptr[renglon] = malloc(ncolumnas * sizeof(int));
 if (renglonptr[renglon] == NULL)
 {
 printf("\nError al reservar memoria para el renglon[%d]\n",renglon);
 exit(0);
 }
 printf("\n%d %p %d", renglon, renglonptr[renglon],
renglonptr[renglon]);
 if (renglon > 0)
 printf(" %d",((int)renglonptr[renglon] –
(int)renglonptr[renglon-1]));
 }
 return 0;
}
```

METODO 4:

Con este método reservamos un bloque de memoria para contener primero el arreglo completo. Después creamos un arreglo de punteros para apuntar a cada renglón. Así, aunque estamos usando un arreglo de punteros, el arreglo real en memoria es continuo. El código es este:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
 int **rptr;
 int *aptr;
 int *pruebaptr;
 int k;
 int nrenglones = 5; /* Ambos, nrenglones y ncolumnas pueden ser */
 int ncolumnas = 8; /* evaluados o leidos en tiempo de ejecucion */
 int renglon, columna;
```

```
/* ahora reservamos memoria para el arreglo completo */
aptr = malloc(nrenglones * ncolumnas * sizeof(int));
if (aptr == NULL)
{
puts("\nError al reservar memoria para el arreglo completo.");
exit(0);
}
/* ahora reservamos espacio para los punteros a renglones */
rptr = malloc(nrenglones * sizeof(int *));
if (rptr == NULL)
{
puts("\nError al reservar memoria para los punteros");
exit(0);
}
/* y ahora hacemos que los punteros “apunten” */
for (k = 0; k < nrenglones; k++)
{
rptr[k] = aptr + (k * ncolumnas);
}
/* Ahora demostramos que los punteros a renglones se han incrementado */
printf("\nDemostramos que los punteros a renglones se han incrementado:");
printf("\n\nIndice Puntero(dec) Diferencia(dec)");
for (renglon = 0; renglon < nrenglones; renglon++)
{
printf("\n%d %d", renglon, rptr[renglon]);
if (renglon > 0)
printf(" %d",((int)rptr[renglon] – (int)rptr[renglon-1]));
}
printf("\n\nY ahora mostramos el arreglo:\n");
for (renglon = 0; renglon < nrenglones; renglon++)
{
for (columna = 0; columna < ncolumnas; columna++)
```

```
 {
 rptr[renglon][columna] = renglon + columna;
 printf("%d ", rptr[renglon][columna]);
 }
 putchar('\n');
 }
puts("\n");
 /* Y aquí es donde demostramos que efectivamente estamos manejando un
 arreglo bidimensional contenido en un bloque continuo de memoria */
 printf("Demostrando que los elementos son continuos en memoria:\n");
 pruebaptr = aptr;
 for (renglon = 0; renglon < nrenglones; renglon++)
 {
 for (columna = 0; columna < ncolumnas; columna++)
 {
 printf("%d ", *(pruebaptr++));
 }
 putchar('\n');
 }
 return 0;
}
```

Punteros a funciones:

Hasta este punto hemos discutido el uso de punteros con objetos de datos. C permite también la declaración de punteros a funciones. Los punteros a funciones tienen variedad de usos y algunos de estos serán expuestos aquí.

```c
/* parámetro por referencia */
#include <stdio.h>
void cuadrado(int *);
int main()
{   int a = 5;

    printf("Valor original = %d\n", a);

    cuadrado(&a);

    printf("Valor al cuadrado = %d\n", a);

    return 0;
}
void cuadrado(int * nro)
{
     *nro = *nro * *nro;
}
```

Envía la dirección de la variable (un puntero)

Recibe un puntero a un entero

Valor de la variable apuntada por **nro**

Consideremos el siguiente problema real:

Tenemos que escribir una función que sea capaz de ordenar virtualmente cualquier colección de datos que pueda ser contenida en un arreglo. Sea este un arreglo de cadenas, de enteros, flotantes, e incluso estructuras. El algoritmo de ordenación puede ser el mismo para todos.

Por ejemplo, puede ser un simple algoritmo de ordenación por el método de la burbuja, o el mucho más complejo algoritmo de ordenación quick sort o por shell sort.

Usaremos un simple algoritmo de burbuja para nuestros fines didácticos.

Ejemplo 7:

```c
#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);

int compare(int *m, int *n);

int main(void)

{
```

```
 int i;
 putchar('\n');
 for (i = 0; i < 10; i++)
 {
 printf("%d ", arr[i]);
 }
 bubble(arr,10);
 putchar('\n');
 for (i = 0; i < 10; i++)
 {
 printf("%d ", arr[i]);
 }
 return 0;
}
void bubble(int *p, int N)
{
 int i, j, t;
 for (i = N-1; i >= 0; i--)
 {
 for (j = 1; j <= i; j++)
 {
 if (compare(&p[j-1], &p[j]))
 {
 t = p[j-1];
 p[j-1] = p[j];
 p[j] = t;
 }
 }
 }
}
int compare(int *m, int *n)
{
```

```
 return (*m > *n);
}
```

Ejemplo 8:

```
#include <stdio.h>
int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(void *m, void *n);
int main(void)
{
 int i;
 putchar('\n');
 for (i = 0; i < 10; i++)
 {
 printf("%d ", arr[i]);
 }
 bubble(arr,10);
 putchar('\n');
 for (i = 0; i < 10; i++)
 {
 printf("%d ", arr[i]);
 }
 return 0;
}
void bubble(int *p, int N)
{
 int i, j, t;
 for (i = N-1; i >= 0; i--)
 {
 for (j = 1; j <= i; j++)
 {
```

```
 if (compare((void *)&p[j-1], (void *)&p[j]))
 {
 t = p[j-1];
 p[j-1] = p[j];
 p[j] = t;
 }
 }
 }
}
int compare(void *m, void *n)
{
 int *m1, *n1;
 m1 = (int *)m;
 n1 = (int *)n;
 return (*m1 > *n1);
}
```

Punteros - Guía Práctica

1- ¿Qué cambios debes hacer en las definiciones de la parte izquierda para que sean equivalentes a las descripciones de la parte derecha?

struct informacion_celda c; struct informacion_celda **c_ptr;	// variable de tipo estructura informacion_celda // puntero a estructura informacion_celda;

2- ¿Se pueden hacer las siguientes asignaciones? ¿Qué declara exactamente la línea 3?

```
1  struct informacion_celda c;
2  struct informacion_celda *c_ptr = &c;
3  struct informacion_celda d;
4  struct informacion_celda *d_ptr = c_ptr;
```

3- Considera la siguiente declaración y asignación:

```
1  struct informacion_celda c;
2  struct informacion_celda *c_ptr;
3
4  c_ptr = *c;
```

¿Es correcta? Y si lo es, ¿Qué contiene la variable c_ptr (no se pregunta por lo que apunta, sino su contenido)?

4- Si se declara una variable como “struct informacion_celda c;”, ¿qué tipo de datos es el que devuelve la expresión “&c.ptr_operador”?

5- Dado el siguiente código:

```
1  struct pack3
2  {
3      int a;
4  };
5  struct pack2
6  {
7      int b;
```

```
8     struct pack3 *next;
9  };
10 struct pack1
11 {
12    int c;
13    struct pack2 *next;
14 };
15
16 struct pack1 data1, *data_ptr;
17 struct pack2 data2;
18 struct pack3 data3;
19
20 data1.c = 30;
21 data2.b = 20;
22 data3.a = 10;
23 dataPtr = &data1;
24 data1.next = &data2;
25 data2.next = &data3;
```

Decide si las siguientes expresiones son correctas y en caso de que lo sean escribe a que datos se acceden.

Expresión	Correcta	Valor
data1.c		
data_ptr->c		
data_ptr.c		
data1.next->b		
data_ptr->next->b		
data_ptr.next.b		
data_ptr->next.b		
(*(data_ptr->next)).b		
data1.next->next->a		
data_ptr->next->next.a		
data_ptr->next->next->a		
data_ptr->next->a		
data_ptr->next->next->b		

6- Supongamos que se escriben las siguientes declaraciones y asignaciones en un programa:

```
1  info_celda c;
2  info_celda_ptr c_ptr = &c;
3  info_operador op;
4  info_operador_ptr op_ptr = &op;
```

La estructura "c" contiene el campo "ptr_operador" precisamente para almacenar la información relativa al operador. ¿Qué expresión hay que usar en el código para guardar la información del operador "op" como parte de la estructura "c"? Teniendo en cuenta los valores que se asignan en las declaraciones, escribe cuatro versiones equivalentes de esta expresión (utiliza "c","c_ptr", "op" y "op_ptr").

7- Considera las dos versiones del siguiente programa:

Versión 1

```
#include <stdio.h>
struct package
{
   int q;
};

void set_value(struct package data, int value)
{
   data.q = value;
}

int main()
{
   struct package p;
   p.q = 10;
   set_value(p, 20);
   printf("Value = %d\n", p.q);
   return 0;
}
```

Versión 2

```
#include <stdio.h>
struct package
{
   int q;
};

void set_value(struct package *d_ptr, int value)
{
   d_ptr->q = value;
}

int main()
{
   struct package p;
   p.q = 10;
   set_value(&p, 20);
   printf("Value = %d\n", p.q);
   return 0;
}
```

Explique diferencias y similitudes en ambo códigos y coméntelos.

8- Diseñe un programa que muestre el uso de operadores básicos en la declaración de punteros empleando el direccionamiento y el operador indirección.
9- Diseñe un programa, que sume dos variables de tipo entero, por medio de punteros.
10- Programa que lee un arreglo y una matriz usando aritmética de punteros *
11- Explique el siguiente código fuente:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int lista[5] = {10,20,30,40,50};
int *p;
p = lista;
printf("lista[0] equivale a: %d \n",lista[0]);
printf("*lista equivale a: %d \n",*lista);
printf("*p equivale a: %d \n\n\n",*p);
printf("lista[1] equivale a: %d \n",lista[1]);
printf("*(lista+1) equivale a: %d \n",*(lista+1));
printf("*(p+1) equivale a: %d \n\n\n",*(p+1));
printf("lista[2] equivale a: %d \n",lista[2]);
printf("*(lista+2) equivale a: %d \n",*(lista+2));
printf("*(p+2) equivale a: %d \n\n\n",*(p+2));
printf("lista[3] equivale a: %d \n",lista[3]);
printf("*(lista+3) equivale a: %d \n",*(lista+3));
printf("*p equivale a: %d \n\n\n",*(p+3));
printf("lista[4] equivale a: %d \n",lista[4]);
printf("*(lista+4) equivale a: %d \n",*(lista+4));
printf("*(p+4) equivale a: %d \n\n\n",*(p+4));
system("pause");
}
```

12- A partir de una lista de calificaciones de los alumnos, acceder y mostrar la información correspondiente a dicha lista mediante el acceso de un puntero.
13- Implemente la multiplicación de matrices utilizando punteros.
14- Utilizando punteros y funciones con punteros.Sea A una matriz de tamaño nxn, implemente un programa que dado un menú de opciones resuelva:
    • La transpuesta de A (At).
    • Si A es simétrica o antisimétrica.
    • Si A es una matriz triangular superior o triangular inferior