



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:

Edgar Tista García

Asignatura:

Estructuras de Datos y Algoritmos I

Grupo:

1

No. de Práctica(s):

2

Integrante(s):

Ugalde Velasco Armando

*No. de Equipo de
cómputo empleado:*

56

No. de Lista:

Semestre:

2020-2

Fecha de entrega:

17 de febrero de 2020

Observaciones:

CALIFICACIÓN: _____

OBJETIVO

Utilizar apuntadores en lenguaje C para acceder a las localidades de memoria tanto de datos primitivos como de arreglos.

EJEMPLOS DE LA GUÍA DE LABORATORIO

Apuntador a carácter

```
Car cter: x
C digo ASCII: 120
Direcci n de memoria: 6356731

Process returned 0 (0x0)   execution time : 0.161 s
Press any key to continue.
```

En este programa, primero se imprime el car cter ‘**x**’, utilizando la funci n ***printf*** y el especificador de formato ‘**%c**’. Luego, se imprime su valor num rico, utilizando el especificador de formato ‘**%d**’, el cual es 120. Sabemos que en realidad los caracteres son valores num ricos: en este caso, el est ndar de representaci n de caracteres es **ASCII**, donde la letra ‘**x**’ se representa con el n mero 120.

Finalmente, se imprime la direcci n de memoria donde el car cter antes mencionado se encuentra almacenado. Esta tarea se lleva a cabo mediante la utilizaci n de un apuntador a la variable que contiene el car cter.

Apuntador a entero

```
a = 5
b = 5 /*apEnt
b = 6 /*apEnt+1
a = 0 /*apEnt = 0

Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.
```

En este programa se manipulan las variables **a** y **b**, las cuales son de tipo entero. Primero, el apuntador **apEnt** contiene la dirección de memoria de la variable **a**. Se imprime el valor de la variable **a**. Luego, el valor de la variable **a** a la que apunta **apEnt** (5), se le asigna a la variable **b** y se imprime. Después, se lleva a cabo el mismo procedimiento, pero se le suma 1 a dicha cantidad. Finalmente, se le asigna el valor de 0 a la variable **a** a la que apunta **apEnt**, es decir, a la variable **a**.

Apuntador a arreglo

```
Dirección del arreglo en la primera posición: 60fef2
Dirección del arreglo: 60fef2
Dirección del arreglo: 60fef2

Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.
```

Este programa demuestra de forma clara la manera en la que los apuntadores a arreglos funcionan. Primero, se imprime la dirección de memoria del primer elemento del arreglo. Posteriormente, se imprime la dirección del arreglo, utilizando el operador de dirección (**&**) y el identificador de éste. Finalmente, se imprime el valor de un apuntador al primer elemento del arreglo, al igual que en la primera salida.

Podemos percatarnos de que se imprimen los mismos valores en los tres casos, por lo cual podemos confirmar que en realidad un apuntador a un arreglo es equivalente a la dirección de memoria del primer elemento de éste.

Paso de variables por valor y por referencia

```
Pasar valor: 55
55
128
Pasar referencia: 55
55
128
Valor final: 128

Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.
```

Este programa ejemplifica claramente el funcionamiento del **paso por valor y por referencia**.

Primero, se imprime el primer valor de un arreglo de enteros anteriormente creado, utilizando el operador de indirección y una referencia al arreglo.

Después, se ejecuta la función ***pasarValor*** con el valor anterior como parámetro. Primero imprime el valor original y posteriormente le asigna el valor 128, imprimiendo el valor final.

Sin embargo, el valor únicamente es modificado dentro de la función mencionada, ya que claramente se está **pasando por valor** al entero 55.

Después, se imprime el valor actual del primer elemento del arreglo, con el cual se había ejecutado la función anterior. Nos podemos percatar de que, en efecto, el valor permanece igual: es 55.

Posteriormente, se ejecuta la función ***pasarReferencia***, la cual toma como argumento un valor de tipo ***int****, es decir, apuntador a entero. Se introduce como parámetro la dirección al primer elemento del arreglo, la cual coincide con el tipo de dato que especifica la función. Al **pasar por referencia**, en este caso el valor en ***main*** sí se modifica a 128.

Finalmente, se imprime el valor del primer elemento del arreglo en ***main***, el cual es 128, como se esperaba.

Aritmética de direcciones

```
*apArr = 91
*(apArr+1) = 28
*(apArr+2) = 73

Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

En este programa se imprimen los tres primeros elementos de un arreglo de 5 elementos, utilizando **aritmética de punteros**.

Primero, se crea una variable que contenga un apuntador al arreglo. Sin embargo, sabemos que ésta en realidad es la dirección de memoria del primer elemento, por lo que, al utilizar el operador de indirección, obtendremos como salida el valor del primer elemento del arreglo.

Después, se imprimen dos elementos más del arreglo. El segundo se imprime al sumarle una localidad de memoria a la dirección inicial, y utilizando el operador de indirección, como se muestra en el programa. Finalmente, se imprime el valor del tercer elemento, al realizar la misma operación, pero sumando dos localidades de memoria a la dirección del primer elemento del arreglo.

Arreglo unidimensional

```
60fede
60fee0
60fee2
60fee4
60fee6

Process returned 0 (0x0)   execution time : 0.041 s
Press any key to continue.
```

En este programa se crea un arreglo unidimensional de enteros, con 5 elementos. Después, se declara e inicializa una variable que contiene un apuntador a dicho arreglo. Finalmente, se utiliza un ciclo **for** para recorrer el arreglo.

En cada repetición, se imprime la dirección de memoria de cada elemento, iniciando desde el primer elemento y aumentando una unidad cada iteración. Sin embargo, debemos notar que las direcciones de memoria que se muestran en pantalla aumentan dos unidades por cada elemento. Lo anterior es debido a que el arreglo es de enteros **short**, los cuales ocupan 2 bytes en memoria. Al sumarle una unidad al puntero, éste se traslada al siguiente bloque de memoria, que, en este caso, es otro **short**, por lo que se recorren 2 bytes.

Arreglo bidimensional

```
60fec4 60fec8 60fecc
60fed0 60fed4 60fed8
60fedc 60fee0 60fee4
Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

En este programa se realiza el mismo procedimiento que en el anterior. Las únicas diferencias son el tamaño de los elementos en el arreglo, que en este caso es de 4 bytes (tipo **int**), y que este arreglo tiene dos dimensiones.

Sin embargo, debemos recordar que las dimensiones del arreglo no modifican la forma en la que se almacenan: podemos observar que las direcciones de memoria siguen siendo contiguas, es decir, aumentan 4 unidades por cada elemento.

Cifrado César

```
*** CIFRADO CÉSAR ***
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
Elegir una opción:
1) Cifrar
2) Descifrar.
3) Salir.
1
Ingresar la palabra a cifrar (en mayúsculas): HOLA
El texto HOLA cifrado es: KROD
```

En este programa se implementa el cifrado César. Primero, se muestra un menú con las 3 opciones que se pueden observar en pantalla: cifrar, descifrar y salir.

Al escoger la opción 1 (**cifrar**), nos es solicitada una palabra en mayúsculas, la cual será el texto por cifrar. En el programa existen dos arreglos declarados globalmente, los cuales contienen el abecedario y un abecedario cifrado, respectivamente. Al elegir dicha opción e introducir la palabra, se ejecuta la función **cifrar**, la cual toma como argumento una variable de tipo **char***, es decir, un puntero a carácter. Como sabemos, las cadenas son en realidad arreglos de caracteres, por lo que, en este caso, el argumento que se utiliza para dicha función es una referencia al arreglo de caracteres donde se encuentra almacenada nuestra palabra. Es decir, estamos “**pasando por referencia**”.

En la función se ejecuta un ciclo **for** anidado. En el ciclo interno se compara cada letra de la palabra a cifrar con el abecedario original, con el fin de identificar el carácter. Cuando se encuentra la letra, se imprime el carácter correspondiente en el abecedario cifrado. Esta acción se repite con cada carácter de la palabra, gracias al ciclo exterior.

```
*** CIFRADO CÉSAR ***
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
Elegir una opción:
1) Cifrar
2) Descifrar.
3) Salir.
2
Ingresar la palabra a descifrar (en mayúsculas): KROD
El texto KROD descifrado es: HOLA
```

Ahora bien, si desciframos el texto anteriormente encriptado, observaremos que éste es decodificado satisfactoriamente, como se muestra en la captura de pantalla. Esto es posible gracias a la ejecución de la función **descifrar**, la cual realiza exactamente el mismo procedimiento que la función **cifrar**, pero, en lugar de buscar las letras en el arreglo del abecedario original, lo hace en el abecedario cifrado, para así imprimir el carácter correspondiente en el abecedario original.

EJERCICIO 1

No fue posible compilar el programa inicial. Su objetivo principal era asignar valores a las variables **p1**, **p2** y **p3**, mediante el uso de punteros. Sin embargo, el código contenía los siguientes errores:

```
int *p1, *p2, p3, *p4;
```

El primer error se encontraba en esta línea: como se puede observar, la variable **p3** se encontraba definida como un entero, ya que no contenía el operador *****.

La solución fue agregarlo, como se puede apreciar en la siguiente línea:

```
int *p1, *p2, *p3, *p4;
```

Sabemos que el primer paso para la manipulación de apuntadores es su **declaración e inicialización**. Sin embargo, en el programa original su inicialización se llevó a cabo de forma errónea, como se logra observar en la siguiente captura de pantalla:

```
p1=&w;  
*p2=&x;  
*p3=y;  
p4= *p1;
```

La primera inicialización es correcta, ya que a la variable **p1** se le asigna una dirección de memoria, es decir, un puntero, el cual en este caso es a la variable **w**, de tipo entero.

Después, se utiliza el operador de indirección para asignarle un valor a las variables **p2** y **p3**, pero éstas aún no se encuentran inicializadas, por lo que dicha operación es inválida. Para resolver este error fue necesario remover el operador ***** en ambas variables.

Finalmente, a la variable **p4** se le asignó el valor (***p1**), el cual es un entero, por lo que no era una operación correcta. De igual forma, se removió el operador ***** de la variable **p1** para resolver este problema. Podemos observar este fragmento de código corregido en la siguiente captura de pantalla:


```

p1=&w;
// Eliminar el operador * de los apuntadores
// p2 y p3 para inicializarlos
p2=&x;
p3=&y;
// Eliminar el operador * de p1
p4=p1;

```

Finalmente, se asignaron valores a las variables **r1**, **r2** y **r3**. En la primera asignación, se suma un puntero (**p1**) con un entero (***p2**), y se asigna a una variable de tipo entero (**r1**). Si bien dicha operación es válida, el valor que probablemente se intente obtener no es el correcto. La corrección fue agregar el operador de indirección a la variable **p1**, para así sumar el valor que contiene y no su dirección de memoria.

Después, a **r2** se le asignó el valor que resultaba de realizar una multiplicación entre punteros: (**p3 * p4**). Sin embargo, ésta es una operación inválida, por lo que fue necesario añadir el operador de indirección a ambos apuntadores para obtener el valor correcto.

Finalmente, se utilizó el operador de indirección en la variable **r3**, lo cual es inválido, ya que **r3** no es un apuntador. Para resolver este problema fue necesario remover este operador.

```

r1= p1 + *p2;
r2= p3 * p4;
*r3= *p2 + *p3;

```

El fragmento de código con las correcciones mencionadas es el siguiente:

```

r1= (*p1) + (*p2);
r2= (*p3) * (*p4);
r3= (*p2) + (*p3);
printf("Los Resultados son: %d, %d y %d",r1,r2,r3);

```

Finalmente, se imprimieron los valores correctos correspondientes a cada variable:

```

Los Resultados son: 30, 300 y 50
Process returned 32 (0x20)   execution time : 0.038 s
Press any key to continue.

```

EJERCICIO 2

El programa compiló y se ejecutó correctamente.

a) Explica qué hace el programa.

En el programa se declara un arreglo de dimensiones 4, 5 y 2 de forma global. Además, se declara e inicializa un puntero al primer elemento de dicho arreglo, como se puede observar en las siguientes líneas:

```
int arr1[4][5][2],  
  
point = arr1;
```

Después, se ejecutan dos ciclos **for** anidados, para asignarle a cada elemento del arreglo un valor. El ciclo externo recorre los planos, el ciclo medio recorre las filas y el interior recorre las columnas del arreglo. Como se puede observar, los contadores coinciden con los índices respectivos para acceder a los elementos del arreglo, según sus dimensiones.

```
// Planos  
for (i=0; i<4; i++)  
{  
    // Filas  
    for(j=0; j<5; j++)  
    {  
        // Columnas  
        for(k=0; k<2; k++)  
        {  
            // Asignar a cada elemento la variable var, de valor inicial 1  
            arr1[i][j][k]=var;  
            // Aumentar dos unidades el valor de la variable var en cada iteración.  
            var+=2;  
        }  
    }  
}
```

En cada iteración, a cada elemento del arreglo se le asigna el valor de la variable **var**, la cual aumenta 2 unidades en cada repetición. El valor inicial de esta variable es 1, ya que fue declarada e inicializada anteriormente de esta forma.

Finalmente, se declaran e inicializan tres variables de tipo entero: a, b y c, a las cuales se les asignan los siguientes valores:

```

// arr[0][2][0]
int a= *(point+4);
printf("%d = %d\n", arr1[0][2][0], a);

// arr[1][3][0]
int b= *(point+16);
printf("%d = %d\n", arr1[1][3][0], b);

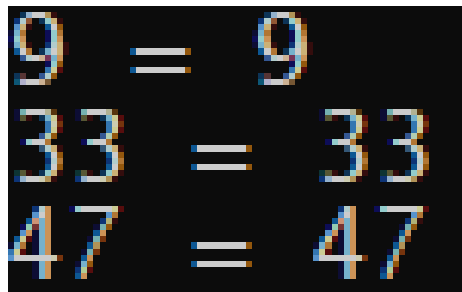
// arr[2][1][1]
int c= *(point+23);
printf("%d = %d\n", arr1[2][1][1], c);

```

b) Indica cuáles son los índices del arreglo que se almacenan en las variables a, b y c. Comprueba tu razonamiento con alguna captura de pantalla.

Podemos observar que los valores asignados son obtenidos mediante el operador de indirección y un apuntador. Se utiliza la aritmética de punteros para acceder a determinado elemento del arreglo. Sabemos que los arreglos se almacenan de forma contigua, por lo tanto, es posible recorrer sus elementos aumentando el valor de un puntero a su primera posición, que, en este caso, es la variable *point*. Partiendo de esta aseveración, es posible deducir los elementos correspondientes en el arreglo, los cuales se muestran en los comentarios del código.

Finalmente, se imprimen ambos números en cada caso, para corroborar su igualdad: el elemento correspondiente en el arreglo y el valor de la variable, determinado con el puntero. Los valores son iguales, como se esperaba:



```

9 = 9
33 = 33
47 = 47

```

b) Utilizando el apuntador y un solo ciclo, recorre el arreglo de tal manera que se modifiquen todos sus valores de la siguiente manera: plano 0, valores múltiplos de 5; plano 1, valores múltiplos de 6; plano 2, valores múltiplos de 7 y plano 3, valores múltiplos de 3.

Para lograr dicho objetivo, se utilizaron dos variables auxiliares de tipo entero: un acumulador (**acum**) y la variable **aumento**, que indicaba cuál era el múltiplo en cada caso.

Primero, ambas variables se declararon e inicializaron al valor 5, ya que el primer plano contendría valores múltiplos de 5, iniciando con este número. Para lograr asignar los valores correspondientes a todos los elementos del arreglo, fue necesario implementar un ciclo **for** que llevara a cabo 40 iteraciones, ya que éste es el número de elementos que contiene el arreglo.

Las operaciones principales que se llevarían a cabo en cada iteración son las siguientes:

```
// Modificar cada elemento, con su dirección de memoria.  
*(point) = acum;  
// Aumentar la variable acumulador con su respectivo incremento.  
acum += aumento;
```

La primera operación consiste en la asignación del valor correspondiente a cada elemento del arreglo. La variable **acum** contiene dicho valor.

Después, se le adiciona el aumento correspondiente a la variable **acum**, el cual depende del plano en el que se encuentre el ciclo en ese momento. Es decir, para el plano 0 el aumento es 5, para generar los múltiplos correspondientes, y así sucesivamente. Para lograr cambiar el valor del aumento y “reiniciar” la variable acumuladora para cada plano, se implementaron las siguientes condicionales:

```
.....  
if (i == 9)  
{  
    aumento = 6;  
    acum = 6;  
}  
else if (i == 19)  
{  
    aumento = 7;  
    acum = 7;  
}  
else if (i == 29)  
{  
    aumento = 3;  
    acum = 3;  
}  
printf("elemento %d: %d\n", i+1, *(point));
```

Cada plano tiene 10 elementos, por lo tanto, cada que se asignaban 10 valores se completaba uno de ellos. La décima iteración se realizaba cuando el valor del contador **i** era 9, lo cual indicaba que se habían asignado los 10 valores correspondientes al plano 0, por lo que se procedía a asignar los elementos del plano 1. Se asignó el aumento correspondiente (**6**), y se reinició la variable **acum** a su valor inicial correspondiente, igual al valor del aumento.

Esta condición se impuso para los planos restantes: el plano 2, cuyos elementos se asignarían en los índices 20 a 29, y el plano 3, cuyos elementos se asignarían en los índices 30 a 39, finalmente, resultando en los “**puntos de ruptura**” 19 y 29, correspondientemente.

Se agregó la función **printf** para verificar que los datos de salida fueran los esperados.

Además, al final del ciclo se realizaría un incremento del apuntador, para “recorrerlo” a la siguiente dirección de memoria, es decir, el siguiente elemento del arreglo:

```
// Incrementar el puntero para acceder a la siguiente localidad de memoria,  
// es decir, al siguiente elemento del arreglo.  
point++;
```

A continuación, podemos observar la salida del programa, la cual cumple con las condiciones establecidas en el problema:

```
elemento 1: 5  
elemento 2: 10  
elemento 3: 15  
elemento 4: 20  
elemento 5: 25  
elemento 6: 30  
elemento 7: 35  
elemento 8: 40  
elemento 9: 45  
elemento 10: 50  
elemento 11: 6  
elemento 12: 12  
elemento 13: 18  
elemento 14: 24  
elemento 15: 30  
elemento 16: 36  
elemento 17: 42  
elemento 18: 48  
elemento 19: 54  
elemento 20: 60  
elemento 21: 7  
elemento 22: 14  
elemento 23: 21  
elemento 24: 28  
elemento 25: 35  
elemento 26: 42  
elemento 27: 49  
elemento 28: 56  
elemento 29: 63  
elemento 30: 70  
elemento 31: 3  
elemento 32: 6  
elemento 33: 9  
elemento 34: 12  
elemento 35: 15  
elemento 36: 18  
elemento 37: 21  
elemento 38: 24  
elemento 39: 27  
elemento 40: 30
```

EJERCICIO 3

Se cumplió con los requisitos solicitados en el programa. Primero, se solicitan al usuario dos números enteros, almacenándolos en las variables **a** y **b**, como se muestra en la siguiente captura de pantalla:

```
int a, b;
printf("Por favor, introduzca dos enteros: ");
scanf("%d%d", &a, &b);
```

Después, se ejecuta la función *calcular*, la cual tiene como argumentos dos punteros a enteros: **&a** y **&b**. Es decir, estamos **pasando por referencia** estas variables, para así modificar su valor en la función *main*.

```
void calcular(int *x, int *y)
{
    *y = pow(*x, *y);
    *x = *y / *x;
}
```

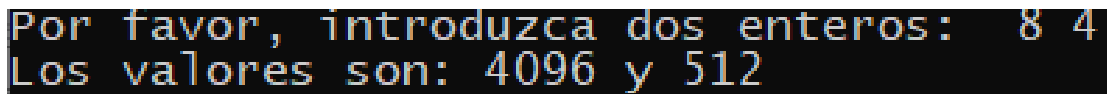
La tarea de esta función es modificar los valores de **a** y **b**, como se especifica en cada inciso. Primero, se debe obtener el valor del resultado de elevar **a** a la potencia **b**. Dicha acción se realiza en la primera línea, utilizando la función *pow* incluida en la librería *math.h*. El valor final se almacena en la variable **b**, ya que no es posible utilizar variables auxiliares, y, si se almacenara en **a**, el siguiente paso no se podría realizar.

El siguiente paso es obtener el resultado de dividir el valor anterior sobre el primer valor: **a**. El valor final es almacenado en **a**. Esta tarea es realizada en la segunda línea de la función.

```
calcular(&a, &b);
printf("Los valores son: %d y %d\n", b, a);
```

Finalmente, se lleva a cabo la ejecución de la función y se imprimen los valores correspondientes. Se invirtió el orden de impresión de las variables **a** y **b**, ya que el resultado solicitado en el inciso **a** se almacenó en la variable **b**, y viceversa.

A continuación, podemos observar una captura de pantalla de la salida del programa, con los números ejemplificados en el planteamiento del problema. Los números resultantes son los esperados.



```
Por favor, introduzca dos enteros: 8 4
Los valores son: 4096 y 512
```

EJERCICIO 4

El programa cumplió con los requerimientos establecidos. Primero, se declaró el arreglo y las variables contadoras, y se solicitaron los 20 elementos del arreglo por medio de dos ciclos anidados y la utilización de la función **scanf**.

```
int arr[5][4], i, j;

for(i = 0; i < 5; i++)
{
    for(j = 0; j < 4; j++)
    {
        printf("Digite el elemento [%d][%d]: ", i, j);
        scanf("%d", &arr[i][j]);
    }
}
```

Después, se ejecutó la función **modificarArreglo**, la cual se encargaba de realizar las modificaciones pertinentes a éste.

```
void modificarArreglo(int arr[5][4])
{
    int i, j;

    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 4; j++)
        {
            if (i % 2 == 0)
            {
                arr[i][j] *= 2;
            }
            else
            {
                arr[i][j] *= 3;
            }
        }
    }
}
```

El argumento utilizado fue el arreglo creado. Sabemos que en realidad este valor es una referencia a su primer elemento, por lo que lo estamos **pasando por referencia**, para así mutar y modificar el arreglo en esta función exterior. La función utilizó dos ciclos anidados para recorrer cada elemento del arreglo. En cada iteración se comprobaba la condición establecida: si el número del contador **i** era par, el elemento aumentaba el **doble** de su valor, de lo contrario, aumentaba el **triple**. Lo anterior se implementó mediante el uso del operador módulo (**%**), como se puede observar en el código: si (**i mod 2**) era igual a 0, implicaba que **i** fuera un múltiplo de dos, por lo tanto, era par.

Finalmente, se imprimió cada elemento del arreglo para corroborar el funcionamiento del programa. En la siguiente captura de pantalla podemos observar la salida del programa para un conjunto de datos aleatorio proporcionado por el usuario:

```
Digite el elemento [0][0]: 8
Digite el elemento [0][1]: 5
Digite el elemento [0][2]: 4
Digite el elemento [0][3]: 3
Digite el elemento [1][0]: 7
Digite el elemento [1][1]: 5
Digite el elemento [1][2]: 4
Digite el elemento [1][3]: 3
Digite el elemento [2][0]: 21
Digite el elemento [2][1]: 7
Digite el elemento [2][2]: 6
Digite el elemento [2][3]: 4
Digite el elemento [3][0]: 5
Digite el elemento [3][1]: 8
Digite el elemento [3][2]: 9
Digite el elemento [3][3]: 4
Digite el elemento [4][0]: 2
Digite el elemento [4][1]: 1
Digite el elemento [4][2]: 3
Digite el elemento [4][3]: 5
Elemento [0][0]: 16
Elemento [0][1]: 10
Elemento [0][2]: 8
Elemento [0][3]: 6
Elemento [1][0]: 21
Elemento [1][1]: 15
Elemento [1][2]: 12
Elemento [1][3]: 9
Elemento [2][0]: 42
Elemento [2][1]: 14
Elemento [2][2]: 12
Elemento [2][3]: 8
Elemento [3][0]: 15
Elemento [3][1]: 24
Elemento [3][2]: 27
Elemento [3][3]: 12
Elemento [4][0]: 4
Elemento [4][1]: 2
Elemento [4][2]: 6
Elemento [4][3]: 10
```

La salida del programa fue la esperada: los valores en las filas con índice par fueron aumentados al doble, y los valores en las demás filas fueron aumentados al triple.

CONCLUSIONES

Podemos concluir que se cumplió el objetivo planteado inicialmente, ya que se completaron satisfactoriamente los ejercicios propuestos en la práctica: se utilizaron apuntadores para acceder a los valores de datos primitivos, así como de arreglos.

En los ejercicios 1 y 3 se utilizaron apuntadores a datos primitivos, y en los ejercicios 2 y 4 se manipularon arreglos mediante la utilización de apuntadores, lo cual, como ya se mencionó, fue el objetivo principal de esta práctica. Por ende, pienso que los ejercicios fueron adecuados para dicho fin. Además, se puso en práctica la capacidad de analizar y corregir el código ya existente, lo cual, en mi opinión, es una habilidad fundamental en la programación.