



TECNOLOGICO  
NACIONAL DE MEXICO

# Arreglos (Matrices) y Punteros

Arquitectura de Programación para el control de Hardware





TECNOLOGICO  
NACIONAL DE MEXICO

APH

# DEFINICIÓN



# Definición

Los arreglos o matrices, son variables que pueden almacenar muchos elementos del mismo tipo. Los elementos individuales conocidos como elementos se almacenan secuencialmente y se identifican de forma única por el índice del arreglo.

Características de los arreglos:

- Puede contener cualquier número de elementos
- Los elementos deben ser del mismo tipo
- El índice está basado en cero
- El tamaño del arreglo (número de elementos) debe especificarse en la declaración



# Declaración de un arreglo

- Los arreglos se declaran como las variables.

## Syntax

```
type arrayName[size];
```

- size* se refiere al número de elementos
- size* debe ser una constante entera

## Example

```
int a[10];      // Arreglo con 10 elementos enteros
char s[25];     // Arreglo con 25 caracteres
```



# Inicialización de un arreglo

- Los arreglos pueden ser inicializados con una lista en la declaración :

## Sintaxis

```
type arrayName[size] = {item1, ..., itemn};
```

- Todos los elementos del arreglo deben de ser del mismo tipo:

## Ejemplo

```
int a[5] = {10, 20, 30, 40, 50};  
  
char b[5] = {'a', 'b', 'c', 'd', 'e'};
```



# Acceso a los elementos de un arreglo

- Se puede acceder a los elementos de una arreglo a través de un índice.

## Syntax

```
arrayName[index]
```

- *index* puede ser una variable o una constante
- El primer elemento en un arreglo tiene un indice 0
- C no proporciona ninguna comprobación de los límites

## Example

```
int i, a[10]; //An array that can hold 10 integers

for(i = 0; i < 10; i++) {
    a[i] = 0; //Initialize all array elements to 0
}
a[4] = 42; //Set fifth element to 42
```



# Arreglos multidimensionales

- Se pueden agregar múltiples dimensiones en la declaración de un arreglo.

## Sintaxis

```
type arrayName[size1] ... [sizen];
```

- Tres dimensiones tienden a ser las más usadas en práctica común

## Ejemplo

```
int a[10][10];           //10x10 array for 100 integers  
  
float b[10][10][10];   //10x10x10 array for 1000 floats
```



# Arreglos multidimensionales

## Sintaxis

```
type arrayName[size0]...[sizen] =  
    {{item}, ..., item},  
    {item, ..., item} } ;
```

## Ejemplo

```
char a[3][3] = {{'X', 'O', 'X'},  
                 {'O', 'O', 'X'},  
                 {'X', 'X', 'O'}};  
  
int b[2][2][2] = {{{0, 1}, {2, 3}}, {{4, 5}, {6, 7}}};
```

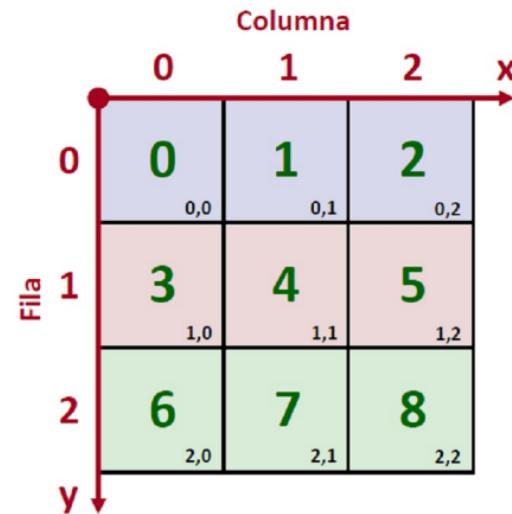


# Visualizando arreglos de 2 dimensiones

```
int a[3][3] = {{0, 1, 2},  
                {3, 4, 5},  
                {6, 7, 8}};
```

Row, Column

a[y][x]		
Row 0	a[0][0] = 0;	
Row 1	a[0][1] = 1;	
Row 2	a[0][2] = 2;	
	a[1][0] = 3;	
	a[1][1] = 4;	
	a[1][2] = 5;	
	a[2][0] = 6;	
	a[2][1] = 7;	
	a[2][2] = 8;	



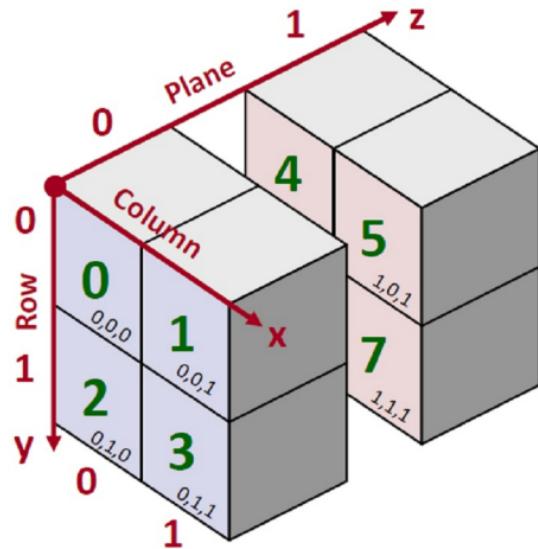
# Visualizando arreglos de 3 dimensiones

```
int a[2][2][2] = {{{0, 1}, {2, 3}},  
                   {{4, 5}, {6, 7}}};
```

Plane, Row, Column

a[z][y][x]

	Plane 0	Plane 1
	a[0][0][0] = 0;	a[1][0][0] = 4;
Row 0	a[0][0][1] = 1;	a[1][0][1] = 5;
Row 1	a[0][1][0] = 2;	a[1][1][0] = 6;
	a[0][1][1] = 3;	a[1][1][1] = 7;



# Ejemplo

```
*****
* Print out 0 to 90 in increments of 10
*****
int main(void)
{
    int i = 0;
    int a[10] = {0,1,2,3,4,5,6,7,8,9};

    while (i < 10)
    {
        a[i] *= 10;
        printf("%d\n", a[i]);
        i++;
    }

    while (1);
}
```



# Arreglo de caracteres (Strings)

Las cadenas de texto son arreglos de caracteres cuyo último elemento es un carácter nulo '\0' con un valor ASCII de 0. C no tiene un tipo de datos de cadena nativo, por lo que las cadenas siempre deben tratarse como arreglo de caracteres.

- Están encerrados entre comillas dobles "string"
- Terminan con un carácter nulo '\0'
- Debe manipularse como arreglo de caracteres (tratando elemento por elemento)
- Puede inicializarse con una cadena de texto literal constante



# Arreglo de caracteres (Strings)

- Las cadenas de texto se declaran como cualquier arreglo.

sintaxis

```
char arrayName[length];
```

- la longitud debe ser uno mayor que la longitud de la cadena para acomodar el carácter nulo de terminación '\0'
- A Char matriz con n elementos tiene cadenas con n-1 Char



# Arreglo de caracteres (Strings)

- **Length** debe ser uno mayor que la longitud de la cadena para acomodar el carácter nulo '\0'
- Un arreglo de caracteres con n elementos almacena cadenas de texto con n-1 caracteres
- Ejemplo:

## ejemplo

```
char str1[10];           //Holds 9 characters plus '\0'
```

```
char str2[6];            //Holds 5 characters plus '\0'
```



# Inicialización de un arreglo de caracteres (Strings)

Un arreglo de caracteres puede ser inicializado con una cadena de texto literal constante:

## Sintaxis

```
char arrayName[] = "Microchip";
```

- El tamaño del arreglo no es necesario
- Tamaño determinado automáticamente por la longitud de la cadena de texto
- El carácter NULL '\0' se agrega automáticamente



# Inicialización de un arreglo de caracteres (Strings)

- Un arreglo de caracteres puede ser inicializado con una cadena de texto literal constante:

## sintaxis

```
char str1[] = "Microchip"; //10 chars "Microchip\0"

char str2[6] = "Hello";    //6 chars "Hello\0"

//Declaración alternativa de string - requiere tamaño
char str3[4] = {'P', 'I', 'C', '\0'};
```



# Inicialización de un arreglo de caracteres (Strings)

- Inicialización de una cadena de texto en código, elemento por elemento:

## Sintaxis

```
arrayName[0] = char1;
arrayName[1] = char2;
arrayName[n] = '\0';
```



# Inicialización de un arreglo de caracteres (Strings)

- Caracter Null '\0' debe ser adicionado manualmente

## Ejemplo

```
str[0] = 'H';
str[1] = 'e';
str[2] = 'l';
str[3] = 'l';
str[4] = 'o';
str[5] = '\0';
```



# Comparando cadenas de texto

- Las cadenas no pueden compararse como variables ordinarias con operadores relacionales: no podemos hacer `if(str1 == str2)` como lo hacemos `if(x == y)`.
- Para comparar dos cadenas, es necesario compararlas carácter por carácter.
- C proporciona funciones de manipulación / comparación de cadenas como parte de su librería estándar.
- Cuando se usa `strcmp()` para comparar dos cadenas, devuelve 0 (FALSE) cuando coinciden, por lo que su lógica debe invertirse cuando se usa como una expresión condicional.



# Comparando cadenas de texto

## Ejemplo

```
char str[] = "Hello";  
  
if (!strcmp(str, "Hello"))  
    printf("The string is \"%s\".\n", str);
```





TECNOLOGICO  
NACIONAL DE MEXICO

APH

# PUNTEROS Y ARREGLOS



# Punteros y arreglos

- Los elementos de la matriz ocupan ubicaciones de memoria consecutivas. Debido a esto, si conoce la dirección del primer elemento de una matriz, puede moverse fácilmente a través de la matriz incrementando la dirección o agregando un desplazamiento a esa dirección.
- Si bien podríamos hacer esto con la sintaxis de matriz ordinaria incrementando o agregando un valor al desplazamiento de la matriz, también podríamos hacer esto con punteros.

```
int x[3] = {1,2,3};
```

16-bit Data Memory (RAM)	
	Address
FFFF	0x07FE
x[0]	0x0800
x[1]	0x0802
x[2]	0x0804
FFFF	0x0806



# Inicialización de un puntero a un arreglo

- Crear un puntero a un arreglo es como crear un puntero a cualquier otra variable. Sin embargo, debido a la naturaleza única de los arreglos, existen formas adicionales de inicializar punteros a arreglos.

```
int x[5] = {1,2,3,4,5};  
int *p;
```

- Podemos inicializar el puntero para que apunte al arreglo utilizando cualquiera de estos tres métodos:

```
p = x;          //Funciona solo con arreglos  
p = &x;          //Funciona para arreglos y variables  
p = &x[0];       //Esta es la forma más obvia
```





TECNOLOGICO  
NACIONAL DE MEXICO

APH

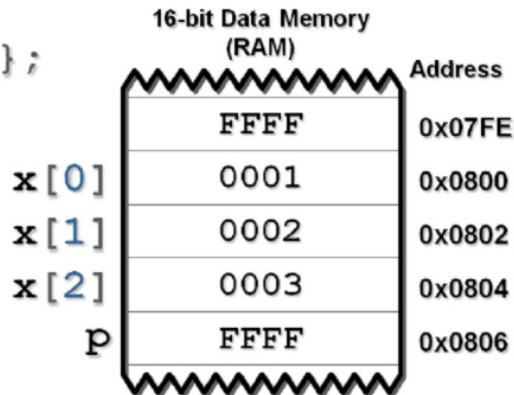
# ARITMÉTICA DE PUNTEROS



# Aritmética de punteros

- Una de las características más poderosas del uso de punteros con arreglos es que incrementar un puntero siempre lo mueve al siguiente elemento del arreglo, independientemente del número de bytes que ocupe cada elemento.

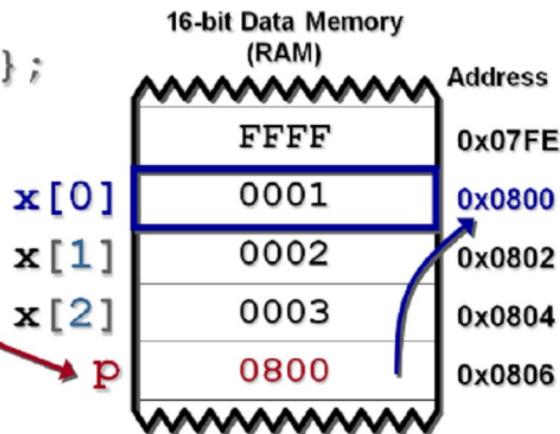
```
int x[3] = {1,2,3};  
int *p;  
  
p = &x;  
p++;
```



# Aritmética de punteros

- Primero, establecemos el puntero en la dirección del primer elemento (que es la dirección de la matriz en sí).

```
int x[3] = {1,2,3};  
int *p;  
  
p = &x;  
p++;
```



# Aritmética de punteros

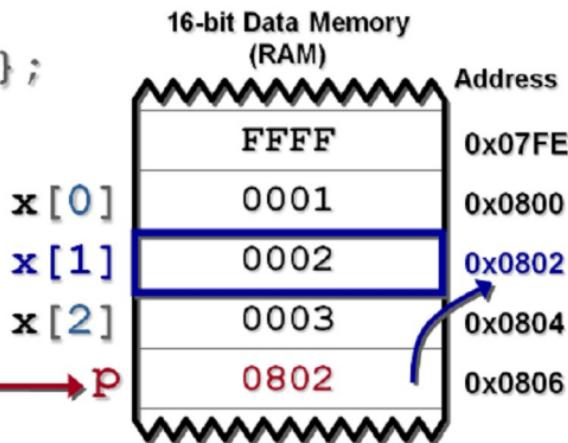
- A continuación, incrementamos el puntero para que ahora apunte al siguiente elemento de la matriz.

```
int x[3] = {1,2,3};
```

```
int *p;
```

```
p = &x;
```

```
p++;
```



# Aritmética de punteros

## Punteros incrementales

- Cuando incrementa o reduce un puntero, siempre se incrementará por el número de bytes ocupados por el tipo al que apunta.
- Por ejemplo, si tenemos un puntero float, incrementar el puntero incrementará la dirección que contiene en 4, ya que las variables flotantes ocupan 4 bytes de memoria.

```
float *ptr;
```

```
ptr = &a; →
```

```
ptr++; →
```

**Incrementing ptr moves it  
to the next sequential  
float array element**

16-bit Data Memory Words



# Aritmética de punteros

## Incrementos más grandes

- Si usamos el operador de asignación compuesta para sumar o restar un número mayor para un salto más largo, el puntero será modificado por un múltiplo del número de bytes ocupados por el tipo al que apunta.

```
float *ptr;
```

```
ptr = &a; →
```

**Adding 6 to ptr moves it 6  
float array elements  
ahead (24 bytes ahead)**

```
ptr += 6; →
```

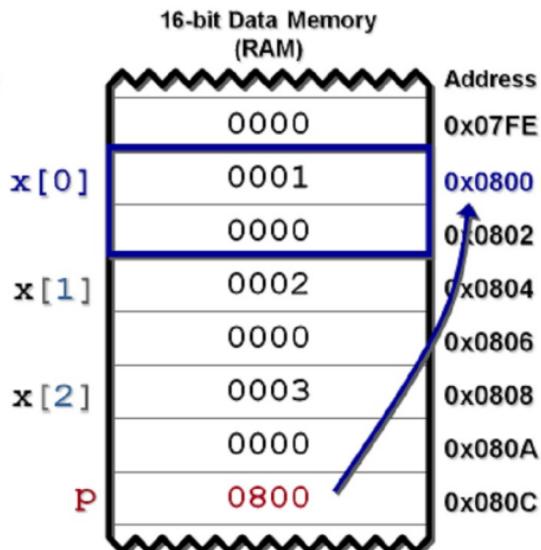
16-bit Data Memory Words



# Aritmética de punteros

- En el siguiente ejemplo inicializamos el puntero p para que apunte a la matriz x.

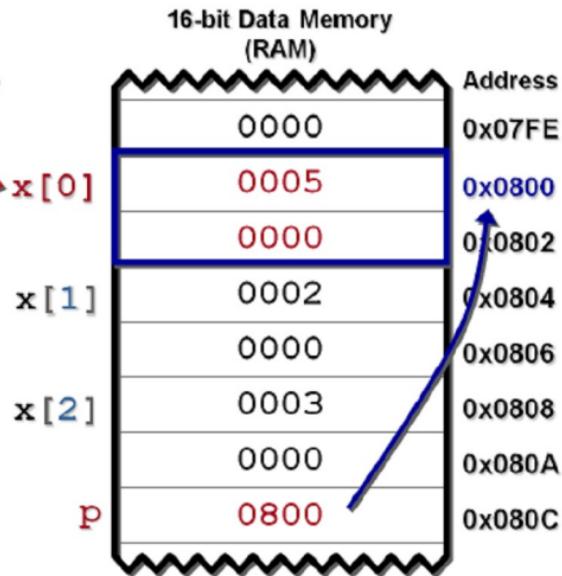
```
{  
    long x[3] = {1,2,3};  
    long *p = &x;  
  
    *p += 4;  
    p++;  
    *p = 0xDEADBEEF;  
    p++;  
    *p = 0xF1D0F00D;  
    p -= 2;  
    *p = 0xBADF00D1;  
}
```



# Aritmética de punteros

- Esta primera línea suma 4 al elemento apuntado por p, que en este caso es x[0], el cual tiene el valor de 1. Por lo que  $1 + 4 = 5$ .

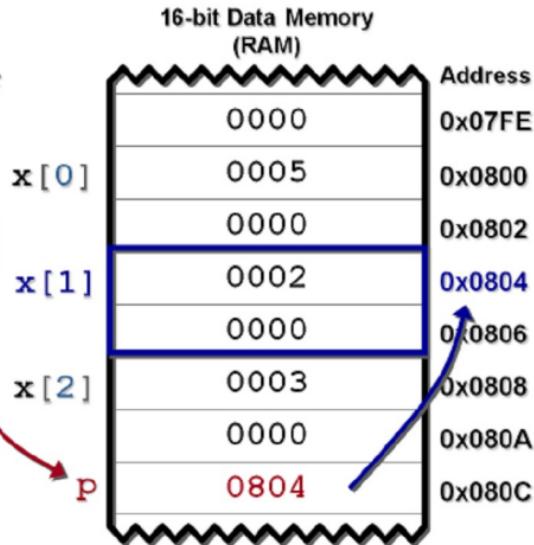
```
{  
    long x[3] = {1,2,3};  
    long *p = &x;  
  
    *p += 4;  
  
    p++;  
    *p = 0xDEADBEEF;  
    p++;  
    *p = 0xF1D0F00D;  
    p -= 2;  
    *p = 0xBADF00D1;  
}
```



# Aritmética de punteros

- Ahora, incrementamos el puntero. Como estamos apuntando a un long, el puntero se incrementará en 4 bytes.

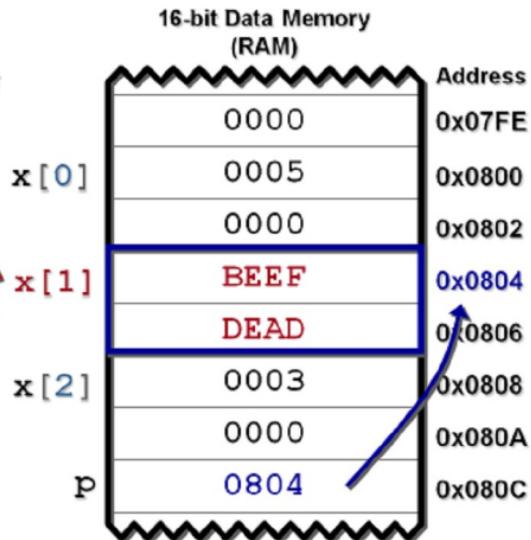
```
{  
    long x[3] = {1,2,3};  
    long *p = &x;  
  
    *p += 4;  
    p++; // Red arrow points here  
    *p = 0xDEADBEEF;  
    p++;  
    *p = 0xF1D0F00D;  
    p -= 2;  
    *p = 0xBADF00D1;  
}
```



# Aritmética de punteros

- Ahora, asignamos el valor 0xDEADBEEF a x[1] mediante el puntero que se acaba de incrementarse.

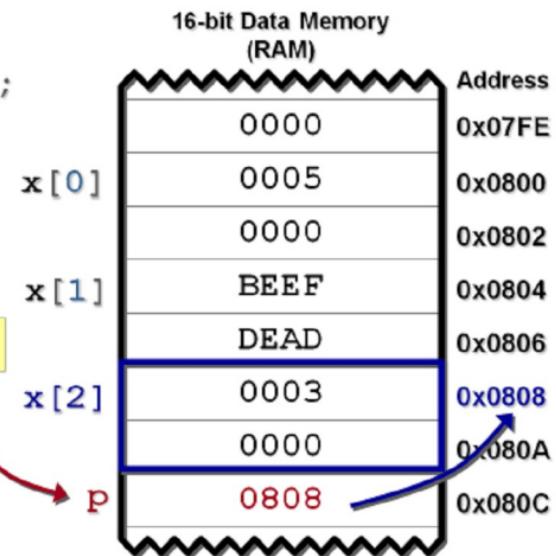
```
{  
    long x[3] = {1,2,3};  
    long *p = &x;  
  
    *p += 4;  
    p++;  
    *p = 0xDEADBEEF;  
    p++;  
    *p = 0xF1D0F00D;  
    p -= 2;  
    *p = 0xBADF00D1;  
}
```



# Aritmética de punteros

- Incrementa el puntero nuevamente (en 4 bytes).

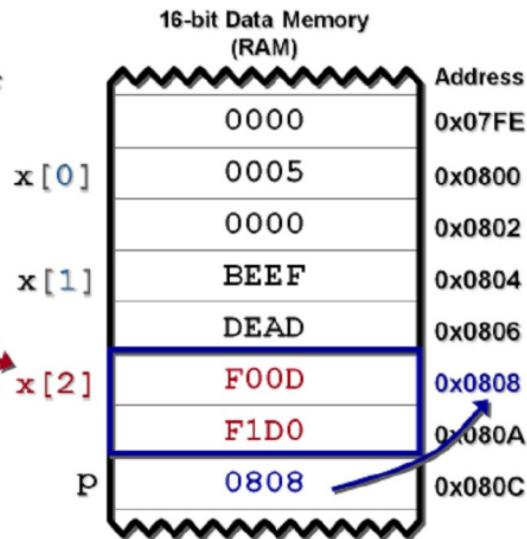
```
{  
    long x[3] = {1,2,3};  
    long *p = &x;  
  
    *p += 4;  
    p++;  
    *p = 0xDEADBEEF;  
    p++; // Red arrow points here  
    *p = 0xF1D0F00D;  
    p -= 2;  
    *p = 0xBADF00D1;  
}
```



# Aritmética de punteros

- Ahora asignamos el valor 0xF1D0F00D a x[2] a través del puntero.

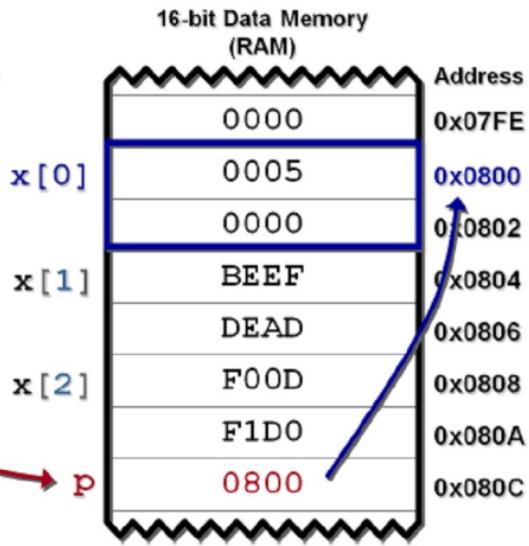
```
{  
    long x[3] = {1,2,3};  
    long *p = &x;  
  
    *p += 4;  
    p++;  
    *p = 0xDEADBEEF;  
    p++;  
    *p = 0xF1D0F00D; // Linea resaltada en amarillo  
    p -= 2;  
    *p = 0xBADF00D1;  
}
```



# Aritmética de punteros

- Ahora, queremos reducir el puntero en 2 elementos. Dado que una longitud es de 4 bytes, disminuiremos p en  $2 * 4 = 8$  bytes.

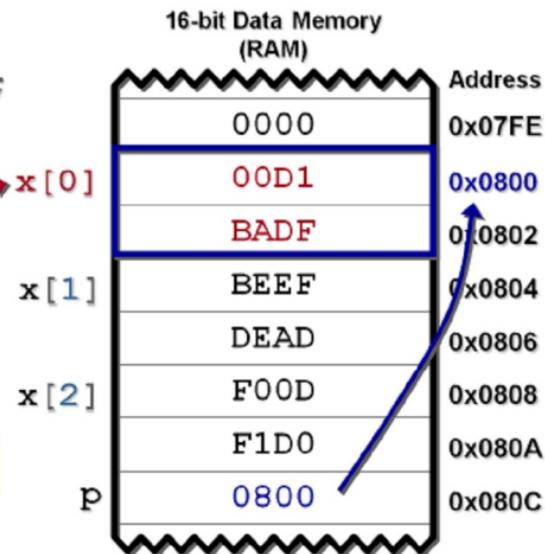
```
{  
    long x[3] = {1,2,3};  
    long *p = &x;  
  
    *p += 4;  
    p++;  
    *p = 0xDEADBEEF;  
    p++;  
    *p = 0xF1D0F00D;  
    p -= 2; // Red arrow points here  
    *p = 0xBADF00D1;  
}
```



# Aritmética de punteros

- Y finalmente, sobrescribimos el valor anterior en  $x[0]$  con el valor 0xFABF00D1 mediante el puntero .

```
{  
    long x[3] = {1, 2, 3};  
    long *p = &x;  
  
    *p += 4;  
    p++;  
    *p = 0xDEADBEEF;  
    p++;  
    *p = 0xF1D0F00D;  
    p -= 2;  
    *p = 0xBADF00D1;  
}
```



# Conclusión

- El identificador de una matriz es considerado un puntero a su primer elemento [2].
  - Esto tiene importantes implicaciones prácticas, y es el secreto para comprender la mecánica de matrices; de punteros, y de las cadenas de caracteres en C++ (un tipo especial de matrices).
- Por ejemplo, cuando una matriz es pasada como argumento a una función, el identificador es considerado como un puntero al primer elemento, así que coloquialmente suele decirse: "las matrices se pasan por referencia"; por supuesto, dentro de la función llamada, el argumento aparece como una variable local tipo puntero.



# Conclusión

- Podríamos decir que el nemónico de una matriz C++ encierra una dualidad.
  - En momentos puede ser considerado como representante de una matriz. Por ejemplo, cuando lo utilizamos con la notación de subíndices o en el operador **sizeof**.
  - En otros casos adquiere la personalidad de puntero. Por ejemplo, cuando utilizamos con él el álgebra de punteros (en cierta forma, recuerda la famosa dualidad onda-partícula de la luz que estudiamos en bachiller).
- Esta dualidad hace que a efectos prácticos, el identificador de una matriz sea sinónimo de la dirección de su primer elemento; de modo que si **m** es una matriz de elementos tipo**X** y **pm** un puntero-a-tipo**X**, la asignación **pm = &m[0];** puede también ser expresada como **pm = m;**



# Ejemplo

Para un arreglo **int A[]={2, 4, 5, 8,1};**

**M nos da la dirección base del primer elemento**

- El valor del i-esimo elemento se puede obtener si imprimiéramos:

**Print A[i]** ó también **Print \*(A+i)**

- Y la dirección del i esimo elemento:

**Print &A[i]** ó también **Print (A+i)**

```
1  ****
2 // FileName:          Arreglos1.cpp
3 // Program version: Dev-C++ 5.11
4 // Company:           TECNM - ITCH
5 // Description:       Arreglos y punteros
6 // Authors:            ALFREDO CHACON
7 // Updated:           03/2021
8 // Nota:              RELACION ENTRE PUNTEROS Y ARREGLOS
9 ****
10 #include <stdio.h>
11
12 ****
13 PROTOTIPOS DE FUNCIONES
14 ****
15 ****FUNCION MAIN****
16 int main(){
17     int A[]={2,5,6,8,1};
18     printf("%d\n",A);
19     printf("%d\n",&A[0]);
20     printf("%d\n",A[0]);
21     printf("%d\n",*A);
22     return 0;
23 }
```



# Ejemplo

```
1  ****
2 // FileName:          Arreglos2.cpp
3 // Program version: Dev-C++ 5.11
4 // Company:           TECNM - ITCH
5 // Description:       Arreglos y punteros
6 // Authors:           ALFREDO CHACON
7 // Updated:           03/2021
8 //Nota:               RELACION ENTRE PUNTEROS Y ARREGLOS
9 ****
10 #include <stdio.h>
11
12 ****
13      PROTOTIPOS DE FUNCIONES
14 ****
15 ****FUNCION MAIN*****
16 int main(){
17     int A[]={2,5,6,8,1};
18     int i;
19     for(i=0 ; i<5 ;i++){
20         printf("Direccion = %d\n",&A[i]);
21         printf("Direccion = %d\n",A+i);
22         printf("Valor = %d\n",A[i]);
23         printf("Valor = %d\n",*(A+i));
24     }
25     return 0;
26 }
27 }
```





TECNOLÓGICO  
NACIONAL DE MÉXICO

APH

# POST-INCREMENT/DECREMENT EN PUNTEROS



# Post-Increment/Decrement en punteros

- En el post-incremento, la operación de incremento tendrá lugar después de que se utilice el puntero.
- Se debe tener cuidado con la precedencia de los operadores al realizar operaciones aritméticas con punteros:

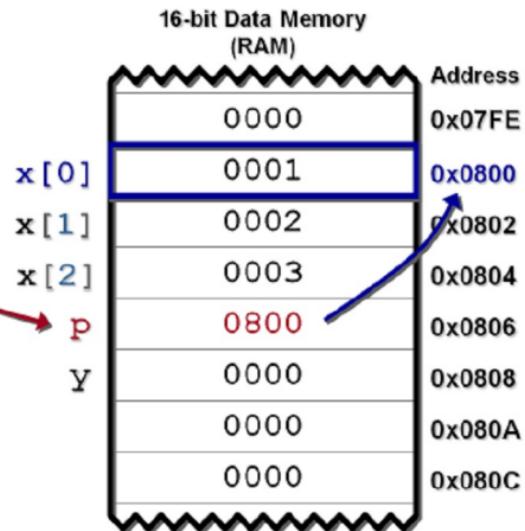
Syntax	Operation	Description by Example
<code>p++</code> <code>*p++</code> <code>* (p++)</code>	Post-Increment Pointer	<code>z = * (p++) ;</code> is equivalent to: <code>z = *p;</code> <code>p = p + 1;</code>
<code>(*p)++</code>	Post-Increment data pointed to by Pointer	<code>z = (*p) ++;</code> is equivalent to: <code>z = *p;</code> <code>*p = *p + 1;</code>



# Post-Increment/Decrement en punteros

- En el siguiente ejemplo declaramos el arreglo de enteros x inicializado con los valores 1,2 y 3. De igual forma declaramos un entero llamado y, y un puntero p inicializado al primer elemento del arreglo.

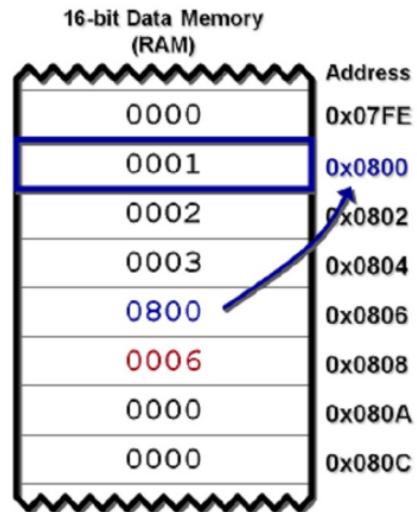
```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + *(p++);  
    y = 5 + (*p)++;  
}
```



# Post-Increment/Decrement en punteros

- En esta línea primero utilizamos el valor apuntado por p ( $x[0] = 1$ ) y a ese valor le suma un 5. El resultado de  $5+1 = 6$  se asigna a la variable y.

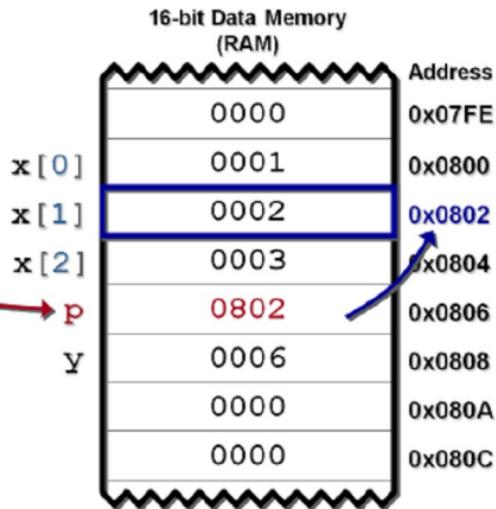
```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + *(p++);  
  
    y = 5 + (*p)++;  
}
```



# Post-Increment/Decrement en punteros

- Una vez realizada la suma, el puntero en sí se incrementa para apuntar al siguiente entero del arreglo.

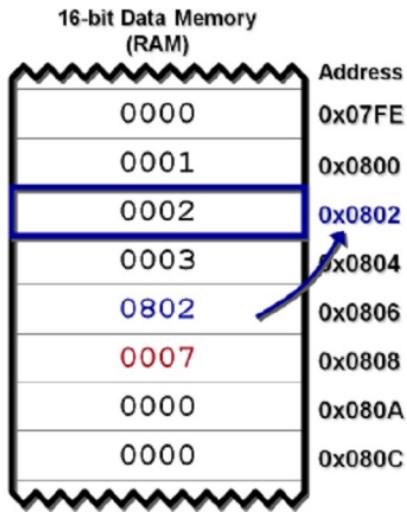
```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + *(p++);  
    y = 5 + (*p)++;  
}
```



# Post-Increment/Decrement en punteros

- En la siguiente línea, al igual que antes, primero utilizamos el valor apuntado por p, que ahora es  $x[1]=2$ , para realizar la suma  $5 + 2$ . El resultado de 7 se almacena en la variable y.

```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + * (p++);  
    y = 5 + (*p)++;  
}
```



# Post-Increment/Decrement en punteros

- Una vez realizada la suma anterior, ahora incrementamos el valor apuntado por p. Entonces el valor en x[1] se incrementa de 2 a 3, debido a que se está utilizando una operación de post-incremento del dato apuntado por el puntero (Post-Increment data pointed to by Pointer).

```
{  
    int x[3] = {1, 2, 3};  
    int y;  
    int *p = &x;  
  
    y = 5 + * (p++) ;  
    y = 5 + (*p)++;  
}
```

	Address
x[0]	0000 0x07FE
x[1]	0001 0x0800
x[2]	0003 0x0802
p	0802 0x0804
y	0007 0x0806
	0000 0x0808
	0000 0x080A
	0000 0x080C





TECNOLOGICO  
NACIONAL DE MEXICO

APH

# PRE-INCREMENT/DECREMENT EN PUNTEROS



# Pre-Increment/Decrement en punteros

- El pre-incremento funciona igual que el post-incremento, a diferencia que ahora la operación de incremento tendrá lugar antes de que se utilice el puntero. Fuera de esto se utilizan las mismas reglas.

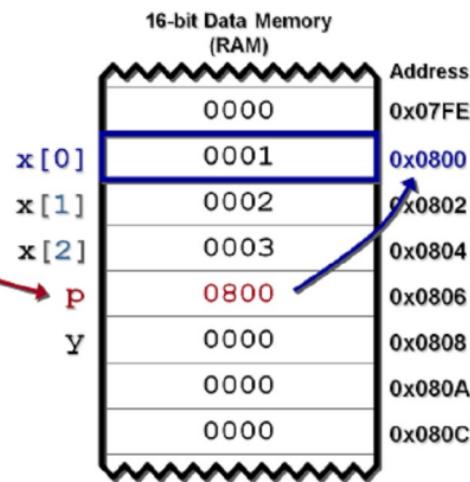
Syntax	Operation	Description by Example
$++p$ $++*p$ $* (++p)$	Pre-Increment Pointer	$z = * ( ++p ) ;$ is equivalent to: $p = p + 1;$ $z = *p;$
$++ (*p)$	Pre-Increment data pointed to by Pointer	$z = ++ (*p) ;$ is equivalent to: $*p = *p + 1;$ $z = *p;$



# Pre-Increment/Decrement en punteros

- En el siguiente ejemplo declaramos el arreglo de enteros x inicializado con los valores 1,2 y 3. De igual forma declaramos un entero llamado y, y un puntero p inicializado al primer elemento del arreglo.

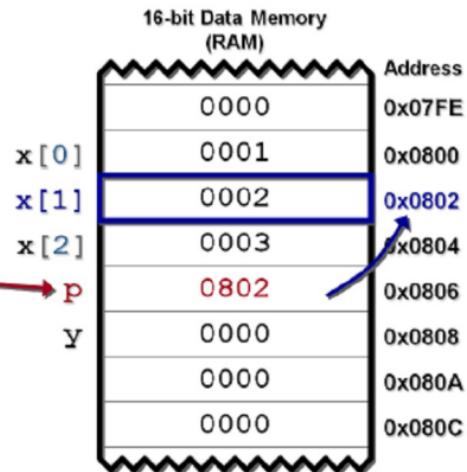
```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + *(++p);  
  
    y = 5 + ++(*p);  
}
```



# Pre-Increment/Decrement en punteros

- En la siguiente línea incrementamos primero el puntero, de modo que ahora apunte a  $x[1]$  en lugar de  $x[0]$ .

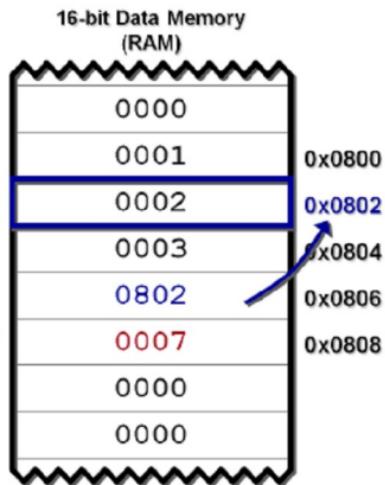
```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + *(++p);  
    y = 5 + ++(*p);  
}
```



# Pre-Increment/Decrement en punteros

- Ahora que se ha incrementado el puntero, usaremos el nuevo valor al que apunta en la operación de suma.
- Entonces a la variable y se la asigna el valor de 7, resultado de la suma 5 + 2.

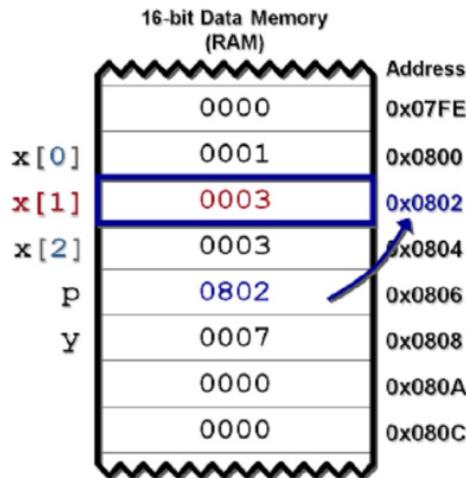
```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + * (++p);  
  
    y = 5 + ++(*p);  
}
```



# Pre-Increment/Decrement en punteros

- Ahora en la siguiente línea, primero incrementamos el dato apuntado por p. Entonces el valor que reside en x[1] se incrementa de 2 a 3.

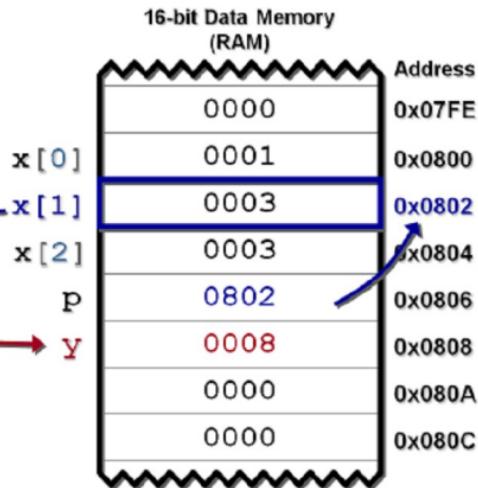
```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + * ( ++p );  
    y = 5 + ++(*p);  
}
```



# Pre-Increment/Decrement en punteros

- Después de realizar el incremento, el valor apuntado por p se utiliza para realizar la operación de suma. Entonces en la variable y se asigna el valor de 8, resultado de la suma 5 + 3.

```
{  
    int x[3] = {1,2,3};  
    int y;  
    int *p = &x;  
  
    y = 5 + * (++p);  
    y = 5 + ++(*p);  
}
```





TECNOLÓGICO  
NACIONAL DE MÉXICO

APH

# PUNTEROS Y CADENAS DE TEXTO

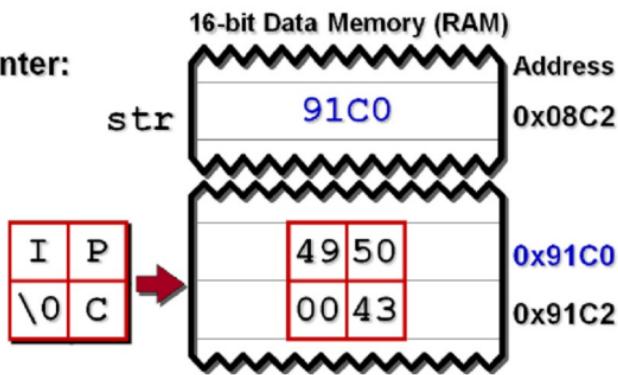


# Punteros y cadenas de texto

- Una cadena de texto puede ser declarada con un puntero al igual que se hizo con un arreglo de caracteres. La cadena puede inicializarse cuando se declara o puede asignarse después. La cadena en sí se almacenará en la memoria y al puntero se le dará la dirección del primer carácter de la cadena.

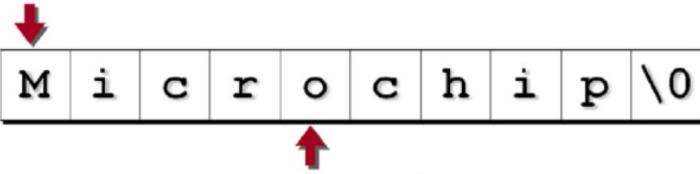
### **String declaration with a pointer:**

```
char *str = "PIC";
```



# Punteros y cadenas de texto

- Cuando se inicializa un puntero a cadena de texto apunta al primer carácter. Puede incrementar o agregar un desplazamiento al puntero para acceder a los caracteres siguientes.

```
char *str = "Microchip";  
str  


|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| M | i | c | r | o | c | h | i | p | \0 |
|---|---|---|---|---|---|---|---|---|----|

  
str += 4
```



# Punteros y cadenas de texto

- Los punteros también se pueden usar para acceder a los caracteres a través de un desplazamiento:

```
char *str = "Microchip";
*str == 'M'
    ↓
    M | i | c | r | o | c | h | i | p | \0
    ↑
* (str + 4) == 'o'
```



# Punteros y cadenas de texto

## Punteros VS arreglos (Inicialización en la declaración)

- Inicializar una cadena de caracteres cuando se declara es esencialmente lo mismo para un puntero y un arreglo. El carácter NULL '\0' se agrega automáticamente a las cadenas en ambos casos (siempre y cuando el arreglo guarde el espacio para el carácter NULL).

```
char *str = "PIC";
```

```
char str[] = "PIC";
```

OR

```
char str[4] = "PIC";
```



# Punteros y cadenas de texto

## Punteros VS arreglos (Asignación en código)

- Se puede asignar una cadena de texto completa a un puntero y se debe asignar un conjunto de caracteres carácter por carácter .
- Ejemplo con puntero:

```
char *str;  
str = "PIC";
```



# Punteros y cadenas de texto

## Punteros VS arreglos (Asignación en código)

- Recordar que en un arreglo de caracteres, se debe agregar explícitamente el carácter NULL '\0' al arreglo.
- Ejemplo con arreglo:

```
char str[4];  
  
str[0] = 'P';  
str[1] = 'I';  
str[2] = 'C';  
str[3] = '\0';
```



# Punteros y cadenas de texto

## Comparación de cadenas

- La forma correcta de comparar cadenas es usar la función de biblioteca estándar strcmp() que comparará las cadenas carácter por carácter.

```
int strcmp ( const char * s1, const char * s2);
```



# Punteros y cadenas de texto

## Comparación de cadenas

- Ejemplo de comparación del puntero a cadena str con el literal cadena “Microchip”. La función strcmp devuelve un 0 si las cadenas son iguales.

```
#include <string.h>

char *str = "Microchip";

int main(void)
{
    if (0 == strcmp(str, "Microchip"))
        printf("They match!\n");

    while(1);
}
```

