Programación de Sistemas CCPG1008

Federico Domínguez, PhD.

Unidad 3 – Sesión 1: Representación de datos y punteros

Contenidos

Repaso de notación hexadecimal

Representación de datos

Punteros

Repaso de notación hexadecimal

Se representa en C y usualmente en libros de texto, referencias, etc. con: ***0x***

Ejemplos:

0x01 es equivalente a 1 en decimal y binario

0x0E es equivalente a 14 en decimal y 1110 en binario

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Tipos de datos en C

C declaration	32-bit	64-bit
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`long long int`	8	8
`char *`	4	8
`float`	4	4
`double`	8	8

La representación de los datos en memoria pueden variar de acuerdo a la arquitectura de la máquina.

Un valor de varios bytes puede ser representado en memoria de dos formas:

Little endian: Byte menos significativo primero

Big endian: Byte más significativo primero

Por ejemplo, tenemos la variable:

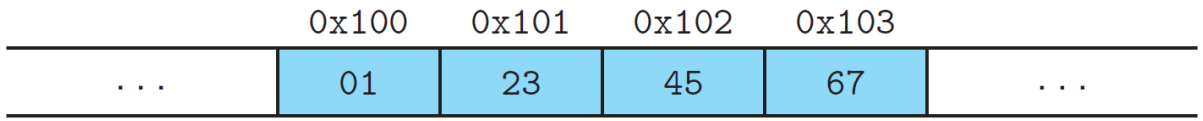
```
int x = 19088743;
```

Asumamos que esa variable está en la posición de memoria 0x100. Sabiendo que el valor de x en hexadecimal es:

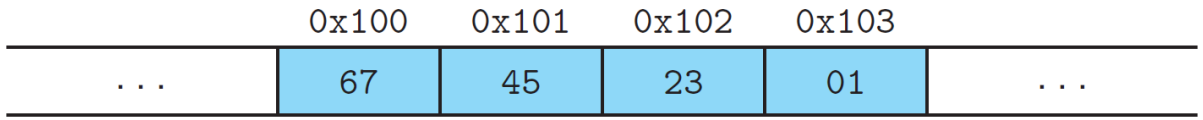
0x01234567

Entonces:

Big endian



Little endian



Codificación de tipos de datos enteros sin signo

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

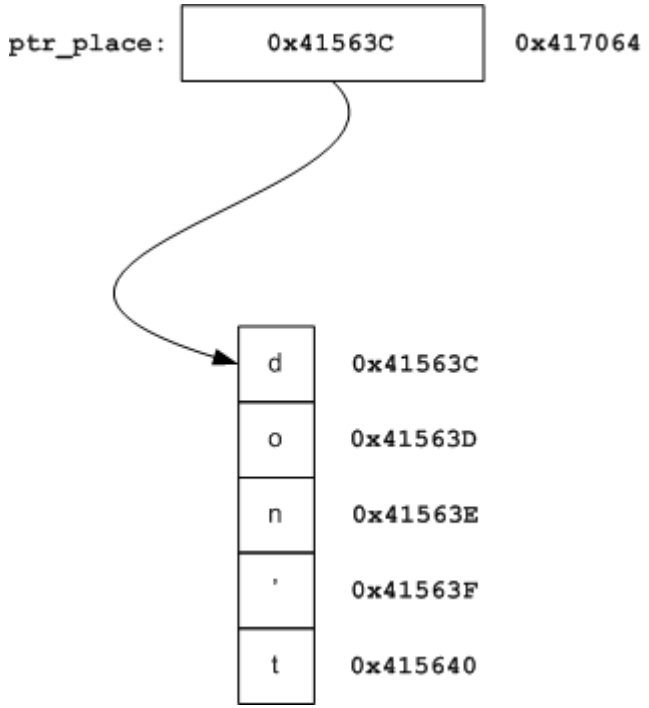
$$\begin{aligned}
B2U_4([0001]) &= 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+0+0+1 = 1\\
B2U_4([0101]) &= 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+4+0+1 = 5\\
B2U_4([1011]) &= 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 8+0+2+1 = 11\\
B2U_4([1111]) &= 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 8+4+2+1 = 15
\end{aligned}$$

Codificación de tipos de datos con signo usando **two's complement**

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

$$\begin{aligned}
B2T_4([0001]) &= -0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+0+0+1 = 1\\
B2T_4([0101]) &= -0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+4+0+1 = 5\\
B2T_4([1011]) &= -1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = -8+0+2+1 = -5\\
B2T_4([1111]) &= -1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = -8+4+2+1 = -1
\end{aligned}$$

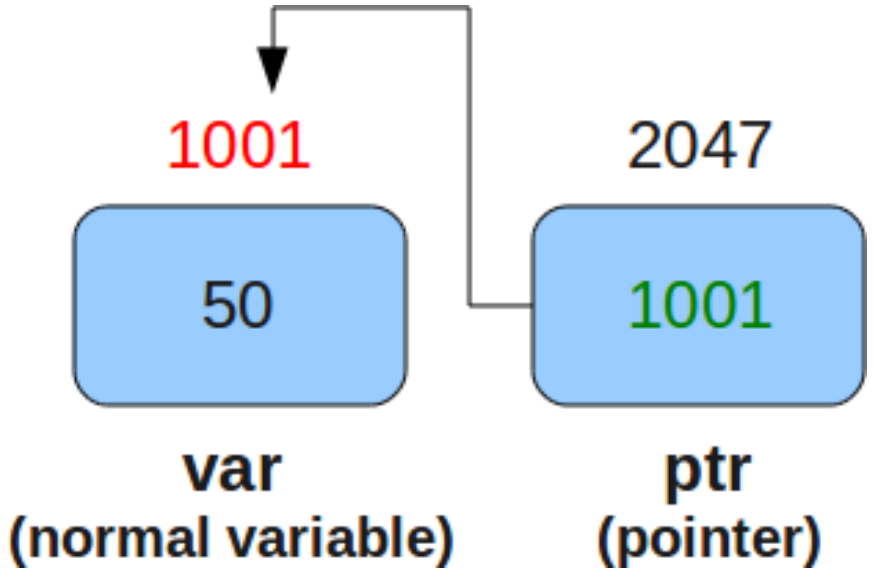
Punteros


ptr_place:
0x41563C
0x417064
d
0x41563C
o
0x41563D
n
0x41563E
'
0x41563F
t
0x415640

Un **puntero** es una variable que contiene la dirección de otra variable.

Punteros son una de las características distintivas de C y C++.

Un puntero es un tipo de referencia.



Punteros en C

Existen dos operadores unitarios para gestionar punteros:

- * indirección o dereferenciación, retorna el valor al cual apunta un puntero
- & referenciación, retorna la dirección en memoria de una variable

```
int y = 0, x = 5; /* declaración de un entero */
int *ip; /* declaración de un puntero */

ip = &x; /* ip apunta ahora a la variable x */
y = *ip; /* el valor de y es  5 */

*ip = 0; /* el valor de x es 0 */
```

Usando el operador *, punteros pueden ser tratados como cualquier otra variable.

El operador * dereferencia un puntero …

```
*ip = *ip + 100;
z = *ip * -5;
```

El operador * es como cualquier otro operador, tiene un cierto nivel de precedencia…

```
*ip = 5; /* ip apunta al valor 5 */
++*ip; /* ip apunta al valor 6 */
(*ip)++; /* ahora ip apunta al valor 7 */
*ip++; /* ahora ip apunta a otra dirección de memoria!!! */
```

Los punteros son muy útiles para enviar parámetros a una función por referencia.

Existen dos formas de enviar parámetros en una función, por referencia y por valor.

Por valor:

```
void swap(int x, int y) /* Por valor*/
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}

x = 5;
y = 7;
swap( x, y);
/* En este punto x será 5 y y 7 */
```

Los punteros son muy útiles para enviar parámetros a una función por referencia.

Existen dos formas de enviar parámetros en una función, por referencia y por valor.

Por referencia:

```
void swap(int *px, int *py) /* Por referencia */
{
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}

x = 5;
y = 7;
swap( &x, &y);
/* En este punto x será 7 y y 5 */
```

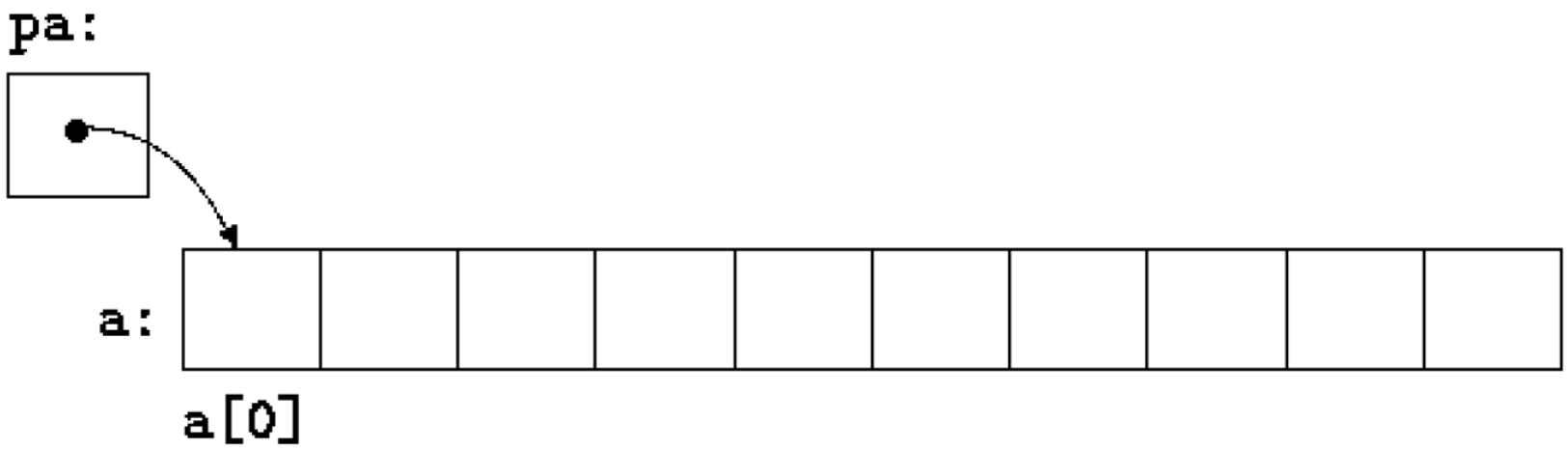
Los **arreglos** en C están fuertemente relacionados con los **punteros**.

```
int a[10];  /* Arreglo de 10 enteros */
```


```
a[0] = 52; /* Posición 0 contiene 52 */
a[1] = 20; /* Posición 1 contiene 20 */
```

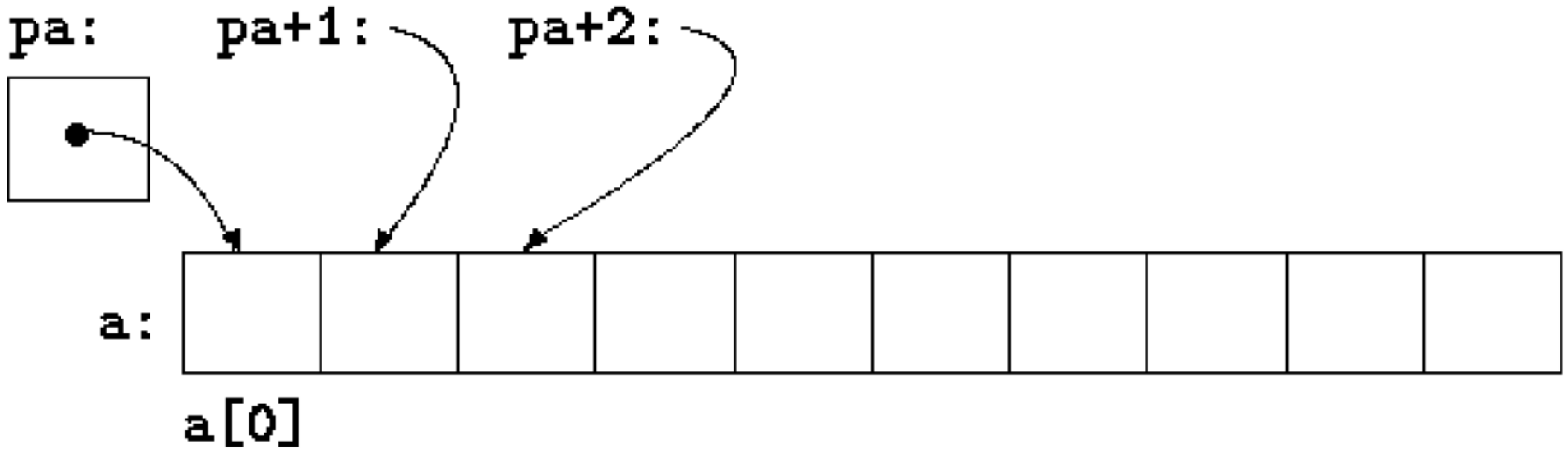
Los **arreglos** en C están fuertemente relacionados con los **punteros**.

```
int *pa;
pa = &a[0];
```

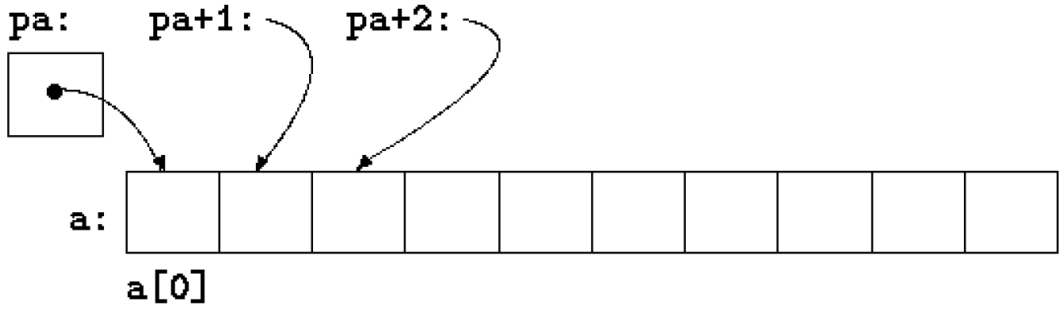


Los **arreglos** en C están fuertemente relacionados con los **punteros**.

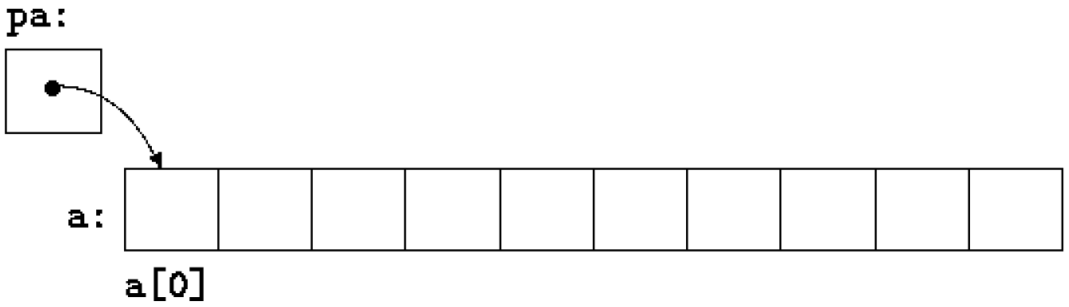
```
pa = &a[0];
x = *(pa + 1); /* Ahora x contiene el valor de a[1] */
x = *(pa + 2); /* Ahora x contiene el valor de a[2] */
```



Los **arreglos** en C están fuertemente relacionados con los **punteros**.




```
x = *(pa + i);    /* Son equivalentes */
x = a[i];         /* Son equivalente */
```




```
pa = &a[0]; /* Son equivalentes */
pa = a;     /* Son equivalente */
```

La equivalencia entre punteros y arreglos permite la manipulación de arreglos.

```
/* strlen: return length of string s */
int strlen(char *s)
{
        int n;
        for (n = 0; *s != '\0'; s++)
                n++;
        return n;
}
```

La equivalencia entre punteros y arreglos permite la manipulación de arreglos usando **aritmética de punteros**.

```
/* strlen: return length of string s */
int strlen(char *s)
{
        char *p = s;
        while (*p != '\0')
                p++;

        return p - s;
}
```

La aritmética de punteros permite ciertas operaciones entre punteros del mismo arreglo.

Siempre y cuando p y q sean punteros al mismo arreglo.

Comparaciones lógicas
- p > q

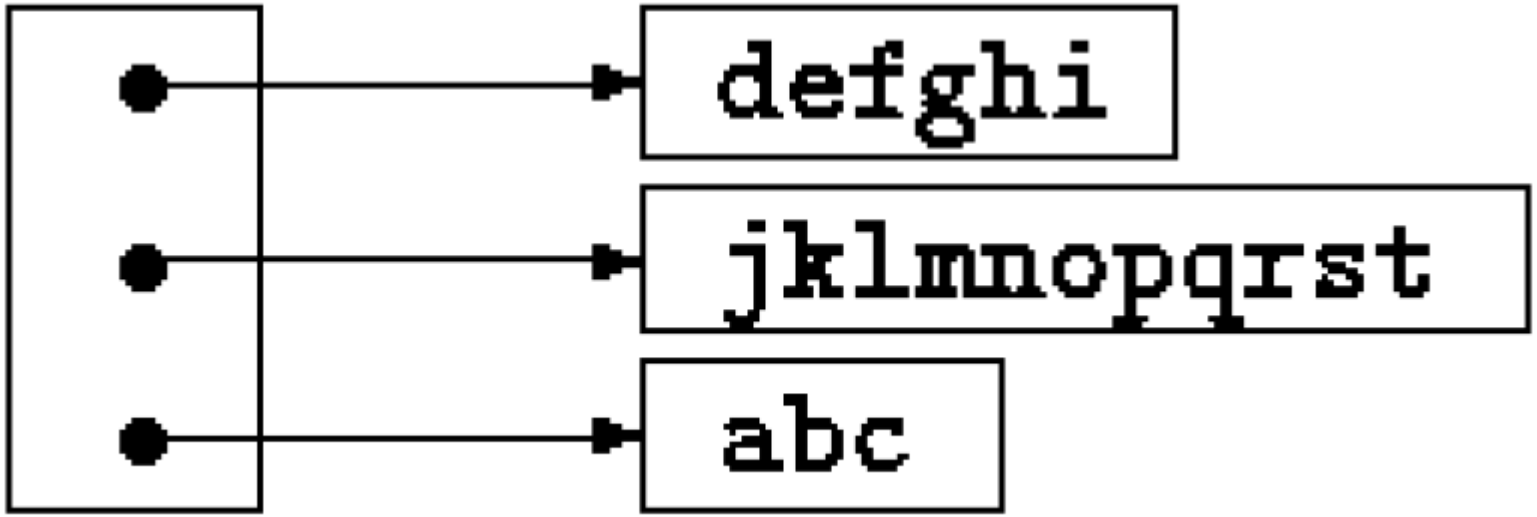
Suma y resta
- p + q
- p – q

Suma y resta con enteros
- p++
- p + 5

La aritmética de punteros es consistente con el tipo de datos del puntero.
- En este caso, el valor de p aumenta en 20, ya que *int* tiene 4 bytes:
  - *int* *p; p = p + 5;

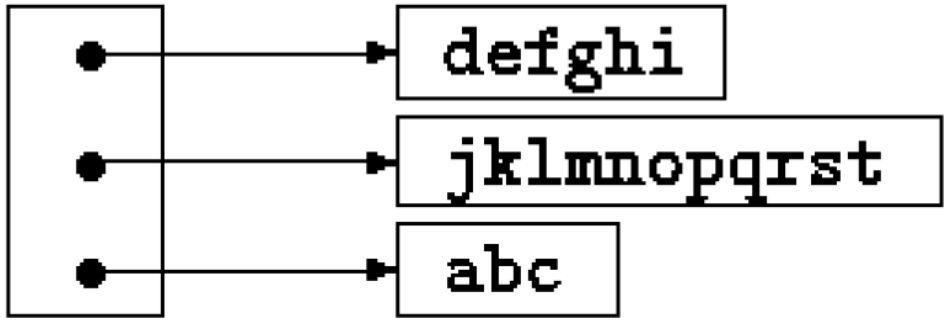
Los **arreglos de punteros** sirven para crear estructuras de datos complejas.

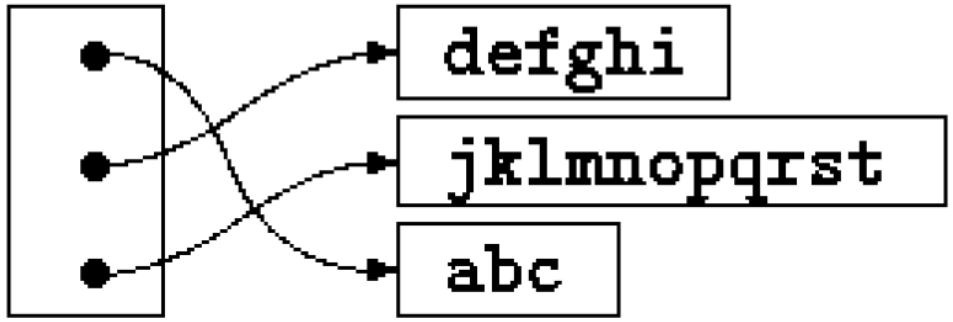
```
char *lineptr[MAXLINES];
```



Los arreglos de punteros permiten la gestión eficiente de datos.

Un algoritmo para ordenar arreglos de caracteres es mucho más rápido si simplemente se ordenan las direcciones de memoria de los arreglos de punteros.





Un arreglo de punteros puede ser tratado como cualquier puntero.

```
/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
        while (nlines-- > 0)
                printf("%s\n", *lineptr++);
}
```

Demostración

Para la próxima clase...

Lectura:

- Computer Systems, Bryant y O'Hallaron. Secciones 2.1.1-6, 2.2.1
- Capítulo 5 de **The C programming Language** (Brian W. Kernighan and Dennis M. Ritchie)

Práctica:

- Uso de punteros en C