



Introducción a la Estructura de Datos

¿Qué es Información?

Unidad básica de información (BIT)

1 bit  2 posiciones (0 – 1)

n bit  2^n posiciones

Enteros binarios

Enteros Positivos \rightarrow 00100110 $\rightarrow 2^1 + 2^2 + 2^5 = 2 + 4 + 32 = 38$

Números Binarios Negativos

Notación de complemento a uno 11011001 = -38

El primer dígito representa positivo o negativo, en caso de ser negativo se invierten todos los valores de los otros BITS

Con n bits se puede representar desde $-2^{(n-1)}-1$ hasta $2^{(n-1)}-1$

Observar que se puede definir el 0 de dos maneras 0+ y 0-

Notación de complemento a dos 11011010 = -38

En este caso se suma un 1 a la representación del complemento a 1 del número negativo.

Con n bits se puede representar desde $-2^{(n-1)}$ hasta $2^{(n-1)}-1$

Estas no son las únicas maneras de representar Enteros Binarios

Números Reales

La más utilizada es: **Notación de Punto Flotante**

El número real se representa por un número llamado **MANTISA** multiplicado por una base elevada a una potencia entera, llamada **EXPONENTE**.

Ej. El número 387,53 se representa como 38753×10^{-2}

Normalmente se usan 32 bits, MANTISA (24) EXPONENTE (8)

Ej. 000000001001011101100001 11111110

Esto nos permite representar números desde $2^{23-1} \times 10^{127}$ hasta números como 10^{-128} , el factor que limita la precisión es el número de dígitos significativos de la mantisa, por ej. el número 10.000.001 requiere 24 dígitos y deberá representarse como $10.000.000 (1 \times 10^7)$

Cadena de Caracteres

Interpretación no solo de números sino de caracteres.

Si 8 bits representan un carácter, se pueden representar hasta 256 caracteres diferentes.

Esta selección es totalmente arbitraria, algunas computadoras utilizan 7 bits, otras 8 bits, y otras 10 bits.

Definimos como **BYTE** al grupo de bits que permiten representar un carácter en una computadora.

Esta pauta puede hacerse como se desee, pero se intenta que sea de tipo CONSISTENTE.

Por último observemos que la cadena de bits, 00100110, puede ser el número 38 (binario) el 26 (decimal codificado) o el símbolo & (carácter).

Estos son distintos **TIPOS DE DATOS**

Hardware y Software

- La memoria de una computadora es simplemente un grupo de **bits** (0 o 1)
- Los bits están agrupados en unidades más grandes, los **bytes**.
- Toda computadora tiene un conjunto de **tipo de datos** nativos, o sea, que se construyo para manejar determinado tipo de datos.
- Alojarse en una dirección de memoria la suma de las pautas de bits, alojados en otras 2 direcciones de memoria, estas son algunas de las funciones que están previstas en una computadora.
- Un lenguaje de programación de alto nivel, ayuda a simplificar esta tarea al programador.

El concepto de implantación

- Primero debemos separar el concepto de TIPO DE DATOS que se representa en una computadora, por el de **TIPO DE DATO ABSTRACTO**.
- Implantación de Hardware: se diseña y construye el circuito necesario para ejecutar la operación requerida.
- Implantación de Software: se escribe un programa que consta de instrucciones existentes en el hardware para interpretar en la forma deseada las cadenas de caracteres y ejecutar las operaciones requeridas.
- De acá en adelante cuando digamos **Implantación** hablaremos de **IMPLANTACION DE SOFTWARE**.

El concepto de implantación

EJEMPLO

Hw. MOVE(origen, destino, largo)

Sw. MOVEVAR(origen, destino)

 MOVE(o, d, 1)

 for (i=1; i<d; i++)

 MOVE(o[i], d[i], 1);

El concepto de implantación

EJEMPLO

Definir CONCATVAR(c1, c2, c3) para concatenar dos cadenas de caracteres de longitud variable.

	5	H	E	L	L	O	
--	---	---	---	---	---	---	--

	9	E	V	E	R	Y	B	O	D	Y	
--	---	---	---	---	---	---	---	---	---	---	--

1	4	H	E	L	L	O	E	V	E	R	Y	B	O	D	Y	
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	--

```
Sw.  z = c1 + c2;  
      MOVE( z, c3, 1 )  
      for (i=1; i<= c1; MOVE( c1[i], c3[i], 1 ) );  
      for (i=1; i<= c2; {  
          x = c1 + i;  
          MOVE( c2[i], c3[x], 1); }
```


Tipo de datos Abstracto

Una herramienta útil para especificar las propiedades lógicas de los tipos de datos, es el **tipo de dato abstracto ADT**, el cual es, una colección de valores y un conjunto de operaciones con esos valores.

La colección y las operaciones forman una construcción matemática que puede implantarse utilizando una estructura de datos particular, ya sea de Hw o de Sw.

El término ADT se refiere al concepto matemático que define el tipo de datos.

Al definir un ADT no nos importa la eficiencia en tiempo y espacio.

Ej. ADT Rational

```
abstract typedef <integer, integer> RATIONAL;  
condition RATIONAL[1] <> 0;
```

```
abstract RATIONAL makerational( a, b )  
int a, b;  
precondition b <> 0;  
postcondition      makerational[0] == a;  
                  makerational[1] == b;
```

```
abstract RATIONAL add( a, b )  
RATIONAL a, b;  
postcondition      add[1] == a[1] * b[1];  
                  add[0] == a[0] * b[1] + a[1] * b[0];
```

```
abstract RATIONAL mult( a, b )  
RATIONAL a, b;  
postcondition      mult[0] == a[0] * b[0];  
                  mult[1] == a[1] * b[1];
```

```
abstract RATIONAL equal( a, b )  
RATIONAL a, b;  
postcondition      equal == (a[1] * b[0] == a[0] * b[1]);
```

Ej. ADT para cadena de caracteres

```
abstract typedef <char> STRING;
```

```
abstract length(s)
```

```
STRING s;
```

```
postcondition    length == len (s);
```

```
abstract STRING concat( s1, s2 )
```

```
STRING s1, s2;
```

```
postcondition    concat == s1 + s2;
```

```
abstract STRING substr( s1, i, j )
```

```
STRING s1;
```

```
int i, j;
```

```
precondition    0 <= i < len( s1);
```

```
0 <= j <= len( s1 ) - i;
```

```
postcondition    substr == sub( s1, i, j );
```

Ej. ADT para arreglos

```
abstract typedef <eltype, ub> ARRTYPE( ub, eltype);  
condition type( ub ) == int;
```

```
abstract eltype extract( a, i )  
ARRTYPE( ub, eltype ) a;  
int i;  
precondition          0 <= i < ub;  
poscondition          extract == ai
```

```
abstract store( a, i, elt )  
ARRTYPE( ub, eltype ) a;  
int i;  
eltype elt;  
precondition          0 <= i < ub;  
postcondition         a[i] == elt;
```

Implantación de arreglos unidimensionales

Ej. Lenguaje C

```
int    b[100]
```

Reserva 100 localidades sucesivas de memoria, cada una lo bastante grande para contener un solo entero.

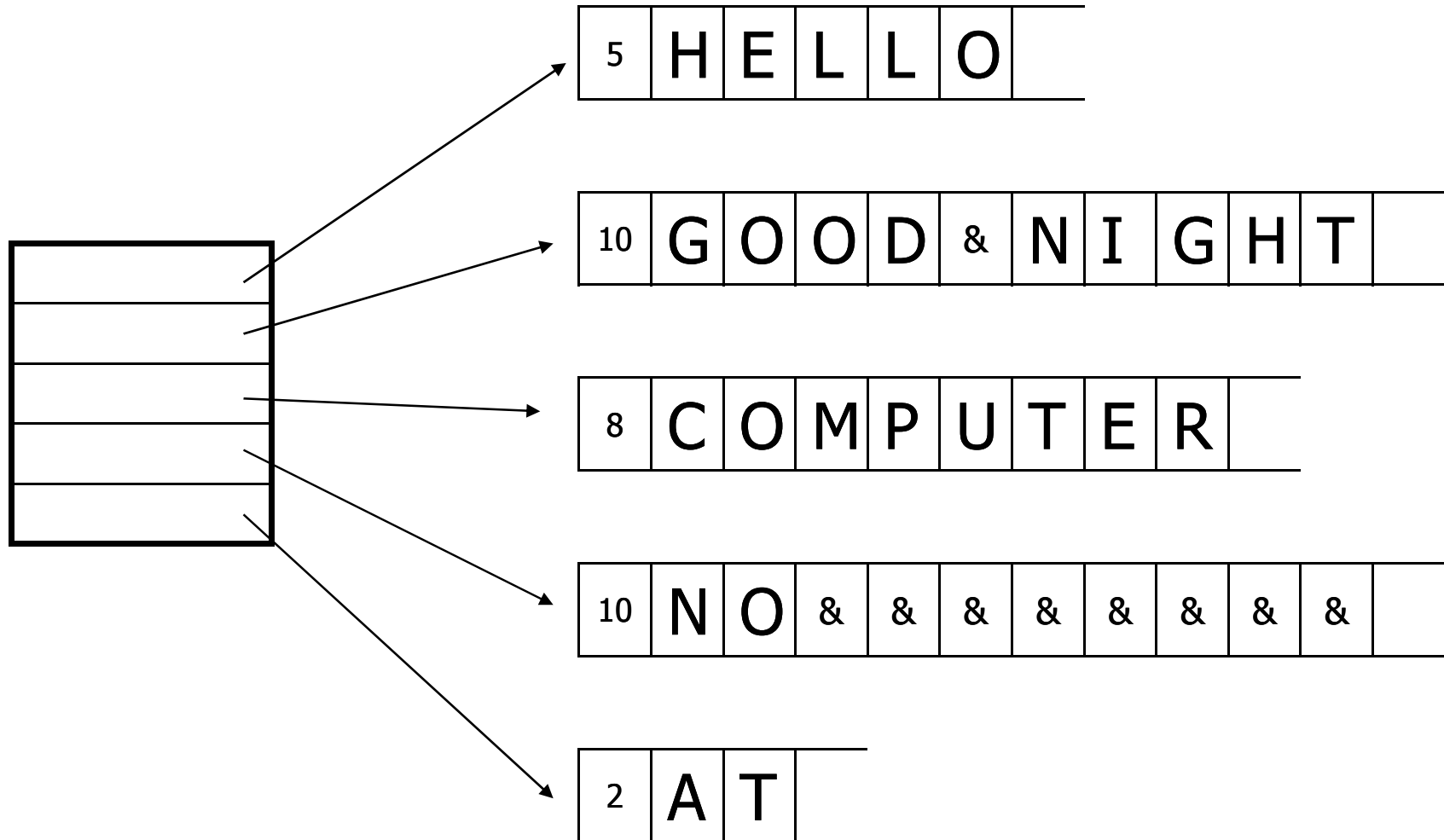
La dirección de la primer localidad de b se denomina **dirección base**.

Suponiendo que el tamaño de cada elemento individual del arreglo es x, para acceder al elemento número 8 del arreglo debemos dirigirnos a la posición de memoria

$$\text{base} + 8 * x$$

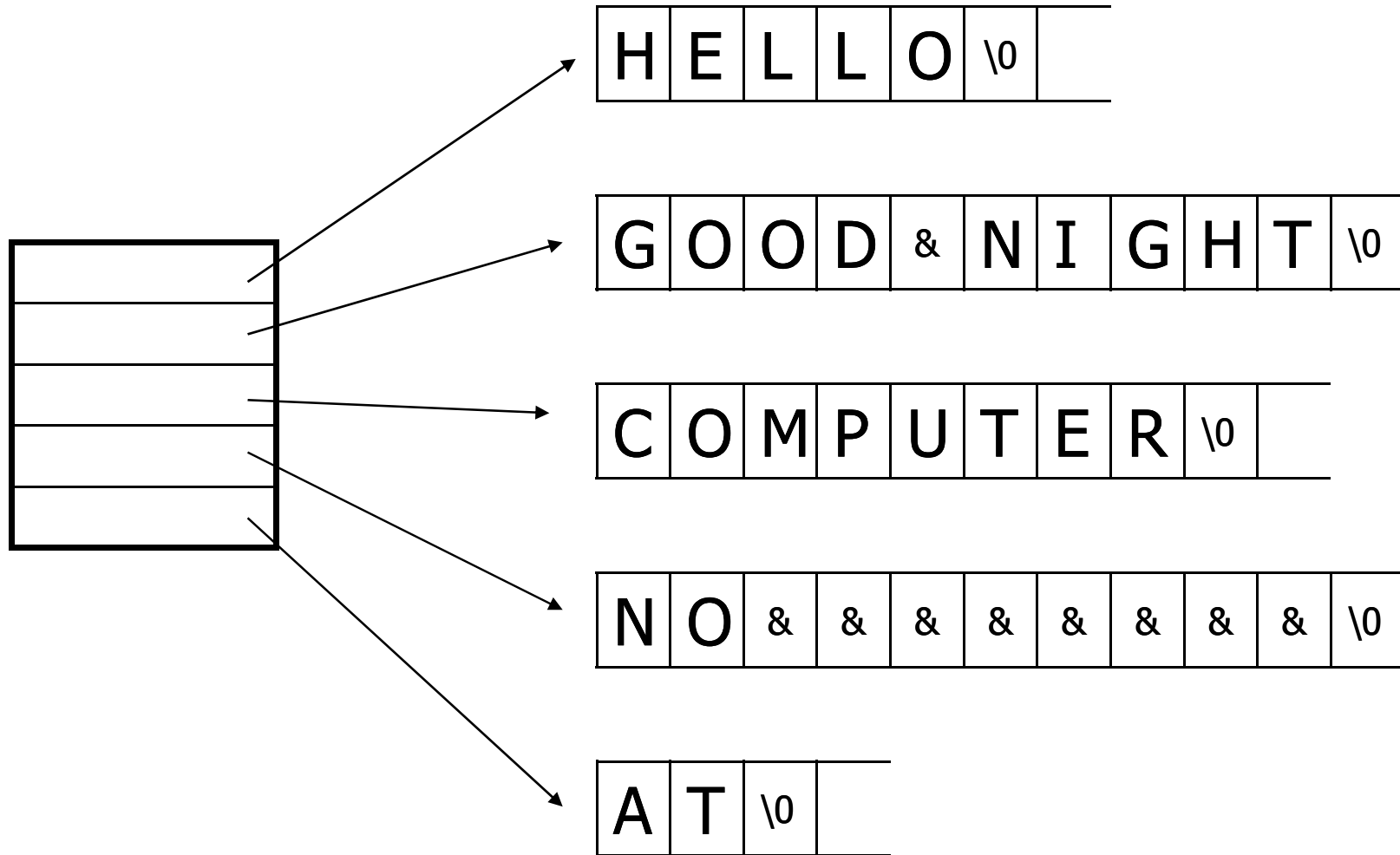
¿Cómo implantar una arreglo de elementos de longitud variable?

OPCION 1



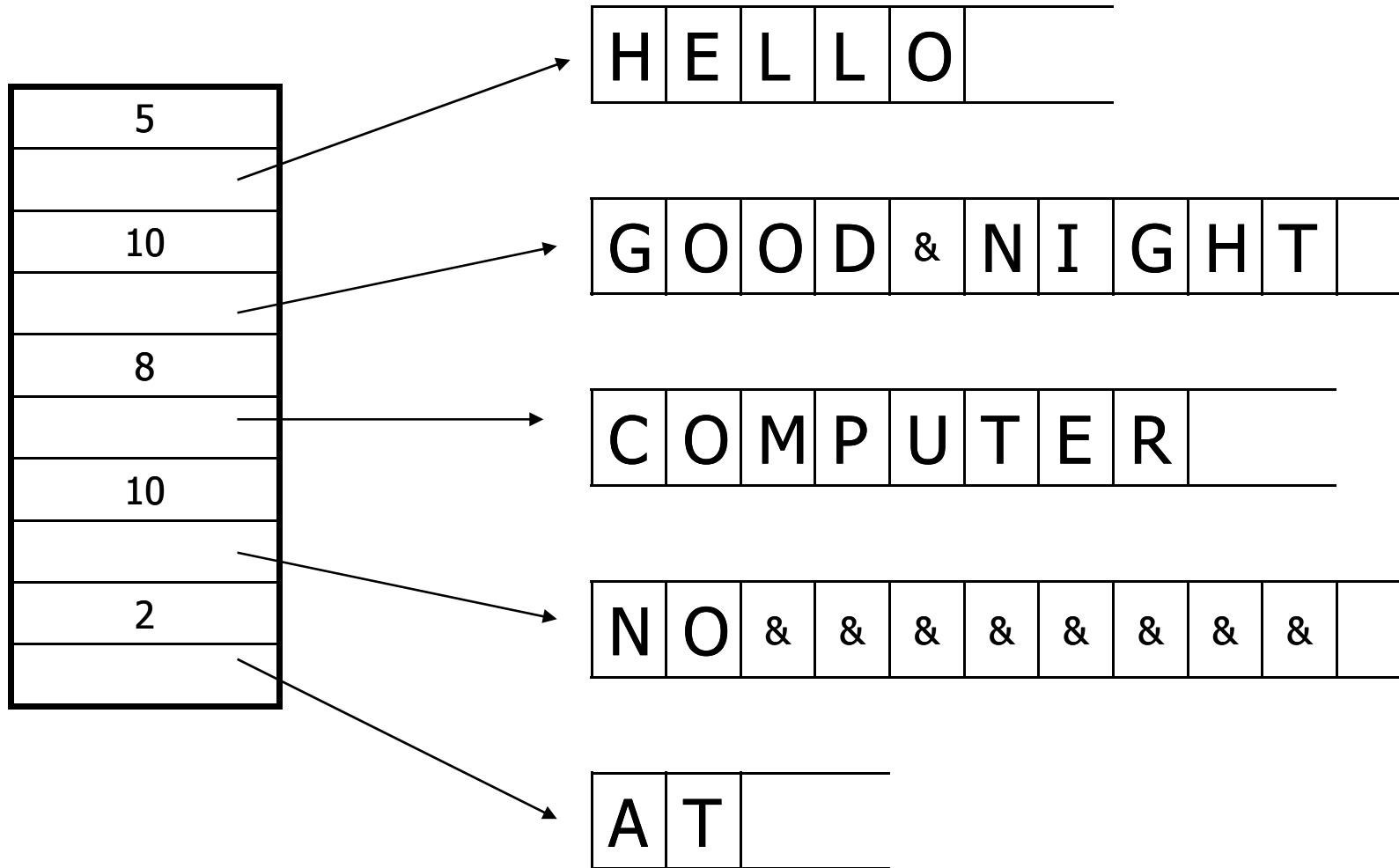
¿Cómo implantar una arreglo de elementos de longitud variable?

OPCION 2



¿Cómo implantar una arreglo de elementos de longitud variable?

OPCION 3



Implantación de arreglos bidimensionales

Ej. Lenguaje C

```
int    b[3][5]
```

Lo anterior reserva un arreglo de 3 elementos, cada uno de los cuales es un arreglo de 5 elementos.

A esto lo llamamos arreglo bidimensional, y así lo imaginamos aunque el computador lo trate como un arreglo unidimensional de manera interna.

El método para representar en memoria este arreglo es el denominado **RENGLON-MAYOR**. De esta manera el primer renglón del arreglo ocupa el primer conjunto de localidades de memoria.

Suponiendo que deseamos alcanzar la dirección de memoria del elemento $b[i_1][i_2]$ debemos calcularlo mediante

$$\text{base}(b) + (i_1 * r_2 + i_2) * x$$

Representación de arreglos bidimensionales

	Columna 0	Columna 1	Columna 2	Columna 3	Columna 4
Renglón 0					
Renglón 1					
Renglón 2					

$$\text{base}(b) + (i_1 * r_2 + i_2) * x$$

$$\text{base}(b) + (1 * 5 + 3) * x$$

Representación en la memoria del computador

OPCION 1

encabezado

Renglón 0

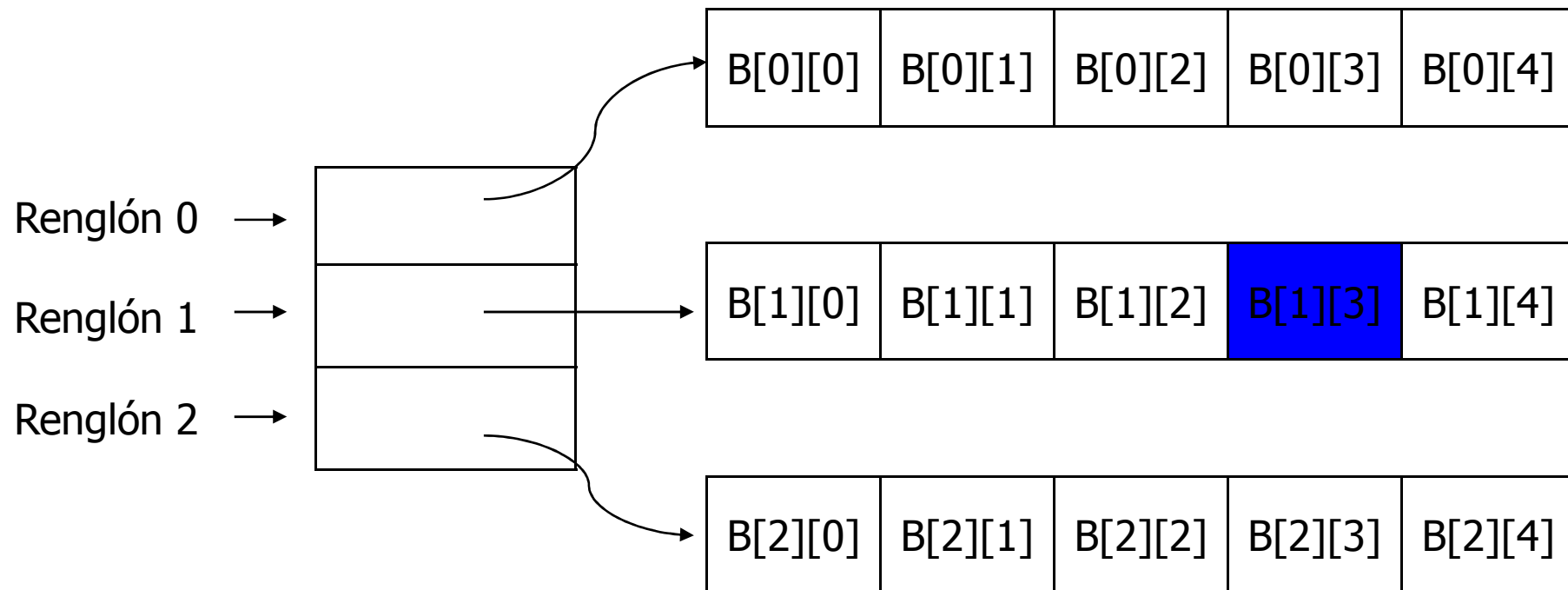
Renglón 1

Renglón 2

0
2
0
4
B[0][0]
B[0][1]
B[0][2]
B[0][3]
B[0][4]
B[1][0]
B[1][1]
B[1][2]
B[1][3]
B[1][4]
B[2][0]
B[2][1]
B[2][2]
B[2][3]
B[2][4]

Representación en la memoria del computador

OPCION 2



Implantación de arreglos multidimensionales

Ej. Lenguaje C

```
int    b[3][2][4];
```

Podemos imaginar este arreglo como un “cubo”, con valores alojados en cada uno de los elementos que lo conforman.

Pero podemos ir más allá de las 3 dimensiones.

```
int    c[7][15][3][5][8][2];
```

El método de **RENGLON-MAYOR** puede extenderse para arreglos de más de 2 dimensiones.

*Analizar el lugar que ocupan cada arreglo en memoria.

La posición en memoria para localizar cualquier ubicación de estos arreglos, se puede calcular mediante:

$$\text{base}(\text{ar}) + (\text{in} * \text{rn} * (i_{(n-1)} + r_{(n-1)} * (\dots + r_3 * (i_2 + r_2 * i_1) \dots))) * x$$