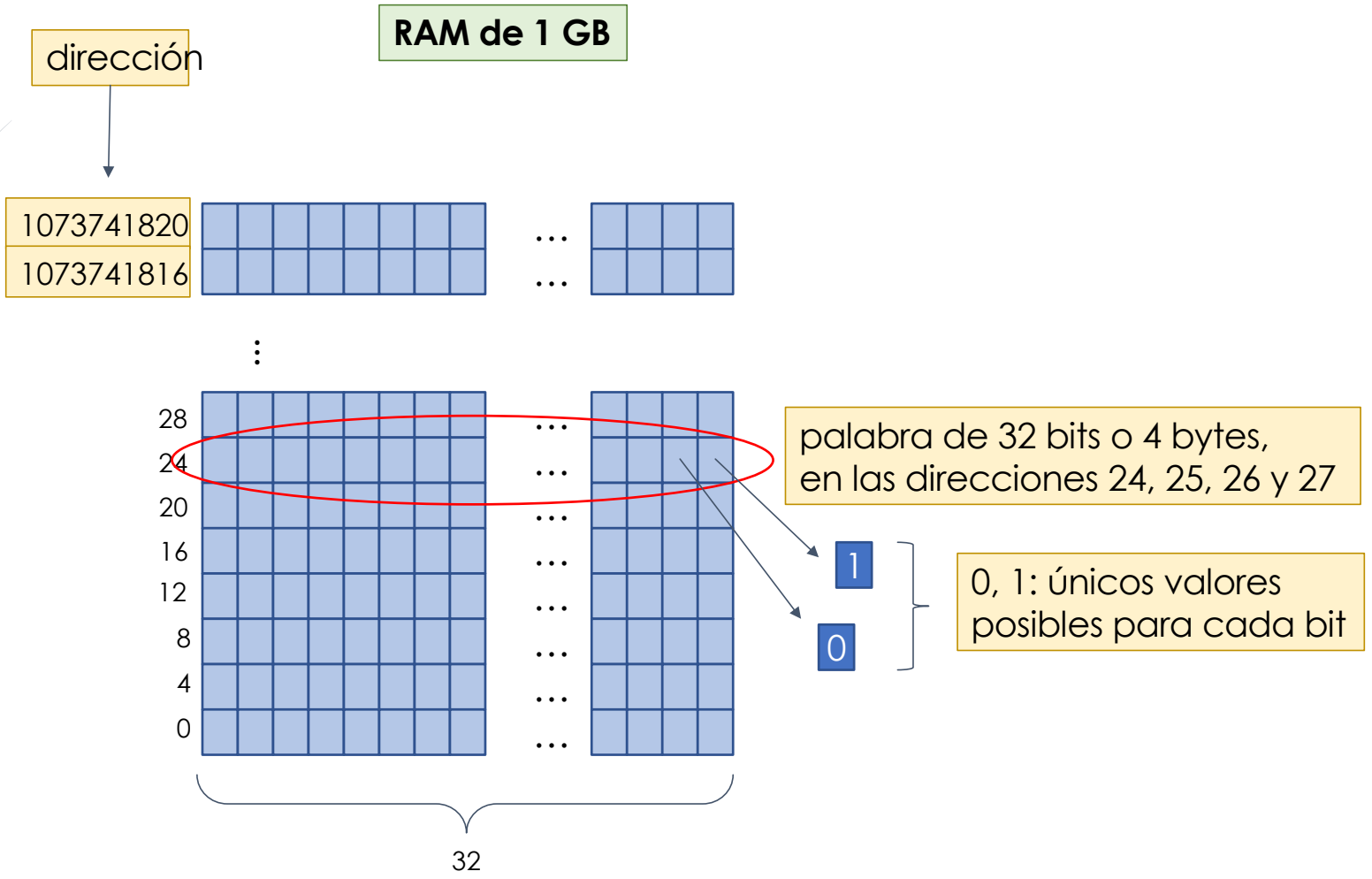


Tipos de datos

Fundamentos de Lenguajes de Programación – IIC2560

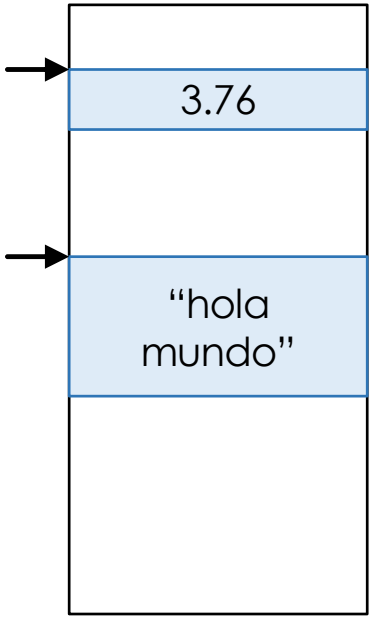
Yadran Eterovic S. (yadran@uc.cl)


RAM de 1 GB
dirección
1073741820
1073741816
28
24
20
16
12
8
4
0
palabra de 32 bits o 4 bytes,
en las direcciones 24, 25, 26 y 27
1
0
0, 1: únicos valores
posibles para cada bit
32

Las variables de un programa “viven” en la memoria del computador

Cada una tiene:

- dirección (ubicación) en memoria
- tamaño, en número de bytes o en número de palabras
- valor



Un **tipo de datos** define

- una **colección de valores**
- y un **conjunto de operaciones predefinidas** sobre esos valores

Primitivo

Numérico

`Integer`

El hardware de los computadores permite varios tamaños de *integers*

P.ej., Java: **`byte`**, **`short`**, **`int`**, **`long`** … todos con signo

P.ej., C++, C#: incluyen *integers* sin signo

En computadores en que las palabras tienen 32 bits

… podemos representar 2^{32} patrones diferentes de bits

… los números desde el 0 hasta el $2^{32} – 1 = 4{,}294{,}967{,}295_{10}$

Con 32 bits, el valor del número representado como

$$x_{31}\, x_{30} \cdots x_1\, x_0$$

… es

$$(x_{31} \times 2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Estos números positivos, incluyendo el 0, se llaman **números sin signo** (*unsigned numbers*) o **enteros sin signo** (*unsigned integers*)

También tenemos que representar **números negativos**

P.ej., podríamos agregar un signo, representado en un bit adicional:

$0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1_2 = 421_{10}$ $\quad$ $1\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1_2 = -421_{10}$

Esta representación se llama **signo y magnitud**

$0\,1\,1\,0\,1\,0\,0\,1\,0\,1_2 = 421_{10}$ $1\,1\,1\,0\,1\,0\,0\,1\,0\,1_2 = -421_{10}$

Problemas de la representación *signo y magnitud*:

- el bit de signo, ¿es el de más a la izquierda o el de más a la derecha?
- al sumar, se necesita un paso adicional para saber el valor de este bit
- hay dos ceros —uno positivo y otro negativo ($0\,0\ldots0\,0_2$ y $1\,0\ldots0\,0_2$)

Hay otras opciones, todas con sus pros y sus contras

No habiendo una mejor opción obvia

… los arquitectos computacionales eligieron finalmente la representación llamada ***complemento de 2*** que hacía más simple el hardware:

- 0s a la izquierda significa positivo
- 1s a la izquierda significa negativo

Todo computador hoy en día, y desde 1965, usa complemento de 2 para representar números con signo

Veámoslo para 32 bits (próxs. diaps.)

[muchos computadores hoy día usan 64 bits y en el pasado usaban 16 bits —la representación en *complemento de 2* es independiente del número de bits: básicamente, lo único que se ve afectado es el rango de números representados]

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$$
$$\vdots$$
$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2{,}147{,}483{,}645_{10}$$
$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2{,}147{,}483{,}646_{10}$$
$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2{,}147{,}483{,}647_{10}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2{,}147{,}483{,}648_{10}$$
$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2{,}147{,}483{,}647_{10}$$
$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2{,}147{,}483{,}646_{10}$$
$$\vdots$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$

⋮

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2{,}147{,}483{,}645_{10}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2{,}147{,}483{,}646_{10}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2{,}147{,}483{,}647_{10}$

esta mitad positiva, de 0 a $2{,}147{,}483{,}647_{10}$ $(= 2^{31} - 1)$ usa la misma representación que antes

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2{,}147{,}483{,}648_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2{,}147{,}483{,}647_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2{,}147{,}483{,}646_{10}$

⋮

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$

número más negativo, $-2{,}147{,}483{,}648_{10}$ $(= -2^{31})$; no tiene un número positivo correspondiente

secuencia de números negativos de magnitud decreciente, desde $-2{,}147{,}483{,}647_{10}$ $(= 1000\ldots001_2)$ hasta -1_{10} $(= 1111\ldots111_2)$

Así, para 32 bits en *complemento de 2*, el valor del número representado como

$$x_{31}\, x_{30} \cdots x_1\, x_0$$

… es

$$(\mathbf{x_{31} \times -2^{31}}) + (\mathbf{x_{30} \times 2^{30}}) + (\mathbf{x_{29} \times 2^{29}}) + \cdots + (\mathbf{x_1 \times 2^1}) + (\mathbf{x_0 \times 2^0})$$

Notar la diferencia en la interpretación del valor del bit 31 en el caso de los números sin signo :

$$(x_{31} \times 2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Primitivo

Numérico

`Floating-point`

El hardware de los computadores permite varios tamaños de *integers*

P.ej., Java: **`byte`**, **`short`**, **`int`**, **`long`** … todos con signo

P.ej., C++, C#: incluyen *integers* sin signo

Representación de números de punto flotante

$$\pm\ 1.xxxxxxxx\ldots \times 2^{yyy\ldots}$$

El número total de bits para representar un número real queda fijo al momento de diseñar el computador (igual que en el caso de los números enteros)

- p.ej., para 32 bits

  … el signo, el 1, los *xxx*…, el 2 y los *yyy*… deben repartirse los 32 bits



en 32 bits

⟹ compromiso entre el número de bits para la fracción —o mantisa— y el número de bits para el exponente:

hay que quitarle un bit a la fracción (los *xxx*…) para agregárselo al exponente (los *yyy*…), y viceversa

… es decir, compromiso entre ***precisión*** y ***rango***:

- mayor número de bits de la fracción ⟹ mayor precisión de la fracción
- mayor número de bits del exponente ⟹ mayor rango de los números representables

En la práctica, desde 1985 se usa el estándar *IEEE 754*

Fracción:

23 bits —los 23 bits menos significativos de la palabra (del bit 0 al bit 22)

el 1 a la izquierda del punto decimal es ***implícito***

$\Longrightarrow$ el ***significante*** tiene 24 bits

Signo:

el bit más significativo (bit 31)

$\Longrightarrow$ facilita las comparaciones < 0, $= 0$, > 0

Exponente:

8 bits : bits 23 al 30

inmediatamente a continuación del signo

… facilita la ordenación usando instrucciones de comparación de números enteros

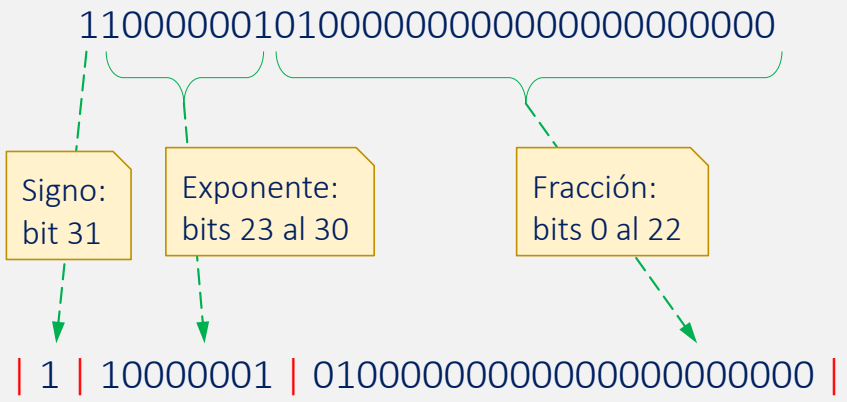
… siempre que todos los exponentes tengan el mismo signo:

… ⟹ se los representa como números sin signo desplazados (*biased*) en 127

(en lugar de representarlos como números enteros en complemento de 2)

El valor de un número en punto flotante en IEEE 754 es

$$(-1)^{signo} \times (1+fracción) \times 2^{exponente-127}$$



Combinaciones especiales de *exponente* y *fracción*:

0 : *exponente* = 0 y *fracción* = 0

número no normalizado : *exponente* = 0 y *fracción* ≠ 0

infinito : *exponente* = 255 (o 2047) y *fracción* = 0

⟹ los exponentes "válidos" son desde el 1 al 254, que representan los valores –126 a +127

NaN (*not a number*): *exponente* = 255 (o 2047) y *fracción* ≠ 0

Primitivo

Numérico

`Complex`

Pares ordenados de valores en punto flotante

P.ej., Fortran, Python

Primitivo

`Boolean`

El más simple de todos los tipos

Su rango de valores tiene sólo dos elementos:

- uno para verdadero
- uno para falso

Primitivo

Caracteres individuales

Tradicionalmente, ASCII, de 8 bits: 128 caracteres

… también, ISO 8859-1, de 8 bits: 256 caracteres

Almacenados usando codificación numérica

Hoy, Unicode, de 16 bits:

- Java, JavaScript, Python, Perl, C#, F#

También, UTF-32 (UCS-4), de 4 bytes

Strings de caracteres: Los valores son secuencias de caracteres

Los strings constantes se usan para etiquetar el output

Input/output de todo tipo de datos se hace en términos de strings

Son esenciales en cualquier programa que manipula caracteres

Los dos aspectos de diseño más importantes:

- ¿Son un tipo especial de *array* de caracteres o un tipo primitivo?
- ¿Deberían tener longitud estática o dinámica?

Las operaciones más comunes:

- asignación, concatenación, referencia a substrings, comparación, *pattern matching*

Asignación y comparación se ven afectadas por la posibilidad de strings de diferentes largos

Pattern matching puede ser provisto directamente por el lenguaje o a través de una librería (función, clase)

En C y C++ los strings son almacenados en arrays de caracteres individuales (`char`):

- los strings terminan con un carácter especial, `null`, representado por 0

Como alternativa a mantener variables que almacenan el largo del strings

Funciones comunes de librería:

- `strcpy`, mueve strings
- `strcat`, concatena un string a otro
- `strcmo`, compara lexicográficamente
- `strlen`, devuelve el número de caracteres del string, sin contar null

P.ej.,

```
char str[] = “apples”;
```

… especifica que el array `str` almacena `apples0`

Parámetros y valores de retorno, cuando corresponde, son punteros a *arrays* of `char`, o literales string

Funciones comunes de librería:

- `strcpy`, mueve strings
- `strcat`, concatena un string a otro
- `strcmo`, compara lexicográficamente
- `strlen`, devuelve el número de caracteres del string, sin contar null

Función insegura: no hay protección en contra de rebalsar el string destino, porque `strcpy` no conoce su largo:

- la clase `string` de la librería estándar de C++ es segura

Parámetros y valores de retorno, cuando corresponde, son punteros a *arrays* of `char`, o literales string

La longitud de un string puede ser estática y quedar definida al momento de la creación del string:

- strings en Python, la clase String de Java y en Ruby

Strings de largo estático

… o bien puede ser variable hasta un máximo fijo declarado en la definición:

- strings en C, y strings estilo C en C++

Strings de largo dinámico limitado

… o puede ser variable sin un máximo:

- strings en JavaScript, Perl y la librería estándar de C++

Strings de largo dinámico:

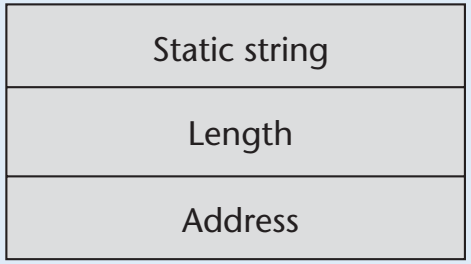
- requiere asignación y devolución dinámica de memoria

Los strings podrían ser provistos directamente por el hardware

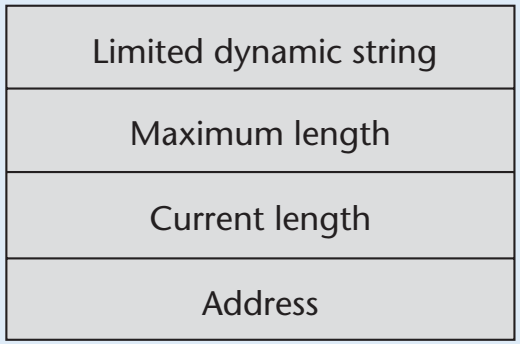
… pero en general se usa software para implementar su almacenamiento, recuperación y manipulación

… a través de **descriptores**

Descriptor para strings estáticos, sólo necesario durante la compilación



Descriptor para strings dinámicos limitados, excepto en C y C++



Strings dinámicos requieren manejo de memoria

El string puede ser almacenado en una lista ligada:

- cuando el string crece, las celdas adicionales pueden provenir de cualquier parte en el heap
- almacenamiento extra para los links
- complejidad de las operaciones

El string puede ser almacenado en celdas consecutivas de memoria:

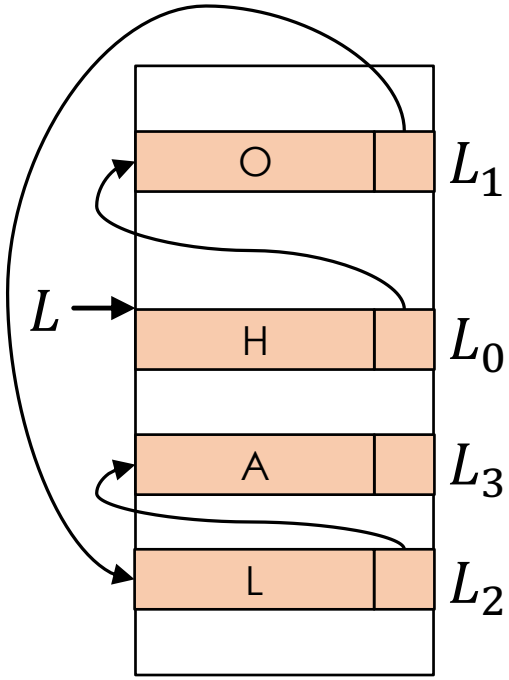
- cuando el string crece, típicamente es necesario buscar otra área de la memoria en que quepa el nuevo string completo
- el string original es copiado a esta área y la memoria que ocupaba es devuelta

Secuencia de largo variable de celdas del mismo tamaño

Las celdas ocupan posiciones cualquiera en la memoria, pero están conectadas entre ellas mediante punteros:

- palabras que contienen la dirección de la próxima celda

No permite acceso eficiente por índice

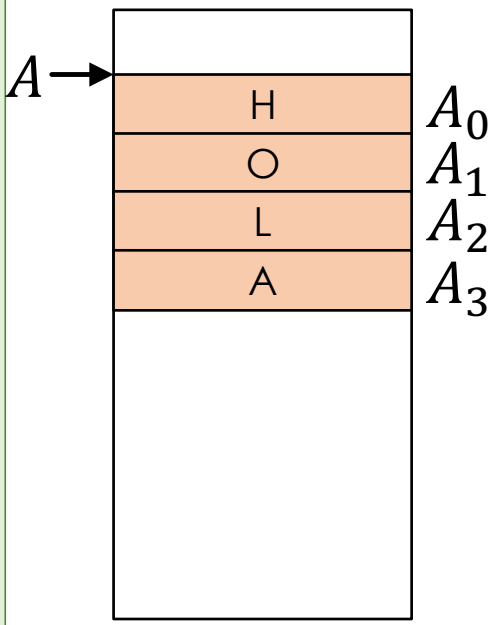


Es una secuencia de largo fijo de “celdas” de memoria del mismo tamaño (en número de bytes o de palabras)

… y que almacenan valores del mismo tipo (todos *integer*, o todos *strings*)

Se almacena de manera contigua en memoria

Permite acceso por índice en tiempo $O(1)$



Un **array** es una agregación de elementos del mismo tipo

… en que un elemento individual es identificado por su posición dentro de la agregación, relativa al primer elemento

Un **array** es una agregación de elementos del mismo tipo

… en que un elemento individual es identificado por su posición dentro de la agregación, relativa al primer elemento

Decisiones de diseño:

- ¿Qué tipos de datos son válidos para los índices?
- ¿Se chequea el rango de los índices en las referencias a elementos?
- ¿En qué momento se define el rango de los índices?
- ¿En qué momento se le asigna memoria a un array?
- ¿En qué momento se puede inicializar un array?
- ¿Qué clases de *slices* ("secciones") se permiten?

Los elementos individuales de un array son referenciados mediante un mecanismo sintáctico de dos niveles:

<nombre><selector>

- *mapping* desde el nombre del array y el conjunto de los valores de los índices a un elemento en la agregación:

nombre_del_array(*lista_de_valores_de_Índices*) ⟶ elemento

- la sintaxis es relativamente universal:

<nombre>(*<lista_de_índices>*)

<nombre>[*<lista_de_índices>*]

Los elementos individuales de un array son referenciados mediante un mecanismo sintáctico de dos niveles:

Nombre de la agregación

<nombre><selector>

- Posiblemente dinámico
- Uno o más índices

- *mapping* desde el nombre del array y el conjunto de los valores de los índices a un elemento en la agregación:

nombre_del_array(*lista_de_valores_de_Índices*) ⟶ elemento

- la sintaxis es relativamente universal:

<nombre>(*<lista_de_índices>*) Fortran, Ada

<nombre>[*<lista_de_índices>*] La mayoría de los lenguajes

Los elementos individuales de un array son referenciados mediante un mecanismo sintáctico de dos niveles:

Nombre de la agregación

<nombre><selector>

- Posiblemente dinámico
- Uno o más índices

- *mapping* desde el nombre del array y el conjunto de los valores de los índices a un elemento en la agregación:

nombre_del_array(*lista_de_valores_de_Índices*) ⟶ elemento

- la sintaxis es relativamente universal:

<nombre>(*<lista_de_índices>*) Fortran, Ada

<nombre>[*<lista_de_índices>*] La mayoría de los lenguajes

C

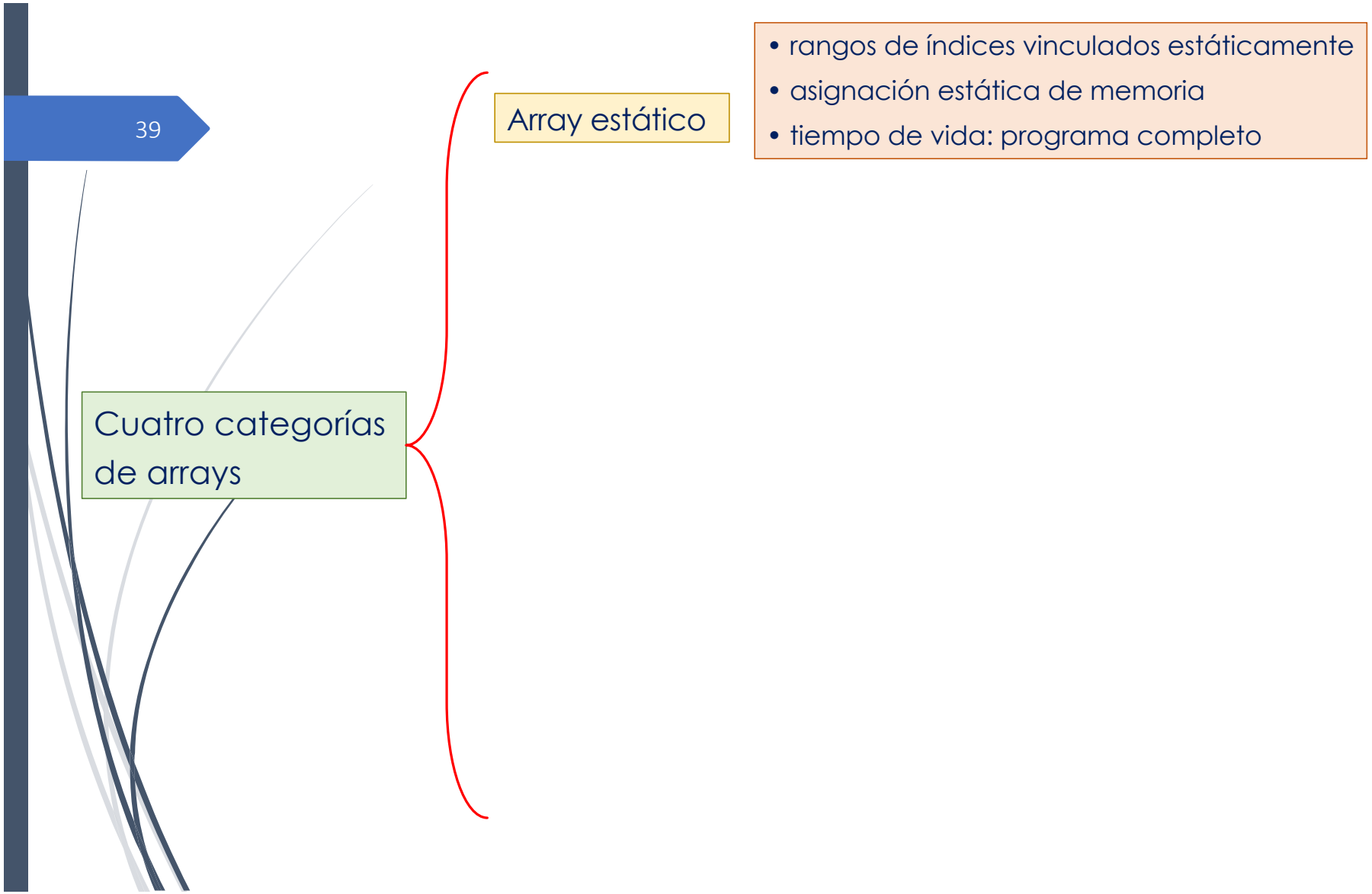
```
char upper[26];
```

Fortran

```
character, dimension (1:26) :: upper
character (26) upper    ! shorthand notation
```

El tipo de datos de un array es especificado por dos tipos de datos:

- el tipo de datos de los elementos
- el tipo de datos de los índices, a menudo, *integer*


Cuatro categorías de arrays
Array estático
• rangos de índices vinculados estáticamente
• asignación estática de memoria
• tiempo de vida: programa completo

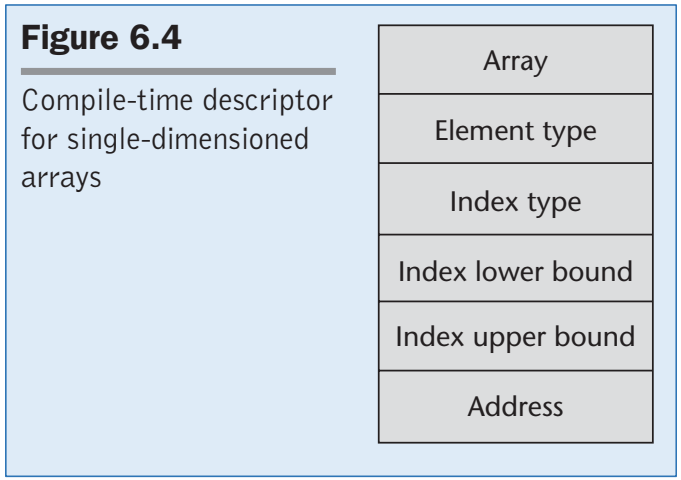


Figure 6.4

Compile-time descriptor for single-dimensioned arrays

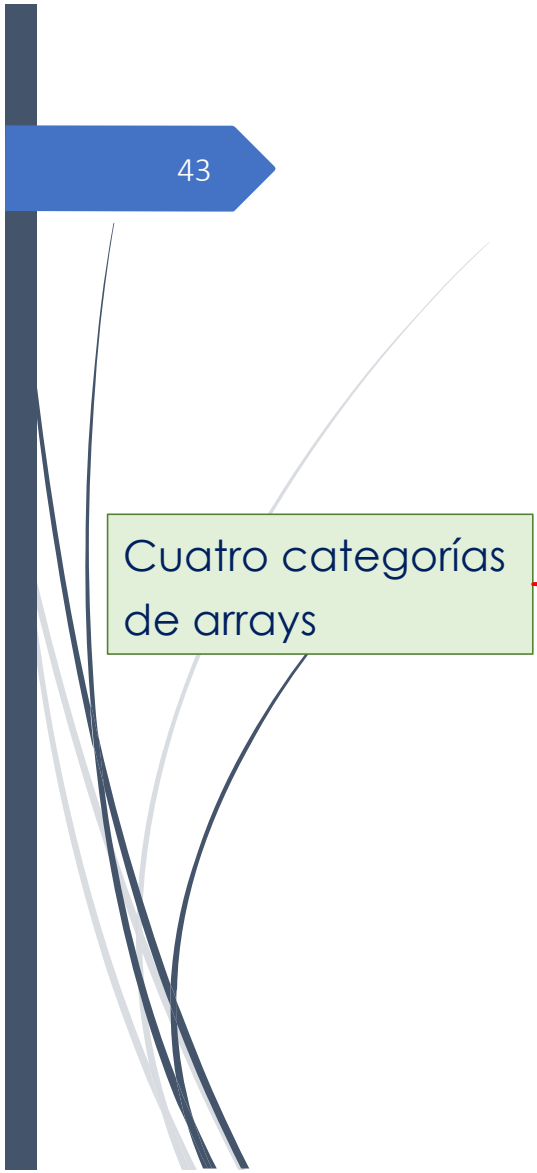
Figure 6.6

A compile-time descriptor for a multidimensional array



Cuatro categorías de arrays

- Array estático
  - rangos de índices vinculados estáticamente
  - asignación estática de memoria
  - tiempo de vida: programa completo
- Array fijo, dinámico en el stack
  - rangos de índices vinculados estáticamente
  - asignación dinámica de memoria del stack durante la elaboración de la declaración
  - tiempo de vida: llamada a subrutina



Cuatro categorías de arrays

- **Array estático**
  - rangos de índices vinculados estáticamente
  - asignación estática de memoria
  - tiempo de vida: programa completo
- **Array fijo, dinámico en el stack**
  - rangos de índices vinculados estáticamente
  - asignación dinámica de memoria del stack durante la elaboración de la declaración
  - tiempo de vida: llamada a subrutina
- **Array fijo, dinámico en el heap**
  - rangos de índices vinculados cuando el programa lo pide durante su ejecución
  - asignación dinámica de memoria del heap durante la elaboración de la declaración
  - tiempo de vida: más general

Cuatro categorías de arrays

- **Array estático**
  - rangos de índices vinculados estáticamente
  - asignación estática de memoria
  - tiempo de vida: programa completo
- **Array fijo, dinámico en el stack**
  - rangos de índices vinculados estáticamente
  - asignación dinámica de memoria del stack durante la elaboración de la declaración
  - tiempo de vida: llamada a subrutina
- **Array fijo, dinámico en el heap**
  - rangos de índices vinculados cuando el programa lo pide durante su ejecución
  - asignación dinámica de memoria del heap durante la elaboración de la declaración
  - tiempo de vida: más general
- **Array dinámico en el heap**
  - la vinculación de los rangos de los índices y la asignación de memoria son dinámicas … y pueden cambiar durante el tiempo de vida del array

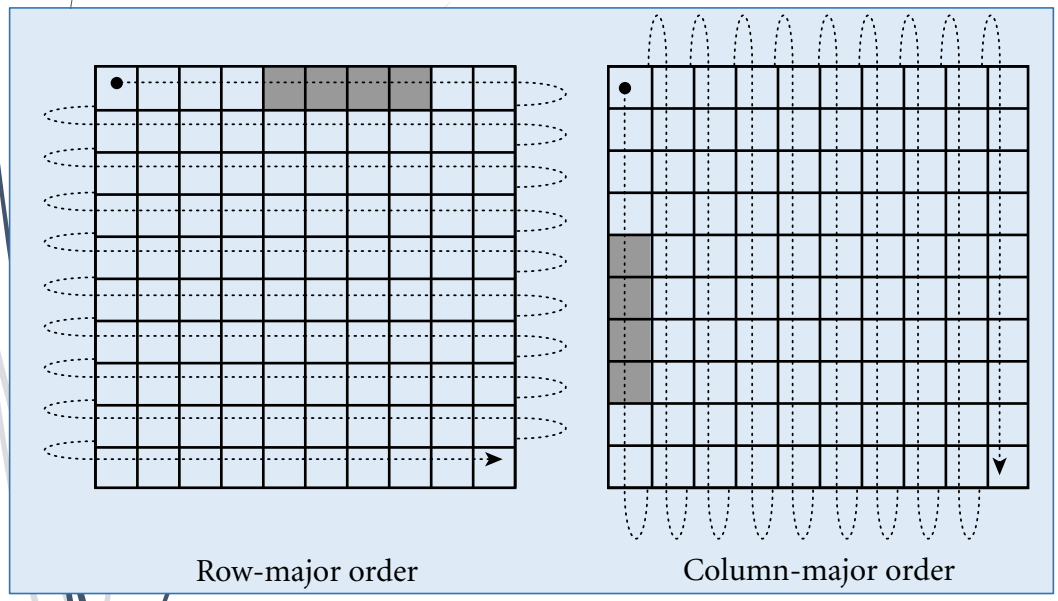


Figure 6.6

A compile-time descriptor for a multidimensional array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 0
⋮
Index range n – 1
Address

3 4 7
6 2 5
1 3 8

Almacenamiento
row major order

3, 4, 7, 6, 2, 5, 1, 3, 8

location(a[i,j]) = address of a[0, 0]
+ ((((number of rows above the *ith* row) * (size of a row))
+ (number of elements left of the *jth* column)) *
element size)

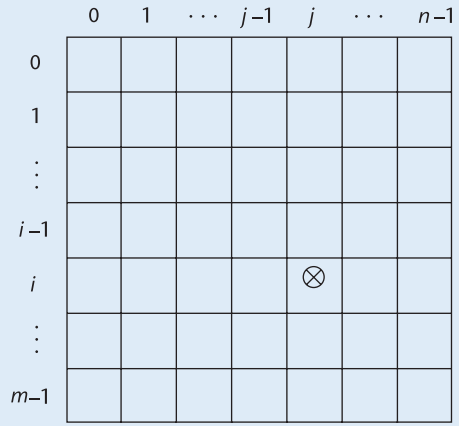
location(a[i, j]) = address of a[0, 0] + (((i * n) + j) *
element_size)

location(a[i, j]) = address of a[row_lb, col_lb]
= (((i − row_lb) * n) + (j − col_lb)) * element_size

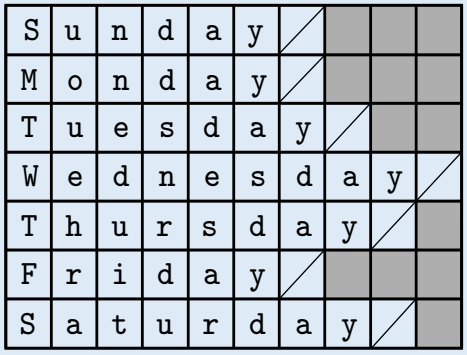
location(a[i, j]) = address of a[row_lb, col_lb]
− (((row_lb * n) + col_lb) * element_size)
+ (((i * n) + j) * element_size)

Figure 6.5

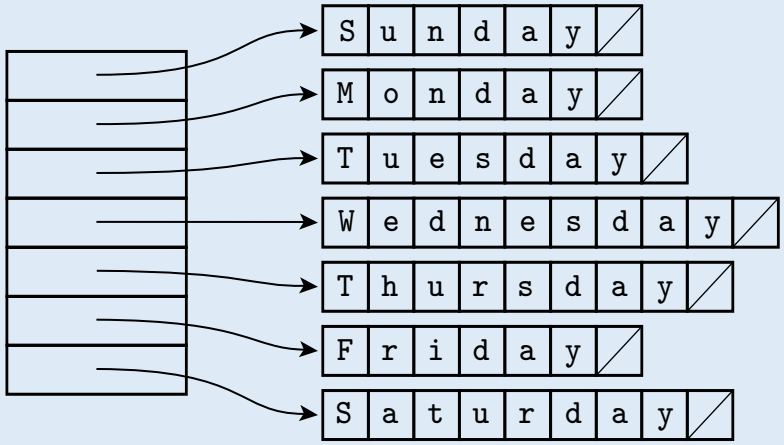
The location of the [i,j] element in a matrix



```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

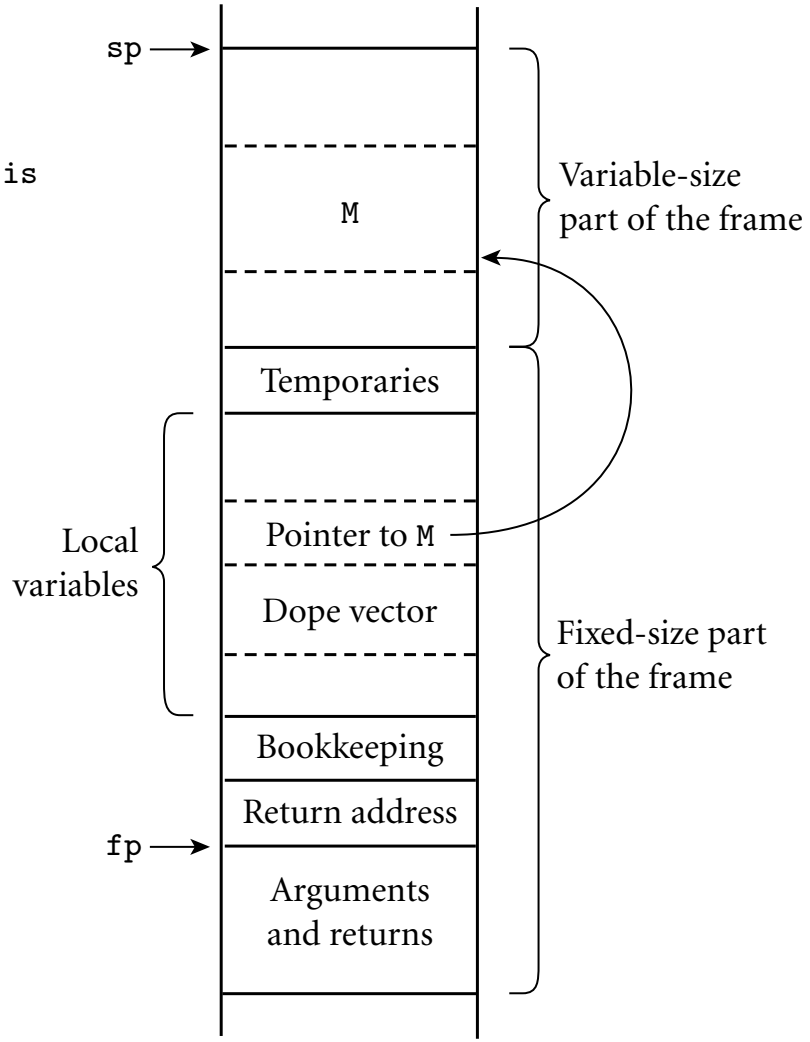



```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```



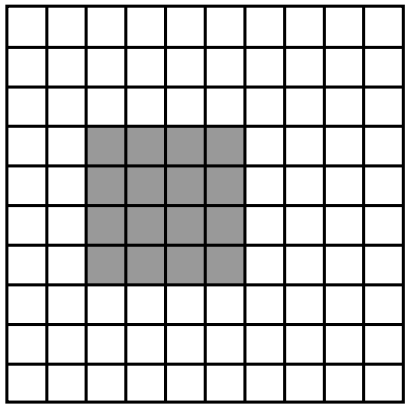
```
-- Ada:
procedure foo(size : integer) is
M : array (1..size, 1..size)
        of long_float;
...
begin
    ...
end foo;
```

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```

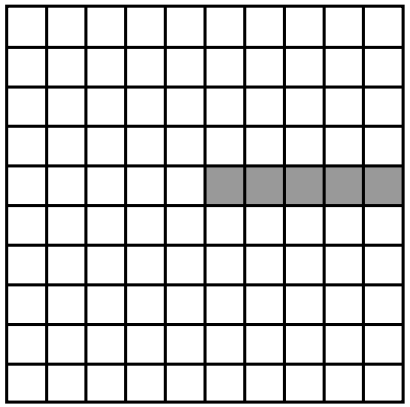


Array *slices* ("secciones" de arrays) en Fortran 90:

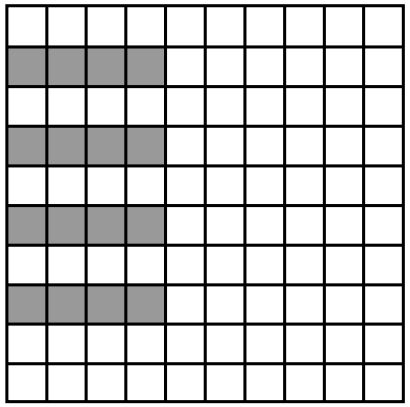
```
real, dimension (10,10) :: matrix
```



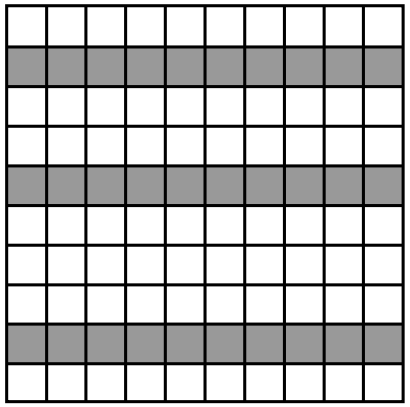
`matrix(3:6, 4:7)`



`matrix(6:, 5)`



`matrix(:4, 2:8:2)`



`matrix(:, (/2, 5, 9/))`

Un tipo de datos **recursivo** es uno cuyos objetos pueden contener referencias a otros objetos del mismo tipo

La mayoría son *records* (o `structs`):

- deben contener algo además de las referencias
- usados para construir estructuras de datos ligadas

En lenguajes que usan el modelo de variables por referencia (toda variable y todo campo de un record es una referencia), un record de tipo *foo* puede incluir fácilmente una referencia a otro record de tipo *foo*

En lenguajes que usan el modelo de variables por valor, los tipos recursivos requieren la noción de un *puntero*

Un tipo de datos **recursivo** es uno cuyos objetos pueden contener referencias a otros objetos del mismo tipo

La mayoría son *records* (o `structs`):

- deben contener algo además de las referencias
- usados para construir estructuras de datos ligadas

En lenguajes que usan el modelo de variables por referencia (toda variable y todo campo de un record es una referencia), un record de tipo *foo* puede incluir fácilmente una referencia a otro record de tipo *foo*

ML, Lisp

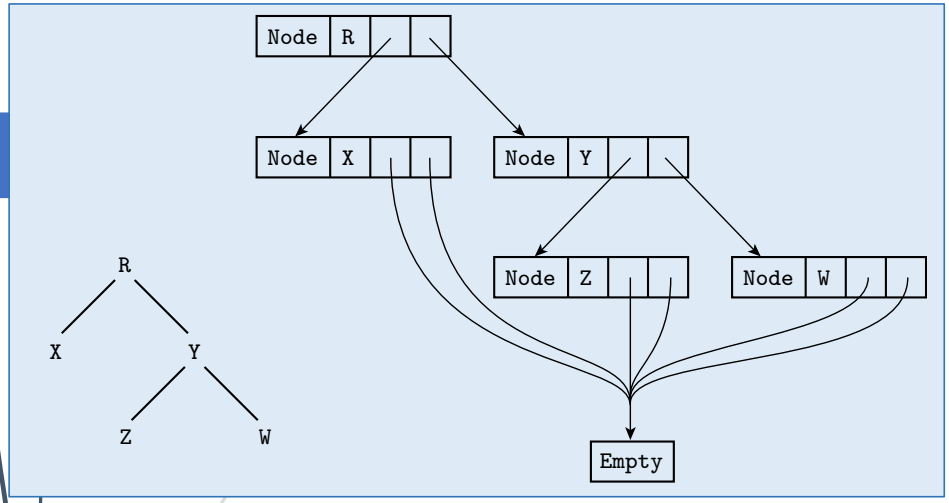
En lenguajes que usan el modelo de variables por valor, los tipos recursivos requieren la noción de un *puntero*

Ada, C

Un tipo de datos **referencia** es similar a un puntero, con una diferencia:

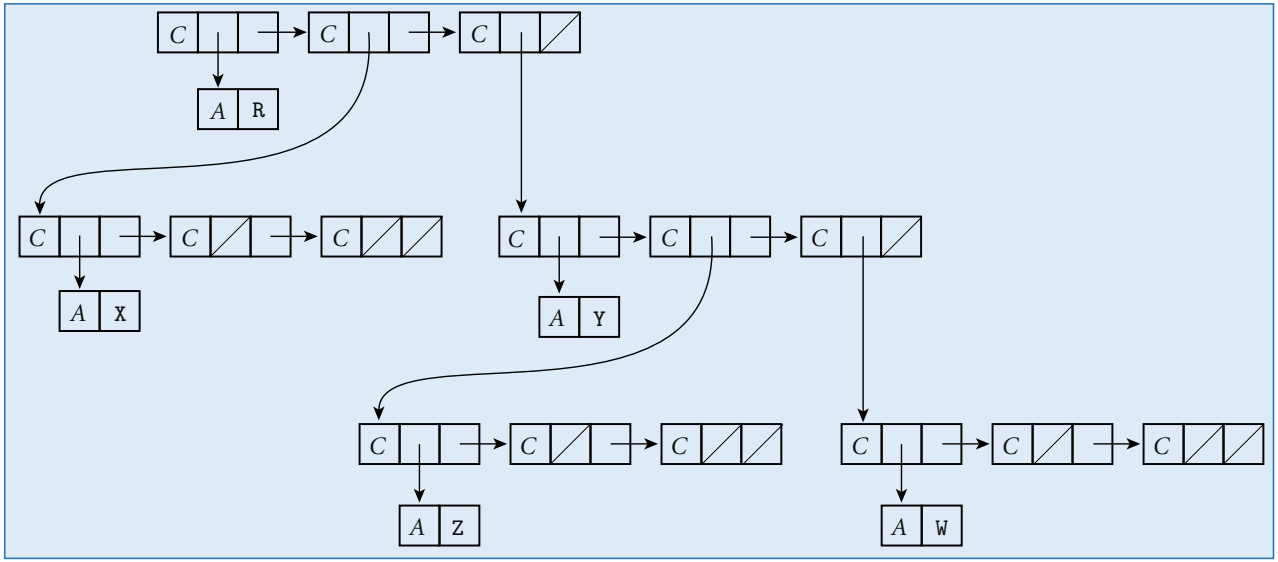
Un puntero se refiere a una dirección de memoria

… una referencia se refiere a un objeto o a un valor en memoria



Implementación de un árbol en ML:

```
type chr_tree = Empty
    | Node of char * chr_tree * chr_tree;;
```



Implementación del árbol en Lisp

Un tipo de datos **puntero** es uno en que las variables tienen un rango de valores consistente en *direcciones de memoria*

+ el valor especial **nil**

Dos usos:

- la capacidad del *direccionamiento indirecto* de los lenguajes de assembly
- manejo de almacenamiento dinámico en el *heap* —variables **heap-dynamic**

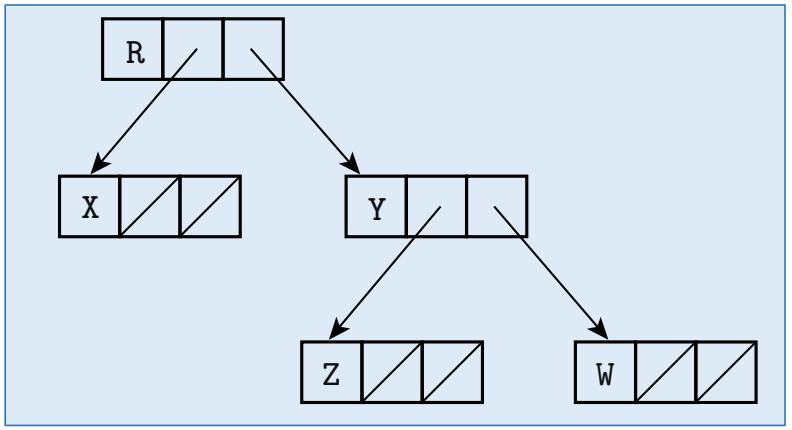
Las variables no tienen identificadores —son **anónimas**
… pueden ser referenciadas sólo por variables de tipo puntero

```
type chr_tree;
type chr_tree_ptr is access chr_tree;
type chr_tree is record
    left, right : chr_tree_ptr;
    val : character;
end record;
```

Declaración del árbol anterior en Ada

```
struct chr_tree {
    struct chr_tree *left, *right;
    char val;
};
```

Declaración del árbol anterior en C



Implementación típica de un árbol en un lenguaje con punteros explícitos

Preguntas de diseño:

- ¿Cuáles son el alcance (o ámbito) y el tiempo de vida de una variable puntero?
- ¿Cuál es el tiempo de vida de una variable *heap-dynamic*?
- ¿Hay restricciones para el tipo de valores a los que los punteros pueden apuntar?
- ¿Usamos los punteros para manejo de almacenamiento dinámico, direccionamiento indirecto, o ambos?

En la mayoría de los casos:

Los punteros (y las referencias) son valores individuales —direcciones de memoria— almacenados en celdas de memoria

Se tiende a equipararlos con direcciones de memoria

… pero no son lo mismo:

- un puntero es una referencia a un objeto —concepto de alto nivel
- una dirección de memoria es la ubicación de una palabra en la memoria —concepto de bajo nivel

Dos operaciones fundamentales:

- asignación —se asigna una dirección útil al valor de una variable (de tipo) puntero
- desreferenciación

Dos operaciones fundamentales:

- **asignación** —se asigna una dirección útil al valor de una variable (de tipo) puntero
- desreferenciación

Si las variables puntero se usan sólo para manejar almacenamiento dinámico:

- el mecanismo de asignación de memoria sirve para inicializar la variable

Un operador (`new`) o un sub-programa *built-in* (`malloc`)

Si los punteros se usan para direccionamiento indirecto a variables que no son *heap dynamic*:

- se necesita un mecanismo para obtener primero la dirección de una variable, que luego se asigna a la variable puntero

Un operador (`&`) o un subprograma *built-in*

¿Cómo se interpreta una variable puntero que aparece en una expresión?

- una referencia al contenido de la celda de memoria a la que está vinculada —una dirección
- una referencia al valor en la celda de memoria apuntada por la celda de memoria a la cual la variable puntero está vinculada —una referencia indirecta

¿Cómo se interpreta una variable puntero que aparece en una expresión?

- una referencia al contenido de la celda de memoria a la que está vinculada —una dirección
- una referencia al valor en la celda de memoria apuntada por la celda de memoria a la cual la variable puntero está vinculada —una referencia indirecta

Una referencia normal a un puntero (y exactamente como es interpreta-da una variable no puntero)

Desreferenciación del puntero:

- sigue una referencia a través de un nivel de indirección
- segunda operación fundamental en punteros

Dos operaciones fundamentales:

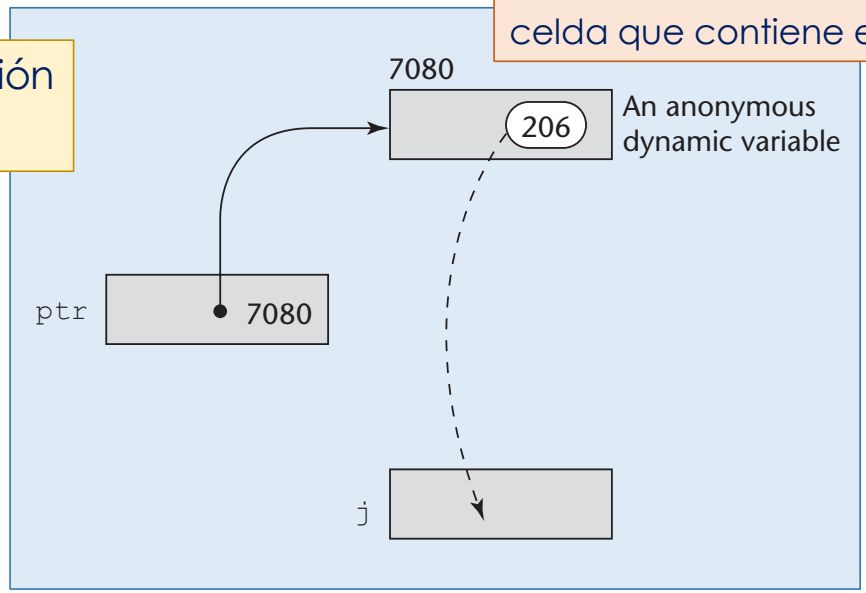
- asignación —se asigna una dirección útil al valor de una variable (de tipo) puntero
- **desreferenciación** —sigue una referencia a través de un nivel de indirección

`7080` : dirección de memoria de la celda que contiene el valor `206`

Efecto de la asignación

```
j = *ptr
```

`*` : operador unitario prefijo para desreferenciación explícita



Punteros (o referencias) ***dangling***:

El puntero contiene la dirección de una variable *heap dynamic* que ha sido *deallocated*

Variables *heap-dynamic* perdidas:

Variable *heap-dynamic* con espacio de memoria asignado, a la cual el programa ya no tiene acceso —***garbage***

Punteros (o referencias) ***dangling***:

El puntero contiene la dirección de una variable *heap dynamic* que ha sido *deallocated*

Variables *heap-dynamic* perdidas:

Variable *heap-dynamic* con espacio de memoria asignado, a la cual el programa ya no tiene acceso —***garbage***

1. Se crea una variable *heap-dynamic* y se hace que el puntero `p1` apunte a ella

2. Al puntero `p2` se le asigna el valor de `p1`

3. La variable apuntada por `p1` es *deallocated* explícitamente, pero `p2` no es modificado; `p2` es un puntero *dangling*

Punteros (o referencias) ***dangling***:

El puntero contiene la dirección de una variable *heap dynamic* que ha sido *deallocated*

Variables *heap-dynamic* perdidas:

Variable *heap-dynamic* con espacio de memoria asignado, a la cual el programa ya no tiene acceso —***garbage***

La variable ya no es útil para su propósito original

… y su espacio de memoria no puede ser reasignado para otro uso en el programa

1. Se crea una variable *heap-dynamic* y se hace que el puntero `p1` apunte a ella
2. Al puntero `p2` se le asigna el valor de `p1`
3. La variable apuntada por `p1` es *deallocated* explícitamente, pero `p2` no es modificado; `p2` es un puntero *dangling*

1. Se crea una variable *heap-dynamic* y se hace que el puntero `p1` apunte a ella
2. Se hace que `p1` apunte a otra variable *heap-dynamic*, de modo que la primera variable se vuelve inaccesible

Punteros (o referencias) ***dangling***:

- la localidad apuntada puede haber sido reasignada a una nueva variable *heap dynamic*
- si esta no es del mismo tipo que la antigua, entonces los chequeos de tipos de datos cuando se usa el puntero son inválidos
- si el puntero es usado para modificar el valor de la variable, entonces el valor de la nueva variable será destruido
- la localidad podría estar siendo usada por el sistema de gestión de memoria como un puntero en una cadena de bloques de memoria disponibles ⟶ un cambio en su valor haría que el sistema falle

Posibles soluciones al problema de los punteros *dangling*

Lápidas:

- Toda variable *heap-dynamic* incluye una celda especial —una *lápida*— que es un puntero a la variable

… la variable puntero apunta a la lápida, no a la variable *heap-dynamic*

Cuando una variable es *deallocated*, su lápida permanece, pero se pone en *nil* ⟶ un puntero nunca apuntará a una variable *deallocated*

… y cualquier referencia a un puntero que apunta a una lápida *nil* es identificada como un error

Costosas en tiempo —cualquier acceso a una variable requiere un nivel adicional de indirección— y espacio -–la memoria ocupada por las lápidas permanece ocupada aún cuando las variables ya no existan más

Posibles soluciones al problema de los punteros *dangling*

Candados y llaves:

- Los punteros son pares ordenados (*llave*, *dirección*), en que *llave* es un valor entero

… y las variables *heap-dynamic* tienen una celda adicional, *candado*, que almacena un valor entero

… cuando se asigna memoria a una variable *heap-dynamic*, se crea un valor entero que se almacena tanto en el *candado* de la variable como en la *llave* del puntero

Todo acceso a un puntero desreferenciado compara el valor de la llave del puntero con el valor del candado de la variable → sólo si coinciden, el acceso es legal

Posibles soluciones al problema de los punteros *dangling*

Deallocation de variables *heap-dynamic* no depende del programador:

- Si el programa no puede *deallocate* variables, entonces no habrá punteros *dangling*

El sistema de *run-time* debe *deallocate* las variables implícitamente cuando ya no son más útiles → *garbage collection* automática

Posibles soluciones al problema de los punteros *dangling*

Deallocation de variables *heap-dynamic* no depende del programador:

- Si el programa no puede *deallocate* variables, entonces no habrá punteros *dangling*

El sistema de *run-time* debe *deallocate* las variables implícitamente cuando ya no son más útiles → *garbage collection* automática

Conteo de referencias

Marcar y barrer

Conteo de referencias:

¿Cuándo un objeto ya no es más útil? Cuando no hay punteros que apunten al objeto → se asocia un *contador de referencias* a cada objeto, que cuenta el número de punteros que apuntan al objeto

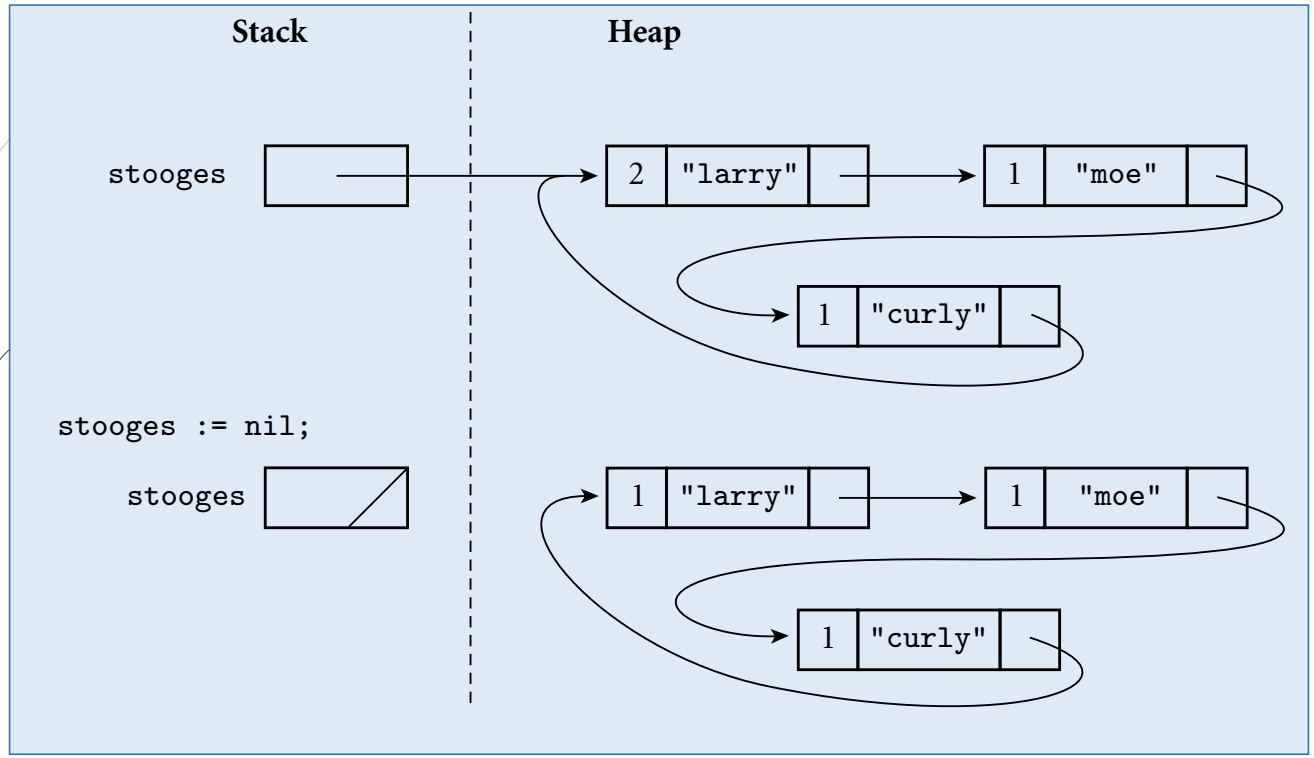
… cuando el objeto es creado, el contador se pone en 1

… cuando un puntero es asignado a otro, se decrementa el contador del objeto hasta ese momento correspondiente al lado izquierdo de la asignación, y se incrementa el contador del objeto correspondiente al lado derecho

… cuando se retorna de una subrutina, se debe decrementar el contador de cualquier objeto apuntado por un puntero local que va a ser destruido

… cuando un contador llega a cero, el objeto correspondiente es recoletado, y, **recursivamente** …

Problema con el conteo de referencias



Marcar y barrer:

¿Cuándo un objeto es útil? Cuando se puede llegar a él siguiendo una cadena de punteros válidos partiendo de algo que tenga nombre, es decir, algo fuera del heap

1. El recolector de basura recorre el heap, marcando tentativamente cada bloque de memoria como "inútil"

Marcar

2. Empezando desde todos los punteros fuera del heap, el recolector explora recursivamente todos los datos ligados, marcando cada bloque que va encontrando como "útil"

3. El recolector vuelve a recorrer el heap, ahora moviendo cada bloque aún marcado como "inútil" a la free list

Barrer

Marcar y barrer:

¿Cuándo un objeto es útil? Cuando se puede llegar a él siguiendo una cadena de punteros válidos partiendo de algo que tenga nombre, es decir, algo fuera del heap

1. El recolector de basura recorre el heap, marcando tentativamente cada bloque de memoria como "inútil"

2. Empezando desde todos los punteros fuera del heap, el recolector explora recursivamente todos los datos ligados, marcando cada bloque que va encontrando como "útil"

3. El recolector vuelve a recorrer el heap, ahora moviendo cada bloque aún marcado como "inútil" a la free list

El recolector necesita saber sónde empieza y termina cada bloque

El recolector debe poder encontrar los punteros contenidos dentro de cada bloque

El recolector necesita saber sónde empieza y termina cada bloque

Marcar y barrer:

¿Cuándo un objeto es útil? Cuando se puede llegar a él siguiendo una cadena de punteros válidos partiendo de algo que tenga nombre, es decir, algo fuera del heap

1. El recolector de basura recorre el heap, marcando tentativamente cada bloque de memoria como "inútil"

2. Empezando desde todos los punteros fuera del heap, el recolector explora recursivamente todos los datos ligados, marcando cada bloque que va encontrando como "útil"

3. El recolector vuelve a recorrer el heap, ahora moviendo cada bloque aún marcado como "inútil" a la free list

Colocar un puntero a un descriptor de tipo cerca del inicio de cada bloque

El recolector necesita saber sónde empieza y termina cada bloque

El recolector debe poder encontrar los punteros contenidos dentro de cada bloque

El recolector necesita saber sónde empieza y termina cada bloque

`curr` apunta al bloque que está siendo revisado

El bloque anterior es apuntado por `prev`

Las celdas grises indican punteros que han sido invertidos
→ ¿en la práctica?

