



UNIVERSIDAD TÉCNICA
FEDERICO SANTA MARÍA

DEPARTAMENTO
DE INFORMÁTICA

LENGUAJES DE PROGRAMACIÓN

Equipo de Profesores:

Jorge Díaz Matte - José Luis Martí Lara

Wladimir Ormazabal Orellana

Unidad 3

Tipos de Datos

3.1 Sistemas de Tipos de Datos

3.2 Tipos de Datos Ordinales o Simples

3.3 Arreglos y Strings

3.4 Registros y estructuras relacionadas

3.5 Colecciones

3.6 Punteros y Referencias

3.1 Sistemas de Tipos de Datos

Sistema de Tipos de Datos

DEFINICIÓN: Define la manera en que un lenguaje clasifica en tipos los valores y expresiones, y cómo interactúan estos tipos; además, permite la definición de nuevos tipos y determinar si éstos son correctamente usados (mediante ciertas reglas).

- Ventajas: permite verificar el uso correcto del lenguaje y detectar errores de tipos, tanto en tiempo de compilación o ejecución. Ayuda a modularizar (ej.: bibliotecas y paquetes).
- Desventajas: sistema muy estricto puede rechazar programas correctos. Para mitigar esto, se introducen “lagunas” y conversiones explícitas no verificadas.

Tipos de Datos

DEFINICIÓN: Define conjunto de valores de datos y conjunto de operaciones predefinidas sobre los objetos de datos.

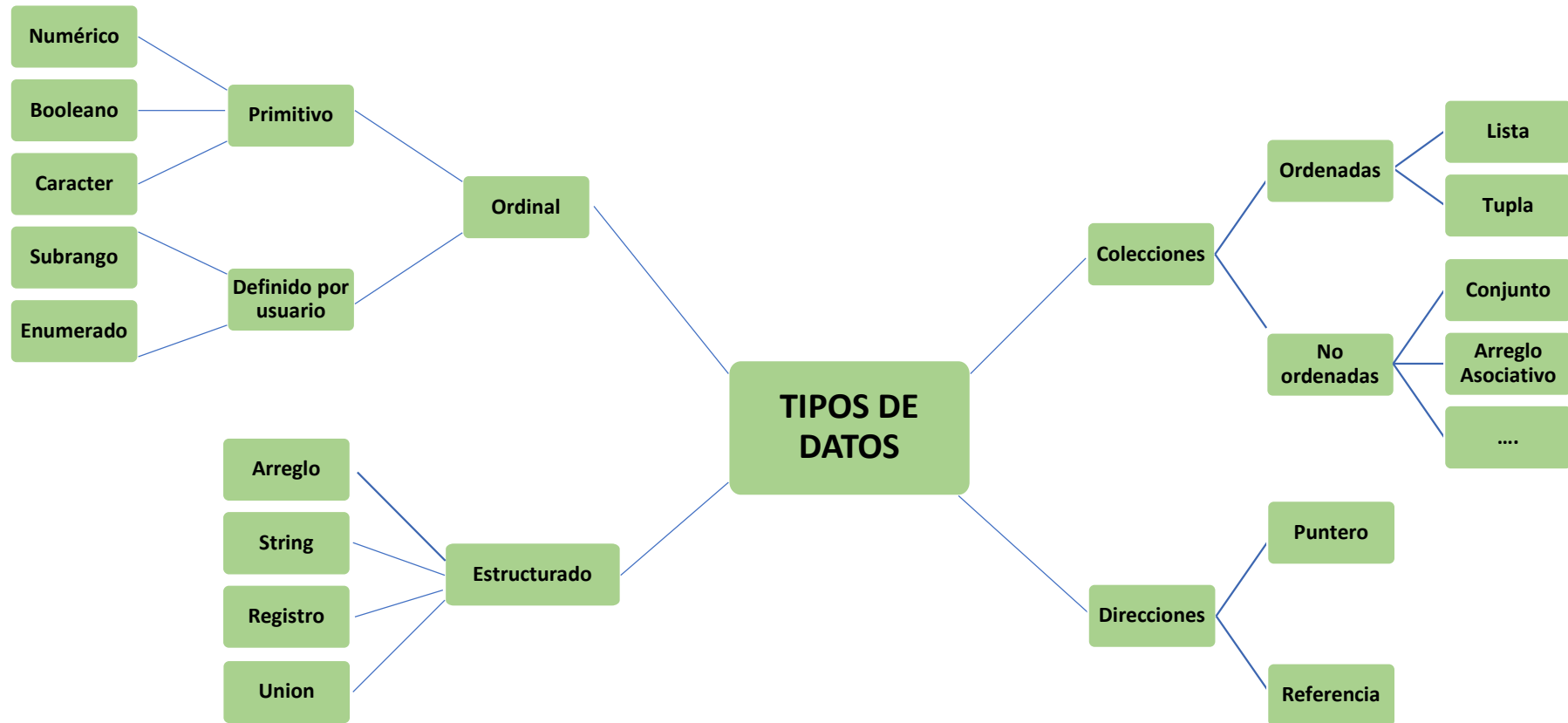
- Disponibilidad de gran número de tipos de datos ayuda a calzar los objetos de datos con un determinado problema.
- Los lenguajes de programación proveen un conjunto de tipos primitivos y mecanismos o constructos que permiten definir nuevos tipos orientados a resolver un problema específico.

Tipos de Datos

Ejemplos:

- Primitivos (ej.: enteros, punto flotante y caracteres)
- Estructurados (ej.: arreglos y registros)
- Definidos por usuario (más legible y seguro)
- Abstractos (modulariza datos y código)

Taxonomía de Tipos de Datos



Conceptos básicos

DEFINICIONES:

- Chequeo o Verificación de tipo: asegurar que los operandos de un operador son de tipos compatibles.
- Tipo compatible: es un tipo legal, o que mediante reglas del lenguaje puede ser convertido en uno legal.
- Conversión de tipos: se denomina coerción a la conversión automática y casting a la conversión definida por el programador.
- Error de tipo: la aplicación de un operador a un tipo inapropiado.

Equivalencia de Tipos

DEFINICIÓN: Se dice que dos tipos son equivalentes, si un operando de un tipo en una expresión puede ser sustituido por otro tipo sin necesidad de coerción.

- Reglas de compatibilidad de tipos del lenguaje determinan los operandos aceptables para cada operador.
- Se dice compatible, porque en tiempo de compilación o ejecución los tipos de los operandos pueden ser implícitamente convertidos (coerción) para ser aceptable por el operador.
- Equivalencia es una forma más estricta de compatibilidad, pues no requiere coerción.
- La compatibilidad en escalares son simples, sin embargo para tipos estructurados las reglas son más complejas.

Compatibilidad de Tipos en Estructuras

Equivalencia de tipo nominal

- Las variables están en la misma declaración o en declaraciones que usan el mismo nombre de tipo.
- Fácil implementación, pero muy restrictivo.

Equivalencia de tipo estructural

- Si los tipos tienen una estructura idéntica, pero pueden tener diferente nombre.
- Más flexible, pero de difícil implementación.

Compatibilidad de Tipos en Estructuras

Ejemplos en Pascal:

TYPE

mes_t = 1..12;

VAR

indice : mes_t;
contador: integer;

=====

TYPE

celsius = real;
fahrenheit = real;

{otro ejemplo}

TYPE

tipo1 = ARRAY [1..10] OF integer;
tipo2 = ARRAY [1..10] OF integer; *{tipo1 <> tipo2}*
tipo3 = tipo2; *{equivalencia declarativa}*

Compatibilidad de Tipos en Estructuras

Ejemplo en C:

```
struct s1 {int c1; real c2 };  
struct s2 {int c1; real c2 };
```

```
s1 x;  
s2 y = x; /* error de compatibilidad */
```

=====

```
typedef char* pchar; /* define nombre, no nuevo tipo */  
pchar    p1, p2;
```

```
char* p3 = p1;
```

Taxonomía de los Tipos de Datos: Tipificación

DEFINICIÓN: se pueden adoptar diferentes decisiones de diseño para el sistema de tipos de un lenguaje, que impactan en la eficiencia de ejecución, la rapidez de programación, la confiabilidad y seguridad, entre otros. La tipificación de los datos se puede clasificar en:

- Estática o dinámica,
- Explícita o implícita,
- Fuerte o débil.

Estos criterios de clasificación no son ortogonales entre sí y normalmente están fuertemente acoplados.

Tipificación Estática vs. Dinámica

DEFINICIÓN: los sistemas de tipos de datos se pueden clasificar según el momento en que se realiza el chequeo de tipos:

- Tipificación estática: se determina el tipo de todas las variables y expresiones antes de la ejecución, y luego permanece fijo. Los tipos pueden ser determinados explícitamente (declarados) o inferidos mediante reglas (ej.: C). Aplica sólo si todos los tipos son ligados estáticamente.
- Tipificación dinámica: se determina el tipo durante la ejecución, y normalmente éste es inferido. Tipos asociados a una misma variable pueden variar según valor asignado. Errores sólo pueden ser detectados durante la ejecución del programa (ej.: Scheme y Python).

Tipificación Explícita vs. Implícita

DEFINICIÓN: dependiendo del grado de exigencia para definir los tipos asociados a todos los objetos de datos, es posible clasificar los sistemas de tipos como:

- Tipificación explícita: todos los tipos de datos asociados a variables y otros elementos del programa deben ser necesariamente declarados y estar bien definidos. Es muy deseable para una tipificación estática.
- Tipificación implícita: tipos de datos no se declaran y se infieren a través de reglas tales como nombre de variables o tipos de datos en expresiones.

Tipificación Fuerte vs. Débil (1)

DEFINICIÓN: sistemas de tipos se pueden clasificar según lo grado de exigencia impuesto en la verificación para detectar potenciales errores de tipos.

- Tipificación fuerte: siempre detecta errores de tipo (estática o dinámicamente). Establece restricciones fuertes sobre cómo operaciones aceptan valores de diferentes tipos de datos, e impiden la ejecución si tipos son erróneos. Promueve tipificación estática y explícita.
- Tipificación débil: se realizan implícitamente conversiones que permiten relajar restricciones, generando más flexibilidad y agilidad, pero podrían generarse errores no detectados.

Tipificación Fuerte vs. Débil (2): la primera con...

Tipificación estática:

- Eficiencia de ejecución: permite al compilador asignar memoria y generar código que manipule los datos eficientemente.
- Seguridad y confiabilidad: permite detectar mayor número de errores, y por lo tanto reducir errores de ejecución. También permite verificar mejor compatibilidad de interfaces en la integración de módulos/componentes de programa.

Tipificación explícita:

- Legibilidad: se mejora, al documentar bien los tipos usados en el programa.
- Ambigüedad: permite eliminar ambigüedades que podrían surgir en el proceso de traducción.

Tipificación Fuerte vs. Débil (3):

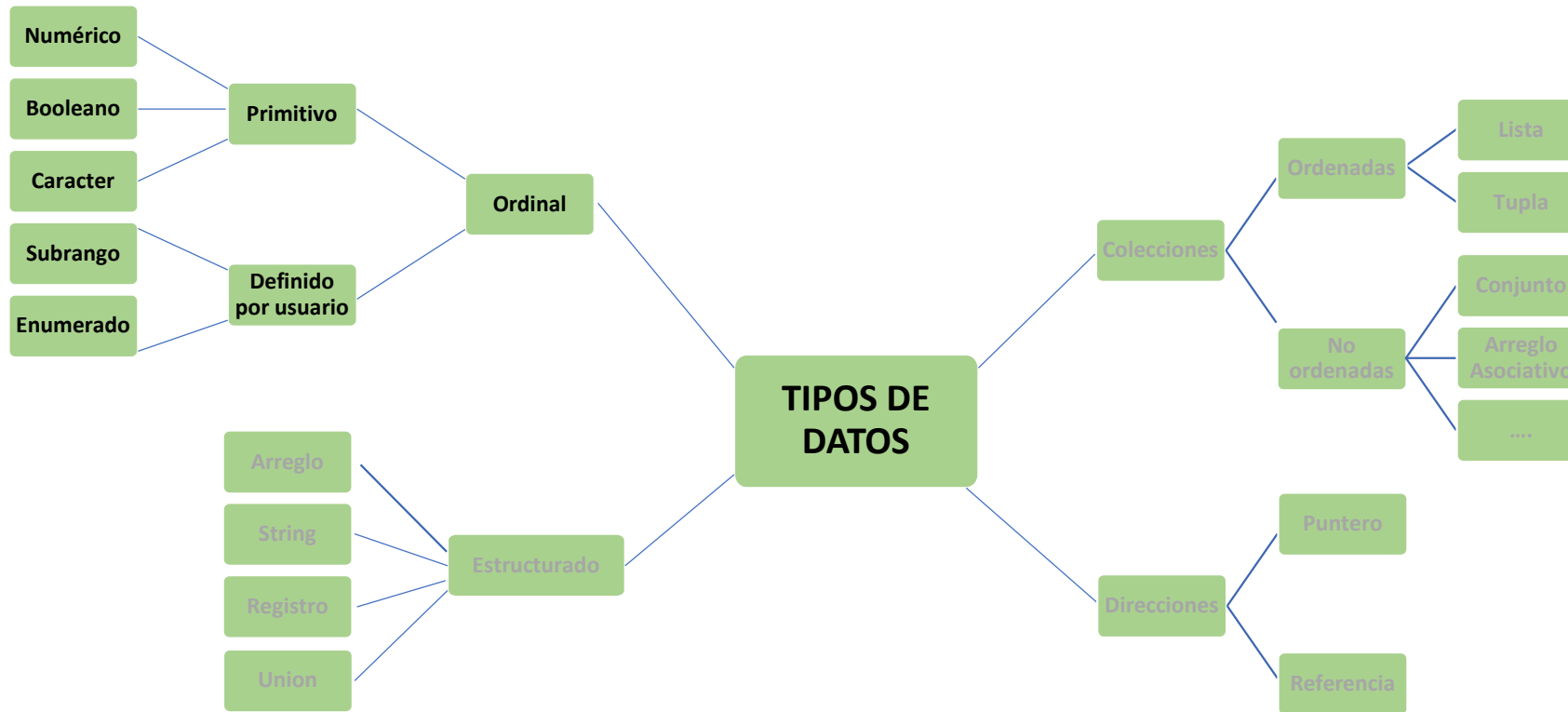
Críticas a la tipificación fuerte:

- Fuerza a los programadores a una disciplina rígida de programación, reduciendo facilidad de escritura de código y diseñar buenos programas.
- Aumenta el esfuerzo de programación al tener que cumplir con las exigencias del lenguaje, entregando mayor información de tipos.
- Hace más difícil el aprendizaje y la legibilidad debido a la mayor complejidad del lenguaje.
- Lenguajes modernos permiten una mayor flexibilidad en la tipificación estática y combinan -en la medida que sea seguro-, algún tipo de tipificación dinámica e implícita.

Resumen

Un sistema de tipos de datos consiste en:

1. Conjunto de tipos primitivos y operadores predefinidos.
2. Mecanismo para definir y construir nuevos tipos y posibles operadores (por el usuario, estructurados o abstractos).
3. Conjunto de reglas para establecer equivalencia, compatibilidad e inferencia de tipos, con un mecanismo de conversión de tipos.
4. Un sistema de chequeo de tipos que puede ser estático o dinámico, o una combinación de ambos.



3.2 Tipos de Datos Ordinales o Simples

Tipo Ordinal o simple

DEFINICIÓN: Un tipo ordinal es aquel que puede ser asociado a un número natural (conjunto ordenado). Incluye tipos primitivos y tipos definidos por el usuario.

- **Tipos primitivos**: no están basados en otro tipo del lenguaje.
 - Numérico (entero, decimal, punto fijo, punto flotante)
 - Caracter
 - Booleano
- **Tipos definidos por el usuario**: típicamente basados en tipos primitivos.
 - Enumerado
 - Subrango

Tipos Primitivos (1)

DEFINICIÓN: Corresponden a tipos de datos que no están definidos en términos de otros tipos de datos. Muchos de estos tipos se soportan directamente por el hardware.

- Numérico:
 - Entero (ej.: C y Java permite diferentes tipos de enteros: *signed*, *unsigned*, *short*, *long*)
 - Punto flotante (ej.: C y Java permite *float* y *double*)
 - Decimal (típicamente 4 bits por dígito decimal)
- Booleano:
 - Mejora legibilidad. Típicamente ocupa un byte (ej.: Java)

Tipos Primitivos (2)

- Caracter:
 - Tamaño de un byte; típicamente código ASCII (ej.: ISO 8859)
 - Tamaño variable (UTF-8 hasta 6 Bytes)
 - Unicode con 16b (UTF-16) usado en Java, o 32b (UTF-32)

Tipos Primitivos: Representación de Números (1)

CARACTERÍSTICAS:

- Conjunto finito y una aproximación al concepto matemático.
- Rango de representación y precisión depende del largo del registro.
- Soporte directo del hardware para varios tipos básicos.

Tipos Primitivos: Representación de Números (2)

TIPOS:

- **Números enteros** (ej.: representación C-2, C-1).
- **Números de punto flotante**: representa una aproximación a números reales (ej.: estándar IEEE 754).
- **Decimal**: Típicamente para aplicaciones de negocios (ej.: COBOL usa 4 bits en código BCD). Es más preciso para manipular números en base 10, pero ocupa más memoria.
- **Complejo**. Algunos lenguajes lo soportan y lo representan como dos números de punto flotante (ej.: Python y Scheme).

Tipos Primitivos: Representación de Números (3)

Estándar IEEE 754: representa un número de punto flotante en precisión simple (32b) y precisión doble (64b).

- Precisión simple: 1b de signo (s), 8b de exponente (e) y 23b de mantisa (m).
 - Representación: $(-1)^s \times 1.m \times 2^{e-127}$
 - Rango aproximado de $\pm 10^{38}$ con precisión de 10^{-38} .
- Precisión doble: 1b de signo (s), 11b de exponente (e) y 52b de mantisa (m).
 - Representación: $(-1)^s \times 1.m \times 2^{e-1024}$
 - Rango aproximado de $\pm 10^{308}$ con precisión de 10^{-308} .



Tipos Primitivos: Representación del Tipo Caracter

DEFINICIÓN: Usado para facilitar la comunicación de datos. Un sistema de caracteres define un conjunto de caracteres y un sistema de codificación para cada elemento, usando patrones de bits o bytes (códigos).

- Estándares más conocidos:
 - **ASCII:** 7 bits con subconjunto de control (32) e imprimibles (96).
 - **EBCDIC:** Códigos de 8 bits definido por IBM.
 - **UTF-8:** Códigos de largo variable de bytes, compatible con ASCII y puede representar cualquier caracter de Unicode.
 - **UTF-16:** Largo de 2B o 4B, superconjunto de UCS-2 (2B usada en Python y hasta Java 1.4)
- Lenguajes usan caracteres de escape (ej.: `\"`) para representar caracteres sin afectar la sintaxis del lenguaje.

Tipos Definidos por el Usuario (1): Subrango

DEFINICIÓN: Subsecuencia contigua de un tipo ordinal ya definido (ej.: 1..12).

- Características:
- Introducido por Pascal y usado en Modula-2 y ADA.
- Mejora lectura y fiabilidad.
- Se implementan en base a tipo entero.

- Ejemplo: Pascal

```
TYPE  
mayúscula = 'A'..'Z';  
indice = LUNES .. VIERNES;
```

Tipos Definidos por el Usuario (2): Enumerado

DEFINICIÓN: Tipo donde se enumeran (exhaustivamente) todos los posibles valores a través de constantes literales. Enumeración establece una relación de orden.

- Características:
- Existe relación de orden permite definir operadores relacionales, predecesor y sucesor.
- Mejoran facilidad de lectura y fiabilidad.
- Se implementan normalmente mapeando los constantes según su posición a un subrango de enteros.
- Normalmente no se usan en E/S.

Tipos Definidos por el Usuario (3): Enumerado

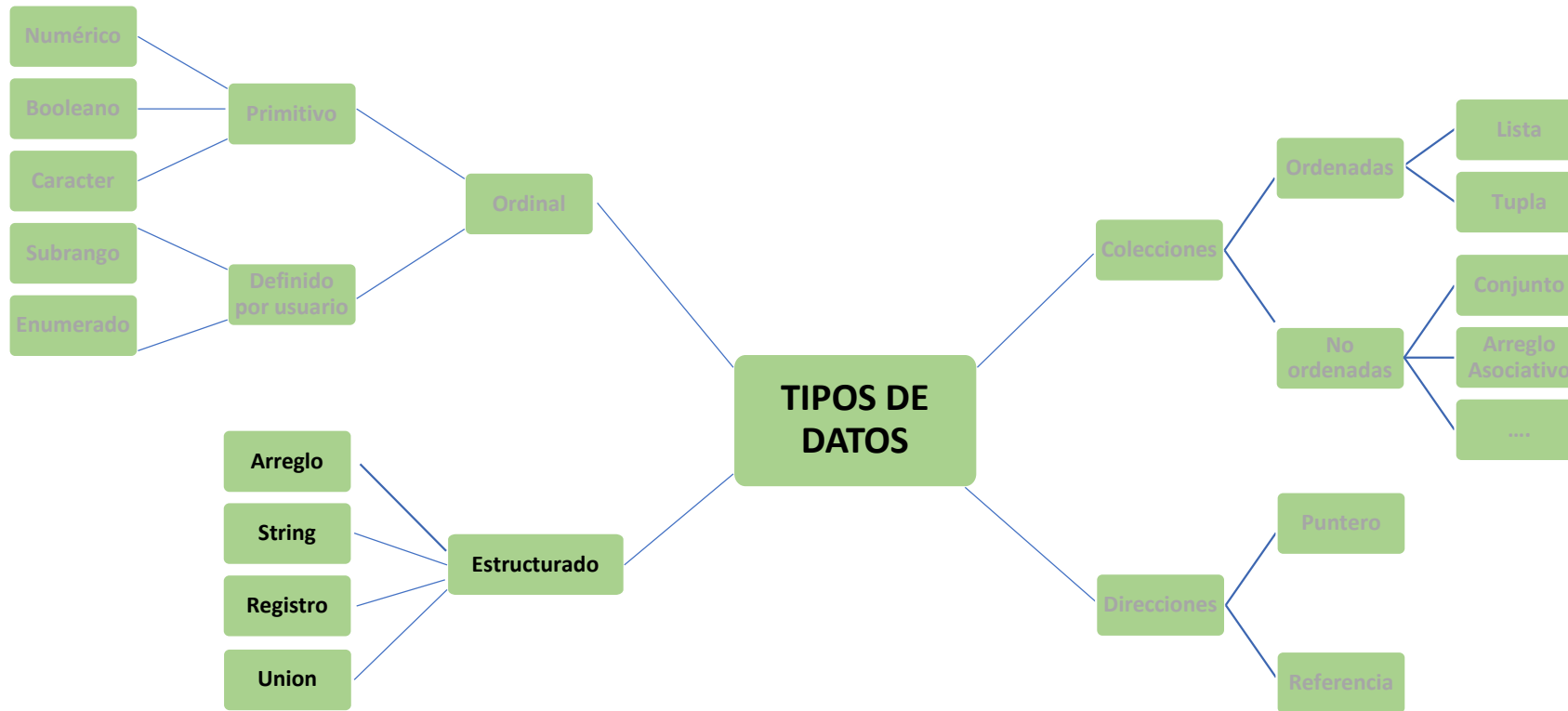
Ejemplos en C y C++

Sintaxis:

```
<enum-type>      ::= enum [<identifier>] { <enum-list> }  
<enum-list>      ::= <enumerador> | <enum-list> , <enumerador>  
<enumerador>     ::= <identificador> | <identificador> = <constant-exp>
```

Código:

```
enum color {rojo, amarillo, verde=20, azul};  
color col = rojo;  
color* cp = &col;  
  
if (*cp == azul) // ...
```



3.3 Arreglos y Strings

Tipo Arreglo

DEFINICIÓN: Es un tipo estructurado consistente en un conjunto homogéneo y ordenado de elementos que se identifican por su posición relativa mediante un índice.

- Existe un tipo asociado a los elementos y otro al índice.

Tipo Arreglo: índices

DEFINICIÓN: Definen un mapeo desde el conjunto de índices al conjunto de elementos del arreglo.

Sintaxis:

- FORTRAN y ADA: usan paréntesis.
- Pascal, C, C++, Modula-2 y Java: usan corchetes.

Tipo Arreglo: índices

Tipo de datos del índice:

- C, C++, Java y Perl: sólo enteros.
- ADA, PASCAL: enteros y enumerados.

Prueba de rango de índice:

- C, C++, Perl: sin prueba de rango.
- Ada, Pascal, Java, C#: con prueba de rango.

Tipo Arreglo: ligado (1)

Arreglo Estático: el rango de índice y la memoria son ligados antes de la ejecución (siendo ésta más eficiente).

- Ejemplo: único tipo en lenguajes antiguos, como Fortran77.

Tipo Arreglo: ligado (2)

Arreglo Dinámico Fijo de Stack: el rango de índice es ligado estáticamente, pero la memoria se asigna dinámicamente (uso más eficiente de la memoria).

- Ejemplo: arreglos definidos dentro de un procedimiento o función en Pascal y C (sin *static*).

Arreglo Dinámico de Stack: el rango de índice y la memoria son ligados dinámicamente, permaneciendo fijos durante tiempo de vida de la variable (más flexible que anterior).

- Ejemplo: ADA y algunas implementaciones de C y C++.

Tipo Arreglo: ligado (3)

Arreglo Fijo de Heap: similar a dinámico de *stack*, donde el tamaño puede variar, pero permanece fijo una vez asignada la memoria (la mayor flexibilidad).

- Ejemplo: C con *malloc()*.

Arreglo Dinámico de Heap: el tamaño puede variar durante su tiempo de vida (la mayor flexibilidad).

- Ejemplo: PERL y Python.

Tipo Arreglo: ligado (4)

- Ejemplo: Arreglo Dinámico de *Stack* en C:

```
void foo(int n) {  
    int a[n];  
  
    for (int i=0; i<n; i++) {  
        ...  
    }  
    ...  
}
```

- Ejemplo: Arreglos de *Heap*

C:

```
char *str, *s, *t;  
...  
str = malloc(strlen(s) + strlen(t)+1);  
strcat(strcpy(str, s), t);
```

PERL: @a = split (" ", \$text);

Tipo Arreglo: inicialización

- Fortran

```
INTEGER MES(12);  
DATA MES /31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31/
```

- ANSI C y C++

```
char *mensaje = "Hola mundo\n";  
char *dias[] = {"lu", "ma", "mi", "ju", "vi", "sa", "do"};
```

- Java

```
int[] unArreglo = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};  
String[] nombres = {"Pedro", "Maria", "Jose"};
```

- Pascal y Modula-2 no lo permiten

Tipo Arreglo: multidimensionales

- Fortran IV: primera versión de Fortran que permite declarar hasta 7 dimensiones.

- Pascal: Permite sólo dos dimensiones

```
TYPE  
      matriz = ARRAY [subindice, subindice] OF  
      real;
```

- C y C++:

```
real matriz [DIM1][DIM2];
```


Tipo Arreglo: operadores

- APL: provee varios operadores (soporte de vectores y matrices con operadores).
- ADA: permite la asignación.
- Pascal, C y Java: no tienen soporte especial (sólo selector con subíndice []).
- C++: permite definir una clase arreglo por el usuario y operadores tales como subíndice, asignación, inicialización, etc.
- Python: soporta arreglos mediante listas y provee varios operadores (ej.: pertenencia, concatenación, subrango).

Tipo Arreglo: implementación

- Un arreglo es una abstracción del lenguaje, y debe ser mapeado a la memoria como un arreglo unidimensional de celdas (arreglo de bytes).
- Ejemplo: dirección de lista[k]

```
dir(lista[0]) + (k)*tamaño  
dir(lista[bajo]) + (k-bajo)*tamaño
```

- Arreglos bidimensionales se almacenan como fila de columnas, o viceversa.

Tipo String (1)

DEFINICIÓN: corresponde a una secuencia de caracteres usado para procesamiento de texto y para E/S. Típicamente implementado en base a un arreglo de caracteres.

Aspectos de diseño:

- ¿Es un tipo propio del lenguaje o un arreglo de caracteres?
- ¿El largo es estático o dinámico (memoria)?

Tipo String (2)

Operaciones típicas:

- Asignación, copia y concatenación
- Largo y comparación
- Referencia a *substring*
- Reconocimiento de patrón

Tipo String (3)

Ejemplo: C

```
char str[20];

...
if (strcmp(str, "Hola")){
    ...
else {
    ...
}
```

Ejemplo: Java

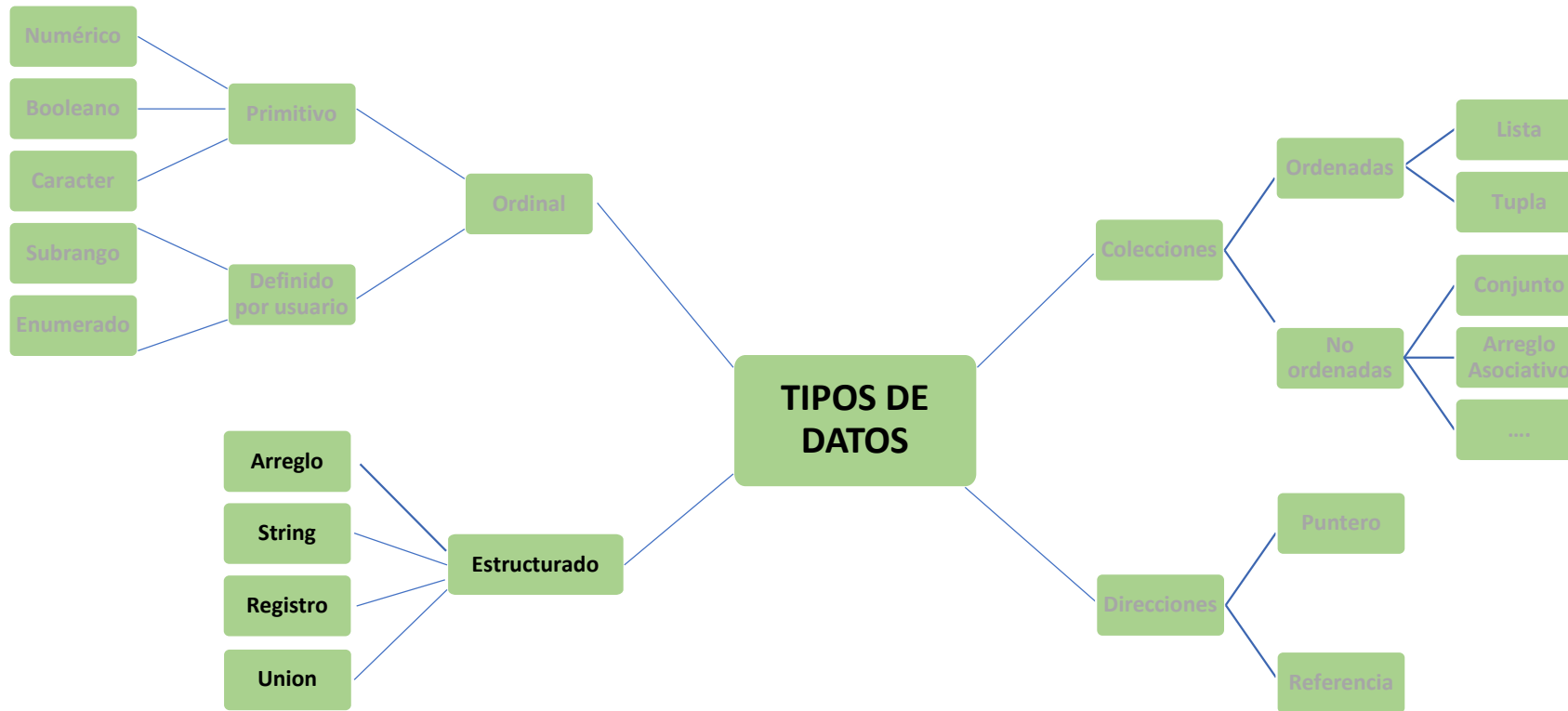
```
"Este es un String" // literal
String palindrome = "reconocer"; // referencia String
char[] ArregloHola = { 'h', 'o', 'l', 'a' }; // arreglo
String StringHola = new String(ArregloHola);
```

Tipo String (4): largo

Diseño de string considera:

- **Largo estático**: Fortran77, Pascal y Java.
- **Largo dinámico limitado**: C y C++ (se usa carácter especial de término).
- **Largo dinámico**: JavaScript, Python y Perl (es el más flexible, pero es más costoso de implementar y ejecutar)

ADA soporta los tres tipos.



3.4 Registros y Uniones

Tipo Registro

DEFINICIÓN: conjunto posiblemente heterogéneo de elementos de datos, donde cada elemento individual (denominado campo o miembro) es identificado con un nombre. Útil para procesamiento de datos en archivos.

Estilos:

- Cobol usa: <campo> OF <nombre_registro>
- Otros (Pascal, Ada, C, Java, ...): var.campo

Operaciones:

- Típicamente sólo asignación.

Tipo Registro: ejemplos

COBOL: el tipo registro fue introducido por COBOL, que forma parte de la DATA DIVISION del programa; usa números de nivel para definir una jerarquía estructurada del registro.

```
01 EMP-REC.  
  02 EMP-NAME.  
    05 FIRST PIC X(20).  
    05 MID   PIC X(10).  
    05 LAST  PIC X(20).  
  02 HOURLY-RATE PICTURE IS 99V99.
```

Tipo Registro: ejemplos

Pascal:

```
TYPE empleado_t = RECORD
    nombre : RECORD
        primer: PACKED ARRAY [1..10] OF char;
        paterno: PACKED ARRAY [1..10] OF char;
        materno: PACKED ARRAY [1..10] OF char;
    END {nombre};
    sueldo : real
END;

VAR empleado1 : empleado_t;
BEGIN
    ...
    empleado1.sueldo := 550000.00;
    ...
END.
```

Tipo Registro: ejemplos

C, C++:

```
typedef struct {  
    struct {  
        char primer[10];  
        char paterno[10];  
        char materno[10];  
    } nombre;  
    float sueldo;  
} empleado_t;
```

```
empleado_t empleado1;  
empleado1.sueldo = 550000.00;  
strcpy(empleado1.nombre.primero, "Juan");
```

Referencias Elípticas

- Algunos lenguajes sólo permiten referencias de calificación completa (C y C++).
- Otros permiten referencias elípticas, una forma más conveniente de programar (Pascal, Cobol y PL/I).
- Ejemplo: Pascal

```
empleado.nombre.primer := 'Juan';  
empleado.nombre.paterno := 'Perez';  
empleado.nombre.materno := 'Machuca';
```

```
WITH empleado.nombre DO  
  BEGIN  
    primer := 'Juan';  
    paterno := 'Perez';  
    materno := 'Machuca';  
  END
```

Tipo Union

DEFINICIÓN: tipo que permite almacenar diferentes tipos de datos en diferentes tiempos en una misma variable. Permite ahorrar memoria.

Alternativas de diseño:

- ¿Se debe declarar como parte de un registro?
- ¿Se permite prueba (dinámica) de tipo?
- ¿Requiere uso de un discriminador o marca de tipo?
- ¿Se les puede incrustar en un registro?

Lenguajes que lo soportan:

- Fortran, Ada, C y C++.

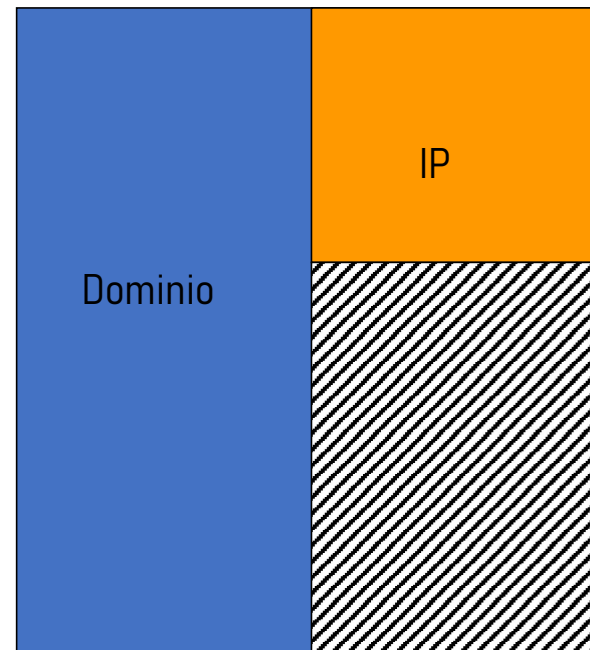
Tipo Union: comentarios

- Reserva espacio de memoria igual al mayor miembro definido.
- Todos los miembros comparten la memoria y comienzan desde la misma dirección.
- Su uso es en general poco seguro, lo que hace que muchos lenguajes no sean de tipificación fuerte.
- Java y C# no provee este tipo de estructura por ser inseguras.

Tipo Union: ejemplos

C y C++:

```
union dirección  
{  
    char dominio[20];  
    int IP[4];  
};
```

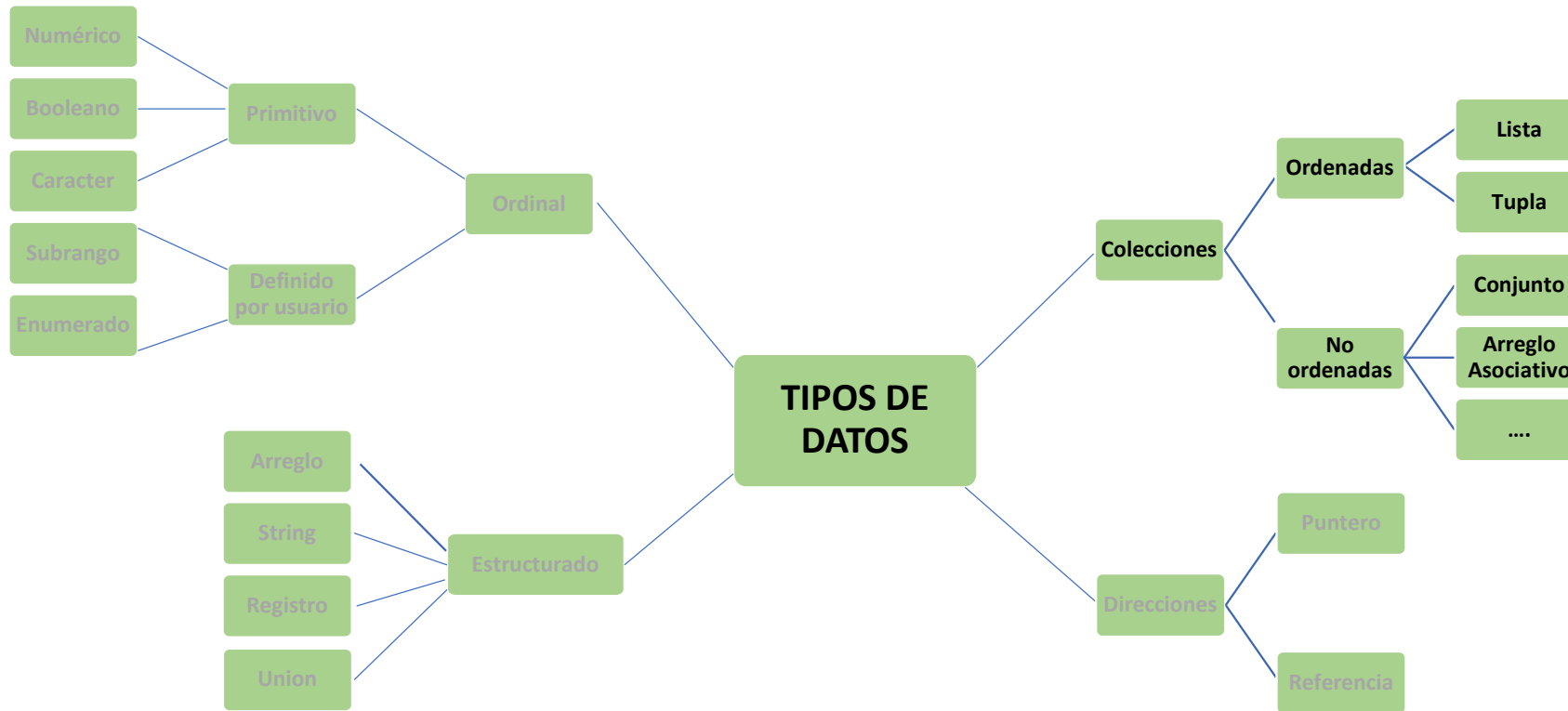


Tipo Union: ejemplos

```
typedef enum {CARTESIANO, POLAR} coordenada_t;
struct complejo_t {
    coordenada_t tipo_coord; // discriminadores
    union {
        struct{real rad,ang;} p;
        struct{real x,y;} c;
    } tag;
} u,v;

#define polar tag.p
#define cartesiano tag.c

if (u.tipo_coord == POLAR) {
    u.tag.p.ang = PI/2;    // más engorroso
    u.polar.rad = 34.7;    // mejor lectura
}
```

3.5 Colecciones

Colecciones

DEFINICIÓN: lenguajes modernos incluyen tipos de datos que agrupan conjunto de elementos (que pueden ser de diferentes tipos), con operadores para construirlos y acceder a sus elementos.

Tipos de colecciones:

- **Ordenadas**: los elementos tienen definida una relación de orden (lineal) en base a la posición:
 - Arreglos, vectores, listas y tuplas.
 - Stacks y colas.
- **No Ordenadas**: no existe un ordenamiento de los elementos.
 - Conjuntos.
 - Arreglos asociativos, tablas de *hash* o diccionarios.

Tipo Conjunto

DEFINICIÓN: permite almacenar un conjunto no ordenado de elementos de datos.

- Caso Pascal y Modula-2: define un tipo base (el tamaño es dependiente de la implementación y ésta se realiza mediante arreglo de bits); no permite manejar eficientemente conjuntos grandes.
- Caso Python: corresponde a una implementación más flexible, que no requiere definir tipo base pero que ocupa más memoria. Los elementos pueden ser de diferentes tipos.

Tipo Conjunto: ejemplos (1)

Pascal:

```
TYPE      character = SET OF char;
VAR       vocal, letras, no_vocal: character;
          c : char;

BEGIN
  vocal := ['A', 'E', 'I', 'O', 'U'] + ['a', 'e', 'i', 'o', 'u'];
  letras := ['A'..'Z'] + ['a'..'z'];
  no_vocal := letras - vocal;
  IF (c IN vocal) THEN
    ...
```

Operadores:

[...], +, *, -, [] (conjuntos)
>=, <= (relacionales)
IN (inclusión)

Tipo Conjunto: ejemplos (2)

Python:

```
>>> A = {1, 2, 3, 4, 5}
```

```
>>> B = { 4, 5, 6, 7, 8, 9}
```

```
>>> 0 in A
```

False

```
>>> 4 in A
```

True

```
>>> A | B
```

{1, 2, 3, 4, 5, 6, 7, 8, 9}

```
>>> A & B
```

{4, 5}

```
>>> A - B
```

{1, 2, 3}

```
>>> A ^ B
```

{1, 2, 3, 6, 7, 8, 9}

Operadores:

in (\in)

| (\cup)

& (\cap)

-

^ (exclusión mutua)

Tipo Conjunto: ejemplos (3)

Scheme: es un derivado de LISP, un lenguaje funcional basado fundamentalmente en listas.

- En principio, todo es una lista, y las funciones son listas (con notación polaca prefija).
- Ejemplos:

```
`(a ((a b) c) d (e f))  
(car (quote (a b c d)))  
(eqv? ((1 2) (cons 1 (2))))
```

Tipo Conjunto: ejemplos (4)

[Java](#): ejemplo de implementación de una cola

```
import java.util.LinkedList

Queue q = new LinkedList();
q.add("elemento 1");
q.add("elemento 2");
q.add("elemento 3");
Object firstElement = q.element();
Object firstElement = q.remove();
```

Tipo Arreglo Asociativo

DEFINICIÓN: conjunto no ordenado de elementos de datos que son indexados por un igual número de valores llamadas “claves”.

Aspectos de diseño:

- ¿Cómo se referencian los elementos?.
- ¿Es el tamaño estático o dinámico?.

Lenguajes que lo soportan:

- Perl, Python y Ruby.
- Java no lo tiene como tipo del lenguaje, pero sí lo soporta mediante bibliotecas (*package*).

Tipo Arreglo Asociativo

Ejemplo: Python

```
>>> tel = {'Pedro':123453, 'Maria':234534, 'Juan':453345, 'Ana':645342}
```

```
>>> tel
```

```
{'Juan': 453345, 'Pedro': 123453, 'Ana': 645342, 'Maria': 234534}
```

```
>>> tel['Juan']
```

```
453345
```

```
>>> list(tel.keys())
```

```
['Juan', 'Pedro', 'Ana', 'Maria']
```

```
>>> sorted(tel.keys())
```

```
['Ana', 'Juan', 'Maria', 'Pedro']
```

```
>>> 'Pedro' in tel
```

```
True
```

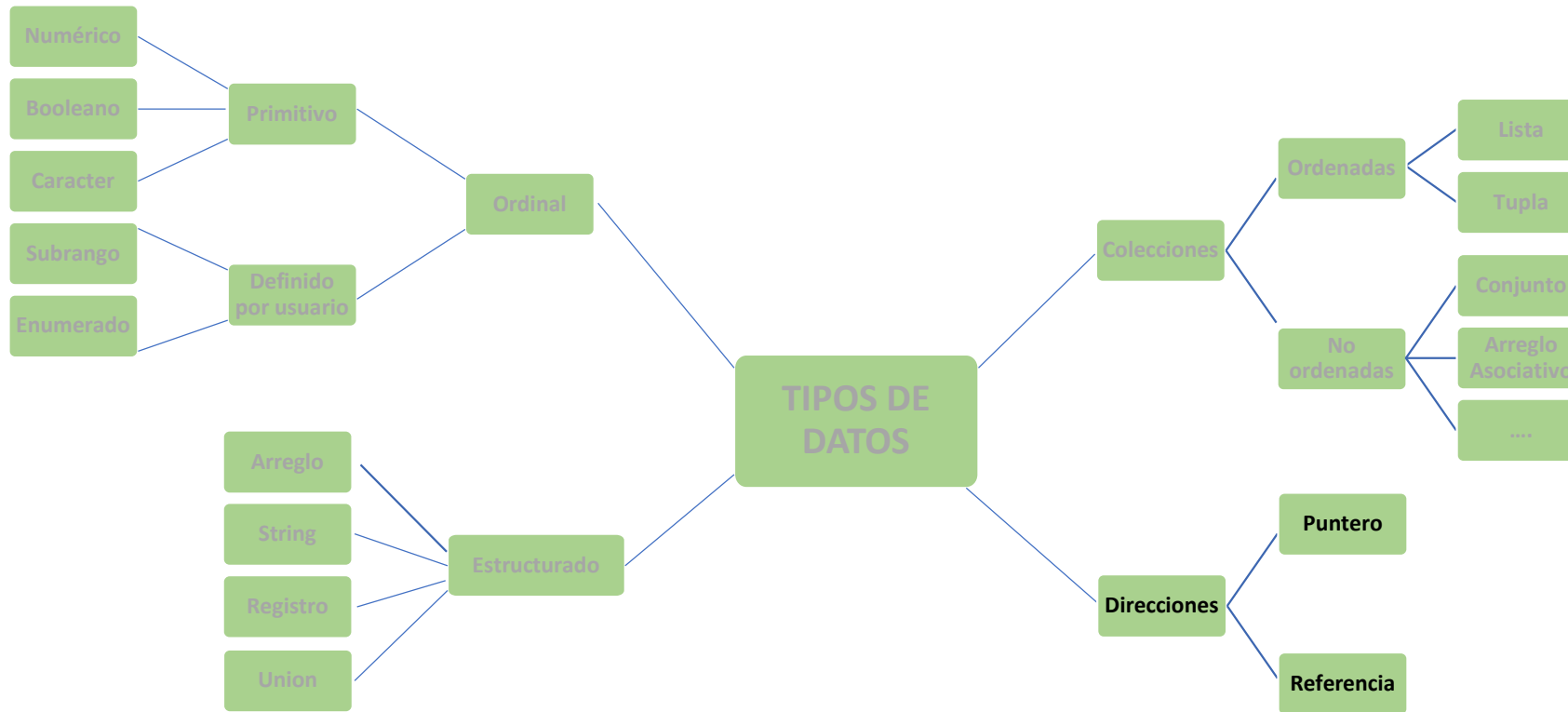
```
>>> 'Catalina' in tel
```

```
False
```

```
>>> del tel['Maria']
```

```
>>> tel
```

```
{'Juan': 453345, 'Pedro': 123453, 'Ana': 645342}
```



3.6 Punteros y Referencias

Tipo Puntero

DEFINICIÓN: su valor corresponde a una dirección de memoria, habiendo un valor especial nulo que no apunta a nada.

- Estrictamente *no* corresponde a un tipo estructurado, aun cuando se definen en base a un operador de tipo.

Aplicaciones:

- Método de gestión dinámica de memoria: permite mediante variable el acceso a la memoria dinámica de *heap*.
- Método de direccionamiento indirecto: útil para diseñar estructuras (ej.: Listas, árboles o grafos).

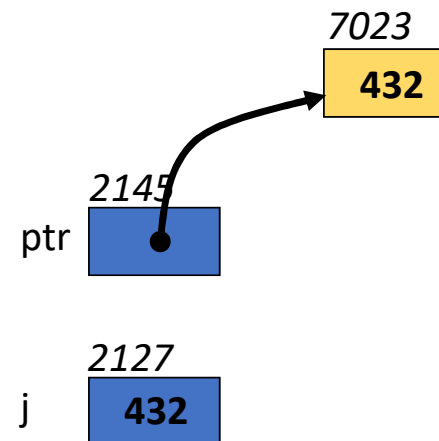
Tipo Puntero: Operaciones

CLASES DE OPERADORES:

- Asignación: asigna como valor a una variable la dirección a algún objeto de memoria del programa.
- Desreferenciación: entrega el valor almacenado en el objeto apuntado. Puede ser explícito o implícito.

Ejemplo en C:

```
int *ptr, j;  
ptr = (int*) malloc(sizeof(int));  
*ptr = 432;  
j = *ptr;
```



Tipo Puntero: C y C++

CARACTERÍSTICAS: extremadamente flexibles, a ser usados con cuidado.

- Son usados para la gestión dinámica de la memoria (*heap*) y para direccionamiento.
- Son un reflejo de la programación de sistemas en el origen de C.
- Se usa desreferenciación explícita (*) y un operador para obtener dirección de una variable (&).
- Definen un tipo de datos al que apunta un puntero.
 - void* puede apuntar a cualquier tipo!
- Soporta una aritmética de punteros basada en el tipo de datos del puntero.

Tipo Puntero: C y C++

```
struct nodo_t {  
    tipodato    info;  
    struct nodo_t *siguiente;  
};  
  
typedef nodo_t* enlace_t;  
  
enlace_t nodo;
```

```
#ifndef C++  
nodo = (enlace_t) malloc(sizeof(nodo_t));  
#elif  
nodo = new nodo_t; /* caso C++ */  
#endif  
  
(*nodo).info = dato;  
  
/* forma más conveniente de referirse es */  
nodo->siguiente = NULL;
```

Tipo Puntero: C y C++

- En C y C++ un arreglo es, en realidad, una constante de tipo puntero, que direcciona a la base del arreglo y que permite el uso de la aritmética de punteros. Ejemplo:

```
int a[10];
int *pa;

pa = &a[0];
pa = a;                                /* hace lo mismo que la línea anterior */

for (int i=0; i<10; i++)                /* los tres for hacen lo mismo */
    printf(a[i]);

for (int i=0; i<10; i++)                /* los tres for hacen lo mismo */
    printf(*(pa+i));

for (int i=0; i<10; i++)                /* los tres for hacen lo mismo */
    printf(*(a+i));}
```

Tipo Puntero: C

- Aritmética de Punteros. Ejemplo:

```
#define ALLOCSIZE 1000
static char allocbuf[ALLOCSIZE];
static char* allocp = allocbuf;
```

```
char *alloc(int n) {
    if (allocbuf + ALLOCSIZE - allocp >= n) {
        allocp += n;
        return allocp - n;
    } else return 0;
}
```

```
void afree(char *p) {
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

/ tamaño del buffer*/
/*... el buffer*/
/* primera posición libre*/*

/ retorna puntero a "n" caracteres*/
/* si cabe*/*

/ antigua direccion*/*

Tipo Referencia

DEFINICIÓN: es un tipo de variable que realiza desreferenciación implícita en la asignación, haciéndola más segura en su uso.

Ejemplos de Referencias:

- C++: después de su inicialización, las referencias permanecen constantes. Son útiles para usar parámetros pasados por referencia en funciones.
- Java: extiende el concepto de C++, haciendo que referencias se hagan sobre objetos, en vez de ser direcciones, eliminando los punteros.
- C#: permite el uso de referencias al estilo Java y punteros como C++.

Tipo Referencia: C++

Características:

- Variable de referencia se inicializa con una dirección en el momento de declararla o definirla.
- Referencia permanece constante, actuando como un alias.
- Cuando se realiza una asignación, no se requiere desreferenciar la variable.
- Ejemplo:

```
int valor = 3;  
int &ref_valor = valor;      /* inicializa */  
  
ref_valor = 100;
```

Tipo Referencia: C++ y Parámetros en Funciones

- Su uso en parámetros de funciones permite paso por referencia (comunicación bidireccional), y mejora la legibilidad.
- Inicialización se produce en el momento de la invocación.

/ Ejemplo en C */*

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
...
int x, y;
swap(&x,&y);
...
```

/ Ejemplo en C++ */*

```
void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
...
int x, y;
swap(x,y);
...
```

Tipo Referencia: Java

Comentarios:

- En Java las referencias son variables que apuntan a objetos, y no permiten otro uso (sólo tipos primitivos de datos usan semántica de valor para las variables).
- No existe operador de desreferenciación.
- Una asignación provoca que apunte a nuevo objeto.

Ejemplo:

```
Punto a;  
a = new Punto(3, 4);  
Punto b = a;  
Punto c = new Punto(7, 5);  
a = c;
```

Gestión del *Heap*

- En general, para el manejo de objetos de memoria dinámicos, punteros y referencias son implementados en el *heap*.
- Su implementación debiera atacar los problemas de *basura y dangling*, que estrechamente se relaciona con el manejo del *heap*.
- También se debe abordar cómo organizar y administrar eficientemente la memoria.

Gestión del *Heap*

Tamaño único: el caso más simple es administrar objetos (celdas) de memoria de un tamaño único:

- Celdas libres se pueden enlazar con punteros en una lista.
- La asignación es simplemente tomar suficientes celdas (contiguas) de la lista anterior
- La liberación es un proceso más complicado (se debiera evitar *dangling* y basura).

Tamaño variable: es lo normalmente requerido por los lenguajes, pero es complejo de administrar e implementar.

Gestión del *Heap*

El Administrador de la memoria *heap* marca cada celda de memoria que administra como **libre** o **ocupada**, según si la celda está disponible o asignada, de acuerdo a su propio conocimiento. No obstante, una celda puede tener cuatro estados reales:

- **Libre**: no tiene referencias y está marcada correctamente como libre por el administrador de memoria.
- **Basura**: no tiene referencia y no está marcada como libre; por ende, el administrador no la puede reasignar.
- **Ocupada**: tiene al menos una referencia y está asignada, es decir marcada correctamente por el administrador como ocupada.
- **Dangling** (“quedar colgado”): tiene alguna referencia y está marcada como libre; luego, el administrador la podría reasignar.

Gestión del *Heap*: Basura

Basura

- Pérdida de acceso a un objeto de memoria asignado en el *heap*, por no existir variables que apunten a él. Sucede porque se asigna una nueva dirección a la variable puntero que permitía el acceso.
- Produce pérdida o fuga de memoria, que puede causar inestabilidad en la ejecución del programa.

Gestión del *Heap*: Basura

Basura

- Ejemplo en C:

```
tipo* p;
```

```
p = (tipo *) malloc(sizeof(tipo));
```

```
...
```

```
p = (tipo *) malloc(sizeof(tipo));           /* se pierde la variable asignada anteriormente */
```

- Para evitar este problema, muchos lenguajes modernos introducen mecanismos para resolverlo (ej.: recolector de basura).

Gestión del *Heap*: Recolección de Basura

Contadores de referencia: enfoque impaciente

- Se mantiene un contador de referencia por cada celda.
- Dicho contador se incrementa con una nueva referencia y se decrementa cuando ésta se pierde.
- Una celda se libera tan pronto el contador llega a cero, es decir, se convierte en basura.

Marcar y barrer: enfoque perezoso

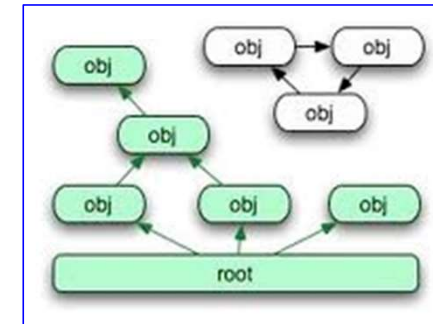
- Se acumula basura hasta que se agota la memoria.
- Al ocurrir lo anterior, se identifican las celdas de basura y se pasan a la lista de celdas libres.

Gestión del *Heap*: Recolección de Basura

Marcar y barrer:

```
void* allocate (int n)
{
    if (!hay_espacio) { /* aplicar mark&sweep */
        1) marcar todos los objetos del heap como basura;
        2) for (todo puntero p) { /* barrer */
            if (p alcanza objeto o en el heap)
                marcar o como NO basura;
        } /* for */
        3) liberar todos los objetos marcados como basura
    }

    if (hay_espacio) {
        asignar espacio;
        return puntero al objeto;
    } else return NULL;
}
```



Gestión del *Heap*: Recolección de Basura

Contadores de referencia:

- Requiere bastante memoria para mantener contadores.
- Asignaciones a punteros requieren de más tiempo de ejecución para mantener contadores.
- Produce una liberación gradual de la memoria, tan pronto se deja de usarla.

Marcar y barrer:

- Basta un bit por celda para marcar basura, siendo económico.
- Puede producir tiempos muertos significativos que afectan al funcionamiento del programa.
- Mal desempeño cuando queda poca memoria.
- Desventaja se mitiga si aumenta frecuencia de llamado.

Gestión del *Heap*: *Dangling*

Dangling

- Un puntero apunta a una localización de memoria del *heap* que ha sido liberada (e incluso nuevamente asignada).
- Se pueden producir efectos indeseados y peligrosos para el correcto funcionamiento del programa.

Gestión del *Heap*: *Dangling*

```
typedef tipo* ptipo;  
tipo  x;  
ptipo p, q;  
  
p = (ptipo) malloc(sizeof(tipo));  
...  
q = p;  
...  
free(p);                               /* q aun mantiene referencia al heap */  
...  
*q = x;                                /* puede que variable de heap sea reasignada */  
...
```

Gestión del *Heap*: *Dangling*

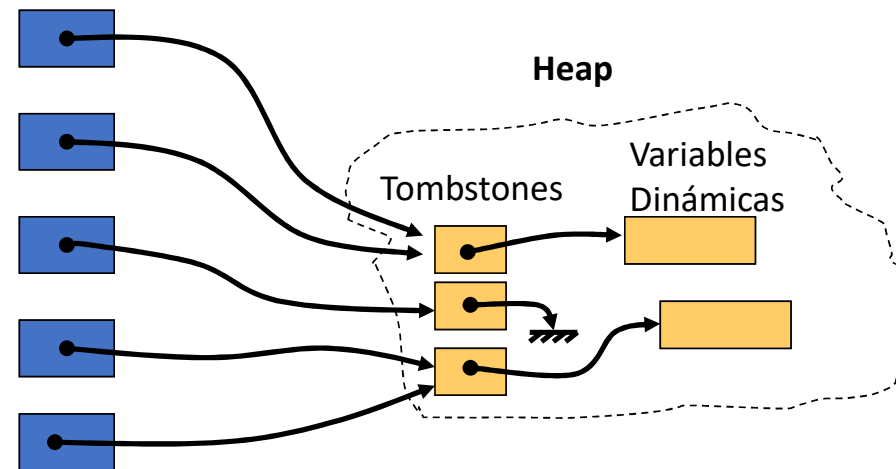
Métodos de Resolución:

- No permitir liberar memoria explícitamente.
- Lápida sepulcral (*tombstone*).
- Cerradura y Llave (*lock & key*).

Gestión del *Heap*: *Dangling*

Lápida sepulcral:

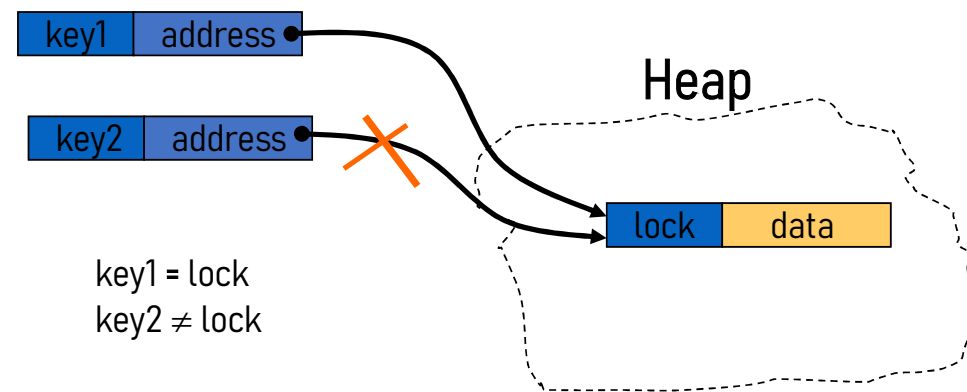
- Acceso se realiza indirectamente a través de una lápida.
- Si un objeto es liberado, la lápida permanece.
- Son costosos en tiempo y memoria.



Gestión del *Heap*: *Dangling*

Cerradura y Llave:

- Puntero es un par <clave, dirección>.
- Cada objeto de memoria en el *heap* mantiene una cabecera (cerradura) que almacena un valor.
- El acceso sólo es permitido si la clave del puntero o referencia coincide con el valor de la cerradura.



Gestión del *Heap*: ideas finales

- Mayor parte de los lenguajes requieren variables de tamaño variable.
- Mantención de celdas asignadas y libres se hace más difícil y costosa.
- Se requiere más memoria para mantener información sobre tamaño, estado, etc.
- Se produce fragmentación de la memoria.

Unidad 3

Tipos de Datos

FIN

3.1 Sistemas de Tipos

3.2 Tipos de Datos Ordinales o Simples

3.3 Arreglos y Strings

3.4 Registros y estructuras relacionadas

3.5 Colecciones

3.6 Punteros y Referencias