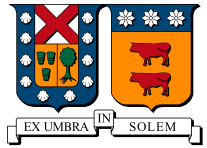
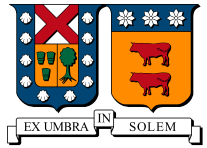


Tipos de Datos

Roberto Díaz
roberto.diazu@usm.cl

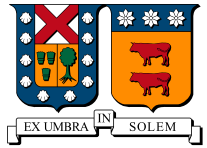
Universidad Técnica Federico Santa María
Departamento de Informática – Santiago, Chile

2021-2



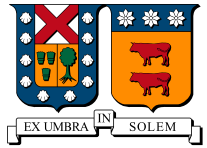
Sistema de Tipos

Sistema de Tipos



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).

Sistema de Tipos



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 (e.g. bibliotecas y paquetes).

Sistema de Tipos

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).

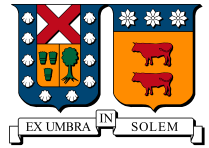
- **Ventajas**: permite verificar el uso correcto del lenguaje y detectar errores de tipos, tanto en tiempo de compilación o ejecución. Ayuda a modularizar (e.g. bibliotecas y paquetes).
- **Desventajas**: sistema muy estricto puede rechazar programas correctos. Para mitigar esto, se introducen “lagunas”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

Tipos de Datos

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

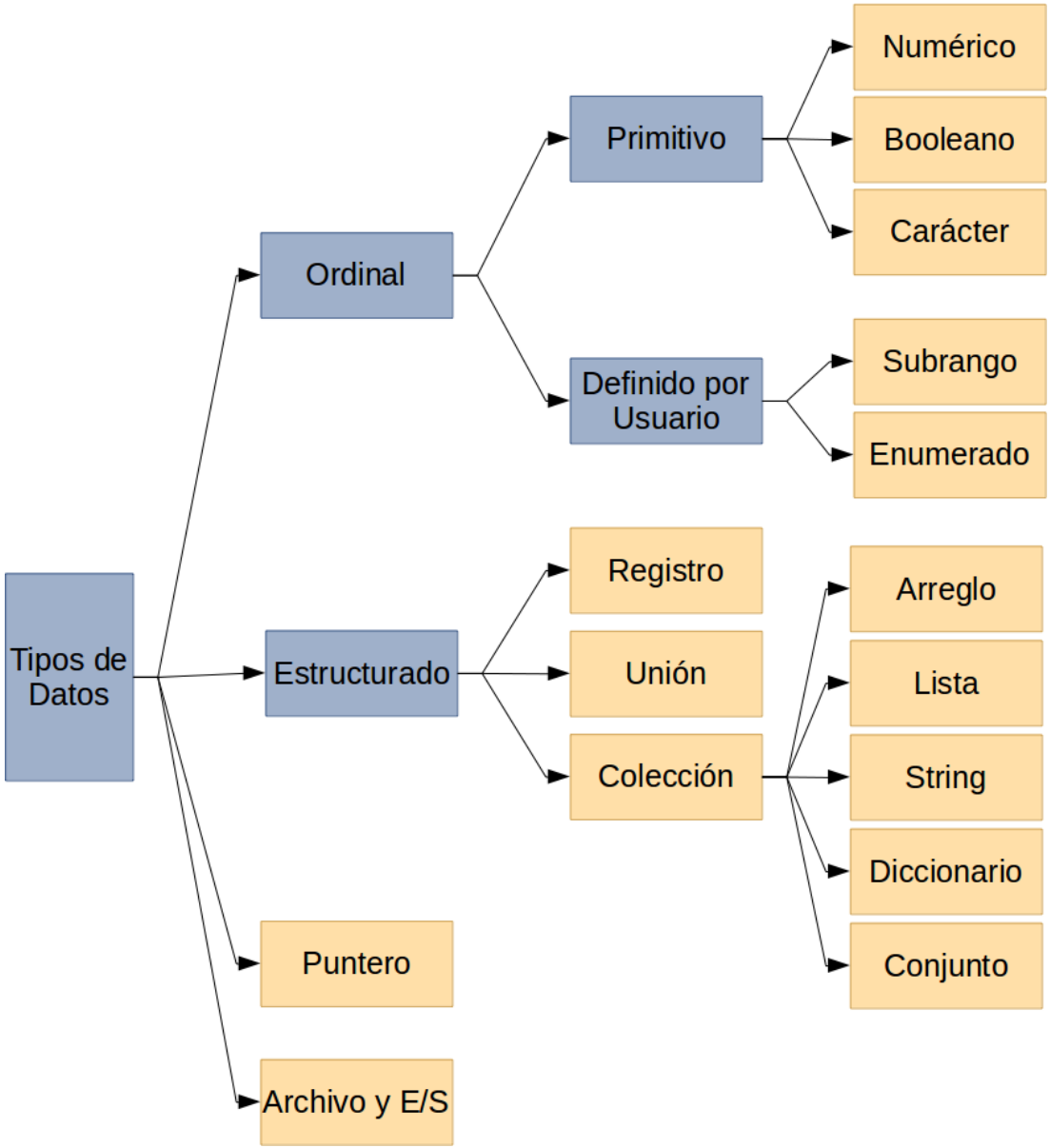
- Disponibilidad de gran número de tipos de datos ayuda a calzar los objetos de datos con un determinado problema.
- Lenguajes proveen conjunto de tipos primitivos y mecanismos o constructos que permiten definir nuevos tipos orientados a resolver un problema específico

Tipos de Datos
Ejemplos



- **Primitivos** (entero, punto flotante y caracter)
- **Estructuras de datos** (arreglo y registro)
- **Tipos definidos por usuario** (enumeración y subrango)
- **Tipos de datos abstractos** (listas, árbol)

Tipos de Datos
Taxonomía



Conceptos Básicos

Supuestos:

- Asignación es un operador binario, con una variable y una expresión como operandos (o argumentos).
- Subprograma es un operador cuyos parámetros son sus operandos.

Conceptos Básicos

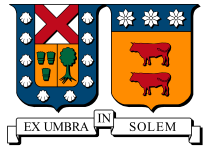
- **Verificación de tipo (*Type Checking*)**: asegurar que los operandos de un operador son de tipo compatible
- **Tipo compatible**: es un tipo legal, o que mediante reglas del lenguaje puede ser convertido en uno legal.
- **Conversión de tipos**: se denomina **coerción** a la conversión automática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

Equivalencia de Tipos

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 de 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

Ejemplo: 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;
```

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

Sistemas de tipos se pueden clasificar según el tiempo en que se realiza la verificación (*type checking*).

- **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. Aplica sólo si todos los tipos son ligados de manera estática.
- **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 (e.g. Scheme y Python).

Tipificación
Explícita vs Implícita

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 programas 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

Sistemas de tipos se pueden clasificar según el grado de exigencia impuesto en la verificación para detectar potenciales errores de tipo (*type checking*).

- **Tipificación fuerte**: siempre detecta errores de tipo (de forma estática o dinámica). 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
Razones

Tipificación estática:

- **Eficiencia de ejecución**: permite al compilador de antemano 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 o 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
Críticas

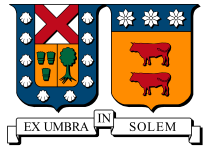
- Fuerza a los programadores a una disciplina rígida de programación, reduciendo facilidad de escritura de código y diseñar buenos programas.
- Aumenta el esfuerzo de programación al tener que cumplir con las exigencias del lenguaje, entregando mayor información de tipos.
- Hace más difícil el aprendizaje 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

Sistema de Tipos consiste en:

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



Tipos de Datos Ordinales o Simples

Tipo Ordinal

Un tipo ordinal (o simple) 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 basado en otro tipo del lenguaje
  - **Numérico** (entero, decimal, punto fijo, punto flotante)
  - **Carácter**
  - **Booleano**
- **(Tipos ordinales) definidos por el usuario**: Típicamente basados en tipos primitivos.
  - **Enumerado**
  - **Subrango**

Tipos Primitivos

Corresponde 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.

Tipos Primitivos

Corresponde 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** (e.g. C y Java permiten diferentes tipos de enteros: `short`, `long`, `signed`, `unsigned`)
  - **Punto flotante** (e.g. C y Java permiten `float` y `double`)
  - **Decimal** (típicamente 4 bits por dígito decimal)

Tipos Primitivos

Corresponde 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** (e.g. C y Java permiten diferentes tipos de enteros: `short`, `long`, `signed`, `unsigned`)
  - **Punto flotante** (e.g. C y Java permiten `float` y `double`)
  - **Decimal** (típicamente 4 bits por dígito decimal)
- **Booleano**:
  - Mejora legibilidad. Típicamente ocupa un byte (e.g. C++ y Java)

Tipos Primitivos

Corresponde 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** (e.g. C y Java permiten diferentes tipos de enteros: `short`, `long`, `signed`, `unsigned`)
  - **Punto flotante** (e.g. C y Java permiten `float` y `double`)
  - **Decimal** (típicamente 4 bits por dígito decimal)
- **Booleano**:
  - Mejora legibilidad. Típicamente ocupa un byte (e.g. C++ y Java)
- **Carácter**:
  - Tamaño de un byte; típicamente código ASCII (ISO 8859)
  - Tamaño variable (UTF-8 hasta 6 Bytes)
  - Unicode con 16b (UTF-16) usado en Java, o 32b (UTF-32)

Representación de Números

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.

Representación de Números

- **Enteros**: signo-magnitud, 1-Complemento, 2-Complemento.
- **Punto Flotante**: aproximación números reales (estándar IEEE 754).
- **Decimal**: *Binary Coded Decimal* o BCD, 4 bits por decimal. Más preciso, pero ocupa más memoria.
- **Complejo**: Algunos lenguajes lo soportan y lo representan como dos números de punto flotante (e.g. Python y Scheme).

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).

$$(-1)^s \times 1.m \times 2^{e-127}$$

  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).

$$(-1)^s \times 1.m \times 2^{e-1023}$$

  Rango aproximado de $\pm 10^{308}$ con precisión de 10^{-308} .

Tipo Carácter

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/UTF-32**: Largo de 2B o 4B, superconjunto de UCS-2 (2B usada en Python y hasta Java 1.4)

Lenguajes usan caracteres de escape (e.g. \") para representar caracteres sin afectar la sintaxis del lenguaje.

Tipo Enumerado

Tipo donde se enumeran (exhaustivamente) todos los posibles valores a tra-vés de constantes literales. Enumeración establece una relación de orden.

- Existe relación de orden que permite definir operadores relacionales como predecesor y sucesor.
- Mejoran facilidad de lectura y confiabilidad.
- Se implementan normalmente mapeando las constantes según su posición a un subrango de enteros.

Tipo Enumerado
Ejemplo C/C++

Sintaxis:

```
enum-type = "enum" [identifier] "{" enum-list "}"
enum-list = enumerator | enum-list "," enumerator
enumerator = identifier | identifier "=" constant-exp
```

Código:

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

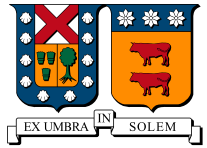
Tipo Subrango

Subsecuencia contigua de un tipo ordinal ya definido (e.g. `1..12`).

- Introducido por Pascal. Usado en Modula-2 y ADA.
- Mejoran facilidad de lectura y confiabilidad.
- Se implementan en base a tipo entero.

Ejemplo Pascal:

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



Arreglos y Strings

Tipo Arreglo

Es un tipo estructurado consistente en un conjunto homogéneo de elementos que se identifican por su posición relativa mediante un índice. Existe un tipo asociado a los elementos y otro al índice.

Índices de Arreglo

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 de datos del índice**:
  - C, C++, Java y Perl: sólo enteros.
  - Ada, Pascal: enteros y enumeración.
- **Prueba de rango de índice**:
  - C, C++, Perl: sin prueba de rango
  - Ada, Pascal, Java, C#: con prueba de rango.

Categorías de Ligado

- **Arreglo Estático**: rango de índice y memoria ligado antes de la ejecución (ejecución más eficiente)
  Ejemplo: único tipo en Fortran77.

Categorías de Ligado

- **Arreglo Estático**: rango de índice y memoria ligado antes de la ejecución (ejecución más eficiente)
  Ejemplo: único tipo en Fortran77.
- **Arreglo Dinámico Fijo de Stack**: rango de índice ligado estáticamente, pero 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).

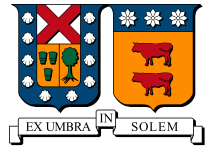
Categorías de Ligado

- **Arreglo Estático**: rango de índice y memoria ligado antes de la ejecución (ejecución más eficiente)
  Ejemplo: único tipo en Fortran77.
- **Arreglo Dinámico Fijo de Stack**: rango de índice ligado estáticamente, pero 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**: rango de índice y memoria asignada dinámicamente; permanece fijo durante tiempo de vida de la variable (más flexible que el anterior).
  Ejemplo: ADA y algunas implementaciones de C y C++.

Categorías de Ligado

- **Arreglo Fijo de Heap**: Similar a dinámico de stack, donde tamaño puede variar, pero permanece fijo una vez asignada la memoria.
  Ejemplo: C con malloc().

Categorías de Ligado



- **Arreglo Fijo de Heap**: Similar a dinámico de stack, donde tamaño puede variar, pero permanece fijo una vez asignada la memoria.
  Ejemplo: C con malloc().
- **Arreglo Dinámico de Heap**: Tamaño puede variar durante su tiempo de vida (la mayor flexibilidad).
  Ejemplo: Perl y Python.

Arreglos Dinámicos de Stack
Ejemplo

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

Arreglos de Heap
Ejemplo

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

Inicialización de Arreglos
Ejemplo

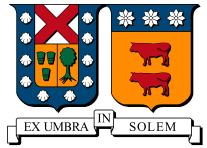
C y C++:

```
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"};
```

Arreglos Multidimensionales
Ejemplo



Pascal

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

C y C++

```
int matrix[dim1][dim2];
```

Operadores con Arreglos

- 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 (e.g. pertenencia, concatenación, subrango).

Implementación de Arreglos

La memoria es un arreglo unidimensional de celdas:

- Un arreglo es una abstracción del lenguaje
- Un arreglo debe ser mapeado a la memoria (arreglo de bytes)

Ejemplo: dirección de `lista[k]`

```
dir(lista[0]) + (k)*tamano
```

En C, siendo `arr` un arreglo, estas expresiones son equivalentes:

```
arr[k];
*(arr + k);
*(&arr[0] + k);
```

Arreglos bidimensionales se almacenan como filas (o columnas) consecutivas.

String

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 tamaño es estático o dinámico (memoria)?

Operaciones típicas:

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

Largo del String

Diseño de string considera:

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

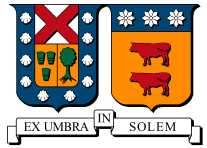
String
Ejemplos

C

```c
char str[20];
...
if (strcmp(str,"Hola") == 0){
...
} else {
...
}
```

Java

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



Registros y Estructuras Relacionadas

Tipo Registro

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.

- **Selección** permite acceder a un campo. Ejemplo C y C++: `var.campo`
- Típicamente solo soporta la operación de **Asignación**

Tipo Registro
Ejemplo

C y C++

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

empleado_t pelao, guaton;

guaton.sueldo = 550000.00;
strcpy(pelao.nombre.primer, "Juan");
```

Tipo Unión

Tipo que permite almacenar diferentes tipos de datos en diferentes tiempos en una misma variable.

- Se pueden definir varios miembros, pero solo un miembro puede contener un valor a la vez.
- Todos los miembros comparten la memoria y comienzan desde la misma dirección.
- Reserva espacio de memoria igual al mayor miembro definido.
- Su uso es en general poco seguro y hace que varios lenguajes no sean de tipificación fuerte.
- Soportado por Fortran, Ada, C y C++
- Java y C# no provee este tipo de estructura por ser inseguras.

Tipo Unión
Ejemplo

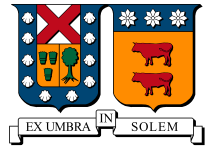
```c
union Data {
    int i;
    float f;
    char str[20];
};
...
union Data data;
printf("Memory size: %d\n", sizeof(data)); //Memory size: 20
data.i = 10;
printf("data.i: %d\n", data.i);      //data.i: 10
data.f = 220.5;
printf("data.f: %f\n", data.f);      //data.f: 220.5
strcpy(data.str, "C Programming");
printf("data.str: %s\n", data.str); //data.str: C
    Programming
```

Tipo Archivo

Tipo especial, que facilita la comunicación de un programa con el mundo externo (E/S), que es uno de los aspectos más difíciles, pero necesario, en el diseño de los lenguajes. Permite leer datos que existen antes de la ejecución del programa y escribir datos que persisten.

Clasificaciones:

- Persistente vs. Temporal.
- Método de acceso (secuencial, directo, indexado, etc.)
- Estructurado (secuencia de bit o bytes, o reconoce tipos de datos estructurados).



Colecciones

Colecciones de Datos

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

Permite almacenar un conjunto *no ordenado* de elementos de datos.

Lenguajes con tipificación explicita usualmente requieren definir un tipo base (e.g. C++, Java), mientras que otros admiten cualquier tipo.

Tipo Conjunto
Ejemplo

Java:

```
...
Set<Integer> A = Set.of(1,2,3,4,5);
Set<Integer> B = Set.of(4,5,6,7,8);

System.out.println(A.contains(0));
System.out.println(A.contains(1));

//union
Set<Integer> U = new HashSet<>(A);
U.addAll(B);

//interseccion
Set<Integer> I = new HashSet<>(A);
I.retainAll(B);
```

Tipo Conjunto
Ejemplo

Python

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> type(A)
<class 'set'>
>>> 0 in A
False
>>> 1 in A
True
>>> A | B
{1, 2, 3, 4, 5, 6, 7, 8}
>>> A & B
{4, 5}
>>> A - B
{1, 2, 3}
>>> A ^ B
{1, 2, 3, 6, 7, 8}
```

Tipo Arreglo Asociativo

Conjunto *no ordenado* de elementos de datos que son indexados por un igual número de valores llamadas ”claves”.
También conocidos como Hashes o Diccionarios.

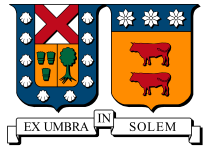
- Soportados directamente por Python, Ruby y Lua
- Parte de librerías estándar de Java, C++ y C#

Tipo Arreglo Asociativo
Ejemplo

Python

```
>>> tel = {'Pedro':123453, 'Maria':234534, 'Juan':453345, '
    Ana':645342}
>>> tel
{'Pedro': 123453, 'Maria': 234534, 'Juan': 453345, 'Ana':
    645342}
>>> tel['Juan']
453345
>>> tel.keys()
dict_keys(['Pedro', 'Maria', 'Juan', 'Ana'])
>>> 'Pedro' in tel
True
>>> 'Catalina' in tel
False
>>> del tel['Maria']
>>> tel
{'Pedro': 123453, 'Juan': 453345, 'Ana': 645342}
```

Tipo Arreglo Asociativo
Ejemplo

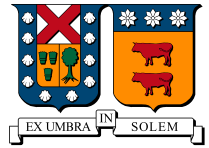


Java

```
Map map = new HashMap();
map.put("Pedro",123453);
map.put("Maria",234534);
map.put("Juan",453345);
map.put("Ana",645342);

map.get("Pedro");          //123453

map.get("Catalina");       //null
```



Tipo Puntero y Referencia

Tipo Puntero

Su valor corresponde a una dirección de memoria, habiendo un valor especial nulo (`nil` o `null`) que no apunta a nada.
Estrictamente no corresponde a un tipo estructurado, aún 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 (e.g. Listas, árboles o grafos).

Tipo Puntero
Operadores

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.

Ejemplos:

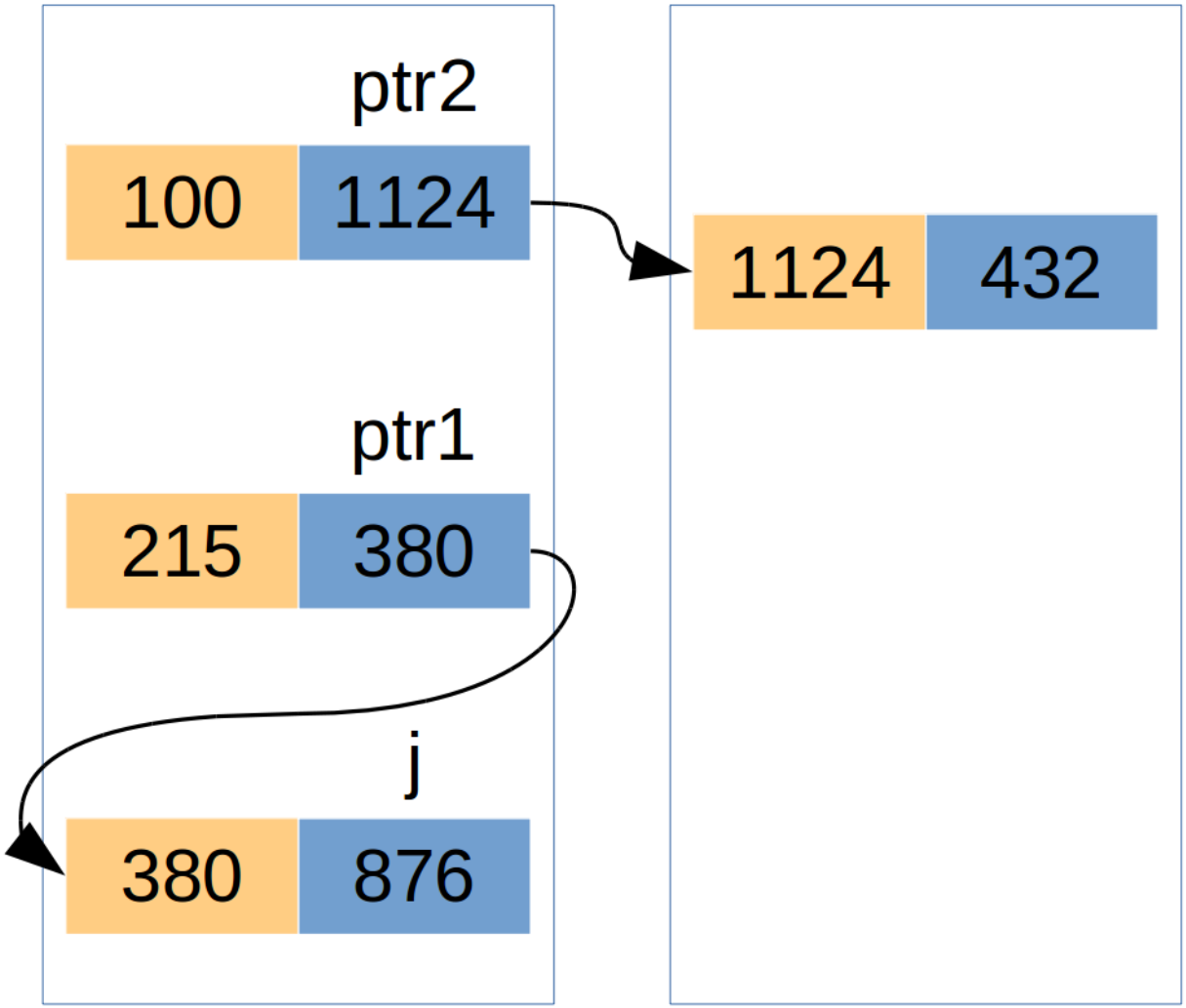
- Operador `*` en C y C++ (e.g. `*ptr`)
- Operador `^` en Pascal (e.g. `ptr^`)

Tipo Puntero
Ejemplo C

```c
int *ptr1, j;
j = 876;
ptr1 = &j;

int *ptr2;
ptr2 = (int*)malloc(sizeof(int));
*ptr2 = 432;
free(ptr2);
```

Tipo Puntero
Ejemplo C



Puntero en C y C++

Los punteros en C y C++ son extremadamente flexibles y deben ser utilizados con mucho 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.

Estructuras y Punteros
Ejemplo C/C++

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

enlace_t nodo;

#ifndef C++
nodo = (enlace_t) malloc(sizeof(struct nodo_t));
#elif
nodo = new nodo_t; /* caso C++ */
#endif

(*nodo).info = dato;

/* forma mas conveniente de referirse es */
nodo->siguiente = NULL;
```

Arreglo y Puntero
Ejemplo C/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.

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
int *pa;

pa = &a[0];
pa = a;

for (int i=0; i<10; i++)
  printf("%d\n",a[i]);
for (int i=0; i<10; i++)
  printf("%d\n",*(pa+i));
for (int i=0; i<10; i++)
  printf("%d\n",*(a+i));
```

Tipo Referencia

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

Ejemplos de Referencias:

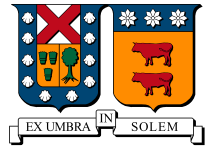
- **C++**: las referencias después de su inicialización permanecen constantes. Son útiles para usar parámetros pasados por referencia en funciones.
- **Java**: extiende el concepto de C++, haciendo que referencias se hagan sobre objetos, en vez de ser direcciones, eliminando los punteros.
- **C#**: permite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

Tipo Referencia
C++

- 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.

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

Tipo Referencia
C++ - Parámetros de 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.

Tipo Referencia
C++ - Parámetros de 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.

C

```
void swap(int *a, int *b){
  int tmp = *a; *a = *b; *b = tmp;
}

int a,b;
swap(&a,&b);
```

Tipo Referencia
C++ - Parámetros de 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.

C

```
void swap(int *a, int *b){
  int tmp = *a; *a = *b; *b = tmp;
}

int a,b;
swap(&a,&b);
```

C++

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

int a,b;
swap(a,b);
```

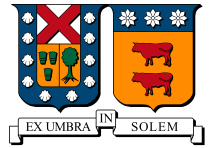
Tipo Referencia
Java

- 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:

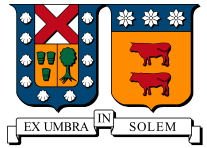
```
Punto a;
a = new Punto(3,4);
Punto b = a;
Punto c = new Punto(7,5);
a = c;
```

Funciones como referencia



Posibilidades:

- Variable que referencia a una función
- Paso de una función como parámetro en un subprograma



Implementación de Punteros y Referencias

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.
- ***Basura***: no tiene referencias y no está marcada como libre; por ende, no se puede reasignar.
- **Ocupada**: tiene al menos una referencia y está asignada, es decir marcada como ocupada.
- ***Dangling***: tiene una referencia y esta márcada como libre. Se puede reasignar.

Gestión del Heap

Punteros y referencias para el manejo de objetos de memoria dinámicos son implementados en el heap.

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

- Celdas libres se pueden enlazar con punteros en una lista.
- Asignación es simplemente tomar suficientes celdas (contiguas) de la lista anterior
- Liberación es un proceso más complicado (se debieran evitar problemas como el dangling y basura).

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

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.

```
p = (int*)malloc(sizeof(int));
q = p;
free(p);
*q = x;
```

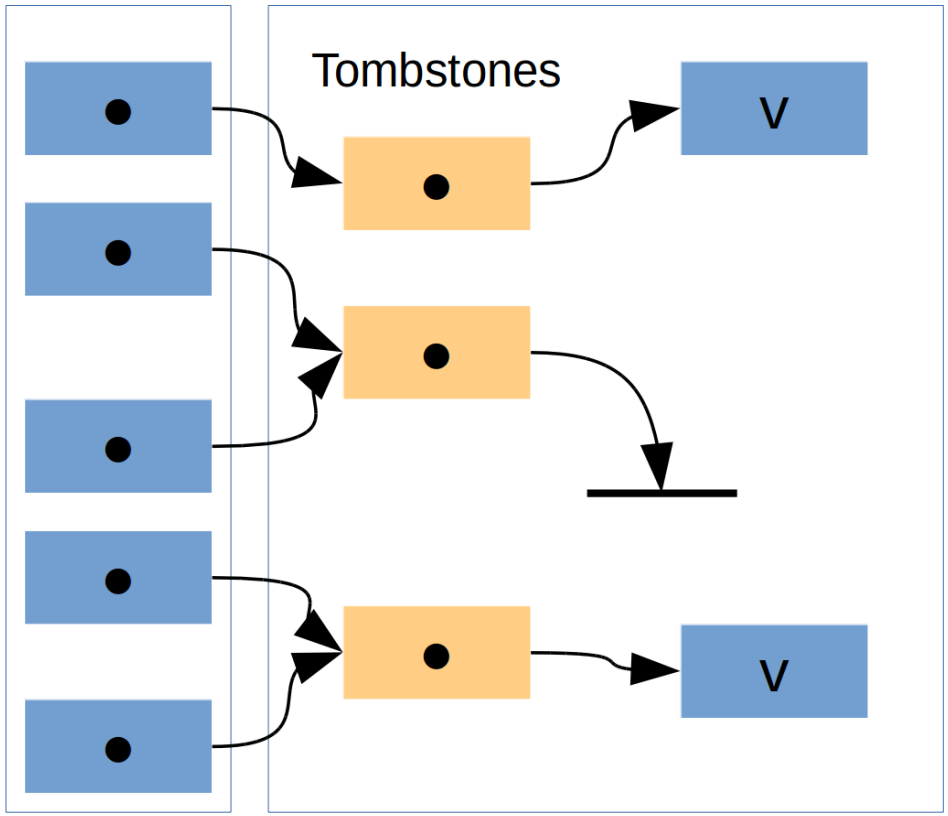
Dangling
Soluciones

Métodos de solución al problema de Dangling:

- Lápida sepulcral (tombstone).
- Cerraduras y Llaves (locks y keys).
- No permitir liberar memoria explícitamente.

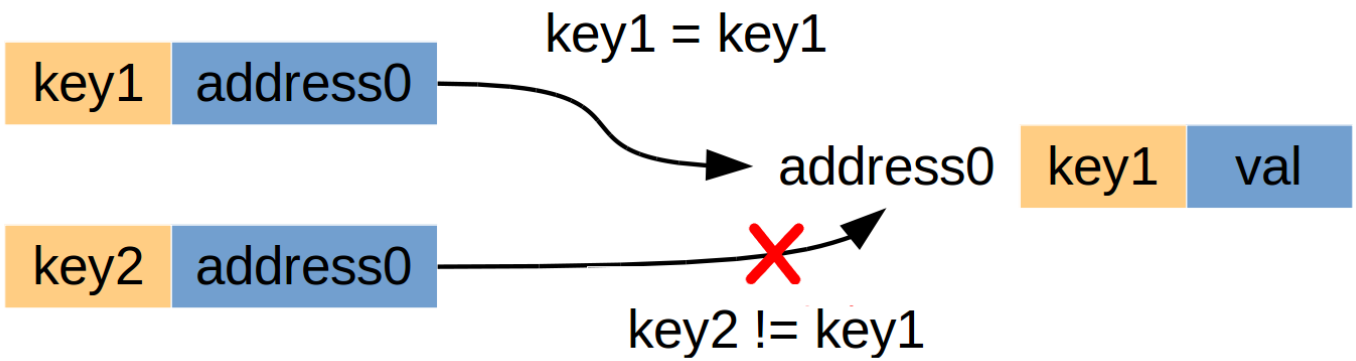
Dangling
Lápida Sepulcral

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



Dangling
Cerraduras y Claves

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



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.

```
int *p;
...
p = (int*)malloc(sizeof(int));
...
p = (int*)malloc(sizeof(int));
```

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 (mark & sweep): 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.

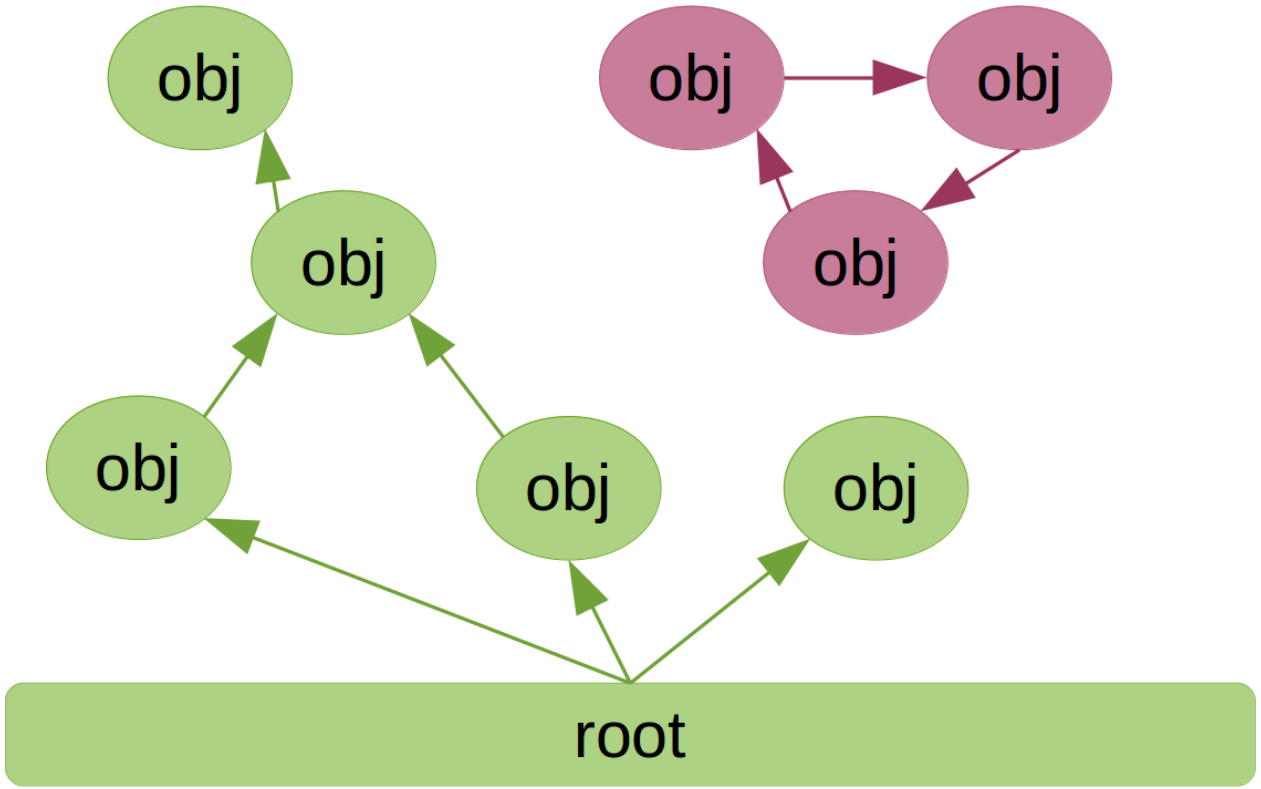
Contadores de referencia:

- 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.

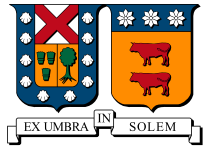
Garbage
Marcar y Barrer



Garbage
Ejemplo de Marcar y Barrer

```
void* allocate (int n){
  if (!hay_espacio) { /* aplicar mark&sweep */
    markGarbage(); /* marcar objetos del heap como basura */
    for (todo puntero p) { /* barrer */
      if (p alcanza objeto obj en el heap)
        marcar obj como NO basura;
    } /* for */
    freeGarbage(); /* liberar objetos marcados como basura*/
  }
  if (hay_espacio) {
    asignar espacio;
    return puntero al objeto;
  } else return NULL;
}
```

Problema de Gestión de Memoria



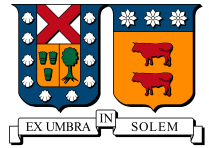
Mayor parte de los lenguajes requieren variables de tamaño variable.

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.

Referencias



- Capítulos VI de [Sebesta, 2011]
- Capítulos VIII de [Louden, 2011]
- Capítulo V y VI de [Tucker, 2006]