

Algoritmos y Estructuras de Datos II

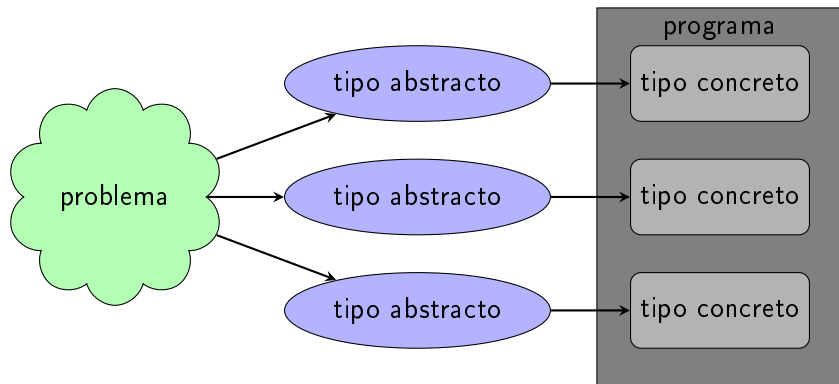
Tipos concretos

6 de abril de 2020

Tipos de datos

- Enumerados
- Tuplas
- Arreglos
- Punteros

Tipos abstractos de datos y su representación



Tipos de datos

Conceptualmente distinguimos dos clases de tipos de datos:

- ▶ Tipos de datos **concretos**:
 - ▶ son provistos por el lenguaje de programación, decimos que son *nativos*,
 - ▶ es un concepto **dependiente** del lenguaje de programación,
 - ▶ comúnmente: enteros, char, string, booleanos, arreglos, reales,
- ▶ Tipos de datos **abstractos**:
 - ▶ **surgen de analizar el problema** a resolver,
 - ▶ es un concepto **independiente** del lenguaje de programación,
 - ▶ eventualmente se implementará utilizando tipos concretos,
 - ▶ eso da lugar a una **implementación** o **representación** del tipo abstracto
 - ▶ ejemplo: si se quiere desarrollar una aplicación para un gps que calcule ciertos caminos óptimos, seguramente surgirá considerar un grafo donde las aristas son segmentos de rutas.

Tipos concretos

En estas filmas presentaremos los tipos concretos más complejos que tenemos en el lenguaje de la materia, y que están presentes en la mayoría de los lenguajes de programación.

Saltearemos los mas simples como booleanos, enteros, char, reales, string.

Presentamos a continuación los tipos enumerados, tuplas, arreglos y punteros.

Tipos enumerados

Declaración

- ▶ Un tipo enumerado consiste de una serie finita de constantes, que serán los únicos elementos del tipo.
- ▶ En nuestro lenguaje el programador puede definir un nuevo tipo enumerado de la siguiente forma:

```
type E = enumerate  
    elem1  
    elem2  
    .....  
    elemk  
end enumerate
```

Tipos enumerados

Uso

- ▶ El tipo E así definido tiene **únicamente** los k elementos elem1, elem2, ..., elemk.
- ▶ En un programa se puede declarar una variable del tipo enumerado E y asignarla a una de las constantes que define:

```
var e: E  
e := elem2
```

- ▶ En nuestro lenguaje permitimos realizar ciclos **for** donde el índice tome los valores de un tipo enumerado:

```
for i:= elem1 to elemk do .... od
```

Tipos enumerados

- ▶ Cuando definimos un tipo enumerado, sus elementos tendrán el orden en que fueron escritos en la definición. En el ejemplo anterior $\text{elem1} < \text{elem2} < \dots < \text{elemk}$.
- ▶ El tipo concreto **char** también puede verse como un tipo enumerado donde las constantes estarán ordenadas lexicográficamente ('a' < 'b' < 'c' < ...).

Tuplas

Declaración

También llamados registros (*records*) o estructuras (*structs*). Las tuplas representan productos cartesianos de distintos tipos. Se puede definir un nuevo tipo mediante una tupla de la siguiente forma:

```
type tperson = tuple  
    name: string  
    age: nat  
    weight: real  
end tuple
```

Tuplas

Uso

- ▶ El tipo `tperson` así definido corresponde al producto $\text{string} \times \text{nat} \times \text{real}$, y `name`, `age` y `weight` se llaman **campos**.
- ▶ Para acceder a los campos de un elemento de tipo tupla utilizamos el operador de acceso escrito con un punto:

```
var manu : tperson  
manu.name := "Emmanuel"  
manu.age := 33  
manu.weight := 68
```

- ▶ En la mayoría de los lenguajes de programación, cuando se ejecuta un programa los elementos de un tipo tupla son alojados en espacios contiguos de memoria y el espacio que ocupa es igual a la suma de los espacios que ocupan sus campos.

Arreglos

Declaración

- ▶ Los arreglos son colecciones de tamaño fijo de elementos del mismo tipo.
- ▶ En general los lenguajes de programación implementan los arreglos alojándolos en espacios contiguos de memoria, y el acceso a cada uno de sus elementos se obtiene en tiempo constante.
- ▶ Para declarar un arreglo en un programa debemos indicar de qué tipo **T** son los elementos y dos números naturales **M** y **N** indicando el primero y último índice respectivamente:

```
var a: array[M..N] of T
```

- ▶ El arreglo **a** tiene **N-M** elementos de tipo **T**.

Arreglos

Índices

- ▶ Comúnmente en otros lenguajes como C, los índices siempre comienzan en 0, y entonces solo debo indicar cuántos elementos tiene el arreglo.
- ▶ Nuestro lenguaje es más flexible, permitiendo indicar el índice del comienzo y final del arreglo, y también permitimos utilizar como índices a los elementos de un tipo enumerado:

```
var a: array[elem1..elemk] of T
```

donde elem1 ... elemk son los elementos del tipo enumerado definido previamente.

```
var page : array['a'..'z'] of nat
```

Arreglos

- ▶ También es frecuente la utilización de arreglos multidimensionales,
- ▶ ejemplos:
var b: array[1..N,1..M] of string
var c: array[1..N,'a'..'z',1..M] of nat
- ▶ Para acceder a los elementos de un arreglo utilizamos los corchetes, separando con comas cuando tenemos más de una dimensión:

`b[2,3] := "casa"`

`c[2,'x',1] := 25`

Arreglos

Ejemplo inicialización

- ▶ Si quiero inicializar el arreglo `c` puedo utilizar tres **for** anidados, de la siguiente manera:

```
for i:= M to N do
  for k:= 'a' to 'z' do
    for d:= 1 to M do
      c[i,k,d]:= 0
    od
  od
od
```

Punteros

Declaración

Dado un tipo T , un **puntero a T** es un tipo de datos que representa el lugar en la memoria en donde está alojado un elemento de tipo T .

Por ejemplo se puede declarar un puntero a **nat** así:

```
var p: pointer to nat
```

Punteros

Operaciones

Mediante punteros el programador puede manipular la memoria disponible al ejecutar un programa.

- ▶ Para **reservar** un nuevo bloque de memoria donde pueda almacenar un elemento del tipo al que apunta se utiliza la operación **alloc**

`alloc(p)`

- ▶ Puedo acceder al valor en el bloque de memoria apuntado por `p` mediante la operación `*`

`*p := 10`

decimos que `p` *apunta* al valor 10.

Punteros

Operaciones

- ▶ Para **liberar** un bloque de memoria que haya sido reservado antes se utiliza la operación **free**

`free(p)`

Luego de realizar un `free`, `p` ya no apunta a un bloque de memoria reservado por el programa, por lo cual no se puede saber qué valor tiene la expresión `*p`.

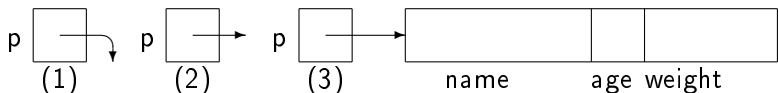
- ▶ Existe una constante para representar punteros que no apunten a nada, a la que llamamos **null**

`p := null`

Punteros

Representación gráfica

Hay distintas representaciones gráficas, una para cada una de las posibles situaciones. Dado p : **pointer to tperson**



- ▶ En la situación (1), el valor de p es **null**, p no señala ninguna posición de memoria.
- ▶ En la situación (2) la posición de memoria señalada por p no está reservada, por ejem. inmediatamente después de `free(p)`.
- ▶ En la situación (3) el valor de p es la dirección de memoria donde se aloja la `tperson` representada gráficamente al final de la flecha, por ejemplo, inmediatamente después de `alloc(p)`.

Punteros

Representación gráfica

- ▶ En la situación (3), $\star p$ denota la `tperson` que se encuentra señalada por `p`, y por lo tanto, $\star p.name$, $\star p.age$ y $\star p.weight$, sus campos.
- ▶ Esta notación permite acceder a la información alojada en la `tperson` y modificarla mediante asignaciones a sus campos (por ejemplo, $\star p.name := \text{"Juan"}$).

Punteros

Notación

- ▶ Una notación conveniente para acceder a los campos de una tupla señalada por un puntero es la flecha “→”.
- ▶ Así, en vez de escribir $\star p.name$, podemos escribir $p \rightarrow name$ tanto para leer ese campo como para modificarlo.
- ▶ Esta notación reemplaza el uso de dos operadores (“ \star ” y “ $.$ ”) por uno visualmente más apropiado (por ejemplo, $p \rightarrow name :=$ “Juan”).
- ▶ La notación $\star p$ y sus derivadas $\star p.name$, $p \rightarrow name$, etc. sólo pueden utilizarse en la situación (3).

Punteros

Punteros colgantes y **null**

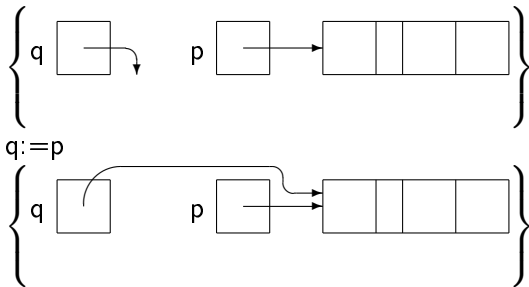
En la situación (2) el valor de p es inconsistente, no debe utilizarse ni accederse una dirección de memoria no reservada ya que no se sabe, a priori, qué hay en ella (en particular puede haber sido reservada para otro uso y al modificarlo se estaría corrompiendo información importante para tal uso). Los punteros que se encuentran en la situación (2) se llaman comúnmente referencias o punteros colgantes (dangling pointers).

En la situación (1) el valor de p es **null**, es decir que p no señala ninguna posición de memoria. Por ello, no tiene sentido intentar acceder a ella.

Punteros

Aliasing

Como vemos, los punteros permiten manejar explícitamente direcciones de memoria. Esto no es sencillo, aparecen situaciones que con los tipos de datos usuales no se daban. Por ejemplo:



Punteros

Aliasing

Como se ve, después de la asignación, q y p señalan a la misma tupla, por lo que cualquier modificación en campos de $*q$ también modifican los de $*p$ (claro, ya que son los mismos) y viceversa. Estamos en presencia de lo que se llama **aliasing**, es decir, hay 2 nombres distintos ($*p$ y $*q$) para el mismo objeto y al modificar uno se modifica el otro. Programar correctamente en presencia de aliasing es muy delicado y requiere gran atención.

Punteros

Administración de la memoria

Siempre hemos asumido que no es necesario ocuparse de reservar y liberar espacios de memoria para las variables. Los punteros como p y q son variables, así que **tampoco es necesario reservar y liberar espacio para ellos**. Pero las operaciones `alloc` y `free` son las responsables de reservar y liberar explícitamente espacio **para los objetos que p y q señalan**.

Esta posibilidad significa ciertas libertades: el programador puede decidir exactamente cuándo reservar espacio para una tupla. Por otro lado, significa también más responsabilidad: el programador es el que debe encargarse de liberar el espacio cuando deje de ser necesario.

Punteros

Administración de la memoria

Pero el verdadero beneficio de los punteros radica en que permiten una gran flexibilidad para representar estructuras complejas, y por lo tanto, para implementar diferentes tipos abstractos de datos que veremos en las próximas semanas.