

# Containere și iteratori

- Un *container* este o grupare de date în care se pot adăuga (insera) și din care se pot șterge (extrage) obiecte.
- Un container poate fi definit ca fiind o colecție de date care suportă cel puțin următoarele operații:
  - *adăugarea* unui element în container;
  - *ștergerea* unui element din container;
  - *returnarea numărului de elemente* din container (dimensiunea containerului);
  - furnizare *acces* la obiectele stocate (de obicei folosind iteratori) - *căutarea* unui obiect în container.
- TAD
- Ce container de date este potrivit într-o anumită aplicație?

## Iteratori

- Iteratorii pot fi văzuți ca o generalizare a referințelor, și anume ca obiecte care referă alte obiecte. Iteratorii sunt des utilizați pentru a parcurge un container de obiecte.
- Sunt importanți în programarea generică: un container trebuie doar să furnizeze un mecanism de accesare a elementelor sale folosind iteratori.
- Iteratorul va conține
  - o referință spre containerul pe care-l iterează
  - o referință spre elementul curent din iterație, referință numită în general *curent* (cursor).
- Iterarea elementelor containerului se va face mutând referința “curent” (în funcție de tipul iteratorului) în container atâta timp cât referința este validă (adică mai sunt elemente de iterat în container).
- Există mai multe categorii de iteratori, în funcție de maniera de iterare a containerului:
  1. iteratori unidirecționali (cu control într-o direcție);
  2. iteratori bidirecționali (cu control în două direcții);
  3. iteratori cu acces aleator ;
  4. *read-write* (permite ștergere și inserare de elemente în container)

Vom prezenta specificația TAD **Iterator** cu o interfață minimală (numărul minimal de operații) pentru un iterator unidirecțional.

## TAD Iterator domeniu

$$\mathcal{I} = \{i \mid i \text{ este un iterator pe un container} \\ \text{având elemente de tip } TElement\}$$

**operații** (interfața TAD-ului Iterator)

- $creeaza(i, c)$ 
  - $pre : c$  este un container
  - $post : i \in \mathcal{I}$ , s-a creat iteratorul  $i$  pe containerul  $c$ , (elementul  $curent$  din iterator referă 'primul' element din container)
- $element(i, e)$ 
  - $pre : i \in \mathcal{I}$ ,  $curent$  este valid (referă un element din container)
  - $post : e \in TElement$ ,  $e$  este elementul curent din iterație (elementul din container referit de  $curent$ )
- $valid(i)$ 
  - $pre : i \in \mathcal{I}$
  - $post : valid = \begin{cases} adev, & \text{dacă } curent \text{ referă o poziție validă} \\ & \text{din container} \\ fals, & \text{contrar} \end{cases}$
- $următor(i)$ 
  - $pre : i \in \mathcal{I}$ ,  $curent$  este valid
  - $post : curent'$  referă 'următorul' element din container față de cel referit de  $curent$

- Orice *container* va avea în interfața sa o operație

–  $iterator(c, i)$

$$\begin{aligned} pre : c & \text{ container} \\ post : i & \in \mathcal{I}, i \text{ este un iterator pe containerul } c \end{aligned}$$

- Folosind iteratori putem crește foarte mult gradul de genericitate a algoritmilor care lucrează pe containere.

Tipărirea elementelor din containerul  $c$  se va face în felul următor:

**Subalgoritm** Tiparire( $c$ )

*pre:*  $c$  este un container de date

*post:* elementele containerului  $c$  au fost tipărite

**iterator**( $c, i$ )

{containerul își obține iteratorul}

**CatTimp** **valid**( $i$ ) **executa**

{cât timp iteratorul e valid}

**element**( $i, e$ )

{se obține elementul curent din iterație}

**tipareste**( $e$ )

{se tipărește elementul curent}

**următor**( $i$ )

{se deplasează iteratorul}

**SfCatTimp**

**SfSubalgoritm**