Containere şi iteratori

- Un *container* este o grupare de date în care se pot adăuga (insera) şi din care se pot şterge (extrage) obiecte.
- Un container poate fi definit ca fiind o colecţie de date care suportă cel puţin următoarele operaţii:
  - *adăugarea* unui element în container;
  - *ştergerea* unui element din container;
  - *returnarea numărului de elemente* din container (dimensiunea containerului);
  - *căutarea* unui obiect în container.
  - furnizare *acces* la obiectele stocate (de obicei folosind iteratori) - *căutarea* unui obiect în container.
- TAD
- Ce container de date este potrivit într-o anumită aplicaţie?

Iteratori

- Iteratorii pot fi văzuţi ca o generalizare a referinţelor, şi anume ca obiecte care referă alte obiecte. Iteratorii sunt des utilizaţi pentru a parcurge un container de obiecte.
- Sunt importanţi în programarea generică: un container trebuie doar să furnizeze un mecanism de accesare a elementelor sale folosind iteratori.
- Iteratorul va conţine (Figura 1)

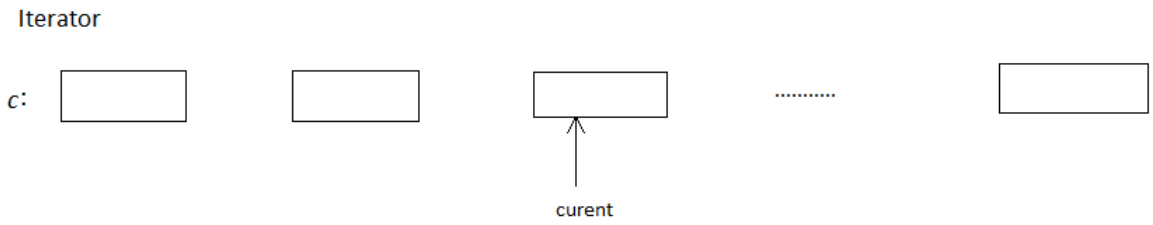

Figura 1: Iterator pe un container c.

  - o referinţă spre containerul pe care-l iterează
  - o referinţă spre elementul curent din iteraţie, referinţă numită în general *curent* (cursor).

- Iterarea elementelor containerului se va face mutând referinţa "curent" (în funcţie de tipul iteratorului) în container atâta timp cât referinţa este validă (adică mai sunt elemente de iterat în container).
- Există mai multe categorii de iteratori, în funcţie de maniera de iterare a containerului:
  1. iteratori unidirecţionali (cu control într-o direcţie);
  2. iteratori bidirecţionali (cu control în două direcţii);
  3. iteratori cu acces aleator ;
  4. *read-write* (permite ştergere şi inserare de elemente în container)

Vom prezenta specificaţia TAD Iterator cu o interfaţă minimală (numărul minimal de operaţii) pentru un iterator unidirecţional.

TAD Iterator
domeniu

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

operaţii (interfaţa TAD-ului Iterator)

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

Observaţii

- pentru simplitate, operaţiile **creeaza** şi **prim** se pot combina în operaţia **creeaza** (constructor) cu specificaţia de mai jos
- 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)

- vom considera în cele ce urmează varianta simplificată de mai sus

- Orice *container* va avea în interfaţa sa o operaţie
  - iterator(c, i)

    $pre:$ c *container*
    $post:$ $i \in \mathcal{I}, i$ este un iterator pe containerul c

- Operaţia **iterator** din interfaţa containerului apelează, în general, constructorului iteratorului

```
  Subalgoritm iterator(c, i)
pre:  c este un container de date
post: i este un iterator pe containerul c
    {se creeaza iteratorul i pe containerul c}
    creeaza(i, c)
  SfSubalgoritm
```

- Folosind iteratori putem creşte 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 îşi obţine iteratorul}
    CatTimp valid(i) executa
      {cât timp iteratorul e valid}
      element(i, e)
      {se obţine elementul curent din iteraţie}
      tipareste(e)
      {se tipăreşte elementul curent}
      următor(i)
      {se deplasează iteratorul}
    SfCatTimp
  SfSubalgoritm
```

Articol şi tablou

Articolul (înregistrarea)

- Structură de date statică.
- Un articol reprezintă reuniunea unui număr fix de componente care pot avea tipuri diferite (caracter *neomogen*), numite *câmpuri*, care constituie logic o unitate de prelucrare.

$$\mathsf{Persoana} = \{nume, data\_nasterii, adresa, ID\}$$

.

- Folosite pentru descrierea unor obiecte care constau din mai multe componente.
- Operaţii specifice: **creare**, **selecţie** şi **modificare** componente.

Tabloul

- Structură de date statică.
- Notaţii
  - $[n] = \{1, 2, \ldots, n\}, n \in \mathbb{N}^*$
  - colecţie C de elemente având tipul generic *TElement*.

  Aplicaţia $f : [n_1] \times [n_2] \times \cdots \times [n_k] \to C$, care ataşează fiecărui k-uplu de indici $(i_1, i_2, \ldots, i_k)$ unde $i_j \in [n_j]$, un element $c_{i_1,i_2,\ldots,i_k}$ defineşte un tablou k-dimensional $T(n_1, n_2, \ldots, n_k)$.
- În cazul particular $k = 1$ se obţine un tablou unidimensional numit *vector*, $f : [n] \to C$, având elementele $c_1 = f(1), c_2 = f(2), \ldots, c_n = f(n)$.
  - În memoria internă elementele unui vector vor ocupa în ordine locaţii succesive de memorie ⇒ **reprezentare secvenţială**.
    - **Detaliu de implementare** - spaţiul de memorie necesar stocării elementelor vectorului se poate aloca dinamic, în timpul execuţiei programului.
  - Identificarea elementelor se face cu ajutorul *indicilor*.
- Un tablou este static: nu pot fi inserate sau şterse elemente(celule).
- Tablourile sunt foarte mult folosite în programare şi pentru reprezentarea altor structuri de date ⇒ pentru tablourile statice limbajele de programare includ notaţii specifice.
- Memorare a tablourilor
  - secvenţial (ordine lexicografică a indicilor)
  - înlănţuit (cazul matricilor rare)