



Ce este STL (Standard Template Library) ?

Autor: Prof. Tudor Sorin

Creatorul limbajului C++, Bjarne Stroustrup, și-a pus problema să adauge limbajului anumite elemente astfel încât scrierea programelor să se facă **rapid**¹, **standardizat**² și programele obținute să fie **performante**³.

Atunci când scriem programe, utilizăm două noțiuni fundamentale: **algoritmi** și **structuri de date**. De aici, rezultă că:

1. Limbajul trebuie să conțină anumite clase care să gestioneze structurile de date frecvent utilizate. Astfel, vom avea clase specializate în gestionarea vectorilor, listelor liniare, stivelor, cozilor, heap-urilor, etc. Astfel de clase se numesc **clase de tip container**, iar obiectele generate prin instanțierea lor se numesc **containere**. Prin urmare, o structură de tip listă liniară este un container, o structură de tip stivă este un container, ș.a.m.d.
2. Întrucât o structură de date trebuie să opereze cu date de orice tip, clasele de tip container utilizează din plin **mecanismul template**. Astfel, la modul general, *un container se definește ca un obiect care poate reține alte obiecte*.
3. Așa cum am văzut, structurile de date nu sunt suficiente pentru realizarea programelor. Din acest motiv, este necesar să avem anumite funcții care "incorporează" algoritmi cei mai des folosiți. Funcțiile care incorporează algoritmi, trebuie să poată lucra cu orice tip de dată (la modul general cu orice obiect). Cerința este satisfăcută dacă aceste funcții utilizează și ele mecanismul template. De asemenea, este foarte important ca aceste funcții să poată lucra cu datele stocate în containere.

¹ Un program se realizează cu mult mai repede dacă limbajul dispune de funcții și clase, care pot fi adaptate ușor pentru rezolvarea unor probleme concrete, cerute de practică.

² Scrierea programelor nu mai este de mult o activitate "artizanală". Este esențial ca un program să poată fi înțeles și de alți programatori, nu numai de cei care l-au scris. De exemplu, o firmă cu acest profil, trebuie să-și poată ușor modifica programele, chiar dacă unii dintre cei care le-au realizat nu mai lucrează acolo.

³ Pentru a obține programe performante prin utilizarea funcțiilor și claselor puse la dispoziție de limbaj, este esențial ca acestea din urmă să aibă, la rândul lor, la bază algoritmi performanți.

CE ESTE STL (STANDARD TEMPLATE LIBRARY) ?

Ca terminologie, vom utiliza termenul “**algorithm**” și cu semnificația de *funcție template care rezolvă o anumită problemă des întâlnită în activitatea curentă și care, de regulă, prelucrează date (obiecte) stocate în containere*.

4. O problemă foarte importantă este dată de accesul la datele (obiectele) aflate în container. Dacă la un container de tip vector, accesul se poate realiza ușor, el este mai dificil la un container de tip listă liniară sau la unul de tip de arbore de căutare. Din acest motiv, s-au generalizat pointerii și s-a ajuns la **iteratori**. Un iterator indică un anumit obiect din container. Pornind de la el se pot accesa datele și, dacă este cazul, metodele obiectului memorat. De asemenea, se poate obține imediat, printr-o simplă incrementare, un iterator care indică obiectul următor din container.

FOARTE IMPORTANT. Pentru a accesa datele stocate în containere, algoritmi utilizează iteratorii. Acesta este motivul pentru care se spune că “*iteratorii reprezintă un liant între containere și algoritmi*”.

⇒ Prin **STL (Standard Template Library)** înțelegem o colecție de clase de tip container, algoritmi și iteratori care poate fi folosită pentru a realiza rapid programe performante și standardizate. Orice compilator actual de C++ trebuie să permită utilizarea STL-ului⁴. Prin folosirea STL-ului, obținem programe portabile (pot fi compilate, teoretic, cu orice compilator de C++) și, mai ales, ușor de descifrat pentru oricine este familiarizat cu acest mod de lucru⁵.

Învățați să vă autodocumentați! Prezintă o adresă la care puteți găsi o documentație pentru **STL**:

“http://www.sgi.com/tech/stl/table_of_contents.html”.

✓ După cum ați văzut, containerele conțin obiecte. De cele mai multe ori, acestea sunt de tipuri simple, predefinite, cum ar fi: `int`, `float`, `char`... Mai mult, aceste obiecte sunt supuse unor operații de comparare. De asemenea, anumite containere sunt de tip vector, listă liniară, ș.a.m.d. Din acest motiv, vom folosi uneori termenii de dată, valoare pentru obiecte sau de componentă în cazul vectorului, element în cazul listelor (înțelegând prin aceasta, atât obiectul memorat, cât și informația de adresă). Sperăm ca astfel de exprimări să nu fie considerate o “crimă”...

⁴ În varianta Borland C++, STL-ul nu poate fi folosit decât parțial. Atunci când a fost realizat compilatorul (la modul general, mediul de programare), STL-ul era încă în faza de proiectare. Acesta este unul dintre motivele pentru care recomandăm utilizarea compilatoarelor prezentate în prima parte a cărții.

⁵ Elevii sunt avantajați dacă utilizează STL-ul în concursurile școlare. Teoretic, nimeni nu-i poate opri să-l utilizeze, este un standard. Totuși, utilizarea STL-ului nu-i scutește de pregătirea teoretică. Se poate spune că, între doi “olimpici” cu aceleași cunoștințe, cel care cunoaște STL-ul are un anumit avantaj.

Mai mult despre containere

⇒ **Containere.** Un container este un obiect care reține alte obiecte. El rezultă prin instanțierea unei clase de tip container. Containerele sunt de două mari categorii: **containere secvență** și **containere asociative**. De asemenea, există o categorie aparte de containere, numite **adaptori**.

A) Containere secvență - rețin obiectele într-o ordine oarecare. Ordinea obiectelor este stabilită de cel care folosește containerul respectiv. Containerele de tip secvență permit inserarea și ștergerea obiectelor.

Exemple de clase ale căror obiecte sunt containere de tip secvență: **vector** (gestionează un vector unde fiecare componentă reține un obiect) și **list** (gestionează o listă liniară dublu înlănțuită, unde fiecare element al listei reține un obiect).

B) Containere asociative - fiecare obiect are o **cheie** și poate fi regăsit rapid după aceasta. Regăsirea, ștergerea și inserarea unui obiect se fac rapid, în $O(\log n)$, unde n este numărul obiectelor care se găsesc în container. Pentru astfel de containere se preferă memorarea obiectelor sub **forma unor arbori de căutare echilibrați**, motiv pentru care operațiile amintite se efectuează atât de eficient. În mod evident, inserarea unui obiect se face automat, acolo unde îi este locul, conform cheii pe care o are, nu acolo unde dorește programatorul, ca în cazul containerelor secvență.

Exemple de clase ale căror obiecte utilizează containere asociative: **set** gestionează o mulțime de obiecte, unde fiecare obiect coincide cu cheia sa, unică; **map** gestionează mai multe perechi de forma (**cheie, valoare**), unde cheia este unică; **multiset**, la fel ca **set**, dar pot exista mai multe obiecte cu aceeași cheie; **multimap**, la fel ca **map**, dar pot exista mai multe perechi cu aceeași cheie.

C) Adaptori - sunt tot containere, dar nu sunt definiți separat, ci pornind de la alte containere se restricționează accesul numai la anumite metode. De exemplu, dacă dorim să lucrăm cu o stivă, se poate folosi un vector cu acces numai la unul dintre capete.

Exemple de adaptori: **stack** (stivă), **queue** (coadă), **priority_queue** (coadă cu prioritate sau, așa cum este cunoscut din cursul de algoritmi, structura **heap**).

NOTAȚII STANDARD. Atunci când prezentăm containerele, metodele de acces la obiectele memorate, utilizăm următoarele notații:

<code>size_type</code>	un tip întreg fără semn
<code>key_type</code>	tipul obiectului care constituie cheia
<code>value_type</code>	tipul obiectului memorat - în cazul obiectelor de tip map și multimap , este alcătuit din cheie și valoare

Mai mult despre iteratori

⇒ **Iteratori.** Pentru a se accesa cu ușurință obiectele unui container (“a naviga” printre ele) se utilizează **iteratorii**. Iteratorii sunt asemănători pointer-ilor, dar cu ei se pot face mult mai multe lucruri. Dacă în cazul containerelor de tip secvență, iteratorii sunt aproape la fel ca pointerii, în cazul containerelor asociative, aceștia permit “navigarea” ușoară, printr-o simplă incrementare, printre nodurile unui arbore de căutare.

DE REȚINUT:

- 1) Iteratorii sunt obiecte ale unor clase.
- 2) Dacă `it` este o variabilă care reține un iterator care indică un anumit obiect al containerului, atunci:
 - prin `*it` se adresează obiectul;
 - prin `it->metoda` / `it->data_membre` se adresează o metodă / dată membru a obiectului memorat;
 - prin `++it` se obține adresa obiectului “următor”. Dacă semnificația incrementării este ușor de explicat în cazul containerelor secvențiale (`++it` va indica obiectul următor în secvență), în cazul containerelor asociative semnificația poate fi de a indica următorul obiect cu cheia “mai mare” sau “mai mică”⁶.
- 3) Prin convenție, s-a stabilit că toate containerele secvențiale și asociative să aibă un **prim element** și un **ultim element**. Acesta din urmă nu conține niciun obiect, el este folosit pentru a marca sfârșitul elementelor din container. Nu există ca în cazul pointer-ilor un iterator nul, există numai un iterator care indică ultimul element din structură.
- 4) Din punct de vedere al sensului de parcurgere a obiectelor din container, iteratorii sunt de două feluri:
 - a) **iterator** - prin aplicarea repetată a operatorului “++” se parcurg obiectele în ordinea stabilită, cea secvențială, în cazul containerelor secvențiale și cea dată de relația de ordine pe mulțimea cheilor, în cazul containerelor asociative;
 - b) **reverse_iterator** - prin aplicarea repetată a operatorului “++” se parcurg obiectele în ordinea inversă impusă de secvență, în cazul containerelor secvențiale și ordinea inversă relației de ordine pe mulțimea cheilor, în cazul containerelor asociative.

⁶ În cazul tipurilor numerice, semnificația lui “mai mare” sau “mai mic” este evidentă. În cazurile altor tipuri, semnificația lui “mai mare” sau “mai mic” trebuie precizată (programată). Pe parcursul lecțiilor următoare, veți înțelege și cum se realizează aceasta.

CE ESTE STL (STANDARD TEMPLATE LIBRARY) ?

5) Clasele care utilizează containeri conțin unele sau toate metodele de mai jos:

<code>iterator begin()</code>	returnează un <code>iterator</code> care indică primul obiect al containerului
<code>iterator end()</code>	returnează un <code>iterator</code> care indică ultimul element al containerului
<code>reverse_iterator rbegin()</code>	returnează un <code>reverse_iterator</code> care indică primul obiect al containerului, dacă le considerăm pe acestea în ordine inversă
<code>reverse_iterator rend()</code>	returnează un <code>reverse_iterator</code> care indică ultimul element al containerului dacă le considerăm pe acestea în ordine inversă

6) Declararea unei variabile care reține un iterator se face prin utilizarea operatorului de rezoluție, ca mai jos:

- a) `nume_clasa_container::iterator nume_iterator;`
- b) `nume_clasa_container::reverse_iterator nume_iterator;`

7) Parcurgerea cu ajutorul iteratorilor a obiectelor dintr-un container, se poate face ca mai jos:

```
for(it=ob_container.begin();it!=ob_container.end();it++)
... // unde it este iterator

for(it=ob_container.rbegin();it!=ob_container.rend();it++)
... // unde it este reverse_iterator
```

8) În funcție de operațiile care se pot efectua cu ei, principal, iteratorii se pot împărți în următoarele categorii:

a) Iteratori de tip Input - permit accesul la obiectele din container. Operațiile permise cu aceștia sunt: ++ (incrementare, pentru a indica următorul obiect), `=*p` (citiri de obiecte din container), `->` (accesul la datele sau metodele obiectului indicat de iterator), `==` și `!=` (testarea egalității sau a inegalității a doi iteratori).

b) Iteratori de tip Output - permit memorarea obiectelor în container. Operațiile permise cu aceștia sunt: ++, `*p=` (atribuiri de obiecte în container).

c) Iteratori de tip Forward - permit operațiile care se pot efectua cu iteratorii de tip **Input** și **Output**.

d) Iteratori de tip Bidirectional - permit operațiile care se pot efectua cu iteratorii de tip **Forward** și, în plus, este permisă și operația -- (se obține un iterator care indică obiectul care precede în secvență obiectul indicat de iteratorul memorat).

e) Iteratori de tip Random - permit operațiile care se pot efectua cu iteratorii de tip **Bidirectional**, dar, în plus, se permit operații genul `iterator+k`, unde `k` este un întreg, adică se poate accesa direct un obiect al containerului, fără a fi necesar să parcurgem toate elementele care îl preced. De asemenea, accesul direct la obiecte se poate obține prin supraîncărcarea operatorului `[]`.

În final, menționăm că diversele containere permit lucrul cu o anumită categorie de iteratori. De asemenea și algoritmi lucrează cu anumiți iteratori.

Un exemplu mult simplificat pentru înțelegerea noțiunilor de container, iterator și algoritm

Această lecție este facultativă și se adresează doar acelor elevi care vor să înțeleagă principiile care stau la baza programării prin utilizarea STL-ului. În ea, vom prezenta o clasă, **de tip container**, care permite lucrul cu vectori. Clasa o vom numi **vectoras<T>**. Pentru a putea lucra cu vectori, vom utiliza **iteratori** obținuți prin instanțierea clasei **iterator**. De asemenea, se va da un exemplu de **algoritm** pentru a vedea, în linii mari, cum au fost aceștia obținuți.

Menționăm că, în vederea atingerii scopului propus, vom utiliza procedee cu mult simplificate față de cele care au stat la baza obținerii STL-ului. Totuși, parcurgerea acestei lecții constituie și o recapitulare a cunoștințelor însușite în lecțiile precedente.

Începem prin a prezenta sursa clasei **vectoras<T>**:

```
template <class T> class vectoras
{ // numarul de componente
  int n;
  // pointer catre vectorul alocat dinamic
  T* h;
public:
  // clasa interioara ce are rolul de a genera obiectele iterator
  class iterator
  { T* p;
  public:
    iterator() {p=0;}
    iterator(T* adr) {p=adr;};
    T& operator*() {return *p;}
    void operator++()
    { p++; }
    void operator=(iterator it1) {p=it1.p;}
    int operator!=(iterator it1) {return p!=it1.p;}
  };
  // metoda care returneaza un iterator catre prima componenta a
  // vectorului
  iterator begin();

  // metoda care returneaza un iterator catre ultima componenta a
  // vectorului
  iterator end();

  // constructorul clasei
  vectoras(int n);

  // metoda de acces la componentele vectorului
  T& operator[](int k);

  // metoda de adaugare a unei componente la sfarsitul unui vector
  void push_back (T x);
};
```

CE ESTE STL (STANDARD TEMPLATE LIBRARY) ?

```
template <class T> vectoras<T>::vectoras(int n)
{ this->n=n;
  h=new T[n+1];
}

template <class T> T& vectoras<T>::operator[](int k)
{ return h[k]; }

template <class T> typename vectoras<T>::iterator
vectoras<T>::begin()
{ return h; }

template <class T> typename vectoras<T>::iterator
vectoras<T>::end()
{ return h+n; }

template <class T> void vectoras<T>::push_back(T x)
{
  T* adr=new T[n+1];
  int i;
  for (i=0;i<n;i++) adr[i]=h[i];
  adr[n]=x;
  delete[] h;
  n++;
  h=adr;
}
```

Să analizăm clasa prezentată!

1. Cerința este ca vectorul să poată conține componente de orice tip. Din acest motiv, clasa este una de tip **template**, iar componentele vectorului au tipul formal **T**.
2. O altă cerință este ca numărul de componente să fie cunoscut în timpul executării programului. De aici, rezultă că este necesar ca vectorul să fie alocat dinamic. Prin urmare, clasa va conține data membru **h**, care reține un pointer către prima componentă a vectorului. De asemenea, data membru **n** va reține numărul de componente ale vectorului. Alocarea vectorului în memorie și inițializarea datelor membru este făcută de către constructorul clasei:

```
template <class T> vectoras<T>::vectoras(int n)
{
  this->n=n;
  h=new T[n+1];
}
```

3. Pentru a accesa componentele vectorului în mod clasic, așa cum suntem obișnuiți, vom supraîncărca operatorul "[]" astfel încât metoda rezultată să întoarcă o referință către componenta de indice **k**.

```
template <class T> T& vectoras<T>::operator[](int k)
{ return h[k]; }
```

Din acest moment, se pot scrie deja primele programe care permit lucrul cu vectori.

CE ESTE STL (STANDARD TEMPLATE LIBRARY) ?

Exemplu. Iată secvențele prin care se citesc și se afișează doi vectori. Unul cu componente de tip `float` (stânga), altul cu componente de tip `char` (dreapta):

<pre>vectoras<float> v(5); int i; for (i=0;i<5;i++) cin>>v[i]; for (i=0;i<5;i++) cout<<v[i];</pre>	<pre>vectoras<char> v(5); int i; for (i=0;i<5;i++) cin>>v[i]; for (i=0;i<5;i++) cout<<v[i];</pre>
--	---

4. Clasa trebuie înzestrată cu iteratori. Ce sunt ei? Niște pointeri, dar mai evoluți. E drept, efectul utilizării iteratorilor este evident în cazul altor structuri, mai puțin cea de vector. Oricum, pentru simplitate, am preferat să exemplificăm crearea acestora pentru vectori.

Ce știm despre iteratori?

- **Ei sunt obiecte** - prin urmare, vom considera clasa `iterator`, care este înzestrată cu un constructor fără parametri. Clasa `iterator` este inclusă în clasa `vectoras<T>`. Iată-o:

```
class iterator
{ T* p;
public:
    iterator() { p=0; }
    iterator(T* adr) { p=adr; };
    T& operator*() { return *p; }
    void operator++() { p++; }
    void operator=(iterator it1) { p=it1.p; }
    int operator!=(iterator it1) { return p!=it1.p; }
};
```

- **Un iterator indică un anumit obiect al containerului.** În cazul de față, iteratorul va indica o anumită componentă a vectorului. Prin urmare, clasa va avea ca dată membru un pointer către tipul formal `T`, adică tipul elementelor vectorului, numit `p`. Constructorul inițializează cu 0 data membru.

- În vederea parcurgerii elementelor structurii, **variabilele de tip iterator se pot incrementa.** Semnificația incrementării este de a indica adresa următorului obiect al containerului. În cazul de față, semnificația este de a indica următoarea componentă a vectorului. Pentru a fi permisă incrementarea, clasa conține o metodă care supraîncarcă operatorul `++`. În fapt, ea va incrementa pointer-ul din data membru `p`, pointer ce reține adresa unei componente.

- **Iteratorii suportă comparări de genul `!=` (inegal).** Din acest motiv, clasa conține o metodă care supraîncarcă operatorul `!=` și returnează o valoare diferită de 0 în cazul în care iteratorul obiectului curent nu este egal cu iteratorul transmis ca parametru. În fapt, se compară conținuturile datelor membru, `p`.

- **Între variabilele care conțin iteratori sunt permise atribuiri.** Prin urmare, clasa va conține o metodă care supraîncarcă operatorul `=` și care permite copierea în obiectul curent a conținutului altui iterator.

- **Pornind de la un iterator, se poate adresa conținutul obiectului,** la fel cum pornind de la un pointer, se poate adresa conținutul unei variabile indicată de el.

CE ESTE STL (STANDARD TEMPLATE LIBRARY) ?

Pentru a realiza aceasta, `vectoras<T>` conține o clasă care supraîncarcă operatorul "*" și care întoarce o referință la obiectul (componenta) indicat de iterator.
- În plus, clasa `iterator` este înzestrată cu un **constructor de copiere**. Acesta va fi folosit, așa cum veți vedea, de alte metode ale clasei `vectoras<T>`. Practic, constructorul de copiere obține un obiect iterator pornind de la un pointer către un obiect al containerului.

5. Așa cum am învățat în lecția precedentă, clasa container conține, de regulă, două metode care returnează adresa primului obiect din container și adresa ultimului element (cel care nu reține nicio valoare). Pentru aceasta, containerul va avea două metode: `iterator begin()` și `iterator end()`. Prima returnează un iterator ce conține adresa primei componente a vectorului, conținutul datei membru `h`. A doua returnează adresa componentei `n+1` care, în mod convențional, nu conține nici un obiect. Să observăm faptul că cele două metode returnează un obiect de tip iterator, dar valoarea returnată este de fapt de tipul `T*`. Este posibil, deoarece am înzestrat clasa iterator cu un constructor de copiere.

```
template <class T> typename vectoras<T>::iterator
vectoras<T>::begin()
{ return h; }

template <class T> typename vectoras<T>::iterator
vectoras<T>::end()
{ return h+n; }
```

De acum, putem accesa obiectele prin intermediul iteratorilor pe care i-am construit.

Exemplu

Iată secvențele prin care, cu ajutorul iteratorilor, se citesc și se afișează doi vectori: unul, cu componente de tip `float` (stânga) și altul, cu componente de tip `char` (dreapta):

<pre>vectoras<float> v(5); vectoras<float>::iterator it; for (it=v.begin();it!=v.end(); ++it) cin>>*it; for (it=v.begin();it!=v.end(); ++it) cout<<*it<<" ";</pre>	<pre>vectoras<char> v(5); vectoras<char>::iterator it; for (it=v.begin();it!=v.end(); ++it) cin>>*it; for (it=v.begin();it!=v.end(); ++it) cout<<*it<<" ";</pre>
--	--

6. De regulă, clasele care generează containere secvențiale conțin o metodă prin care se poate adăuga la sfârșitul unui container un obiect. În cazul nostru, se poate adăuga la sfârșitul unui vector o componentă cu o anumită valoare. Metoda se numește `push_back()`. De fapt, vă dați seama, algoritmul constă în alocarea dinamică a unui vector cu `n+1` componente de tipul `T`, copierea valorilor primelor `n` componente în vectorul nou alocat, adăugarea noii valori, transmisă ca parametru, pe poziția `n+1`, eliberarea memoriei ocupate de vechiul vector, memorarea în `n` a noului număr al componentelor și în `h`, un pointer către noul vector.

CE ESTE STL (STANDARD TEMPLATE LIBRARY) ?

```
template <class T> void vectoras<T>::push_back(T x)
{ T* adr=new T[n+1];
  int i;
  for (i=0;i<n;i++) adr[i]=h[i];
  adr[n]=x;
  delete[] h;
  n++;
  h=adr;
}
```

lată și un exemplu de utilizare a acestei metode:

```
vectoras<int> v(5);
int i;
for (i=0;i<5;i++) v[i]=i+1;
v.push_back(6);
for (i=0;i<6;i++) cout<<v[i];
```

OBSERVAȚIE. Pentru vectori, deși prezentă, metoda este un exemplu de ineficiență. Desigur, nu se poate spune același lucru dacă am avea o aceeași metodă pentru o listă liniară dublu înlănțuită.

Rămâne să exemplificăm modul în care se realizează un algoritm. Algoritmul accesează elementele containerului (în exemplu, vectorul) prin intermediul iteratorilor. El este sub forma unei funcții template și calculează suma elementelor din container:

```
long suma(vectoras<int>::iterator st, vectoras<int>::iterator fin)
{ long s=0; vectoras<int>::iterator it;
  for(it=st;it!=fin;++it) s+=*it;
  return s;
}
```

lată și o secvență care utilizează algoritmul: se citește vectorul și se afișează suma elementelor:

```
vectoras<int> v(5);
vectoras<int>::iterator it;
for (it=v.begin();it!=v.end(); ++it) cin>>*it;
cout<<"Suma este "<<suma(v.begin(),v.end());
```

infobits.ro | Editura L&S INFO-MAT vă recomandă ...

“[Complemente de C++](#)”, autori: Tudor Sorin, Vlad Tudor (Huțanu)