STL în concursuri

Marius Andrei

În cadrul acestui articol vă vom prezenta una dintre facilitățile oferite de limbajul C++, care poate fi folosită în cadrul concursurilor de programare; este vorba despre Standard Template Library.

STL (***S**tandard **T**emplate **L**ibrary*) privit foarte simplu, din punctul de vedere al concurentului, este o colecție de structuri de date și algoritmi aplicați asupra acestora.

Evident că este vorba de mult mai mult decât atât, însă pentru scopul articolului de față considerăm că evidențierea acestei laturi este suficientă. Pentru scopurile în care vom folosi noi *STL*-ul nu este nevoie de cunoștințe avansate de programare orientată obiect.

STL-ul se bazează pe câteva *template*-uri (șabloane) care fac aplicabili toți algoritmii pentru orice tip de structuri de date.

Pentru a vă arăta importanța ***STL***-ului trebuie precizat de la început faptul că această librărie este standard. Toate exemplele au fost testate atât cu ***gcc***, cât și cu ***Visual C++***.

Așadar, librăria poate fi folosită în concursuri, atât în cele pentru liceu (olimpiade), cât și în cele studențești (***ACM***). Datorită faptului că librăria este relativ nouă, ea nu este disponibilă în ***Borland C++*** 3.1.

Prima informație de care avem nevoie pentru a putea utiliza ***STL*** sunt fișierele ce trebuie incluse. Includem mai mult decât ne este necesar, însă aceste *header*-e ne vor folosi mai încolo, în exemplele următoare.

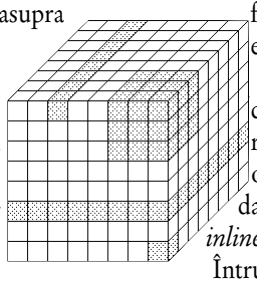
Veți observa că în partea de început a programului nu apare nimic anormal, în afară de faptul că nu toate *header*-ele au extensia *.h* și că *namespace*-ul utilizat trebuie să fie `std`:

```
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <algorithm>
#include <queue>
#include <assert.h>
using namespace std;
```

Vectori

Vom considera ca prim exemplu crearea unui vector și sortarea elementelor acestuia. Pentru a defini un vector care conține numere întregi se poate folosi construcția:

```
vector<int> a;
```



Nu există nici o limită asupra dimensiunii vectorului. Ea poate fi precizată de la început, însă nu este obligatoriu. De asemenea, putem preciza și o valoare inițială pentru elementele vectorului respectiv:

```
vector<int> a(50,1);
```

În exemplul anterior este definit un vector `a` care conține `50` de elemente care sunt inițializate cu valoarea `1`.

Accesarea unui element se realizează, așa cum suntem obișnuiți, folosind construcția `a[i]`, pentru al `i`-lea element, numerotarea fiind aceeași cu cea din *C/C++*, (începând de la indicele `0`).

Dacă vrem să adăugăm elementul 2 nu trebuie decât să scriem `a.push_back(2)`, și l-am adăugat la sfârșit.

Redimensionarea se realizează prin `a.resize(15)`. Astfel putem mări sau micșora lungimea vectorului.

Lungimea vectorului ne este tot timpul disponibilă printr-un apel de forma `a.size()`.

Sortarea vectorului se realizează folosind construcția:

```
sort(&a[0],&a[N]); //N reprezintă lungimea vectorului
```

Întrebarea evidentă aici se referă la motivul pentru care cineva ar folosi aceste clase, când poate face totul foarte simplu, folosind, de exemplu, `qsort`.

Primul lucru spectaculos este că sortarea rulează de până la 20 de ori mai repede, în special datorită implementărilor *inline*.

Întrucât uneori ne permitem de 20 de ori mai mult timp pentru o sortare, s-ar putea spune că nu se câștigă prea mult, dar mai e mult de învățat de la ***STL***.

Funcția `sort` are doi parametri: primul element și elementul imediat următor ultimului element.

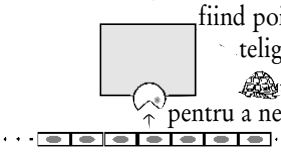
De obicei se scrie:

```
sort(a.begin(), a.end()),
```

însă pentru a înțelege această construcție trebuie să știm câte ceva despre iteratori.

Iteratori

Trebuie să ne gândim la iteratori ca fiind pointeri "inteligenți", folosiți pentru a ne deplasa printre elementele structurii noastre de date (pentru toate tipurile de structuri, nu doar pentru vectori).



De exemplu, declararea unui iterator pe o structură de tip vector de numere întregi se poate face cu ajutorul construcției:

```
vector<int>::interator i;.
```

Pentru aceste structuri de date există doi iteratori impliciți și anume `begin()` și `end()`, care permit obținerea unui iterator pentru primul element, respectiv pentru elementul de după ultimul element.

Pe parcurs vom prezenta mai multe detalii referitoare la folosirea iteratorilor.

Cozi de priorități

Declararea unei cozi de priorități (*heap*) se realizează cu ajutorul unei construcții de forma:

```
priority_queue<int> a;
```

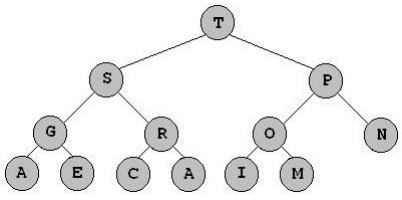
De obicei implementarea unei cozi de priorități se face cu ajutorul *heap*-urilor.

Introducerea de elemente într-o coadă de priorități se realizează folosind o construcție similară cu:

```
a.push(8);
```

Determinarea dimensiunii cozii de priorități se face cu ajutorul funcției `size()` specifică structurii de date:

```
int dim = a.size();
```



Determinarea valorii elementului cu cea mai mare prioritate (elementul din vârful *heap*-ului) se realizează cu ajutorul funcției `top()` specifică structurii de date:

```
int varf = a.top();
```

Extragerea elementului cu cea mai mare prioritate se realizează cu ajutorul funcției `pop()` specifică structurii de date:

```
a.pop();
```

Există susținători ai ideii potrivit căreia implementarea unui *heap* este imediată și este preferabil să avem control total asupra structurii de date pe care o implementăm.

Există argumente în favoarea acestei idei dar, din alt punct de vedere, ar putea fi de preferat să știm sigur că nu vom greși și că vom mai câștiga cîteva minute care se pot dovedi prețioase.

Tabele de dispersie

Cu ajutorul acestor tabele se reprezintă o relație 1 la 1 între două tipuri de date. Primul element este cheia, iar al doilea este valoarea. În exemplul următor vom asocia câte un număr întreg fiecărui oraș:

```
map<string,int> a;
a["Bucuresti"] = 10;
a["Cluj"] = 7;
a["Iasi"] = 3;
a["Constanta"] = 8;
```

Dacă dorim să tipărim numărul corespunzător *Clujului* vom scrie:

```
cout << a["Cluj"]
```

Efectul acestei construcții va consta în afișarea valorii `7` la ieșirea standard.

Dacă nu există cheia, se va afișa valoarea `0` (în general, se va lua în considerare constructorul implicit dacă este vorba de alt tip de dată decât număr întreg) .

Astfel, `a["Timisoara"]` va avea valoarea `0`.

Este interesant de menționat faptul că structura de date creată prin `map<int, int>` poate fi privită ca un vector de întregi infinit de lung (la ambele capete).

Deja începem să ne dăm seama de câteva dintre avantajele ***STL***-ului.

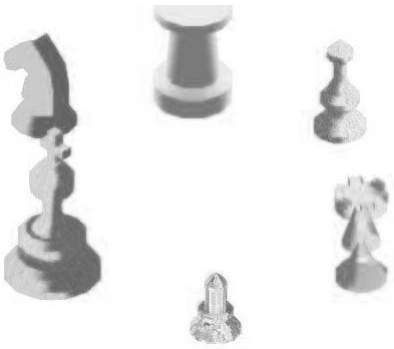
Mulțimi

Poate cel mai folositor lucru pentru un concurent sunt mulțimile. Ele se declară folosind o construcție de tipul:

```
set<int> a;
```

Operațiile de bază cu mulțimile sunt inserarea, ștergerea și căutarea de elemente care se pot realiza cu ajutorul funcțiilor `insert()`, `erase()` și `find()`:

```
a.insert(7);
a.erase(5);
if (a.find(7) != a.end())
  cout << Gasit;
```



Pe lângă aceste operații de bază, sunt definite intersecția, reuniunea și diferența.

Implementarea operațiilor cu mulțimi nu mai pare a fi foarte simplă, iar una cu arbori echilibrați în mod sigur nu e facilă, mai ales în momentul în care trebuie implementat algoritmul de ștergere a unui nod.

În plus, orice operație are ordinul de complexitate $O(\log N)$!

Pe lângă timpul câștigat (puțini se pot lăuda cu numai 20 de minute pentru a implementa un arbore *AVL* cu ștergeri), nu trebuie să ne facem griji că ar putea exista greșeli de implementare.

Folosirea iteratorilor

Pentru exemplificare vom parcurge elementele unei mulțimi și elementele unei tabele de dispersie.

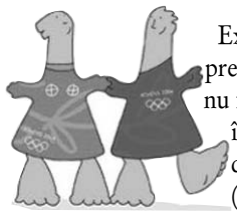
```
set<int> a;
map<string,string> b;

for(set<int>::iterator i =
   a.begin(); i != a.end(); ++i)
   cout << *i << endl;
for(map<string,string>::
        iterator i = b.begin();
             i != b.end(); ++i)
   cout << i->first << " => " <<
             i->second << endl;
```

Este necesar să folosim construcția "`!= a.end()`" și nu "`< a.end()`", deoarece datele nu sunt obligatoriu ordonate în memorie.

make_pair

Toate exemplele de până acum utilizează numere întregi. Așa cum am mai zis, putem pune orice în loc de întregi, orice structură sau clasă. Pentru aceasta, este suficient să definim operatorul "<".

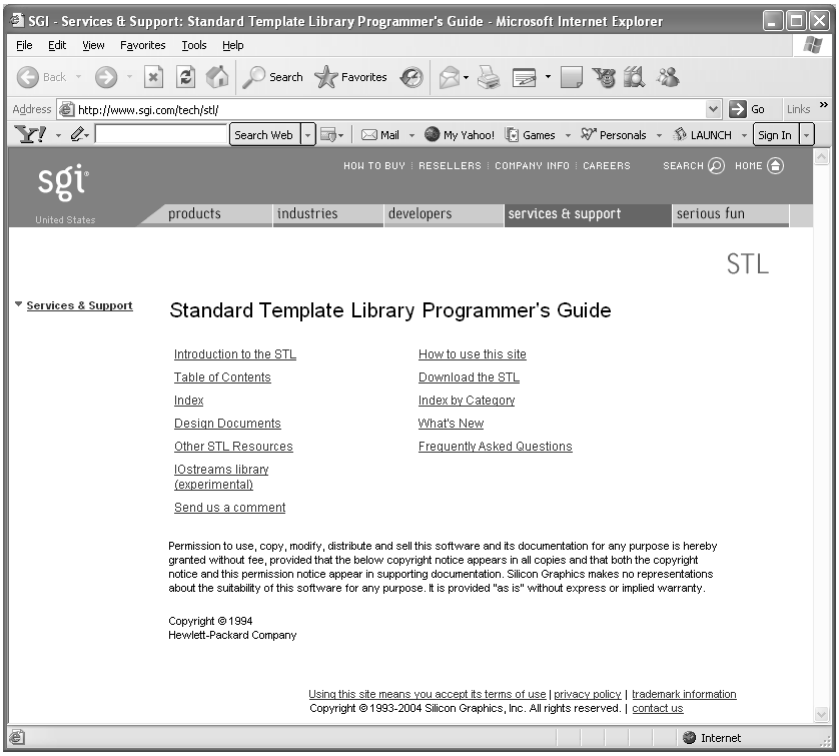
```
bool operator< (const MyStruct
          &a, const MyStruct &b)
{
   // returnează true dacă a<b,
   // false dacă a>=b
}
```



Există o clasă predefinită care nu face decât să încapsuleze două date (oricare două date) și anume clasa `pair`.

Ca exemplu, vom prezenta o secvență în care vom declara un vector ale cărui elemente sunt perechi de numere întregi și vom sorta elementele vectorului respectiv.

```
int N,x,y;
vector< pair<int,int> > a;
//este esențial să existe un spațiu
//între ">" și ">"!
cin >> N;
for(int i=0;i<N;++i) {
   cin >> x >> y;
   a.push_back(make_pair(x,y));
   // make_pair creează o clasă
   // pair<int,int>
}
sort(a.begin(),a.end());
//sortează după prima coordonată
//(primul întreg),
//iar în caz de egalitate după a doua
//coordonată.
```



Mult mai mult

Tot ce am prezentat este doar o mică parte din facilitățile oferite de ***STL***.

Pentru o prezentare completă vă recomandăm să accesați adresa **http://www.sgi.com/tech/stl/**.

Este clar că folosirea ***STL***-ului reprezintă un avantaj. Timpul de implementare este redus, timpul de rulare este mai mic, posibilatea apariției erorilor în structura noastră de date este inexistentă.

Unii concurenți ar putea să protesteze împotriva folosirii ***STL***-ului. Vor spune că nici nu mai trebuie să știi arbori ***AVL*** (sau arbori roșu-negru), nici *hash*-uri, nici măcar un *heap*. Vom deveni simplii utilizatori ai structurilor, în loc să le putem implementa de fiecare dată. Poate e adevărat, însă ***STL***-ul nu ne ajută să găsim metoda de rezolvare. Ne ajută doar să implementăm mai repede rezolvarea.

Dar în concursuri, față de cei care folosesc limbajul *Pascal*, *STL* oferă un avantaj fără discuții. Deși nu mai sunt foarte mulți programatori *Pascal*, aceștia există, mulți dintre ei fiind chiar foarte buni. Aceștia vor acuza folosirea ***STL***-ului și dintr-un anumit punct de vedere au dreptate. Problemele propuse sunt în general alese în așa fel încât să ofere șanse egale atât programatorilor în C++, cât și celor care programează în *Pascal*. Acum apare un avantaj în favoarea celor care aleg limbajul C++.

Totuși, nu putem interzice folosirea ***STL***-ului deoarece această librărie este un standard.

STL-ul este o evoluție, însă nici nu trebuie privit ca un element care trebuie obligatoriu cunoscut.

Pentru a-l utiliza în concursuri, este necesară o cunoaștere bună a librăriei, pentru că altfel apare riscul de a o folosi incorect sau de a pierde timp cu folosirea sa.

De asemenea, trebuie menționat faptul că depanarea programelor care folosesc ***STL*** este foarte dificilă, iar la compilare vor apărea o mulțime de avertismente legate de numele prea

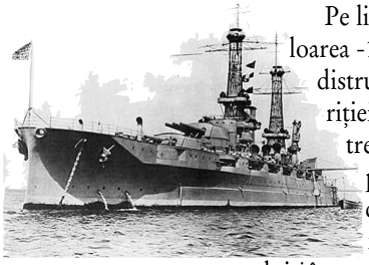
lungi ce vor fi trunchiate la 255 de caractere.

Un exemplu complet

La final vom prezenta un exemplu complet de folosire a ***STL***-ului.

Problema a fost propusă spre rezolvare în tabara de pregătire de la *Orăștie*. Enunțul și rezolvarea îi aparțin lui *Radu Berinde*.

Enunț



Pe mare va avea loc o mare bătălie între N vapoare. Vapoarele sunt considerate a fi puncte și sunt date prin coordonatele lor carteziene x și y.

Din motive greu de înțeles, vapoarele nu pot ataca decât vapoarele care se află la stânga și mai jos (mai exact, un vapor la poziția x_1, y_1 poate ataca alt vapor la poziția x_2, y_2 dacă și numai dacă $x_1 > x_2$ și $y_1 > y_2$).

Deoarece această bătălie are loc în zona *Triunghiului Bermudelor*, vapoarele apar (se teleportează) pe rând în zona bătăliei. Vapoarele sunt identificate prin numere întregi cuprinse între 1 și N, în ordinea apariției lor.

În momentul în care un vas apare, dacă există alt vas care a apărut deja și care poate să îl atace pe cel nou, vasul nou este distrus instantaneu. Dacă nu, vasul cel nou rămâne pe mare și distruge toate vasele pe care le poate ataca.

Dându-se coordonatele la care apar pe rând vapoarele, să se afle pentru fiecare vapor dacă este distrus sau nu în momentul apariției sale și dacă nu este distrus, să se precizeze numărul total de vapoare rămase pe mare după apariția sa.

Date de intrare

Pe prima linie a fișierului de intrare **`sea2.in`** se află numărul natural N reprezentând numărul de vapoare ce vor apărea pe mare.

Pe fiecare dintre următoarele N linii se află câte o pereche de numere întregi separate printr-un spațiu, reprezentând coordonata x respectiv y a vaporului corespunzător liniei (mai exact, pe linia i din fișier sunt scrise coordonatele vaporului identificat prin i - 1, pentru orice valoare i cuprinsă între 2 și N + 1).

Date de ieșire

Fișierul de ieșire **`sea2.out`** va conține N linii, pe fiecare dintre acestea aflându-se câte un număr întreg.

Pe linia i se va afla valoarea -1 dacă vaporul i este distrus în momentul apariției, sau un număr întreg strict pozitiv reprezentând numărul de vapoare de pe mare după apariția vaporului i în caz contrar.

Restricții și precizări

- $1 \le N \le 200.000$;
- coordonatele sunt numere întregi strict pozitive mai mici sau egale cu 260000;
- nu vor exista două vase cu aceeași coordonată x sau cu aceeași coordonată y.

Soluție

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <set>
#include <utility>
using namespace std;

set< pair<int,int> > Ships;
set< pair<int,int> >::iterator
                        It, It2, t;

int main() {
  FILE *fi = fopen("sea2.in",
                          "rt");
  FILE *fo = fopen("sea2.out",
                          "wt");
  int x, y, nr;

  for (fscanf(fi, "%d", &nr);
                    nr; nr--) {
    fscanf(fi, "%d %d", &x, &y);
    It = Ships.upper_bound(
              make_pair(x, y));
    if (It != Ships.end() &&
               It->second > y) {
      fprintf(fo, "-1\n");
      continue;
    }
    if (Ships.size()) {
      for (It2 = It;
       It2 != Ships.begin() &&
              (--(t = It2))->
                    second < y;
       It2 = t
      );
      Ships.erase(It2, It);
    }
    Ships.insert(
              make_pair(x, y));
    fprintf(fo, "%d\n",
                Ships.size());
  }
  fclose(fo);
  fclose(fi);
  return 0;
}
```