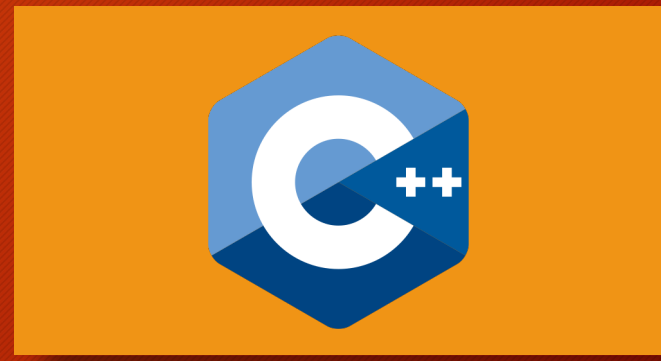
Introducere în STL



11 martie 2023

Iulian Oleniuc

Introducere în OOP

- Programarea Orientată pe Obiecte este o paradigmă de programare axată pe ideea de încapsulare a datelor.
- În acest sens, datele specifice unui anumit task sunt grupate într-o singură structură, împreună cu codul care operează asupra lor.
- Aceste structuri de date se numesc clase, iar variabilele de tip clasă (instanțele clasei) se numesc obiecte.
- Utilizarea claselor reprezintă un mod eficient de abstractizare a datelor, ducând la scrierea de cod lizibil, robust, ușor de menținut, bla bla.

Exemplu: Vectori dinamici

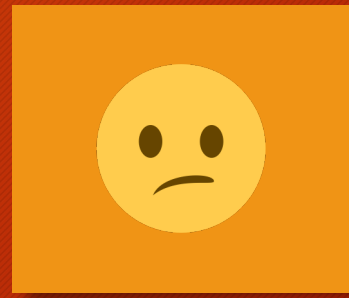
- Probabil cel mai clasic exemplu de structură de date care poate fi modelată mult mai ușor cu ajutorul claselor este vectorul (pseudo) dinamic.
- Concret, să presupunem că dorim să construim (și să afișăm) o listă cu toate numerele prime mai mici sau egale cu n.
- Practic, avem nevoie de un vector care inițial este vid și la finalul căruia adăugăm succesiv câte un element.

Varianta fără OOP

```cpp
#include <bits/stdc++.h>
using namespace std;

bool prime(int n) { ... }
const int MAX_SIZE = 10000;

int main() {
    int len = 0;
    int vec[MAX_SIZE];

    int n; cin >> n;
    for (int i = 1; i <= n; i++)
        if (prime(i))
            vec[len++] = i;
    for (int i = 0; i < len; i++)
        cout << vec[i] << ' ';
    cout << '\n';
    return 0;
}
```

- de fiecare dată când declarăm un astfel de vector, trebuie să declarăm și o variabilă care să-i stocheze lungimea curentă
- acest dezavantaj devine și mai problematic atunci când avem nevoie de vectori (statici sau nu) de vectori dinamici (exemplu: *liste de adiacență*)
- instrucțiunea `vec[len++] = i` nu este suficient de expresivă

Varianta C (pseudo-OOP)

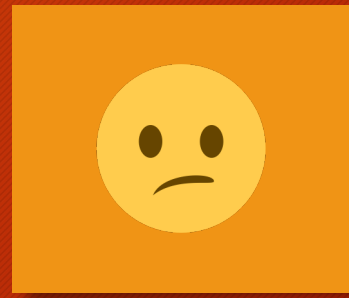


```cpp
#include <bits/stdc++.h>
using namespace std;

bool prime(int n) { ... }
const int MAX_SIZE = 10000;

struct Vector {
    int len = 0;
    int vec[MAX_SIZE];
};

int size(Vector& vec) {
    return vec.len;
}

int at(Vector& vec, int pos) {
    return vec.vec[pos];
}
```

```cpp
void pushBack(Vector& vec, int val) {
    vec.vec[vec.len++] = val;
}

int main() {
    int n; cin >> n;
    Vector vec;
    for (int i = 1; i <= n; i++)
        if (prime(i))
            pushBack(vec, i);
    for (int i = 0; i < size(vec); i++)
        cout << at(vec, i) << ' ';
    cout << '\n';
    return 0;
}
```

- datele asociate unui vector (`len` și `vec`) sunt separate de restul programului
- funcționalitățile specifice vectorilor dinamici sunt implementate în funcții cu nume sugestive

Varianta C (pseudo-OOP)



```c
#include <bits/stdc++.h>
using namespace std;

bool prime(int n) { ... }
const int MAX_SIZE = 10000;

struct Vector {
    int len = 0;
    int vec[MAX_SIZE];
};

int size(Vector& vec) {
    return vec.len;
}

int at(Vector& vec, int pos) {
    return vec.vec[pos];
}
```

```c
void pushBack(Vector& vec, int val) {
    vec.vec[vec.len++] = val;
}

int main() {
    int n; cin >> n;
    Vector vec;
    for (int i = 1; i <= n; i++)
        if (prime(i))
            pushBack(vec, i);
    for (int i = 0; i < size(vec); i++)
        cout << at(vec, i) << ' ';
    cout << '\n';
    return 0;
}
```

- fiecare apel de funcție începe cu adresa vectorului pe care lucrăm
- în implementarea acestor funcții, de fiecare dată când accesăm un câmp al vectorului, trebuie să scriem `vec.`
- câmpurile sunt publice, deci le putem modifica (**incorect!**) și fără să apelăm funcțiile specifice

Varianta C++ (OOP)

```
class Vector {
    int len = 0;
    int vec[MAX_SIZE];
public:
    int size() { return len; }
    int at(int pos) { return vec[pos]; }
    void pushBack(int val) { vec[len++] = val; }
};

int main() {
    int n; cin >> n;
    Vector vec;
    for (int i = 1; i <= n; i++)
        if (prime(i))
            vec.pushBack(i);
    for (int i = 0; i < vec.size(); i++)
        cout << vec.at(i) << ' ';
    cout << '\n';
    return 0;
}
```

- variabilele (proprietățile) și funcțiile (metodele) declarate în cadrul unei clase se numesc membri
- membrii unei clase pot fi privați (by default) sau publici, cei din urmă fiind accesibili din orice zonă a programului
- **în C++**, keyword-ul `struct` este perfect echivalent cu `class`, singura diferență fiind că membrii unui struct sunt în mod implicit publici, pe când membrii unei clase sunt privați

Constructori și destructori

- Constructorul este o metodă apelată automat atunci când un obiect este creat, având în general rolul de a-i inițializa variabilele membre.
- Similar, destructorul este apelat automat atunci când un obiect este distrus, adică atunci când iese din scopul său (dacă este variabilă locală) sau după execuția funcției `main` (dacă este variabilă globală).
- O clasă poate avea mai mulți constructori, în ideea că poate fi instanțiată în moduri diferite, însă nu poate avea mai mult de un destructor, căci destructorii sunt funcții fără parametri.

Exemple

```cpp
Vector(int len, int val = 0) {
    this->len = len;
    for (int i = 0; i < len; i++)
        vec[i] = val;
}

Vector(int len, int vec[]) {
    this->len = len;
    for (int i = 0; i < len; i++)
        this->vec[i] = vec[i];
}

int main() {
    int vec[] = {3, 1, 4, 6, 8};
    Vector vec1(4); // {0, 0, 0, 0}
    Vector vec2(3, 5); // {5, 5, 5}
    Vector vec3(3, vec); // {3, 1, 4}
    return 0;
}
```

- Primul constructor creează un vector de lungime `len` umplut cu valoarea `val`, care by default este `0`.
- Al doilea constructor inițializează vectorul cu primele `len` elemente din vectorul C-style `vec`.
- `this` este un pointer către obiectul curent.
- Dacă nu definim explicit un constructor, compilatorul creează unul default, cu zero parametri, care nu face nimic.
- Similar pentru destructor.

```cpp
~ofstream() {
    // ...
    close();
}
```

Exemplu: Alocare dinamică de memorie

```
class Vector {
    int len;
    int *vec;

public:
    Vector(int len, int val = 0) {
        this->len = len;
        vec = new int[len];
        for (int i = 0; i < len; i++)
            vec[i] = val;
    }

    ~Vector() {
        delete[] vec;
    }

    int size() { return len; }
    int& at(int pos) { return vec[pos]; }
};
```

- Implementarea de până acum a clasei `Vector` alocă vectorul intern `vec` în mod static, deci pe stivă (dacă obiectul este local).
- În general nu ne dorim asta, deoarece stiva are foarte puțină memorie la dispoziție (1-2 MB).
- Soluția este să-l alocăm dinamic pe `vec`, în zona de memorie heap, care este **mult** mai mare.
- Este foarte important să-l și *de*alocăm pe `vec`, iar acest lucru se face de obicei în destructor.

Supraîncărcarea operatorilor

```
Vector& operator+=(int val) {
    vec[len++] = val;
    return *this;
}

int& operator[](int pos) {
    return vec[pos];
}

int main() {
    Vector vec;
    vec += 3;
    vec += 1;
    vec += 4;
    vec[1] += 8;
    cout << vec[1] << '\n'; // 9
    return 0;
}
```

- Un feature C++ foarte util este faptul că ne permite să supraîncărcăm operatori, adică să le definim un comportament specific în funcție de tipul obiectelor la care se referă.
- Astfel, operatorii definiți de noi pot fi priviți pur și simplu drept funcții.
- Există niște reguli nescrise în legătură cu anumiți operatori; de exemplu, cei de asignare (compuși sau nu) trebuie să returneze adresa obiectului curent, pentru a permite expresii de genul `a = b = c = 3`.
- Exemplu complex: clasă pentru numere mari.

Template-uri

- Mulți algoritmi și structuri de date rămân neschimbați indiferent de tipul de date pe care îl folosesc.
- De exemplu, sortarea unui vector de întregi nu diferă practic cu nimic de sortarea unui vector de string-uri.
- Similar, o clasă care stochează un vector de întregi nu ar trebui să difere semnificativ de una care stochează un vector de string-uri.
- Din acest motiv, în C++ putem template-iza funcții/clase, pentru a le putea refolosi pentru diverse tipuri de date.

Template funcție

```cpp
template<class T>
bool prime(T n) {
    if (n < 2)
        return false;
    for (T d = 2; d * d <= n; d++)
        if (n % d == 0)
            return false;
    return true;
}

int main() {
    cout << prime(7) << '\n';
    cout << prime(618618618618LL) << '\n';
    return 0;
}
```

- Această funcție testează dacă un număr este prim și funcționează la fel de bine atât atunci când tipul `T` al lui `n` este `int`, cât și atunci când este `long long int`.

Template clasă

```cpp
template<class T>
class Vector {
    int len;
    T *vec;

public:
    Vector(int len, T val = T()) {
        this->len = len;
        vec = new T[len];
        for (int i = 0; i < len; i++)
            vec[i] = val;
    }

    ~Vector() { delete[] vec; }
    int size() { return len; }
    T& at(int pos) { return vec[pos]; }
};

int main() {
    Vector<int> vecInt(13);
    Vector<string> vecStr(7, "InfoGym");
    return 0;
}
```

- De data aceasta, clasa `Vector` poate stoca vectori de orice tip.
- Expresia `T()` returnează un nou obiect de tipul `T`, inițializat folosind constructorul său default.
- Chiar dacă, tehnic, tipurile de genul `int` și `double` nu sunt clase, expresiile `int()` și `double()` sunt valide și returnează valoarea `0`.

Standard Template Library (STL)

- C++ ne pune la dispoziție biblioteca STL, care implementează în mod generic o grămadă de algoritmi și structuri de date clasice.
- Aceste clase sunt foarte vaste și este aproape imposibil să cunoaștem toate detaliile despre ele, așa că este normal (chiar recomandat) să fie folosite cu documentația în față (link-uri în paragraful precedent) – lucru permis la orice concurs.
- Totuși, STL-ul este foarte consistent în denumiri (funcția care returnează numărul de elemente dintr-un container se numește întotdeauna `size`) și „gândire” (intervalele cu care se lucrează sunt mereu închise la stânga și deschise la dreapta).

Clasa `vector`

- Clasa vector este de departe cea mai folosită componentă a STL-ului.
- Este folosită pentru stocarea de vectori alocați dinamic, de data asta în adevăratul sens al cuvântului.
- În acest sens, vectorul își modifică reala dimensiune de fiecare dată (nu chiar…) când adăugăm sau eliminăm elementul de la final.
- Astfel, vectorul ocupă exact atât spațiu cât are nevoie (de fapt cel mult de două ori mai mult).

Iterarea unui `vector`

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n; cin >> n;
    vector<int> v(n);
    for (int i = 0; i < int(v.size()); i++)
        cin >> v[i];
    for (int i = 0; i < int(v.size()); i++)
        cout << v[i] << '\n';
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
        cout << *it << '\n';
    for (auto it = v.begin(); it != v.end(); it++)
        cout << *it << '\n';
    for (int i : v)
        cout << i << '\n';
    for (int& i : v)
        cin >> i;
    return 0;
}
```

- Elementele vectorului pot fi accesate, atât pentru citire cât și pentru modificare, prin indexul lor.
- O altă metodă folosește niște iteratori care imită pointerii clasici.
- Iteratorii pentru începutul și sfârșitul vectorului (+1) sunt dați de metodele `begin` și `end`.
- Sintaxa cu `:` se numește range-based for și este doar o simplificare a versiunii cu iteratori.

Alte metode ale clasei `vector`

- `front()` și `back()` returnează o adresă (**nu iterator!**) către primul/ ultimul element din vector
- `rbegin()` și `rend()` returnează iteratori către primul/ ultimul (+1) element din șirul obținut prin citirea în sens invers a vectorului
- `empty()` returnează `true` dacă vectorul este gol
- `clear()` golește conținutul vectorului
- `capacity()` returnează numărul de elemente alocate de vector

Alte metode ale clasei `vector`

- `resize(size, val)` schimbă dimensiunea vectorului, eventualele elemente noi fiind inițializate cu `val`
- `reserve(cap)` schimbă capacitatea vectorului (alocă din avans elemente pentru a optimiza apelurile la `push_back`)
- `push_back(val)` adaugă `val` la finalul vectorului
- `pop_back()` elimină elementul de la finalul vectorului
- `emplace_back(...)` adaugă `T(...)` la finalul vectorului

Există *matrice* STL?

- Da, putem scrie `vector<vector<int>>`.
- În C++17, compilatorul va deduce automat tipul template-ului din constructor.
- În general este mult mai comod și eficient să folosim vectori din STL pentru tablouri:
- Îi putem declara local fără să ne facem probleme în legătură cu stiva, folosesc exact atâtă memorie cât au nevoie, au elementele inițializate automat cu zero (sau cu ce valoare dorim) etc.

```
int m, n; cin >> m >> n;
vector<vector<int>> mat1(m + 1, vector<int>(n + 1));
vector mat2(m + 1, vector<int>(n + 1)); // C++17
```

Când sunt realocați vectorii din STL?

- **Mit:** Vectorii din STL sunt nașpa. Pot depăși limita de memorie a problemei, alocând mai multe elemente decât e necesar. În plus, sunt foarte ineficienți, pentru că unele `push_back`-uri nu se efectuează în `O(1)`.
- **Realitate:** Un apel la `push_back` se efectuează în `O(1)` dacă `size < capacity`. Altfel, se va produce o realocare. Capacitatea se **dublează**, conținutul vechiului vector este copiat în cel nou, iar la final se adaugă și elementul nou.

Când sunt realocați vectorii din STL?

- Practic, dacă începem cu un vector vid și facem `n` `push_back`-uri, se vor produce doar `O(log n)` realocări, iar timpul consumat de acestea va fi:
- `1 + 2 + 4 + … + 2^(log n) = O(n)`
- Deci, complexitatea nu diferă de cea a unei implementări obișnuite.
- Vectorii din STL sunt cum nu se poate mai eficienți!

Clasa `stack` (stivă)

- `top()`
- `pop()`
- `push()`

Clasa `queue` (coadă)

- `front()` primul element adăugat în coadă
- `back()` ultimul element adăugat în coadă
- `push()`
- `pop()`
- **exemplu:** click pe C++17

Clasa deque

- deque = **d**ouble **e**nded **que**ue
- Deque-ul este o coadă cu două capete, deci putem adăuga sau șterge elemente atât de la începutul cât și de la sfârșitul ei, în `O(1)`.
- `operator[](index)`
- `push_back()`
- `push_front()`
- `pop_back()`
- `pop_front()`

Clasa `set`

- Clasa `set` modelează o mulțime (în sensul matematic) dinamică de elemente de același tip.
- Elementele set-ului sunt ținute în ordine crescătoare, în funcție de operatorul `<` al tipului respectiv (dacă este vorba de o clasă definită de noi, atunci trebuie să-i supraîncărcăm operatorul `<`).
- Funcțiile prezentate în continuare rulează în timp logaritmic.
- `insert(val)`
- `erase(val)`
- `count(val)` returnează frecvența lui `val` în set (deci `0` sau `1`)

Clasa `set`

- `lower_bound(val)` returnează un iterator către primul element `>= val`
- `upper_bound(val)` returnează un iterator către primul element `> val`
- `1 2 2 3 3 3 4 6 6 _` `[lower_bound(2), upper_bound(2)]`
- `1 2 2 3 3 3 4 6 6 _` `[lower_bound(4), upper_bound(4)]`
- `1 2 2 3 3 3 4 6 6 _` `[lower_bound(5), upper_bound(5)]`
- `1 2 2 3 3 3 4 6 6 _` `[lower_bound(7), upper_bound(7)]`

Clasa `map`

- Este o extensie a `set`-ului, în sensul că fiecărui element din mulțime (numit cheie) îi este asociată o valoare.
- Practic, `map<K, V>` este un „vector” cu elemente de tipul `V` și indecși de tipul `K`.
- `operator[](key)` returnează sau modifică valoarea asociată cheii `key`
- Dacă, atunci când încercăm să accesăm această cheie, ea nu există în map, atunci este adăugată imediat, fiindu-i asociată valoarea `V()`.

Iterarea unui `map`

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    map<string, int> id;
    int n; cin >> n;
    for (int i = 0; i < n; i++) {
        string str; cin >> str;
        if (!id.count(str))
            id[str] = id.size() + 1;
    }
    for (auto it : id)
        cout << it.first << ' ' << it.second << '\n';
    for (auto it : id) {
        string str; int num; tie(str, num) = it;
        cout << str << ' ' << num << '\n';
    }
    for (auto [str, num] : id)
        cout << str << ' ' << num << '\n';
    return 0;
}
```

- În cazul `map`-ului, iterarea este mai interesantă, deoarece avem de a face cu perechi de forma `(key, val)`.
- Aceste perechi sunt chiar obiecte de tipul `pair<K, V>`.
- Sintaxa `[...]` se numește destructuring și este validă doar din C++17.

Clasa `pair`

- Stochează perechi de obiecte de tipuri eventual diferite.
- Are două variabile membre publice: `first` și `second`.
- Generalizare: `tuple`.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<double> v = {1.2, 3.14, 1.618, 1.3, 2};
    pair<double, int> mx;
    for (int i = 0; i < 5; i++)
        mx = max(mx, make_pair(v[i], i));
    cout << mx.first << ' ' << mx.second << '\n';
    return 0;
}
```

Clasa `string`

- Suportă toate metodele clasei `vector`, fiind practic un `vector` de caractere, plus următoarele:
- `operator+(str)`
- `operator+=(str)`
- operatori relaționali
- `c_str()` returnează un pointer către string-ul C-style echivalent
- `substr(pos, len)`

Biblioteca `algorithm`

- Conține funcții pentru algoritmi generici, printre care bine-cunoscutele `min`, `max` și `swap`.
- Majoritatea acestor algoritmi au nevoie să compare obiecte ale tipului folosit, dar sunt implementați în așa fel încât să folosească doar operatorul `<`.
- Dar, chiar dacă acesta este deja definit, uneori vrem să folosim alt criteriu de comparație (de exemplu la sortare).
- Prin urmare, aceste funcții au întotdeauna un parametru opțional la final, care este o funcție de comparare.

Funcția `sort`

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int v[] = {6, 1, 8, 3, 1, 4};
    sort(v, v + 6);
    sort(v + 2, v + 5);
    vector<int> w = {3, 1, 4, 6, 1, 8};
    sort(w.begin(), w.end());
    sort(w.begin() + 2, w.end() + 5);
    sort(w.rbegin(), w.rend());
    sort(w.begin(), w.end(), [](int x, int y) {
        return x > y;
    });
    bool asc; cin >> asc;
    sort(w.begin(), w.end(), [&](int x, int y) {
        return asc ? x < y : x > y;
    });
    return 0;
}
```

- Articol mai vechi.
- Dacă vrem să sortăm subsecvența `[x, y)`, vom pasa ca parametri doi iteratori către pozițiile `x` și `y`.
- Sintaxa `[](...) { ... }` se numește funcție lambda și returnează un obiect de tip funcție.
- Complexitate: `O(n log n)`.

Funcțiile `lower_bound` și `upper_bound`

- Funcționează exact ca cele din `set`, dar pe vectori.
- Ne scapă de implementarea căutării binare (pe vectori).
- Exemplu: articol.

Funcția next_permutation

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int perm[] = {1, 2, 3, 4};
    do {
        for (int i : perm)
            cout << i << ' ';
        cout << '\n';
    } while (next_permutation(begin(perm), end(perm)));
    return 0;
}
```

- Dacă permutarea dată admite succesor, acesta este generat (în timp liniar) în `perm` și se returnează `true`.

Probleme

- https://www.pbinfo.ro/probleme/2225/complementar
- https://www.pbinfo.ro/probleme/2629/h3
- https://www.pbinfo.ro/probleme/2217/map
- https://www.pbinfo.ro/probleme/2628/h2
- https://www.pbinfo.ro/probleme/2631/h4
- https://www.pbinfo.ro/probleme/3626/min-len-subseq