Introducere

1 Generalităţi

Structurile de date reprezintă colecţii de elemente (date) în care există anumite relaţii structurale. Fiecare element dintr-o astfel de structură are o anumită poziţie în cadrul structurii şi există un mod specific de acces al acestui element.

În informatică, structurile de date sunt utilizate pentru memorarea şi manipularea eficientă unor mulţimi dinamice de date. Denumirea de *mulţimi dinamice* provine de la faptul că acestea pot suferi modificări prin operaţii de inserare şi/sau extragere de elemente.

Reprezentarea elementelor unei mulţimi

În anumite situaţii, de exemplu în cazul listelor înlănţuite sau a arborilor, un element al unei mulţimi dinamice de date este reprezentat printr-o structură / un obiect, care dispune de mai multe câmpuri, dintre care unele conţin informaţii de interes, iar altele pointeri către alte elemente ale mulţimii de date (următor, precedent în cazul listelor înlănţuite sau părinte, fii în cazul arborilor).

Unele structuri presupun existenţa unui câmp *cheie*, care identifică în mod unic fiecare element.

În cazul cheilor provenind din mulţimi bine ordonate (ex: numere întregi, caractere, şiruri de caractere) pot fi efectuate operaţii de ordonare / sortare sau de determinare a elementului cu cheia minimă respectiv maximă din mulţimea dinamică păstrată în structura de date respectivă.

Operaţiile efectuate asupra unei mulţimi dinamice de date pot fi de două tipuri:

1. **Cereri** - operaţii care extrag informaţii din mulţimea de date, cea mai importantă fiind *căutarea*

2. **Operaţii de modificare a mulţimii** - inserearea / ştergerea unui element.

Există un număr mare de diferite structuri de date, fiecare având proprietăţi specifice şi fiind adaptate rezolvării unui anumit tip de probleme. Alegerea structurii de date potrivite în cadrul unui program ar trebui să ţină cont de rezultatul dorit, de complexitatea de timp şi memorie pe care le presupune utilizarea acelei structuri pentru problema dată.

Această carte este destinată celor care interesaţi de domeniu, dar mai ales studenţilor la informatocă. În capitolele ce urmează de vor fi prezentate o parte dintre cele mai des

utilizate structuri de date, împreună cu exemple de aplicaţii, algoritmi în pseudo-cod, indicaţii de implemetare în C++ precum şi o serie de probleme propuse spre rezolvare.

2 Biblioteca STL

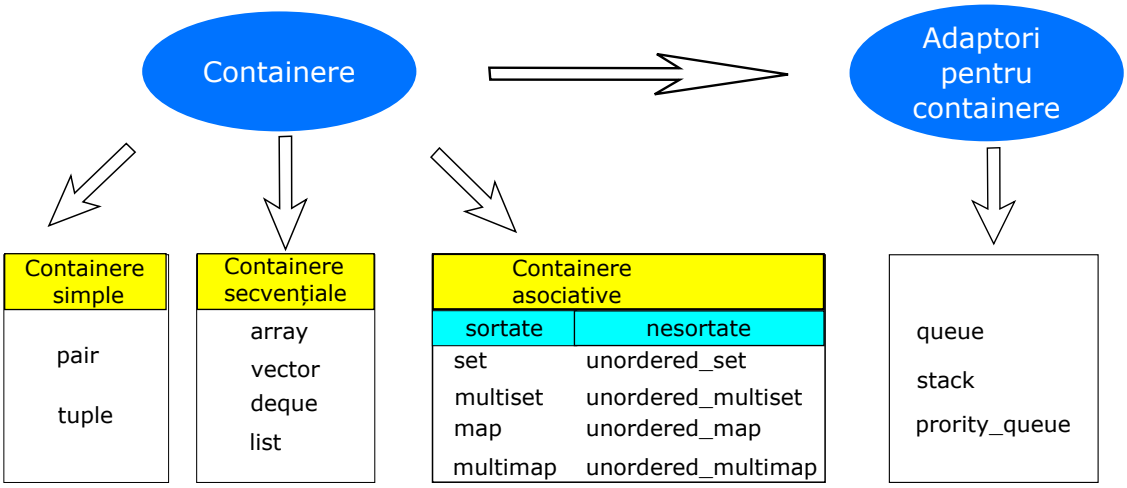

Figura 1: Containerele principale din STL

Prezentăm în coninuare containerele simple şi containerele scvenţiale. Apoi, la fiecare capitol care descrie o structură implementată şi în STL, va fi prezentată pe scurt clasa corespunzătoare, împreună cu exemple de utilizare.

Biblioteca STL din C++ conţine o serie de clase şablon - *template* - care implementează structuri de date uzuale, precum stive, cozi, liste înlănţuite, cozi de prioritate şi altele, dar şi algorimi, precum sortare şi căutare, funcţii şi iteratori.

Pentru implementarea structurilor de date se utilizează containere, adaptori de containere şi iteratori.

Şabloanele permit utilizarea structurilor de date indiferent de tipul datelor, pe care dorim să îl memorăm.

Containerele sunt cele care stochează efectiv informaţia, de exemplu tablourile - *arrays* - sau listele înlănţuite, iar adaptorii de containere au la bază un container simplu, cu anumite reguli de acces specifice (adapteză un container la cerinţele problemei). Astfel de adaptori sunt de exemplu stivele.

Iteratorii permit o iterare uniformă prin elementele structurilor, fără ca utilizatorul să fie nevoit să cunoască modul de acces al elementelor. Iteratorii sunt reprezentaţi prin pointeri către elemente ale structurii. Cei mai importanţi iteratori sunt *begin()* şi *end()*, care permit accesul la începutul, respectiv după ultimul element al stucturii. Iterarea se face apoi cu ajutorul operatorilor de incrementare ++ şi decrementare −−.

Algoritmii implementaţi în STL au ca parametri containere din STL şi operează pe acestea. În figura 1 sunt reprezentate tipurile principale de containere implementate în STL.

Prezentăm în coninuare containerele simple şi containerele scvenţiale. Apoi, la fiecare capitol care descrie o structură implementată şi în STL, va fi prezentata pe scurt clasa corespunzătoare, împreună cu exemple de utilizare.

2.1 Containere simple - tuple, pair

```
template <class T1, class T2, class T3, ..> struct tuple;
```

Această clasă *template* defineşte un container care grupează două sau mai multe valori, care pot fi de orice tip într-o singură variabilă.

Clasa *tuple* dispune de o funţie **swap()**, care permite schimbarea valorilor câmpurilor cu alte valori de acelaşi tip, de funcţia **get()** care returnează valorile tuplu-lui, precum şi de un operator de atribuire.

Există de asemenea câteva funcţii ne-membre care operează cu obiecte de tip *tuple*, dintre care cea mai importantă este:

```
tuple<T1,T2, T3...> make_tuple (T1 x, T2 y, T3 z,...);
```

care creează un obiect de tip *tuple* pe baza a tuturor valorilor x, y, z, ...

Exemplu - declararea variabilelor de tip tuple

```
#include<iostream>
#include<tuple>

int main()
{
    // declararea obiectului pers1 de tipul tuple,
    // care retine un sir de caractere si doua
    //numere intregi = id si varsta
    std::tuple<std::string, int, int> pers1;

    // declararea obiectului pers2 si initializarea acestuia
    std::tuple<std::string,int,int>pers2("Ion", 112, 23);

    //copierea variabilei pers1 in variabila pers3
    std::tuple<std::string, int, int> pers3(pers1);

    //initializarea cu lista de initializare
    std::tuple<std::string, int, int> pers3 = {"Ana", 110, 32};
    return 0;
}
```

Exemplu - accesul la membrii unui tuplu

```
#include<iostream>
#include<tuple>

int main()
{
        int id, varsta;
        std::string nume;

        std::cout << "Nume=";
        std::getline(std::cin, nume);
        std::cout << "Id="; std::cin >> id;
        std::cout << "Varsta="; std::cin >> varsta;

        //crearea unui tuplu utilizand valorile citite
        pers1 = std::make_tuple(nume, id, varsta);
        //alternativa a functiei make_tuple
        pers1 = { nume, id, varsta };

        //accesarea campurilor variabilei cu get<indice>
        std::cout << "Nume: " << std::get<0>(pers1) << "\n";
        std::cout << "id:" << std::get<1>(pers1) << "\n";
        std::cout << "Varsta:" << std::get<2>(pers1) << "\n";

        return 0;
}
```

```
template <class T1, class T2> struct pair;
```

Această clasă *template* defineşte un container care grupează două valori, care pot fi de orice tip, într-o singură variabilă. Clasa este un caz particular de tuplu şi dispune de doi membri publici: *first* şi *second*, care permit accesul la cele două valori, precum şi funcţii de manipulare asemenea tipului de date *tuple*.

Exemplu - declararea variabilelor de tip pair

```
#include<iostream>

int main()
{
    //declararea obiectului pers1 de tipul pair,
    //care retine un sir de caractere
    //si un numar intreg
    std::pair<std::string, int> pers1;

    //declararea obiectului pers2 si
    //initializarea acestuia cu valorile "Ion" si 124
    std::pair<std::string, int> pers2("Ion", 124);

    //copierea variabilei pers1 in variabila pers3
    std::pair<std::string, int> pers3(pers1);

    //initializarea cu lista de initializare
    std::pair<std::string, int> pers4 = {"Ana", 110};
    return 0;
}
```

Exemplu - accesul la membrii variabilelor de tip pair

```
#include<iostream>

int main()
{
    std::pair<std::string, int> pers1("Maria'", 125);
    int id;
    std::string nume;

    std::cout << "Nume=";
    std::getline(std::cin, nume);
    std::cout << "Id="; std::cin >> id;

    //crearea unei perechi utilizand valorile citite
    pers1 = std::make_pair(nume, id);

    //alternativa a functiei make_pair,
    //utilizand lista de initializare
    pers1 = { nume, id };
    std::cout << "Prima persoana" << "\n";

    //accesarea campurilor variabilei
    std::cout << "Nume: " << pers1.first << "\n";
    std::cout << "id:" << pers1.second << "\n\n";

    //accesarea campurilor cu get
    std::cout << "A doua persoana" << "\n";
    std::cout << "Nume: " << std::get<0>(pers2) << "\n";
    std::cout << "id:" << std::get<1>(pers2) << "\n";

    return 0;
}
```

2.2 Containere secvenţiale - vector, array

```
template<class T, class Alloc = allocator<T>>class vector;
```

Clasa *vector* defineşte un container care stochează o secvenţă de elemente de tip *T*. Tipul T al elementelor din vector trebuie stabilit atunci când se declară un astfel de container. Elementele păstrate în vector se stochează în zone de memorie succesive, atfel încât accesul se poate realiza prin poziţie.

Un container de tip vector are dimensiune variabilă şi utilizează intern alocare dinamică de memorie. Atunci când se adaugă un element nou la vector, dacă memoria alocată nu este suficientă, se realocă o nouă zonă de memorie, se copiază elementele din zona veche şi se dealocă zona veche de memorie. Aceste acţiuni se realizează automat şi sunt gestionate de către *alocator*.

Deoarece operaţiile de realocare şi copiere sunt costisitoare din punct de vedere al timpului de execuţie, acest lucru nu se realizează la fiecare nouă adăugare a unui element, ci alocatorul alocă de obicei ceva mai multă memorie decât este necesară pe moment. Numărul de poziţii alocate la fiecare adăugare depinde de obicei de numărul de adăugări efectuate până la momentul respectiv şi depinde de implementările diferitelor biblioteci.

Astfel numărul de elemente stocate efectiv în vector poate fi diferit de numărul de poziţii alocate, obiectul dispunând de două funcţii membre:

- **size()** - returnează numărul de elemente stocat în vector
- **capacity()** - returnează numărul de poziţii alocate pentru vector

Alocarea memoriei necesare pentru stocarea elementelor din vector se poate realiza prin apelul consecutiv al funcţiei care inserează elemente în container sau prin utilizarea funcţiei **reserve(numar)** care alocă *numr* poziţii de memorie.

```
std::vector<int> vector = {1};
vector.reserve(10);

std::cout << "Capacitatea vectorului:"
                  << vector.capacity() << "\n"; // 10 pozitii
std::cout << "Lungimea vectorului:"
                  << vector.size() << "\n"; // lungime 1

if (vector.empty())
        std::cout << " vectorul este vid";
else
        std::cout << " vectorul nu este vid";

vector.clear();
// capacitatea vectorului este 10, iar lungimea 0
```

Un vectorul care nu conţine niciun element se numeşte *vector vid*, iar această proprietate este verificată prin apelul funcţiei **empty()**, care întoarce un răspuns boolean.

Dintr-un vector care conţine mai multe elemente putem obţine un vector vid utilizând funcţia **clear()**, care şterge toate elementele din vector, fără a elibera zonele de memorie ocupate de acestea.

Clasa *vector* dispune de o serie de constructori, care permit declararea şi iniţializarea obiectelor de acest tip. Câteva exemple sunt prezentate mai jos:

```
//declararea unui vector vid de nr intregi
std::vector<int>first;

//vector ce contine elementul 4 de 100 de ori
std::vector<int>second(4, 100);

//vector ce contine elementele {1, 2, 3}
std::vector<int>third = {1, 2, 3};

//vector ce contine elementele lui second
std::vector<int>fourth(second.begin(),second.end());

//vector copie a lui third
std::vector<int>fifth(third);
```

Observaţie: În exemplul de mai sus, vectorul *fourth* se obţine prin ***iterare*** a vectorului *second*, cu ajutorul iteratorilor *begin()* şi *end()*. La finalu secţiuni este prezentată pe scurt noţiunea de iterator împreună cu un exemplu pentru vector.

Adăugarea unui element la sfârşitul unui vector se realizează cu ajutorul funcţiei **push_back**. Adăugarea unui element nou determină creşterea dimensiunii vectorului cu o unitate. Dacă prin aceasta se depăşeste memoria alocată, adică capacitatea vectorului, atunci se produce realocare de memorie. Argumentul funcţiei trebuie să fie un obiect de tipul celor stocate în vector.

Ştergerea unui element la sfârşitul unui vector se realizează cu ajutorul funcţiei **pop_back**. Această funcţie este apelată fără parametri şi determină modificarea lungimii vectorului, dar nu şi a capacităţii acestuia.

```
std::vector<int> vector = { 1,2,3 };
// stergerea elementului de pe ultima pozitie
vector.pop_back();

// inserarea valorii 4 la finalul sirului de valori
vector.push_back(4);
// vectorul contine valorile: { 1, 2, 4 }
```

Accesul elementelor dintr-un container de tip vector este gestionat cu ajutorul operatorului [], precum şi a unor funcţii membre:

- **operatorul []** - returnează valoarea elementului de pe o poziţie dată ca parametru. În momentul utilizării acestui operator este necesar să ne asigurăm de faptul că în vector există poziţia solicitată prin parametru, deoarece operatorul nu verifică acest aspect, fapt ce cauzează un comportament nedefinit.
- **funcţia at()** - returnează valoarea elementului de pe o poziţie dată ca parametru. Această funcţie verifică şi semnalează printr-o excepţie de tipul *out_of_range* dacă poziţia se află în afara intervalului de poziţii valide
- **funcţia front()** - returnează referinţa elementului de pe prima poziţie
- **funcţia back()** - returnează referinţa elementului de pe ultima poziţie

Exemplu:

```
#include<iostream>
#include <vector>
int main()
{
    std::vector<int> vector = { 1, 2, 3, 4, 5 };

    std::cout << "Primul element: " << vector.front() << "\n";
    std::cout << "Ultimul element: " << vector.back() << "\n";
    std::cout << "Al doilea element: " << vector[1] << "\n";
    std::cout << "Al patrulea element: "<< vector.at(3) << "\n";

    return 0;
}
```

Parcurgerea elementelor dintr-un vector se realizează cu ajutorul instrucţiunii repetitive *for* într-unul din următoarele moduri:

- **metoda clasică** - prin intermediul for-ului variabila *poz* reţine, la un moment dat, fiecare poziţie din vector, iar elementele containerului pot fi accesate cu ajutorul *operatorului [ ]* sau al funcţiei *at()*.

```
for (int poz = 0; poz < vector.size(); poz++)
        std::cout << vector[poz] << " ";
```

- **range base for** - variabila *element* va reţine, pe rând, toate elementele vectorului, acestă metodă nepermiţând o parcurgere parţială a containerului.
  Din punct de vedere sintactic for-ul este format din: tipul elementelor din vector, urmat de numele unei variabile ce va reţine pe rând fiecare element din container, apoi operatorul ":" şi numele vectorului (containerului) parcurs.

```
for(int element : vector)
        std::cout << element << " ";
```

- **utilizând iteratori** - variabila *it* reţine, la un moment dat, un pointer către fiecare element din vector.
  Sintactic observăm o manieră asemănătaore cu cea a *for-ului clasic*.

```
std::vector<int>::iterator it;
for (it = vector.begin(); it < vector.end(); it++)
        std::cout << *it << " ";
```

```
template < class T, size_t N > class array;
```

Clasa *array* defineşte un container generic care stochează o secvenţă de elemente de tip T, asemenea unui *vector*, principala diferenţa a celor două containere fiind variabilitatea dimensiunii. Array-ul este un container caracterizat prin dimensiunea fixă de N poziţii. Tipul T al elementelor din array, precum şi capacitatea acestuia N, trebuie stabilite atunci când se declară un astfel de container.

Pentru manipularea şi parcurgerea elementelor dintr-un array se pot utiliza funcţii prezentate în cadrul secţiunii *vector*, cu următoarele observaţii:

- lungimea array-ului este egală cu capacitatea acestuia, iar această valoare este determinată folosind funcţia **size()**
- alocarea de memorie se realizează în momentul declarării variabilei, motiv pentru care nu există funcţia **reserve()**
- nu putem apela funcţii de inserare sau de ştergere a unui element în/din container, dimensiunea array-ului fiind fixă.

```
#include<iostream>
#include <array>

int main()
{
        //declararea unui array de lungime 7,
        //ce contine valori default de tip float
        std::array<float, 7> first;

        //declararea unui array de lungime 10, ce contine
        // valorile intervalului [1,6] pe primele 6 pozitii
        std::array<int, 10> second = { 1, 2, 3, 4, 5, 6 };

        //declararea unui array copie a lui second
        std::array<int, 10> third(second);

        std::cout << "Lungimea celui de-al doilea array:"
                << second.size() << "\n"; // lungime 10

        if (second.empty())
                std::cout << " vectorul este vid";
        else
                std::cout << " vectorul nu este vid";
```

```
        for (int poz = 0; poz < second.size(); poz++)
                std::cout << second[poz] << " ";
        // se va afisa: { 1, 2, 3, 4, 5, 6, 0, 0, 0, 0 }

        third.at(6) = 7;
        for(int element : third)
                std::cout << element << " ";
        // se va afisa: { 1, 2, 3, 4, 5, 6, 7, 0, 0, 0 }

        return 0;
}
```

Iteratori

Iteratorii sunt obiecte care *pointează* spre un element aflat într-o colecţie de elemente şi care permit iterarea prin acea colecţie utilizând operatori de incrementare (++) sau decrementare (–). Utilizarea iteratorilor contribuie la o eficientizare a timpului de execuţie, întrucât se lucrează cu pointeri către elementele din container, astfel copierea acestora este evitată.

Un alt avantaj oferit de către iteratori este acela că, utilizatorii nu sunt nevoiţi să ştie, cum trebuie iterat printr-un container, de exemplu listă înlănţuită sau arbore binar (map), acest lucru fiind gestionat de către iterator. O parcurgere, din punct de vedere al codului scris este absolut similară pentru oricare tip de container.

Containerele din STL conţin suport ce permite parcurgerea acestora utilizând iteratori cu ajutorul funcţiilor **begin()** şi **end()**, care returnează un pointer către primul, respectiv către prima poziţie de după ultimul element al colecţiei.

Exemplu

```
#include<iostream>
#include <vector>

int main()
{
        std::vector<int> vector = { 1, 2, 3, 4, 5, 6 };

        std::vector<int>::iterator it;
        for (it = vector.begin(); it != vector.end(); it++)
                std::cout << *it << " ";

        return 0;
}
```

Observaţie! Deoarece iteratorul reţine un pointer către un element din container este necesar să utilizăm **operatorul *** pentru a obţine valoarea elementului.

Pentru a accesa elementul de pe o poziţie specifică din cadrul containerului se pot utiliza funcţiile **std::next()** şi **std::prev()** sau operatorii de adunare şi scădere.

Exemplu

```
#include<iostream>
#include <vector>

int main()
{
        std::vector<int> vector = { 1, 2, 3, 4, 5, 6 };

        // iterator catre primul element din vector
        std::vector<int>::iterator begin_it = vector.begin();

        //iterator catre urmatoarea pozitie de dupa ultimul
        //element al vectorului
        std::vector<int>::iterator end_it = vector.end();

        std::cout << *begin_it << " "; // se va afisa val. 1
        //std::cout << *end_it << " "; // eroare

        auto next_it = std::next(begin_it, 2);
        std::cout << *next_it << " "; // se va afisa val. 3

        // alternativa a functiei std::next()
        std::cout << *(begin_it + 2) << " ";

        auto prev_it = std::prev(end_it, 1);
        std::cout << *prev_it << " "; // se va afisa val. 6

        // alternativa a functiei std::prev()
        std::cout << *(end_it - 1) << " ";

        return 0;
}
```