Structuri de date
Curs - anul I - sem II

Universitatea "Transilvania" din Brașov

22 februarie 2022

Structura cursului

Structura cursului

Structura cursului

1. Introducere în SD. Biblioteca STL

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare
5. Heap-uri binare. Cozi de prioritate

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare
5. Heap-uri binare. Cozi de prioritate
6. Arbori binari de căutare

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare
5. Heap-uri binare. Cozi de prioritate
6. Arbori binari de căutare

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare
5. Heap-uri binare. Cozi de prioritate
6. Arbori binari de căutare
7. Arbori AVL

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare
5. Heap-uri binare. Cozi de prioritate
6. Arbori binari de căutare
7. Arbori AVL
8. Arbori roșu-negru.

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare
5. Heap-uri binare. Cozi de prioritate
6. Arbori binari de căutare
7. Arbori AVL
8. Arbori roșu-negru.
9. Îmbogățirea arborilor

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare
5. Heap-uri binare. Cozi de prioritate
6. Arbori binari de căutare
7. Arbori AVL
8. Arbori roșu-negru.
9. Îmbogățirea arborilor
10. B-arbori

Structura cursului

1. Introducere în SD. Biblioteca STL
2. Liste înlănțuite
3. Stive. Cozi. Aplicații.
4. Tabele de repartizare
5. Heap-uri binare. Cozi de prioritate
6. Arbori binari de căutare
7. Arbori AVL
8. Arbori roșu-negru.
9. Îmbogățirea arborilor
10. B-arbori
11. Structuri de date avansate.

Condiții de promovare

Nota la examen constă din:

- Nota la laborator = media notelor pentru temele de laborator. Programele vor fi elaborate în limbajul C++. Nota minimă pentru promovarea laboratrorului este 5.

Atenție citiți criteriile de evaluare pentru laborator de pe elearning.

Condiții de promovare

Nota la examen constă din:

- Nota la laborator = media notelor pentru temele de laborator. Programele vor fi elaborate în limbajul C++. Nota minimă pentru promovarea laboratrorului este 5.
- Nota la examenul scris = este necesar minim 5 pentru promovare

Atenție citiți criteriile de evaluare pentru laborator de pe elearning.

Condiții de promovare

Nota la examen constă din:

- Nota la laborator = media notelor pentru temele de laborator. Programele vor fi elaborate în limbajul C++. Nota minimă pentru promovarea laboratrorului este 5.
- Nota la examenul scris = este necesar minim 5 pentru promovare
- Nota finală = 0.4 * Nota Lab + 0.6 * Nota Examen. La medie se adaugă puncte suplimentare (dacă există)

Atenție citiți criteriile de evaluare pentru laborator de pe elearning.

Introducere

Structuri de date

- colecții de date în care există anumite relații structurale.

Introducere

Structuri de date

- colecții de date în care există anumite relații structurale.
- fiecare element are o anumită poziție în cadrul structurii și există un mod specific de acces al acestui element.

Introducere

Structuri de date

- colecții de date în care există anumite relații structurale.
- fiecare element are o anumită poziție în cadrul structurii și există un mod specific de acces al acestui element.
- utilizate pentru memorarea și manipularea eficientă cu calculatorul a unor mulțimi dinamice de date

Structuri de date

Reprezentarea elementelor

- deseori prin structuri/obiecte cu mai multe câmpuri (informații + referințe către alte elemente)
- uneori unul dintre câmpuri = cheie (identifică elementul)
- cheile din mulțimi bine ordonate permit operații de ordonare/sortare a elementelor

Structuri de date

Operații de bază

1. Cereri:

Structuri de date

Operații de bază

1. Cereri:
   - $CAUTA(S, k)$

Structuri de date

Operații de bază

1. Cereri:
   - $CAUTA(S, k)$
   - $MINIM(S)$ / $MAXIM(S)$

Structuri de date

Operații de bază

1. Cereri:
   - $CAUTA(S, k)$
   - $MINIM(S)$ / $MAXIM(S)$
   - $SUCCESOR(S, x)$ /$PREDECESOR(S, x)$

Structuri de date

Operații de bază

1. Cereri:
   - $CAUTA(S, k)$
   - $MINIM(S)$ / $MAXIM(S)$
   - $SUCCESOR(S, x)$ /$PREDECESOR(S, x)$
2. Operații de modificare a mulțimii:

Structuri de date

Operații de bază

1. Cereri:
   - $CAUTA(S, k)$
   - $MINIM(S)$ / $MAXIM(S)$
   - $SUCCESOR(S, x)$ /$PREDECESOR(S, x)$
2. Operații de modificare a mulțimii:
   - $INSERTIE(S, x)$

Structuri de date

Operații de bază

1. Cereri:
   - $CAUTA(S, k)$
   - $MINIM(S)$ / $MAXIM(S)$
   - $SUCCESOR(S, x)$ /$PREDECESOR(S, x)$
2. Operații de modificare a mulțimii:
   - $INSERTIE(S, x)$
   - $STERGERE(S, x)$

Biblioteca STL - Generalități

- STL = Standard Template Library.

Biblioteca STL - Generalități

- STL = Standard Template Library.
- set de clase template C++ care implementează algoritmi și structuri de date frecvent utilizate, precum vectori, liste, cozi sau stive.

Biblioteca STL - Generalități

- STL = Standard Template Library.
- set de clase template C++ care implementează algoritmi și structuri de date frecvent utilizate, precum vectori, liste, cozi sau stive.
- Componente principale:

Biblioteca STL - Generalități

- STL = Standard Template Library.
- set de clase template C++ care implementează algoritmi și structuri de date frecvent utilizate, precum vectori, liste, cozi sau stive.
- Componente principale:
  - Containere - utilizate pentru managementul multimilor de obiecte - vector, list, etc.

Biblioteca STL - Generalități

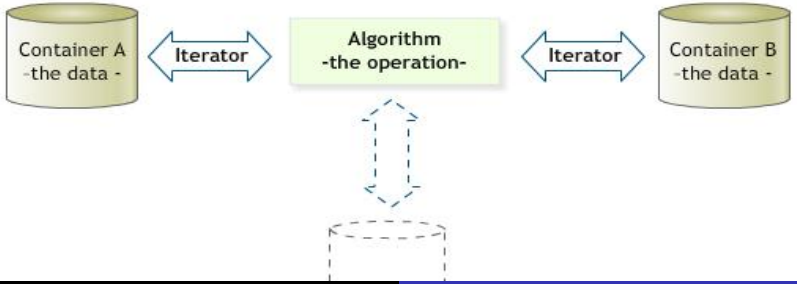
- STL = Standard Template Library.
- set de clase template C++ care implementează algoritmi și structuri de date frecvent utilizate, precum vectori, liste, cozi sau stive.
- Componente principale:
  - Containere - utilizate pentru managementul multimilor de obiecte - vector, list, etc.
  - Algoritmi - se aplică asupra containerelor - ex. sortare, căutare.

Biblioteca STL - Generalități

- STL = Standard Template Library.
- set de clase template C++ care implementează algoritmi și structuri de date frecvent utilizate, precum vectori, liste, cozi sau stive.
- Componente principale:
  - Containere - utilizate pentru managementul multimilor de obiecte - vector, list, etc.
  - Algoritmi - se aplică asupra containerelor - ex. sortare, căutare.
  - Iteratori - permit navigarea printre obiectele unui container.

Biblioteca STLI

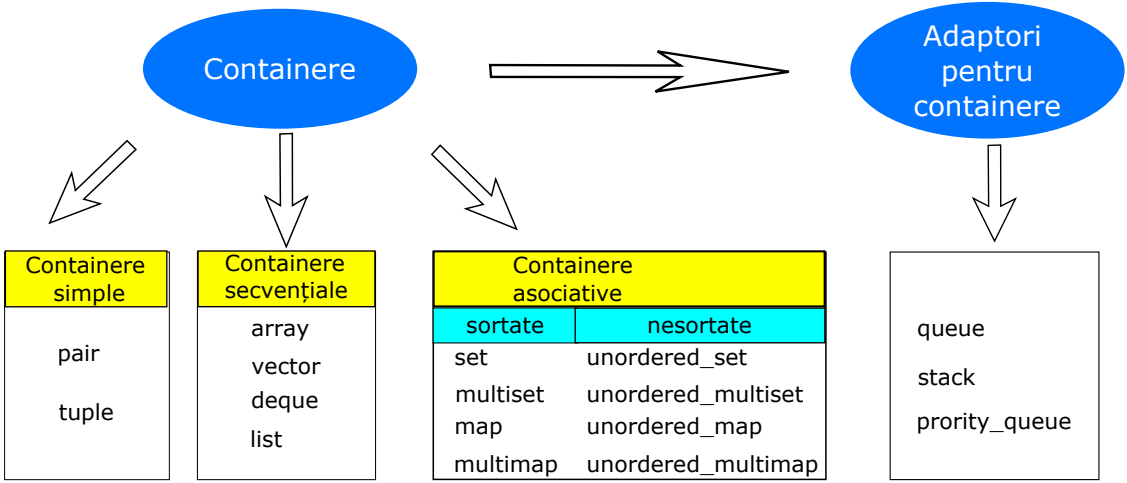
1. Templates
   make algorithms independent of the data types
2. Iterators
   make algorithms independent of the containters



Containere - General

- **Containere**: sunt tipuri de date din STL care conțin date
- **Adaptori -*adapters***: - sunt tipuri de date din STL care adaptează un container la o interfață specifică (coadă, stivă).

Containere - General



Structuri elementare

- Elemente care dispun de mai multe câmpuri. Pot fi definite în C++ folosind:

Structuri elementare

- Elemente care dispun de mai multe câmpuri. Pot fi definite în C++ folosind:
  - struct (C sau C++)

Structuri elementare

- Elemente care dispun de mai multe câmpuri. Pot fi definite în C++ folosind:
  - struct (C sau C++)
  - tuple (STL)

Structuri elementare

- Elemente care dispun de mai multe câmpuri. Pot fi definite în C++ folosind:
  - struct (C sau C++)
  - tuple (STL)
- Tablouri = mulțimi de date de același tip, stocate în zone de memorie succesive. În (C/C++)

Structuri elementare

- Elemente care dispun de mai multe câmpuri. Pot fi definite în C++ folosind:
  - struct (C sau C++)
  - tuple (STL)
- Tablouri = mulțimi de date de același tip, stocate în zone de memorie succesive. În (C/C++)
  - tablouri alocate static: int tablou[10];

Structuri elementare

- Elemente care dispun de mai multe câmpuri. Pot fi definite în C++ folosind:
  - struct (C sau C++)
  - tuple (STL)
- Tablouri = mulțimi de date de același tip, stocate în zone de memorie succesive. În (C/C++)
  - tablouri alocate static: int tablou[10];
  - tablouri definite cu ajutorul pointer-ilor și alocate dinamic: int *v=new int[10];

Structuri elementare

- Elemente care dispun de mai multe câmpuri. Pot fi definite în C++ folosind:
  - struct (C sau C++)
  - tuple (STL)
- Tablouri = mulțimi de date de același tip, stocate în zone de memorie succesive. În (C/C++)
  - tablouri alocate static: int tablou[10];
  - tablouri definite cu ajutorul pointer-ilor și alocate dinamic: int *v=new int[10];
  - array și vector din STL

Containere simple - *pair*

```cpp
#include <iostream>


int main()
{
    std::pair<int, int> coord;
    coord.first = 23;
    coord.second = 10;

    std::pair<std::string, std::string> persoana("Ionescu", "Maria");

    std::pair<std::string, std::string> prieten = persoana;

    std::pair<std::string, int> elev1 = { "Popescu Andrei", 9 };

    std::pair<std::string, int> elev2 = std::make_pair("Ileana", 7);

    return 0;
}
```

Containere simple - *tuple*

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

int main()
{
    std::tuple<std::string, int, int, int, float> elev;
    std::tuple <std::string, std::pair<int, int>> oras;

    elev = std::make_tuple("Ionescu", 9, 8, 10, 9.00);
    //obtinerea unui camp din tuplu - cu get
    std::cout << "nume: "<<std::get<0>(elev)<<"\n";
    std::cout << "note: " << std::get<1>(elev) << ", " << std::get<2>(elev);
    std::cout << ", " << std::get<3>(elev);

    oras = { "Brasov", {50, 20} };

    std::string nume;
    std::pair<int, int>coord;
    std::tie(nume, coord) = oras;

    return 0;
}
```

Containere secvențiale - tablouri în STL - array

template< class T, std::size_t N> struct array;

- echivalentul unui vector de dimensiune fixă

Containere secvențiale - tablouri în STL - array

template< class T, std::size_t N> struct array;

- echivalentul unui vector de dimensiune fixă
- la declarare trebuie specificat tipul elementelor stocate și dimensiunea vectorului

Containere secvențiale - tablouri în STL - array

template< class T, std::size_t N> struct array;

- echivalentul unui vector de dimensiune fixă
- la declarare trebuie specificat tipul elementelor stocate și dimensiunea vectorului
- elementele sunt stocate în zone de memorie succesive

Containere secvențiale - tablouri în STL - array

template< class T, std::size_t N> struct array;

- echivalentul unui vector de dimensiune fixă
- la declarare trebuie specificat tipul elementelor stocate și dimensiunea vectorului
- elementele sunt stocate în zone de memorie succesive
- elementele pot fi accesate prin poziție

Containere secvențiale - tablouri în STL - array

template< class T, std::size_t N> struct array;

- echivalentul unui vector de dimensiune fixă
- la declarare trebuie specificat tipul elementelor stocate și dimensiunea vectorului
- elementele sunt stocate în zone de memorie succesive
- elementele pot fi accesate prin poziție
- un *array* dispune de o functie, care returnează dimensiunea vectorului

Containere secvențiale - tablouri în STL - array

template< class T, std::size_t N> struct array;

- echivalentul unui vector de dimensiune fixă
- la declarare trebuie specificat tipul elementelor stocate și dimensiunea vectorului
- elementele sunt stocate în zone de memorie succesive
- elementele pot fi accesate prin poziție
- un *array* dispune de o functie, care returnează dimensiunea vectorului
- în C++20 sunt definiți operatori de comparație

Containere secvențiale - tablouri în STL - array

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

int main()
{
    const int N = 10;
    std::array<int, N> my_array= { 1,2,3,4,5 }; // restul el sunt initializate cu 0

    for (int index = 0; index < my_array.size(); index++)
    {
        my_array[index]++;
    }

    return 0;
}
```

Containere secvențiale - iterarea prin array

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


int main()
{
    const int N = 20;
    std::array < std:: string, N > nume = { "Ionescu", "Popescu", "Maria", "Dinu", "Elena" };

    //1. cu indici
    for (int index = 0; index < nume.size(); ++index)
        std::cout << nume[index] << " ";

    //2. cu iteratori - vom discuta la cursul urmator!

    //3. cu : => se considera pe rand elementele din array

    for (std::string x : nume)
        if (x != "")
            std::cout << x << " ";

}
```

permite selectarea pe rând a fiecărui element
poate fi folosit pentru orice fel de container iterabil

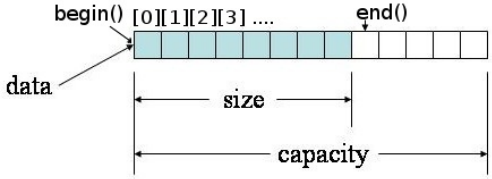
x este un ELEMENT din array-ul nume NU un indice!!!

Containere de tip secvență - *vector*

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

```
std::vector<int> first; //vector fara elemente
std::vector<int> second(4, 20); //vector cu 4 elem egale cu 20
std::vector<int> third = { 1, 2, 3, 4 };
```

- container de tip secvență de dimensiune modificabilă.

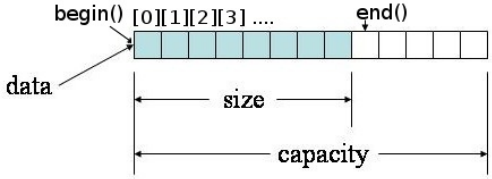


Containere de tip secvență - *vector*

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

```
std::vector<int> first; //vector fara elemente
std::vector<int> second(4, 20); //vector cu 4 elem egale cu 20
std::vector<int> third = { 1, 2, 3, 4 };
```

- container de tip secvență de dimensiune modificabilă.
- zone de memorie alăturate pentru stocarea elementelor.

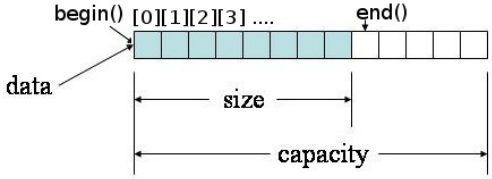


Containere de tip secvență - *vector*

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

```
std::vector<int> first; //vector fara elemente
std::vector<int> second(4, 20); //vector cu 4 elem egale cu 20
std::vector<int> third = { 1, 2, 3, 4 };
```

- container de tip secvență de dimensiune modificabilă.
- zone de memorie alăturate pentru stocarea elementelor.
- memorie alocată dimanic $\Rightarrow$ necesită uneori realocarea

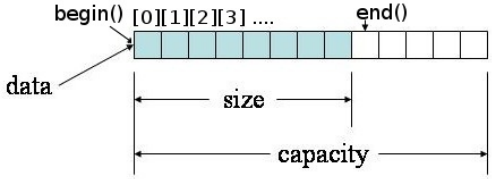


Containere de tip secvență - *vector*

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

```
std::vector<int> first; //vector fara elemente
std::vector<int> second(4, 20); //vector cu 4 elem egale cu 20
std::vector<int> third = { 1, 2, 3, 4 };
```

- container de tip secvență de dimensiune modificabilă.
- zone de memorie alăturate pentru stocarea elementelor.
- memorie alocată dimanic $\Rightarrow$ necesită uneori realocarea
- tipul datelor stocate trebuie stabilit la declarare

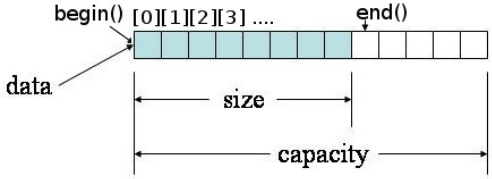


Containere de tip secvență - *vector*

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

```
std::vector<int> first; //vector fara elemente
std::vector<int> second(4, 20); //vector cu 4 elem egale cu 20
std::vector<int> third = { 1, 2, 3, 4 };
```

- container de tip secvență de dimensiune modificabilă.
- zone de memorie alăturate pentru stocarea elementelor.
- memorie alocată dimanic $\Rightarrow$ necesită uneori realocarea
- tipul datelor stocate trebuie stabilit la declarare
- accesul elementelor prin poziție



Containere de tip secvență - *vector* - adăugarea unui element

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

int main()
{
    std::vector<char> litere;
    int nr = 10;
    for (int index = 0; index < nr; index++)
        litere.push_back('a' + index);

    for (int index = 0; index < litere.size(); index++)
        std::cout << litere[index] << " ";

    for (char lit: litere)
    {
        if (lit == 'a' || lit == 'e' || lit == 'i')
            std::cout << "vocala\n";
        else std::cout << "consoana\n";
    }
}
```

Cu push_back - se adaugă un nou element, după ultimul element din vector.

Elementele din vector pot fi accesate prin poziție.

Parcurgerea element cu element cu :

Containere de tip secvență - *vector* - Exemplu

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

int main()
{
    std::vector<int> numere;
    int nr = 10;

    for (int index = 0; index < nr; index++)
        numere[index] = index + 1;

    for (int numar : numere)
        std::cout << numar << " ";
    std::cout << std::endl;
}
```

Este corect codul?

Containere de tip secvență - *vector* - Exemplu

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

int main()
{
    std::vector<int> numere;
    int nr = 10;

    for (int index = 0; index < nr; index++)
        numere[index] = index + 1;

    for (int numar : numere)
        std::cout << numar << " ";
    std::cout << std::endl;
}
```

Este corect codul?

NU! De ce?

Containere de tip secvență - *vector* - Exemplu

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

int main()
{
    std::vector<int> numere;
    int nr = 10;

    for (int index = 0; index < nr; index++)
        numere[index] = index + 1;

    for (int numar : numere)
        std::cout << numar << " ";
    std::cout << std::endl;
}
```

Când declar un vector, nu se alocă automat memorie pentru elementele sale!

Cum doar am declarat vectorul și nu am alocat memorie, NU pot accesa elementele prin poziție, deoarece deocamdata NU există nico poziție în vector.
Va da eroare la execuție

Cu push_back - se realocă memorie, daca s-a ajuns la capătul memoriei alocate

Containere de tip secvență - *vector* - Dimensiune

(Global Scope)

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

int main()
{
    std::vector<int> numere;
    std::cout << "before push_back:\n";
    std::cout << "size =" << numere.size();
    std::cout<< std::endl;
    std::cout << "capacity =" << numere.capacity();
    std::cout << std::endl;

    for (int index = 1; index <= 10; index++)
        numere.push_back(index);

    std::cout << "\n\n after push_back:\n";
    std::cout << "size =" << numere.size();
    std::cout << std::endl;
    std::cout << "capacity =" << numere.capacity();
    std::cout << std::endl;

}
```

Microsoft Visual Studio Debug Console

```
before push_back:
size =0
capacity =0


 after push_back:
size =10
capacity =13

D:\Facultate\SD\CursMate\TestInf
Press any key to close this wind
```

size() - numărul de element din vector.

capacity() - memoria alocată - nr de poziții disponibile

Containere de tip secvență - *vector* - redimensionarea vector

Observații:

- Prin declararea unui vector nu se alocă memorie pentru elemente.

Containere de tip secvență - *vector* - redimensionarea vector

Observații:

- Prin declararea unui vector nu se alocă memorie pentru elemente.
- După declararea

$$vector < int > tablou;$$

vectorul **tablou** are atât **size** cât și **capacity** egale cu 0.

Containere de tip secvență - *vector* - redimensionarea vector

Observații:

- Prin declararea unui vector nu se alocă memorie pentru elemente.
- După declararea

$$vector < int > tablou;$$

vectorul **tablou** are atât **size** cât și **capacity** egale cu 0.

- Se poate inițializa un vector și atunci se va aloca memorie pentru atâtea elemente cu câte s-a inițializat.

Containere de tip secvență - *vector* - redimensionarea vector

Observații:

- Prin declararea unui vector nu se alocă memorie pentru elemente.
- După declararea

$$vector < int > tablou;$$

vectorul **tablou** are atât **size** cât și **capacity** egale cu 0.

- Se poate inițializa un vector și atunci se va aloca memorie pentru atâtea elemente cu câte s-a inițializat.
- Elementele se pot accesa prin poziție în intervalul $[0, size()]$.

Containere de tip secvență - *vector* - redimensionarea vector

Observații:

- Prin declararea unui vector nu se alocă memorie pentru elemente.
- După declararea

$$vector < int > tablou;$$

vectorul **tablou** are atât **size** cât și **capacity** egale cu 0.

- Se poate inițializa un vector și atunci se va aloca memorie pentru atâtea elemente cu câte s-a inițializat.
- Elementele se pot accesa prin poziție în intervalul $[0, size()]$.
- Se poate aloca memorei cu **reserve(dim)** - în acest moment **capacity** = *dim* DAR **size = 0**.

Containere de tip secvență - *vector* - redimensionarea vector

Observații:

- Prin declararea unui vector nu se alocă memorie pentru elemente.
- După declararea

$$vector < int > tablou;$$

vectorul **tablou** are atât **size** cât și **capacity** egale cu 0.

- Se poate inițializa un vector și atunci se va aloca memorie pentru atâtea elemente cu câte s-a inițializat.
- Elementele se pot accesa prin poziție în intervalul $[0, size()]$.
- Se poate aloca memorei cu **reserve(dim)** - în acest moment **capacity** = *dim* DAR **size = 0**.
- Se poate redimensiona vectorul cu **resize(dim)** - în acest moment **capacity = size = dim** și toate elementele sunt inițializate cu 0.

Containere de tip secvență - *vector* - redimensionarea vector

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

int main()
{
    std::vector<int> numere;
    numere.resize(10);
    std::vector<int> numere2;
    numere2.reserve(10);
    std::cout << "numere:\n";
    std::cout << "size =" << numere.size();
    std::cout << std::endl;
    std::cout << "capacity =" << numere.capacity();
    std::cout << std::endl;

    std::cout << "\n\n"<< "numere2:\n";
    std::cout << "size =" << numere2.size();
    std::cout << std::endl;
    std::cout << "capacity =" << numere2.capacity();
    std::cout << std::endl;
}
```

```
numere:
size =10
capacity =10


numere2:
size =0
capacity =10

D:\Facultate\SD\CursMate\Tes
Press any key to close this
```

resize(nr) - alocă memorie și adaugă 0 de nr ori.

reserve(nr) - alocă memorie.

Containere de tip secvență - *vector*

Iteratorii: - pentru parcurgerea containerelor - vor fi discutați la cursul următor.