

PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 11

STL

Containere

Introducere – simplă clasă vector

- Fie clasa *MyVector*:

```
class MyVector
{
    size_t sz;
    int *v;
public:
    MyVector(void): sz(0), v(0) {};
    size_t GetSize(void){return sz; };
    int GetItem(size_t idx){ return v[idx];}
    void PushBack(int);
    int PopBack(void);
    ~MyVector(void){ delete []v; };
};
```



Introducere – simplă clasă vector

► Definițiile funcțiilor *PushBack* și *PopBack*:

```
void MyVector::PushBack(int x)
{
    int* aux = new int[sz+1];
    for(size_t i=0;i<sz;i++)
    {
        aux[i] = v[i];
    }
    aux[sz] = x;
    sz++;
    delete []v;
    v = aux;
}
```

```
int MyVector::PopBack(void)
{
    int tmp = v[sz-1];
    int* aux = new int[sz-1];
    for(size_t i=0;i<sz-1;i++)
    {
        aux[i] = v[i];
    }
    sz--;
    delete []v;
    v = aux;
    return tmp;
}
```



Introducere – simplă clasă vector

- Utilizarea tipului de date *MyVector*:

```
int main(void)
{
    MyVector v;

    v.PushBack(1);
    v.PushBack(2);
    v.PushBack(3);
    v.PushBack(4);
    v.PushBack(5);

    for(size_t i = 0; i<v.GetSize(); i++)
    {
        cout << v.GetItem(i) <<endl;
    }
    return 0;
}
```



Introducere – *class MyVector* – lift up

- Fie clasa template *MyVector*:

```
template<class T>
class MyVector
{
    size_t sz;
    T*v;
public:
    MyVector(void): sz(0), v(0) {};
    size_t GetSize(void){return sz; };
    T GetItem(size_t idx){ return v[idx];}
    void PushBack(T);
    T PopBack(void);
    ~MyVector(void){ delete []v; };
};
```



Introducere – *class MyVector* – lift up

- Definițiile funcțiilor template *PushBack* și *PopBack*:

```
template<typename T>
void MyVector<T>::PushBack(T x)
{
    T* aux = new T[sz+1];
    for(size_t i=0; i<sz; i++)
    {
        aux[i] = v[i];
    }
    aux[sz] = x;
    sz++;
    delete []v;
    v = aux;
}
```

```
template <typename T>
T MyVector<T>::PopBack(void)
{
    T tmp = v[sz-1];
    T* aux = new T[sz-1];
    for(size_t i=0; i<sz-1; i++)
    {
        aux[i] = v[i];
    }
    sz--;
    delete []v;
    v = aux;
    return tmp;
}
```



Introducere – *class MyVector* – lift up

```
int main(void)
{
    MyVector<double> v;

    v.PushBack(1.1);
    v.PushBack(2.2);
    v.PushBack(3.3);
    v.PushBack(4.4);
    v.PushBack(5.5);

    for(size_t i = 0; i<v.GetSize(); i++)
    {
        cout << v.GetItem(i) <<endl;
    }
    return 0;
}
```



Introducere – adăugarea unei clase iterator

```
▶ template<class T>
class MyVector
{
    size_t sz;
    T*v;
public:
    class MyIterator
    {
        //...
    };
    MyIterator Begin(void){return v;};
    MyIterator End(void){return v+sz;};

    MyVector(void): sz(0), v(0) {};
    size_t GetSize(void){return sz; };
    T GetItem(size_t idx){ return v[idx];}
    void PushBack(T);
    T PopBack(void);
    ~MyVector(void){ delete []v; };
};
```

▶

Introdurre – *class MyIterator*

```
► class MyIterator
{
    int *p;
public:
    MyIterator(void):p(0){};
    MyIterator(int *x):p(x){};
    MyIterator(const MyIterator& mit) : p(mit.p) {}
    MyIterator& operator++() {++p;return *this;}
    MyIterator operator++(int)
    {
        MyIterator tmp(*this);
        operator++();
        return tmp;
    }
    bool operator==(const MyIterator& rhs) {return p==rhs.p;}
    bool operator!=(const MyIterator& rhs) {return p!=rhs.p;}
    int& operator*() {return *p;}
};
```

Introdurre – utilizzare iterator

```
int main(void)
{
    MyVector<int> v;

    v.PushBack(1);
    v.PushBack(2);
    v.PushBack(3);
    v.PushBack(4);
    v.PushBack(5);

    for(MyVector::MyIterator it = v.Begin(); it!=v.End(); it++)
    {
        cout << *it << endl;
    }
    return 0;
}
```



Introducere - STL

- ▶ Librăria standard este un set de componente specificate în standardul ISO C++, furnizate cu un comportament identic de către orice implementare C++.
- ▶ Componentele sunt reutilizabile
- ▶ În anii '70 componentele folosite de programatori erau structurile de control și funcțiile
- ▶ În anii '80 programatorii foloseau clase dintr-o gamă largă de biblioteci dependente de platformă
- ▶ Odată cu standardul STL din anul '97 se introduc componente definite prin clase independente de platformă
- ▶ Structurile de date sunt colecții de date (containeri) organizate după diverse reguli



Introducere - STL

- ▶ În C++ structurile de date sunt obiecte ce conțin colecții de obiecte:
 - ▶ Clasa vector reprezintă un vector de obiecte de tip `int`
 - ▶ Prin utilizarea template-urilor se redefinește clasa vector la `vector<T>` astfel încât se extinde acest tip de date la `vector<char>`, `vector<double>`, `vector<Angajat>` sau orice tip de dată
 - ▶ Similar se poate proceda cu implementarea structurilor de tip stivă, lista, arbori, grafuri etc.
- ▶ STL este o bibliotecă de clase template dar conține și implementări ale structurilor de date.
- ▶ În C și C++ elementele unui tablou sunt accesate prin intermediul pointerilor. În C++ STL elementele unui container sunt accesate prin intermediul iteratorilor care sunt tot pointeri dar care se comportă inteligent.



Introducere - STL

- ▶ Containerii implementează operații primitive.
- ▶ Algoritmii ce utilizează containeri sunt independenți de tipurile de date conținute.
- ▶ În STL s-a evitat folosirea moștenirii și a funcțiilor virtuale din considerente de performanță.
- ▶ S-a evitat utilizarea operatorilor new și delete în favoarea alocatorilor (permit metode de control pentru alocare și dealocare de memorie)
- ▶ Managementul erorilor



STL - structură

- ▶ **STL cuprinde trei elemente principale:**
 - ▶ Containeri – obiecte ce conțin obiecte
 - ▶ Iteratori – pointeri „inteligenți” pentru acces la elementele unui container
 - ▶ Algoritmi – funcționalități de acces și prelucrare asupra elementelor containerilor.



Containere

Containere	Secvențiale	vector	Implementează vectori alocați dinamic.	#include <vector>
		list	Lista liniară dublu înlănțuită	#include < list >
		deque	Asemănător containerului vector, operațiile putându-se realiza la ambele capete.	#include < deque >
	Asociative	set	Mulțime sortată de elemente unice.	#include < set >
		multiset	Mulțime sortată de elemente.	#include < set >
		map	Stocază perechi sortate de tip <cheie, valoare> în care o cheie identifică în mod unic o valoare.	#include < map >
		multimap	Stocază perechi sortate de tip <cheie, valoare> în care o cheie identifică una sau mai multe valori.	#include < map >
	Adaptive	stack	Structură de tip stivă.	#include < stack >
		queue	Structură de tip coadă	#include < queue >
		priority_queue	Structură de tip coadă în care elementelor le sunt asociate priorități.	#include < queue >



Iteratori

- ▶ Iteratorii sunt obiecte care se comportă asemănător pointerilor și care sunt utilizați pentru a accesa elementele unui container.
- ▶ Iteratorii se aseamănă cu pointerii, dar sunt de fapt obiecte ce adresează alte obiecte. Cu ajutorul lor pot fi adresate elemente ale containerelor care aparțin anumitor intervale. Iteratorii reprezintă interfața de comunicație între algoritmi și containere, fiind preluați ca parametrii de către algoritmi. Containerele le furnizează algoritmilor o cale de acces către elementele lor prin intermediul iteratorilor.
- ▶ Algoritmii furnizează funcționalități de acces și prelucrare asupra elementelor containerelor



Iteratori

Iteratori	Acces aleatoriu	Stochează și regăsește valori. Elemente pot fi accesate aleatoriu.
	Bidirecționali	Stochează și regăsește valori. Iteratorul poate înainta și reveni.
	Înainte	Stochează și regăsește valori. Iteratorul poate doar înainta.
	De intrare	Regăsește dar nu stochează valori. Iteratorul poate doar înainta.
	De ieșire	Stochează dar nu regăsește valori. Iteratorul poate doar înainta.
Algoritmi	Funcții globale care oferă servicii generale cum ar fi sortări, reordonări, modificări, copieri, căutări etc.	#include <algorithm>



Algoritmi

- ▶ Algoritmii STL se împart în patru mari categorii:
 - ▶ Algoritmi care modifică ordinea elementelor în container - modifying sequence operations: *copy()*, *replace()*, *transform()* și *remove()*.
 - ▶ Algoritmi care nu modifică ordinea elementelor în container – non-modifying sequence operations: *for_each()*, *find()*, *count()* și *equal()*.
 - ▶ Algoritmi de sortare și operații similare: *sort()*, *equal_range()*, *merge()* și *includes()*.
 - ▶ Algoritmi generali pentru operații numerice: *min()*, *max()*.



Containere

- ▶ Containerele *secvențiale* sunt colecții liniare și ordonate de date în care accesul se face pe baza poziției elementului în cadrul containerului.
 - ▶ vector (adaugare pe la un singur capăt)
 - ▶ list (adaugare pe la ambele capete)
 - ▶ deque (adăugare pe la ambele capete)
 - ▶ Ordinea elementelor este dată de ordinea în care au fost adăugate
 - ▶ Container *asociative* se diferențiază prin faptul că stocarea elementelor se face pe baza unor chei. Ordinea elementelor este dată de valorile cheilor și relația dintre ele. Accesul este direct prin intermediul cheii.
 - ▶ set
 - ▶ multiset
 - ▶ map
-
- ▶ ▶ multimap

Containere

- ▶ Containere adaptive – adaugă funcționalități containerelor secvențiale.
- ▶ Nu pot fi parcurse cu ajutorul iteratorilor intrucât nu sunt folosite în mod independent
- ▶ Programatorul trebuie să aleagă containerul de bază căruia să îi aplice un container adaptiv.
- ▶ Un container *stack* poate adapta containere *vector*, *list*, *deque*, un container *queue* poate adapta *list* și *deque*, *priority_queue* poate adapta un *vector* sau *deque*.



Containere secvențiale - exemple

```
class CStudent
{
private:
    int nrMat;
    char nume[20];
public:
    CStudent(int nr = 0, char* n = "Student"):nrMat(nr);
    CStudent(const Student& s);
    int GetNrMat(void);
    char* GetNume(void);
    void SetNume(char* n);
    bool operator<(Student& s);
};
```



Containere secvențiale - exemple

```
void main(void){  
    vector<int> vectInt;  
    vectInt.push_back(5);  
    vectInt.push_back(0);  
    vectInt.push_back(15);  
    vectInt.push_back(13);  
    for(int i=0; i<vectInt.size(); i++)  
        cout<<vectInt[i].GetNrMat()<<" "<<  
vectInt[i].GetNume()<<endl;  
    cout<<endl;  
  
    CStudent stud1(1, „Popescu”);  
    CStudent stud2(2, „Marian”);  
    CStudent stud3(3, „Gica”);  
    vector<Student> vectStud;  
    vectStud.push_back(stud1);  
    vectStud.push_back(stud2);  
    vectStud.push_back(stud3);  
    for(int i=0; i<vectStud.size(); i++)  
        cout<<vectStud[i]<<" ";  
    cout<<endl;
```

```
list<int> listInt;
listInt.push_front(-4);
listInt.push_back(15);
listInt.insert(listInt.begin(),19);
listInt.insert(listInt.end(),3);

list<int>::iterator it;
for(it=listInt.begin(); it!=listInt.end(); it++)
    cout<<*it<<" ";
cout<<endl;
listInt.sort();
cout<<"Lista sortata"<<endl;
for(it=listInt.begin(); it!=listInt.end(); it++)
    cout<<*it<<" ";
cout<<endl;
```



```
list<Student> listStud;  
listStud.push_back(stud2);  
listStud.push_front(stud3);  
listStud.insert(listStud.end(), s1);
```

```
list<Student>::iterator it;  
for(it=listStud.begin(); it!=listStud.end(); it++)  
    cout<<(*it).GetNrMat()<<" "<< vectInt[i].GetNume()<<endl;  
cout<<endl;
```

```
listStud.sort();
```

```
for(it=listStud.begin(); it!=listStud.end(); it++)  
    cout<<*it<<" ";  
cout<<endl;
```



Vă mulțumesc !

