

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect.

P00

Curs-6

Gavrilut Dragos

STL (Standard Template Library)

- ▶ Un set divers de template-uri pentru:
 - ▶ Containere (template-uri ale unor clase care pot contine alte clase)
 - ▶ Iteratori (pointeri pentru parcurgerea template-urilor)
 - ▶ Algoritmi (functii care se pot apela pe un container)
 - ▶ Adaptori
 - ▶ Alocatori
 - ▶ Altele
- ▶ Utilizarea STL-ului se face incluzand hedererele specifice template-ului pe care vrem sa il folosim.
- ▶ Pentru usurinta se adauga si `“using namespace std;”`
- ▶ Obiectele din STL se pot apela fie prin numele direct (`“vector<int> x”`) daca folosim `“using namespace std;”` sau prin namespace (`“std::vector<int> x”`) altfel.
- ▶ **Pot exista mai multe implementari pentru obiectele din STL (acestea depind puternic de compilator) → ceea ce inseamna ca si performanta acestora este puternic legata de implementarea lor**

STL-Containere

- ▶ Cu date tinute secvential in memorie
 - ▶ vector
 - ▶ Array
 - ▶ list
 - ▶ forward_list
 - ▶ deque
 - ▶ Adaptorii (stack, queue, priority_queue)
- ▶ Asociativi
 - ▶ Ordonati: set, multiset, map, multimap
 - ▶ Neordonati: unordered_set, unordered_map, unordered_multiset, unordered_multimap

STL - Vector

- ▶ Reprezinta un array uni-dimensional in care se pot tine obiecte
- ▶ Constructori:

App.cpp

```
using namespace std;  
#include <vector>  
  
void main(void)  
{  
    vector<Type> v;  
    vector<Type> v(Size);  
}
```

- ▶ Alocarea obiectelor se face dinamic
- ▶ Pentru elementele adaugate intr-un vector se face o copie
- ▶ Redimensionarea unui obiect de tipul vector se face tot dinamic

STL - Vector

- ▶ Inserarea intr-un vector se face cu urmatoarele comenzi: push_back, insert
- ▶ Pentru elimiare de elemente putem folosi: pop_back, erase sau clear
- ▶ Pentru realocare: metoda resize si reserve
- ▶ Pentru access la elemente: operatorul [] si metoda “at”

App.cpp

```
using namespace std;
#include <vector>

void main(void)
{
    vector<int> v;
    v.push_back(1);v.push_back(2);
    int x = v[1];
    int y = v.at(0);
}
```

STL - Vector

App.cpp

```
using namespace std;
#include <vector>

class Integer
{
    int Value;
public:
    Integer() : Value(0) {}
    Integer(int v) : Value(v) {}
    Integer(const Integer &v) : Value(v.Value) {}
    void Set(int v) { Value = v; }
    int Get() { return Value; }
};

void main(void)
{
    vector<Integer> v;
    Integer i(5);
    v.push_back(i);
    i.Set(6);
    printf("i=%d,v[0]=%d\n", i.Get(), v[0].Get());
}
```

În urma executiei se afiseaza i=6,v[0]=5

STL - Vector

Un exemplu de functionare

App.cpp

```
class Integer
{
    int Value;
public:
    Integer() : Value(0) { printf("[%p] Default ctor\n", this); }
    Integer(int v) : Value(v) { printf("[%p] Value ctor(%d)\n", this,v); }
    Integer(const Integer &v) : Value(v.Value) { printf("[%p] Copy ctor from (%p,%d)\n",
                                                    this, &v, v.Value); }

    Integer& operator= (const Integer& i) { Value = i.Value; printf("[%p] op= (%p,%d)\n",
                                                                    this, &i, i.Value); return *this; }

    void Set(int v) { Value = v; }
    int Get() { return Value; }
};

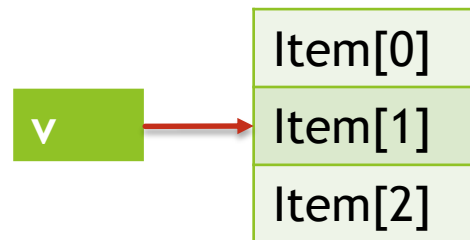
void main(void)
{
    vector<Integer> v;
    Integer i(5);
    for (int tr = 0; tr < 5; tr++)
    {
        i.Set(1000 + tr);
        v.push_back(i);
    }
}
```

STL - Vector

Un exemplu de functionare

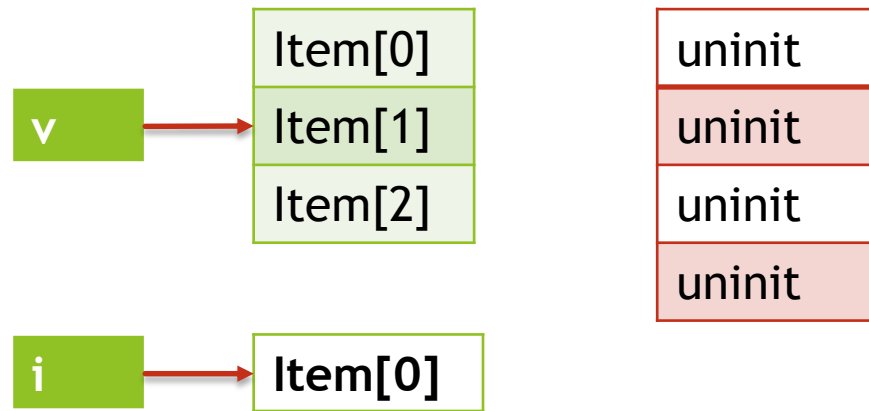
Pas	Output
-	[0098FC90] Value ctor(5)
tr=0	[00D3A688] Copy ctor from (0098FC90,1000)
tr=1	[00D3A6C8] Copy ctor from (00D3A688,1000) [00D3A6CC] Copy ctor from (0098FC90,1001)
tr=2	[00D3A710] Copy ctor from (00D3A6C8,1000) [00D3A714] Copy ctor from (00D3A6CC,1001) [00D3A718] Copy ctor from (0098FC90,1002)
tr=3	[00D3A688] Copy ctor from (00D3A710,1000) [00D3A68C] Copy ctor from (00D3A714,1001) [00D3A690] Copy ctor from (00D3A718,1002) [00D3A694] Copy ctor from (0098FC90,1003)
tr=4	[00D3A6D8] Copy ctor from (00D3A688,1000) ... [00D3A6E8] Copy ctor from (0098FC90,1004)

STL - Vector



`v.push_back(i)`

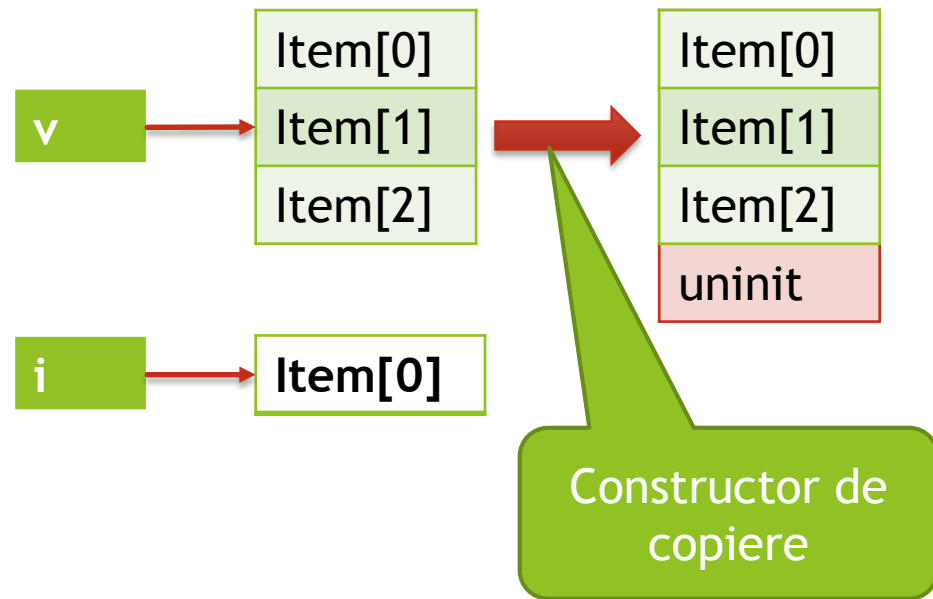
STL - Vector



`v.push_back(i)`

- Aloca spatiu pentru Count+1 elemente

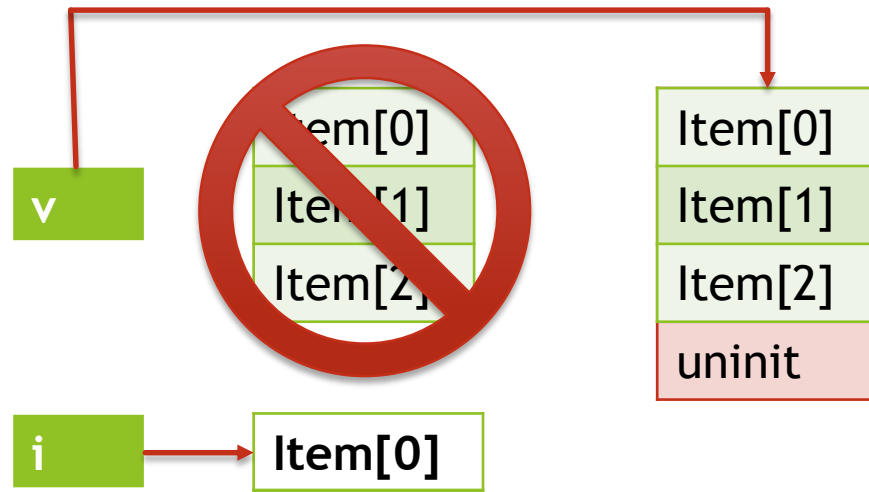
STL - Vector



v.push_back(i)

- ▶ Aloca spatiu pentru Count+1 elemente
- ▶ Copie primele Count elemente din vectorul vechi in cel nou

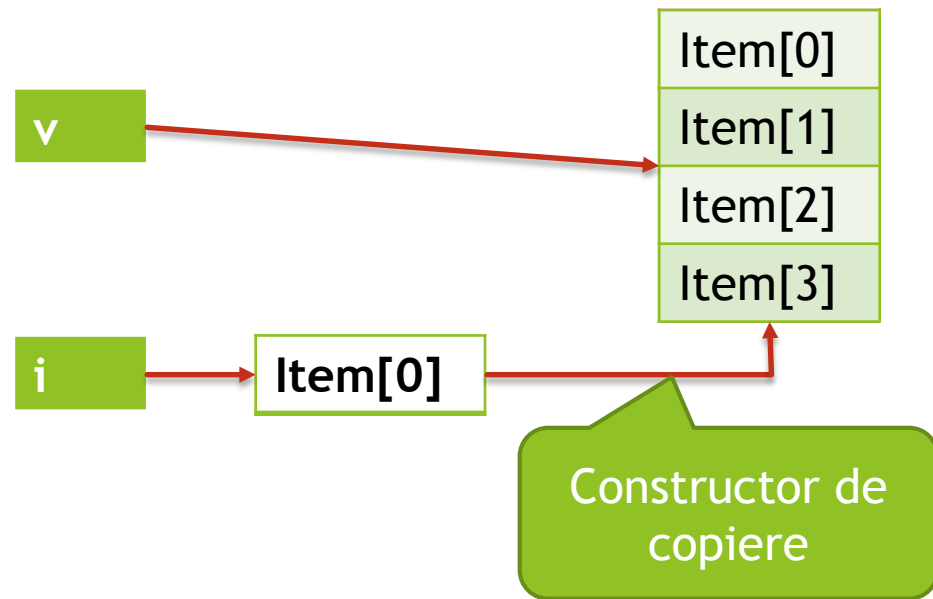
STL - Vector



`v.push_back(i)`

- ▶ Aloca spatiu pentru `Count+1` elemente
- ▶ Copie primele `Count` elemente din vectorul vechi in cel nou
- ▶ Dealloc elementele vechi si leg obiectul `v` de array-ul nou

STL - Vector



`v.push_back(i)`

- ▶ Aloca spatiu pentru `Count+1` elemente
- ▶ Copie primele `Count` elemente din vectorul vechi in cel nou
- ▶ Dealloc elementele vechi si leg obiectul `v` de array-ul nou
- ▶ Copii itemul nou in spatiu alocat pentru el

STL - Vector

Test case:

App-1.cpp

```
#define INTEGER_SIZE 1000
class Integer
{
    int Values[INTEGER_SIZE];
public:
    Integer() { Set(0); }
    Integer(int value) { Set(value); }
    Integer(const Integer &v) { CopyFrom((int*)v.Values); }
    Integer& operator= (const Integer& i) { CopyFrom((int*)i.Values); return *this; }

    void Set(int value)
    {
        for (int tr = 0; tr < INTEGER_SIZE; tr++)
            Values[tr] = value;
    }
    void CopyFrom(int* lista)
    {
        for (int tr = 0; tr < INTEGER_SIZE; tr++)
            Values[tr] = lista[tr];
    }
    void Set(const Integer &v)
    {
        CopyFrom((int*)v.Values);
    }
};
```

STL - Vector

Test case:

App-1.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

App-3.cpp

```
void main(void)
{
    Integer *v = new Integer[100000];
    Integer i;
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v[tr].Set(i);
    }
}
```

App-2.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;
    v.reserve(100000);
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

App-4.cpp

```
void main(void)
{
    Integer **v = new Integer*[MAX];
    for (int tr = 0; tr < MAX; tr++)
    {
        v[tr] = new Integer(tr);
    }
}
```

STL - Vector

Studiul s-a desfasurat in urmatorul fel:

- ▶ Fiecare dintre cele 4 aplicatii au fost executate de 10 ori si s-au masurat timpii in milisecunde
- ▶ S-au masurat doi timpi: timpul de initializare si timpul de setare a datelor in vector

App-1.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;

    for (int tr = 0; tr < 100000; tr++)
    {
        ...
    }
}
```

Durata initializare
vector

Durata setare a
datelor in vector

- ▶ Testele s-au realizat cu urmatoarele specificatii software si hardware:
 - ▶ OS: Windows 8.1 Pro
 - ▶ Compiler: cl.exe [18.00.21005.1 for x86]
 - ▶ Hardware: Dell Latitude 7440 - i7 - 4600U, 2.70 GHz, 8 GB RAM

STL - Vector

Rezultate:

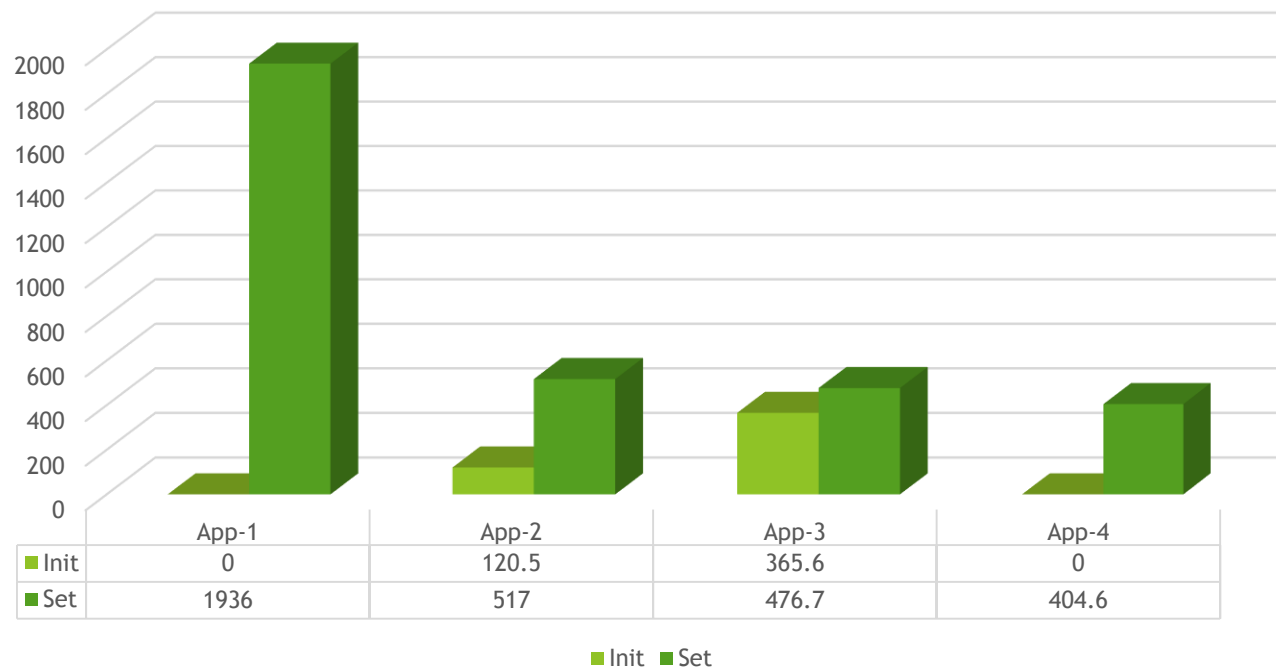
Alg	Timp	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
App 1	Init	0	0	0	0	0	0	0	0	0	0
	Set	1985	1922	1937	1922	1922	1953	1937	1938	1922	1922
App 2	Init	140	125	157	125	109	110	110	109	110	110
	Set	531	515	515	516	516	515	531	516	515	500
App 3	Init	406	359	360	375	359	360	359	359	359	360
	Set	485	485	468	469	485	468	469	485	469	484
App 4	Init	0	0	0	0	0	0	0	0	0	0
	Set	422	453	453	437	375	391	390	375	375	375

- ▶ App (1..4) reprezinta cele 4 metode pe care le-am descris inainte
- ▶ “Init” reprezinta seria de timpi necesari pentru initializare
- ▶ “Set” reprezinta seria de timpi necesari pentru setarea datelor
- ▶ T1 .. T10 sunt timpii rezultati in urma testelor

STL - Vector

Rezultate:

Timpi medii de executie pentru cele 4 metode



- ▶ App (1..4) reprezinta cele 4 metode pe care le-am descris inainte
- ▶ “Init” reprezinta seria de timpi necesari pentru initializare
- ▶ “Set” reprezinta seria de timpi necesari pentru setarea datelor

STL - Vector

- Pentru parcurgerea unui vector se pot folosi iteratori. Iteratorii pot fi vazuti ca un echivalent al unui pointer pentru un element al containerului

App.cpp

```
void main(void)
{
    vector<int> v;
    v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4); v.push_back(5);
    vector<int>::iterator it;
    it = v.begin();
    while (it < v.end())
    {
        printf("%d ", (int)(*it));
        it++;
    }
}
```

- Iteratorii supraincarca mai multi operatori. Operatorul ++,-- , adunare, scadere sunt folosit pentru deplasarea iteratorului.
- Operatorul pointer (*) e folosit pentru conversia (cast) la tipul folosit in template-ul de la vector (in cazul de mai sus, la int)

STL - Vector

- ▶ Accesul la elemente se poate face si print utilizarea operatorilor + si -> dintr-un iterator.
- ▶ Pe langa acest lucru, template-ul vector ofera si doua metoda “front” si “back” prin care se pot accesa primul si ultimul element.

App.cpp

```
class Number
{
public:
    int Value;
    Number() : Value(0) {}
    Number(int val) : Value(val) {}
};

void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));

    vector<Number>::iterator i = v.begin();
    printf("%d\n", i->Value);
    printf("%d\n", (i + 2)->Value);
    printf("%d %d", v.front().Value, v.back().Value);
}
```

- ▶ Exemplul de mai sus afiseaza 1 3 1 5

STL - Vector

- Template-ul “vector” ofera pe langa iteratorul normal si un alt iterator (reverse_iterator) care permite parcurgerea lui in ordine inversa

App.cpp

```
class Number
{
...
};
void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));

    vector<Number>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    vector<Number>::reverse_iterator rit;
    for (rit = v.rbegin(); rit != v.rend(); rit++)
        printf("%d ", rit->Value);
}
```

- Exemplul de mai sus afiseaza “1 2 3 4 5 5 4 3 2 1”

STL - Vector

► Inserarea intr-un vector

App.cpp

```
class Number
{
    ...
};
void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));
    vector<Number>::iterator i = v.begin();

    v.insert(i + 2, Number(10));

    while (i < v.end())
    {
        printf("%d,", i->Value);
        i++;
    }
}
```

► Exemplul de mai sus afiseaza 1,2,10,3,4,5

STL - Vector

- Stergerea elementelor din vector (v.erase(iterator))

App.cpp

```
class Number
{
    ...
};
void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));
    vector<Number>::iterator i = v.begin();
    v.erase(i+2);
    v.erase(i+3);
    while (i < v.end())
    {
        printf("%d,", i->Value);
        i++;
    }
}
```

- Exemplul de mai sus afiseaza 1,2,4

STL - Vector

- ▶ Codul precent functioneaza. Dar in urma stergerii, un iterator se poate invalida. Ideal ar fi de folosit `vector::begin` si `vector::end` in loc sa salvam un iterator intr-o variabila si sa o folosim.

App.cpp

```
class Number
{
    ...
};
void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));
    vector<Number>::iterator i = v.begin();
    v.erase(i);
    v.erase(i+1);
    while (i < v.end())
    {
        printf("%d,", i->Value);
        i++;
    }
}
```

- ▶ Exemplul de mai sus produce o eroare la runtime (“i” este invalidat)

STL - Vector

- ▶ Codul precent functioneaza. Dar in urma stergerii, un iterator se poate invalida. Ideal ar fi de folosit `vector::begin` si `vector::end` in loc sa salvam un iterator intr-o variabila si sa o folosim.

App.cpp

```
class Number
{
    ...
};
void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));
    v.erase(v.begin());
    v.erase(v.begin()+1);
    vector<Number>::iterator i = v.begin();
    while (i < v.end())
    {
        printf("%d,", i->Value);
        i++;
    }
}
```

- ▶ Exemplul de mai sus functioneaza corect

STL - Vector

- ▶ Operatorii de comparatie (==, >=, > , <, !=, <=) sunt suprascrisi
- ▶ Doi vectori sunt egali daca au acelasi numar de elemente si daca operatorul de egalitate intre elementele de pe aceleasi pozitie returneaza true.

App.cpp

```
class Number
{
    ...
};
void main(void)
{
    vector<Number> v1;
    vector<Number> v2;
    for (int tr = 0; tr < 10; tr++)
    {
        v1.push_back(Number(tr));
        v2.push_back(Number(tr));
    }
    if (v1 == v2)
        printf("V1 egal cu V2");
    else
        printf("V2 este diferit de V2");
}
```

- ▶ Exemplul de mai sus nu compileaza pentru nu avem operator== definit in Number

STL - Vector

- In continuare codul nu functioneaza (operatorul ==) trebuie sa fie const la randul lui

App.cpp

```
class Number
{
    bool operator==(const Number &n1) { return Value == n1.Value; }
};
void main(void)
{
    vector<Number> v1;
    vector<Number> v2;
    for (int tr = 0; tr < 10; tr++)
    {
        v1.push_back(Number(tr));
        v2.push_back(Number(tr));
    }
    if (v1 == v2)
        printf("V1 egal cu V2");
    else
        printf("V2 este diferit de V2");
}
```

STL - Vector

- ▶ Codul compileaza si functioneaza core.
- ▶ Operatorul de egalitate se poate inlocui si cu o functie friend de felul urmator: “`bool friend operator==(const Number & n1, const Number & n2);`”

App.cpp

```
class Number
{
    bool operator==(const Number &n1) const { return Value == n1.Value; }
};
void main(void)
{
    vector<Number> v1;
    vector<Number> v2;
    for (int tr = 0; tr < 10; tr++)
    {
        v1.push_back(Number(tr));
        v2.push_back(Number(tr));
    }
    if (v1 == v2)
        printf("V1 egal cu V2");
    else
        printf("V2 este diferit de V2");
}
```

STL - Vector

- Verificarea operatorilor < sau > se face in felul urmator:
Codul de mai jos e un pseudocod care arata cum se face comparatia pentru operatorul <

App.cpp

```
Function compare(Vector v1,Vector v2)
    Size = MINIM(v1.Size,v2.Size)
    for (i=0;i<Size;i++)
        if (v1[i]<v2[i])
            return "v1 mai mic ca v2";
        end if
    end for
    if (v1.Size<v2.Size)
        return "v1 mai mic ca v2";
    end if
    return "v1 NU este mai mic ca v2";
End Function
```

STL - Vector

- Verificarea operatorilor < sau > se face in felul urmator:

App.cpp

```
class Number
{
    bool operator<(const Number &n1) const { return Value < n1.Value; }
};
void main(void)
{
    vector<Number> v1;
    vector<Number> v2;

    v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));
    v2.push_back(Number(1)); v2.push_back(Number(2)); v2.push_back(Number(4));

    if (v1 < v2)
        printf("OK");
    else
        printf("NOT-OK");
}
```

- Codul compileaza si afiseaza “OK” pentru ca v1[2]=3 si v2[2]=4, iar 3<4

STL - Vector

- Codul de mai jos NU compileaza.

App.cpp

```
class Number
{
    bool operator>(const Number &n1) const { return Value < n1.Value; }
};
void main(void)
{
    vector<Number> v1;
    vector<Number> v2;

    v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));
    v2.push_back(Number(1)); v2.push_back(Number(2)); v2.push_back(Number(4));

    if (v1 > v2)
        printf("OK");
    else
        printf("NOT-OK");
}
```

- Template-ul vector are nevoie doar de implementarea operatorului < (daca definim operatorul > dar nu si <) codul nu compileaza. Similar si pentru “!=“ (se foloseste operator == si nu !=)

STL - Vector

- Vectorii se pot asigna intre ei

App.cpp

```
class Number
{
    ...
};
void main(void)
{
    vector<Number> v1;
    vector<Number> v2;

    v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));
    v2 = v1;
    for (int i = 0; i < v2.size(); i++)
        printf("%d ", v2[i].Value);
}
```

- Codul compileaza si afiseaza 1 2 3

STL - Vector

- ▶ Metode din template-ul vector care se pot folosi pentru a obtine informatii despre statusul unui obiect:
 - ▶ `size()` → numarul de elemente din vector
 - ▶ `capacity()` → spatial prealocat pentru un vector
 - ▶ `max_size()` → numarul maxim de elemente care pot exista in acel vector
 - ▶ `empty()` → “true” daca vectorul nu are nici un element
 - ▶ `data()` → pointer catre elementele de tipul “Type” folosit in template.

App.cpp

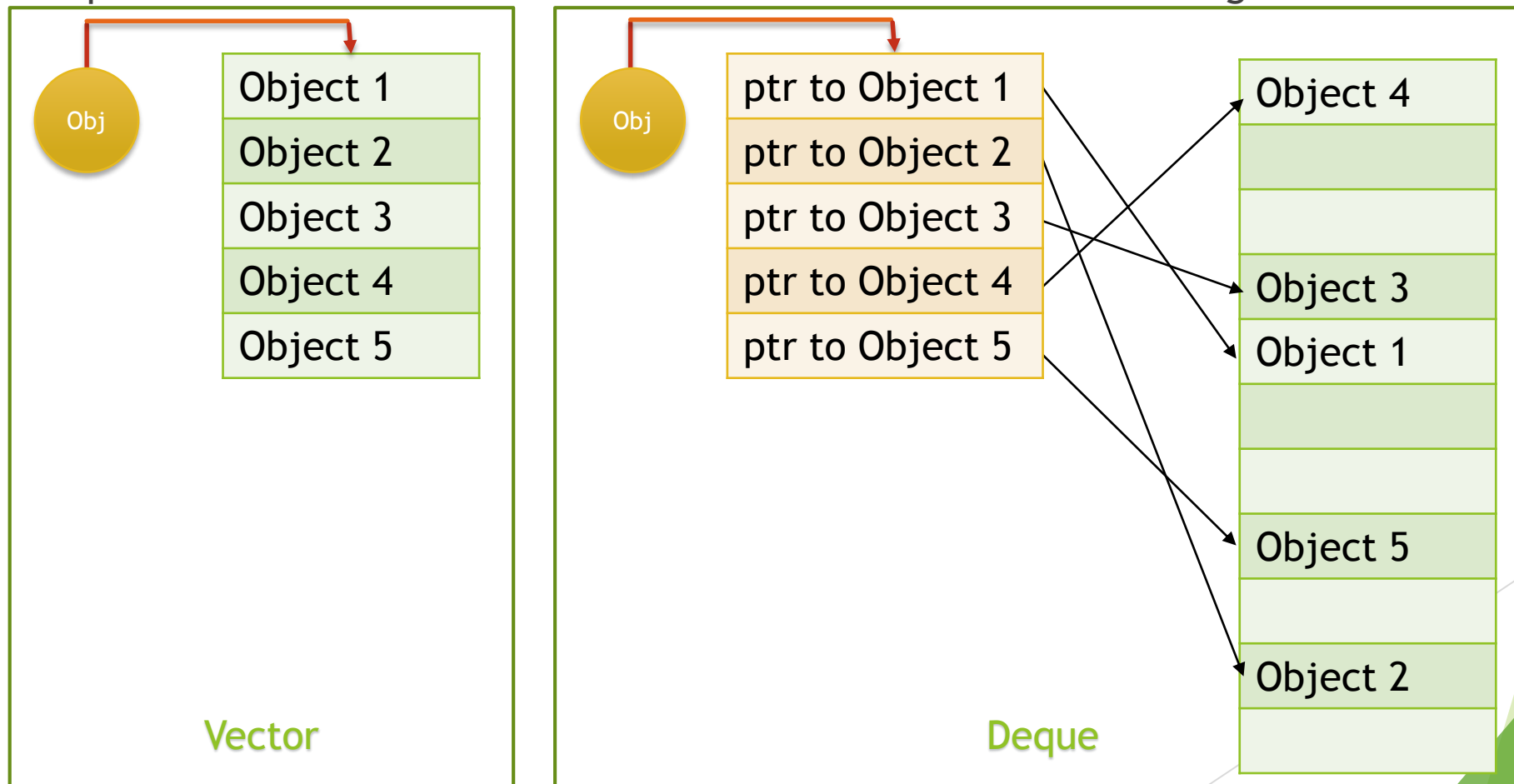
```
class Number { ... };  
void main(void)  
{  
    vector<Number> v;  
    v.push_back(Number(0)); v.push_back(Number(1));  
    printf("%d ", v.max_size());  
    printf("%d ", v.size());  
    printf("%d ", v.capacity());  
    Number *n = v.data();  
    printf("[%d %d]", n[0].Value, n[1].Value);  
}
```

STL - Deque

- ▶ Template-ul “deque” reprezinta un container foarte similar cu “vector”.
- ▶ Diferenta de baza este ca elementele nu sunt tinute consecutive in memorie. Din acest motiv metoda “data” nu este disponibila pentru un container deque
- ▶ Acelasi lucru se poate spune si despre metodele reserve si capacity
- ▶ In schimb apar metode noi (push_front si pop_front)
- ▶ Restul metodelor sunt identice ca si comportament cu cele din clasa “vector”
- ▶ Pentru utilizare “**#include <deque>**”

STL - Deque

Comparatie intre metoda prin care template-ul “vector” tine datele in memorie fata de template-ul “deque”. Exemplul are rol ilustrativ - in realitate un “deque” tine mai multe blocuri de memorie la care le face el management.



STL - Deque

Daca reluam experimentul precedent - doar ca de data asta folosim si “deque”

App-1.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

App-2.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;
    v.reserve(100000);
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

App-5.cpp

```
void main(void)
{
    deque<Integer> v;
    Integer i;
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

STL - Deque

Rezultate:

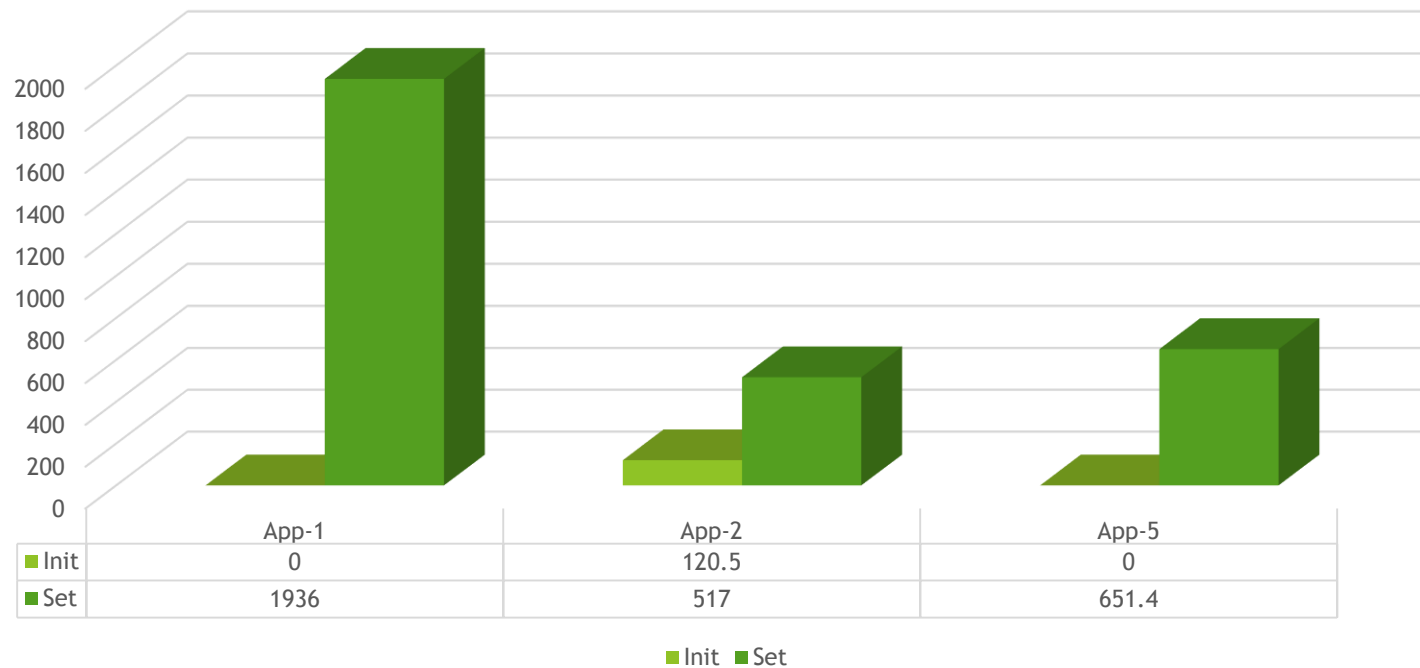
Alg	Timp	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
App 1	Init	0	0	0	0	0	0	0	0	0	0
	Set	1985	1922	1937	1922	1922	1953	1937	1938	1922	1922
App 2	Init	140	125	157	125	109	110	110	109	110	110
	Set	531	515	515	516	516	515	531	516	515	500
App 5	Init	0	0	0	0	0	0	3509	0	0	0
	Set	687	657	656	640	656	640	656	641	641	640

- ▶ App (1,2) reprezinta cele 2 metode care folosesc template-ul vector
- ▶ App (5) reprezinta metoda care foloseste template-ul deque
- ▶ “Init” reprezinta seria de timpi necesari pentru initializare
- ▶ “Set” reprezinta seria de timpi necesari pentru setarea datelor
- ▶ T1 .. T10 sunt timpii rezultati in urma testelor

STL - Deque

Rezultate:

Timpi de executie pe cele 3 metode



- ▶ App (1,2) reprezinta cele 2 metode care folosesc template-ul vector
- ▶ App (5) reprezinta metoda care foloseste template-ul deque
- ▶ “Init” reprezinta seria de timpi necesari pentru initializare
- ▶ “Set” reprezinta seria de timpi necesari pentru setarea datelor

STL - Array

- ▶ Template-ul “array” a fost introdus in C++11 in STL.
- ▶ Reprezinta un vector cu dimensiune fixa
- ▶ Are aproximativ acelasi support ca si clasa vector (iteratori, operatori supraincarcati, etc)
- ▶ Nu are functiile de adaugare (fiind vector fix nu are sens)
- ▶ Nu are functii pentru stergerea elementelor (din acelasi motiv)
- ▶ Are in schimb o serie de functii noi pe care clasa vector nu le are (fill)
- ▶ Pentru utilizare “**#include <array>**”

STL - Array

- ▶ Exemplu de utilizare a template-ului array

App.cpp

```
class Number
{
    ...
};
void main(void)
{
    array<Number,5> v;
    v.fill(Number(2));
    for (int tr = 0; tr < v.size(); tr++)
        printf("%d ", v[tr].Value);

    for (array<Number, 5>::iterator it = v.begin(); it < v.end(); it++)
        it->Value = 10;

    printf("%d ", v.at(3).Value);
}
```

- ▶ Din cauza ca alocarea este predefinita, “array” este echivalent cu un template de tipul “vector” care foloseste metoda resize()

STL - Array & vector

- ▶ Atat template-ul array cat si template-ul vector au doua metode de acces la elemente: operatorul[] si metoda “at”.
- ▶ Ambele intorc o referinta catre obiect si au doua forme:
 - ▶ `Type& operator[](size_type)`
 - ▶ `const Type& operator[](size_type) const`
 - ▶ `Type& at(size_type)`
 - ▶ `const Type& at(size_type)`
 - ▶ `size_type` este definit ca un `size_t` (`typedef size_t size_type;`)
- ▶ Exista insa diferente intre cele doua implementari (operator[] si metoda “at”)

STL - Array & vector

- Operator[] si metoda “at” din template-ul array

App.cpp

```
reference at(size_type _Pos)
{
    if (_Size <= _Pos)
        _Xran();
    return (_Elems[_Pos]);
}
```

App.cpp

```
reference operator[](size_type _Pos)
{
    #if _ITERATOR_DEBUG_LEVEL == 2
        if (_Size <= _Pos)
            _DEBUG_ERROR("array subscript out of range");
    #elif _ITERATOR_DEBUG_LEVEL == 1
        _SCL_SECURE_VALIDATE_RANGE(_Pos < _Size);
    #endif
    _Analysis_assume_(_Pos < _Size);
    return (_Elems[_Pos]);
}
cu
#define _Analysis_assume_(expr) // pentru release
```

Metoda compilare	Valoare _ITERATOR_DEBUG_LEVEL
Release	0
Release (_SECURE_SCL)	1
Debug	2

STL - List

- ▶ Template-ul “list” reprezinta o lista dublu inlantuita
- ▶ Elementele nu mai sunt tinute intr-un spatiu continuu ci in functie de cum au fost allocate
- ▶ Accesul la elemente este posibil doar prin parcurgerea listei (prin iteratori). Se pot accesa rapid primul si ultimul element.
- ▶ Iteratorul din lista nu implementeaza operatorul (<) - elementele nefind in ordine in memorie nu se pot compara. Implementeaza operatorul == si implicit !=.
- ▶ Iteratorul din lista nu implementeaza nici operatorul + sau - ci doar ++ sau --
- ▶ Metode pentru adaugare de elemente: push_back si push_front
- ▶ Pentru stergerea elementelor se poate folosi erase sau pop_front, pop_back
- ▶ Suporta si o serie de metode noi: merge, splice (lucru cu alte liste)
- ▶ Pentru utilizare “**#include <list>**”

STL - List

- ▶ Exemplu de utilizare a template-ului list

App.cpp

```
class Number { ... };
void main(void)
{
    list<Number> v;
    v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
    v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

    list<Number>::iterator it;
    for (it = v.begin(); it < v.end(); it++)
        printf("%d ", it->Value);

    it = v.begin()+3;
    v.insert(it, Number(20));
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = ++v.begin();
    v.erase(it);
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);
}
```

- ▶ Codul nu compileaza pentru ca iteratorul din lista nu suporta comparatie < sau >

STL - List

- ▶ Exemplu de utilizare a template-ului list

App.cpp

```
class Number { ... };
void main(void)
{
    list<Number> v;
    v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
    v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

    list<Number>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = v.begin()+3;
    v.insert(it, Number(20));
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = ++v.begin();
    v.erase(it);
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);
}
```

- ▶ Codul nu compileaza pentru ca iteratorul din lista nu suporta operatia de adunare

STL - List

► Exemplu de utilizare a template-ului list

App.cpp

```
class Number { ... };
void main(void)
{
    list<Number> v;
    v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
    v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

    list<Number>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = v.begin(); it++; it++; it++;
    v.insert(it, Number(20));
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = ++v.begin();
    v.erase(it);
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);
}
```

► Codul compileaza si afiseaza: “5 4 3 0 1 2 5 4 3 20 0 1 2 5 3 20 0 1 2”

STL - Forward List

- ▶ Template-ul “forward_list” reprezinta o lista simplu inlantuita. A fost adaugat in C++11 in STL
- ▶ Se respecta aproximativ aceleasi considerenta ca si la lista dublu inlantuita.
- ▶ Iteratorul insa nu mai are supraincarcat operatorul-- ci doar ++ (lista este unidirectionala).
- ▶ Dispar si anumite metode care se regaseau in list: push_back si pop_back
- ▶ Pentru utilizare “**#include <forward_list>**”

STL - Forward List

- ▶ Exemplu de utilizare a template-ului forward_list

App.cpp

```
class Number { ... };  
void main(void)  
{  
    forward_list<Number> v;  
    v.push_front(Number(0)); v.push_front(Number(1)); v.push_front(Number(2));  
  
    forward_list<Number>::iterator it;  
    for (it = v.begin(); it != v.end(); it++)  
        printf("%d ", it->Value);  
  
    it = v.begin();  
    it++; it++;  
    v.insert_after(it, Number(20));  
    for (it = v.begin(); it != v.end(); it++)  
        printf("%d ", it->Value);  
  
    it = ++v.begin();  
    v.erase_after(it);  
    for (it = v.begin(); it != v.end(); it++)  
        printf("%d ", it->Value);  
}
```

- ▶ Codul compileaza si afiseaza: “2 1 0 2 1 0 20 2 1 20”

STL

Metod	vector	Deque	array	list	forward_list
Access elemente	operator[] .at() .front() .back() .data()	operator[] .at() .front() .back()	operator[] .at() .front() .back() .data()	.front() .back()	.front()
Iteratori	.begin() .end()	.begin() .end()	.begin() .end()	.begin() .end()	.begin() .end()
Reverse Iterator	.rbegin() .rend()	.rbegin() .rend()	.rbegin() .rend()	.rbegin() .rend()	
Information	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .max_size()

STL

Metod	vector	Deque	array	list	forward_list
Dimensiune Capacitate	.resize() .reserve() .capacity()	.resize()		.resize()	.resize()
Adaugare	.push_back() .insert()	.push_back() .push_front() .insert()		.push_back() .push_front() .insert()	.push_front() .insert_after()
Stergere	.clear() .erase() .pop_back()	.clear() .erase() .pop_back() .pop_front()		.clear() .erase() .pop_back() .pop_front()	.clear() .pop_front() .erase_after()

STL - Adaptorii

- ▶ Adaptorii nu sunt containere - in sensul ca nu au ei o implementare prin care sa poata sa stocheze datele intr-un anumit fel
- ▶ In schimb, adaptorii folosesc alti containeri care au aceasta proprietate pentru a stoca datele
- ▶ Adaptorii deasemenea, functioneaza doar daca containerul pe care il folosesc implementeaza anumite functii
- ▶ Adaptorii NU au iteratori. In schimb se pot folosi iteratorii containerului pe care il folosesc
- ▶ Adaptorii:
 - ▶ stack
 - ▶ queue
 - ▶ priority_queue

STL - Stack

- ▶ Adaptorul care este o expresie pentru o stiva (LIFO - Last In First Out)
- ▶ Pentru utilizare “**#include <stack>**”
- ▶ Containerul implicit este “deque” (in exemplu de mai jos “s” foloseste deque iar “s2” vector)

App.cpp

```
void main(void)
{
    stack<int> s;
    s.push(10); s.push(20); s.push(30);

    stack<int, vector<int>> s2;
    s2.push(10); s2.push(20); s2.push(30);
}
```

- ▶ Implementeaza urmatoarele metode: push, pop, top, empty, size
- ▶ Implementeaza si o metoda “_Get_container” prin care putem crea iteratori pe baza containerului folosit

STL - Queue

- ▶ Adaptorul care este o expresie pentru o coada (FIFO - First In First Out)
- ▶ Pentru utilizare “**#include <queue>**”
- ▶ Containerul implicit este “deque” (in exemplu de mai jos “s” foloseste deque iar “s2” list)

App.cpp

```
void main(void)
{
    queue<int> s;
    s.push(10); s.push(20); s.push(30);

    queue<int, list<int>> s2;
    s2.push(10); s2.push(20); s2.push(30);
}
```

- ▶ Implementeaza urmatoarele metode: push, pop, back, empty, size
- ▶ Implementeaza si o metoda “_Get_container” prin care putem crea iteratori pe baza containerului folosit

STL - Priority Queue

- ▶ Adaptorul care este o expresie pentru o coada in care fie fiecare element are o prioritate asociata lui. Elementele sunt ordonate dupa prioritate in coada.
- ▶ Pentru utilizare “**#include <queue>**”
- ▶ Containerul implicit este “vector”

App.cpp

```
void main(void)
{
    priority_queue<int> s;
    s.push(10); s.push(5); s.push(20); s.push(15);
    while (s.empty() == false)
    {
        printf("%d ", s.top());
        s.pop();
    }
}
```

- ▶ Codul de mai sus afiseaza “20 15 10 5”
- ▶ Implementeaza urmatoarele metode: push, pop, top, empty, size
- ▶ Implementeaza si o metoda “_Get_container” prin care putem crea iteratori pe baza containerului folosit

STL - Priority Queue

- Un priority_queue poate primi si o clasa care trebuie sa implementeze “operatorul ()” si care sa fie folosita pentru comparatie.

App.cpp

```
class CompareModule
{
    int modValue;
public:
    CompareModule(int v) : modValue(v) {}
    bool operator() (const int& v1, const int& v2) const
    {
        return (v1 % modValue) < (v2 % modValue);
    }
};

void main(void)
{
    priority_queue<int, vector<int>, CompareModule> s(CompareModule(3));
    s.push(10); s.push(5); s.push(20); s.push(15);
    while (s.empty() == false)
    {
        printf("%d ", s.top());
        s.pop();
    }
}
```

- Exemplul de mai sus afiseaza “5 20 10 15”

STL - Adattori

Metoda	stack	queue	Priority_queue
push	Container.push_back	Container.push_back	Container.push_back
pop	Container.pop_back	Container.pop_front	Container.pop_back
top	Container.back	N/A	Container.front
back	N/A	Container.back	N/A
size	Container.size	Container.size	Container.size
empty	Container.empty	Container.empty	Container.empty

Adattori	vector	deque	list	array	forward_list
stack	DA	DA	DA	NU	NU
queue	NU	DA	DA	NU	NU
priority_queue	DA	DA	NU	NU	NU