

Curs 5

- Template (Programare generica)
- STL – Standard Template Library
- Tratarea exceptiilor in C++

Curs 4

- C++ Core Guidelines
- Clase si obiecte
- Clase predefinite: string, vector
- Template

C++ Core Guideline Checker

Linter/Code analyzer: software care analizează codul sursa (code analysis) a unui program și semnalează automat erori de programare, buguri, cod suspect, probleme de formatare, etc.

Activăți: Proiect->Properties->Code Analysis -> Enable Code Analysis on Build

Selectați checkere: Proiect->Properties->Code Analysis->Microsoft-> Active rules combo

<https://docs.microsoft.com/en-us/cpp/code-quality/using-the-cpp-core-guidelines-checkers?view=msvc-160>

Puteți alege ce seturi de reguli (guideline) să se verifice, puteți aplica multiple reguli (în combo selectați „Choose multiple rule set”)

Pentru reguli din C++ core Guideline puteți selecta toate seturile de reguli cu numele: „Cpp Core....”

La compilare se efectuează analiza codului, se raportează warninguri pentru încălcări de reguli din guideline:

<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>. Erorile se vad în fereastra „Error List”.

Ex: **Avoid calling new and delete explicitly, use std::make_unique<T> instead (r.11**
<http://go.microsoft.com/fwlink/?LinkId=845485>).

Warningul conține un sumar plus un link către regula din guideline.

Fiecare regula are explicații, exemple de cod, motivație, soluție propusă pentru problema rezolvare.

Este o metodă bună pentru:

- Explora bunele practici în scrierea de aplicații industriale
- exploră guideline-ul și învăță despre bunele practici în scrierea de cod C++ Modern
- Explora diferite alternative disponibile în C++
- modernizează cod C++ existent

Pentru alte platforme: puteți folosi clang-tidy: <http://clang.llvm.org/extra/clang-tidy/>

Funcții/clase parametrizate - Template - programare generică

- permite crearea de cod generic
- în loc să repetăm implementarea unei funcții pentru fiecare tip de date, putem crea o funcție parametrizată după una sau mai multe tipuri
- o metodă convenientă de reutilizare de cod și de scriere de cod generic
- codul C++ se generează automat înainte de compilare, înlocuind parametru template cu tipul efectiv.
- Important: În cazul claselor metodelor care folosesc template tot codul ar trebui scris în fișierul .h Implementarea efectiva este generată de compilator

Function template:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

```
int sum(int a, int b) {                                template<typename T> T sumTemp(T a, T b) {  
    return a + b;                                     return a + b;  
}  
  
double sum(double a, double b) {  
    return a + b;  
}  
  
int sum = sumTemp<int>(1, 2);  
cout << sum;  
double sum2 = sumTemp<double>(1.2, 2.2);  
cout << sum2;
```

- T este parametru template (template parameter), este un tip de date, argument pentru funcția sum
- Instantierea templatului → crearea codului efectiv înlocuind T cu tipul int:
 - `int sum = sumTemp<int>(1, 2);`

Class template:

Putem parametriza o clasa după unu sau mai multe tipuri

Templatul este ca o matrice, înlocuind parametrul template cu un tip de date se obține codul c++, in acest caz o clasa.

```
template<typename ElementType>
class DynamicArray {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * e - is a generic element
     */
    void addE( ElementType r);
    /**
     * Delete the element from the given position
     * poz - the position of the elem to be deleted, poz>=0;poz<size
     * returns the deleted element
     */
    ElementType& deleteElem(int poz);

    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)
     */
    ElementType& get(int poz);
    /**
     * Give the size of the array
     * return the number of elements in the array
     */
    int getSize();
    /**
     * Clear the array
     * Post: the array will contain 0 elements
     */
    void clear();
private:
    ElementType *elems;
    int capacity;
    int size;
};
```

In general parametrizarea se face după un tip, dar putem avea și parametri valoare pentru un template

```
//in fisierul buffer.h                                     //instantiere
template<typename T,int N>                                Buffer<Pet, 10> buff;
class Buffer {
private:
    T elems[N];
public:
    T& operator[](int poz);
};

//in cazul calaselor template
//inclusiv definitiile se pun in header
template<typename T, int N>
T& Buffer<T, N>::operator[](int poz) {
    if (poz < 0 || poz >= N) {
        ...
    }
    return elems[poz];
}
```

Programare generica

Mecanismul de template permite:

- parametrizare după un tip (sau chiar o valoare) fără a pierde din precizie.
Permite scrierea de algoritmi generali (independent de tipul datelor)
- Verificare de tip întârziata. Se verifica la compilare în momentul instantiei templatului daca tipul primit ca parametru template are metodele dorite (asemănător cu duck typing dar e la compilare)
- Posibilitatea de a transmite constante si de a face calcule in timpul compilării
- codul rezultat este eficient (la instantierea templatului se generează cod C++ care este compilat/optimizat de compilator ca si orice cod scris de programator)

Programare generica se referă la crearea de algoritmi generali unde prin general se înțelege ca algoritmul poate lucra cu orice tipuri de date care satisfac un set de cerințe (au un set de operații)

Tipuri abstracte de date (Abstract Data Types)

ADT

- separat interfața (ce vede cel care folosește) de implementare (cum e implementat)
- specificații abstrakte (fără referire la detaliu de implementare)
- ascundere detaliu de implementare (data protection)

Clase

- header: conține declarația de clasa / metode
- specificații pentru fiecare metoda
- folosind modifierul private , reprezentarea (câmpurile clasei), metodele care sunt folosite doar inter pot fi protejate de restul aplicației (nu sunt vizibile în afara clasei)

Exemplu: Variante de vector dinamic generic (acomodează orice tip de element)

- `typedef Telem = <type name>`
 - nu pot avea în același program liste pentru 2 sau mai multe tipuri de elemente (int, Rational)
- implementare cu `void*`
 - nu pot adăuga constante (1, 3.5) pot adăuga doar adrese
 - gestiunea memoriei devine mai dificilă (similar cu varianta C)
 - în multe locuri trebuie să folosesc `cast`
 - pot adăuga în același lista adrese la elemente de tipuri diferite
- implementare cu template
 - elimina neajunsurile abordărilor anterioare
 - lista poate conține atât adrese ca și obiectele, pot adăuga valori simple
 - pot instanția clasa template pentru oricâte tipuri

Atribute statice in clasa (câmpuri/metode).

Atributele statice dintr-o clasa aparțin clasei nu instanței (obiectelor)

Caracterizează clasa, nu face parte din starea obiectelor

Ne referim la ele folosind operatorul scope “::”

Sunt asemănătoare variabilelor globale doar ca sunt definite în interiorul clasei – retine o singură valoare chiar dacă am multiple obiecte

Obs: Variabilele statice trebuie inițializate în fișierul .cpp

Keyword : **static**

```
/*
 * New data type to store rational numbers
 * we hide the data representation
 */
class Rational {
public:
    /**
     * Get the nominator
     */
    int setUp();
    /**
     * get the denominator
     */
    int getDown();

    // functie statica
    static int getNrInstance(){
        return nrInstances;
    }

private:
    int a;
    int b;
    // declarare membru static
    static int nrInstances;
};
```

```
// in cpp
// initializare membru static (obligatoriu în cpp dacă nu este const)
int Rational:: nrInstances =0;
```

Clase/functii **friend**.

- **friend** permite accesul unei funcții sau clase la câmpuri/metode private dintr-o clasa
- O metoda controlata de a încalcă încapsularea
- punând declarația funcției precedat de **friend** în clasa, funcția are acces la membrii privați ai clasei
- Funcția **friend** nu este membru a clasei (nu are acces la this), are doar acces la membrii privați din clasa
- O clasa B este **friend** cu class of class A daca are acces la membri privati al lui A. Se declara clasa cu cuvântul rezervat **friend** în fata.

Clasa friend

```
class ItLista {  
public:  
    friend class Lista;  
...  
template<typename E>  
class Set {  
    friend class Set_iterator<E> ;
```

Functie friend

```
class List {  
public:  
    friend void friendMethodName(int param);
```

Când folosim **friend**

putem folosi la supraincarcarea operatorilor:

```
class AClass {  
private:  
    friend ostream& operator<<(ostream& os, const AClass& ob);  
    int a;  
};  
ostream& operator<<(ostream& os, const AClass& ob) {  
    return os << ob.a;  
}
```

Util si pentru:

```
class AClass {  
public:  
    AClass operator+(int nr); //pentru: AClass a;  a+7  
private:  
    int a;  
    friend AClass operator+(int nr, const AClass& ob); //pentru: AClass a;  7+a  
};
```

Standard Template Library (STL)

- The Standard Template Library (STL), este o bibliotecă de clase C++, parte din C++ Standard Library
- Oferă structuri de date și algoritmi fundamentali, folosiți la dezvoltarea de programe în C++
- STL oferă componente generice, parametrizabile. Aproape toate clasele din STL sunt parametrizate (Template).
- STL a fost conceput astfel încât componentele STL se pot compune cu ușurință fără a sacrifica performanță (generic programming)
- STL conține clase pentru:
 - containere, iteratori
 - algoritmi, function objects
 - allocators

Selected Standard Library Headers	
<code><algorithm></code>	<code>copy(), find(), sort()</code>
<code><array></code>	<code>array</code>
<code><chrono></code>	<code>duration, time_point</code>
<code><cmath></code>	<code>sqrt(), pow()</code>
<code><complex></code>	<code>complex, sqrt(), pow()</code>
<code><forward_list></code>	<code>forward_list</code>
<code><iostream></code>	<code>fstream, ifstream, ofstream</code>
<code><future></code>	<code>future, promise</code>
<code><ios></code>	<code>hex, dec, scientific, fixed, defaultfloat</code>
<code><iostream></code>	<code>istream, ostream, cin, cout</code>
<code><map></code>	<code>map, multimap</code>
<code><memory></code>	<code>unique_ptr, shared_ptr, allocator</code>
<code><random></code>	<code>default_random_engine, normal_distribution</code>
<code><regex></code>	<code>regex, smatch</code>
<code><string></code>	<code>string, basic_string</code>
<code><set></code>	<code>set, multiset</code>
<code><sstream></code>	<code>istrstream, ostrstream</code>
<code><stdexcept></code>	<code>length_error, out_of_range, runtime_error</code>
<code><thread></code>	<code>thread</code>
<code><unordered_map></code>	<code>unordered_map, unordered_multimap</code>
<code><utility></code>	<code>move(), swap(), pair</code>
<code><vector></code>	<code>vector</code>

* A tour of c++, Bjarne Stroustrup

Containeri

Un container este o grupare de date în care se pot adăuga (insera) și din care se pot șterge (extrage) obiecte. Implementările din STL folosesc şablonane ceea ce oferă o flexibilitate în ceea ce privește tipurile de date ce sunt suportate

Containerul gestionează memoria necesară stocării elementelor, oferă metode de acces la elemente (direct și prin iteratori)

Toate containerele oferă funcționalități (metode):

- accesare elemente (ex.: `[]`)
- gestiune capacitate (ex.: `size()`)
- modificare elemente (ex.: `insert`, `clear`)
- iterator (begin(), end())
- alte operații (ie: `find`)

Decizia în alegerea containerului potrivit pentru o problemă concretă se bazează pe:

- funcționalitățile oferite de container
- eficiența operațiilor (complexitate).

Containere - Clase template

- Container de tip secvență (Sequence containers): `vector<T>`, `deque<T>`, `list<T>`
- Adaptor de containere (Container adaptors): `stack<T, ContainerT>`, `queue<T, ContainerT>`, `priority_queue<T, ContainerT, CompareT>`
- Container asociativ (Associative containers): `set<T, CompareT>`, `multiset<T, CompareT>`, `map<KeyT, ValueT, CompareT>`, `multimap<KeyT, ValueT, CompareT>`, `bitset<T>`

Standard Container Summary	
<code>vector<T></code>	A variable-size vector (§9.2)
<code>list<T></code>	A doubly-linked list (§9.3)
<code>forward_list<T></code>	A singly-linked list
<code>deque<T></code>	A double-ended queue
<code>set<T></code>	A set (a <code>map</code> with just a key and no value)
<code>multiset<T></code>	A set in which a value can occur many times
<code>map<K,V></code>	An associative array (§9.4)
<code>multimap<K,V></code>	A map in which a key can occur many times
<code>unordered_map<K,V></code>	A map using a hashed lookup (§9.5)
<code>unordered_multimap<K,V></code>	A multimap using a hashed lookup
<code>unordered_set<T></code>	A set using a hashed lookup
<code>unordered_multiset<T></code>	A multiset using a hashed lookup

* A tour of C++, Bjarne Stroustrup

Container de tip secvență (Sequence containers):

Vector, Deque, List sunt containere de tip secvență, folosesc reprezentări interne diferite, astfel operațiile uzuale au complexități diferite

- Vector (Dynamic Array):
 - elementele sunt stocate secvențial în zone continue de memorie
 - Vector are performanțe bune la:
 - Accesare elemente individuale de pe o poziție dată (constant time).
 - Iterare elemente în orice ordine (linear time).
 - Adăugare/Ștergere elemente de la sfârșit (constant amortized time).
- Deque (double ended queue) - Coadă dublă (completă)
 - elementele sunt stocate în blocuri de memorie (chunks of storage)
 - Elementele se pot adăuga/șterge eficient de la ambele capete
- List
 - implementat ca și listă dublă înlățuită
 - List are performanțe bune la:
 - Ștergere/adăugare de elemente pe orice poziție (constant time).
 - Mutarea de elemente sau secvențe de elemente în liste sau chiar și între liste diferite (constant time).
 - Iterare de elemente în ordine (linear time).

Operații / complexitate

<pre>#include <vector> void sampleVector() { vector<int> v; v.push_back(4); v.push_back(8); v.push_back(12); v[2] = v[0] + 2; int lg = v.size(); for (int i = 0; i<lg; i++) { cout << v.at(i) << " "; } }</pre>	<pre>#include <deque> void sampleDeque() { deque<double> dq; dq.push_back(4); dq.push_back(8); dq.push_back(12); dq[2] = dq[0] + 2; int lg = dq.size(); for (int i = 0; i<lg; i++) { cout << dq.at(i) << " "; } }</pre>	<pre>#include <list> void sampleList() { list<double> l; l.push_back(4); l.push_back(8); l.push_back(12); while (!l.empty()) { cout << " " << l.front(); l.pop_front(); } }</pre>
--	--	---

Vector : timp constant O(1) random access; insert/delete de la sfârșit

Deque: timp constant O(1) insert/delete la oricare capat

List: timp constant O(1) insert / delete oriunde în listă

Vector vs Deque

- Accesul la elemente de pe orice poziție este mai eficient la vector
- Inserare/ștergerea elementelor de pe orice poziție este mai eficient la Deque (dar nu e timp constant)
- Pentru liste mari Vector aloca zone mari de memorie, deque aloca multe zone mai mici de memorie – Deque este mai eficient în gestiunea memoriei

Container asociativ (Associative containers):

Sunt eficiente în accesare elementelor folosind chei (nu folosind poziții ca și în cazul containerelor de tip secvență).

- set
 - mulțime - stochează elemente distincte. Elementele sunt folosite și ca și cheie
 - nu putem avea două elemente care sunt egale
 - se folosește arbore binar de căutare ca și reprezentare internă
- Map, unordered_map
 - dicționar - stochează elemente formate din cheie și valoare
 - nu putem avea chei duplicate
- Bitset
 - container special pentru a stoca biți (elemente cu doar 2 valori posibile: 0 sau 1,true sau false, ...).

```
void sampleMap() {  
    map<int, Product*> m;  
    Product *p = new Product(1, "asdas", 2.3);  
    //add code <=> product  
    m.insert(pair<int, Product*>(p->getCode(), p));  
  
    Product *p2 = new Product(2, "b", 2.3);  
    //add code <=> product  
    m[p2->getCode()] = p2;  
  
    //lookup  
    cout << m.find(1)->second->getName()<<endl;  
    cout << m.find(2)->second->getName()<<endl;  
}  
  
#include <string>  
#include <vector>  
#include <unordered_map>  
#include <print>  
using std::string;  
using std::vector;  
using std::unordered_map;  
using std::print;  
void countProductPerType(const vector<Product>& prods) {  
    unordered_map<string, int> type2Count;  
    for (auto& p : prods) {  
        //type2Count.find(key) == type2Count.end()  
        if (type2Count.contains(p.getType())) {  
            type2Count[p.getType()]++;  
        } else {  
            type2Count[p.getType()] = 1;  
        }  
    }  
    //Iterate and print key-value pairs of unordered_map  
    for (auto pair : type2Count) {  
        print("{}:{}," ,pair.first,pair.second);  
    }  
}
```

Iteratori in STL

Iterator: obiect care gestionează o poziție (currentă) din containerul asociat. Oferă suport pentru traversare (++,-), derefențiere (*it).

Iteratorul este un concept fundamental în STL, este elementul central pentru algoritmi oferiti de STL.

Fiecare container STL include funcții membre begin() și end(), perechea de iteratori descrie o secvență [first, last) - interval deschis: first inclusiv, last exclusiv

end() - arată după ultimul element, nu este corect să incercăm să luăm valoarea (*)

```
void sampleIterator() {
    vector<int> v;
    v.push_back(4);
    v.push_back(8);
    v.push_back(12);
    //Obtain an the start of the iteration
    vector<int>::iterator it = v.begin();
    while (it != v.end()) {
        //dereference
        cout << (*it) << " ";
        //go to the next element
        it++;
    }
    cout << endl;
}
```

Permite decuplarea între algoritmi și containere

Existe mai multe tipuri de iteratori:

- iterator input/output (istream_iterator, ostream_iterator)
- forward iterators, iterator bidirectional, iterator random access
- reverse iterators

In funcție de tipul iteratorului putem avea diferite operații suportate: ++,!-,*(forward) – (bidirectional) it+3, it-6 (random access)

Iterator adaptors

```
vector<int> v2(6); //trebuie sa pregatim loc pentru elemente
copy(v.begin(), v.end(), v2.begin());
vector<int> v3;
//back_inserter este un adaptor - face push_back la *it=elem
copy(v.begin(), v.end(), back_inserter(v3));
vector<int> v3;
//!!! gresit, functia copy nu face loc pentru elemente in vectorul destinatie
copy(v.begin(), v.end(), v3.begin()); //segmentation fault
```

Input iterator adapter

```
int main() {
    using namespace std;
    //create a istream iterator using the standard input
    istream_iterator<int> start(cin);
    istream_iterator<int> end;
    vector<int> v4;
    //copy readed ints into v4 (until EOF(ctrl+z) or cin fail)
    copy(start, end, back_inserter(v4));
    for (int e : v4) {
        cout << e << ",";
    }
}
```

Implementare iterator VectorDinamic

```
class IteratorVector {
private:
    const VectorDinamic& v;
    int poz = 0;
public:
    IteratorVector(const VectorDinamic& v) :v{v} {}
    bool valid() const {
        return poz < v.size();
    }
    Element& element() const {
        return v.elems[poz];
    }
    void next() {
        poz++;
    }
};
```

Putem suprascrie operatori: *, ++, ==, != pentru a crea iteratori similar cu cei din STL
(ForwardIterator)

Daca dorim sa folosim vectorul intr-un **foreach** avem nevoie de metodele begin() si end()

<pre>IteratorVector VectorDinamic::begin() const { return IteratorVector(*this); } IteratorVector VectorDinamic::end() const { return IteratorVector(*this, lg);</pre>	<pre>Element& operator*() { return element(); } IteratorVector& operator++() { next(); return *this;</pre>
---	--

Putem folosi:

<pre>//testam iteratorul auto it = v.begin(); while (it != v.end()) { auto p = *it; assert(p.getPrice() > 0); ++it; }</pre>	<pre>for (auto& p : v) { std::cout << p.getType() << std::endl; assert(p.getPrice() > 0); }</pre>
--	--

STL Algorithms

Există o mulțime de algoritmi implementați în STL. Ele se află în modulul `<algorithm>` și în namespace-ul std.

Selected Standard Algorithms	
<code>p=find(b,e,x)</code>	<code>p</code> is the first <code>p</code> in <code>[b:e)</code> so that <code>*p==x</code>
<code>p=find_if(b,e,f)</code>	<code>p</code> is the first <code>p</code> in <code>[b:e)</code> so that <code>f(*p)==true</code>
<code>n=count(b,e,x)</code>	<code>n</code> is the number of elements <code>*q</code> in <code>[b:e)</code> so that <code>*q==x</code>
<code>n=count_if(b,e,f)</code>	<code>n</code> is the number of elements <code>*q</code> in <code>[b:e)</code> so that <code>f(*q,x)</code>
<code>replace(b,e,v,v2)</code>	Replace elements <code>*q</code> in <code>[b:e)</code> so that <code>*q==v</code> by <code>v2</code>
<code>replace_if(b,e,f,v2)</code>	Replace elements <code>*q</code> in <code>[b:e)</code> so that <code>f(*q)</code> by <code>v2</code>
<code>p=copy(b,e,out)</code>	Copy <code>[b:e)</code> to <code>[out:p)</code>
<code>p=copy_if(b,e,out,f)</code>	Copy elements <code>*q</code> from <code>[b:e)</code> so that <code>f(*q)</code> to <code>[out:p)</code>
<code>p=move(b,e,out)</code>	Move <code>[b:e)</code> to <code>[out:p)</code>
<code>p=unique_copy(b,e,out)</code>	Copy <code>[b:e)</code> to <code>[out:p)</code> ; don't copy adjacent duplicates
<code>sort(b,e)</code>	Sort elements of <code>[b:e)</code> using <code><</code> as the sorting criterion
<code>sort(b,e,f)</code>	Sort elements of <code>[b:e)</code> using <code>f</code> as the sorting criterion
<code>(p1,p2)=equal_range(b,e,v)</code>	<code>[p1:p2)</code> is the subsequence of the sorted sequence <code>[b:e)</code> with the value <code>v</code> ; basically a binary search for <code>v</code>
<code>p=merge(b,e,b2,e2,out)</code>	Merge two sorted sequences <code>[b:e)</code> and <code>[b2:e2)</code> into <code>[out:p)</code>

A tour of C++, Bjarne Stroustrup

Este doar un subset, o listă mai completă de algoritmi disponibili găsiți aici:
<http://en.cppreference.com/w/cpp/algorithm>

În general sunt funcții care primesc o pereche de iteratori (`begin()`, `end()`).
Perechea de iteratori descrie o secvență de elemente (un range) – $[a,b)$

Astfel acești algoritmi pot fi folosiți cu orice container STL, cu array-uri (`int a[]`), cu pointeri.

```
#include <vector>
#include <algorithm>

int main(){
    std::vector<int> v{ 3,2,8,1,4,5,7,6 };
    std::sort(v.begin(),v.end());
    for (auto a : v) {
        std::cout << a << " ";
    }
    std::cout << std::endl;
}

#include <algorithm>

int main(){
    int v[]{ 3,2,8,1,4,5,7,6 };
    std::sort(v,v+8);
    for (auto a : v) {
        std::cout << a << " ";
    }
    std::cout << std::endl;
}
```

Predicat / Functor

Majoritatea algoritmilor in STL au ca parametru un predicat.

Predicat poate fi:

- o funcție care returnează bool

```
bool simpleFct(int a) {
    return a % 2 == 0;
}
.....
int nrPare = count_if(v.begin(), v.end(), simpleFct);
```

- **function object/functor:** orice obiect care supraîncarcă operatorul "()" și returnează bool

```
class FunctionObj {
public:
    bool operator()(int a){return a % 2 == 0;}
};
.....
int nrPare = count_if(v.begin(), v.end(), FunctionObj{});
```

- **funcție lambda**

```
int nrPare = count_if(v.begin(), v.end(), [] (int a) {return a % 2 == 0;});
```

Exista functori gata definiti in STL (#include <functional>)

```
#include <functional>
....
vector<int> v{ 1,2,3,4,5,6 };
sort(v.begin(), v.end(), less<int>());
....
sort(v.begin(), v.end(), greater<int>());
```

Funcții lambda

Funcții anume (fără nume), se pot defini direct în locul în care e nevoie de o funcție

Foarte utili în cazul algoritmilor STL (și nu numai)

Practic este o metodă ușoară de a crea functori (e doar o sintaxă simplificată, compilatorul generează o clasa care suprascrie operator())

Sintaxa:

[capture-list](params){body}

capture-list – care sunt variabilele din scopul curent care se vad în interiorul funcției lambda

poate fi vid [] - nu captează nimic – nu se vede nici o variabilă

[=] - se vad toate variabilele din afara în corpul funcției lambda, se transmit prin valoare

[&] - se vad toate variabilele din afara în corpul funcției lambda, se transmit prin referință

[a,&b] – se vede a (prin valoare) și b (prin referință)

params – parametrii funcției lambda – exact ca și în cazul funcțiilor obișnuite

body – corpul funcției lambda

```
sort(v.begin(), v.end(), [](const Pet& p1, const Pet& p2) {
    return strcmp(p1.getType(), p2.getType());
});
```

Functii care primesc ca parametru alte functii (higher order functions, callback)

Parametru formal poate fi pointer la functie:

```
vector<Pet> PetStore::generalSort(bool(*maiMicF)(const Pet&, const Pet&)) {
    vector<Pet> v{ rep.getAll() };//fac o copie
    for (size_t i = 0; i < v.size(); i++) {
        for (size_t j = i + 1; j < v.size(); j++) {
            if (!maiMicF(v[i], v[j])) {
                //interschimbam
                Pet aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
        }
    }
    return v;
}
```

In acest caz putem apela cu o functie cu același semnatura sau un lambda care nu captează nimic

```
bool cmpSpecies(const Pet& p1, const Pet& p2) {
    return p1.getSpecies() < p2.getSpecies();
}

...
vector<Pet> PetStore::sortBySpecies() {
    return generalSort(cmpSpecies);
}

vector<Pet> PetStore::sortBySpecies() {
    return generalSort([](const Pet&p1, const Pet&p2) {
        return p1.getSpecies() < p2.getSpecies();
    });
}
```

Daca vrem sa permitem apelul cu functii lambda care capteaza ceva folosim clasa `function`:

```
#include <functional>
vector<Pet> PetStore::filtreaza(function<bool(const Pet&)> fct) {
    vector<Pet> rez;
    for (const auto& pet : rep.getAll()) {
        if (fct(pet)) {
            rez.push_back(pet);
        }
    }
    return rez;
}

...
vector<Pet> PetStore::filtrarePret(int pretMin, int pretMax) {
    return filtreaza([](const Pet& p) {
        return p.getPrice() >= pretMin && p.getPrice() <= pretMax;
    });
}
```

Tratarea exceptiilor

Situări anormale apar în timpul execuției (nu există fișier, nu mai există spațiu pe disk, etc), trebuie să tratam aceste situații

In general situații în care o funcție/metoda nu poate efectua operația promisa

Problema: Cum raportam eroarea, cum propagam eroarea (in cazul in care avem un lanț de apeluri, cum separam partea in care apare eroarea de partea unde tratam eroarea tratarea

Logica aplicației (in general aici putem trata eroarea – mesaj pe

Layer consola/fereastra, raspuns HTTP, etc)

Layer

Layer

Layer

Low level implementation - in general aici apar erori (nu pot scrie in fisier, nu mai am memorie, etc), in general in aceasta parte a aplicatiei nu vrem/putem rezolva situatia

Obs: este valabil in general -nu doar in cazul UI-GraspController-Repository.
Codul este in general organizat pe nivele logice, actiunea efectuata este
rezultatul unui lant de apeluri de metode

Soluții pentru raportarea de erori

In limbaje fără mecanismul de excepții soluțiile sunt (ex in C):

- returnare cod de eroare
- setarea de flag-uri (variabile globale)

Probleme cu aceste abordări:

- implicit se ignora eroarea (daca nu verific valoare de return sau flagurile)
- se compune greu (diferite coduri, daca pe stack-ul de apel cineva ignoră eroarea nu mai e propagat)
- fluxul normal este amestecat cu tratarea situațiilor excepționale

Alternative:

1 Putem folosi un tip de date special care modelează ideea: returnam optional ceva. In C++ (începând cu standardul c++ 17) există tipul de date **std::optional<T>**, în headerul: #include <optional>

Clase similare există și în alte limbaje (Optional, Maybe, etc)

<https://en.cppreference.com/w/cpp/utility/optional>

Ex. `optional<Pet>` poate conține un obiect Pet sau poate e gol. Putem folosi de exemplu la funcția de căutare după id (`optional<Pet> find(int id);`) funcția să returneze un optional gol dacă nu s-a gasit Pet.

2 (Doar în c++ 23) Putem folosi un tip de date special care modelează ideea: returnăm o valoare dacă operația a reușit sau returnăm un cod de eroare dacă a apărut o problemă

std::expected<ValueType,ErrorType>, în headerul #include <expected>

<https://en.cppreference.com/w/cpp/utility/expected>

Putem folosi excepții (raportăm eroarea aruncând o excepție)

Tratarea excepțiilor în c++

excepții - situații anormale ce apar în timpul execuției

tratarea excepțiilor - mod organizat de a gestiona situațiile exceptionale ce apar în timpul execuției

O excepție este un eveniment ce se produce în timpul execuției unui program și care provoacă întreruperea cursului normal al execuției.

Elemente:

- **try block** marchează blocul de instrucțiuni care poate arunca excepții.
- **catch block** bloc de instrucțiuni care se executa în cazul în care apare o excepție (tratează excepția).
- Instrucțiunea **throw** mecanism prin care putem arunca (genere excepții) pentru a semnala codului client apariția unei probleme.

```
void testTryCatch() {  
    ...  
    try {  
        //code that may throw an exception  
        ErrorClass errObj;  
        throw errObj;  
        //code  
    } catch (ErrorClass& e){//e- ca si un param. de functie  
        //error handling - daca eroarea era de tip ErrorClass  
        // sau orice alt tip derivat din ErrorClass  
        cout << "Error occurred.";  
    } catch (...) {  
        //error handling - intra aici la orice eroare  
    }  
}
```

Tratarea exceptiilor

- Codul care susceptibil de a arunca exceptie se pune intr-un bloc de try.
- Adaugam unu sau mai multe sectiuni de **catch**. Blocul de instructiuni din interiorul blocului catch este responsabil sa trateze exceptia aparuta.
- Daca codul din interiorul blocului try (sau orice cod apelat de acesta) arunca exceptie, se transfera execuția la clauza catch corespunzătoare tipului exceptiei aparute. (exception handler)

```
void testTryCatchFlow(bool throwEx) {  
    // some code  
    try {  
        cout << "code before the exception" << endl;  
        if (throwEx) {  
            cout << "throw (raise) exception" << endl;  
            throw MyError();  
        }  
        cout << "code after the exception" << endl;  
    } catch (MyError& error) {  
        cout << "Error handling code " << endl;  
    }  
}  
  
testTryCatchFlow(0);  
testTryCatchFlow(1);
```

- Clauza catch nu trebuie neaparat sa fie in același metodă unde se arunca exceptia. Exceptia se propagă.
- Când se arunca o exceptie, se caută cel mai apropiat bloc de **catch** care poate trata exceptia ("unwinding the stack").
- Dacă nu avem clauză **catch** în funcția în care a apărut exceptia, se caută clauza **catch** în funcția care a apelat funcția.
- Căutarea continuă pe stack până se găsește o clauză **catch** potrivită. Dacă exceptia nu se tratează (nu există o clauză **catch** potrivită) programul se oprește semnalând eroarea apărută.
- Potrivirea cu clauza catch:
 - foarte asemănător cu potrivirea intre parametru actual și parametru formal

Excepții - obiecte

- Când se aruncă o excepție se poate folosi orice tip de date.

Tipuri predefinite (int, char, etc) sau tipuri definite de utilizator (obiecte). Nu este recomandat să folosim pointeri (chiar dacă este posibil)

- Este recomandat să se creeze clase pentru diferite tipuri de excepții care apar în aplicație
- Se arunca un obiect (**nu referință sau pointer**) și se prinde prin referință (pentru a evita copierea)
- Obiectul excepție este folosit pentru a transmite informații despre eroarea apărută

```
class POSError {
public:
    POSError(string message) :
        message(message) {
    }
    const string& getMessage() const {
        return message;
    }
private:
    string message;
};
```

```
class ValidationError: public POSError {
public:
    ValidationError(string message) :
        POSError(message) {
    }
};
```

```
void Sale::addSaleItem(double quantity, const Product& product) {
    if (quantity < 0) {
        throw ValidationError("Quantity must be positive");
    }
    saleItems.push_back(SaleItem{quantity, product});
}

try {
    pos.enterSaleItem(quantity, product);
    cout << "Sale total: " << pos->getSaleTotal() << endl;
} catch (ValidationError& err) {
    cout << err.getMessage() << endl;
}
```