

Curs 13

- Modern C++ (C++ 11/14)
- Resource management – RAI
- Thread
- ASM - optimizari

Curs 12 - Organizarea interfețelor utilizator

- **Separate Presentation**
- **Observer – push/pull**
- **diagrame UML de secvență**
- **Șabloane de proiectare: adapter, strategy; composite**

New features in C++ 11/14

Evoluția C++

- versiune curenta C++17 (înainte: C++14, C++11, C++98, C++ with classes, C)
- finalizare specificații C++ 20
- isocpp.com - website C++ ISO standard

C++11/14/17 design goals:

Make simple things simple, make hard things possible

More abstractions without sacrificing efficiency

C++ is “expert friendly” But is not just a language for "experts"

Elementele noi ar trebui sa faciliteze un stil mai bun de programare.

Însă orice facilitare se poate folosi greșit (Any feature can be overused, there are many "dark corners" in the language)

Majoritatea compilatoarelor implementează funcționalitățile noi din limbaj, din standardul C++ curent și oferă standard library (STL) conform specificațiilor din standard

- gcc (mingw)
- clang
- microsoft compiler

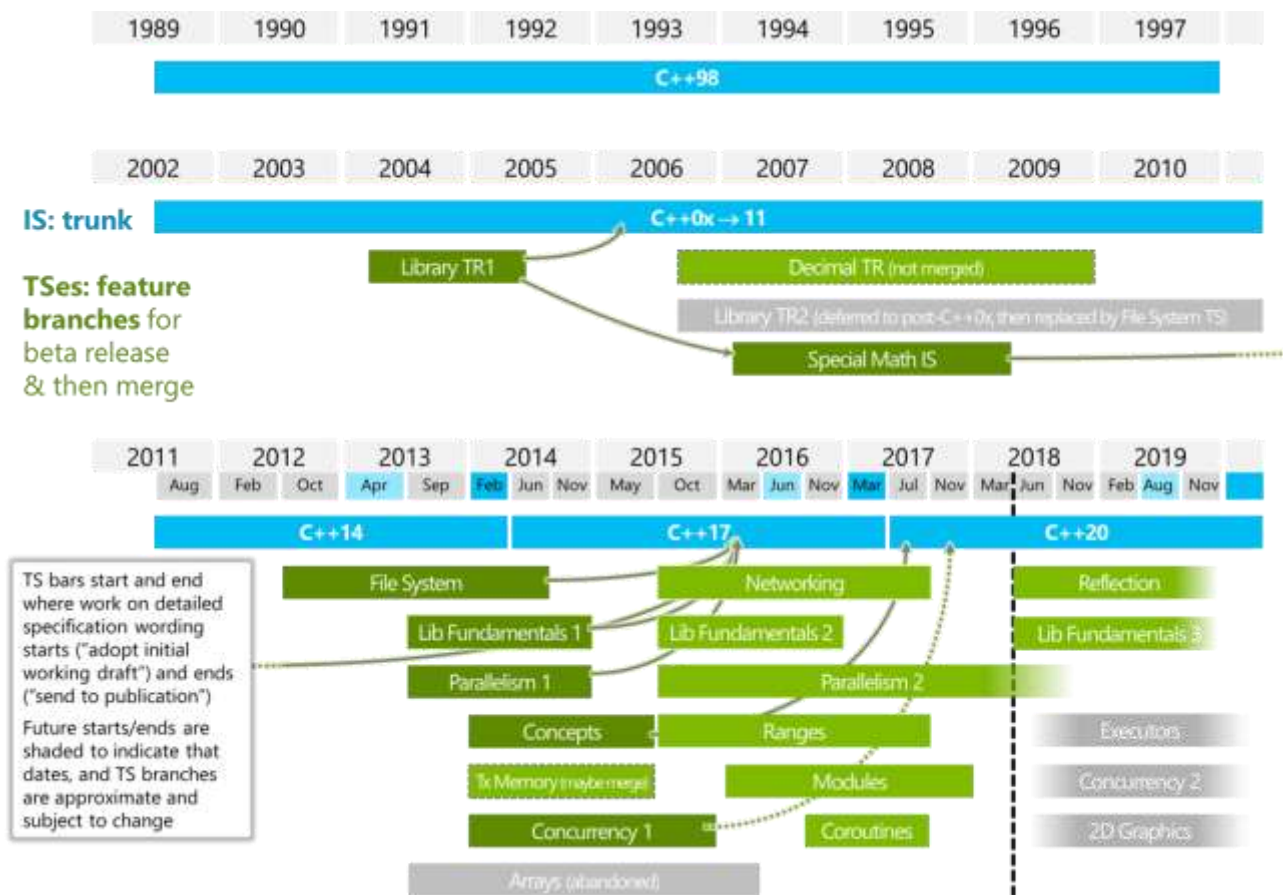
Bjarne Stroustrup:

- www.stroustrup.com/
- Talks: C++11 Style, The essence of c++, etc

Herb Sutter

- Talks: One C++, Back to basics, Why C++ , etc

Evolutie C++



Noutăți in c++ 17

Noutăți in biblioteca standard:

`std::variant`, `std::optional`, `std::any`

`std::string_view`, `std::filesystem`, `std::byte`

variante cu execuție paralele pentru algoritmi (`copy`, `find`, `sort`)

Facilități noi in limbaj:

`structured bindings`, `constexpr if`, `folding expressions`

Noutăți in c++ 20

Noutăți in biblioteca standard:

`concepts library`, `std::span`

Facilități noi in limbaj:

`module` – creare de cod izolat in module, compilare separata,
separare interfață/implementare

`ranges` – descrie secvențe de elemente

`coroutine` – funcții care permit suspendarea/repornirea

`concepts` - pentru template metaprograming, predicate verificate
la compilare

Evolutia codului C++ (exemplu)

C (old C++ code)

```
int cmpInt(const void *a, const void *b) {
    int aa = *(int *) a;
    int bb = *(int *) b;
    return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}

void sortC() {
    const int size = 3;
    int array[size];
    array[0]=2; array[1]=1; array[2]=0;
    qsort(array, size, sizeof(int), cmpInt);
    int i;
    for (i=0;i<size;i++){
        printf("%d ",array[i]);
    }
}
```

C++98

```
bool cmpF(int a, int b) {
    return b < a;
}

void sortC98() {
    vector<int> v;
    v.push_back(3); v.push_back(2); v.push_back(1);
    sort(v.begin(), v.end(), cmpF);
    for (vector<int>::iterator it = v.begin(); it!=v.end(); ++it)
    {
        cout << (*it) << endl;
    }
}
```

C++14

```
void sortC14() {
    vector<int> v { 3, 2, 1 };
    sort(v.begin(), v.end(), [](int a, int b) {
        return b<a;
    });
    for (auto elem : v) {
        cout << elem << endl;
    }
}
```

C++11: Mai rapid, mai general, type safe – make simple things simple

Uniform initialization

- Sintaxa comuna pentru inițializare de obiecte in C++11.
- Permite inițializarea ușoară pentru `std::vector` or `std::map` (sau orice container) la un set de valori.

<pre>vector<int> v { 3, 2, 1 } struct SampleStruct { int a, b; }; SampleStruct s { 1, 2 }; SampleStruct sarray[] = { { 1, 2 }, { 2, 5 }, { 3, 4 } }; class MyClass { public: MyClass(int a); MyClass obj{2}; MyClass array[] = {{1},{7}};</pre>	<pre>v.push_back(3); v.push_back(2); v.push_back(1);</pre>
--	--

- Implicit type narrowing

<pre>class MyClass { public: MyClass(int a); MyClass obj { 2 }; //fine MyClass obj3 { 2.3 }; //error: narrowing conversion of '2.3'</pre>	<pre>class MyClass { public: MyClass(int a); ... MyClass obj2(2.3); //works silently convert to 2 (maybe a compile warning)</pre>
--	---

- resolve "most vexing type"

<pre>class MyClass { public: MyClass(); MyClass obj{}; obj.f(); //fine</pre>	<pre>class MyClass { public: MyClass(); MyClass obj(); //is this a variable? obj.f(); //compile error... is of non-class type 'MyClass()'</pre>
---	--

- Minimize redundant typenamees

<pre>MyClass f() { return {3}; }</pre>	<pre>MyClass typeNameRedundancy() { return MyClass(3); }</pre>
--	--

Uniform initialization mechanics - initializer lists in clasele proprii

Exista un nou tip de date: `std::initializer_list`, acesta poate fi folosit ca si orice colecție

Pentru a suporta initializer list in clasele proprii:

- se definește un constructor care primește ca parametru `std::initializer_list`
- apoi se poate inițializa folosind același sintaxa ca si la vector de ex.

```
class MyClassWithUI{
public:
    MyClassWithUI(initializer_list<int> l){
        for_each(l.begin(),l.end(),[&](int a){
            elems.push_back(a);
        });
    }
    vector<int> getElems(){
        return elems;
    }
private:
    vector<int> elems;
};
```

Obs. Constructorul cu `initializer_list` are prioritate (fata de alți constructori definiți in clasa)

```
vector<int> v2 { 10 };
cout<<v2.size()<<endl; //print 1, 1 element (10) in the vector

vector<int> v3(10);
cout<<v3.size()<<endl; //print 10, elems: 0,0,0,0,0,0,0,0,0,0
```

Auto

Tipul variabilei se deduce automat de compilator pe baza expresiei care inițializează variabila .

Poate fi folosit si pentru tipul de return al unei funcții (se deduce din expresia de la return) (since C++14).

```
MyClass someFunction() {
    return MyClass(3);
}
auto someFct()->int{
    return 7;
}
void sampleAuto() {
    int a = 6;
    auto b = 6; //b is int
    auto c = someFunction(); //c is MyClass, the return type
    auto d = someFct(); //d is int
}
```

Permite scriere de cod generic (in general la templaturi dar nu numai)

```
void doThings(vector<int> v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        cout << (*it);
    }
}

void doThingsA(vector<int> v) {
    for (auto it = v.begin(); it != v.end(); ++it) {
        cout << (*it);
    }
}
```

Schimbam metoda doThings – transmitem prin referința sa evitam copierea, punem const fiindcă nu schimbam v

```
void doThings(const vector<int>& v) {
    for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it) {
        cout << (*it);
    }
}

void doThingsA(vector<int> v) {
    for (auto it = v.begin(); it != v.end(); ++it) {
        cout << (*it);
    }
}
```


Funcții Lambda

Funcții anonime care sunt capabile sa captureze variabile din domeniul exterior de vizibilitate

```
sort(v.begin(), v.end(), [](int a, int b) {  
    return b<a;  
});  
  
for_each(v.begin(), v.end(), [](int a) {cout<<a;});
```

Compilatorul creează functorul in locul nostru. Practic creează o clasa care definește operatorul ()

Variabile vizibile in interiorul unei lambda:

```
vector<Person> all{{3, "Ion"}, {5, "Pop"}};  
int someId=5;  
auto p =find_if(all.begin(),all.end(), [&](const Person& p){  
    return p.id==someId;  
});  
if (p!=all.end()){  
    return *p  
}
```

Sintaxa general

`[capture-list](params)->ret{body}`

capture-list – specify variables that are visible inside the *body*

capture-list examples:

- `[]` captures nothing
- `[&]` captures all by reference
- `[=]` captures all by value
- `[a,&b]` capture a by value, b by reference

ret – return type of the lambda

if omitted: if the body is just a return statement - ret is the type of the returned expression else is void

Resource management – RAI

Orice resursa ar trebui sa fie proprietatea unui "handler":

- încapsularea resursei (memorie, fișier) este o abstractizare folositoare (vector, string, iostream, file, thread, etc)
- Handler – este o clasa cu responsabilitatea de a crea/gestiona/elibera resursa
- In constructor se obține/inițializează resursa iar in destructor se eliberează resursa

```
class Vector{
public:
    Vector(initializer_list<int> l); //acquire memory
    ~Vector(); //release memory
};

int fct() {
    vector<int> v{1,2,3,4};
    ...
//end of scope, destructor automatically invoked (memory released)
}
```

Orice resursa (memory, file handler, thread, ...) ar trebui sa aibă un (scoped) handler

Orice responsabilitate ar trebui încapsulată într-o clasa handler (ex. Desenare obiect geometric, ștergerea de pe ecran)

Handler – trebuie alocat pe stack

- scoped – se dealoca automat când execuția părăsește domeniul de vizibilitate a variabilei

RAII – Resource acquisition is initialization

- achiziție in timpul construcției (constructor)
- eliberare la distrugere (destructor)

Folosind idea de handlers:

- este mult mai ușor sa gestionam memoria (fara resource leak)
- ascundem complexitatea (anumite resurse pot fi ne-trivial de alocat/eliberat)
- mult mai ușor sa scriem exception safe code (chiar daca metoda arunca excepție destructorul variabilelor locale se apelează)

De ce folosim pointeri – resursa (memorie) care nu e gestionata de un obiect handler

Guideline: Now raw pointers

```
void pointerNotNeeded(){
    MyClass* c = new MyClass(7);
    //do some computation...
    if (c->getA()<0){
        throw runtime_error("Negative not allowed");//!!!leak
    }
    //do some computation...
    if (c->getA()>10){
        return;//!!!!leak
    }
    delete c; //only if we are lucky
}

void noPointer(){
    MyClass c{7};
    //do computation
    if (c.getA()<0){
        throw runtime_error("Negative not allowed");
    }
    //computation
    if (c.getA()>10){
        return;
    }
}
```

Exista motive legitime de a folosi pointeri (pointerul nu e un lucru rău):

- este o abstractizare buna a memoriei din calculator
- poate fi folosit in interiorul claselor de tip handler:
 - vector are un pointeri la array-ul de elemente
 - pentru a implementa structura de tree sau lista înlănțuita
 - poate fi folosit pentru a reprezenta o poziție
- putem folosi pentru a facilita polimorfismul

Guideline updatat: **No owning raw pointers**

- nu folosiți pointer pentru obiecte pe care trebuie sa le distrugeți (reprezintă idea de ownership). Pointer este owning daca este responsabil cu memoria referita (trebuie sa eliberezi memoria folosind pointerul)

Owning pointers: Smart pointers

In cazul in care totuși avem nevoie de owning pointer:

- Ex. folosim o funcție care returnează un owning pointer (owning pointer => trebuie sa eliberam memoria după ce am folosit obiectul)

Ar trebui sa existe o clasa handler pentru el.

Mai bine folosim una dintre clasele handler existente din biblioteca standard (**unique_ptr**, **shared_ptr**, **weak_ptr** – header `<memory>`)

Smart pointers se comporta ca si un pointer normal (oferă operațiile uzuale de pointer `*`, `++`, etc) dar are in plus o funcționalitate: gestiune automata a memoriei, range checking, etc.

```
void sampleNastyAPI() {
    MyClass* rez = someAPI();
    //do computation
    if (rez->getA() < 0) {
        throw runtime_error("Negative not allowed");
    }
    //computation
    if (rez->getA() == 0) {
        return;
    }
    delete rez;
}

void hadleWithSmartPointers() {
    unique_ptr<MyClass> rez(someAPI());
    //do computation
    if (rez->getA() < 0) {
        throw runtime_error("Negative not allowed");
    }
    //computation
    if (rez->getA() == 0) {
        return;
    }
}
```

std::unique_ptr este un smart pointer care preia responsabilitatea de delocare. Când **unique_ptr** este distrus acesta distruge si obiectul referit (apelează destructor).

Același pointer poate fi înglobat doar de o singura instanța de **unique_ptr** (non copyable).

Smart pointers

```
void smartPointerSample() {
    MyClass* rez = someAPI(); //we leak rez
    unique_ptr<MyClass> smartP{someAPI()};

    vector<MyClass*> v;
    //when v is destroyed we leak 3 MyClass objects
    v.push_back(new MyClass());
    v.push_back(new MyClass());
    v.push_back(new MyClass());

    vector<unique_ptr<MyClass>> smartV;
    smartV.push_back(unique_ptr<MyClass>(new MyClass()));
    smartV.push_back(unique_ptr<MyClass>(new MyClass()));
    smartV.push_back(unique_ptr<MyClass>(new MyClass()));
}
```

shared_ptr : similar cu `unique_ptr` (încapsulează un pointer și gestionează dealocarea), dar implementează reference counting:

- pot exista multiple instanțe de **shared_ptr** care referă același pointer, **shared_ptr** ține numărul de instanțe **shared_ptr** existente pentru același pointer (reference count)
- când numărul de referințe ajunge la 0 obiectul referit de pointer este dealocat

Move semantic &&

Why do we have to deal with (owning) pointers

- We are working with legacy code
- We are using a library (some old library)
- Somebody in the project don't know the guideline: No owning raw pointers :)
- Return a pointer from a function to avoid the unnecessary copy of large objects (starting from c++11 there is a better alternative)

Returning large object from a function

class LargeMatrix. We want to implement sum of 2 matrices

Option1 – Problem: who does the delete; No raw pointer

```
LargeMatrix* sum(const LargeMatrix& a, const LargeMatrix& b){  
    LargeMatrix* rez = new LargeMatrix();  
    //rez=a+b  
    return rez;  
}  
...  
LargeMatrix* rez = sum(a,b);
```

Option2 – Problem: who does the delete; Caller not even know that he may need to delete

```
LargeMatrix& sum2(const LargeMatrix& a, const LargeMatrix& b){  
    LargeMatrix* rez = new LargeMatrix();  
    //rez=a+b  
    return *rez;  
}  
...  
LargeMatrix& rez = sum(a,b);
```

Option3 – Problem: ugly interface. Function with side effect Consider operator +;

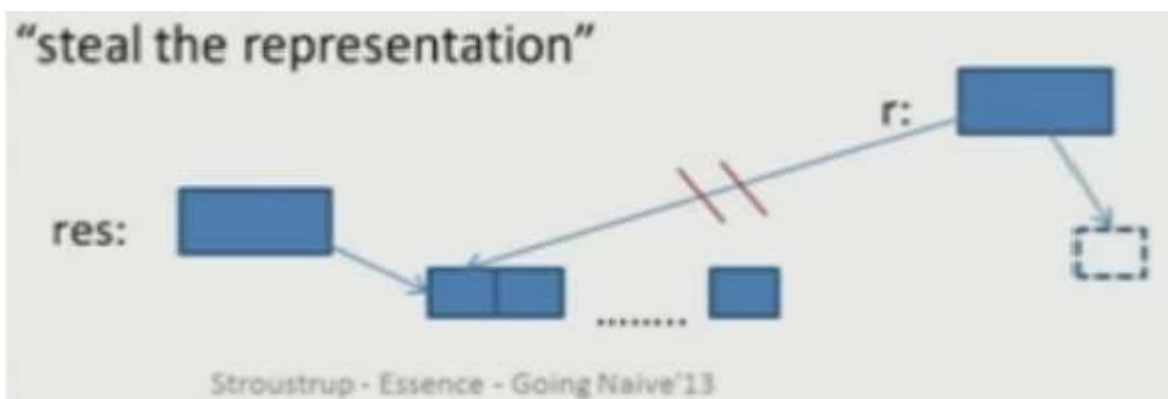
```
void sum2(const LargeMatrix& a, const LargeMatrix& b,){  
    LargeMatrix* rez = new LargeMatrix();  
    //rez=a+b  
    return *rez;  
}  
...  
LargeMatrix rez;  
sum(a,b,rez);
```

Returning large object from a function

Best option: return an handler.

```
LargeMatrix sum3(const LargeMatrix& a, const LargeMatrix& b){  
    LargeMatrix rez;  
    //rez=a+b  
    return rez;  
}  
...  
LargeMatrix res = sum3(a,b);
```

Copy may be expensive (compiler invoke copy constructor)



In

C++11 a new kind of constructor: move constructor used for move operation
don't copy, steal the representation and leave an empty object behind

```
class LargeMatrix {  
public:  
    LargeMatrix(LargeMatrix&& ot) { //move constructor  
        elems = ot.elems;  
        ot.elems = {};  
    }  
private:  
    int *elems; //large array of numbers  
};
```

C++11 style guidelines

Style

- No naked pointers
 - Keep them inside functions and classes
 - Keep arrays out of interfaces (prefer containers)
 - Pointers are implementation-level artifacts
 - A pointer in a function should not represent ownership
 - Always consider `std::unique_ptr` and sometimes `std::shared_ptr`
- No naked **new** or **delete**
 - They belong in implementations and as arguments to resource handles
- Return objects “by-value” (using move rather than copy)
 - Don’t fiddle with pointer, references, or reference arguments for return values



Stroustrup - C++11 Style - Feb'12

RAII and move semantics

All the standard library containers provide move semantics:

- vector, list, map, set, unordered_map

You can return a local vector object by value from a function

- no copy just a move operation (2-3 assignments even if the number of elements is very large)

Other standard resource handlers are providing move semantics:

- threads, locks
- istream, ostream
- unique_ptr, shared_ptr

Time

Modulul `<chrono>` din standard library oferă funcționalități de măsurare a timpului (în namespaceul `using namespace std::chrono`).

Tipuri de date de interes în namespaceul `chrono`:

`std::chrono::system_clock` - the system-wide real time wall clock.

`std::chrono::high_resolution_clock` - the clock with the smallest tick period provided by the implementation

time_point - point in time, **duration** – period of time

folosind metoda **duration_cast** putem calcula timpul scurs între două `time_point` uri.

```
#include <chrono>
using namespace std::chrono;

void measureElapsedTime() {
    auto t0 = high_resolution_clock::now();
    fct_we_want_to_measure();
    auto t1 = high_resolution_clock::now();
    auto elapsed = duration_cast<seconds>(t1 - t0);
    cout << elapsed.count() << "sec\n";
}

void fct_we_want_to_measure() {
    ...
}
```

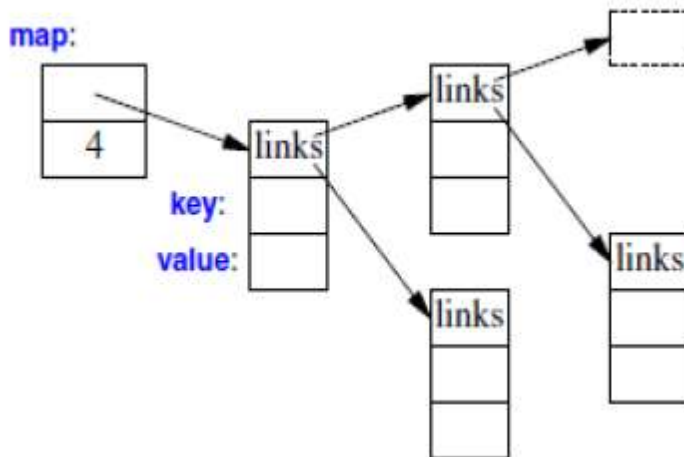
Când vorbim despre performanța, este foarte important să măsurăm. Doar măsurarea poate fi punctul de pornire pentru optimizarea codului.

Exercițiu: îmbunătățire funcție `genereazaPet`, măsurători pentru diferite variante de implementare

Containere STL – dictionare

std::map

- in headerul <map>
- dictionar implementat cu o structura de arbore de căutare (red/black tree)
- este un container optimizat pentru lookup: $O(\log_2(n))$
- elementele sunt perechi (Pair p.first, p.second)



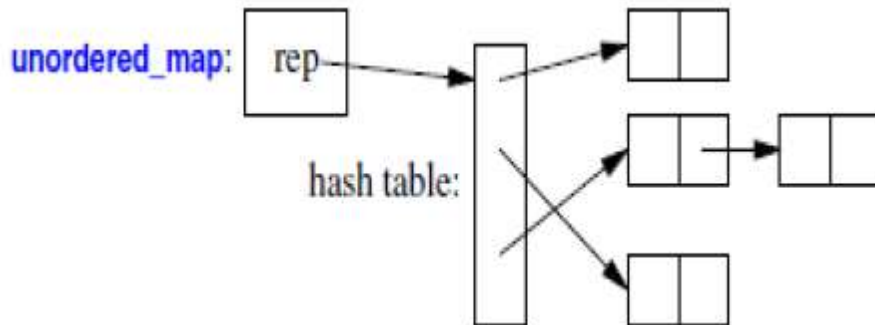
la constructor se poate da funcția de comparare a cheilor (folosit in construirea/parcurgerea structurii de tree)

```
void testMaps() {
    std::map<string, int>
    phoneBook{{"Ion",1234},{ "Vasile",32432} ,{ "Maria",765432 } };
    //add a new entry
    phoneBook["Iulia"] = 111145;
    //lookup key: iterator find( const Key& key );
    auto val = phoneBook.find("Ion");
    cout << (*val).first << (*val).second << "\n";
    //not found
    if (phoneBook.find("0") == phoneBook.end()) {
        cout << "Not found\n";
    }
    //stergere - prin iterator/prin element/prin range
    phoneBook.erase("Ion");
    if (phoneBook.find("Ion") == phoneBook.end()) {
        cout << "Ion erased\n";
    }
    phoneBook.erase(phoneBook.find("Iulia"));
    phoneBook.erase(phoneBook.find("Iulia"),phoneBook.end());
}
```

Unordered_map

std::unordered_map

- in headerul `<unordered_map>`
- implementat folosind o tabela de dispersie
- optimizat pentru lookup: $O(1)$ daca avem un hashing function ideal
- interfața (metodele publice) este identic cu `map`



In librăria standard exista definite funcții de hashing pentru tipurile predefinite

In cazul in care dorim sa avem ca si chei obiecte de ale noastre (user defined), trebuie sa cream o functie de hashing custom

```
struct MyClass {
    string name;
    int phoneNr;
    bool operator==(const MyClass& a) const{
        return name == a.name;
    }
};

struct MyHash {
    //acesta se va folosi pentru hashing
    size_t operator()(const MyClass& m) const {
        return std::hash<string>()(m.name) ^ std::hash<int>()(m.phoneNr);
    }
};

void testHashMaps() {
    std::unordered_map<MyClass, int, MyHash> phoneBook{{{ "Ion",1234},1234},
                                                         {{ "Vasile",32432 },32432},
                                                         {{ "Maria",765432 },765432}};

    //transformam intr-o lista
    std::vector<MyClass> l;
    for (auto pair : phoneBook) {
        l.push_back(pair.first);
    }

    for (auto o : l) {
        cout << o.name << "\n";
    }
}
```

Exercițiu: modificat repository de Pet sa folosească map in loc de vector

Fire de execuție – thread

Paralel - execuția simultană a mai multor taskuri (metode)

Concurent = paralel + acces simultan la același resurse

Este folosit pentru a mari volumul de procesare efectuat, calculatoarele/laptopurile curente au mai multe procesoare, sunt capabile să execute concurent mai multe taskuri.

Fiecare task se execută în același proces, dar pe un fir de execuție propriu

C++ oferă posibilitatea de a executa taskuri în paralel folosind clasa `std::thread` din modulul `<thread>`

```
#include <thread>

void func1(int n) {
    for (int i = 0; i < n; i++) {
        cout << i << "\n";
    }
}

void testThreads() {
    //executam func1 cu parametrul 1000 pe un fir de executie separat
    std::thread t1{func1,1000};

    //executam func2 cu parametrul 1000 pe un fir de executie separat
    std::thread t2{ func1,1000 };

    //asteptam pana se termina
    t1.join();
    t2.join();
}
```

Exercițiu: La generarePet să folosim un thread pentru a nu bloca interfața grafică în timp ce se execută generarea.

Destructorul de la clasa thread distruge threadul (apelează `terminate`), trebuie apelată metoda `detach()` de la thread dacă dorim ca firul de execuție să continue chiar și după ce se distruge obiectul thread.

```
QObject::connect(btnGenereaza10000, &QPushButton::clicked, [this]() {
    std::thread t{ [this]() {
        this->btnGenereaza10000->setEnabled(false);
        ctr.genereazaAleatorAnimale(1000);
        QMessageBox::information(nullptr, "Info", "Am terminat generarea...");
        this->btnGenereaza10000->setEnabled(true);
    } };
    t.detach();
});
```

Compiler explorer

Oferă posibilitatea de a vedea codul ASM generat de compilator

Se poate experimenta cu diferite variante compilatoare, opțiuni de compilare

<https://godbolt.org/>

The screenshot displays the Godbolt Compiler Explorer web application. The browser address bar shows <https://godbolt.org/#>. The interface includes a top navigation bar with tabs for 'Compiler Explorer', 'C++', 'Editor', 'Diff View', and 'More'. Below this, the 'C++ source #1' tab is active, showing the following C++ code:

```
1 int f(int a, int b, int c){
2     return a+b+c;
3 }
4
5
6
7 int main(){
8     int b = 9;
9     return f(2,5,b);
10 }
```

The 'x86-64 clang 4.0.0 (Editor #1, Compiler #1)' tab is also active, displaying the generated assembly code for the function `f` and `main`. The assembly is shown in Intel syntax. A tooltip is visible over the `ret` instruction at line 12, explaining its function: 'Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction. More information available in the context menu.'

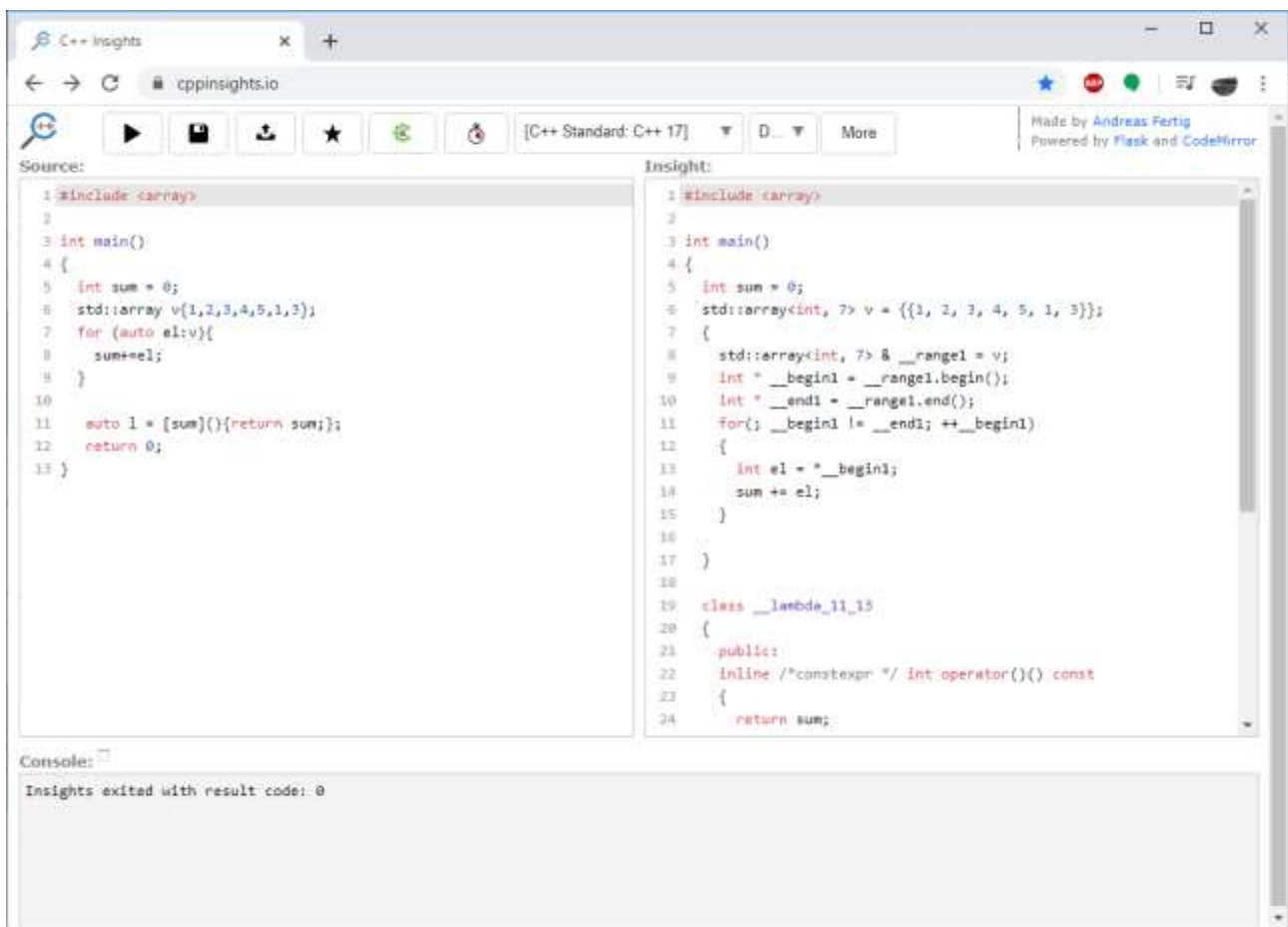
Below the assembly view, a status bar indicates 'clang version 4.0.0 (tags/RELEASE_400/final 299826)-cached'. At the bottom left, a message states 'Compiler exited with result code 0'.

CPP Insights

Poate fi folosit pentru a înțelege mai bine anumite conceptele din C++, :
conversii implicite, rangefor, funcții lambda, template, type deduction etc.

Afișează codul C++ după ce codul scris de programator trece printr-o transformare inițială de către compilator.

<https://cppinsights.io/>



The screenshot displays the C++ Insights web interface. The top navigation bar includes the site name, a search icon, and a dropdown menu for the C++ standard (set to C++17). The main content area is split into two panels: 'Source' on the left and 'Insight' on the right. The 'Source' panel contains the original C++ code, which uses a range-based for loop to iterate over an array and calculate a sum. The 'Insight' panel shows the code after transformation, where the range-based for loop is replaced by a more explicit implementation using iterators and a lambda function. The 'Console' panel at the bottom shows the output of the program, which is the sum of the array elements.

```
Source:
1 #include <array>
2
3 int main()
4 {
5     int sum = 0;
6     std::array v{1,2,3,4,5,1,3};
7     for (auto el:v){
8         sum+=el;
9     }
10
11     auto l = [sum](){return sum;};
12     return 0;
13 }
```

```
Insight:
1 #include <array>
2
3 int main()
4 {
5     int sum = 0;
6     std::array<int, 7> v = {{1, 2, 3, 4, 5, 1, 3}};
7     {
8         std::array<int, 7> & __rangel = v;
9         int * __begin1 = __rangel.begin();
10        int * __end1 = __rangel.end();
11        for(; __begin1 != __end1; ++__begin1)
12        {
13            int el = *__begin1;
14            sum += el;
15        }
16    }
17 }
18
19 class __lambda_11_15
20 {
21 public:
22     inline /*constexpr */ int operator()() const
23     {
24         return sum;
```

Console: Insights exited with result code: 0