Curs 5

Programare generica

Mecanismul de template permite :

- parametrizare dupa un tip, si permite scrierea de algoritmi generali
- se verifica la compilare in momentul instantiei templatului daca tipul primit ca parametru template are metodele dorite

Programarea generica se refera la crearea de algoritmi generali unde prin general se intelege ca algoritmul poate lucra cu orice tipuri de date.

Tipuri abstracte de date

ADT

- se separa interfata (ce vede cel care foloseste) de implementare (cum e implementat)
- specificatii abstracte (fara referire la detaliile de implementare)
- ascundere detalii de implementare

Clase

- header : contine declaratia de clasa + metode
- specificatii pentru fiecare metoda
- folosind modificatorul private, reprezentarea (campurile clasei), metodele care sunt folosite doar intern pot fi protejate de restul aplicatiei (nu sunt vizibile in afara clasei)

Atribute statice in clasa (campuri / metode)

Atributele statice, keyword : static dintr-o clasa apartin clasei, nu instantei (obiectelor)
Aceste atribute caracterizeaza clasa, nu face parte din starea obiectelor

Ne referim la ele folosind operatorul " :: "

Sunt asemanatoare cu variabilele globale doar ca sunt definite in interiorul clasei - retin o singura valoare chiar daca am multiple obiecte. Obs : Variabilele statice trebuie initializate in fisierul .cpp

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

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

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

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

Standard Template Library (STL)

- STL este o biblioteca de clase C++, care face parte din parte din C++ Standard Library
- Ofera structuri de date si algoritmi fundamentali
- STL ofera componente generice, parametrizabile. Aproape toate clasele din STL sunt parametrizate (Template)
- STL contine clase pentru :
  - containere, iteratori
  - algoritmi

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

Containeri

Un container este o grupare de date in care se pot adauga (insera) si din care se pot sterge (extrage) obiecte. Implementarile din STL folosesc sabloane ceea ce ofera o flexibilitate in ceea ce priveste tipurile de date ce sunt suportate.

Containerul gestioneaza si memoria necesara stocarii elementelor, ofera metode de acces la elemente (direct si prin iteratori)

Containerele ofera functionalitati (metode) :

- accesare elemente (ex : [])
- gestiune capacitate (ex : size())
- modificare elemente (ex : insert, clear)
- iterator (begin(), end())
- alte operatii

Decizia in alegerea containerului potrivit pentru o problema concreta se bazeaza pe :

- functionalitatile oferite de container
- eficienta operatiilor (complexitate)

Containere - Clase template

- Container de tip secventa : vector<T>, deque<T>, list<T>
- Adaptor de containere
- Container asociativ

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

Container de tip secventa :

Vector, Deque, List sunt containere de tip secventa, folosesc reprezentari interne diferite, astfel operatiile uzuale au complexitati diferite

- Vector (Dynamic Array) :
  - elementele sunt stocate secvential in zone continue de memorie
  - vector are performante bune la :
    1. accesare elemente individuale de pe o pozitie data (constant time)
    2. iterare elemente in orice ordine (linear time)
    3. adaugare/stergere elemente de la sfarsit (constant amortized time)

- Deque (double ended queue) - coada cu acces la ambele capete
  - elementele sunt stocate in blocuri de memorie
  - elementele se pot adauga/sterge eficient de la ambele capete

- List :
  - implementat ca si lista dublu inlantuita
  - list are performante bune la :
    1. stergere/adaugare de elemente pe orice pozitie (constant time)
    2. mutare de elemente sau secvente de elemente in liste sau chiar si intre liste diferite (constant time)
    3. iterare de elemente in ordine (linear time)

Operatii / complexitate

```cpp
#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) << " ";
 }
}
```

```cpp
#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) << " ";
  }
}
```

```cpp
#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();
 }
}
```

Vector : timp constant O(1) random access; insert/delete de la sfarsit
Deque : timp constant O(1) insert/delete la orice capat
List : timp constant O(1) insert/delete oriunde in lista

Container asociativ

Sunt eficiente in accesarea elementelor folosind chei (nu folosind pozitii ca in cazul containerelor de tip secventa)

- set
  multime - stocheaza elemente distincte, elementele sunt folosite si ca si cheie
  nu putem avea doua elemente care sunt egale
  se foloseste arbore binar de cautare ca si reprezentare interna
- map, unordered_map
  dictionar care stocheaza elementele formate din cheie si valoare
  nu putem sa avem chei duplicate
- bitset
  container special pentru a stoca biti (elemente cu doar 2 valori posibile : 0 sau 1)

```
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 gestioneaza o pozitie (curenta) din containerul asociat. Ofera suport pentru traversare (++,--) si dereferentiere (*it).

Fiecare container STL include membre begind() si end(), perechea de iteratori descrie o secventa de tipul [first, last) - first inclusiv, last exclusiv.

end() - arata dupa ultimul element, nu este corect sa incercam sa luam valoare cu (*).

```
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;
}
```

Exista mai multe tipuri de iteratori :

- iterator input_output (istream_iterator, ostream_iterator)
- forward iterators, bidirectional iterators, random access iterators
- reverse iterators

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 sa suprascriem operatorii *, ++, ++, ==, != pentru a crea iteratori similari cu cei din STL
Daca dorim sa folosim vectorul intr-un **range-based for loop (foreach)** avem nevoie de metodele begin() si end() in clasa VectorDinamic

```
IteratorVector VectorDinamic::begin() const
{
	return IteratorVector(*this);
}

IteratorVector VectorDinamic::end() const
{
	return IteratorVector(*this, lg);
}
```

```
Element& operator*() {
	return element();
}
IteratorVector& operator++() {
	next();
	return *this;
}
```

Acum putem folosi :

```
//testam iteratorul
auto it = v.begin();
while (it != v.end())
{
	auto p = *it;
	assert(p.getPrice() > 0);
	++it;
}
```

```
for (auto& p : v) {
	std::cout << p.getType() << std::endl;
	assert(p.getPrice() > 0);
}
```

STL Algorithms

O multime de algoritmi sunt implementati in STL, Ei se afla in modulul <algorithm> si namespace-ul std.

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

In general sunt functii face primesc o pereche de iteratori (begin(), end()). Perechea de iteratori descrie o secventa de elemente (un range) = [a,b)

Acesti algoritmi pot fi folositi 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.
Un predicat este o functie care primeste un parametru (sau mai multi) si returneaza un bool (true sau false).
Exemple de predicate :

- **Functie normala** :

```
bool simpleFct(int a) {
    return a % 2 == 0; // întoarce true dacă a este par
}

int nrPare = count_if(v.begin(), v.end(), simpleFct);
```

- **Functor (Function Object)** :

Un functor este o clasa care suprascrie oferatorul () ca sa se comporte ca o functie.

```
class FunctionObj {
public:
    bool operator()(int a) { return a % 2 == 0; }
};

int nrPare = count_if(v.begin(), v.end(), FunctionObj());
```

- **Functie lambda** :

O lambda este o functie scrisa direct in locl unde ai nevoie de ea, fara sa o declari separat.

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

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

```
#include <functional>
vector<int> v{1, 2, 3, 4, 5, 6};

sort(v.begin(), v.end(), less<int>());    // sortare crescătoare
sort(v.begin(), v.end(), greater<>());    // sortare descrescătoare
```

Functii lambda

Functii (fara nume), se pot defini direct in locul unde e nevoie de o functie. Este utila in cazul algoritmilor STL. E o sintaxa simplificata de a creea functori in care compilatorul genereaza o clasa care suprascrie operatorul ().

Sintaxa :
[capture-list](params){body}
capture-list = care sunt variabilele din scopul curent care se vad in interiorul functiei lambda
poate sa fie vid [] - nu capteaza nimic, nu se vede nici o variabila
[=] - se vad toate variabilele din afara in corpul functiei lambda, se transmit prin valoare
[&] - se vad toate variabilele din afara in corpul functiei lambda, se transmit prin referinta
[a, &b] - se vede a (prin valoare) si b (prin referinta)
params = parametrii functiei lambda
body = corpul functiei 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

```
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;
}
```

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

Tratarea exceptiilor

Situatii anormale apar in timpul executiei (nu exista fisier, nu mai exista spatiu pe disk, etc), trebuie sa tratam aceste situatii.

Elemente :

- **try** block marcheaza blocul de instructiuni care poate arunca exceptii
- **catch** block bloc de instructiuni care executa in cazul in care apare o exceptie
- instructiunea **throw** este under the bus mecanism prin care putem arunca (genera exceptii)

```
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 ocurred.";
 } catch (...) {
      //error handling - intra aici la orice eroare
 }
}
```