

Capitolul 5. Templates

Motivație

Să se scrie o funcție care găsește maximumul dintre 2 valori.

Motivație

```
#include <iostream>
```

```
int get_max(int a, int b){  
    return (a<b)?b:a;  
}
```

```
float get_max(float a, float b){  
    return (a<b)?b:a;  
}
```

```
double get_max(double a, double b){  
    return (a<b)?b:a;  
}
```

```
int main(){  
    std::cout << get_max(2, 3) << std::endl;  
    std::cout << get_max(2.2f, 3.3f) << std::endl;  
    std::cout << get_max(2.5, 3.5) << std::endl;  
  
    return 0;  
}
```

Este nevoie de 3 funcții care fac același lucru, însă pentru tipuri de date diferite.

```
3  
3.3  
3.5
```

```
Process returned 0 (0x0)   execution time : 0.032 s  
Press any key to continue.
```

Motivație

Să se scrie o clasă care implementează o listă de date simplu înlănțuită.

Motivație

```
class Nod{
private:
    int data;
    Nod *next;
public:
    Nod(int data):data(data), next(nullptr){}
    void set_next(Nod *next) {
        this->next = next;
    }
    Nod* get_next() {
        return next;
    }
    int get_data(){
        return data;}
};
```

Motivație

```
class Lista_int{
private:
    Nod *head;
    Nod *tail;
public:
    Lista():head(nullptr), tail(nullptr){}
    void insert_elem(int data){
        Nod *temp = new Nod(data);
        if (head==nullptr) {
            head=tail=temp;
        }
        else {
            tail->set_next(temp);
            tail = tail->get_next();
        }
    }
    void display(){
        while(head){
            std::cout << head->get_data() << " ";
            head = head->get_next();
        }
        std::cout << std::endl;
    }
    // ...
};
```

Motivație

```
int main(){
    Lista l_int;
    l_int.insert_elem(5);
    l_int.insert_elem(10);
    l_int.insert_elem(14.3);
    l_int.display();
    return 0;
}
```

5 10 14

Process returned 0 (0x0) execution time : 0.031 s
Press any key to continue.

Problemă dacă inserăm o valoare reală!

Motivație

```
class Nod_int{
private:
    int data;
    Nod_int *next;
public:
    Nod_int(int data):data(data), next(nullptr){}
    void set_next(Nod_int *next) {
        this->next = next;
    }
    Nod_int* get_next() {
        return next;
    }
    int get_data(){
        return data;
    }
};
```

```
class Nod_double{
private:
    double data;
    Nod *next;
public:
    Nod_double(double data):data(data), next(nullptr){}
    void set_next(Nod_double *next) {
        this->next = next;
    }
    Nod_double* get_next() {
        return next;
    }
    double get_data(){
        return data;
    }
};
```


Motivație

```
class Lista_int{
private:
    Nod_int *head;
    Nod_int *tail;
public:
    Lista_int():head(nullptr), tail(nullptr){}
    void insert_elem(int data){
        Nod_int *temp = new Nod_int(data);
        if (head==nullptr) {
            head=tail=temp;
        }
        else {
            tail->set_next(temp);
            tail = tail->get_next();
        }
    }
    void display(){
        while(head){
            std::cout << head->get_data() << " ";
            head = head->get_next();
        }
        std::cout << std::endl;
    }
};
// ...
```

```
class Lista_double{
private:
    Nod_double *head;
    Nod_double *tail;
public:
    Lista_double():head(nullptr), tail(nullptr){}
    void insert_elem(double data){
        Nod_double *temp = new Nod_double(data);
        if (head==nullptr) {
            head=tail=temp;
        }
        else {
            tail->set_next(temp);
            tail = tail->get_next();
        }
    }
    void display(){
        while(head){
            std::cout << head->get_data() << " ";
            head = head->get_next();
        }
        std::cout << std::endl;
    }
};
// ...
```

Motivație

```
int main(){
    Lista_int l_int;
    l_int.insert_elem(5);
    l_int.insert_elem(10);
    l_int.insert_elem(15);
    l_int.display();
    return 0;
}
```

5 10 15

Process returned 0 (0x0) execution time : 0.036 s
Press any key to continue.

```
int main(){
    Lista_double l_double;
    l_double.insert_elem(5.1);
    l_double.insert_elem(10.1);
    l_double.insert_elem(15.1);
    l_double.display();
    return 0;
}
```

5.1 10.1 15.1

Process returned 0 (0x0) execution time : 0.030 s
Press any key to continue.

Motivație

Rescrierea funcțiilor/claselor în întregime doar pentru a corespunde unui nou tip de date este redundantă => este necesar un mecanism care poate adapta codul la orice tip de date.

Templates

def

Templates sunt un mecanism al limbajului C++ care permite claselor și funcțiilor să opereze cu tipuri de date generice. Templates sunt declarate o singură dată și instanțele template se generează în momentul compilării programului.

Tipuri de template:


- function template: familie de funcții
- class template: familie de clase
- alias template: un alias către o familie de tipuri (C++11)
- variable template: familie de variabile (C++14)
- constrângeri și concepte: concept (C++20)

Templates

Sintaxă:

parantezele unghiulare <> fac parte din sintaxa

template <lista_parametri> declaratie



Templates – lista de parametri

Sintaxă:

`template <lista_parametri> declaratie`

- listă de parametri, separați prin virgulă
- sunt acceptate 3 tipuri de parametri (cu posibilitatea menționării unei valori implicite):
 1. parametri tip (type template parameter):
 - cuvintele cheie `typename`, `class` (în acest caz, sunt interschimbabile): reprezintă orice tip de date definit de către limbaj sau de utilizator
 2. parametri non-tip (non-type template parameter):
 - referință (la un obiect sau la o funcție) a unei `lvalue`
 - un tip de valoare întreagă: `bool`, `char`, `(un)signed integer`
 - un tip de pointer (către un obiect sau o funcție)
 - un tip de pointer către membru (obiect membru sau funcție membră)
 - enumerație
 - `std::nullptr_t`; (începând cu C++11)
 - un tip de valoare în virgulă mobilă: `float`, `double`, `long double` (începând cu C++20)
 3. parametri template (template template parameter)

Templates – declaratia

Sintaxă:

```
template <lista_parametri> declaratie
```

declaratie poate reprezenta:

- o clasă/structură/reuniune;
- o clasă membră sau tip de enumerație membră;
- o funcție sau o funcție membră;
- o dată membră statică în domeniul de definiție al unui namespace;
- o variabilă sau o dată membră statică în domeniul de definiție al clasei (începând cu C++14);
- un alias al unui template (începând cu C++11);
- o specializare a unui template.

Templates – instanțiere

- De sine stătător, un template nu reprezintă un tip de date, o funcție sau un obiect => nu generează cod dacă template-ul este doar definit.
- Dacă, în timpul compilării, este întâlnită utilizarea template-ului, atunci este creată și funcția/clasa declarată în template, cu tipul de date specificat de argumente.

Templates – instanțiere

```
#include <iostream>

template<typename T>
T f(T t){
    return t/2;
}

int main(){
    std::cout << f(2) << std::endl;
    std::cout << f(2.5) << std::endl;
    return 0;
}
```

```
template<class T>
T f(T t){
    return t/2;
}
```

```
int f(int t){
    return t/2;
}
```

```
double f(double t){
    return t/2;
}
```

Templates – instanțiere

```
#include <iostream>
```

```
template<typename T>
```

```
T f(T t){
```

```
    return t/2;
```

```
}
```

```
int main(){
```

```
    std::cout << f(2) << std::endl;
```

```
    std::cout << f(2.5) << std::endl;
```

```
    return 0;
```

```
}
```

Lista de parametri

Lista de argumente

Template de clase

Un template de clase definește o familie de clase.

Declarare:

```
template<lista_parametri> cheie_clasa nume_clasa{...};  
cheie_clasa poate fi unul dintre: class, struct, union
```

Instanțiere explicită (utilizată, în special, la crearea bibliotecilor statice):

```
template cheie_clasa nume_clasa <lista_argumente>;
```

Intanțiere implicită:

```
nume_clasa<lista_argumente> nume_obiect;
```

Template de clase – exemplu #1

```
#include <iostream>

template<typename T> class C{
private:
    T *elemente;
    int dim;
public:
    C():elemente(nullptr), dim(0){}
    C(T *elemente, int dim):dim(dim){
        this->elemente = new T[dim];
        for(int i=0; i<dim; i++){
            *(this->elemente+i) = *(elemente+i);
        }
    }
    void display(){
        for(int i=0; i<dim; i++){
            std::cout << *(this->elemente+i) << "\t";
        }
        std::cout << std::endl;
    }
};
```

Parametru de tip

```
int main(){
    double d_vec[] = {1.23, 3.45};
    int i_vec[] = {1, 3};
    std::string s_vec[] = {"str1", "str 2"};

    C<double> c1(d_vec, 2);
    C<int> c2(i_vec, 2);
    C<std::string> c3(s_vec, 2);

    c1.display();
    c2.display();
    c3.display();

    return 0;
}
```

Sunt create 3 clase diferite, conform template

```
1.23      3.45
1          3
str1      str 2

Process returned 0 (0x0)   execution time : 0.155 s
Press any key to continue.
```

Template de clase – exemplu #2

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
```

Parametru de tip

Parametru non-tip

```
template<class T, int dim> class C{
private:
    T elemente[dim];
public:
    C(){
        for(int i=0; i<dim; i++){
            elemente[i] = static_cast <T> (rand()) /
static_cast <T> (rand()));
        }
    }
    void display(){
        std::cout << std::fixed << std::setprecision(3);
        for(int i=0; i<dim; i++){
            std::cout << elemente[i] << "\t";
        }
        std::cout << std::endl;
    }
};
```

```
int main(){
    C<int, 5> c1;
    C<float, 6> c2;
    C<double, 4> c3;

    c1.display();
    c2.display();
    c3.display();

    return 0;
}
```

```
0      0      1      0      1
0.203  1.384  20.287  0.251  0.888  2.218
25.503  0.024  0.931  0.991
```

```
Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

Template de clase – exemplu #3

```
#include <iostream>
```

```
template<class U> class A{  
public:  
    U a;  
    A() {std::cout << "Constructor template A\n";}  
};
```

Parametru template



```
template<class T, template<class U> class A> class C{  
private:  
    A<T> obj;  
public:  
    C() {}  
    void display(){  
        std::cout << obj.a << std::endl;  
        std::cout << typeid(obj.a).name() << std::endl;  
    }  
};
```

```
int main(){  
    C<int, A> c;  
    c.display();  
  
    return 0;  
}
```

```
Constructor template A
```

```
0  
i
```

```
Process returned 0 (0x0)   execution time : 0.029 s  
Press any key to continue.
```

Template de funcții

Un template de funcții definește o familie de funcții.

Sintaxă:

```
template<lista_parametri> declaratie_functie
```

Instanțiere explicită (utilizată, în special, la crearea bibliotecilor statice):

```
template tip_returnat nume_functie [<lista_argumente>]  
(lista_parametri);
```

Dacă tipurile de date ale argumentelor pot fi deduse din context, atunci lista de argumente poate să lipsească

Instanțiere implicită:

```
nume_functie [<lista_argumente>] (lista_parametri);
```

Template de funcții

```
#include <iostream>

template<typename T>
void f(T s)
{
    std::cout << s << '\n';
}

int main()
{
    f<double>(1); // instantiaza si apeleaza f<double>(double)
    f<>('a');     // instantiaza si apeleaza f<char>(char)
    f(7);         // instantiaza si apeleaza f<int>(int)
    void (*pf)(std::string) = f; // instantiaza f<string>(string)
    pf("V");      // apeleaza f<string>(string)
}
```


Template – specializări

Dacă ne dorim să avem o implementare diferită a template-ului pentru un anumit tip de date din lista de parametri, putem defini un comportament separat pentru tipul respectiv de date folosind o specializare.

Template – specializări

```
#include <iostream>

template <class T>
class mycontainer {
    T element;
public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

template <>
class mycontainer <char> {
    char element;
public:
    mycontainer (char arg) {element=arg;}
    char increase () {
        if ((element>='a') && (element<='z')) {
            return element+'A'-'a';
        }
    }
};
```

```
int main () {
    mycontainer<int> myint (7);
    mycontainer<char> mychar ('j');
    std::cout << myint.increase() << std::endl;
    std::cout << mychar.increase() << std::endl;
    return 0;
}
```

Template – specializări

```
#include <iostream>
```

```
template <class T>  
T increase(T arg){  
    return ++arg;  
}
```

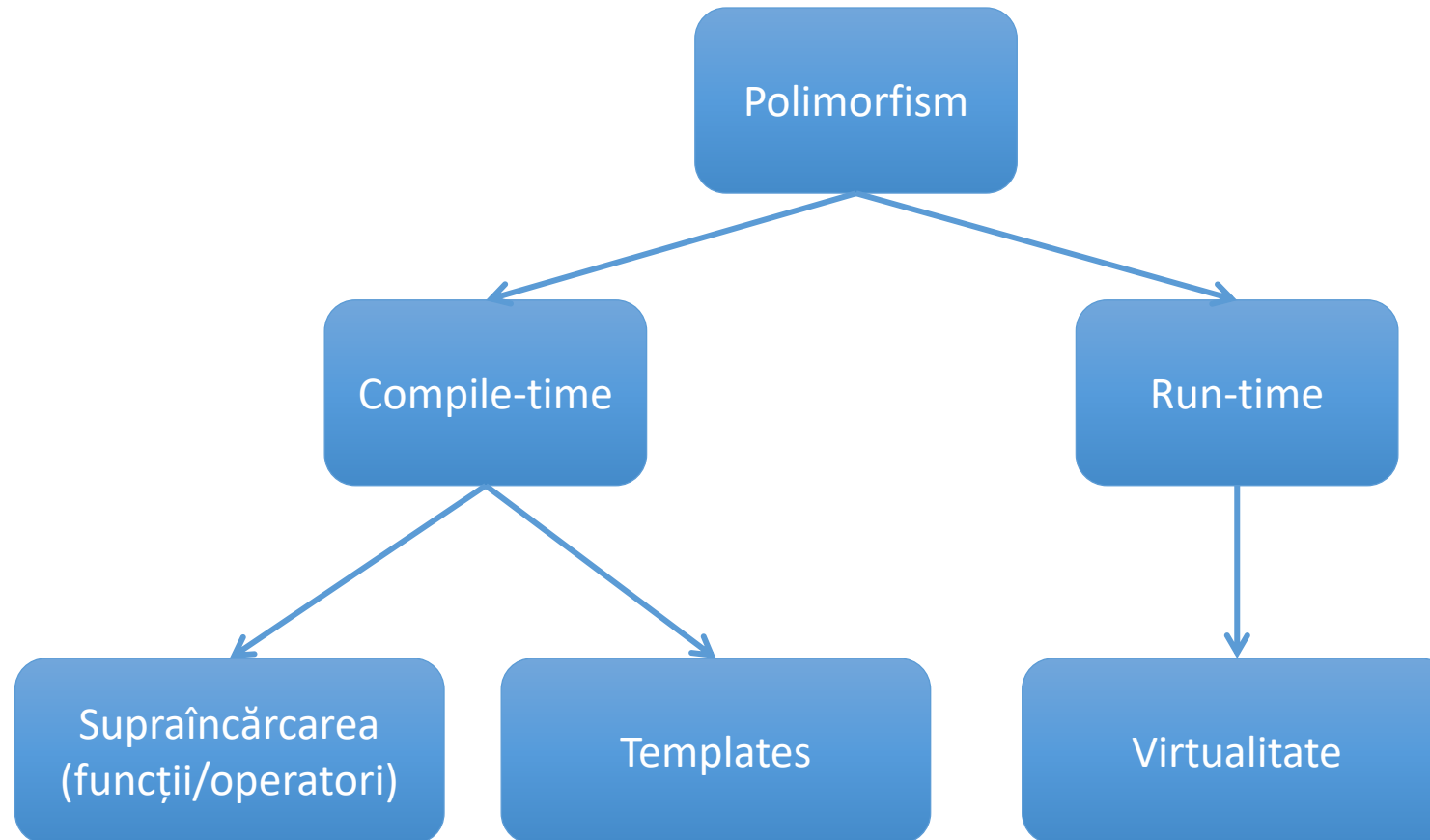
```
template <>  
char increase(char arg){  
    if ((arg>='a') && (arg<='z')){  
        return arg+'A'-'a';  
    }  
}
```

```
int main(){  
    std::cout << increase(5) << std::endl;  
    std::cout << increase(5.0) << std::endl;  
    std::cout << increase('b') << std::endl;  
    return 0;  
}
```

Principiile POO

1. Încapsulare = gruparea/învelirea/încapsularea datelor și a funcțiilor ce acționează asupra acestora într-un singur container (clasă).
2. Abstractizare = procedeul prin care se expun lumii exterioare doar funcționalitățile importante, fără a se intra în prea multe detalii.
3. Moștenire = procesul prin care o clasă este extinsă într-o nouă clasă prin preluarea datelor și funcțiilor membre.
4. Polimorfism = abilitatea obiectelor de tipuri diferite de a avea o aceeași interfață, însă cu implementări diferite.

Polimorfism



Motivație – soluție

Să se scrie o funcție care găsește maximul dintre 2 valori.

```
#include <iostream>

template<typename T>
T get_max(T a, T b){
    std::cout << "A fost utilizat tipul de date: " << typeid(T).name() << std::endl;
    return (a<b)?b:a;
}

int main(){
    std::cout << get_max(2, 3) << std::endl;
    std::cout << get_max(2.2f, 3.3f) << std::endl;
    std::cout << get_max(2.5, 3.5) << std::endl;
    return 0;
}
```

Motivație – soluție

Să se scrie o clasă care implementează o listă de date simplu înlănțuită.

```
#include <iostream>

template<typename T>
class Nod{
private:
    T data;
    Nod *next;
public:
    Nod(T data):data(data), next(nullptr){}
    void set_next(Nod *next) {this->next = next;}
    Nod* get_next() {return next;}
    T get_data() {return data;}
};
```

```
template<typename T>
class Lista{
private:
    Nod<T> *head, *tail;
public:
    Lista():head(nullptr), tail(nullptr){}
    void insert_elem(T data){
        Nod<T> *temp = new Nod<T>(data);
        if (head==nullptr) {
            head=tail=temp;
        }
        else {
            tail->set_next(temp);
            tail = tail->get_next();
        }
    }
    void display(){
        while(head){
            std::cout << head->get_data() << " ";
            head = head->get_next();
        }
    }
};
```

Motivație – soluție

Să se scrie o clasă care implementează o listă de date simplu înlănțuită.

```
int main() {  
    Lista <double> lista;  
    lista.insert_elem(5.5);  
    lista.insert_elem(10.5);  
    lista.insert_elem(12.4);  
    lista.display();  
    return 0;  
}
```


Sfârșit capitol 5