C9: Functii si clase template

Cuprins:

- Functii template
- Clase template
- Derivare din si de clase template

C9: Functii si clase template

Functii template

- template = sablon

- utilizate pentru a substitui implementari multiple pentru o functie atunci cand diferenta dintre implementari apare numai prin faptul ca folosim alte tipuri de date (semnatura: acelasi numar de parametri, dar tipuri diferite ale acestora)

Exemplu: s-ar putea ca intr-o aplicatie sa avem nevoie de mai multe functii care sa calculeze suma elementelor unor vectori; vectorii difera prin tipul de date stocate : int, double, complex precum si prin dimensiune

```
int suma(int* vec,int dim)
{
   int s(0);

   for (int i=0; i<dim ;i++)
            s=s+vec[i];

   return s;
}
//functia e specializata pentru int
```

```
double suma(double * vec,int dim)
{
 double s(0);

   for (int i=0; i<dim ;i++)
            s=s+vec[i];

   return s;
}
//pentru cazul in care vectorul este de tip
//double functia trebuie supradefinita si e
//specializata pentru tipul double
```

```
complex suma(complex * vec,int dim)
{
 complex s(0);//constructor cu parametri

   for (int i=0; i<dim ;i++)
            s=s+vec[i];// operator de atribuire si de adunare pentru tipul de date complex

   return s; //constructor de copiere pentru tipul de date complex
}
```

pentru cazul in care vectorul este de tip complex functia trebuie supradefinita si e specializata pentru tipul complex.

```
class complex
{ double re,im;
  public:
//automat au fost generati constructorul de copiere si operatorul de atribuire
//mai trebuie implementat un constructor care initializeaza cu 0; operator+
       complex(double r=0,double i=0){ re=r; im=i;}

       friend complex operator+(const complex &x,const complex &y){
              return complex(x.re+y.re,x.im+y.im);        }

      friend ostream& operator<<(ostream &d, const complex& x)
       {   d<<x.re<<" j"<<x.im;
          return d;        }//daca vrem sa afisam suma
};
```

- *este recomandat sa se gaseasca o cale de generalizare a acestor functii specializate pentru anumite tipuri de date, dar cu comportament (implementare) similara*
- *generalizarea este facuta cu "template" –uri - iar tipul de date care se poate modifica intre implementari devine un fel de parametru pentru functie*

Sablonul pentru functia suma:

```
template <typename T>
T sum(T* vec, int dim)
{
   T s(0);

   for (int i=0; i<dim ;i++)
            s=s+vec[i];

   return s;
}
//inlocuieste cele 3 implementari
//specializate pentru int, double si,
//respectiv, complex cu implementarea
//parametrizata dupa T
```

Functiile template folosesc cuvantul template pentru a preciza ca e vorba de o functie speciala si intre **< >** se precizeaza care sunt **tipurile generice** folosite de functie: **< Continut >**

Continutul poate sa fie un tip de date generic (sau o enumerare de tipuri de date generice):
ex. Tipul generic T e specificat prin:

typename T sau **class** T

Putem sa enumeram mai multe tipuri generice: <typename X, typename Y>

Functia este **parametrizata** dupa tipurile de date generice, nu **specializata** ca in cazurile precedente.

OBS: Daca nu foloseam functia template, trebuia sa rescriem acelasi cod de mai multe ori si nu se schimba decat tipul de date manevrat.

```
int main()
  {
    int n;
    cin>>n;

    int *v=new int[n];
    double *v1=new double[n];
    complex *v2=new complex[n]; //constructor fara parametru pentru complex

    for (int i=0; i<n ;i++){
      v[i]=i;
      v1[i]=(double)i/2;
      v2[i]=complex(i,i); //constructor cu parametri pentru complex
    }

    cout<<sum(v,n)<<endl;   //apel functie template pentru tipul int; specializare int
        //sau pot sa precizez ce specializare are  :   sum<int> (v,n);
    cout<<sum(v1,n)<<endl; //apel functie template pentru tipul double; specializare double
    cout << sum (v2,n)<<endl;//apel functie template pentru tipul complex; specializare complex
    //folosesc operatorul de afisare pentru tipul de date complex; a fost implementat
    //ce face compilatorul cand intalneste un astfel de apel de functie?
    // cate functii suma exista dupa compilare?
     //la compilare ->se genereaza functii - cate specializari am facut
    return 0;
    }
```

Alt exemplu :

```
void afis(const int& x)
{
  cout <<x<<endl;
}

void afis(const double& x)
{
  cout <<x<<endl;
}

void afis(const complex& x)
{
  cout <<x<<endl;
}
//utilizare

      int x=3;
      double dd=2.00;

      afis(x);
      afis(dd);
      afis(complex(10,10));
```

```
template <typename T>
void afis(const T& x)
{
  cout <<x<<endl;
}
//functie parametrizata dupa T

//utilizare

int x=3;
double dd=2.00;

afis(x);//functia template e specializata pt. int
afis<int>(x);//pot sa precizez specializarea, dar
//nu e obligatoriu

afis(dd);   //functia template e specializata pt.
            //double
//echivalent cu: afis<double>(dd);

afis(complex(10,10)); //functia template e
                      //specializata pt. complex
 //echivalent: afis<complex>(complex(10,10));
```

C9: Functii si clase template

Functii template - Definitii si observatii:

Functiile sablon (template) sunt functii speciale care folosesc **tipuri generice(*generic types*).**

Functionalitatea lor poate sa fie adaptata pentru mai mult de un tip de date fara a rescrie functia pentru fiecare tip de date in parte - **parametrizare dupa tip.**

Cuvantul template informeaza compilatorul ca e vorba de o functie speciala.

Un parametru template este un tip special de parametru care poate sa fie folosit pentru a transmite un tip generic ca argument intr-o functie(cum un parametru obisnuit al unei functii poate fi folosit ca sa transmita valori in acea functie, la fel si un parametru template poate sa transmita un tip de date intr-o functie).

Functiile template se declara astfel:

```
template <class tip1, … ,class tipN>
tip_de_date_returnat nume_functie(semnatura){…};
```

sau

```
template <typename tip1, …, typename tipN >
tip_de_date_returnat nume_functie(semnatura){…};
```

Nu exista nicio deosebire intre cele doua forme.

//Functie pentru afisarea minimului dintre 2 valori de tip int, double sau float (etc).

```
template <class T, class U>
inline void getMin (T a, U b)
{
  cout << (a<b?a:b);
}
    //utilizare
       int x=3;
       double dd=3.10;
       //pot preciza exact ce tipuri de date folosesc la apel
        getMin<double,int>(dd,x);  //functia e specializata pentru tipurile de date double si int

      //sau, acelasi lucru:
        getMin (dd,x); //functia e specializata pentru tipurile de date double si int
```

OBS:

Tipul unui parametru template este determinat in momentul compilarii => **La apelul unei functii template – legarea se face in mod static =>**

=>NU o sa putem avea functii virtuale template.

C9: Functii si clase template

Parametrii care nu reprezinta un tip de date in functii template

In afara de tipuri generice, functiile template pot sa fie parametrizate si dupa parametri obisnuiti, similari cu cei folositi in functii.

```
template <typename T, int dim>
T sum(T* vec) {
   T s(0);
    for (int i=0; i<dim ;i++)
    s=s+vec[i];
    return s;
}
//utilizare
int n=10;
int *v=new int[n];
double *v1=new double[n];

for (int i=0; i<n ;i++){
          v[i]=i;
          v1[i]=(double)i/2;
}
 cout<<sum<int,10>(v)<<endl; //functia e specializata pentru int, iar dim este 10
 cout<<sum<double,10>(v1)<<endl; //functia e specializata pentru double, iar dim este 10
```

C9: Functii si clase template

Tipuri generice default (C++11):

```
template <typename T=int, int dim=10>
T sum(T* vec) {
   T s(0);
    for (int i=0; i<dim ;i++)
    s=s+vec[i];
    return s;
}

//apel pentru un vector de intregi v si un vector cu elemente double v1

 cout<<" suma pt. vector de tip double cu dimensiunea 5: "  <<  sum<double,5>(v1)  <<endl;
 cout<<"  suma pt. vector de tip int cu dimensiunea 10: "<< sum<int>(v)<<endl;
 cout<<" suma pt. vector de tip int cu dimensiunea 10: "<< sum<>(v)<<endl;
 cout<<" suma pt. vector de tip int cu dimensiunea 10: "<< sum(v)<<endl;
```

OBS: Nu toate versiunile de compilatoarele permit argumente (tipul sau valoarea) template default pentru functii (permis de la C++11 in sus).

Dati exemple de functii care pot fi facute template!

“Teme”

1. Realizati o functie template care interschimba valorile a doua variabile

2. Realizati o functie template pentru sortarea elementelor unui vector.

Ce trebuie implementat intr-o clasa (tip de date definit de utilizator) astfel incat acel tip sa specializeze functia de sortare?

3. Implementati o functie pentru afisarea elementelor unui vector, parametrizata dupa tipul T si dimensiunea de tip int - dim.

C9: Functii si clase template

Metode ale unei clase ca functii template

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

class VectorInt10
 {
      int buf[10];

 public:

        template<typename T>
        void copie_cu_convesie(T sursa[10]){
              for(int i = 0; i<10; i++)
                    buf[i] = (int)sursa[i];
        }

//alte metode
};
```

```
int main()
{
   int v[10];
   double v1[10];
   float v2[10];

  for (int i = 0; i<10; i++) {
            v[i]=i;
            v1[i]=(double)i/2;
            v2[i]=(float)i/3;
    }

    VectorInt10 vobj;
    vobj.copie_cu_convesie(v);
    vobj.copie_cu_convesie(v1);
    vobj.copie_cu_convesie(v2);
    //echivalent cu :
//  vobj.copie_cu_convesie<float>(v2);

    return 0;
    }
```

C9: Functii si clase template

Clase template

Exista posibilitatea realizarii de clase template – cu atribute template si/sau functii membre care folosesc atribute de tipuri generice

Sintaxa:

```
template <class tip1, … ,class tipN>
class nume_clasa
{//tip1 a;  //…
  //tipN*vec;
  //metode
};
```

Dati exemple de clase care ar putea sa fie implementate ca template!

sau:

```
template <typename tip1, …, typename tipN >
class nume_clasa
{//…
};
```

OBS: Ca si in cazul functiilor template, se pot transmite si parametri care nu sunt tipuri de date intre <>

```
#include <iostream>
using namespace std;
#include <math.h>
//tip parametrizat pereche<X,Y>
template <typename X, typename Y>
class pereche
{ X elem1;
  Y elem2;
public:
 pereche(const X &e1, const Y &e2){
        elem1=e1;
        elem2=e2;       }

 friend ostream& operator<<(ostream &d,const pereche &x){
         d<<x.elem1<<" si " << x.elem2<<endl;
         return d;       }
};
    int main(){
      pereche<int, double> p(4,3.00);//instantiere a unui
//obiect pereche specializat pentru tipurile int si double
        cout<<p;
        complex x(1,1);
        pereche<complex,double> p1(x, x.getModul());
        cout<<p1;
        return 0; }
```

```
class complex
{ double re,im;

public:
complex(double r=0,double i=0){re=r;im=i;}

friend ostream& operator<<(ostream &d,const complex &x)
 { d<<x.re<<" j"<<x.im;
   return d;
}

 double getModul()
 {  return sqrt(pow(re,2)+pow(im,2));}
};
```

Pentru tipurile cu care se face specializarea clasei pereche trebuie sa avem implementate metodele:

- operator=
- operator<<
- constructor fara param. (unde se foloseste? pot sa fac altfel implementarea?)

C9: Functii si clase template

Tipuri template default

```
template <typename X, typename Y=X>
class pereche
{ X elem1;
  Y elem2;//….
};
//instantiez un obiect de tip pereche specializat pentru tipurile de date int si int
    Pereche<int,int> p;
//sau
    Pereche<int> p;
```

```
template <typename X=int, typename Y=X>
class pereche
{ X elem1;
  Y elem2;//…
};
//instantiez un obiect de tip pereche specializat pentru tipurile de date int si int
    Pereche<int,int> p;
//sau
    Pereche<int> p;
//sau
    Pereche<> p;
```

OBS: pentru declararea obiectelor de un anumit tip parametrizat (pt. specializare) trebuie sa precizez clar care e tipul de date pt. specializare. **Nu pot sa scriu: Pereche p!**

C9: Functii si clase template

Observatii:

1. Functiile template sunt un exemplu de polimorfism la compilare(compile time polymorphism) ca si supradefinirea functiilor.
2. Fiecare apel specializat al unei functii/clase template conduce, la compilare, la generarea cate unei instante pentru fiecare specializare.
3. Daca o functie/clasa template contine o variabila locala statica/atribut static – fiecare instantiere a clasei template va contine copia proprie a variabilei/atributului static.

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

template <typename T>
void fun(const T& x)
{
  static int i = 10;
  i = i + x;
  cout<<i;
  }
```

```
int main()
{
   fun<int>(1);                  //11
   cout << endl;

   fun<int>(2);                  //13
   cout << endl;

   fun<double>(1.1);             //11
   cout << endl;

  return 0;
}
```

Templateuri si separarea header-interfata

Din punct de vedere al compilatorului, clasele si functiile template nu sunt functii si clase obisnuite.

Ele sunt compilate la cerere, adica **codul nu e compilat pana la instantierea** cu argumente specifice.

In acel moment, **compilatorul genereaza o functie specifica** pentru acele argumente ale functiei template.

Observatie

Codul este de obicei impartit in mai multe fisiere sursa si header pentru a separa interfetele de implementari.

Deoarece clasele/functiile template sunt compilate la cerere, apar probleme de linkage cand **implementarea si declaratia functiilor friend e facuta in fisiere separate**.

In acest caz interfata si implementarea pot sa fie facute in acelasi fisier sau, daca e necesar, exista metode specifice pentru rezolvarea problemelor de separare : http://www.codeproject.com/Articles/48575/How-to-define-a-template-class-in-a-h-file-and-imp)

C9: Functii si clase template

Ce diferente apar daca separ headerul de implementare; daca am functii friend?

```
template <typename X, typename Y>
class pereche;                          //precizez ca va fi implementat tipul template pereche<X,Y>

template <typename X, typename Y>
ostream& operator<<(ostream &d,const pereche<X,Y>& x); //precizez ca va fi impl. o functie
                                                       //operator<< parametrizata dupa X si Y
template <typename X, typename Y>
class pereche
{   X elem1;
    Y elem2;
public:
          pereche(const X& , const Y &);
          X getElem1();
          Y getElem2();
          friend ostream& operator<< <>(ostream &d,const pereche& x);//precizez ca functia
                                                                    //friend e template
};

 template <typename X, typename Y>
 ostream& operator<<(ostream &d,const pereche<X,Y>& x){
        d<<x.elem1<<" si "<<x.elem2<<endl;
        return d;  }
```

```
template <typename X, typename Y>
pereche<X,Y>::pereche(const X &e1,
                      const Y &e2){
    elem1=e1;
    elem2=e2;
}

template <typename X, typename Y>
X pereche<X,Y>::getElem1(){
    return elem1;
}

template <typename X, typename Y>
Y pereche<X,Y>::getElem2() {
    return elem2;
}
```

```
int main()
{
  pereche<int, double> p(4,3.00);
   cout<<p.getElem1()<<" ";
   cout<<p.getElem2()<<endl;
   cout<<p;

   return 0;
   }
```

Metodele clasei sunt template.

```
//ce metode mai trebuie sa existe pentru
//tipurile cu care se face specializarea clasei
//pereche?
```

Constructori de copiere.

Ce clase se preteaza la implementari template (se pot parametriza)?

Clasele template sunt foarte utile pentru dezvoltarea de clase de tip container/colectie:

- tablouri 1,2,3,..-n - dim
- liste
- stiva
- cozi
- multimi, arbori
- tabele de dispersie, etc

C9: Functii si clase template

```
//Vector template
#include <iostream>
using namespace std;

//s-ar putea parametriza si dupa dim; tema
template <typename T>
class vector
{ int dim;
  T* buf;
  public:
    vector(int , T*);
    vector(const vector&);
    ~vector();
    vector & operator=(const vector &);
    T& operator[](int i);
    int getDim();

  //void addElem(int i) - cu realocare de memorie–tema
  //void removeElem(int i) - cu realocare de memorie–tema
  //implementati orice alte metode care vi se par utile (+,- *,etc) – tema
```

```
friend ostream & operator << (ostream & dev,
                              vector & v)
        {
         dev << "Vector:" <<endl;
         dev << "Nr Elem:" << v.dim << endl;
         for(int i = 0; i < v.dim; i++)
                  {
                  dev << v.buf[i];//operator<< pt T
                  dev << endl;
                  }
         return dev;
        }
  //Tema – mutati implementarea in afara clasei ;
//realizati fisiere header si cpp
         };
```

```
template <typename T>           //functie membra template parametrizata dupa T
vector<T>::vector(int d, T*b) {    //face parte din clasa template vector<T>
      dim=d;
      buf=new T[dim];           //constructor fara parametri pt. T
      for (int i=0;i<dim;i++)
        buf[i]=b[i];            //operator=pt. T
    }

template <typename T>
vector<T>::vector(const vector<T>& v) {   //parametrul functiei este de tipul vector
                                          //parametrizat dupa T
      dim=v.dim;
      buf=new T[dim];
      for (int i=0;i<dim;i++)
        buf[i]=v.buf[i];
}

template <typename T>
vector<T>::~vector() {

            delete [] buf;          //destructor pt. T

            }
//Tema: Verificati in constructori, destructor si operator= ca dim nu e 0/ buf nu e NULL
```

```
template <typename T>
vector<T>&  vector<T>::operator=(const vector<T> &v) {
        dim=v.dim;
        if (buf!=NULL) delete [] buf;          //tipul T necesita destructor
        buf=new T[dim];                        //tipul T necesita constructor fara parametri
        for (int i=0;i<dim;i++)
            buf[i]=v.buf[i];                //tipul T necesita operator=
         return *this;
}


template <typename T>
T& vector<T>::operator[](int i) { //returnez adresa un obiect de tipul elem. stocate in vector: T
         return buf[i];
}


template <typename T>
int vector<T>::getDim(){
         return dim;
}
```

```
//pentru a specializa clasa vector vom folosi tipurile: complex si , respectiv, cerc si dreptunghi
//derivate din punct, afisarea presupune afisarea atributelor, perimetrului si ariei
class complex
{ double re,im;
public: complex(double =0,double =0);
          friend ostream& operator<<(ostream &,complex );
};   //s-au generat automat operator=, constructor de copiere si destructor

class Punct
{ protected: double x,y;
  public:      Punct(double i=0,double j=0):x(i),y(j){}
     friend ostream& operator<<(ostream &d,const Punct* p){ //va functiona pt pointeri
         p->afisare(d);   // transmit d ca parametru ca sa pot sa afisez si in fisiere daca de interes
         return d;  }
     //friend ostream& operator<<(ostream &d,const Punct& p){//daca Punct nu era abstract,
     //     p.afisare(d);                    //trebuia sa implementez operatorul si pentru cazul
     //     return d;  }                     // general cand obiectul afisat nu e pointer
     virtual  void afisare(ostream &d) const { //poate vreau sa scriu intr-un fisier
            d<<"Perimetrul: "<<perimetrul()<<endl;
            d<<"Aria: "<<aria()<<endl;
            }
     virtual double perimetrul() const =0 ;
     virtual double aria() const =0 ;
     virtual ~Punct(){}
};
```

```
class Cerc:public Punct
{  double R;
   static const double pi=3.14;
  public:
     Cerc(double i=0,double j=0,double raza=0):Punct(i,j),R(raza){}
     void afisare(ostream &d) const
        {d<<"Raza:"<<R<<endl;
                Punct::afisare(d);                    }
     double perimetrul()  const   {return 2*pi*R;}
     double aria()  const  {return pi*R*R;}
 };

class Dreptunghi:public Punct
{double lung,lat;
   public:
     Dreptunghi(){}
     Dreptunghi(double i=0,double j=0,double l1=0,double l2=0):Punct(i,j),lung(l1),lat(l2){}
     void afisare(ostream &d) const {
                d<<"Laturile:"<<lung<<", "<<lat<<endl;
                Punct::afisare(d);
                }
     double perimetrul()const   {return 2*(lung+lat);}
     double aria()const   {return lung*lat;}
   };
//toate functiile sunt tratate ca inline deoarece au implementarea in header
```

```
int main(){
        int n;
        cin>>n;

        int *v=new int[n];
        double *v1=new double[n];
        complex *v2=new complex[n];

        for (int i=0; i<n ;i++){
            v[i]=i;
            v1[i]=(double)i/2;
            v2[i]=complex(i,i);
         }

        vector<int> vti(n,v);
        cout<<vti<<endl;

        vector<double> vtid(n,v1);
        for (int i=0; i<vtid.getDim() ;i++)
              cout<<vtid[i]<<"  "<<endl;

        vector<complex> vtic(n,v2);
        cout<<vtic<<endl;
```

```
//vtic=vtid;
//no match for 'operator=' in 'vtic = vtid'
//candidates are: vector<T>&
//vector<T>::operator=(const vector<T>&)
//[with T = complex]

vector<complex> cv(vtic);
 vtic=cv;
 cout<<vtic;


Punct**vpc=new Punct*[2];
vpc[0]=new Cerc(0,0,3);
vpc[1]=new Dreptunghi(4,4,4,4);

vector<Punct*> vp(2,vpc);
cout<<vp; //vreau sa afis toate atr.
//operator<< pentru pointeri din Punct
//apelaza o functie virtuala de afisare=>
//am comportamente diferite pentru vpc[0] si
//vpc[1]
return 0;
} //reprezentati vp in mem – si explicati cum
//functioneaza cout<<vp; pe desen
```

C9: Functii si clase template

Derivarea din /de clase template

Voi prezenta doar modele de derivare simpla (dar este permis orice tip de derivare: ex. din mai multe clase de baza template, publica, private, protected sau virtuala).

1. O clasa obisnuita derivata dintr-o clasa template de baza
2. O clasa template derivata dintr-o clasa obisnuita de baza
3. O clasa template derivata din alta clasa template de baza

OBS: O *clasa template* este un sablon pentru o clasa, ea va fi instantiata in functie de tipurile de date pe care le foloseste.
O instantiere conduce la **specializarea** acelei clase.
Procesul se numeste **instantiere**, iar rezulatul se numeste **specializare**.

Cand derivam trebuie sa ne punem problema ce urmeaza sa mostenim in clasa derivata:

o clasa template : Ceva_ce_foloseste<T>
sau specializarea ei : Ceva_ce_foloseste<int>

* clasa obisnuita - derivata dintr-o specializare a unei clase template

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

   template <typename T>
   class Baza{
   protected:
           T a1;
   public:
         Baza(const T &i){a1=i; }
         void afis() {cout<<a1;}
         T geta1(){return a1;}
   };

   class Der: public Baza<int>
   {//se mosteneste a1 de tip int si geta1
           int atr2;
   public:
           Der(int i, int j):Baza(i),atr2(j){}
       // Der(int i, int j):Baza<int>(i),atr2(j){}
           void afis() {      //redefinire
               Baza::afis(); // Baza<int>::afis();
               cout<<atr2;}
   };
```

```
int main()
{
    Der d(2,2); //Der nu e template
    d.afis();

    Baza<double> b(2.2);
    Baza<double> c(3.2);

  return 0;
 }
 //cate specializari ale clasei Baza exista
          //in acest program?
```

* clasa derivata dintr-o clasa template nespecializata

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

template <typename T>
class Baza{
protected:
        T a1;
public:
     Baza(T i){a1=i; }
     void afis() {cout<<a1;}
     T geta1(){return a1;}
};

template <typename T>
class Der: public Baza<T>
{
        int atr2;
public:
        Der(T i, int j):Baza<T>(i),atr2(j){}
        void afis() {
            Baza<T>::afis();
           //cout<<Baza<T>::geta1();
            cout<<atr2;}    };

int main()
{
    Der<int> d(2,2);
    d.afis();

    return 0;
    }
```

//clasa derivata este si ea template

Trebuie sa precizez ca Baza e parametrizata dupa T, altfel am eroare la compilare.

* clasa template derivata dintr-o clasa de baza neparametrizata

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

class Baza{
   protected:     int a1;
   public:
            Baza(int i){a1=i; }
            void afis() {cout<<a1;}
            int  geta1(){return a1;}
};

   template <typename T>
   class Der : public Baza{
  //se mosteneste a1 de tip int si geta1
            T a2;
   public:
             Der(int i, T j):Baza(i),a2(j){}
             void afis() {
                Baza::afis();
                 cout<<a2;
              }
   };
```

```
int main()
{
 Der<int> d(2,2);//Der specializata pt int
 d.afis();

 return 0;
 }
```

* clasa template derivata dintr-o alta clasa template

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

template <typename T>
class Baza{
protected:      T a1;
public:

          Baza(T i){a1=i; }
          void afis() {cout<<a1;}
          T  geta1(){return a1;}
};

template <typename T1,typename T>
class Der : public Baza<T>{
          T1 a2;
public:
          Der(T i, T1 j):Baza<T>(i),a2(j){}
          void afis() {
             cout<<Baza<T>::a1;
             //Baza<T>::afis();
              cout<<a2;
           }    };

 int main()
{
Der<int,int> d(2,2);
//Der e specializata pentru int si int
d.afis();

return 0;
}
```

Probleme propuse:

1. Folosind clasa Vector si Pereche realizati o clasa Dictionar cu facilitatea de cautare a unui cuvant si afisarea definitiei pentru acesta
   Un cuvant apare doar o data in dictionar. Cuvintele stau sortate alfabetic.

2. Implementati sub forma de clase template
- clasa stack (fara a folosi vector pentru stocarea elementelor ca in exemplul urmator)
- clasa matrice (cu supradefinirea operatorilor de *,+,etc)
- clasa arbore binar de cautare – de citit pentru data viitoare
- tabela dispersie (SDA – curs 7) – de citit pentru data viitoare

Specializati clasele pentru tipurile de date pereche, complex, int double, etc.

Folositi arborele/tabela de dispersie in loc de vector pentru a realiza dictionarul de la 1 – care sunt complexitatile pentru adaugarea / cautarea unui element?

```
//Stiva – implementata cu vector- LIFO
#include <iostream>
using namespace std;
#include <assert.h>

template <typename T>
class stack {
            T *p;          //un vector in care stochez elementele
            int nrmax;  // capacitate maxima
            int nrcrt;    // cate elemente am – poate sa lipseasca si ma folosesc de sp
            int sp;       // varful stivei; puteam sa folosesc doar sp, fara nrcrt
public:     stack(int = 15);
            ~stack();
            stack(const stack &);
            stack & operator= (const stack &);
            void push(const T&);   //adaug element in varf
            T pop(void);               //scot element din varf
            int empty(void);          //e stiva goala?
            int full (void);             //e stiva plina?
            friend ostream& operator<<(ostream& dev, stack&s){
                        for (int i=0; i < s.nrcrt; i++)
                                    dev << s.p[i] << " ";
                        dev << "\n";
                        return dev;};
};
```

```
template <typename T>
stack<T>::stack(int n){     //constructor cu parametru – capacitatea stivei
            if (n==0) p=NULL;
                  else p = new T[n];
            nrmax = n;
            nrcrt = 0;
            sp = -1;
}

template <typename T>
stack<T>::stack(const stack <T> & s) {    //constructor de copiere
            p = new T[s.nrmax];
            nrmax = s.nrmax;
             nrcrt = s.nrcrt;
            sp = s.sp;
            for (int i = 0; i < nrcrt; i++)
                        p[i] = s.p[i];
}//sau cu reutilizarea codului din operator=: p=NULL; *this=s;

template <typename T>
stack<T>::~stack()
{
            delete [] p;
}
```

```
template <typename T>
void stack<T>::push(const T& el){
        assert(nrcrt < nrmax);
        p[++sp] = el;
        nrcrt++;
}
//implementati functia cu facilitatea de a
//creste dimensiunea stivei in caz de
//depasire
```

```
template <typename T>
T stack<T>::pop(void) {
        assert(nrcrt > 0);
        nrcrt--;
        return p[sp--];
}
```

```
template <typename T>
int stack<T>::full(void){
        return (nrcrt == nrmax);
        }
```

```
template <typename T>
int stack<T>::empty(void){
        return (nrcrt == 0);
}
```

```
template <typename T>
stack<T> & stack<T>::operator=(const
        stack <T> & s){
    if (p != NULL) delete [] p;
    p=new T[nrmax = s.nrmax];
    nrcrt = s.nrcrt;
    sp = s.sp;
    for (int i = 0; i < nrcrt; i++)
        p[i] = s.p[i];
    return *this;
}
```

```
typedef stack<int> STI;   //tip de date STI: stiva specializata pentru int
int main(void)
{           stack<int> si;        //stiva specializata pentru int sau : STI si;
            stack<char > ss;    //stiva specializata pentru char

            int a[3] = { 1, 2, 3,};
            char s[3] = { '1', '2', '3'};
            for(int i = 0; i < 3; i++) {
                        si.push(a[i]);   //populez stivele cu elemente
                        ss.push(s[i]);          }

            stack<int> sj(si);          //creez cu constructorul de copiere stiva sj
            stack<char> st;
            st = ss;                    //atribuiri intre stive specializate pentru acelasi tip de date

            i = 0;
            while (!sj.empty())   a[i++] = sj.pop(); //scot elementele din stiva si le pun in vector
            i = 0;
            while (!st.empty())   s[i++] = st.pop(); //scot elementele din stiva si le pun in vector

            for(i = 0; i < 3; i++)   cout << a[i] << " ";  cout <<endl;

            for(i = 0; i < 3; i++)  cout <<endl << s[i] << " ";       cout <<endl;
                                                //ce se afiseaza?
```

```
sj.push(21); sj.push(22);   //pun elemente in stive

stack<STI> ssi;                 //creez o stiva de stive  de int
ssi.push (si); ssi.push(sj);  //adaug doua stive de int in ssi

cout << "si: " << si;
cout << "sj: " << sj;

cout << "Stiva de stive: ";
cout << ssi << "\n";                          // Se va apela operatorul << pentru
                                              // elementele componente

//afisam stiva in ordine inversa
si = ssi.pop();
cout << "si: " << si;
 sj = ssi.pop();
cout << "sj: " << sj;

 return 0;

}
```