Facultatea de Matematica si Informatica
Universitatea din Bucuresti

Programare orientata pe obiecte

- suport de curs -

Dobrovat Anca - Madalina

An universitar 2019 – 2020
Semestrul II
Seria 13

Cursuri 9 & 10

22 - 29/04/2020

Agenda cursului

1. Upcasting si downcasting (recapitulare curs anterior)

2. Controlul tipului în timpul rulării programului în C++.

- Mecanisme de tip RTTI (Run Time Type Identification).
- Moştenire multiplă şi identificatori de tip (dynamic_cast, typeid).

3. Parametrizarea datelor. Şabloane în C++. Clase generice

- Funcții şi clase Template: Definiţii, Exemple, Implementare.

1. Upcasting si downcasting (recapitulare)

Polimorfism la executie

- functiile virtuale sunt definite in baza si redefinite in clasa derivata

- pointer de tip baza care arata catre obiect de tip derivat si cheama o functie virtuala definita in baza si in derivata executa FUNCTIA DIN CLASA DERIVATA.

- Pentru clasa derivata: **late binding** - un obiect derivat folosit in locul obiectului de baza isi va folosi functia sa, nu cea din baza (din cauza de late binding).
- decuplare in privinta tipurilor.

Upcasting - Tipul derivat poate lua locul tipului de baza (foarte important pentru procesarea mai multor tipuri prin acelasi cod).

1. Upcasting si downcasting (recapitulare)

Upcasting

Expl.

```
class Pet { public:
  virtual string speak() const { return " "; } };

class Dog : public Pet { public:
  string speak() const { return "Bark!"; } };

int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Late binding for both:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Early binding (probably):
  cout << "p3.speak() = " << p3.speak() << endl;
}
```

1. Upcasting si downcasting (recapitulare)

Clase abstracte si functii virtuale pure

Clasa abstracta = clasa care are cel putin o functie virtuala PURA

Necesitate: clase care dau doar interfata (nu vrem obiecte din clasa abstracta ci upcasting la ea).

Eroare la instantierea unei clase abstracte (nu se pot defini obiecte de tipul respectiv).

Permisa utilizarea de pointeri si referinte catre clasa abstracta (pentru upcasting).

Nu pot fi trimise catre functii (prin valoare).

1. Upcasting si downcasting (recapitulare)

Functii virtuale pure

Sintaxa: **virtual** tip_returnat nume_functie(lista_parametri) =0;
Ex: virtual int pura(int i)=0;

Obs: La mostenire, daca in clasa derivata nu se defineste functia pura, clasa derivata este si ea clasa abstracta ---> nu trebuie definita functie care nu se executa niciodata

UTILIZARE IMPORTANTA: prevenirea “object slicing”.

1. Upcasting si downcasting (recapitulare)

Overload pe functii virtuale

Obs. Nu e posibil overload prin schimbarea tipului param. de intoarcere (e posibil pentru ne-virtuale)

De ce. Pentru ca se vrea sa se garanteze ca se poate chema baza prin apelul respectiv.

Exceptie: pointer catre baza intors in baza, pointer catre derivata in derivata

1. Upcasting si downcasting (recapitulare)

Constructori si virtualizare

OBS. NU putem avea constructori virtuali.

In general pentru functiile virtuale se utilizeaza late binding, dar in utilizarea functiilor virtuale in constructori, varianta locala este folosita (early binding)

De ce?

Pentru ca functia virtuala din clasa derivata ar putea crede ca obiectul e initializat deja

Pentru ca la nivel de compilator in acel moment doar VPTR local este cunoscut

1. Upcasting si downcasting (recapitulare)

Destructori si virtualizare

Este uzual sa se intalneasca.

Se cheama in ordine inversa decat constructorii.

Daca vrem sa eliminam portiuni alocate dinamic si pentru clasa derivata dar facem upcasting trebuie sa folosim destructori virtuali.

1. Upcasting si downcasting (recapitulare)

Destructori si virtualizare

```
class Base1 {public:  ~Base1() { cout << "~Base1()\n"; } };

class Derived1 : public Base1 {public:  ~Derived1() { cout << "~Derived1()\n"; } };

class Base2 {public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {public:  ~Derived2() { cout << "~Derived2()\n"; } };

int main() {
  Base1* bp = new Derived1;
  delete bp; // Afis: ~Base1()
  Base2* b2p = new Derived2;
  delete b2p; // Afis: ~Derived2()  ~Base2()
}
```

1. Upcasting si downcasting (recapitulare)

Destructori virtuali puri

Utilizare: recomandat sa fie utilizat daca mai sunt si alte functii virtuale.
Restrictie: trebuiesc definiti in clasa (chiar daca este abstracta).
La mostenire nu mai trebuiesc redefiniti (se construieste un destructor din oficiu)
De ce? Pentru a preveni instantierea clasei.
Obs. Nu are nici un efect daca nu se face upcasting.

```
class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?
int main() { Derived d; }
```

1. Upcasting si downcasting (recapitulare)

Functii virtuale in destructori

La apel de functie virtuala din functii normale se apeleaza conform VPTR
In destructori se face early binding! (apeluri locale)
De ce? Pentru ca acel apel poate sa se bazeze pe portiuni deja distruse din obiect

```
class Base { public:
  virtual ~Base() {  cout << "Base1()\n";     f();   }
  virtual void f() { cout << "Base::f()\n"; }
};
class Derived : public Base { public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived; // Afis: ~Derived() Base1() Base::f()
  delete bp;
}
```

1. Upcasting si downcasting (recapitulare)

Downcasting

Folosit in ierarhii polimorfice (cu functii virtuale)

Problema: upcasting e sigur pentru ca respectivele functii trebuie sa fie definite in baza, downcasting e problematic

Explicit cast prin: **dynamic_cast**

Daca stim cu siguranta tipul obiectului putem folosi "static_cast".
Static_cast intoarce pointer catre obiectul care satiface cerintele sau 0.

Foloseste tabelele VTABLE pentru determinarea tipului

1. Upcasting si downcasting (recapitulare)

Downcasting

```
class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  Dog* d1 = dynamic_cast<Dog*>(b);  // Afis - 0; Try to cast it to Dog*:
  Cat* d2 = dynamic_cast<Cat*>(b);// Try to cast it to Cat*:
  // b si d2 retin aceeasi adresa
  cout << "d1 = " << d1 << endl;
  cout << "d2 = " << d2 << endl;
cout << "b  = " << b << endl;
}
```

1. Upcasting si downcasting (recapitulare)

Downcasting

```
class Shape {
public: virtual ~Shape() {};
};
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c;
// Upcast: normal and OK
  // More explicit but unnecessary:
  s = static_cast<Shape*>(&c);

 Circle* cp = 0;
  Square* sp = 0;
// Static Navigation of class hierarchies
requires extra type information:
if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // Static navigation is ONLY an efficiency
hack;
  // dynamic_cast is always safer. However:
  // Other* op = static_cast<Other*>(s);
  // Conveniently gives an error message,
while
  Other* op2 = (Other*)s;
  // does not
}
```

2. Controlul tipului în timpul rulării programului în C++

Facilitati C++ adaugate in cadrul polimorfismului la executie:

1) ***Run - Time Type Identification (RTTI)*** - permite identificarea tipului unui obiect in timpul executiei programului;

1) set aditional de 4 operatori de cast (***dynamic_cast, const_cast, reinterpret_cast,*** si ***static_cast***) - pentru o modalitate mai sigura de cast:

Unul dintre operatori, ***dynamic_cast***, este legat direct de mecanismul RTTI.

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

- nu se regaseste in limbajele nepolimorfice (expl. C), intrucat nu e nevoie de informatie la executie, pentru ca tipul fiecarui obiect este cunoscut la compilare (expl. in timpul scrierii);

- in limbajele polimorfice (expl. C++) pot aparea situatii in care tipul unui obiect nu este cunoscut pana la executia programului;

- C++ implementeaza polimorfismul prin mostenire, functii virtuale si pointeri catre clasa de baza care pot fi utilizati pentru a arata catre obiecte din clase derivate, deci nu se poate sti a-priori tipul obiectului catre care se pointeaza. ***Determinarea se face la executie, folosind RTTI.***

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

- ***typeid*** - pentru a obtine tipul obiectului;
- ***#include <typeinfo>***;
- uzual ***typeid(object)***;
- tipul obiectului: predefinit sau definit de utilizator;
- typeid - returneaza o referinta catre un obiect de tip **type_info** care descrie tipul obiectului;
- **typeid(null) genereaza exceptie: bad_typeid**

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

Clasa type_info

Membri publici:

- **bool operator==(const type_info &ob);**
- **bool operator!=(const type_info &ob);**
  - pentru doua obiecte se poate verifica daca au sau nu acelasi tip;

- **bool before(const type_info &ob);**
  - putem verifica daca un type_info precede un alt type_info:
    if(typeid(obiect1).before(typeid(obiect2)))
  - se foloseste in implementarea type_info ca si chei pentru o structura;

- **const char *name( );**
  - **sirul exact intors depinde de compilator** dar contine si tipul obiectului.

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

Exemplu cu tipuri predefinite

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main() {
    int a, b;     float c;    char *p;
      cout << "The type of a is: " << typeid(a).name() <<endl;
            cout << "The type of c is: " << typeid(c).name() << endl;
            cout << "The type of p is: " << typeid(p).name() << endl;
      if(typeid(a) == typeid(b))
       cout << "The types of i and j are the same\n";
      if(typeid(a) != typeid(c))
       cout << "The types of i and f are not the same\n";
}
```

// Pe compilatorul personal s-au afisat: i (pt int), f(pentru float) si Pc(pentru char*)

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

Exemplu cu tipuri definite de utilizator

```
#include <iostream>
#include <typeinfo>
using namespace std;
class myclass1{ … };
class myclass2{ … };
int main() {
    myclass1 ob1;
   myclass2 ob2;
      cout << "The type of ob1 is: " << typeid(ob1).name() <<endl;
           cout << "The type of ob2 is: " << typeid(ob2).name() << endl;
           if(typeid(ob1) != typeid(ob2)) cout << "ob1 and ob2 are of differing types\n";
}

// Pe compilatorul personal s-au afisat: 8myclass1 (pt ob1), 8myclass2(pentru ob2);
```

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

Cea mai importanta utilizare a *typeid* - tipuri polimorfice

```
class Baza {public: virtual void f () { } };// tip polimorfic
class Derivata1: public Baza { };
class Derivata2: public Baza { };
int main() {     Baza *p, b;    Derivata1 d1;    Derivata2 d2;
    p = &b;
    cout << "p is pointing to an object of type "<< typeid(*p).name() << endl;
    p = &d1;
    cout << "p is pointing to an object of type " << typeid(*p).name() << endl;
    p = &d2;
    cout << "p is pointing to an object of type " << typeid(*p).name() << endl;
    return 0;
}
```

// Pe compilatorul personal s-au afisat: 4Baza; 9Derivata1; 9Derivata2

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

Demonstrarea typeid cu referinte

```
class Baza {public: virtual void f () { } };// tip polimorfic
class Derivata1: public Baza { };
class Derivata2: public Baza { };

void WhatType(Baza &ob){
cout << "ob is referencing an object of type " << typeid(ob).name() << endl;}

int main() {    Baza b;    Derivata1 d1;    Derivata2 d2;
    WhatType(b);
    WhatType(d1);
    WhatType(d2);

    return 0;
}
```

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

Alta modalitate de utilizare ***typeid***:

typeid(type-name)

Expl: **cout << typeid(int).name();**
Expl de utilizare:

```
class Baza {public: virtual void f () { } };// tip polimorfic
class Derivata1: public Baza { };
class Derivata2: public Baza { };

void WhatType(Baza &ob)
{
    cout << "ob is referencing an object of type " << typeid(ob).name() << endl;
    if(typeid(ob) == typeid(Baza)) cout << "Baza.\n";
    if(typeid(ob) == typeid(Derivata1)) cout << "Derivata1.\n";
}
```

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

Un exemplu cu o functie, denumita "**factory**" care produce obiecte de diferite tipuri (in general, o astfel de functie se numeste "***object factory***" - ne vom mai intalni cu acest concept la Design Patterns.

```
class Baza {public: virtual void f () { } };// tip polimorfic
class Derivata1: public Baza { };
class Derivata2: public Baza { };

Baza *factory() {
   switch(rand()%2)   {
   case 0:
      return new Derivata1;
   case 1:
      return new Derivata2;
   }
   return 0;
}
```

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

```
class Baza {public: virtual void f () { } };// tip polimorfic
class Derivata1: public Baza { };
class Derivata2: public Baza { };
Baza *factory() { }
int main()
{
    Baza *b;
    int nr1 = 0, nr2 = 0;
    for(int i=0; i<10; i++)    {
        b = factory(); // generate an object
        cout << "Object is " << typeid(*b).name() << endl;
// count it
        if(typeid(*b) == typeid(Derivata1)) nr1++;
        if(typeid(*b) == typeid(Derivata2)) nr2++;    }
    cout<<nr1<<"\t"<<nr2;
    return 0;
}
```

2. Controlul tipului în timpul rulării programului în C++

Run - Time Type Identification (RTTI)

Nu functioneaza cu pointeri void, nu au informatie de tip

2. Controlul tipului în timpul rulării programului în C++

Operatorii de cast

C++ are 5 operatori de cast:

1) operatorul traditional mostenit din C;

2) dynamic_cast;

3) static_cast;

4) const_cast;

5) reinterpret_cast.

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast

- daca vrem sa schimbam tipul unui obiect la executie;
- **se verifica daca un downcast este posibil (si deci valid);**
- daca e valid, atunci se poate schimba tipul, altfel eroare.

Sintaxa:

dynamic_cast <target-type> (expr)

- target-type trebuie sa fie un pointer sau o referinta;

Dynamic_cast **schimba tipul unui pointer/referinte** intr-un alt tip **pointer/referinta**.

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast

Scop: cast pe tipuri polimorfice;

Exemplu:
class B{virtual …};
class D:B {... };

- un pointer D* poate fi transformat oricand intr-un pointer B* (pentru ca un pointer catre baza poate oricand retine adresa unei derivate);
- invers este necesar operatorul dynamic_cast.

In general, dynamic_cast reuseste daca pointerul (sau referinta) de transformat este un pointer (referinta) catre un obiect de tip target-type, sau derivat din aceasta, altfel, incercarea de cast esueaza (dynamic_cast se evalueaza cu null in cazul pointerilor si cu bad_cast exception in cazl referintelor.

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // base pointer points to Derived object

dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK

bp = &b_ob; // base pointer points to Base object

dp = dynamic_cast<Derived *> (bp); // error
```

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast

```
class Base { public:
    virtual void f()
    {
        cout << "Inside Base\n";
    }
};
class Derived : public Base {
public:
    void f()
    {
        cout << "Inside
Derived\n";
    }
};
```

```
int main() {
    Base *bp, b_ob;
    Derived *dp, d_ob;

    dp = dynamic_cast<Derived *> (&d_ob);
    if(dp) { cout << "from Derived * to Derived* OK.\n";
        dp->f();    }
    else cout << "Error\n";

    bp = dynamic_cast<Base *> (&d_ob);
    if(bp) { cout << "from Derived * to Base * OK.\n";
        bp->f();    }
    else cout << "Error\n";

    bp = dynamic_cast<Base *> (&b_ob);
    if(bp)    {  cout << "from Base * to Base * OK.\n";
        bp->f();    }
    else  cout << "Error\n";
```

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast

```
class Base { public:
    virtual void f()
    {
        cout << "Inside Base\n";
    }
};
class Derived : public Base {
public:
    void f()
    {
        cout << "Inside
Derived\n";
    }
};
```

```
/*    Base *bp, b_ob;
    Derived *dp, d_ob; */

    dp = dynamic_cast<Derived *> (&b_ob);
    if(dp)  cout << "Error\n";
    else
cout << "Cast from Base* to Derived* not OK.\n";

      bp = &d_ob; // bp points to Derived object
    dp = dynamic_cast<Derived *> (bp);
    if(dp)  {
        cout << "Casting bp to a Derived * OK\n" <<
            "because bp is really pointing\n" <<
            "to a Derived object.\n";
        dp->f();
    }
    else cout << "Error\n";
```

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast

```
class Base { public:
    virtual void f()
    {
        cout << "Inside Base\n";
    }
};
class Derived : public Base {
public:
    void f()
    {
        cout << "Inside
Derived\n";
    }
};
```

```
/* Base *bp, b_ob;
    Derived *dp, d_ob; */

     bp = &b_ob; // bp points to Base object
    dp = dynamic_cast<Derived *> (bp);
    if(dp)  cout << "Error";
    else    { cout << "Now casting bp to a Derived *\n
is not OK because bp is really \n pointing to a Base
object.\n";    }

    dp = &d_ob; // dp points to Derived object
    bp = dynamic_cast<Base *> (dp);
    if(bp)    { cout << "Casting dp to a Base * is OK.\n";
        bp->f();
    }
    else  cout << "Error\n";
    return 0;
}
```

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast Afisare:

Cast from Derived * to Derived * OK.
Inside Derived
Cast from Derived * to Base * OK.
Inside Derived
Cast from Base * to Base * OK.
Inside Base
Cast from Base * to Derived * not OK.

Casting bp to a Derived * OK
because bp is really pointing
to a Derived object.
Inside Derived

Now casting bp to a Derived *
is not OK because bp is really
pointing to a Base object.
Casting dp to a Base * is OK.
Inside Derived

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast inlocuieste ***typeid***

Fie Base si Derived 2 clase polimorfice.

```
Base *bp;
Derived *dp;

// ... if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

- cast obisnuit;
- if verifica validitatea operatiei de cast

Cel mai indicat:

```
dp = dynamic_cast <Derived *> (bp);
```

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast inlocuieste ***typeid***

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base   {
      public:
          virtual void f() { }
      };

class Derived : public Base {
      public:
      void derivedOnly()     {
         cout << "Is a Derived Object.\n";  }
      };
      int main()
      {
          Base *bp, b_ob;
          Derived *dp, d_ob;
```

```
// use typeid

bp = &b_ob;
 if(typeid(*bp) == typeid(Derived))    {
                dp = (Derived *) bp;
                dp->derivedOnly();    }
else   cout << "Cast from Base to
Derived failed.\n";

bp = &d_ob;
if(typeid(*bp) == typeid(Derived))    {
                dp = (Derived *) bp;
                dp->derivedOnly();    }
 else   cout << "Error, cast should
work!";
      return 0;
      }
```

2. Controlul tipului în timpul rulării programului în C++

Dynamic_cast inlocuieste ***typeid***

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base   {
      public:
          virtual void f() { }
      };

class Derived : public Base {
      public:
      void derivedOnly()     {
         cout << "Is a Derived Object.\n";  }
      };
      int main()
      {
          Base *bp, b_ob;
          Derived *dp, d_ob;
```

```
// use dynamic_cast

bp = &b_ob;
dp = dynamic_cast<Derived *> (bp);

if(dp) dp->derivedOnly();
 else  cout << "Cast from Base to
Derived failed.\n";

 bp = &d_ob;
 dp = dynamic_cast<Derived *> (bp);
if(dp) dp->derivedOnly();
else  cout << "Error, cast should
work!\n";
  return 0;
      }
```

2. Controlul tipului în timpul rulării programului în C++

Static_cast

- este un substitut pentru operatorul de cast clasic;
- lucreaza pe tipuri nepolimorfice;
- poate fi folosit pentru orice conversie standard;
- ne se fac verificari la executie (run-time);

Sintaxa: static_cast <type> (expr)

Expl:

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for(i=0; i<10; i++)
        cout << static_cast<double> (i) / 3 << " ";
    return 0;
}
```

2. Controlul tipului în timpul rulării programului în C++

Const_cast

- folosit pentru a rescrie, explicit, proprietatea de const sau volatile intr-un cast (elimina proprietatea de a fi constant);

- tipul destinatie trebuie sa fie acelasi cu tipul sursa, cu exceptia atributelor const / volatile.

Sintaxa: const_cast <type> (expr)

2. Controlul tipului în timpul rulării programului în C++

Const_cast Exemplu - pointer

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

void sqrval(const int *val) {
    int *p;
// cast away const-ness.
    p = const_cast<int *> (val);
    *p = *val **val; // now, modify object through v
}
int main()
{
    int x = 10;
    cout << "x before call: " << x << endl;
    sqrval(&x);
    cout << "x after call: " << x << endl;
    return 0;
}
```

2. Controlul tipului în timpul rulării programului în C++

Const_cast Exemplu - referinta

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

void sqrval(const int &val) {
// cast away const on val
const_cast<int &> (val) = val * val;
}

int main()
{
int x = 10;
cout << "x before call: " << x << endl;
sqrval(x);
cout << "x after call: " << x << endl;
return 0;
}
```

2. Controlul tipului în timpul rulării programului în C++

Reinterpret_cast

- converteste un tip intr-un alt tip fundamental diferit;

Sintaxa: reinterpret_cast <type> (expr)
Expl:

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

int main()
{
    int* p = new int(65);
    char* ch = reinterpret_cast<char*>(p);
    cout << *p << endl;
    cout << *ch << endl;
    cout << p << endl;
    cout << ch << endl;
    return 0;
}
```

3. Şabloane (Templates) în C++

Nu apare in C++ initial.

Un template reprezintă o altă tehnică de reutilizare de cod din afara claselor (mostenirea si compunerea de obiecte - reutilizare cod din clase).

Reprezintă o altă formă de polimorfism.

Defineşte o clasă de funcţii sau clase care rezolvă o anumită problemă lucrând cu tipuri de date nespecificate, generice.

Putem avea template-uri (sabloane) si pentru functii si pentru clase.

Expl:

- implementarea unui set de funcţii care permit obţinerea elementului maxim dintr-o colecţie (ex. vector) de elemente întregi, float, double, Complex, string, Student, etc…
- clasic: stiva / coada - implementarea notiunii prin templates (sabloane) si apoi refolosirea codului pe elemente întregi, float, double, Complex, string, Student, etc…

3. Şabloane (Templates) în C++

Functii generice

Multi algoritmi sunt generici (nu conteaza pe ce tip de date opereaza).

Inlaturam bug-uri si marim viteza implementarii daca reusim sa refolosim aceeasi implementare pentru un algoritm care trebuie folosit cu mai mute tipuri de date.

O singura implementare, mai multe folosiri.
O functie generica face auto overload (pentru diverse tipuri de date).

Sintaxa:

```
template <class Ttype> tip_returnat nume_functie(lista_de_argumente) {
    // corpul functiei
}
```

Ttype este un nume pentru tipul de date folosit (inca indecis), compilatorul il va inlocui cu tipul de date folosit.

3. Şabloane (Templates) în C++

Functii generice

```
template <class Ttype> // e ok si template <typename Ttype>
Ttype maxim (Ttype V[ ], int n)  {
    Ttype max = V[0];
        for (int i = 1; i < n; i++) {
            if (max < V[i])
            max = V[i];
    }
return max;
}
int main ()
{
int VI[] = {1, 5, 3, 7, 3};
float VF[] = {(float)1.1, (float)5.1, (float)3.1, (float)4.1};
cout << "maxim (VI): " << maxim<int> (VI, sizeof (VI)/sizeof (int))<< endl;
cout << "maxim (VF): " << maxim<float> (VF, sizeof (VF)/ sizeof (double)) << endl;
}
```

3. Şabloane (Templates) în C++

Functii generice

Specificatia de template trebuie sa fie imediat inaintea definitiei functiei.

```
template <class Ttype>
int i; // this is an error
void swapargs(Ttype &a, Ttype &b)
{
    Ttype temp;
    temp = a;
    a = b;
    b = temp;
}
```

3. Şabloane (Templates) în C++

Functii generice

Putem avea functii cu mai mult de un tip generic.

- compilatorul creeaza atatea functii cu acelasi nume cate sunt necesare (d.p.d.v. al parametrilor folositi).

```
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
    cout << x << ' ' << y << '\n';
}

int main()
{
      myfunc(10, "I like C++");
      myfunc(98.6, 19L);
      return 0;
}
```

3. Şabloane (Templates) în C++

Functii generice

Overload pe sabloane

Sablon: overload implicit

Putem face overload explicit

Se numeste "**specializare explicita**"

In cazul specializarii explicite versiunea sablonului care s-ar fi format in cazul tipului de parametrii respectivi nu se mai creeaza (se foloseste versiunea explicita)

3. Şabloane (Templates) în C++

Functii generice

Overload pe sabloane - Specializare explicita

```
template <class T> T maxim( T  a, T  b)
{  cout<<"template"<<endl;
    if (a>b) return a; // operatorul < trebuie sa fie definit pentru tipul T
 return b;
}

template < > char * maxim ( char* a, char* b)
{cout<<"supraincarcare neconst"<<endl;
 if (strcmp(a,b)>0) return a;
 return b;
}
```

3. Şabloane (Templates) în C++

Functii generice

Overload pe sabloane - Specializare explicita

```
template <class T> T maxim( T  a, T  b)
{  cout<<"template"<<endl;
    if (a>b) return a; // operatorul < trebuie sa fie definit pentru tipul T
 return b; }

template <> const char * maxim(const char* a,const char* b)
{  cout<<"supraincarcare const"<<endl;
 if (strcmp(a,b)>0) return a;
 return b; }

 template <> char * maxim ( char* a, char* b)
{  cout<<"supraincarcare neconst"<<endl;
 if (strcmp(a,b)>0) return a;
 return b; }
```

3. Şabloane (Templates) în C++

Functii generice

Overload pe sabloane - Specializare explicita

```
// pot exista ambele variante
//daca nu exista template<> const char* -pt "ab" se alege sablonul general
//daca nu exista template<> char* -pt v1 se alege sablonul general
/* NU FACE CONVERSIA NICI (char *) --> (const char *) si nici  (const char *) -
-> (char *) */

int main(int argc, char *argv[])
{
 char v1[10]="abc",v2[10]="bcd";
  cout<<maxim("ab","bc")<<endl;
  cout<<maxim(v1,v2)<<endl;
  cout<<maxim<char *>(v1,"ab")<<endl;
return 0;
}
```

3. Şabloane (Templates) în C++

Functii generice

Overload pe sabloane

Diferita de specializare explicita

Similar cu overload pe functii (doar ca acum sunt functii generice)

Simplu: la fel ca la functiile normale

3. Şabloane (Templates) în C++

Functii generice

Overload pe sabloane

```
// First version of f() template.
template <class X> void f(X a) {
      cout << "Inside f(X a)\n";
}

// Second version of f() template.
template <class X, class Y> void f(X a, Y b) {
      cout << "Inside f(X a, Y b)\n";
}

int main() {
       f(10); // calls f(X)
       f(10, 20); // calls f(X, Y)
       return 0;  }
```

3. Şabloane (Templates) în C++

Functii generice

Overload pe sabloane - ce functie se apeleaza (ordinea de alegere)

pas 1 potrivire FARA CONVERSIE

- prioritate varianta non-template,
- apoi template fara parametrii,
- apoi template cu 1 parametru ,
- apoi template cu mai multi parametrii

pas 2 daca nu exista potrivire exacta

- conversie DOAR la varianta non-template

3. Şabloane (Templates) în C++

Functii generice

Overload pe sabloane - ce functie se apeleaza (ordinea de alegere)

```
template <class T> void f(T t){ … }
template <> void f(float x){ … }
void f(float x){ .. }

int main()
{ f(1); // T = int ('a');
  f(2.5); // T=double;
float x;
  f(x);  //non-template float , prioritar fata de template<>, prioritar fata de template T

f<>(x); // template<> prioritar fata de template general cu T=float
  f<float>(x); // template<> prioritar fata de template general cu T=float
```

3. Şabloane (Templates) în C++

Functii generice

Utilizarea parametrilor standard intr-un template

```
const int TABWIDTH = 8;
// Display data at specified tab position.
template<class X> void tabOut(X data, int tab) {
   for(; tab; tab--)
   for(int i=0; i<TABWIDTH; i++) cout << ' ';
   cout << data << "\n";
}

int main(){
   tabOut("This is a test", 0);
   tabOut(100, 1);
   tabOut('X', 2);
   tabOut(10/3, 3);
   return 0;
}
```

3. Şabloane (Templates) în C++

Clase generice

Sabloane pentru clase nu pentru functii.

Clasa contine toti algoritmii necesari sa lucreze pe un anumit tip de date.

Din nou algoritmii pot fi generalizati, sabloane.

Specificam tipul de date pe care lucram cand obiectele din clasa respectiva sunt create.

Functiile membru ale unei clase generice sunt si ele generice (in mod automat).

Nu e necesar sa le specificam cu template.

3. Şabloane (Templates) în C++

Clase generice

Cozi, stive, liste inlantuite, arbori de sortare

```
template <class Ttype> class class-name {
...
}

class-name <type> ob;
```

Ttype este tipul de date parametrizat.

Ttype este precizat cand clasa e instantiata.

Putem avea mai multe tipuri (separate prin virgula)

3. Şabloane (Templates) în C++

Clase generice

```
template <class T>
class vector  {
    int dim;
    T v[100];
public:
    void citire();
    void afisare();
};

template <class T>
void vector<T>::citire() {
    cin>>dim;
    for(int i = 0; i<dim; i++)
        cin>>v[i];
}
```

```
template <class T>
void vector<T>::afisare()
{
    for(int i = 0; i<dim; i++)
        cout<<v[i]<<" ";
    cout<<"\n";
}

int main()
{
    vector<int> ob1;
    ob1.citire();
    ob1.afisare();
    vector<float> ob2;
    ob2.citire();
    ob2.afisare();*/
return 0;
}
```

3. Şabloane (Templates) în C++

Clase generice

Mai multe tipuri de date generice intr-o clasa

```
template <class Type1, class Type2> class myclass {
   Type1 i;
   Type2 j;
public:
   myclass(Type1 a, Type2 b) { i = a; j = b; }
   void show() { cout << i << ' ' << j << '\n'; }
};

int main() {
   myclass<int, double> ob1(10, 0.23);
   myclass<char, char *> ob2('X', "Templates add power.");
   ob1.show(); // show int, double
   ob2.show(); // show char, char *
   return 0;
}
```

3. Şabloane (Templates) în C++

Clase generice

Sabloanele se folosesc cu operatorii suprascrisi

```
class student {
    string nume;
    float varsta;
} x,y;

template <class T>
void maxim(T a, T b) {
    if (a > b)  cout<<"Primul este mai mare\n";
    else  cout<<"Al doilea este mai mare\n";
}

int main()
{  int a = 3, b = 7;
    maxim(a,b); // ok
    maxim(x,y); // operatorul > ar trebui definit in clasa student
}
```

3. Şabloane (Templates) în C++

Clase generice

Sabloanele se folosesc cu operatorii suprascrisi

Se pot specifica si argumente valori in definirea claselor generalizate.

Dupa “template” dam tipurile parametrizate cat si “parametri normali” (ca la functii).

Acesti “param. normali” pot fi int, pointeri sau referinte; trebuiesc sa fie cunoscuti la compilare: tratati ca si constante.

template <class tip1, class tip2, int i>

3. Şabloane (Templates) în C++

Clase generice

Specializari explicite pentru clase

La fel ca la sabloanele pentru functii

Se foloseste template<>

3. Şabloane (Templates) în C++

Clase generice

Specializare de clasa

```
template <class T>  // sau template <typename T>
class Nume {
    T x;
    public:
    void set_x(T a){x = a;}
    void afis(){cout<<x;}
};

template <>
class Nume<unsigned>  {
    unsigned x;
    public:
    void set_x(unsigned a){x = a;}
    void afis(){cout<<"\nUnsigned "<<x;}
};
```

```
int main()
{

    Nume<int> m;
    m.set_x(7);
    m.afis();

    Nume<unsigned> n;
    n.set_x(100);
    n.afis();
    return 0;
}
```

3. Şabloane (Templates) în C++

Clase generice

Argumente default si sabloane

Putem avea valori default pentru tipurile parametrizate.

```
template <class X=int> class myclass { //...
```

Daca instantiem myclass fara sa precizam un tip de date atunci int este tipul de date folosit pentru sablon.

Este posibil sa avem valori default si pentru argumentele valori (nu tipuri).

3. Şabloane (Templates) în C++

Clase generice

Argumente default si sabloane

```
template <class AType=int, int size=10>
class atype {
    AType a[size]; // size of array is passed in size
public:
    atype();
};

template <class AType, int size>
atype<AType,size>::atype() {       for(int i=0; i<size; i++) a[i] = i;     }

int main() {
atype<int, 100> intarray; // integer array, size 100
atype<double> doublearray; // double array, default size
atype<> defarray; // default to int array of size 10
return 0;
}
```

3. Şabloane (Templates) în C++

Typeid si clasele template

Tipul unui obiect care este o instanta a unei clase template este determinat, in parte, de tipul datelor utilizate in cadrul datelor generice cand obiectul este instantiat.
2 instante de tipuri diferite au fost create cu date diferite.

```
template <class T> class myclass
{
   T a;
public:
   myclass(T i)
   {
      a = i;
   }
// ...
};
```

3. Şabloane (Templates) în C++

Typeid si clasele template

template <class T> class myclass{ ... };

```
int main() {
    myclass<int> o1(10), o2(9);
    myclass<double> o3(7.2);

    cout << "Type of o1 is " << typeid(o1).name() << endl;
    cout << "Type of o2 is " << typeid(o2).name() << endl;
    cout << "Type of o3 is " << typeid(o3).name() << endl;

    if(typeid(o1) == typeid(o2))  cout << "o1 and o2 are the same type\n";
    if(typeid(o1) == typeid(o3))  cout << "Error\n";
    else  cout << "o1 and o3 are different types\n";

    return 0;
}
```

3. Şabloane (Templates) în C++

Typeid si clasele template

//Afisare compilator personal:

Type of o1 is 7myclassIiE

Type of o2 is 7myclassIiE

Type of o3 is 7myclassIdE

o1 and o2 are the same type
o1 and o3 are different types

3. Şabloane (Templates) în C++

dynamic_cast si clasele template

Tipul unui obiect care este o instanta a unei clase template este determinat, in parte, de tipul datelor utilizate in cadrul datelor generice cand obiectul este instantiat.

2 instante de tipuri diferite au fost create cu date diferite.

template <class T> class myclass{ ... };

myclass<int> si myclass < double> sunt 2 instante diferite.

Nu se poate folosi dinamic_cast pentru a schimba tipul unui pointer dintr-o instanta intr-un pointer dintr-o instanta diferita.

3. Şabloane (Templates) în C++

dynamic_cast si clasele template

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

template <class T>
class Num { protected:
    T val;
public:  Num(T x)  { val = x;  }
    virtual T getval( )  { return val;   }
};

template <class T>
class SqrNum : public Num<T>  {
public:
    SqrNum(T x) : Num<T>(x) { }
    T getval( )  {  return val * val;    }
};
```

```
int main()
{
    Num<int> *bp, nob(2);
    SqrNum<int> *dp, sob(3);
    Num<double> dob(3.3);

bp = dynamic_cast<Num<int> *> (&ob);
    if(bp)
    {
        cout << "Cast from SqrNum<int>* to
Num<int>* OK.\n";
        cout << "Value is " << bp->getval() <<
endl;
    }
    else
        cout << "Error\n";
return 0;
}
```

3. Şabloane (Templates) în C++

dynamic_cast si clasele template

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

template <class T>
class Num { protected:
    T val;
public:  Num(T x)  { val = x;  }
    virtual T getval( )  { return val;   }
};

template <class T>
class SqrNum : public Num<T>  {
public:
    SqrNum(T x) : Num<T>(x) { }
    T getval( )  {  return val * val;    }
};
```

```
int main()
{
    Num<int> *bp, nob(2);
    SqrNum<int> *dp, sob(3);
    Num<double> dob(3.3);

dp = dynamic_cast<SqrNum<int> *> (&nob);
    if(dp)  cout << "Error\n";
    else     {
        cout << "Cast from Num<int>* to
SqrNum<int>* not OK.\n";
        cout << "Can't cast a pointer to a
base object into\n";
        cout << "a pointer to a derived
object.\n";
    }
return 0;
}
```

3. Şabloane (Templates) în C++

dynamic_cast si clasele template

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

template <class T>
class Num { protected:
    T val;
public:  Num(T x)  { val = x;  }
    virtual T getval( )  { return val;   }
};

template <class T>
class SqrNum : public Num<T>  {
public:
    SqrNum(T x) : Num<T>(x) { }
    T getval( )  {  return val * val;    }
};
```

```
int main()
{
    Num<int> *bp, nob(2);
    SqrNum<int> *dp, sob(3);
    Num<double> dob(3.3);

bp = dynamic_cast<Num<int> *> (&dob);
    if(bp)
        cout << "Error\n";
    else
        cout << "Can't cast from
Num<double>* to Num<int>*.\n";
    cout << "These are two different
types.\n";

    return 0;
}
```

Perspective

Cursul 10:

Parametrizarea datelor. Şabloane în C++. Clase generice (continuare)

- Funcţii şi clase Template: Definiţii, Exemple, Implementare.
- Clase Template derivate.
- Specializare.