Cursul 6 - CLASE în C++, continuare

- Supraîncărcare operatori
- Constructor de conversie vs.Operator de conversie
- Clase parametrizate
  - Declarație template
  - Definirea membrilor
  - Instanțiere(Specializare), Specializare utilizator
  - Friend în template
  - Membri statici în template
- Relația de derivare
  - Sintaxa
  - Tipuri de derivare
  - Constructori, Destructori
  - Conversii standard
  - Copiere
  - Conflicte

CLASE: Supraîncărcare operatori

- Operatorii **=, [], (), ->** trebuie implementați ca funcții membru nestatice; asta asigură că primul operand este "l-value"
- Precedența şi asociativitatea este aceeaşi, nu pot fi schimbate
- Semantica este definită de programator; este de preferat ca aceasta să fie după modelul tipurilor fundamentale
- Operatorii **=, &** şi virgulă au înțeles predefinit pentru obiectele unei clase. Pot fi puşi în **private** pentru a anula acest lucru, pot fi redefiniți
- Toți operatorii supraîncărcați, exceptând **operator=**, sunt moşteniți de clasele derivate
- Dacă sunt mai multe definiții pentru operatori (sunt supraîncărcați), la apel sistemul alege, după regulile obişnuite de la funcții, varianta corectă

CLASE: Supraîncărcare operatori - Exemple

```
class X{
        public:
                void operator+(int);
                void operator+(double);
                X(int);
        };
        void operator+(X, X);
        void operator+(double, X);
        void f(X a){
                a+99;    // a.operator+(99)
                a+5.55;  // a.operator+(5.55)
                33 + a;  // ::operator+(X(33), a)
                1.1 + a; // ::operator+(1.1, a)
        }
class Y{
        public:
                Y* operator&(); // adresa, operator unar
                Y operator&(Y); // operatorul logic &, binar
                Y operator++(int); // ++ postfix, unar
                Y operator&(Y, Y); // Eroare, prea multe argumente
                Y operator/(); // Eroare, prea putine argumente
                //
};
```

CLASE: Supraîncărcare operatori

- Minimizarea numărului funcțiilor ce au acces direct la reprezentare:
  - Operatorii ce modifică valoarea primului argument, ca membri (ex. +=)
  - Operatorii ce produc noi valori, ca funcții externe (ex. +, −,)

```
class complex{
    public:
    complex& operator+=(const complex a);
    //…
    private:
    double re, im;
};
inline complex& complex:: operator+=(const complex a){
    re += a.re; im += a.im; return *this;
}
complex operator+(complex a, complex b){
    complex s = a;
    return s+= b;
}
```

CLASE: Constructor de conversie

- Constructorul de conversie defineşte o conversie(ce se aplică implicit) de la un tip (de baza) la un tip utilizator

X::X(*tip_de_baza* m)

```
punct(int i) : x(i), y(0) {} // int -> punct
data(int d):zi(d), luna(luna_curenta()), an(anul_curent())
    {} // int -> data
complex(double r) : re(r), im(0){} // double -> complex

void f(){
    complex z = 2; // z = complex(2)
    3 + z; // complex(3) + z;
    z += 3; // z += complex(3);
    3.operator+=(z); // eroare
    3 += z; // eroare
}
```

CLASE: Constructor de conversie

- Constructorul de conversie poate suplini definițiile unor operatori; pentru clasa complex nu-i necesar a defini:

```
complex operator+(double, complex);
complex operator+(complex, double);
```

- Un constructor de conversie nu poate defini:
  - O conversie implicită de la un tip utilizator la un tip de bază
  - O conversie de la o clasă nouă la o clasă definită anterior, fără a modifica declarațiile vechii clase
  - Soluția: Operatorul de conversie

CLASE: Operatorul de conversie

- este o funcție membru:

```
X::operator T()const;
```

  care defineşte o conversie de la tipul X la tipul T
- Exemplu:

```
class Clock{
public:
   Clock(int = 12, int = 0, int = 0);
   Clock tic_tac();
   friend ostream& operator<<(ostream&, Clock&);
   Clock operator++();
   Clock operator++(int);
   operator int(); // Conversie Clock --> int
                   // Ora 8:22 AM devine 822
                   // Ora 8.22 PM devine 2022
private:
   int ora;
   int min;
   int ap;  // 0 pentru AM, 1 pentru PM
};
```

CLASE: Conversie

- Ambiguitate constructor de conversie/operator de conversie:

```
class Apple {
public:
  operator Orange() const; // Apple -> Orange
};
class Orange {
public:
  Orange(Apple); // Apple -> Orange
};
void f(Orange) {}
int main() {
   Apple a;
   f(a);
}
//error C2664:'f' : cannot convert parameter 1 from
//'class Apple' to 'class Orange'
```

CLASE: Conversie

- Ambiguitate operatori conversie/supraîncărcare funcții:

```
class Orange {};
class Pear {};
class Apple {
public:
  operator Orange() const; // Apple -> Orange
  operator Pear() const;   // Apple -> Pear
};
// Supraincarcare eat():
void eat(Orange);
void eat(Pear);

int main() {
    Apple c;
    eat(c); // Error: Apple -> Orange or Apple -> Pear ???
}
```

Clase parametrizate

- Programare generică : programare ce utilizează tipurile ca parametri
- În C++ - mecanismul template: clase template, funcții template
- Programatorul scrie o singură definiție template iar C++, pe baza parametrilor, generează specializări care sunt compilate cu restul programului sursă
- O funcție template poate fi supraîncărcată cu:
  - Funcții template cu acelaşi nume dar parametri template diferiți
  - Funcții non – template cu acelaşi nume dar cu alți parametri
- Clasele template se mai numesc tipuri parametrizate

Clase parametrizate

- Declarație **`template`**:

  `template` **<** *lista_argumente_template* **>** *declaratie*

  lista_argumente_template :: *argument_template*|
  lista_argumente_template, *argument_template*

  argument_template :: *tip_argument*|*declaratie_argument*

  tip_argument :: **`class`** *identificator*|**`typename`** *identificator*

  declaratie_argument::<*tip*> *identificator*

  declaratie:: *declarația unei clase sau funcții*

Clase parametrizate

```
template< class T, int i > class MyStack{};

template< class T1, class T2 > class X{};
template< typename T1, typename T2 > class X{};

template<class T> class allocator {};
template<class T1, typename T2 = allocator<T1> >
class stack { };
stack<int> MyStack; // al doilea argument este implicit

class Y {...};
template<class T, T* pT> class X1 {...};
template<class T1, class T2 = T1> class X2 {...};
Y aY;
X1<Y, &aY> x1;
X2<int> x2;
```

Clase parametrizate

- Funcțiile membru ale unei clase **template** sunt funcții template parametrizate cu parametrii clasei template respective

- Definirea membrilor unei clase **template:**
  - Definiție **inline**, la specificarea clasei – nu este specificat explicit **template**
  - Definiție în afara specificării clasei – trebuie declarată explicit **template:**

template <*lista_argumente_template* >
nume_clasa_template<*argum*>::*nume_functie_membru* **(***parametri***)**
{
    //Corp functie
}

- Compilatorul nu alocă memorie la declararea unui **template**

Clase parametrizate

- *Instanțiere* **`template`**: procesul de generare a unei declarații de clasă (funcție) de la o clasă(funcție) **`template`** cu argumente corespunzătoare

```
template class MyStack<class T,int n>{…};
template class MyStack<int, 6>;
template MyStack<int, 6>::MyStack();

template<class T> void f(T) {...}
template void f<int> (int);
template void f(char);
```

Clase parametrizate – Exemplul 1

```
template <class Elt>
class Cell
{
public:
        Cell();
        Cell(const Cell&);  // constructor de copiere
        ~Cell();
        Elt getVal() const;
        void setVal(Elt);
        Cell& operator =(const Cell&);
private:
        Elt* val;
};
template <class Elt>
Cell<Elt>::Cell()
{
        val = new Elt;
}
```

Clase parametrizate - Exemplu 1

```
template <class Elt>
Cell<Elt>::Cell(const Cell<Elt>& c)
{
        val = new Elt;
        *val = *(c.val);
}

template <class Elt>
Cell<Elt>::~Cell()
{
        delete val;
}
template <class Elt>
Cell<Elt>& Cell<Elt>::operator =(const Cell<Elt>& c)
{
        *val = *(c.val);
        return *this;
}
```

Clase parametrizate - Exemplu 1

```
#include "cell.h"

template <class Elt>
class Array
{
        public:
                Array(int = 1);
                Array(const Aray&);
                ~Array();
                Elt get(int);
                void set(int, Elt);
                Array& operator =(const Array&);
                Cell<Elt>& operator [](int)
                const Cell<Elt>& operator [](int) const;
        private:
                Cell<Elt> *arr;
                int nOfComp;
};
```

Clase parametrizate - Exemplu 1

```
template <class Elt>
Array<Elt>::Array(int nMax)
{
        arr = new Cell<Elt>[nMax];
        if(arr == NULL) throw "Insuf. memory.";
        nOfComp = nMax;
};
template <class Elt>
Array<Elt>::~Array()
{
        delete[] arr;
};

template <class Elt>
Elt Array<Elt>::get(int i)
{
        if((i<0)||(i>=nOfComp)) throw "Index out of range.";
        return arr[i].getVal();
};
```

Clase parametrizate - Exemplu 1

- Template: Supraîncărcare operatori

```
template <class Elt>
Cell<Elt>& Array<Elt>::operator [](int i)
{
        if((i<0)||(i>=nOfComp)) throw "Index out of range.";
        return arr[i];
}


// pentru a sorta tablouri de celule de int-uri
// trebuie definit operatorul de comparare:
bool operator >(const Cell<int>& x, const Cell<int>& y)
{
        return x.getVal() > y.getVal();
}
```

Clase parametrizate - Exemplu 1

```
template <class T>
void insert_sort(Array<T>& a, int n)
{
  int i,j;
  Cell<T> temp;
  for(i=1;i<n;i++) {
        temp = a[i];
    j = i - 1 ;
    while((j >= 0) && (a[j] > temp)) {
      a.set(j+1, a[j].getVal());
      j--;
    }
        if (i != (j-1))
                a.set(j+1, temp.getVal());
  }
}

typedef Cell<int> Int;
typedef Cell<char> Char;
```

Clase parametrizate - Exemplu 2

```
template <class T>
class Vector
{
public:
        Vector(int=0);
        T& operator [](int);
        const T& operator [](int) const;
        // …
private:
        T* tab;
        int n;
};
template <class T>
class Matrice
{
public:
        Matrice(int=0, int=0);
        Vector<T>& operator [](int);
        const Vector<T>& operator [](int) const;
private:
        Vector<T>* tabv;
        int m, n;
};
```

Clase parametrizate - Exemplu 2

```
template <class T>
T& Vector<T>::operator [](int i)
{
        cout << "Vector<T>::operator [](int i)" << endl;
        return tab[i];
}
template <class T>
const Vector<T>& Matrice<T>::operator [](int i) const
{
        cout << "Matrice<T>::operator [](int i) const" << endl;
        if (i < 0 || i >= m) throw "index out of range";
        return tabv[i];
}
        Vector<int> v(5);
        v[3] = 3; // apel varianta nonconst
        const Vector<int> vv(5);
        vv[4] = 4;// apel varianta const

        Matrice<double> m(3,5);
        m[1][2] = 5.0;
        const Matrice<double> mm(3,5);
        mm[2][3] = 7.0;
```

Clase parametrizate

- O versiune a unui template pentru un argument template particular este numită o *specializare*
- O definiție alternativă pentru o clasă(funcție) template (de ex. pentru a funcționa când argumentul template este pointer) este numită *specializare definită de utilizator*

Clase parametrizate - Specializări

```
template <class T>
class Vector
{
public:
   Vector(int=0);
   T& operator [](int);
   const T& operator [](int) const;
private:
   T* tab;
   int n;
};
```

- Specializări:

```
Vector<int> vi;
Vector<Punct*> vpp;
Vector<string> vs;
Vector<char*> vpc;
```

Clase parametrizate - Specializări

- Specializare a clasei **`Vector<T>`** pentru pointeri la **`void`**:

```
template<> class Vector<void*>{
// specializare fara parametri template
   void** p; // …
};
```

- Specializare a clasei **`Vector<T>`** pentru pointeri la **`T`**:

```
template<class T> class Vector<T*>{
// specializare cu parametri template
//…
};
```

Clase parametrizate - Specializări

```
template <class T>
void swap(T& x, T& y){
   T t = x; x = y; y = t;
}
```

- Specializare pentru **Vector<T>** :

```
template <class T>
void swap(Vector<T>& a, Vector<T>& b){
   a.swap(b);
}
```

- În clasa **Vector<T>**, metoda:

```
template <class T>
void Vector<T>:: swap(Vector<T>& a){
   swap(v, a.v);
   swap(sz, a.sz);
}
```

Clase parametrizate

- Două instanțieri ale unei clase template sunt echivalente dacă parametrii template ce reprezintă tipuri sunt aceeaşi iar ceilalți au aceleaşi valori

```
MyString<char> s1;
MyString<unsigned char> s2;
typedef unsigned char Uchar;
MyString <Uchar> s3;
```

Clase parametrizate - friend

- Funcții(clase) **`friend`** în clase **`template`**:
  - Dacă funcția **`friend`** accesează un parametru **`template`**, aceasta trebuie să fie **`template`**. În acest caz o instanțiere este **`friend`** doar pentru instanțierile clasei **`template`** cu aceiaşi parametri (**`friend`** "legat").
  - Prieteni template "nelegați" - are alți parametri template
  - Dacă funcția **`friend`** nu accesează parametri **`template`** , este **`friend`** pentru toate instanțierile clasei

Clase parametrizate - friend

```
template <class T> class X
{
   //..
   friend void f1();
   // f1 friend pentru toate specializarile
   friend void f2(X<T>& );
   // f2(X<float>&) friend pentru X<float>
   friend class Y;
   friend class Z<T>;
   friend void A::f3();
   friend void C<T> :: f4(X<double>&);
   //…
};
```

Clase parametrizate

- Membri statici în template –uri
  - Fiecare specializare a unei clase template are copia proprie a mebrilor static, atât date cât şi funcții
  - Datele statice trebuiesc inițializate, "global" sau pe specializări

```
template<class T, int n>
class ClasaN{
        static const int cod;
        //
};
template<class T, int n>
const int ClasaN<T, n>::cod = n;
template<>
const int ClasaN<int, 10>::cod =50
```

Relația de moştenire – Clase derivate

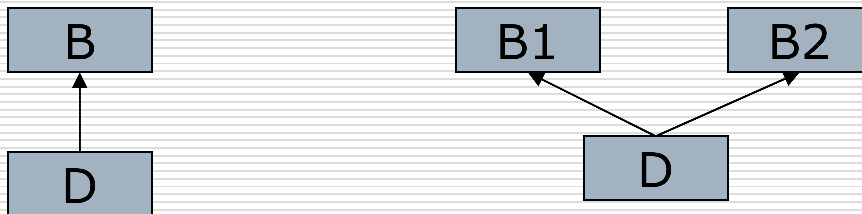
- Structurarea unui concept: definirea unei ierarhii de tipuri şi a unei relații de ordine (parțială) pe aceste tipuri.

- Tipul D este un subtip al (o specializare a) tipului B: toate obiectele de tip D sunt de tip B. Specializarea se poate face prin:
  - Restricționarea domeniului de valori a obiectelor de tip B
  - Adăugarea de noi operații la cele definite pe tipul B
  - În anume condiții, printr-o nouă definire a membrilor

Relația de moştenire – Clase derivate

- Reutilizarea codului existent: o clasă reutilizează structura de date şi codul definit pentru o clasă existentă. Acest lucru se face prin:
  - Îmbogățirea clasei prin adăugarea de noi membri
  - Redefinirea unor membri
  - Restricționarea domeniului de vizibilitate pentru anumiți membri; în acest caz nu mai este vorba de o ierarhie de tipuri ci de o ierarhie de clase; stiva este o lista în care inserția (extragerea) se face doar în top

- În C++ : clasă de bază (superclasă), clasă derivată (subclasă)
- Moştenire simplă, moştenire multiplă



Clase derivate

- Sintaxa:
  - Moştenire simplă:

class *ClsDer*:*tip_moşt ClsBaza* **{};**

  - Moştenire multiplă:

class *ClsDer* :*tip_moşt ClB1*, *tip_moşt ClB2*,…
{ };

tip_moşt :: **public|protected|private**

- Nivele de protecție(acces) a membrilor unei clase:
  - **public**: cunoscut de oricine
  - **protected**: cunoscut de clasa proprietară, prieteni şi de clasele derivate
  - **private**: cunoscut de clasa proprietară şi de prieteni

Clase derivate

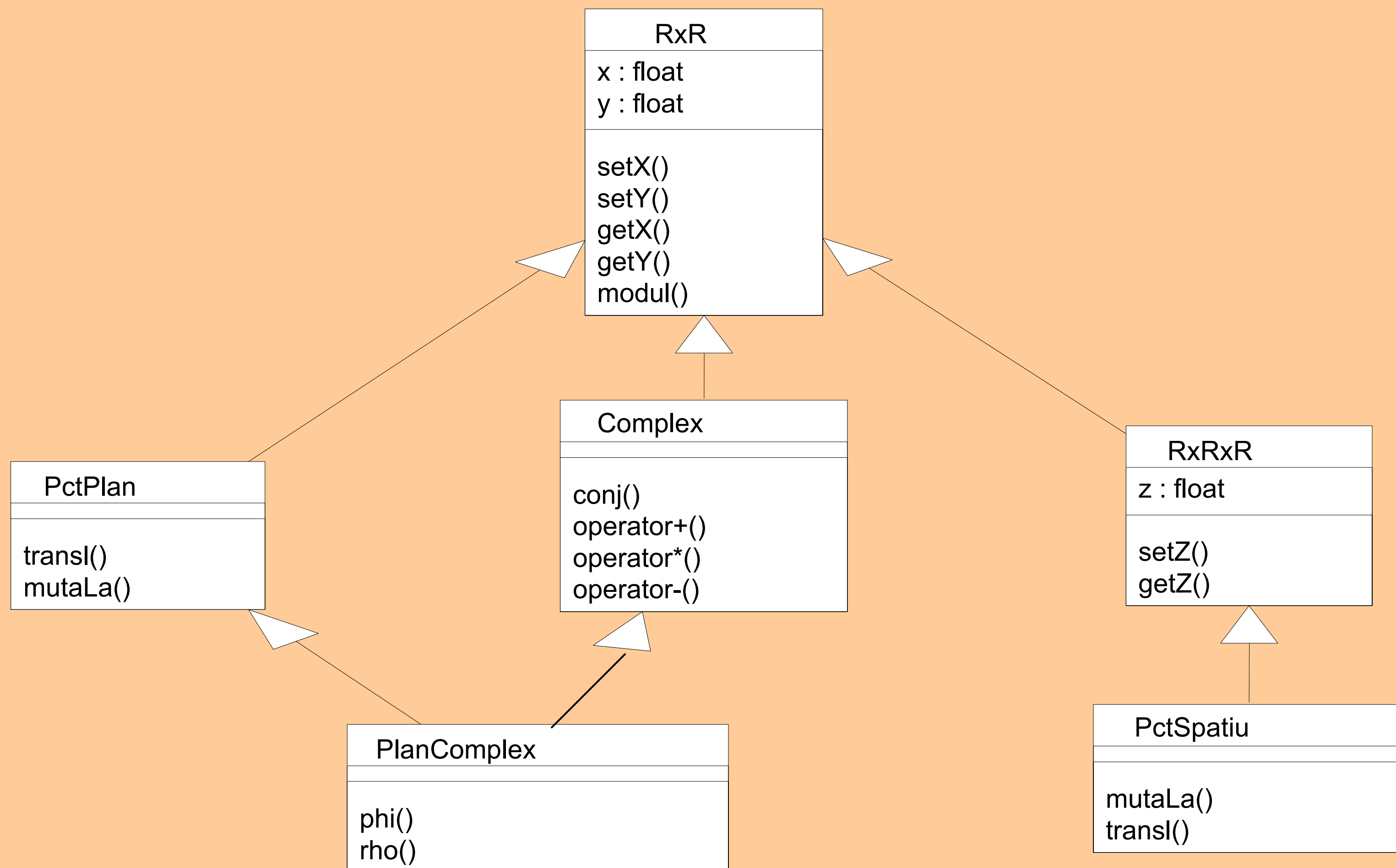
- Tipuri de moştenire:
  - **public:** membii **public** (**protected**) în bază rămân la fel în clasa derivată
  - **protected**: membrii **public** în clasa de bază devin **protected** în clasa derivată
  - **private**: membrii **public** şi **protected** din clasa de bază devin **private** în clasa derivată; este tipul implicit de moştenire
- Relaţia **friend** nu este moştenită, nu este tranzitivă
- În modul de derivare **private** se poate specifica păstrarea protecţiei unor membri:

```
class D:private B{
protected: B::p; public: B::q;
//…
};
```

Clase derivate

- Accesul la membrii : Funcțiile membru ale unei clase de bază pot fi redefinite în clasa derivată:
  - Ele pot accesa doar membrii **`public`** sau **`protected`** din clasa de bază
  - Pot accesa funcțiile din clasa de bază folosind operatorul `::`
- Constructori, Destructori
  - Constructorii şi destructorii nu se moştenesc
  - Constructorii clasei derivate apelează constructorii clasei de bază:
    - Constructorii impliciți nu trebuie invocați
    - Constructorii cu parametri sunt invocați în lista de inițializare
    - Ordinea de apel: constructor clasă de bază, constructori obiecte membru, constructor clasă derivată
  - Obiectele clasei derivate se distrug în ordine inversă: destructor clasă derivată, destructori membri, destructor clasă de bază

Ierarhia RxR



Ierarhia RxR

```
class RxR {
 protected:
  double x, y;
 public:
  RxR(double un_x = 0, double un_y = 0) : x(un_x), y(un_y) {}
  ~RxR() {}
  void setX(double un_x) { x = un_x; }
  double getX() {return x;}
  void setY(double un_y) { y = un_y; }
  double getY() { return y; }
  double modul();
};
class PctPlan : public RxR {
 public:
  PctPlan(double un_x=0, double un_y=0) : RxR(un_x, un_y) {}
  ~PctPlan() {}
  void translCu(double, double);
  void mutaLa(PctPlan&);
};
```

Ierarhia RxR

```
class Complex : public RxR {
 public:
  Complex(double un_x=0, double un_y=0) : RxR(un_x, un_y) {}
  Complex conj();
  Complex operator+ (Complex&);
};

class RxRxR : public RxR {
 protected:
  double z;
 public:
  RxRxR(double un_x, double un_y, double un_z)
        : RxR(un_x, un_y), z(un_z) {}
  void setZ(double un_z) { z = un_z; }
  double getZ() {return z;}
  double modul();
};
class PlanComplex : public PctPlan, public Complex {};
```

Clase derivate - Conversii standard

- Un obiect al clasei derivată poate fi convertit implicit la unul din clasa de bază
- O adresă a unui obiect derivat poate fi convertită implicit la o adresă de obiect din clasa de bază
- Un pointer la un obiect derivat poate fi convertit implicit la un pointer la obiect din clasa de bază
- Conversia reciprocă poate fi definită cu un constructor în clasa derivată

Clase derivate – Copiere

- Copierea o face constructorul de copiere şi **`operator=`**
- În cazul membrilor pointeri aceştia trebuie să existe explicit
- Ordinea de apel a constructorului de copiere:

Clasa de bază	**Clasa derivată**	**Ordinea de apel**
implicit	implicit	clasa baza, clasa derivată
explicit	implicit	clasa baza, clasa derivată
implicit	explicit	constructorul clasei derivate
explicit	explicit	constructorul clasei derivate trebuie sa apeleze constructorul clasei de bază

Clase derivate – Conflicte

- Conflict de metodă: metode cu acelaşi nume în clase incomparabile A, B ce derivează o clasă D
- Conflict de clasă: clasa D derivată din A1 şi A2 iar aceste sunt derivate din B: B este "accesibilă" pe două căi din D