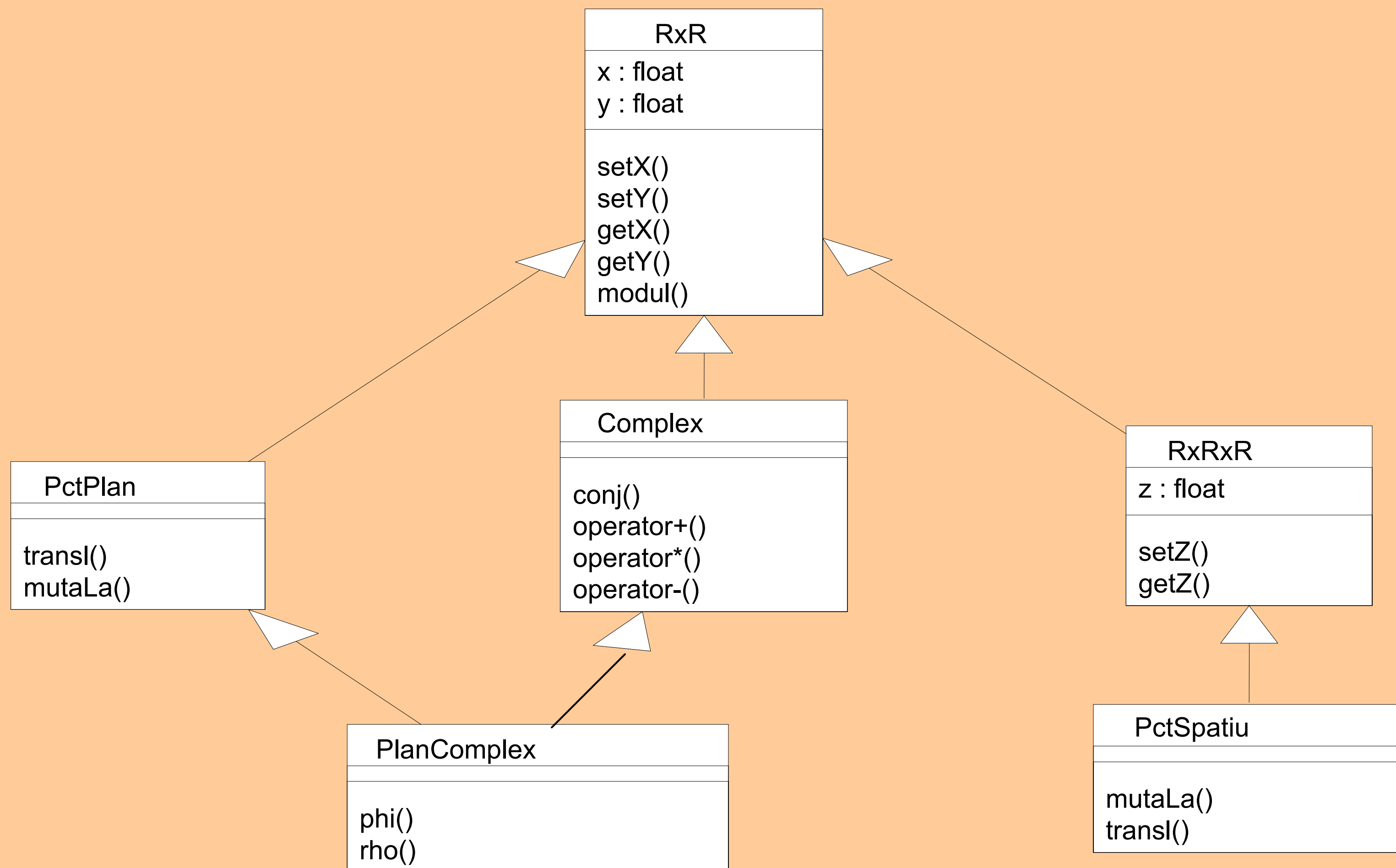
Cursul 7 – Derivare, Polimorfism

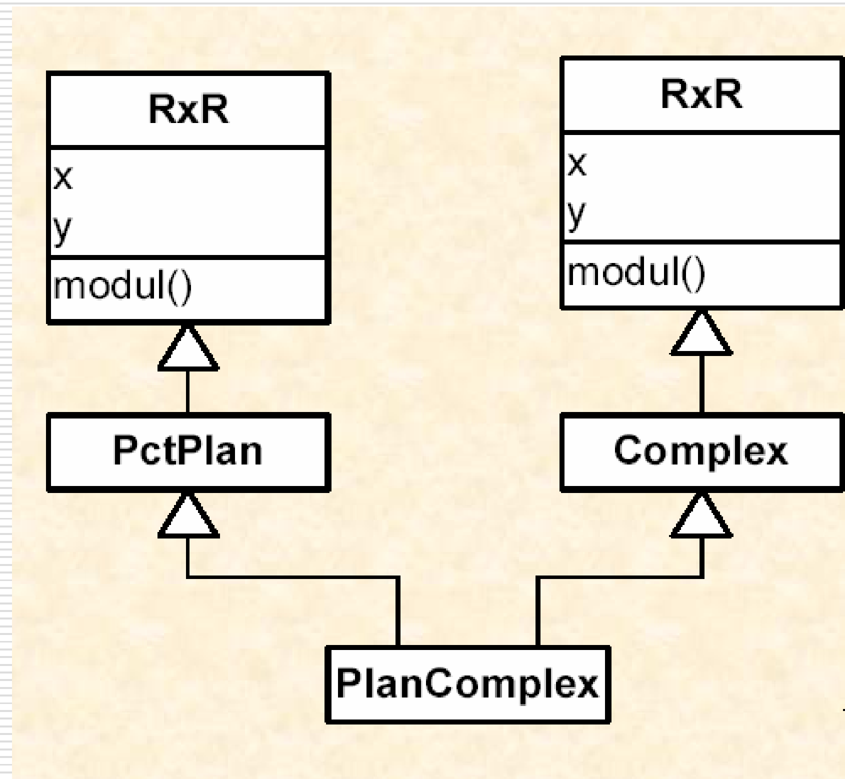
- Conflicte de derivare, derivare virtuală
  - exemplu ierarhia RxR
  - derivare fără partajare
  - derivare cu partajare - clase de bază virtuale
  - constructorii claselor de bază virtuale
- Derivare şi parametrizare
- Polimorfism
  - suprascriere -> legare statică
    - exemplu
  - funcții virtuale -> legare dinamică
    - exemplu

Ierarhia RxR



Clase derivate – Conflicte

- Conflict de clasă: clasa PlanComplex derivată din PctPlan şi Complex iar acestea sunt derivate din 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 PlanComplex : public PctPlan, public Complex{
public:
   PlanComplex(double = 0, double = 0);
   ~PlanComplex(){};
};
```

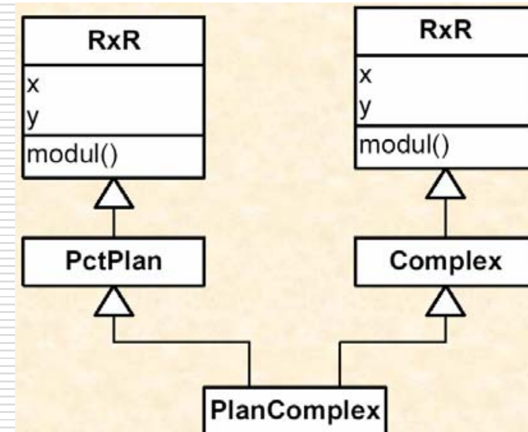
Ierarhia RxR

```
PlanComplex::PlanComplex(double un_x, double un_y) : PctPlan(un_x, un_y),
    Complex(un_x, un_y)
{}
PlanComplex::~PlanComplex()
{
    // nimic
}
void main(void) {
  PlanComplex pc(5, 5);
  cout << pc.modul() << endl;

  PlanComplex pc2;
  pc2.setX(5);
  pc2.setY(5);
  cout << pc2.modul() << endl;
}

// 'PlanComplex::modul' is ambiguous
//could be the 'modul' in base 'RxR' of base 'PctPlan' of class 'PlanComplex'
//or the 'modul' in base 'RxR' of base 'Complex' of class 'PlanComplex'
```

Moştenire fără partajare

- Fiecare clasă derivată are câte un exemplar din datele şi metodele clasei de bază

    PctPlan::x
    PctPlan::y
    Complex::x
    Complex ::y
    PctPlan::modul()
    Complex ::modul()



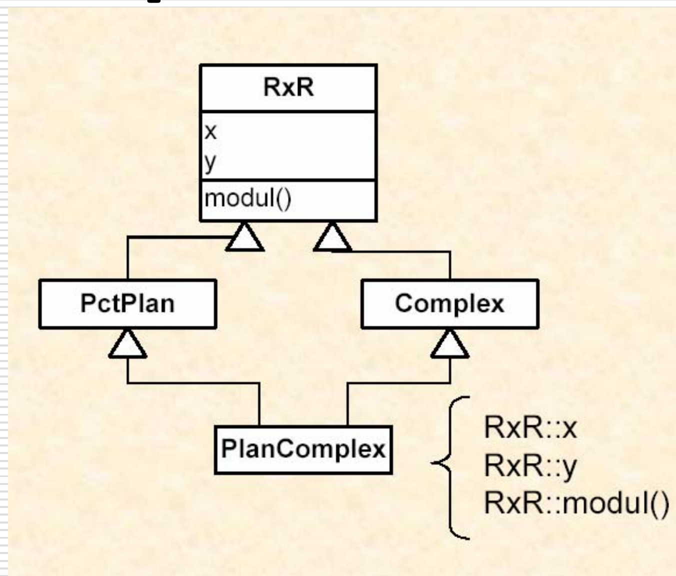
- Soluția (cazul derivării multiple):
  - Clasa de bază virtuală poate fi clasă de bază indirectă de mai multe ori fără a duplica membrii

Partajare: clase derivate virtuale

- Derivare virtuală:

```
class A1:virtual public B{};
class A2:public virtual B{};
class D: public A1, public A2{};
```

```
class PctPlan : virtual public RxR ...
class Complex : virtual public RxR ...
```



Partajare: clase derivate virtuale

```
void main(void) {
  PlanComplex pc(5, 5);
  cout << pc.modul() << endl; //0    ???

  PlanComplex pc2;
  pc2.setX(5);
  pc2.setY(5);
  cout << pc2.modul() << endl; //7.07107

}
```

- Care este explicația acestei diferențe?

Clase derivate virtuale

- Dacă în loc de:

```
PlanComplex::PlanComplex(double un_x, double un_y) :
    PctPlan(un_x, un_y), Complex(un_x, un_y){}
```

implementăm astfel:

```
PlanComplex::PlanComplex(double un_x, double un_y) :
    RxR(un_x, un_y){}
```

rezultatul va fi:

```
7.07107
7.07107
```

Clase derivate virtuale

- În cazul derivării virtuale, constructorul fiecarei clase derivate este responsabil de inițializarea clasei virtuale de bază:

```
class Baza{
public:
     Baza(int){}
     //
};
class D1 : virtual public Baza{
public:
     D1():Baza(1){}
     //
};
class D2 : virtual public Baza{
public:
     D2():Baza(6){}//…
};
```

```
class m1:public D1, public D2{
public:
          m1():Baza(2){}
          //
};
class m2:public D1, public D2{
public:
m2():Baza(3){}
          //
};
```

- Un constructor implicit în clasa de bază virtuală, simplifică lucrurile (dar le poate şi complica!)

Derivare - parametrizare

- Se poate defini o clasă parametrizată prin derivare
  - de la o clasă parametrizată
  - de la o instanță a unei clase parametrizate
  - de la o clasă obişnuită
- Se poate defini o clasă obişnuită prin derivare de la o instanță a unei clase parametrizate

Derivare - parametrizare

```
template <class T>
class Fisier: protected fstream {};

template <class T>
class FisierCitire: public virtual Fisier<T> {};

template <class T>
class FisierScriere: public virtual Fisier<T> {};

template <class T>
class FisierCitireScriere: public FisierCitire<T>,
                           public FisierScriere<T>
                           {};
```

Derivare şi parametrizare - Exemplu

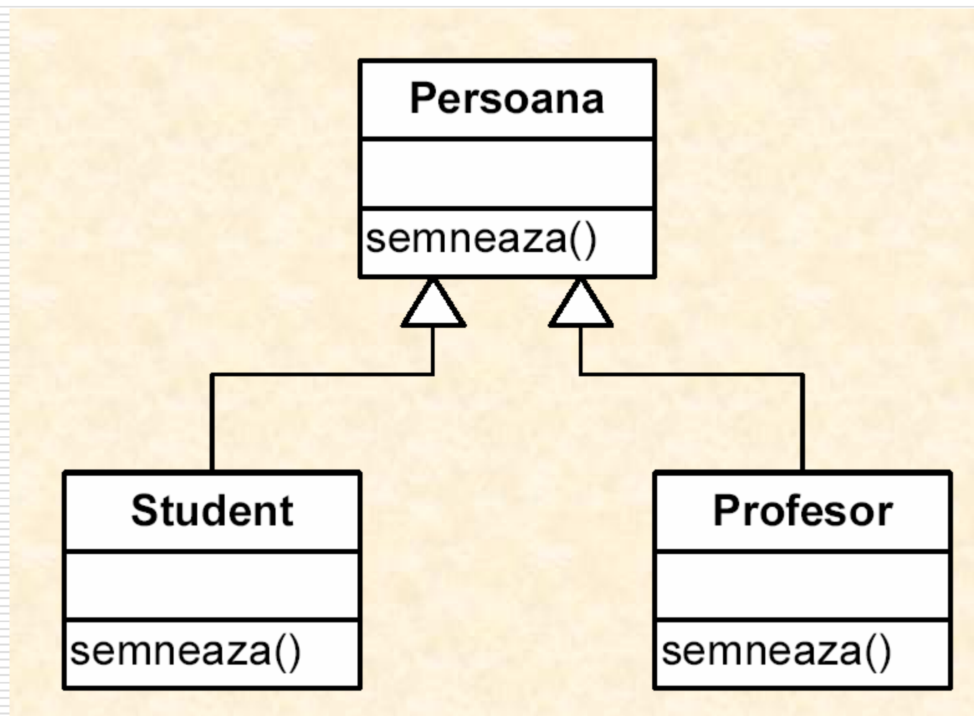
```
template <class TYPE>
class stack {
public:
   explicit stack(int size = 100)  // se foloseste doar explicit
           : max_len(size), top(EMPTY)
    {
         s = new TYPE[size]; assert (s != 0);
    }
   ~stack() { delete []s; }
   void  reset() { top = EMPTY; }
   void  push(TYPE c) { s[++top] = c; }
   TYPE  pop() { return s[top--]; }
   TYPE  top_of()const { return s[top]; }
   bool  empty()const { return top == EMPTY;}
   bool  full()const { return top == max_len - 1;}
private:
   enum   { EMPTY = -1 };
   TYPE*  s;
   int    max_len;
   int    top;
};
```

Derivare şi parametrizare - Exemplu

```
class safe_char_stack : public stack<char> {
public:
   // push, pop sigure
   void  push(char c)
     { assert (!full()); stack<char>::push(c); }
   char  pop()
     {assert (!empty()); return (stack<char>::pop());}
};

template <class Type>
class safe_stack : public stack<Type> {
public:
   // push, pop sigure
   void  push(Type c)
     { assert (!full()); stack<Type>::push(c); }
   char  pop()
     {assert (!empty()); return (stack<Type>::pop());}
};
```

Polimorfism - suprascriere

- Legare statică: asocierea *apel funcție -- corp(implementare) funcție* se face înainte de execuția programului (*early binding*) – la compilare



Polimorfism - suprascriere

- persoanele, studenții, profesorii au abilitatea de a semna:

```
class Persoana {
public:
    //…
    string getNume() const;
    void semneaza();
private:
    string id;
    string nume;
};
class Student:public Persoana {
    //…
    void semneaza();
}
class Profesor:public Persoana {
    //…
    void semneaza();
}
```

Polimorfism - suprascriere

- dar fiecare semnează in felul său:

```
void Persoana::semneaza()
{
    cout << getNume() << endl;
}
void Student::semneaza()
{
    cout << "Student " << getNume() << endl;
}
void Profesor::semneaza()
{
    cout << "Profesor " << getNume() << endl;
}
```

Polimorfism - suprascriere

- Exemplu:

```
Persoana pers("1001","Popescu Ion");
Student stud("1002", "Angelescu Sorin");
Profesor prof("1003","Marinescu Pavel");
pers.semneaza();
stud.semneaza();
prof.semneaza();
```

```
Popescu Ion
Student Angelescu Sorin
Profesor Marinescu Pavel
```

Polimorfism - suprascriere

- Exemplu:

```
// studentul poate semna ca persoana
stud.Persoana::semneaza();

// ... si profesorul poate semna ca persoana
prof.Persoana::semneaza();
```

```
Angelescu Sorin
Marinescu Pavel
```

Polimorfism - suprascriere

- Suprascriere -> legare statică:

```
void semneaza(Persoana& p)
{
   p.semneaza();
};
semneaza(pers);
semneaza(stud);// &stud poate inlocui &p
semneaza(prof);// &prof poate inlocui &p
```

```
Popescu Ion
Angelescu Sorin
Marinescu Pavel
```

Polimorfism - Funcții virtuale

- Legare dinamică: asocierea *apel funcție -- corp(implementare) funcție* se face la execuția programului (*late binding*, *run-time binding*), pe baza tipului obiectului căruia i se transmite funcția ca mesaj. În acest caz, funcția se zice *polimorfă*

Implementare polimorfism în C++:

- Declarația funcției în clasa de bază precedată de cuvântul **virtual**

```
class Persoana {
public:
    //…
    virtual void semneaza();
    //…
};
```

- O funcție virtuală pentru clasa de bază rămâne virtuală pentru clasele derivate. La redefinirea unei funcții virtuale în clasa derivată (*overriding*) nu e nevoie din nou de specificarea **virtual**

Polimorfism – funcții virtuale

- Polimorfism funcții virtuale-> legare dinamică:

```
Persoana pers("1001","Popescu Ion");
Student stud("1002", "Angelescu Sorin");
Profesor prof("1003","Marinescu Pavel");
pers.semneaza();
stud.semneaza();
prof.semneaza();
semneaza(pers);
semneaza(stud);
semneaza(prof);
```

```
Popescu Ion
Student Angelescu Sorin
Profesor Marinescu Pavel
Popescu Ion
Student Angelescu Sorin
Profesor Marinescu Pavel
```

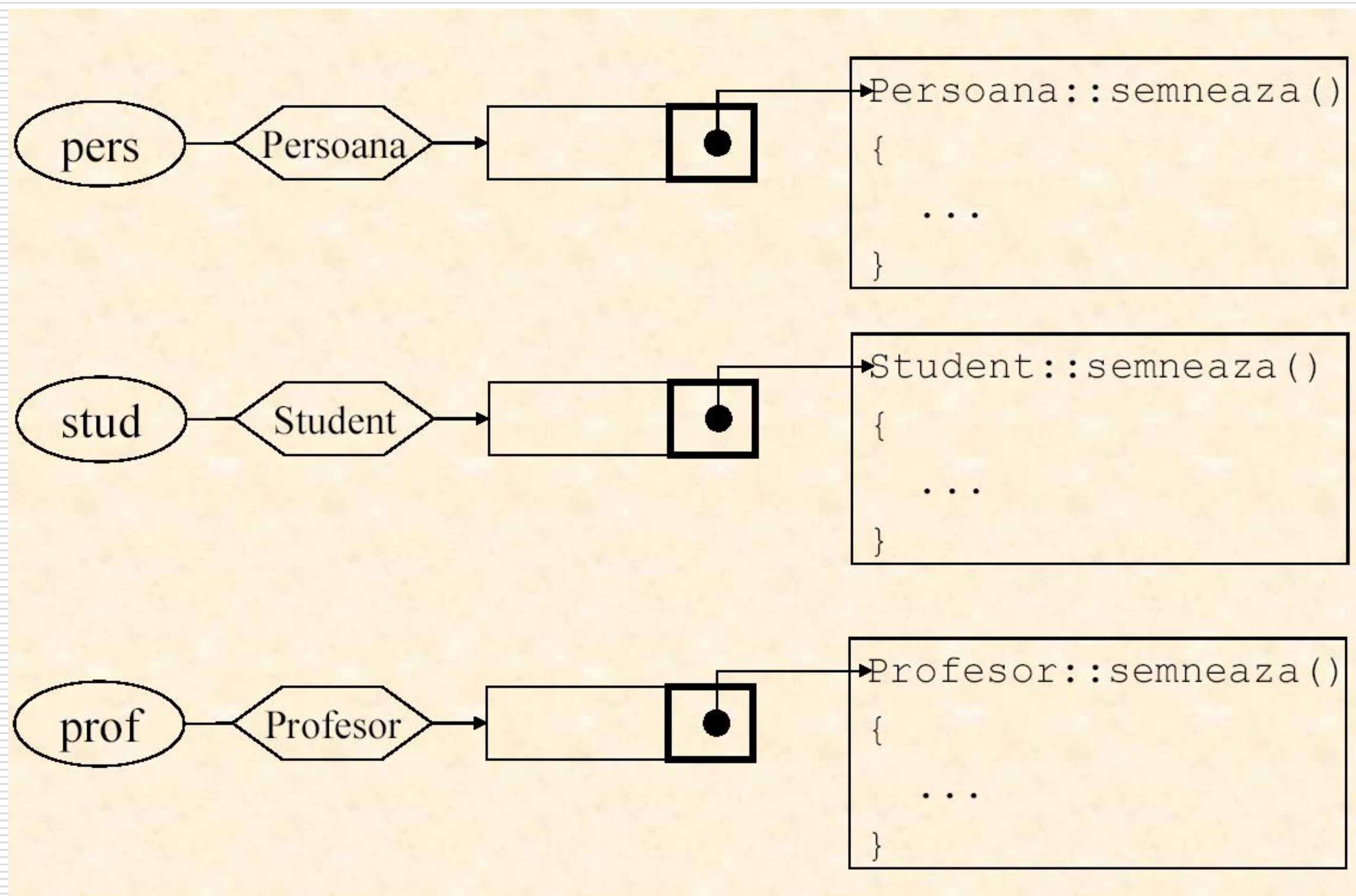
Implementare polimorfism în C++:

- Compilatorul crează o tabelă - VTABLE - pentru fiecare clasă care conține funcții virtuale; adresele tuturor funcțiilor virtuale sunt plasate în această tabelă. Un pointer VPTR este creat în fiecare clasă cu funcții virtuale; el pointează la această tabelă şi alege funcția corectă la un apel polimorfic

```
std::cout << sizeof(pers);
```

- Se obține:
  - 32 dacă semneaza() nu este virtuală
  - 36 dacă semneaza() este virtuală

Implementare polimorfism în C++:



Implementare polimorfism în C++:

- Un constructor nu poate fi virtual
- Un destructor poate fi virtual; destructorii claselor derivate vor fi virtuali ceea ce asigură apelul lor la distrugerea cu **`delete`** a pointerilor la clasa de bază

Destructor virtual

```
class A{
public:
     A(){p = new char[5];}
     ~A(){delete [] p;}
private:
     char* p;
{;
class D:public A{
public:
     D(){q = new char[500];}
     ~D(){delete [] q;}
private:
     char* q;
};
```

```
void f(){
     A* pA;
     pA = new D();
     //…
     delete pA;// doar apel ~A()
}
void main(){
     for(int i = 0; i<9; i++)
         f();
}

// pentru a se apela si ~D()
// se declara virtual ~A(){…}
```

Clase de bază abstracte

- Clasa de bază abstractă: are cel puțin o funcție virtuală pură:

```
class ABC{
public:
    virtual void m1() = 0;
   //
};
```

- Inițializarea unei metode virtuale cu zero este o convenție sintactică: specificarea unei clasei abstracte ce nu poate fi instanțiată

Clase de bază abstracte

- Clasele derivate din clase abstracte trebuie să definească toate metodele virtuale pure - *override* - altfel devin ele însele abstracte
- O clasă abstractă poate să aibă şi alţi membri (ce se moştenesc)
- Numai o metodă virtuală poate fi pură
- O clasă abstractă poate fi folosită în sisteme mari pentru a specifica cerinţele de proiectare. (Fiecare clasă trebuie să conţină metodele: **`listFields()`**, **`listMethods()`**, etc. Atunci ele sunt derivate din clasa abstractă în care acestea sunt metode virtuale pure)

Clase de bază abstracte - Exemplu

```
class figura{
  public:
      virtual void read_figure ()=0;
      virtual void compute_area ();
      virtual void compute_perim ();
      virtual void display_fig ();
  protected:
      double aria;
      double perimetru;
};
class cerc : public figura   //cercul este o figura
{
  public:
      void read_figure ();
      void compute_area ();
      void compute_perim ();
      void display_fig ();
  private:
      double raza;
};
```

Clase de bază abstracte - Exemplu

```
// FILE: Figura.cpp
#include <iostream.h>
#include "Figura.h"

void figura::compute_perim()
{perimetru = 0.0;}

void figura::compute_area()
{aria = 0.0;}

void figura::display_fig (){
   cout << "Aria este " << aria << endl;
   cout << "Perimetrul este " << perimetru
             << endl;
}
```

Clase de bază abstracte - Exemplu

```
// FILE: Cerc.cpp
// IMPLEMENTAREA CLASEI cerc
#include <iostream.h>
#include "cerc.h"
const double pi = 3.1415927;
void cerc::read_figure (){
   cout << "Raza > ";
   cin >> raza;
}
void cerc::compute_perim (){
   perimetru = 2.0 * pi * raza;
}
void cerc::compute_area (){
   area = pi * raza * raza;
}
void cerc::display_fig (){
   cout << "Figura este cerc" << endl;
   cout << "Raza cercului este" << raza << endl;
   figura::display_fig ();
}
```

Clase de bază abstracte - Exemplu

```
void main(){
   figura* get_figure ();//Alegerea unei figuri

   void process_figure(figura&);
   figura* my_fig;            // pointer la o figura

   // Se proceseaza figurile alese
   for (my_fig = get_figure (); my_fig != 0; my_fig = get_figure ())
    {
      process_figure (*my_fig);
      delete my_fig;          // se elibereaza memoria
   }
}

// PROCESAREA UNEI FIGURI
void process_figure(figura& fig){
   fig.read_figure ();
   fig.compute_area ();
   fig.compute_perim ();
   fig.display_fig ();
}
```

Clase de bază abstracte - Exemplu

```
Introdu optiunea pentru un obiect
Enter C (Cerc), D (Dreptunghi), or P (Patrat)
Enter X to exit program
p
Enter latura > 4
Figura este patrat
Latura este 4
Aria este 16
Perimetrul este 16

Introdu optiunea pentru un obiect
Enter C (Cerc), D (Dreptunghi), or P (Patrat)
Enter X to exit program
c
Enter radius > 1
Figura este cerc
Raza cercului este 1
Aria este 3.14159
Perimetrul este 6.28319
```