PROGRAMARE OBIECT-ORIENTATA

LABORATOR 5

- **MOSTENIRE SIMPLA DE TIP PUBLIC-PRIVATE**
- **MOSTENIRE CONSTRUCTORI**
- **ORDINEA DE APEL CONSTRUCTORI/DESTRUCTORI LA MOSTENIRE**
- **EFECTUL MODIFICATORULUI PROTECTED**
- **POLIMORFISM LA MOSTENIRE (SUPRASCRIERE METODE)**

Continuam in acest laborator mostenirea de clase in C++ analizand un exemplu (P5.0) cu metode publice in clasa dar mostenite private in clasa derivata. Astfel in laboratorul 4 am analizat urmatorul exemplu de mostenire cu precizarea ca acum , ca modificator de mostenire folosim private in loc de public:

```
class BAZA
{//creez clasa BAZA
	int x;//membru privat
	public://urmatoarele date sunt publice
		void initX(int n)//Setter pentru x
			{
				x=n;
			}
		void getX()//Getter pentru X
			{
				cout<<x;
			}

};//inchid clasa mereu cu punct si virgula !

class DERIVATA:private BAZA
{//creez clasa derivata
	int y;//membru privat
	public://urmatoarele date sunt publice
		void initY(int n)//setter pentru y
			{
				y=n;
			}
		void getY()//getter pentru y
			{
				cout<<y;
			}
};

int main()
{//creez obiectul de tip clasei derivate.
	DERIVATA D1;
	//D1.initX(10);//Eroare: initX nu este accesibil
	D1.initY(20);//setez valoarea pt y: OK(fara erori)
	//D1.getX();//Eroare: initX nu este accesibil
	D1.getY();//afisez valoarea pt y: OK(fara erori)
}
```

Dupa cum putem observa efectul modificatorului ***private*** este ca obiectul D1 de clasa DERIVATA nu mai are acces la metodele initX() si getX() chiar daca sunt publice in clasa de baza.

Ne dorim sa gasim o solutie prin care, pastrand modificatorul private la mostenire , obiectul D1 sa poata seta prin functiile init() si sa poata extrage prin functii get() valori si pentru atributul clasei de baza x, dar si pentru atributul clasei derivate y.

Solutia consta in:

- renuntarea la metodele initY() si getY() – le vom comenta
- modificarea metodei getX() astfel incat sa nu mai returneze valoarea lui x ci sa o afiseze direct (deoarece o functie nu poate avea mai multe instructiuni return)
- construirea unor metode noi
  - ✓ initXY() in care vom apela metoda initX() si
  - ✓ getXY() in care vom apela metoda getX()
- setarea simultana a lui x si a lui y prin metoda initXY()
- afisarea simultana a lui x si a lui y prin metoda getXY()

Implementarea solutiei va fi urmatorul program (P5.1):

```
#include <iostream>
using namespace std;
class BAZA
{//creez clasa BAZA
	int x;//membru privat
	public://urmatoarele date sunt publice
		void initX(int n)//Setter pentru x
			{
				x=n;
			}
		void getX()//Getter pentru X
			{
				cout<<x; //in loc de return, afisam direct x
			}
};//inchid clasa mereu cu punct si virgula !

class DERIVATA:private BAZA
{//creez clasa derivata
	int y;//membru privat
	public://urmatoarele date sunt publice
		void initXY(int n, int m)//setter pentru x si y
		{
			initX(n); //apelez seterul pentru X cu parametrul n
				//este echivalent cu x=n
			y=m;//setez valoarea pentru y
		}
```

```
		void getXY()//getter pentru x si y
		{
			getX(); //apelam getterul pentru X
			cout<<y<<endl;//afisam y
		}
};


int main()
{//creez obiectul de tip clasei derivate.
	DERIVATA D1;
	D1.initXY(10,20); //ok
	D1.getXY(); //ok
}
```

Comportamentul din functia main : obiectul D1 apeleaza metoda initXY cu 2 parametri, aceasta la randul ei apeleaza metoda initX cu primul parametru, dupa acest pas este setat primul parametru, apoi in mod direct se seteaza al doilea parametru chiar din functia initXY. Obiectul D1 apeleaza si metoda getXY care , la executie apeleaza metoda getX dar afiseaza apoi direct atributul y (fiind atribut propriu din clasa derivata.
Concluzia dupa acest exexmplu este ca private la mostenire nu inseamna “blocarea” accesului in totalitate la metodele clasei de baza, ci dupa cum s-a observat printr-o tehnica indirecta (in cascada) am reusit sa apelam si aceste metode ale clasei de baza.

Exercitiu propus: aplica tehnica din acest exemplu in cazul in care avem o mostenire pe mai multe nivele. (Exemplu: clasa C mosteneste clasa B care la randul ei mosteneste clasa A).

In continuare dorim sa analizam comportamentul constructorilor la mostenirea de clase. Pentru aceasta lucram in urmatorul program (P5.2) :

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

class BAZA
{
	int a;
	public:
		BAZA()
		{
			a=4;
			cout << "APEL CONSTRUCTOR IMPLICIT" << endl;
		}
		BAZA(int aa)
		{
			a=aa;
			cout << "APEL CONSTRUCTOR CU UN ARGUMENT"<< a <<endl;
		}

};
```

```
class DERIVAT:public BAZA
{//compilatorul ne va oferi un constructor implicit pentru DERIVAT
   //din moment ce nu are nici unul creat.
};


int main()
{
   DERIVAT D1; //OK, se foloseste contructorul creat de compilator

   return 0;
}
```

In general in C++, la crearea unui obiect de tipul unei clase derivate este necesar intai apelul unui constructor din clasa de baza apoi urmeaza apelul unui constructor din clasa derivata. In exemplul de mai sus, pentru crearea obiectului D1 ar fi necesar un constructor din clasa de baza , mai exact un constructor implicit , pe care il avem scris, dar este necesar si un constructor implicit din clasa derivate pe care nu il avem.

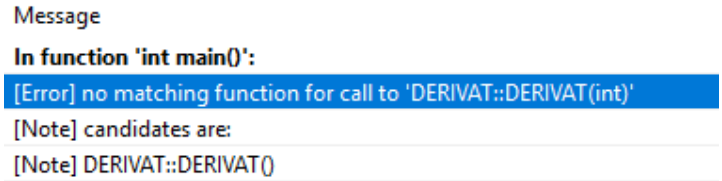
Deoarece codul de mai sus compileaza si ruleaza fara erori, am spune la o prima vedere ca acel constructor implicit al clasei de baza este mostenit si folosit in clasa derivata. De fapt nu este deloc asa – pentru clasa derivata, compilatorul creaza pentru noi un constructor implicit si il foloseste la crearea obiectului D1. Putin mai jos in aceasta lucrare vom dovedi chiar acest fapt *: constructorii nu se mostenesc.

In continuare modificam modul de creare al obiectului D1 din exemplul P5.2 si anume ii dam un parametru pentru a schimba constructorii apelati:

```
int main()
{
   DERIVAT D1(40); //NOT OK, nu exista constructor cu parametri in derivata

   return 0;
}
```

In acest caz se va apela un constructor implicit din clasa de baza (deja scris) si un constructor cu parametru din clasa derivata. Dar cum nu exista nici un constructor in clasa derivata compilatorul ar incerca crearea unuia cu parametri dar nu poate crea decat constructor implicit. Astfel primim eroarea:



Aceasta eroare dovedeste faptul ca se cauta un constructor cu parametri dar nici nu se poate crea unul de catre compilator, acesta poate doar crea unul implicit (mesajul “candidates are: DERIVAT::DERIVAT()”)

Vom corecta noi eroarea prin construirea in clasa derivata a unui constructor cu un parametru:

In exemplul P5.2 dupa crearea constructorului cu un parametru, clasa derivata va arata astfel:

```
class DERIVAT:public BAZA
{
	int b;
	public:
		DERIVAT(int bb)
		{
			b=bb;
			cout<<"APEL CONSTRUCTOR CU UN PARAMETRU DIN DERIVATA";
		}
};
```

Acum compilarea si rularea vor decurge fara erori si vom avea ca output:

```
APEL CONSTRUCTOR IMPLICIT
APEL CONSTRUCTOR CU UN PARAMETRU DIN DERIVATA
--------------------------------
Process exited after 0.6893 seconds with return value 0
Press any key to continue . . .
```

Am afirmat mai devreme si faptul ca la crearea unui obiect cu argument, se foloseste un constructor cu parametru din clasa derivata si unul implicit din clasa de baza. Mesajele de mai sus dovedesc acest lucru.

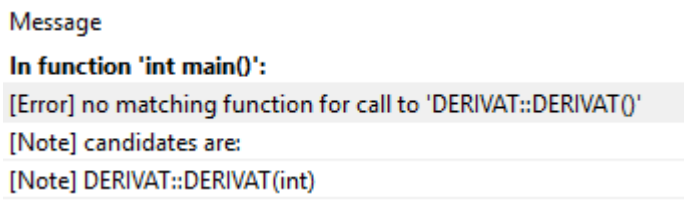
In laboratorul 6 vom invata o tehnica prin care in clasa derivata la definire constructorul cu parametru poate apela un constructor tot cu parametru din clasa de baza , pentru a nu-l mai folosi pe cel implicit.

Acum dorim sa revenim la modul de creare al obiectului D1 fara parametru dar sa pastram in clasa derivata doar constructorul cu parametru. Asadar in exemplul P5.2 modificam definitia obiectului D1:

```
int main()
{
   DERIVAT D1; //NOT OK, compilatorul nu mai creaza constructor

   return 0;
}
```

Ceea ce se observa acum este ca din nou primim eroare :



De data aceasta, compilatorul cauta un constructor implicit, dar nu il mai creaza automat, deoarece avem deja unul cu parametri. Acest lucru zice si compilatorul : “candidates are: DERIVAT::DERIVAT(int)” inseamna ca exista alt constructor nu cel solicitat pentru apelare.

Aceasta eroare ne dovedeste ceea ce spuneam la pagina 4 la * : constructorii nu se mostenesc. Daca s-ar fi mostenit, nu am fi fost nevoiti sa cream constructori si in clasele derivate.

Exact acest lucru facem acum , completam clasa derivate cu constructor implicit astfel incat sa nu mai primim erori nici la obiectele fara parametri nici la obiectele cu parametri. Asadar clasa derivata a exemplului P5.2 va arata astfel:

```
class DERIVAT:public BAZA
{
	int b;
	public:
		DERIVAT()
		{
			b=5;
			cout << "APEL CONSTRUCTOR IMPLICIT DIN DERIVATA" << endl;
		}
		DERIVAT(int bb)
		{
			b=bb;
			cout<<"APEL CONSTRUCTOR CU UN PARAMETRU DIN DERIVATA";
		}
};
```

Acum ne permitem sa cream doua obiecte in functia main , unul fara parametru , altul cu parametru, deci functia main a exemplului P5.2 va fi:

```
int main()
{
   DERIVAT D1; //OK, se foloseste contructorul creat de programator
   DERIVAT D2(300); //OK , avem constructor cu parametru

   return 0;
}
```

Iar la output vom primi:

```
APEL CONSTRUCTOR IMPLICIT
APEL CONSTRUCTOR IMPLICIT DIN DERIVATA
APEL CONSTRUCTOR IMPLICIT
APEL CONSTRUCTOR CU UN PARAMETRU DIN DERIVATA
--------------------------------
Process exited after 0.398 seconds with return value 0
Press any key to continue . . .
```

In acest output, pentru obiectul D1 se afiseaza primele doua mesaje (doar mesaje din constructor implicit), iar pentru obiectul D2 se afiseaza intai mesaj din constructor implicit apoi din constructor cu parametru.

Al treilea exemplu din acest laborator doreste sa prezinte ordinea de apel a constructorulor si destructorilor in clasele de baza si derivate:

Am completat ultima varianta a programului de la P5.2 astfel incat sa avem si destructori, astfel am obtinut programul P5.3:

```
class BAZA
{
	int a;
	public:
		BAZA()
		{
			a=4;
			cout << "APEL CONSTRUCTOR IMPLICIT DIN BAZA" << endl;
		}
		BAZA(int aa)
		{
			a=aa;
			cout << "APEL CONSTRUCTOR CU UN ARGUMENT DIN BAZA"<< a <<endl;
		}
		~BAZA()
		{
			cout<<"APEL DESTRUCTOR DIN BAZA"<< endl;
		}

};

class DERIVAT:public BAZA
{
	int b;
	public:
		DERIVAT()
		{
			b=5;
			cout << "APEL CONSTRUCTOR IMPLICIT DIN DERIVATA" << endl;
		}
```

```
        DERIVAT(int bb)
        {
                b=bb;
                cout<<"APEL CONSTRUCTOR CU UN PARAMETRU DIN DERIVATA"<< endl;;
        }
        ~DERIVAT()
        {
                cout<<"APEL DESTRUCTOR DIN DERIVATA"<< endl;;
        }
};

int main()
{
   DERIVAT D1; //OK, se foloseste contructorul creat de programator
   //DERIVAT D2(300); //OK , avem constructor cu parametru

   return 0;
}
```

In functia main() am lasat doar obiectul D1, pentru a fi creat doar cu constructor implicit si daca executam acest cod dupa compilare obtinem urmatorul output:

```
APEL CONSTRUCTOR IMPLICIT DIN BAZA
APEL CONSTRUCTOR IMPLICIT DIN DERIVATA
APEL DESTRUCTOR DIN DERIVATA
APEL DESTRUCTOR DIN BAZA
```

Dupa cum se poate observa, la crearea obiectului se apeleaza intai constructorul din clasa de baza apoi cel din derivata iar la distrugerea obiectului , ordinea este inversa , intai destructorul din derivate apoi destructorul din baza.

Al patrulea exemplu (P5.4) din acest laborator urmareste descrierea comportamentului pentru modificatorul ***protected*** :

Unele explicatii sunt deja scrise in cod altele le vom nota dupa acest program (P5.4):

```
#include <iostream>
using namespace std;
class BAZA
{
        protected:
                int a,b;//doua variabile de tip protected

//efectul unui modificator de acces este pana la
//urmatorul modificator de acces
        public:
                void SetAB(int n, int m)
                //int n de aici are vizibilitate doar in interiorul
                //functiei SetAB, deci nu este vazut si in clasa derivata
```

```
		{//setter multiplu pentru a si b de tip public
			a=n, b=m;
			//atribui lui a valoarea lui n
			//atribui lui b valoarea lui m
		}
};

class DERIVATA:public BAZA//clasa mostenita cu numele DERIVATA
{//mosteneste public clasa cu numele BAZA
//mostenire publica inseamna ca am acces
//la tot ce este public si protected in clasa de baza
	int c;//atribut fara modificator de acces
	//tot ce nu are modificator de acces este implicit private
	public://setter pentru c declarat public
		void SetC(int n)
		{//atentie ! am voie sa folosesc din nou variabila n
		//pentru ca n are vizibilitate doar in interiorul
		//functiei SetC,
			c=n;//atribui lui c valoarea lui n
			//n este un argument de tip integer
		}

		void GetABC()
	{//afisez toate tributele, cate atribute are clasa DERIVATA ?
			//raspuns: 3 aribute
	//intrebare: cate atribute sunt mostenite? R:2 atribute
	//intrebare: cate atribute sunt proprii? R:1 atribut/
	cout<<a<<" "<<b<<" "<<c<<endl;
//a si b se pot afisa pentru ca sunt metode protected in clasa de baza
//mostenite tot public
	}
};

int main()
{
	DERIVATA D1;
//creez un obiect de tipul clasei derivate cu numele D1
	D1.SetAB(4,5);//apelez setter pentru A si B,
//public in clasa de baza , public in mostenire, deci vizibil in main
	D1.SetC(8);//apelez setter pt C,
//public in clasa de baza , public in mostenire, deci vizibil in main
	D1.GetABC();
//public in clasa de baza , public in mostenire, deci vizibil in main
	D1.a=5;
	D1.b=9;
	D1.c=12;
}
```

Protected este un modificator ce are o actiune situata intre public si private. Datele membre de tip protected (atat atribute cat si metode) sunt accesibile in clasele derivate, dar innacesibile inafara claselor derivate. De fapt spunem ca au efect de public in clasele derivate si efect de private inafara claselor derivate (exemplu : in functia main).

Aceste comportamente se aplica si in programul P5.4 de mai sus. Atributele a, b ale clasei de baza sunt protected deci sunt accesibile in clasa derivata, din acest motiv le putem accesa in metoda GetABC(). Accesarea se transpune aici in afisarea atributelor a si b in mod direct folosind cout<<.

Inafara de comentariile din program care de data aceasta sunt mai bogate , trebuie sa subliniem erorile pe care le primim in cazul ultimelor 3 instructiuni:

Cand D1 acceseaza a sau b si doreste sa ii atribuie valoarea 5 respectiv 9, acestea nu sunt accesibile deoarece a si b sunt protected deci innacesibile in functia main(). Cand ne referim la accesul lui D1 pentru atributul c, acesta nu e accesibil de data aceasta din cauza ca c este private. Deci este bine sa separam clar ca pot exista mai multe motive pentru ca un anumit atribut sa devina innacesibil.

Evidentiem in continuare si erorile pe care le primim la compilare:

```
Message
In function 'int main()':
[Error] 'int BAZA::a' is protected
[Error] within this context
[Error] 'int BAZA::b' is protected
[Error] within this context
[Error] 'int DERIVATA::c' is private
[Error] within this context
```

Daca vom comenta ultimele 3 instructiuni , output-ul programului va fi:

```
4 5 8

--------------------------------
Process exited after 0.3023 seconds with return value 0
Press any key to continue . . .
```

Concluzia este ca protected ne ajuta sa izolam utilizarea unei date membre dintr-o clasa doar in clasele derivate.

Ultimul element pe care dorim sa il studiem in acest laborator este despre **polimorfism** de functii . Vom trata doar Suprascrierea , urmand ca Supraincarcarea de functii sa o prezentam in laboratorul 6.

Polimorfismul de functii reprezinta la baza refolosirea aceluiasi nume de functie (tinand cont inclusiv de minuscule/majuscule !), de cel putin doua ori in acelasi fisier .cpp . Poate fi refolosit acelasi nume de functie atat in interiorul aceleasi clase dar si in clase diferite care sunt si mostenite una din alta.

Asa cum am anuntat mai sus, exista doua variante de polimorfism :

- **Suprascriere**: aceiasi semnatura de functie dar difera corpul functiei
- **Supraincarcare**: semnatura diferita dar si corpul functiei este diferit

In acest laborator vom trata Suprascrierea de functii. Mai exact avem a lucra cu **aceiasi semnatura de functie** ceea ce inseamna :

- Tipul returnat al functiei este acelasi
- Numele este acelasi
- Numarul de parametri este acelasi
- Tipul de date al parametrilor este acelasi

Modificam partial codul programului P5.2 si va rezulta urmatorul cod P5.5 :

```
class BAZA
{
	int a;
	public:
		BAZA()
		{
			a=4;
			//cout << "APEL CONSTRUCTOR IMPLICIT" << endl;
		}
		BAZA(int aa)
		{
			a=aa;
			//cout << "APEL CONSTRUCTOR CU UN ARGUMENT"<< a <<endl;
		}
		void metodaX()
		{
			cout <<"APEL metodaX DIN BAZA";
		}

};

class DERIVAT:public BAZA
{
	int b;
	public:
		DERIVAT()
		{
			b=5;
			//cout << "APEL CONSTRUCTOR IMPLICIT DIN DERIVATA" << endl;
		}
		DERIVAT(int bb)
		{
			b=bb;
			//cout<<"APEL CONSTRUCTOR CU UN PARAMETRU DIN DERIVATA";
		}
		void metodaX()
		{
```

```
                cout <<"APEL metodaX DIN DERIVATA";
            }
};

int main()
{
   DERIVAT D1; //OK, se foloseste contructorul creat de programator
   DERIVAT D2(300); //OK , avem constructor cu parametru
   D1.metodaX();

   return 0;
}
```

Ca si modificari, am scris functia “metodaX” de tip void si fara parametri atat in clasa de baza cat si in clasa derivata. Diferenta este in corpul acestei functii deoarece fiecare afiseaza un mesaj diferit.
In functia main, folosind obiectul D1 am apelat functia “metodaX”. Output-ul este:

```
APEL metodaX DIN DERIVATA
--------------------------------
Process exited after 0.573 seconds with return value 0
Press any key to continue . . .
```

Asadar observam ca obiectul D1 fiind de clasa derivata, se afiseaza mesajul din clasa derivata. Logica imediata ne spune ca daca cream un obiect de tipul clasei de baza , sa spunem B1, apoi apelam cu B1 “metodaX” se va afisa mesajul din clasa de baza.
Codul functiei main() din programul P5.5 va fi in acest caz:

```
int main()
{
   DERIVAT D1; //OK, se foloseste contructorul creat de programator
   DERIVAT D2(300); //OK , avem constructor cu parametru
   D1.metodaX(); //OK , se afiseaza mesajul din clasa derivate
   cout<<endl;
   BAZA B1;
   B1.metodaX();//OK , se afiseaza mesajul din clasa de baza

   return 0;
}
```

Adica output-ul va fi:

```
APEL metodaX DIN DERIVATA
APEL metodaX DIN BAZA
--------------------------------
Process exited after 0.4826 seconds with return value 0
Press any key to continue . . .
```

Dar , totusi… ne punem problema cum am putea obliga un obiect de tipul clasei derivate sa acceseze metodaX din clasa de baza si implicit sa afiseze mesajul din clasa de baza.

De fapt apelul *D1.metodaX();* este o forma prescurtata de apel. Pentru a folosi oblige obiectul D1 sa apeleze metoda din clasa dorita de noi trebuie sa folosim forma integrala acestui apel , care foloseste operatorul de rezolutie si pe care il vom prezenta direct prin forma finala a functiei main() pentru P5.5:

```
int main()
{
   DERIVAT D1; //OK, se foloseste contructorul creat de programator
   DERIVAT D2(300); //OK , avem constructor cu parametru
   D1.DERIVAT::metodaX(); // se apeleaza metodaX din derivata
   cout<<endl;
   D1.BAZA::metodaX(); // se apeleaza metodaX din baza
   cout<<endl;
   BAZA B1;
   B1.BAZA::metodaX();

   return 0;
}
```

Iar output-ul va fi de data aceasta:

```
APEL metodaX DIN DERIVATA
APEL metodaX DIN BAZA
APEL metodaX DIN BAZA
--------------------------------
Process exited after 0.4131 seconds with return value 0
Press any key to continue . . .
```

Ne intereseaza primele doua mesaje, care sunt generate de acelasi obiect D1 si aceiasi functie metodaX(), dar folosind operatorul de rezolutie si clasa corespunzatoare am reusit sa afisam mesajul dorit.

In continuare dorim sa cream clasa CARTE din care sa mostenim clasa REVISTA iar in cele doua sa integram toate elementele prezentate in acest laborator:

```
#include<iostream>
using namespace std;
class CARTE
{
	protected:
		int nr_pag;//va fi accesibil doar din clasa REVISTA
	public:
		void Setnr_pag(int n)
		{
			nr_pag=n;
		}
		void Getnr_pag()
		{
			cout<<nr_pag<<endl;
		}
		void Ruperepag(int a)//metoda suprascrisa
		{
			nr_pag=nr_pag-a;
			cout<<"Au ramas "<<nr_pag<<" pagini dupa rupere."<<endl;
		}
		CARTE()
		{
			nr_pag=80;
			cout<<"Apelare constructor implicit din carte"<<endl;
		}
		CARTE(int nr){
			nr_pag=nr;
			cout<<"Apelare constructor cu parametru din carte"<<endl;
		}

		~CARTE()
		{
			cout<<"Apelare destructor pentru carte"<<endl;
		}
};
class REVISTA:public CARTE
{
	int pret;
	public:
		void Setpret_pagini(int n,int p)
		{
			Setnr_pag(n);
			pret=p;
		}
		int Getpret_pagini()
		{
			Getnr_pag();
			cout<<pret<<endl;
		}
```

```
		int GetALL()
		{
			cout<<nr_pag<<" si "<<pret<<endl;
		}
		void Ruperepag(int a)
		{
			nr_pag=nr_pag-a;
			cout<<"Am rupt "<<a<<" pagini"<<endl;
		}

		REVISTA()
		{
			pret=40;
			cout<<"Apelare constructor implicit din revista"<<endl;
		}
		REVISTA(int p)
		{
			pret=p;
			cout<<"Apelare constructor cu parametru din revista"<<endl;
		}

		~REVISTA()
		{
			cout<<"Apelare destructor pentru revista"<<endl;
		}
};
int main()
{
	REVISTA R1;
	R1.Setpret_pagini(10,100);
	R1.Getpret_pagini();
	R1.GetALL();
	//R1.nr_pag=25;
	//R1.pret=35;
	R1.CARTE::Ruperepag(5);
	R1.REVISTA::Ruperepag(5);

	REVISTA R2(15);
}
```

Explicatii:

- *R1.Setpret_pagini(10,100)* este un setter din clasa derivata cu doi parametri care, pentru a seta valoarea primului atribut apeleaza *Setnr_pag(n)* un alt setter al clasei de baza care este accesibil chiar daca mostenirea este private. Asemanator pentru Getter-ul din clasa derivata care, in interiorul lui apeleaza Getter-ul din clasa de baza.

Output:

```
Apelare constructor implicit din carte
Apelare constructor implicit din revista
10
100
Apelare destructor pentru revista
Apelare destructor pentru carte
```

- Pentru a observa comportamentul mostenirii constructorilor , comentam constructorul implicit din clasa derivata si incercam crearea unui obiect implicit (fara parametri) adica obiectul R1. Deoarece constructorii (dar si destructorii) NU se mostenesc, vom primi eroare:

```
Message
In function 'int main()':
[Error] no matching function for call to 'REVISTA::REVISTA()'
[Note] candidates are:
[Note] REVISTA::REVISTA(int)
```

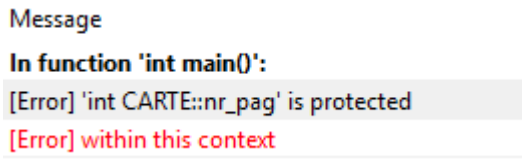
- Ordinea de apel la construirea obiectelor este directa (baza, derivata) iar la distrugere este inversa (derivata, baza)

```
Apelare constructor implicit din carte
Apelare constructor implicit din revista
10
100
Apelare destructor pentru revista
Apelare destructor pentru carte
```

- Datele membre de tip protected sunt accesibile in clasele derivate (pe oricate niveluri) in timp ce inafara claselor derivate sunt innacesibile. GetALL() din REVISTA afiseaza atat numarul de pagini (protected) cat si pretul cartii:

```
Apelare constructor implicit din carte
Apelare constructor implicit din revista
10 si 100
Apelare destructor pentru revista
Apelare destructor pentru carte
```

Iar daca setam direct atributul nr_pag (protected), (eliminam comentariile de la linia `//R1.nr_pag=25;`) primim eroare:



- La suprascrierea unei functii, pentru a preciza clasa din care dorim sa apelam functia suprascrisa, folosim operatorul de rezolutie. In lipsa acestuia, compilatorul va accesa functia dupa tipul obiectului care o acceseaza. R1, de tip REVISTA, apeleaza pe rand metoda Ruperepag(5) din fiecare clasa:

```
Apelare constructor implicit din carte
Apelare constructor implicit din revista
10
100
10 si 100
Au ramas 5 pagini dupa rupere.
Am rupt 5 pagini
Apelare constructor implicit din carte
Apelare constructor cu parametru din revista
Apelare destructor pentru revista
Apelare destructor pentru carte
Apelare destructor pentru revista
Apelare destructor pentru carte

--------------------------------
Process exited after 0.3947 seconds with return value 0
Press any key to continue . . .
```

Mesajul “Au ramas 5 pagini dupa rupere” provine din apelul functiei din clasa CARTE, iar mesajul “Am rupt 5 pagini” provine din apelul functiei din clasa REVISTA.