5. Moştenirea claselor

Obiective

- Înțelegerea modului în care se pot crea clase noi prin moştenirea din clasele existente
- Înțelegerea modului în care moştenirea promovează reutilizarea codului
- Înțelegerea noțiunilor de clasă de bază şi clasă derivată

- *Introducere*

În acest capitol vom studia una dintre cele mai importante facilități pe care le oferă programarea orientată pe obiecte: *moştenirea claselor*. Moştenirea este o formă de reutilizare a codului în care noile clase sunt create din clase existente prin absorbirea atributelor şi comportamentelor lor, prin înlocuirea unor comportamente şi prin adăgarea unor atribute şi comportmante noi. Reutilizarea codului economiseşte timp prețios în dezvoltarea software. Este încurajată reutilizarea secvențelor de cod testate şi de calitate pentru reducerea problemelor care ar putea apărea după ce sistemul ar deveni funcțional.

Atunci când creează o nouă clasă, în loc să rescrie complet noile date membre şi funcții membre, programatorul poate arăta că noua clasă va *moşteni* datele membre şi funcțiile membre dintr-o *clasă de bază* definită anterior. Noua clasă este o *clasă derivată*. Fiecare clasă derivată poate deveni, mai departe, un candidat pentru derivări ulterioare. Atunci când folosim *moştenirea simplă*, o clasă este derivată dintr-o singură clasă de bază. *Moştenirea multiplă* presupune posibilitatea ca o clasă să fie derivată din mai multe clase de bază. Moştenirea simplă este uşor de folosit şi o vom ilustra prin câteva exemple. Moştenirea multiplă este mai complexă şi vom studia doar un exemplu simplu.

O clasă derivată poate adăuga noi date membre şi funcții membre, astfel încât poate fi mai cuprinzătoare decât clasa ei de bază. O clasă derivată este mai particulară decât clasa ei de bază şi reprezintă un grup mai mic de obiecte. Forța mecanismului moştenirii vine din posibilitatea de a defini în clasele derivate adăugiri, înlocuiri sau rafinări ale elementelor moştenite din clasa de bază.

Limbajul C++ oferă trei tipuri de moşteniri: `public`, `protected` şi `private`. În acest capitol ne vom concentra pe moştenirea public şi le vom explica pe scurt pe celelalte două. Moştenirea `private` poate fi folosită ca o modalitate alternativă de compunere a claselor iar moştenirea `protected` este folosită rar. Când derivarea este `public`, fiecare obiect al unei clase derivate poate fi folosit oriunde se foloseşte un obiect al clasei de bază din care a fost derivată acea clasă. Operația inversă nu este posibilă, deci obiectele clasei de bază nu sunt şi obiecte ale clasei derivate. Avantajul acestei relații rezumată prin expresia *un obiect din clasa derivată este şi obiect al clasei de bază* va fi ilustrat în capitolul în care vom introduce o altă facilitate fundamentală a programării orientate pe obiecte, *polimorfismul*. Acesta permite procesarea generică a obiectelor care fac parte din clase derivate din aceeaşi clasă de bază.

Experiența dezvoltării aplicațiilor software arată că adeseori segmente semnificative de cod sunt dedicate tratării cazurilor speciale. Devine dificilă proiectarea acestor sisteme pentru că proiectantul şi programatorul devin preocupați de ele. Programarea orientată pe obiecte prin procesul numit *abstractizare* este o soluție la aceste situații. Dacă programul este supraîncărcat de cazuri speciale, o

soluție la îndemână este utilizarea secvențelor `switch` care pot oferi posibilitatea de a scrie secvențe logice de procesare pentru a trata fiecare caz individual. Vom vedea într-unul dintre capitolele următoare că moştenirea şi polimorfismul sunt alternative mai simple ale logicii `switch`.

Trebuie să facem distincția între relațiile tip „*este un*” sau „*este o*” („*is a*”) şi „*are un*” sau „*are o*” („*has a*”). Aşa cum am văzut deja, o relație „*has a*” înseamnă compunere de clase, adică un obiect al unei clase *are* ca membri unul sau mai multe obiecte ale altor clase. O relație „*is a*” este o moştenire. Acesta este tipul de relație în care obiectele unei clase derivate pot fi tratate şi ca obiecte ale clasei de bază.

În acest capitol discutăm specificatorul de acces la membri numit `protected`. O clasă derivată nu poate accesa membrii `private` ai clasei sale de bază pentru că în caz contrar s-ar încălca ideea de încapsulare a clasei de bază. O clasă derivată poate, însă, să acceseze membrii `public` şi `protected` ai clasei de bază. O clasă derivată poate accesa membrii `private` ai clasei sale de bază doar dacă aceasta a implementat funcții de acces `public` sau `protected` pentru ei.

O problemă a moştenirii este că o clasă derivată moşteneşte şi implementări ale funcțiilor membre `public` din clasa de bază de care nu are nevoie. Când implementarea unui membru din clasa de bază nu este potrivit clasei derivate, acest membru poate fi suprascris în clasa derivată cu o implementare corespunzătoare.

- *Moştenirea: clase de bază şi clase derivate*

Adeseori, un obiect al unei clase *este un* obiect al altei clase în acelaşi timp. Un dreptunghi *este un* patrulater, la fel ca şi un pătrat, un paralelogram sau un trapez. Astfel, clasa `Dreptunghi` se poate spune că moşteneşte clasa `Patrulater`. În acest context, clasa `Patrulater` se numeşte *clasă de bază* şi clasa `Dreptunghi` se numeşte *clasă derivată*. Un dreptunghi *este un* tip particular de patrulater, dar este incorect să afirmăm că un patrulater *este un* dreptunghi. Iată câteva exemple de moştenire:

Clasa de bază	**Clase derivate**

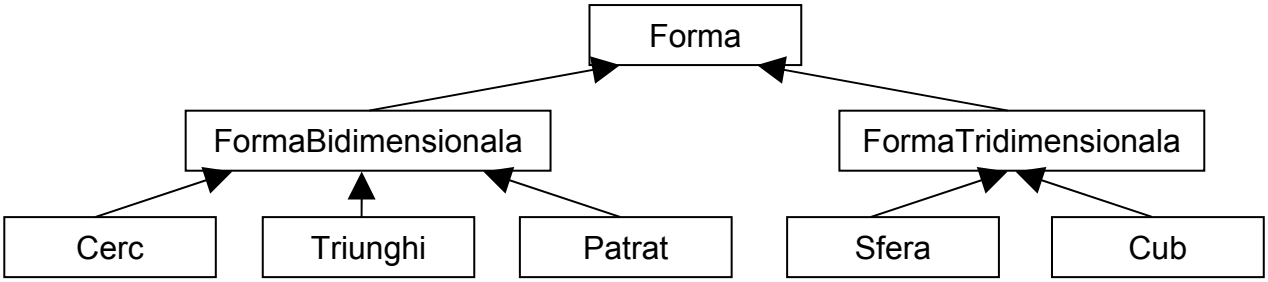
`Forma`	`Cerc`
	`Triunghi`
	`Dreptunghi`

`Credit`	`CreditAuto` `CreditImobiliar`

`Cont`	`ContCurent`
	`ContDepozit`
	`ContPlati`

Moştenirea conduce la structuri ierarhice arborescente. O clasă de bază se află într-o relație ierarhică cu clasele derivate din ea. Bineînțeles că o clasă poate să existe în mod individual. Atunci când clasa este folosită împreună cu mecanismul moştenirii, ea poate deveni clasă de bază care oferă atribute şi comportamente altor clase, sau clasă derivată şi moşteneşte atribute şi comportamente.

O ierarhie simplă de moştenire este ierarhia `Forma` din figura de mai jos.



Formele sunt împărțite în această ierarhie în forme unidimensionale si bidimensionale. `Cerc`, `Triunghi` şi `Patrat` sunt clase derivate din `FormaBidimensionala`. Un obiect al clasei `Cerc` este în acelaşi timp şi o `FormaBidimensionala`, dar şi o `Forma`.

În limbajul C++, sintaxa prin care clasa `FormaBidimensionala` este definită ca o clasă derivată din `Forma` este următoarea:

```
class FormaBidimensionala : public Forma
{
  ...
};
```

Aceasta este o derivare `public` şi este cel mai folosit tip de derivare. În această situație, membrii `public` şi `protected` din clasa de bază sunt moşteniți ca membri `public`, respectiv `protected` ai clasei derivate. Membrii `private` ai clasei de bază nu sunt accesibili din clasele derivate. Funcțile `friend` nu se moştenesc.

- ***Membrii `protected`***

Membrii `public` ai unei clase de bază pot fi accesați de orice funcție din program. Membrii `private` al unei clase de bază sunt accesibili doar funcțiilor membre sau prietenilor clasei.

Nivelul de acces `protected` este un nivel intermediar între accesul `public` şi cel `private`. Membrii `protected` ai unei clase de bază pot fi accesați doar de membrii şi de prietenii clasei de bază şi de membrii şi prietenii claselor derivate. Membrii claselor derivate pot referi membrii `public` şi protected ai clasei de bază folosind numele acestor membri. Datele `protected` depăşesc ideea de încapsulare pentru că o schimbare a membrilor `protected` din clasa de bază poate influența toate clasele derivate. În general, se recomandă ca datele membre să fie declarate `private`, iar `protected` trebuie folosit numai atunci când este strict necesar.

- ***Cast între pointerii la clasa de bază şi pointerii la clasa derivată***

Un obiect al unei clase derivate public poate fi tratat ca obiect al clasei ei de bază. Aceasta face posibilă o serie întreagă de operații.

Pe de altă parte, programatorul trebuie să folosească un cast explicit pentru a converti un pointer la clasa de bază într-un pointer la o clasă derivata deoarece compilatorul consideră că această operație este una periculoasă. Acest proces se numeşte *downcasting a pointer*. Trebuie, însă, acordată atenție dereferențierii unui astfel de pointer, programatorul asigurându-se că tipul pointerului se potriveşte cu tipul obiectului către care pointează. Standardele recente C++ au introdus metode mai elaborate pentru acest tip de conversie prin *run-time type identification* (RTTI), `dynamic_cast` şi `typeid`.

Operația inversă, prin care un pointer al clasei derivate este convertit la un pointer al clasei de bază se numeşte *upcasting a pointer*. Ilustrăm aceste operații de cast prin folosirea claselor `Point`, care este clasă de bază, şi `Circle` care este clasă derivată.

Exemplu

point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>
using std::ostream;
class Point
{
  friend ostream& operator<<(ostream&, const Point&);
  public:
    Point(int = 0, int = 0); //constructor implicit
    void setPoint(int, int); //seteaza coordonatele
    int getX() const { return x; } //returneaza x
    int getY() const { return y; } //returneaza y
  protected: //accesibil din clasele derivate
    int x, y; //x si y coordonatele unui punct
};
#endif
```

point.cpp

```
#include <iostream>
#include "point.h"
//Constructor pentru clasa Point
Point::Point(int a, int b)
  { setPoint(a, b); }
//Seteaza coordonatele x si y ale unui punct
void Point::setPoint(int a, int b)
{
  x = a;
  y = b;
}
//Afiseaza un punct
ostream& operator<<(ostream& output, const Point& p)
{
  output << '[' << p.x << ", " << p.y << ']';
  return output; //pentru cascadarea apelurilor
}
```

circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include <iostream>
using std::ostream;
#include <iomanip>
using std::ios;
using std::setiosflags;
using std::setprecision;
```

```
#include "point.h"

class Circle : public Point //Circle derivata din Point
{
  friend ostream& operator<<(ostream&, const Circle&);
  public:
    //constructor implicit
    Circle(double r = 0.0, int x = 0, int y = 0);
    void setRadius(double); //seteaza radius
    double getRadius() const; //intoarce radius
    double area() const; //calculeaza aria
  protected:
    double radius;
};
#endif
```

circle.cpp

```
#include "circle.h"
//Constructorul clasei Circle apeleaza
//constructorul pentru Point si apoi
//initializeaza raza
Circle::Circle(double r, int a, int b)
  : Point(a, b)
  { setRadius(r); }
//Seteaza raza cercului
void Circle::setRadius(double r)
  { radius = (r > 0 ? r : 0); }
//Returneaza raza cercului
double Circle::getRadius() const
  { return radius; }
//Calculeaza aria cercului
double Circle::area() const
  { return 3.14159 * radius * radius; }
//Afiseaza datele despre cerc in forma
//Centrul = [x, y]; Raza = #.##
ostream& operator<<(ostream& output, const Circle& c)
{
  output << "Centrul = " << static_cast<Point>(c)
         << "; Raza = "
         << setiosflags(ios::fixed | ios::showpoint)
         << setprecision(2) << c.radius;
  return output;
}
```

test_point_circle.cpp

```
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>
#include "point.h"
#include "circle.h"
```

```
int main()
{
  Point *pointPtr = 0, p(30, 50);
  Circle *circlePtr = 0, c(2.7, 120, 89);
  cout << "Punctul p: " << p << "\nCercul c: " << c << '\n';

  //Trateaza Circle ca un Point
  //Este vizibila doar partea care provine din clasa de baza
  pointPtr = &c; //asigneaza adresa unui Circle lui pointPtr
  cout << "\nCercul c (via *pointPtr): "
       << *pointPtr << '\n';

  //Trateaza un Circle ca un Circle
  //cast de la pointer la clasa de baza la pointer
  //la clasa derivata
  circlePtr = static_cast<Circle*>(pointPtr);
  cout << "\nCercul c (via *circlePtr):\n" << *circlePtr
       << "\nAria lui c (via circlePtr): "
       << circlePtr->area() << '\n';

  //PERICULOS: Trateaza Point ca un Circle
  pointPtr = &p; //asigneaza adresa unui Point la pointPtr

  //cast al clasei de baza la clasa derivata
  circlePtr = static_cast<Circle*>(pointPtr);
  cout << "\nPunctul p (via *circlePtr):\n" << *circlePtr
       << "\nAria obiectului la care pointeaza circlePtr: "
       << circlePtr->area() << endl;

  return 0;
}
```

Rulând acest program obținem următorul rezultat:

```
Punctul p: [30, 50]
Cercul c: Centrul = [120, 89]; Raza = 2.70

Cercul c (via *pointPtr): [120, 89]

Cercul c (via *circlePtr):
Centrul = [120, 89]; Raza = 2.70
Aria lui c (via circlePtr): 22.90

Punctul p (via *circlePtr):
Centrul = [30, 50]; Raza = 0.00
Aria obiectului la care pointeaza circlePtr: 0.00
```

Interfața publică a clasei `Point` cuprinde funcțiile membre `setPoint`, `getX` şi `getY`. Datele membre `x` şi `y` ale clasei `Point` sunt `protected`. Astfel, datele membre nu pot fi accesate de clienții clasei, dar clasele derivate din `Point` vor putea folosi în mod direct aceste date moştenite. Dacă aceste date ar fi fost `private`, ar fi

fost nevoie de apeluri ale funcțiilor membre `public` din clasa `Point` pentru accesul acestor date chiar şi în clasele derivate.

Clasa `Circle` este derivată `public` din clasa `Point`. Această derivare este specificată prin prima linie din definiția clasei:

```
class Circle : public Point //Circle mosteneste Point
```

Semnul `:` din header-ul definiției clasei indică această moştenire. Cuvântul cheie `public` arată tipul moştenirii. Toți membrii `public` şi `protected` ai clasei `Point` sunt moşteniți ca `public`, respectiv `protected` în clasa `Circle`. Înseamnă că interfața publică a clasei `Circle` cuprinde membrii `public` ai clasei `Point`, dar şi membrii `public` ai clasei `Circle` care sunt `area`, `setRadius` şi `getRadius`.

Constructorul clasei `Circle` trebuie să invoce constructorul clasei `Point` pentru inițializarea părții din obiect care provine din clasa de bază. Această invocare se face prin lista de inițializare:

```
Circle :: Circle(double r, int a, int b)
   : Point(a, b)  //apelul constructorului clasei de baza
```

În situația în care constructorul clasei `Circle` nu ar fi invocat în mod explicit constructorul clasei `Point`, atunci ar fi fost apelat automat constructorul implicit al clasei `Point` pentru inițializarea datelor membre `x` şi `y`. În acest caz, în lipsa constructorului implicit compilatorul semnalează eroare.

Programul creează `pointPtr` ca pointer la un obiect tip `Point` şi `circlePtr` ca pointer la un obiect `Circle` şi obiectele `p` şi `c` de tip `Pointer` şi `Circle`. Afişarea obiectelor `p` şi `c` se face prin apelul operatorului `<<` supraîncărcat separat pentru fiecare dintre cele două tipuri de date.

Operatorul `<<` supraîncărcat în clasa `Point` poate manipula şi obiecte din clasa `Circle` tipărind partea care provine din clasa `Point` pentru obiectele clasei `Circle`. Apelul se face prin cast de la referința `c` de tip `Circle` la un `Point`. Rezultatul este apelul `operator<<` pentru `Point` şi tipărirea coordonatelor `x` şi `y` folosind formatarea specifică celei definite în clasa `Point`. Prin asignarea

```
pointPtr = &c;
```

este ilustrată operația de *upcasting* prin care un obiect al unei clase derivate este tratat ca obiect al clasei de bază. Rezultatul tipăririi pointerului dereferențiat `*pointPtr` este partea din obiectul `c` care provine din clasa `Point` deoarece compilatorul interpretează apelul `operator<<` ca un apel pentru un obiect tip `Point`. Prin pointerul la clasa se bază „*se vede*” doar partea din obiectul `c` care provine din clasa `Point`.

Asignarea

```
circlePtr = static_cast<Circle*>(pointPtr);
```

ilustrează operația de *downcasting* prin care un pointer la un obiect din clasa de bază este convertit la un pointer la un obiect dintr-o clasă derivată. Operatorul `static_cast` din Standard C++ permite implementarea acestei operații. Pointerul `pointPtr` este transformat înapoi în `Circle*` şi rezultatul operației este asignat pointerului `circlePtr`. Acest pointer a provenit inițial din adresa obiectului `c`, aşa cum se poate vedea în funcția `main`. Rezultatul afişării este conținutul obiectului `c`.

În final, programul asignează un pointer la clasa de bază (adresa obiectului `p`) lui `pointPtr` şi apoi aplică o operație de cast lui `pointPtr` pentru a îl transforma în `Circle*`. Rezultatul celei de-a doua operații este păstrat în `circlePtr`. Astfel, obiectul `p` de tip `Point` este tipărit folosind `operator<<` pentru `Circle` şi pointerul dereferențiat `*circlePtr`. Valoare tipărită pentru rază este 0 fiindcă ea nu există în

obiectul `p`, el fiind un `Point`. Afişarea unui `Point` ca un `Circle` poate conduce, aşadar, la afişarea unor valori nedefinite.

- ***Folosirea funcțiilor membre***

Funcțiile membre ale clasei derivate poate avea nevoie să acceseze anumite date şi funcții membre. O clasă derivată nu poate accesa direct membri `private` ai clasei de bază. Acesta este un aspect crucial de inginerie software in C++. Dacă o clasă derivată ar putea accesa membrii `private` ai clasei de bază, ar încălca principiul încapsulării pentru obiectele clasei de bază. Ascunderea membrilor `private` este de mare ajutor în testarea, depanarea şi modificarea corectă a sistemelor. Dacă o clasă derivată ar putea accesa membrii `private` ai clasei sale de bază, atunci clasele care sunt derivate mai departe din clasele derivate pot accesa şi ele aceşti membri şi aşa mai departe. Propagarea accesului la datele `private` ar anula astfel beneficiile încapsulării datelor în cadrul unei ierarhii de derivare.

- ***Suprascrierea membrilor clasei de bază în clasele derivate***

O clasă derivată poate suprascrie (*override*) o funcție membră a clasei de bază printr-o nouă versiune a acestei funcții cu aceeaşi semnătură. Dacă semnăturile sunt diferite, atunci ar fi vorba de supraîncărcare (*overload*), nu se suprascriere. Când numele acestei funcției suprascrise este menționat în clasa derivată, este selectată automat versiunea din clasa derivată. Poate fi folosită şi veriunea din clasa de bază, dar pentru aceasta trebuie folosit operatorul domeniu.

Exemplu

employee.h

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

class Employee
{
  public:
    Employee(const char*, const char*);//constructor
    void print() const;//tipareste numele si prenumele
    ~Employee();//destructor
  private:
    char* firstName;//string alocat dinamic
    char* lastName; //string alocat dinamic
};
#endif
```

employee.cpp

```
#include <iostream>
using std::cout;

#include <cstring>
#include <cassert>
#include "employee.h"

//Constructorul aloca dinamic spatiu pentru nume si prenume
```

```
//si foloseste strcpy pentru a copia
//numele si prenumele in obiect
Employee::Employee(const char* first, const char* last)
{
  firstName = new char[strlen(first) + 1];
  assert(firstName != 0);//termina programul daca nu se aloca
  strcpy(firstName, first);

  lastName = new char[strlen(last) + 1];
  assert(lastName != 0);//termina programul daca nu se aloca
  strcpy(lastName, last);
}
//Tipareste numele
void Employee::print() const
  { cout << firstName << ' ' << lastName; }

//Destructor
Employee::~Employee()
{
  delete[] firstName;//eliberarea zonei de memorie
  delete[] lastName; //eliberarea zonei de memorie
}
```

hourly.h

```
#ifndef HOURLY_H
#define HOURLY_H

#include "employee.h"

class HourlyWorker : public Employee
{
  public:
  HourlyWorker(const char*,
               const char*,
               double, double);//constructor
  double getPay() const;//calculeaza si returneaza salariul
  void print() const;//suprascrierea functiei
                     //din clasa de baza
  private:
    double wage;//salariul pe ora
    double hours; //ore lucrate pe saptamana
};
#endif
```

hourly.cpp

```
#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::ios;
using std::setiosflags;
using std::setprecision;
```

```
#include "hourly.h"
//Constructorul clasei HourlyWorker
HourlyWorker::HourlyWorker(const char* first,
                           const char* last,
                           double initHours,
                           double initWage)
  : Employee(first, last)//apel al constructorului
                         //clasei de baza
{
  hours = initHours; //ar trebui validat
  wage = initWage;   //ar trebui validat
}
//Returneaza salariul muncitorului
double HourlyWorker::getPay() const
  { return wage * hours; }
//Tipareste numele si salariul
void HourlyWorker::print() const
{
  cout << "Se executa HourlyWorker::print()\n\n";
  Employee::print();//Apelul functiei print din clasa de baza
  cout << " este un lucrator angajat cu ora platit cu "
       << setiosflags(ios::fixed | ios::showpoint)
       << setprecision(2) << getPay() << " EUR" << endl;
}
```

test_hourlyworker.cpp

```
#include"hourly.h"

int main()
{
  HourlyWorker h("Bob", "Smith", 40.0, 10.00);
  h.print();

  return 0;
}
```

Rulând acest program obținem următorul rezultat:

```
Se executa HourlyWorker::print()

Bob Smith este un lucrator angajat cu ora platit cu
400.00 EUR
```

Definiția clasei `Employee` constă din două date membre private `char*`, `firstName` şi `lastName` şi trei funcții membre, un constructor, un destructor şi `print`. Funcția constructor primeşte două şiruri de caractere şi alocă dinamic tablouri de `char` pentru stocarea lor. Folosim macro-ul `assert` pentru a determina dacăa fost alocată memoria pentru `firstName` şi `lastName`. În cazul în care alocarea nu s-a putut realiza, programul se termină cu un mesaj de eroare. O alternativă la folosirea lui `assert` este secvența `try`/`catch` ştiind că operația `new` care nu se desfăşoară corect generează excepția `bad_alloc`. Deoarece datele membre ale lui Employee sunt private, singura modalitate prin care pot fi accesate este funcția membră print care afişează numele şi prenumele angajatului.

Destructorul eliberează memoria care a fost alocată dinamic pentru evitarea fenomenului numit „*memory leak*”.

Clasa HourlyWorker este derivată din clasa Employee şi moştenirea este de tip public. Noua clasă îşi defineşte propria variantă a funcției print care are acelaşi prototip cu cea din clasa de bază Employee. Acesta este un exemplu de suprascriere în clasa derivată a unei funcții din clasa de bază. În acest fel, clasa HourlyWorker are acces la două funcții print. De regulă, în astfel de cazuri funcția din clasa derivată apelează funcția din clasa de bază pentru a implementa o parte din operații. În exemplul nostru, funcția print din clasa derivată apelează funcția print din clasa de bază pentru a tipări numele şi prenumele angajatului, informații care provin din clasa de bază. Funcția print din clasa derivată adaugă valorile datelor din clasa derivată, adică informațiile legate de plata muncitorului. Din cauză că cele două funcții print au semnături identice, apelul versiunii din clasa de bază se face precedând numele funcției de numele clasei şi de operatorul domeniu:

```
Employee::print();
```

- *Moştenirea `public`, `protected` şi `private`*

La derivarea unei clase, clasa de bază poate fi moştenită public, protected sau private. Derivarea protected sau private reprezintă opțiuni folosite rar. În mod obişnuit se foloseşte derivarea public. În tabelul de mai jos este prezentat pentru fiecare tip de moştenire modul de acces în clasa derivată al membrilor clasei de bază.

Specificatorul de acces al datei membre în clasa de bază	**Tipul moştenirii**		
	moştenire `public`	**moştenire `protected`**	**moştenire `private`**
`public`	`public` în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică, funcție `friend` sau funcție nemembră.	`protected` în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică sau funcție `friend`.	`private` în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică sau funcție `friend`.
`protected`	`protected` în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică sau funcție `friend`.	`protected` în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică sau funcție `friend`.	`private` în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică sau funcție `friend`.
`private`	Inaccesibil în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică sau funcție `friend` prin funcții	Inaccesibil în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică sau funcție `friend` prin funcții	Inaccesibil în clasa derivată. Poate fi accesat direct prin orice funcție membră nestatică sau funcție `friend` prin funcții membre

	membre `public` sau `protected` din clasa de bază.	membre `public` sau `protected` din clasa de bază.	`public` sau `protected` din clasa de bază.

La derivarea unei clase dintr-o clasă de bază `public`, membrii `public` din clasa de bază devin membri `public` ai clasei derivate şi membrii `protected` ai clasei de bază devin membri `protected` ai clasei derivate. Membrii `private` ai clasei de bază nu pot fi accesați direct în clasele derivate, dar pot fi accesați prin funcții membre `private` sau `protected` din clasa de bază.

La derivarea unei clase dintr-o clasă de bază `protected`, membrii `public` sau `protected` din clasa de bază devin membri `protected` în clasa derivată. La derivarea dintr-o clasă de bază `private`, membrii `public` şi `protected` ai clasei de bază devin membri `private` în clasa derivată. De exemplu, funcțiile devin funcții utilitare. Derivările `private` şi `protected` nu sunt relații de tip „*este un*” sau „*este o*”.

- *Clase de bază directe şi clase de bază indirecte*

O clasă de bază poate fi *clasă de bază directă* a clasei derivate sau *clasă de bază indirectă* a clasei derivate. O clasă de bază directă a unei clase derivate este listată explicit în header-ul clasei derivate, după semnul `:`. O clasă de bază indirectă nu este listată explicit în header-ul clasei. Ea este moştenită cu două sau mai multe nivele înainte în ierarhia de moştenire.