

Programare orientată pe obiecte

Curs 6

Realizarea flexibilității cu funcțiile friend operator

Situație :

ob+100 //valida

100+ob //invalida

(necesitatea de a poziționa în unele aplicații mereu obiectul în stânga este câteodată o povară care poate determina eșecuri)

b Soluția : supraîncărcarea folosind o funcție **friend**, nu o funcție membru (în acest caz, funcției operator îi sunt transmise explicit ambele argumente - pentru a permite atât **obiect+întreg** cât și **întreg+obiect**, prin supraîncărcare de două ori - o versiune pentru fiecare situație)

b Exemplu Program5

```
class loc {  
    int longitud, latitud;  
public:  
    loc() {}  
    loc(int lg, int lt) {  
        longitud = lg;  
        latitud = lt;  
    }  
}
```

```
void arata() {  
    cout << longitud << " ";  
    cout << latitud << "\n";  
}  
  
loc operator+(loc op2);  
friend loc operator+(loc op1, int op2);  
friend loc operator+(int op1, loc op2);  
};
```

Realizarea flexibilității cu funcțiile friend operator – continuare 1

```
loc loc::operator+(loc op2) {  
    loc temp;  
    temp.longitud = longitud + op2.longitud;  
    temp.latitud = latitud + op2.latitud;  
    return temp;  
}
```

// + este supraincarcat pentru loc + int

```
loc operator+(loc op1, int op2) {  
    loc temp;  
    //observati ordinea operandilor  
    temp.longitud = op1.longitud + op2;  
    temp.latitud = op1.latitud + op2;  
    return temp;  
}
```

// + este supraincarcat pentru int + loc

```
loc operator+(int op1, loc op2) {  
    loc temp;  
    temp.longitud = op1 + op2.longitud;  
    temp.latitud = op1 + op2.latitud;  
    return temp;  
}
```

```
int main() {  
    loc ob1(10, 20), ob2(5, 30), ob3(7, 14);  
  
    ob1.arata();  
    ob2.arata();  
    ob3.arata();  
  
    ob1 = ob2 + 10; //amindoua  
    ob3 = 10 + ob2; //sînt valide  
  
    ob1.arata();  
    ob3.arata();  
    return 0;  
}
```

```
10 20  
5 30  
7 14  
15 40  
15 40
```

Supraîncărcarea operatorilor new și delete

- Este posibil ca new și delete să fie supraîncărcate (spre exemplu dacă doriți să folosiți unele metode speciale de alocare de memorie : rutine de alocare care, atunci când s-a epuizat memoria disponibilă (*heap*), să folosească automat un fișier de pe disc ca memorie virtuală) pe scheletul

```
void *operator new(size_t marime) {  
    //efectueaza alocarea  
    return pointer_la_memorie;  
}  
  
void operator delete(void *p){  
    //memoria liberă este indicată de p  
}
```

- cu `size_t` din `stdlib.h` (este definită și în altele) care exprimă cea mai mare porțiune de memorie contiguă care poate fi alocată (întreg fără semn)

Supraîncărcarea operatorilor new și delete – continuare 1

Exemplu 6

```
class loc {
    int longitud, latitud;
public:
    loc() {}
    loc(int lg, int lt) {
        longitud = lg;
        latitud = lt;
    }
    void arata() {
        cout << longitud << " ";
        cout << latitud << "\n";
    }
    void *operator new(size_t marime);
    void operator delete(void *p);
};

//new supraîncărcat relativ la loc
void *loc::operator new(size_t marime){
    cout << " in new al meu\n";
    return malloc(marime);
}
```

```
//delete supraîncărcat relativ la loc
void loc::operator delete(void *p) {
    cout << " in delete al meu\n";
    free(p);
}
```

```
int main() {
    loc *p1, *p2;
    p1 = new loc(10, 20);
    if (!p1) {
        cout << " eroare de alocare\n";
        exit(1);
    }
    p2 = new loc(-10, -20);
    if (!p2) {
        cout << " eroare de alocare\n";
        exit(1);
    }
}
```

```
p1->arata(); // 10 20
p2->arata(); // -10 -20
```

```
delete p1;
delete p2;
return 0;
}
```

```
in new al meu
in new al meu
10 20
-10 -20
in delete al meu
in delete al meu
```

Supraîncărcarea operatorilor new și delete – continuare 2

- Când new și delete sunt supraîncărcați relativ la o anumită clasă, utilizarea acestor operatori asupra oricărui alt tip de date determină efectuarea operațiilor new și delete inițiale.
- Operatorii suprapuși se aplică doar acelor tipuri pentru care au fost definiți.
- Exemplu : pentru următoarea linie în main(), new va fi efectuat implicit :

```
int *f=new float; //foloseste new implicit
```

Supraîncărcarea operatorilor new și delete – continuare 3

b new și delete redefiniți global

b Exemplu Programul 7

```
class loc {  
    int longitud, latitud;  
public:  
    loc() {}  
    loc(int lg, int lt) {  
        longitud = lg;  
        latitud = lt;  
    }  
    void arata() {  
        cout << longitud << " ";  
        cout << latitud << "\n";  
    }  
};  
  
//new global  
void *operator new(size_t marime){  
    cout << " in new al meu\n";  
    return malloc(marime);  
}
```

```
//delete global  
void operator delete(void *p) {  
    cout << " in delete al meu\n";  
    free(p);  
}  
  
int main() {  
    loc *p1, *p2;  
    p1 = new loc(10, 20);  
    if (!p1) {  
        cout << " eroare de alocare\n";  
        exit(1);  
    }  
    p2 = new loc(-10, -20);  
    if (!p2) {  
        cout << " eroare de alocare\n";  
        exit(1);  
    }  
    float *f = new float;  
    //foloseste, de asemenea, new supraîncărcat  
    if (!f) {  
        cout << " eroare de alocare\n";  
        exit(1);  
    }  
}
```

```
*f = 10.10;  
cout << *f << "\n"; //10.1
```

```
p1->arata(); //10 20  
p2->arata(); //-10 -20
```

```
delete p1;  
delete p2;  
//foloseste delete supraîncărcat  
delete f;  
return 0;  
}
```

```
in new al meu  
in new al meu  
in new al meu  
10.1  
10 20  
-10 -20  
in delete al meu  
in delete al meu  
in delete al meu
```

Supraîncărcarea operatorilor new și delete pentru matrice

- Dacă doriți să puteți alocă memorie matricelor de obiecte folosind sistemul dvs. propriu de alocare, va trebui să supraîncărcați **new** și **delete** a doua oară:

```
//aloca memorie unei matrice de obiecte.
```

```
void *operator new[](size_t marime) {
```

```
    //efectueaza alocarea.
```

```
    return pointer_la_memorie;
```

```
}
```

```
//delete pentru o matrice de obiecte.
```

```
void operator delete[](void *p) {
```

```
    //memoria liberă este indicată de p.
```

```
    //destructor apelat automat pentru fiecare element.
```

```
}
```


Supraîncărcarea operatorilor new și delete pentru matrice – continuare 1

➡ Exemplu Programul 8

```
class loc {  
    int longitud, latitud;  
public:  
    loc() { longitud = latitud = 0; }  
    loc(int lg, int lt) {  
        longitud = lg;  
        latitud = lt;  
    }  
};
```

```
void arata() {  
    cout << longitud << " ";  
    cout << latitud << "\n";  
}
```

```
void *operator new(size_t marime);  
void operator delete(void *p);
```

```
void *operator new[](size_t marime);  
void operator delete[](void *p);
```

```
};
```

```
//new supraîncărcat relativ la loc  
void *loc::operator new(size_t marime){  
    cout << " in new al meu\n";  
    return malloc(marime);  
}
```

```
//delete supraîncărcat relativ la loc  
void loc::operator delete(void *p) {  
    cout << " in delete al meu\n";  
    free(p);  
}
```

```
//new supraîncărcat relativ la loc, pentru matrice  
void *loc::operator new[](size_t marime) {  
    cout << " alocă memorie pentru matrice folosind new[]";  
    cout << " propriu\n";  
    return malloc(marime);  
}
```

```
//delete supraîncărcat relativ la loc, pentru matrice  
void loc::operator delete[](void *p) {  
    cout << "eliberează memorie din matrice folosind delete[]";  
    cout << " propriu\n";  
    free(p);  
}
```

Supraîncărcarea operatorilor new și delete pentru matrice – continuare 2

➡ Exemplu Programul 3 - continuare

```
int main() {  
    loc *p1, *p2;  
    int i;  
  
    p1 = new loc(10, 20); //aloca memorie pentru un obiect  
    if (!p1) {  
        cout << " eroare de alocare\n";  
        exit(1);  
    }  
  
    p2 = new loc[10]; // aloca memorie pentru o matrice  
    if (!p2) {  
        cout << " eroare de alocare\n";  
        exit(1);  
    }  
  
    p1->arata();  
    for (i = 0; i<10; i++) p2[i].arata();  
  
    delete p1; //elibereaza un obiect  
    return 0;  
}
```

```
in new al meu  
aloca memorie pentru matrice folosind new[] propriu  
10 20  
0 0  
0 0  
0 0  
0 0  
0 0  
0 0  
0 0  
0 0  
0 0  
0 0  
in delete al meu  
elibereaza memorie din matrice folosind delete[] propriu
```

Supraîncărcarea unor operatori speciali

- pentru `[]`, `()`, `->` : la supraîncărcare trebuie să fie funcții membre care nu sunt de tip static și nici friend

Supraîncărcarea pentru `[]`

- `[]` este considerat a fi un operator binar :

```
tip nume_clasa::operator[](int i)
{
    //...
}
```

nu este necesar ca parametrii să fie de tip `int`, dar o funcție `operator[]()` este folosită tipic pentru a asigura înscrierea indecșilor într-o matrice și de aceea este folosită, în general, o valoare întreagă:

Exemplu: `o[3]` se transformă în apelarea `operator[](3)`

(valoarea expresiei din operatorul de înscriere este transmisă funcției `operator[]()` cu parametrul său explicit ; pointerul `this` va indica spre `o`, obiectul care a generat apelarea)

```
class untip {
    int a[3];
public:
    untip(int i, int j, int k) {
        a[0] = i;
        a[1] = j;
        a[2] = k;
    }
    int operator[](int i) { return a[i]; }
};
```

```
int main() {
    untip ob(1, 2, 3);
    cout << ob[1]; //afiseaza 2
    return 0;
}
```

Supraîncărcarea pentru [] – continuare 1

- Puteți proiecta funcția operator[]() astfel încât [] să fie folosit atât în partea stângă, cât și în partea dreaptă a unei instrucțiuni de atribuire : specificați valoarea returnată de operator[]() ca referință

➡ Exemplu Programul 9

```
class untip {  
    int a[3];  
public:  
    untip(int i, int j, int k) {  
        a[0] = i;  
        a[1] = j;  
        a[2] = k;  
    }  
    int &operator[](int i) { return a[i]; }  
};
```

```
int main() {  
    untip ob(1, 2, 3);  
    cout << ob[1]; //afiseaza 2  
    cout << " ";  
  
    ob[1] = 25; //[ ] in stinga lui =  
  
    cout << ob[1]; //acum afiseaza 25  
    return 0;  
}
```

Supraîncărcarea pentru [] - continuare 2

- Avantaj al supraîncărcării operatorului []: permite o cale de utilizare sigură a indicilor matricelor în C++.

Exemplu Program 10

```
class untip {
    int a[3];
public:
    untip(int i, int j, int k) {
        a[0] = i;
        a[1] = j;
        a[2] = k;
    }
    int &operator[](int i);
};

//asigura verificarea limitei pentru untip.
int &untip::operator[](int i) {
    if (i<0 || i>2) {
        cout << "\neroare de limita\n";
        _getch();
        exit(1);
    }
    return a[i];
}
```

```
int main() {
    untip ob(1, 2, 3);
    cout << ob[1]; //afiseaza 2
    cout << " ";

    ob[1] = 25; //[] in stinga lui =
    cout << ob[1]; //acum afiseaza 25

    ob[3] = 44; //determina eroare in timpul rularii, 3 in
                // afara limitei

    return 0;
}
```

```
2 25
eroare de limita
```

În acest program, atunci când se execută instrucțiunea
ob[3]=44;

eroarea de depășire a limitelor este depistată de operator[], iar programul se încheie înainte de a se produce vreo pagubă. (În practică, pot fi apelate anumite funcții de tratare a erorilor care rezolvă condițiile de ieșire din limite; nu va trebui ca programul să se încheie.)

Supraîncărcarea pentru ()

- La supraîncărcarea operatorului de apelare a funcției (), nu se crează o nouă cale de apelare a unei funcții ci o funcție operator căreia îi poate fi transmis un număr arbitrar de parametri.
- Exemplu :

double operator()(int a, float f, char *s);

instrucțiunea

o(10, 23.34, “hi”);

este transpusă în următoarea apelare a funcției operator():

operator()(10, 23.34, “hi”);

- b** Trebuie definiți parametrii pe care doriți să îi transmiteți acelei funcții ; argumentele specificate sunt copiate în acei parametri.

Supraîncărcarea pentru () – continuare 1

- Exemplu Programul 11

```
class loc {
    int longitud, latitud;
public:
    loc() {}
    loc(int lg, int lt) {
        longitud = lg;
        latitud = lt;
    }
    void arata() {
        cout << longitud << " ";
        cout << latitud << "\n";
    }
    loc operator+(loc op2);
    loc operator()(int i, int j);
};
```

```
loc loc::operator()(int i, int j) {
    longitud = i;
    latitud = j;
    return *this;
}
```

```
loc loc::operator+(loc op2) {
    loc temp;
    temp.longitud = longitud + op2.longitud;
    temp.latitud = latitud + op2.latitud;
    return temp;
}
```

```
int main() {
    loc ob1(10, 20), ob2(1, 1);
    ob1.arata(); // 10 20
    ob1(7, 8); //poate fi executata singura
    ob1.arata(); // 7 8
    ob1 = ob2 + ob1(10, 10); //poate fi folosita in expresii
    ob1.arata(); // 11 11
    return 0;
}
```

```
10 20
7 8
11 11
```

Obs.: când supraîncărcați (), puteți folosi orice tip de parametri și returna orice tip de valoare. Aceste tipuri vor fi dictate de cerințele programelor dvs.

Supraîncărcarea pentru ->

- Când este supraîncărcat, operatorul pointer -> este considerat un operator unar:

`obiect->element;`

- *Obiect* este obiectul care activează apelarea.
- Funcția `operator->()` trebuie să returneze un pointer spre un obiect de clasă asupra căreia operează `operator->()`.
- *Element* trebuie să fie un element accesibil din cadrul obiectului returnat de `operator->()`.

```
class clasamea {  
public:  
    int i;  
    clasamea *operator->() { return this; }  
};
```

```
int main() {  
    clasamea ob;
```

```
    ob->i = 10; //identic cu ob.i;  
    cout << ob.i << " " << ob->i; //10 10
```

```
    return 0;  
}
```

(echivalența dintre `ob.i` și `ob->i` când `operator->()` returnează pointerul `this`)

Supraîncărcarea pentru operatorul virgulă

Operatorul virgulă poate fi supraîncărcat cu orice operație (binară). Totuși, dacă doriți ca virgula supraîncărcată să acționeze într-un mod similar cu operația sa normală, ea trebuie să renunțe la valoarea din termenul său stâng și să atribuie valoarea operației termenului din dreapta. Într-o listă separată prin virgulă trebuie să se renunțe la toți termenii, în afară de cel din extrema dreaptă (felul în care virgula lucrează implicit în C++). Exemplu Program 12

```
class loc {
    int longitud, latitud;
public:
    loc() {}
    loc(int lg, int lt) {
        longitud = lg;
        latitud = lt;
    }
    void arata() {
        cout << longitud << " ";
        cout << latitud << "\n";
    }
    loc operator,(loc op2);
    loc operator+(loc op2);
};
```

```
loc loc::operator,(loc op2) {
    loc temp;
    temp.longitud = op2.longitud;
    temp.latitud = op2.latitud;
    cout << op2.longitud << " " << op2.latitud << "\n";
    return temp;
}

loc loc::operator+(loc op2) {
    loc temp;
    temp.longitud = longitud + op2.longitud;
    temp.latitud = latitud + op2.latitud;
    return temp;
}
```

Supraîncărcarea pentru operatorul virgulă – continuare 1

Operandul din partea dreaptă este transmis prin `this`, iar valoarea sa este eliminată de către funcția `operator,()`. Funcția returnează valoarea din dreapta operației, adică virgula supraîncărcată se comportă similar cu operația sa implicită.

(Dacă se dorește supraîncărcarea virgulei pentru a face altceva, vor trebui modificate aceste două caracteristici).

```
int main() {
    loc ob1(10, 20), ob2(5, 30), ob3(1, 1);

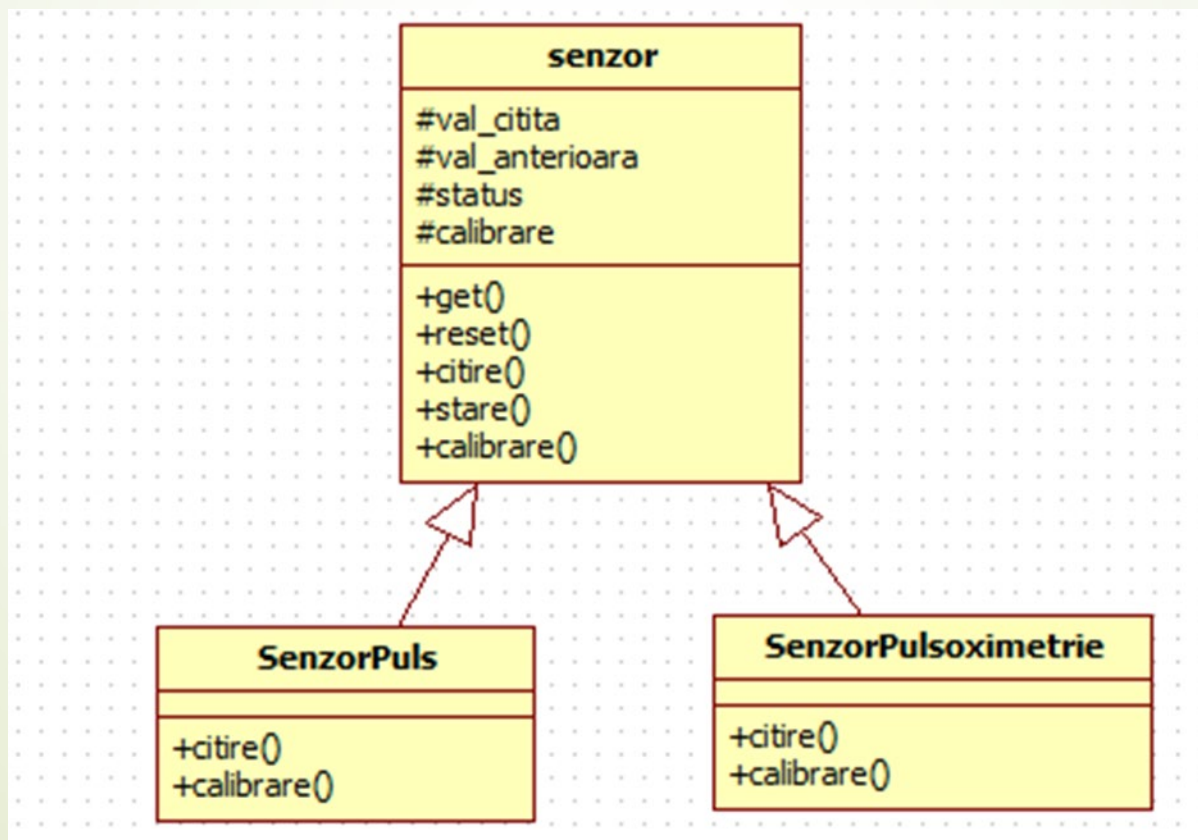
    ob1.arata(); //10 20
    ob2.arata(); //5 30
    ob3.arata(); //1 1
    cout << "\n";

    ob1 = (ob1, ob2 + ob2, ob3); //10 60
                                //1 1
    ob3.arata(); //afiseaza 1 1, valoarea lui ob3

    return 0;
}
```

Capitolul 6. Moștenirea

Moștenirea este una dintre pietrele de temelie ale POO - permite crearea clasificărilor ierarhice : se poate construi o clasă generală care definește trăsăturile comune ale unui set de elemente corelate (aceasta poate fi apoi moștenită de alte clase, particulare, fiecare adăugând doar acele elemente care îi sunt proprii)



Controlul accesului la clasa de bază

```
class nume_clasa_derivata : acces nume_clasa_de_baza{  
    //corpul clasei  
};
```

➤ **acces:** public, private sau protected

➤ **public:** toți membrii publici ai clasei devin membrii publici ai clasei derivate, iar toți membrii **protected** (protejați) ai bazei devin membrii protejați ai clasei derivate (elementele private (particulare) ale bazei rămân particulare pentru bază și nu sînt accesibile membrilor clasei derivate)

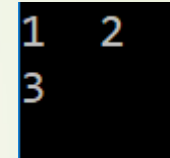
Controlul accesului la clasa de bază – continuare 1

exemplu **Programul 1** - obiectele de tip derivat pot fi accesibile direct membrilor publici din baza

```
class baza {  
    int i, j;  
public:  
    void pune(int a, int b) { i = a; j = b; }  
    void arata() { cout << i << " " << j << "\n"; }  
};
```

```
class derivat : public baza {  
    int k;  
public:  
    derivat(int x) { k = x; }  
    void aratak() { cout << k << "\n"; }  
};
```

```
int main() {  
    derivat ob(3);  
    ob.pune(1, 2); //acces la membrul bazei  
    ob.arata(); // acces la membrul bazei  
  
    ob.aratak(); //foloseste membrul clasei derivate  
    return 0;  
}
```



```
1 2  
3
```

Controlul accesului la clasa de bază – continuare 2

Private: toți membrii publici și protejați ai clasei de bază devin membri privați ai clasei derivate:

exemplu Programul 2 nu va fi compilat deoarece atât pune(), cât și arata() sunt elemente particulare pentru derivat.

```
class baza {
    int i, j;
public:
    void pune(int a, int b) { i = a; j = b; }
    void arata() { cout << i << " " << j << "\n"; }
};

//elementele publice din baza sînt particulare in derivat.
class derivat : private baza {
    int k;
public:
    derivat(int x) { k = x; }
    void aratak() { cout << k << "\n"; }
};
```

```
int main() {
    derivat ob(3);
    ob.pune(1, 2); //eroare, nu poate avea acces la pune()
    ob.arata(); // eroare, nu poate avea acces la arata()

    return 0;
}
```

membrii publici sau protejați ai bazei devin membri particulari ai clasei derivate: sunt încă accesibili membrilor clasei derivate, dar și secțiunilor din program care nu sunt membri nici ai clasei de bază, nici ai celei derivate.

Controlul accesului la clasa de bază – continuare 3

Protected :

- acel membru nu este accesibil altor elemente ale programului care nu sunt membri ai clasei.
- accesul la un membru protejat este același ca și accesul la un membru particular - el este accesibil doar altor membri din aceeași clasă cu a sa.
- **Excepție:** când membrul protejat este moștenit.
- **Exemplu Program3**

```
class baza {  
protected:  
    int i, j; //particulari pentru baza, dar accesibili pentru  
            // derivat  
public:  
    void pune(int a, int b) { i = a; j = b; }  
    void arata() { cout << i << " " << j << "\n"; }  
};
```

```
class derivat : public baza {  
    int k;  
public:  
    //derivat poate sa aiba acces la i si j din baza  
    void punek() { k = i*j; }  
    void aratak() { cout << k << "\n"; }  
};  
  
int main() {  
    derivat ob;  
  
    ob.pune(2, 3); //ok, cunoscut lui derivat  
    ob.arata(); // ok, cunoscut lui derivat  
                //2 3  
    ob.punek();  
    ob.aratak(); //6  
  
    return 0;  
}
```

```
2 3  
6
```


Controlul accesului la clasa de bază – continuare 4

Când o clasă derivată este folosită ca o clasă de bază pentru altă clasă derivată, atunci orice membru protejat al clasei de bază inițiale care este moștenită (ca public) de către prima clasă derivată poate fi moștenit de asemenea, ca protected și de cea de-a doua clasă derivată :
exemplu Programul4 - derivat2 poate, într-adevăr, să aibă acces la i și j.

```
class baza {
protected:
    int i, j;
public:
    void pune(int a, int b) { i = a; j = b; }
    void arata() { cout << i << " " << j << "\n"; }
};
```

```
//i si j mosteniti ca protejati.
class derivat1 : public baza {
    int k;
public:
    void pune() { k = i*j; } //legal
    void aratak() { cout << k << "\n"; }
};
```

```
//i si j mosteniti indirect prin derivat1.
class derivat2 : public derivat1 {
    int m;
public:
    void punem() { m = i - j; } //posibil
    void aratam() { cout << m << "\n"; }
};

int main() {
    derivat1 ob1;
    derivat2 ob2;
    ob1.pune(2, 3);
    ob1.arata(); //2 3
    ob1.punek();
    ob1.aratak(); //6
    ob2.pune(3, 4);
    ob2.arata(); //3 4
    ob2.punek();
    ob2.punem();
    ob2.aratak(); // 12
    ob2.aratam(); // -1
    return 0;}
```

```
2 3
6
3 4
12
-1
```


Controlul accesului la clasa de bază – continuare 5

Dacă, totuși, baza ar fi moștenită ca `private`, atunci toți membrii din baza ar deveni membri `private` ai lui `derivat1`, ceea ce înseamnă că ei nu ar fi accesibili lui `derivat2` (totuși, `i` și `j` ar continua să fie accesibili lui `derivat1`) : exemplu Program5 care conține erori (și nu va fi

```
class baza {compilat)
protected:
    int i, j;
public:
    void pune(int a, int b) { i = a; j = b; }
    void arata() { cout << i << " " << j << "\n"; }
};
```

//acum, toate elementele din baza sînt particulare in
// derivat1.

```
class derivat1 : private baza {
    int k;
public:
    void punek() { k = i*j; } //ok
    void aratak() { cout << k << "\n"; }
};
```

Observație : chiar dacă baza este moștenită ca fiind `private` în `derivat1`, `derivat1` are încă acces la elementele `public` și `protected` din baza. Totuși, el nu poate transmite mai departe acest privilegiu.

//accesul la `i`, `j`, `pune()` si `arata()` nu este mostenit.

```
class derivat2 : public derivat1 {
    int m;
public:
    //ilegal deoarece i si j sînt particulari in derivat1
    void punem() { m = i - j; } //eroare
    void aratam() { cout << m << "\n"; }
};
```

```
int main() {
    derivat1 ob1;
    derivat2 ob2;
```

`ob1.pune(1, 2);` //eroare, nu poate folosi `pune()`
`ob1.arata();` //eroare, nu poate folosi `arata()`

`ob2.pune(3, 4);` //eroare, nu poate folosi `pune()`
`ob2.arata();` //eroare, nu poate folosi `arata()`

```
return 0;
}
```

Moștenirea protected a clasei de bază

- Este posibil ca o clasă de bază să fie moștenită ca protected : toți membrii publici și protejați ai clasei de bază devin membri protejați ai clasei derivate : exemplu Programul6

```
class baza {
protected:
    int i, j; //particulari pentru baza, dar accesibili
              // pentru derivat
public:
    void puneij(int a, int b) { i = a; j = b; }
    void arataij() { cout << i << " " << j << "\n"; }
};

//mosteneste baza ca protected.
class derivat : protected baza {
    int k;
public:
    //derivat poate avea acces la i si j din baza si la
    // puneij(.)
    void punek() { puneij(10, 12); k = i*j; }

    //aici poate avea acces la arataij()
    void aratatot() { cout << k << " "; arataij(); }
};
```

```
int main() {
    derivat ob;

    // ob.puneij(2,3); //illegal, puneij() este membru protejat
                      // al lui derivat
    ob.punek(); // ok, membru public al lui derivat
    ob.aratatot(); // ok, membru public al lui derivat
                  //120 10 12
    //ob.arataij(); //illegal, arataij() este membru protejat
    // al lui derivat

    return 0;
}
```

120 10 12

chiar dacă puneij(.) și arataij() sînt membri publici în baza, ei devin membri protejați în derivat atunci cînd aceasta îi moștenește folosind specificatorul de acces protected, deci ele nu vor fi accesibile pentru main()

Moștenirea din clase de bază multiple

b Este posibil pentru o clasă derivată să moștenească două sau mai multe clase de bază. exemplu Programul 7 (derivat moștenește atât baza1, cât și baza2)

➡ Deci pentru a moșteni mai mult decât o clasă :

- folosiți o listă separată prin virgulă
- utilizați un specificator de acces pentru fiecare bază moștenită

```
class baza1 {  
protected:  
    int x;  
public:  
    void aratax() { cout << x << "\n"; }  
};
```

```
class baza2 {  
protected:  
    int y;  
public:  
    void aratay() { cout << y << "\n"; }  
};
```

```
//mostenire din multiple clase de baza.  
class derivat : public baza1, public baza2 {  
public:  
    void pune(int i, int j) { x = i; y = j; }  
};
```

```
int main() {  
    derivat ob;  
    ob.pune(10, 20); //asigurat prin derivat  
    ob.aratax(); //din baza1  
                // afiseaza 10  
    ob.aratay(); //din baza2  
                //afiseaza 20  
    return 0;  
}
```

10
20

Constructori, destructori și moștenire

Exemplu Program 8

```
class baza {
public:
    baza() { cout << "construieste baza.\n"; }
    ~baza() { cout << "distruge baza.\n"; _getch(); }
};

class derivat : public baza {
public:
    derivat() { cout << "construieste derivat.\n"; }
    ~derivat() { cout << "distruge derivat.\n";
        _getch(); }
};

int main() {
    derivat ob;
    //nu face nimic, decat sa construiasca si sa distruga ob
    _getch();
    return 0;
}
```

```
construieste baza.
construieste derivat.
distruge derivat.
distruge baza.
```

În general: Când este creat un obiect al unei clase derivate, dacă pentru clasa de bază există un constructor, el va fi apelat primul, urmat de constructorul clasei derivate. Când este distrus un obiect al clasei derivate, primul este apelat destructorul său, urmat de cel al clasei de bază, dacă există.

Deci funcțiile constructor sunt executate în ordinea derivării, iar funcțiile destructor în ordinea inversă derivării.

Motivație : *Deoarece o clasă de bază nu știe nimic despre nici o clasă derivată, este necesar ca orice inițializare să se facă independent și anterior oricărei cerințe anterioare de inițializare efectuate de clasa derivată (de aceea ea trebuie executată prima). Tot așa, este logic ca funcțiile destructor să fie executate în ordinea inversă derivării. Deoarece o clasă de bază conține o clasă derivată, distrugerea obiectului de bază implică distrugerea obiectului derivat. De aceea, destructorul derivat trebuie să fie apelat înainte ca obiectul să fie distrus complet.*

Constructori, destructori și moștenire – continuare 1

- caz de moștenire multiplă (atunci când o clasă derivată devine clasă de bază pentru altă clasă derivată exemplu **Programul 9**)

```
class baza {
public:
    baza() { cout << "construieste baza.\n"; }
    ~baza() { cout << "distruge baza.\n"; _getch(); }
};

class derivat1 : public baza {
public:
    derivat1() { cout << "construieste derivat1.\n"; }
    ~derivat1() { cout << "distruge derivat1.\n"; _getch(); }
};

class derivat2 : public derivat1 {
public:
    derivat2() { cout << "construieste derivat2.\n"; }
    ~derivat2() { cout << "distruge derivat2.\n"; _getch(); }
};

int main() {
    derivat2 ob;
    //construieste si distruge ob
    return 0;
}
```

```
construieste baza.
construieste derivat1.
construieste derivat2.
distruge derivat2.
distruge derivat1.
distruge baza.
```

Constructorii, destructorii și moștenire - continuare

- b caz de clase de bază multiple exemplu **Programul 10** (constructorii sunt apelați în ordinea derivării - de la stînga la dreapta - după cum s-a specificat în lista de moștenire a clasei **derivat**. Destructorii sunt apelați în ordine inversă - de la dreapta la stînga.)

```
class baza1 {
public:
    baza1() { cout << "construieste baza1.\n"; }
    ~baza1() { cout << "distruge baza1.\n"; _getch(); }
};

class baza2 {
public:
    baza2() { cout << "construieste baza2.\n"; }
    ~baza2() { cout << "distruge baza2.\n"; _getch(); }
};
```

```
class derivat : public baza1, public baza2 {
public:
    derivat() { cout << "construieste derivat.\n"; }
    ~derivat() { cout << "distruge derivat.\n"; _getch(); }
};

int main() {
    derivat ob;
    //construieste si distruge ob

    return 0;
}
```

```
construieste baza1.
construieste baza2.
construieste derivat.
distruge derivat.
distruge baza2.
distruge baza1.
```


Pointeri către tipuri derivate

- Un pointer către un obiect din clasa de bază poate fi folosit ca pointer spre oricare obiect al oricărei clase derivate
- *Atenție ! Reciproca nu e adevărată !* De asemenea pointerii din clasa de bază folosiți pentru clasa derivată nu pot indica decât spre membrii de tip derivat importați din clasa de bază.

Exemplu **Program20**

Pointeri către tipuri derivate – continuare 1

- pentru a rezolva inconvenientul se poate converti tipul pointerului (deși nu se recomandă) :

```
((derivat *) b) -> set_j(3);
```

```
cout << ((derivat *) b) -> ret_j();
```

- *Atenție la aritmetica pointerilor !* Aceasta este relativă la tipul de bază de aceea **Program21** nu va funcționa corect.
- Pointerii din clasa de bază către tipuri derivate sunt extrem de **utili** la folosirea *funcțiilor virtuale* pentru asigurarea *polimorfismului* (se va vedea în continuare).

```
class baza { // clasa de baza
    int i;
public:
    void set_i(int val) { i = val; }
    int ret_i() { return i; }
};

class derivat : public baza { // clasa derivata
    int j;
public:
    void set_j(int val) { j = val; }
    int ret_j() { return j; }
};
```

```
int main()
{
    baza *b; // pointer catre obiect baza
    derivat d[2]; // obiecte derivate

    b = d;

    d[0].set_i(1);
    d[1].set_i(2);

    cout << b->ret_i() << " ";
    b++; // atentie ! aici incrementarea e relativ la baza,
        // nu la clasa derivata !
    cout << b->ret_i(); // de aceea aici nu se obtine
                        // ce am dorit !

    _getch();
    return 0;}
```


Pointeri către membrii clasei

C++ permite generarea unui tip special de pointeri care indică generic spre un membru al unei clase, nu către un anumit exemplar al acelui membru dintr-un obiect : *pointer către un membru al clasei* sau pe scurt **pointer la membru**.

- Nu este același lucru cu un pointer normal. Un astfel de pointer asigură doar o poziționare (un *offset*) într-un obiect din clasa membrului, unde poate fi găsit acel membru.
- Pentru a avea **acces** la membrul unei clase prin intermediul unui pointer de acest tip, vor trebui folosiți operatorii speciali pentru pointeri la membri: `.*` și `->*`.

Exemplu **Program22 - pentru pointeri la membri**

```
class c1 {  
public:  
    c1(int i) { val = i; }  
    int val;  
    int val_dubla() { return val + val; }  
};
```

```
Iata date obtinute cu pointeri la membri :1 2  
Acum folosim pointeri la functii membre : 2 4
```

```
int main()  
{  
    int(c1.*)*date; // pointer la o data membru  
    int (c1::*func)(); // pointer la o functie membru  
    c1 ob1(1), ob2(2); // crearea a doua obiecte  
  
    date = &c1::val; // obtinem offsetul pentru val  
    func = &c1::val_dubla; // iar aici obtinem offsetul pentru functia val_dubla()  
    // acum putem profita de pointerii la membri obtinuti mai sus  
    cout << "Iata date obtinute cu pointeri la membri :";  
    cout << ob1.*date << " " << ob2.*date << "\n";  
    cout << "Acum folosim pointeri la functii membre : ";  
    cout << (ob1.*func)() << " " << (ob2.*func)() << "\n";  
    return 0;  
}
```

Pointeri către membrii clasei – continuare 1

Dacă dorim să utilizăm un *pointer spre un obiect* se folosește operatorul ->

Exemplu **Program23** (->*).

```
class po {
public:
    po(int i) { val = i; }
    int val;
    int val_dubla() { return val + val; }
};

int main()
{
    int po::*date; // pointer la o data membra
    int (po::*func) (); // pointer la o functie membru
    po ob1(1), ob2(2); // se creaza obiecte initializate
    po *p1, *p2; // pointeri la obiecte

    p1 = &ob1;
    p2 = &ob2; // pointerii spre obiectele create

    date = &po::val; // offset pentru val
    func = &po::val_dubla; // offset pentru functie

    cout << "Valorile sint : ";
    cout << p1->*date << " " << p2->*date << "\n";
    cout << "Valorile dublate : ";
    cout << (p1->*func)() << " " << (p2->*func)() << "\n";
    return 0;
}
```

```
Valorile sint : 1 2
Valorile dublate : 2 4
```

Pointeri către membrii clasei – continuare 2

- Pointerii la membri sunt *diferiți* față de pointerii spre elemente efective ale vreunui obiect:

```
int po::*d;
```

```
int *p;
```

```
po ob;
```

```
p=&ob.val; // aceasta e o adresa a unui element efectiv al
```

```
    //      unui obiect (val)
```

```
d=&po::val;// pe cind acesta e un offset (generic) pentru val
```