8. MECANISMUL MOȘTENIRII. CONSTRUIREA IERARHIILOR DE CLASE

Moștenirea (*inheritance*) reprezintă un mecanism sau o tehnică de programare prin care se pot reutiliza și extinde clasele existente; avantajul constă în faptul că se poate extinde codul scris, fără a mai fi necesară rescrierea celui original. În acest capitol se vor aborda următoarele teme de interes: specificatorii de acces, utilizarea membrilor de tip **protected**, funcțiile constructor, destructor și mecanismul moștenirii, mecanismul moștenirii multiple, prezentarea claselor de bază virtuale.

8.1. Specificatorii de acces

În C++, mecanismul de moștenire se realizează prin declararea unei (unor) clase de bază din care se derivă apoi o altă clasă (alte clase). Clasele derivate "moștenesc" structurile de date și funcțiile clasei de bază. În plus, în clasele derivate pot fi adăugate noi funcții membre, iar membrii moșteniți pot fi modificați. Clasa care este utilizată pentru a deriva din ea clase noi se numește *clasă de bază*. La rândul ei, o clasă derivată poate fi folosită drept clasă de bază pentru a deriva din ea alte clase. Prin urmare se pot construi *ierarhii de clase*, unde fiecare clasă servește drept "părinte" sau "rădăcină" pentru o nouă clasă. Sintaxa pentru declararea unei clase derivate dintr-o clasă de bază este următoarea:

```
class nume_clasa_derivata : specificator_de_acces   nume_clasa_de_baza {
        //   Implementarea clasei
};
```

unde specificator_de_acces poate fi unul din cuvintele cheie: **public**, **private** sau **protected**. Specificatorul de acces determină modul în care elementele clasei de bază sunt moștenite de clasa derivată. Când el este de tip **public**, *toți membrii publici ai clasei de bază devin membri publici ai clasei derivate*. Dacă specificatorul de acces este de tip **private**, *toți membrii publici ai clasei de bază devin membri privați ai clasei derivate*, dar ei pot fi accesibili funcțiilor membre ale clasei derivate. În ambele cazuri, membrii de tip **private** din clasa de bază rămân privați și în clasa derivată, și nu pot fi accesibili claselor derivate. Exemplele care urmează evidențiază aceste aspecte. Astfel, exemplul de mai jos conține o clasă de bază și una derivată, care moștenește toți membrii de tip **public**:

```
// Program P8_1a.CPP  Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {          // Va fi utilizată pentru crearea clasei derivate denumită DERIVATA
        int  x;       // Variabilă de tip private
public:
        void  init_x (int n) { x = n; }
        void  afiseaza_x() { cout << x << '\n'; }
};

class DERIVATA : public BAZA {
// Clasa DERIVATA moșteneste toți membrii publici din clasa BAZA
        int  y;       // Variabilă de tip private
public:
        void  init_y (int n) { y = n; }
        void  afiseaza_y() { cout << y << '\n'; }
};
void  main (void)                  // Programul principal
{       BAZA OB_B;                 // Se creează obiectul OB_B
```

```
        OB_B.init_x (10);           // Are acces la membrul din clasa de bază
        OB_B.afiseaza_x ();         // Are acces la membrul din clasa de bază

        DERIVATA OB_D;              // Se creează obiectul OB_D
        OB_D.init_x (100);          // Are acces la membrul din clasa de bază
        OB_D.init_y (200);          // Are acces la membrul din clasa derivată
        OB_D.afiseaza_x ();         // Are acces la membrul din clasa de bază
        OB_D.afiseaza_y ();         // Are acces la membrul din clasa derivată
}
```

Așa cum se vede, deoarece în acest program, clasa de bază este moștenită ca **public**, membrii de tip **public** ai ei - funcțiile init_x() și afiseaza_x() - devin membri publici ai clasei derivate. Faptul că o clasă derivată moștenește membrii publici din clasa de bază, nu implică și faptul că ea are acces la membrii de tip **private** ai clasei de bază. Acest aspect este evidențiat în exemplul următor:

```
// Program P8_1b.CPP  Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {            // Va fi utilizată pentru crearea clasei derivate denumită DERIVATA
        int  x;         // Variabilă de tip private
public:
        void  init_x (int n) { x = n; }
        void  afiseaza_x() { cout << x << '\n'; }
};

class DERIVATA : public BAZA {
// Clasa DERIVATA moșteneste toți membrii publici din clasa BAZA
        int  y;         // Variabilă de tip private
public:
        void  init_y (int n) { y = n; }
        void afiseaza_suma() { cout << x + y << '\n'; }
        // Aceasta funcție nu poate avea acces la membrul privat x al clasei de bază
        // Apare o eroare de forma `BAZA :: x` is not accessible
        void  afiseaza_y() { cout << y << '\n'; }
};

void  main (void)               // Programul principal
{       BAZA OB_B;              // Se creează obiectul OB_B
        OB_B.init_x (10);       // Are acces la membrul din clasa de bază
        OB_B.afiseaza_x ();     // Are acces la membrul din clasa de bază

        DERIVATA OB_D;          // Se creează obiectul OB_D
        OB_D.init_x (100);      // Are acces la membrul din clasa de bază
        OB_D.init_y (200);      // Are acces la membrul din clasa derivată
        OB_D.afiseaza_x ();     // Are acces la membrul din clasa de bază
        OB_D.afiseaza_y ();     // Are acces la membrul din clasa derivată
        OB_D.afiseaza_suma()    // Apare o eroare de forma `BAZA :: x` is not accessible
}
```

Dacă, în acest exemplu, clasa derivată încearcă (prin intermediul funcției membre afiseaza_suma()) să obțină accesul la variabila x, care este un membru de tip **private** al clasei de bază, se semnalează eroarea precizată în comentariu, deoarece un membru de tip **private** al clasei de bază rămâne privat, indiferent de tipul moștenirii.

În programul P8_1a.CPP se consideră acum că membrii clasei de bază vor fi moșteniți ca privați în clasa derivată. Această schimbare generează erorile care sunt semnalate în comentariile programului de mai jos:

```
// Program P8_1c.CPP   Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {          // Va fi utilizată pentru crearea clasei derivate denumită DERIVATA
      int  x;         // Variabilă de tip private
public:
      void init_x (int n) { x = n; }
      void afiseaza_x() { cout << x << '\n'; }
};

class DERIVATA : private BAZA {   // Clasa DERIVATA mostenește toți membrii de tip public
                                  // din clasa BAZA ca membri de tip private
      int  y;         // Variabilă de tip private
public:
      void init_y (int n) { y = n; }
      void afiseaza_y() { cout << y << '\n'; }
};
void  main (void)                 // Programul principal
{     BAZA OB_B;                  // Se creează obiectul OB_B de tip BAZA
      OB_B.init_x (10);           // Are acces la membrul din clasa de bază
      OB_B.afiseaza_x ();         // Are acces la membrul din clasa de bază

      DERIVATA OB_D;              // Se creează obiectul OB_D de tip DERIVATA
      OB_D.init_x (10);           // Eroare: 'BAZA::init_x(int)' is not accessible
      OB_D.init_y (20);           // Are acces la membrul din clasa derivată
      OB_D.afiseaza_x ();         // Eroare: 'BAZA::afiseaza_x()' is not accessible
      OB_D.afiseaza_y ();         / Are acces la membrul din clasa derivată
}
```

Comentariile ilustrează faptul că funcțiile init_x() și afiseaza_x() devin private pentru clasa derivată, și deci nu sunt accesibile. Trebuie reținut că aceste funcții rămân de tip **public** în interiorul clasei de bază, indiferent de modul în care sunt moștenite de clasele derivate. Acest lucru se observă și din faptul că obiectul OB_B de tip BAZA are acces la aceste funcții, în schimb obiectul OB_D de tip DERIVATA nu are acces la aceste funcții.

O versiune a programului anterior care să elimine erorile semnalate este următoarea, în care membrii publici ai clasei de bază moșteniți ca membri privați, sunt tratați într-adevăr ca membri privați ai clasei derivate:

```
Program P8_1d.CPP  Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {                  // Va fi utilizată pentru crearea clasei derivate DERIVATA
      int  x;                 // Variabilă de tip private
public:
      void init_x(int n) { x = n; }
      void afiseaza_x() { cout << x << '\n'; }
};
```

```
class DERIVATA : private BAZA {          // Clasa DERIVATA moșteneste toți membrii de tip
                                         // public din clasa BAZA ca membri de tip private
        int  y;                          // Variabilă de tip private
public:
        void init_xy(int n, int m) { init_x(n); y = m; }
   // Funcția init_x(int) este accesibilă prin intermediul unei funcții membre a clasei derivate
        void afiseaza_xy() { afiseaza_x(); cout << y << '\n'; }
   // Funcția afiseaza_x() este accesibilă prin intermediul unei funcții membre a clasei derivate
};

void  main (void)                        // Programul principal
{       BAZA OB_B;                       // Se creează obiectul OB_B de tip BAZA
        OB_B.init_x(10);
        OB_B.afiseaza_x();

        DERIVATA OB_D;                   // Se creează obiectul OB_D de tip DERIVATA
        OB_D.init_xy(10, 20);
        OB_D.afiseaza_xy();
}
```

În acest caz, accesul la funcțiile init_x(**int**) și afișează_x() se face din *interiorul* clasei derivate, lucru perfect legal, deoarece acestea sunt membri de tip **private** ai acestei clase.

8.2. Utilizarea membrilor de tip protected

Așa cum am subliniat în secțiunea precedentă, o clasă derivată nu poate avea acces la membrii de tip **private** din clasa de bază. Aceasta înseamnă că dacă o clasă derivată vrea să obțină accesul la un membru al clasei de bază, atunci acel membru trebuie declarat **public**. Totuși, pot exista situații când vrem să păstrăm un membru al clasei de bază ca privat, dar să permitem totuși unei clase derivate să aibă acces la el. Pentru a realiza acest lucru, C++ folosește specificatorul de acces **protected**. Acest specificator este echivalent cu cel de tip **private**, cu singura excepție că *membrii de tip* **protected** *ai clasei de bază sunt accesibili membrilor oricărei clase derivate din clasa de bază*. Ei nu sunt accesibili în afara claselor de bază și derivate. Specificatorul de acces **protected** poate apare oriunde în declarația clasei, deși se recomandă utilizarea lui după declararea membrilor de tip **private** și înaintea celor de tip **public**. Forma generală este următoarea:

```
class nume_clasa  {
private:                 // Opțional
        // Membrii de tip private
protected:
        // Membrii de tip protected
public:
        // Membrii de tip public
};
```

Când un membru de tip **protected** al unei clase de bază este moștenit ca **public**, el devine un membru de tip **protected** în clasa derivată. Dacă în schimb el este moștenit ca **private**, atunci devine **private** în clasa derivată. Menționăm că specificatorul de acces **protected** poate fi folosit și cu structuri și uniuni.

Programul următor arată modul de acces la membrii unei clase de tip **public**, **private** și **protected**:

```
// Program P8_2.CPP  Exemplu de acces la membri unei clase de tip public, private si protected
# include <iostream.h>
```

```
class MOSTRA {
        int  a;                 // Implicit, a este de tip private
protected:
        int b;                  // Relativ la clasa MOSTRA, b este tot de tip private
public:
        int c;
        MOSTRA (int n, int m) { a = n; b = m; }
        int obtine_a() { return  a; }
        int obtine_b() { return  b; }
};
void  main (void)               // Programul principal
{       MOSTRA OB (10, 20);
        OB.b = 99;              // Eroare: `MOSTRA :: b' is not accessible
                                // Observație: b este de tip protected și deci private
        OB.c = 30;              // OK!, variabila c este de tip public
        cout << OB.obtine_a() << ' ';
        cout << OB.obtine_b() << ' ' << OB.c << '\n';
}
```

În linia de comentariu se remarcă faptul că nu se poate obține accesul la variabila b, deoarece aceasta este de tip **protected** și, prin urmare, de tip **private** în clasa MOSTRA.

Programul următor arată ce se întâmplă când membrii de tip **protected** sunt moșteniți ca publici:

```
// Program P8_3.CPP  Mecanismul moștenirii. Clase derivate
// Programul următor arată ce se întâmplă când membrii de tip protected sunt moșteniți ca public
# include <iostream.h>
class BAZA {            // Va fi utilizată pentru crearea clasei derivate DERIVATA
protected:
        int  a, b;      // a și b sunt de tip private în clasa de bază, dar sunt accesibili clasei derivate
public:
        void init_ab (int n, int m) { a = n; b = m; }
        void afiseaza_ab() { cout << a << ' ' << b << '\n'; }
};
class DERIVATA : public BAZA {          // Clasa DERIVATA moștenește toți membrii de tip
                                        // public din clasa BAZA ca membri de tip public
        int  c;                         // Variabila c este privată în clasa DERIVATA
public:
        void init_c (int n) { c = n; }
        void afiseaza_abc() { cout << a << ' ' << b << ' ' << c << '\n'; }
                                // Acesată funcție are acces la variabilele a și b din clasa BAZA
};
void  main (void)                       // Programul principal
{       BAZA OB_B;                      // Se creează obiectul OB_B de tip BAZA
        OB_B.init_ab(10,20);    OB_B.afiseaza_ab();     // Se va afișa 10 20
        DERIVATA OB_D;          // Se creează obiectul OB_D de tip DERIVATA
        OB_D.init_ab(10, 20);
        OB_D.init_c(30);
        OB_D.afiseaza_abc();    // Se va afișa 10 20 30
```

```
        //cout << OB_B.a << " " << OB_B.b << endl;
        //cout << OB_D.a << " " << OB_D.b << endl;
        // Variabilele a și b nu sunt accesibile ca în instrucțiunile anterioare deoarece ele sunt de tip
        // private atât pentru clasa de bază cât și pentru cea derivată
}
```

Întrucât variabilele a și b sunt declarate **protected** în clasa de bază, și sunt moștenite ca **public** de clasa derivată, ele sunt disponibile funcțiilor membre din clasa derivată. Totuși, în exteriorul celor două clase, a și b sunt de tip **private** și deci sunt inaccesibile.

8.3. Construirea ierarhiilor de clase

Mecanismul moștenirii este important deoarece ne oferă o tehnică de extindere a codului existent. Vom explora aceste tehnici, elaborând mai multe programe care utilizează structuri ierarhice.
Un prim exemplu se referă la prelucarea unor numere reale în dublă precizie. Pentru început vom crea o clasă denumită SIGMA care realizează suma numerelor introduse:

```
// Program P8_4a.CPP  Programul calculează suma unor numere reale
# include <stdio.h>
class SIGMA {
public:
        double suma;
        SIGMA (void)        { suma = 0.0; }                 // Constructorul clasei SIGMA
        void introduce (double d) { suma += d; }
        void afiseaza (void) { printf("Suma este %f\n", suma); }
};
void main (void)                    // Programul principal
{       SIGMA X;                    // Se creează obiectul X de tip SIGMA
        X.introduce(17.0);          // Se introduc numerele de însumat
        X.introduce(42.0);
        X.introduce(55.0);
        X.afiseaza();               // Se afișează suma numerelor introduse
}
```

La execuția acestui program, funcția afiseaza() produce pe ecran următorul rezultat:

```
        Suma este 114.000000
```

Dacă se dorește media aritmetică a numerelor introduse se procedează ca în exemplul următor:

```
// Program P8_4b.CPP  Mecanismul moștenirii. Structuri ierarhice
//  Programul calculează media aritmetică a unor numere reale
# include <stdio.h>
class SIGMA {
public:
        double suma;
        SIGMA (void)    { suma = 0.0; }             // Constructorul clasei SIGMA
        virtual void introduce (double d) { suma += d; }
        void afiseaza (void) { printf("Suma este %f\n", suma); }
};
// Cod suplimentar scris în clasa derivată MEDIA, care modifică clasa SIGMA
class MEDIA : public SIGMA {
public:
```

```
        int n;
        MEDIA (void)  { n = 0; }                    // Constuctorul clasei derivate MEDIA
        void introduce (double d) { SIGMA :: introduce(d); n++; }
        void afiseaza_media (void) {
        double media;
        media = n ? suma/n : 0;
        printf ("Media este %f\n", media);
        }
};
void main (void)                                    // Programul principal
{       MEDIA X;                                    // Se creează obiectul X de tip MEDIA
        X.introduce (17.0);                         // Se introduc numerele de însumat
        X.introduce (42.0);
        X.introduce (55.0);
        X.afiseaza();                               // Se afișează suma numerelor
        X.afiseaza_media();                         // Se afișează media numerelor
}
```

La execuția programului, pe ecran se va afișa:

```
        Suma este 114.000000
        Media este  38.000000
```

Se remarcă modul de definire a claselor SIGMA și MEDIA:

```
        calss SIGMA {
        public:
                double suma;
                // Funcții de tip public
        };
        class MEDIA : public SIGMA {
        public:
                int n;
                // Funcții de tip public
        };
```

Operatorul "**:**" permite "derivarea" unei clase dintr-o clasă de bază. În exemplul anterior se folosește clasa SIGMA drept clasă de bază pentru a deriva din ea clasa MEDIA. Singura adăugare la programul anterior a reprezentat-o noua clasă derivată și un apel suplimentar la funcția printf(), efectuat în programul principal. Noua clasă, MEDIA, are câțiva membri suplimentari: ea conține variabila întreagă n, funcția constructor MEDIA() și funcția afiseaza_media(); în plus, ea modifică funcția introduce() din clasa de bază. Clasa MEDIA a fost derivată din clasa SIGMA cu specificatorul **public** și, prin urmare, a moștenit toate variabilele și funcțiile acestei clase. Datorită mecanismului de moștenire nu mai este necesară rescrierea codului care a efectuat operația de adunare și de afișare a rezultatului. Un aspect aparte îl reprezintă numai reutilizarea funcției introduce (**double**) din SIGMA (pentru care s-a utilizat și cuvântul cheie **virtual**) în clasa derivată MEDIA. Din acest exemplu simplu se poate constata că prin mecanismul moștenirii se poate reutiliza și extinde codul existent.

În exemplul următor se va construi o *ierarhie simplă de clase*. Mai exact se vor crea obiecte care vor simula diferite tipuri de aparate de măsură: digitale, analogice, cu scara liniară, cu scara logaritmică etc. Pentru aceasta avem nevoie de o clasă generică, care să înglogeze și să definească trasăturile comune ale acestor tipuri de obiecte. Această clasă va fi utilizată pentru a deriva din ea alte clase mai specializate, corespunzătoare unor tipuri de aparate. Partea pozitivă constă în faptul că

diferitele tipuri de aparate pot reutiliza codul folosit pentru definirea trăsăturilor lor comune. O listă a acestor trăsături comune, care vor reprezenta și denumirile variabilelor din programul următor este:

- rezoluția - numărul de diviziuni ale domeniului de măsurare al aparatului;
- scara - numărul de diviziuni pe unitatea de măsură, corespunzătoare mărimii măsurate;
- ofset - punctul de pe scală corespunzător valorii 0;
- poz_ac - indică valoarea curentă măsurată.

Pe lângă aceste componente, avem nevoie de funcții care să inițializeze parametrii și să afișeze valorile pe ecran. Aici intervine rolul mecanismului de moștenire deoarece prin acest mecanism se pot adapta unele funcții, prin intermediul funcțiilor virtuale.

Observație. Pentru ca o funcție să fie adaptabilă, ea trebuie definită în clasa de bază ca funcție virtuală, astfel încât clasele derivate să-i poată modifica comportarea.

În continuare, se prezintă primul exemplu de reprezentare și prelucrare a unei clase adaptabile de tip APARAT_DE_MASURA:

```
// Program P8_5a.CPP    Mecanismul moștenirii. Structuri ierarhice
// Programul conține clasa APARAT_DE_ MASURA
# include <stdio.h>
# include <math.h>

class APARAT_DE_ MASURA {     // Se creează o clasă de tip APARAT_DE_ MASURA
public:
   double rezolutia, scara, ofset, poz_ac;
   virtual double converteste (double date);
   void init_param ( double r, double s, double o );
   virtual void introduce (double date);
   virtual void afiseaza (void);
};

void APARAT_DE_ MASURA :: init_param ( double r, double s, double o )
{  rezolutia = r; scara = s; ofset = o;
   poz_ac = 0.0;
}
double APARAT_DE_ MASURA :: converteste (double date)
// Implicit, se folosește un afișaj (scală) liniar (liniară)
{  double p;
   p = date * scara - ofset;
   if ( p > rezolutia )  p = rezolutia;                    // Caz de overflow !
   return p;
}

void APARAT_DE_ MASURA :: introduce (double date)
// Această funcție primește datele care sosesc și calculează poziția acului indicator
{   poz_ac = converteste (date);  }

void APARAT_DE_ MASURA :: afiseaza (void)
// Modul de afișare implicit este cel digital
{  int i;
   double valmax = rezolutia * scara;                    // valmax = valoarea maxima ce poate fi afisata
   int gama = log10(valmax) + 1;
   for (i = 0; i < gama + 2; i++ ) putchar ('-');
```

```
   printf ("\n| %d|\n", (int) poz_ac);
   for (i = 0; i < gama + 2; i++ ) putchar ('-');
   putchar ('\n');
}
void main (void)                                    // Program de test
{
 APARAT_DE_ MASURA TUROMETRU;                       // Se creează obiectul TUROMETRU
 TUROMETRU.init_param (120.0, 1.0, 0.0);            // Inițializarea parametrilor
 TUROMETRU.introduce (55.0);                        // Prima valoare de afișat
 TUROMETRU.afiseaza();
 putchar ('\n');
 TUROMETRU.introduce (75.0);                        // A doua valoare de afișat
 TUROMETRU.afiseaza();
}
```

În acest exemplu clasa APARAT_DE_MASURA a fost facută adaptabilă prin declararea funcțiilor converteste(), introduce() și afiseaza() de tip **virtual**. S-a presupus că sistemul de afișaj este digital, iar scala este liniară. În acest caz, acul indicator este un număr folosit în apelul funcției printf(). Până aici, exemplul este banal, întrucât nu s-a folosit mecanismul moștenirii.

Să presupunem că se dorește un obiect de tip APARAT_DE_MASURA analogic. Se dorește ca afișajul acestui aparat să fie orizontal, liniar, iar un ac indicator să se deplaseze de-a lungul scalei și să indice valoarea curentă. Acest nou obiect va avea șablonul unei noi clase denumită APARAT_DE_MASURA_ORIZ derivată din clasa APARAT_DE_MASURA în care nu vom modifica decât funcția afiseaza(). Listingul acesui program este următorul:

```
// Program P8_5b.CPP   Mecanismul moștenirii. Structuri ierarhice
// Programul conține clasa derivată APARAT_DE_MASURA_ORIZ
# include <stdio.h>
# include <math.h>
class APARAT_DE_MASURA {     // Se creează o clasă de tip APARAT_DE_ MASURA
public:
   double rezolutia, scara, ofset, poz_ac;
   virtual double converteste (double date);
   void init_param ( double r, double s, double o );
   virtual void introduce (double date);
   virtual void afiseaza (void);
};
class APARAT_DE_ MASURA _ORIZ : public APARAT_DE_ MASURA {
public:
   void afiseaza (void);                    // Prototipul funcției afiseaza() care va fi modificată
};
void APARAT_DE_ MASURA :: init_param ( double r, double s, double o )
{  rezolutia = r; scara = s; ofset = o;
   poz_ac = 0.0;
}
double APARAT_DE_ MASURA :: converteste (double date)
// Implicit, se folosește un afișaj (scală) liniar (liniară)
{  double p;
```

```
   p = date * scara - ofset;
   if ( p > rezolutia )    p = rezolutia;                    // Caz de overflow !
   return p;
}
void APARAT_DE_ MASURA :: introduce (double date)
// Această funcție primește datele care sosesc și calculează poziția acului indicator
{    poz_ac = converteste (date);    }

void APARAT_DE_ MASURA :: afiseaza (void)  // Această funcție se redefinește în clasa derivată
{         ;         }

void APARAT_DE_ MASURA _ORIZ :: afiseaza (void) // Definirea funcției modificate afiseaza()
// Modul de afișare implicit este cel digital
{  int i;
   for (i = ofset; i <= rezolutia+2; i++) putchar ('-');
   putchar ('\n'); putchar ('|');
   for (i = ofset; i <= rezolutia; i++) {
         if (i % 5) putchar ('.');
         else if (i % 10) putchar ('|');
         else putchar ('0' + i/10);
   }
   putchar ('|'); putchar ('\n'); putchar ('|');
   for (i = ofset; i <= rezolutia; i++) {
         if ( i == poz_ac) putchar('^');
         else putchar(' ');
   }
   putchar ('|'); putchar ('\n');
   for (i = ofset; i <= rezolutia+2; i++)
   putchar ('|');
   putchar ('\n');
}
void main (void)                                           // Program de test
{APARAT_DE_ MASURA _ORIZ   TUROMETRU;      // Se creează obiectul TUROMETRU
 TUROMETRU.init_param (50.0, 1.0, 0.0);    // Inițializarea parametrilor
 TUROMETRU.introduce (11.0);               // Prima valoare de afișat
 TUROMETRU.afiseaza();
 putchar ('\n');  putchar ('\n');
 TUROMETRU.introduce (45.0);               // A doua valoare de afișat
 TUROMETRU.afiseaza();
}
```

Clasa derivată moștenește toți membrii (date și funcții) publici ai clasei de bază. Întrucât funcțiile viruale converteste() și introduce() nu sunt redefinite în clasa derivată, ele vor fi utilizate cu prototipul și definițiile din clasa da bază. Numai pentru funcția virtuală afiseaza() s-a scris o nouă versiune în clasa derivată, conform necesităților impuse de noul obiect ce va fi creat.

Observație. În fiecare versiune a unei funcții virtuale trebuie utilizat același prototip de funcție; în caz contrar doar am redefinit funcția (n-am modificat-o).

Deși, în clasa derivată, pentru funcția afiseaza() nu s-a mai utilizat cuvântul cheie **virtual**, trebuie menționat că o funcție declarată de tipul **virtual**, va rămâne virtuală în toate clasele derivate

ulterior, chiar dacă ierarhia are mai multe nivele de adâncime. Cuvântul cheie **virtual** poate fi inclus și în clasele derivate pentru a arăta că acea funcție poate fi modificată.

8.4. Funcțiile constructor, destructor și mecanismul de moștenire

Clasa de bază, clasa derivată, sau ambele pot avea funcții constructor și/sau destructor. Când o clasă de bază și una derivată conțin atât funcții constructor, cât și funcții destructor, funcțiile constructor sunt executate în ordinea derivării claselor, iar cele destructor în ordine inversă. Programele prezentate până acum n-au pasat argumente nici unui constructor din clasa de bază sau din clasa derivată. Când inițializarea argumentelor se produce numai în clasa derivată, argumentele se pasează normal. În schimb, dacă se pune problema pasării unor argumente unui constructor al unei clase de bază, trebuie stabilit un lanț al pasării argumentelor. Mai întâi sunt pasate constructorului clasei derivate toate argumentele necesare atât clasei de bază, cât și celei derivate; apoi sunt pasate clasei de bază argumentele corespunzătoare, folosindu-se o formă expandată a constructorului din clasa derivată. Sintaxa corectă necesară pasării unui argument din clasa derivată în clasa de bază este următoarea:

```
constructor_clasa_derivata (lista_arg1) : constructor_clasa_de_baza (lista_arg2) {
        // Corpul constructorului clasei derivate
}
```

Este permis atât clasei derivate, cât și clasei de bază să utilizeze același argument. De asemenea este posibil ca funcția constructor a clasei derivate să ignore toate argumentele și doar să le paseze clasei de bază. În continuare sunt prezentate o serie de exemple referitoare la utilizarea constructorilor și destructorilor și modul de pasare a argumentelor.

1. Programul următor indică ordinea de execuția a funcțiilor constructor și destructor.

```
// Program P8_6a.CPP  Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {            // Va fi utilizată pentru crearea clasei derivate DERIVATA
public:
   BAZA () { cout << "Se construieste clasa de baza\n"; }
   ~BAZA () { cout << "Se distruge clasa de baza\n"; }
};

class DERIVATA : public BAZA {
public:
   DERIVATA () { cout << "Se construieste clasa derivata\n"; }
   ~DERIVATA () { cout << "Se distruge clasa derivata\n"; }
};
int  main (void)                    // Programul principal
{       DERIVATA OB;                // Se construiește obiectul OB de tip DERIVATA
        return 0;
}
```

Pe ecran se va afișa:

```
Se construieste clasa de baza
Se construieste clasa derivata
Se distruge clasa derivata
Se distruge clasa de baza
```

Se remarcă apelul în ordinea derivării claselor a funcțiilor constructor și în ordine inversă a funcțiilor destructor.

2. Programul următor arată modul în care este pasat un argument unui funcții constructor al clasei derivate:

```
// Program P8_6b.CPP  Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {                    // Va fi utilizată pentru crearea clasei derivate DERIVATA
public:
   BAZA () { cout << "Se construieste clasa de baza\n"; }
   ~BAZA () { cout << "Se distruge clasa de baza\n"; }
};

class DERIVATA : public BAZA {
   int j;                                   // Variabila j este privată în clasa DERIVATA
public:
   DERIVATA (int n) {       // Constructorul clasei DERIVATA conține în listă parametrul n
         cout << "Se construieste clasa derivata\n";
         j = n;
   }
   ~DERIVATA () { cout << "Se distruge clasa derivata\n"; }
   void afiseaza_j () { cout << "j = " << j << endl; }
};
int  main (void)                // Programul principal
{DERIVATA OB(10);
  OB.afiseaza_j();
  return 0;
}
```

Pe ecran se va afișa:

```
Se construieste clasa de baza
Se construieste clasa derivata
j = 10
Se distruge clasa derivata
Se distruge clasa de baza
```

Se observă că argumentul n este pasat funcției constructor al clasei derivate într-o manieră uzuală.

3. În exemplul care urmează, atât constructorul clasei de bază, cât și cel al clasei derivate conțin un un același argument. Clasa derivată utilizează, dar și pasează argumentul clasei de bază:

```
// Program P8_6c.CPP  Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {                    // Va fi utilizată pentru crearea clasei derivate DERIVATA
   int i;
public:
   BAZA ( int n ) { cout << "Se construieste clasa de baza\n";
   i = n;   }
   ~BAZA () { cout << "Se distruge clasa de baza\n"; }
   void afiseaza_i() { cout << "i = " << i << endl; }
};

class DERIVATA : public BAZA {
   int j;
```

```
public:
   DERIVATA (int n) : BAZA (n) {          // Constructorul clasei derivate
         cout << "Se construieste clasa derivata\n";
         j = n;
   }
   ~DERIVATA () { cout << "Se distruge clasa derivata\n"; }
   void afiseaza_j() { cout << "j = " << j << endl; }
};
void  main (void)                 // Programul principal
{DERIVATA OB (10);
  OB.afiseaza_i ();
  OB.afiseaza_j ();
}
```

Pe ecran se va afișa:

```
Se construieste clasa de baza
Se construieste clasa derivata
i = 10
j = 10
Se distruge clasa derivata
Se distruge clasa de baza
```

Se vede că parametrul n este utilizat de clasa derivată și este pasat și clasei de bază.

4. În majoritatea cazurilor constructorii din clasa de bază și din cea derivată nu vor utiliza același argument. Într-o astfel de situație, când fiecărui constructor trebuie să-i pasăm unul sau mai multe argumente, este necesar să pasăm constructorului din clasa derivată toți parametri necesari atât clasei de bază, cât și celei derivate; apoi clasa derivată pasează clasei de bază argumentele necesare acesteia. Un exemplu în acest sens este prezentat în listingul următor:

```
// Program P8_6d.CPP   Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {                      // Va fi utilizată pentru crearea clasei derivate DERIVATA
   int i;
public:
   BAZA ( int n ) { cout << "Se construieste clasa de baza\n";
   i = n;
   }
   ~BAZA () { cout << "Se distruge clasa de baza\n"; }
   void afiseaza_i() { cout << "i = " << i << endl; }
};

class DERIVATA : public BAZA {          // Implementarea clasei derivate DERIVATA
   int j;
public:
   DERIVATA (int n, int m) : BAZA (m) { // Pasează argumentul clasei de bază BAZA
         cout << "Se construieste clasa derivata\n";
         j = n;
   }
   ~DERIVATA () { cout << "Se distruge clasa derivata\n"; }
   void afiseaza_j() { cout << "j = " << j << endl; }
};
```

```
void  main (void)                    // Programul principal
{DERIVATA OB (10, 40);               // Se creează obiectul OB de tip DERIVATA
  OB.afiseaza_i ();
  OB.afiseaza_j ();
}
```

Pe ecran se va afișa:

```
Se construieste clasa de baza
Se construieste clasa derivata
i = 40
j = 10
Se distruge clasa derivata
Se distruge clasa de baza
```

5. Este important de înțeles că nu este necesar ca funcția constructor din clasa derivată să aibă argumente pentru a pasa unul sau mai multe clasei de bază. Dacă clasa derivată nu are nevoie de nici un argument, ea le ignoră și doar le pasează clasei de bază. În programul următor parametrul n nu este utilizat de constructorul clasei derivate; el este numai pasat constructorului clasei de bază.

```
// Program P8_6e.CPP  Mecanismul moștenirii. Clase derivate
# include <iostream.h>

class BAZA {                    // Va fi utilizată pentru crearea clasei derivate DERIVATA
   int i;
public:
   BAZA ( int n ) { cout << "Se construieste clasa de baza\n";
   i = n;
   }
   ~BAZA () { cout << "Se distruge clasa de baza\n"; }
   void afiseaza_i() { cout << "i = " << i << endl; }
};

class DERIVATA : public BAZA {
public:
   DERIVATA (int n) : BAZA (n) {          // Pasează argumentul clasei de bază
         cout << "Se construieste clasa derivata\n";
}
   ~DERIVATA () { cout << "Se distruge clasa derivata\n"; }
};
void  main (void)                    // Programul principal
{DERIVATA OB (10);                   // Se creează obiectul OB de tip DERIVATA
  OB.afiseaza_i ();
}
```

8.5. Mecanismul moștenirii multiple

Există două modalități prin care o clasă derivată poate moșteni de la mai multe clase:

1. Prima modalitate se referă la faptul că o clasă derivată poate fi folosită drept clasă de bază pentru o altă clasă derivată, creîndu-se astfel o ierarhie de clase. În acest caz, clasa de bază inițială este o *clasă de bază indirectă* a celei de-a doua clase derivate.
2. A doua modalitate se referă la faptul că o clasă derivată poate moșteni în mod direct de la mai multe clase de bază. La utilizarea acestui tip de mecanism se ridică o serie de probleme pe care le vom aborda în acest paragraf.

Când o clasă de bază este folosită pentru a se deriva din ea altă clasă, care la rândul ei este utilizată drept clasă de bază pentru a se deriva altă clasă, funcțiile constructor ale celor trei clase sunt apelate în ordinea derivării, iar funcțiile destructor sunt apelate în ordine inversă. Se observă că se generalizează mecanismul învățat anterior. Astfel, dacă clasa B1 este moștenită de clasa D1, iar la rândul său aceasta este moștenită de clasa D2, atunci se apelează mai întâi, constructorul clasei B1, urmat de cel din clasa D1 și, în final, constructorul clasei D2. Destructorii sunt apelați în ordinea inversă.
Când o clasă derivată moștenește direct de la mai multe clase de bază, se utilizează următoarea declarație expandată:

```
class nume_clasa_derivata : specificator_acces  nume_clasa_de_baza_1,
                            specificator_acces  nume_clasa_de_baza_2,
                            ..., specificator_acces  nume_clasa_de_baza_N
{  // Corpul clasei   };
```

Precizăm că specificatorul de acces poate fi diferit pentru fiecare clasă de bază. Când sunt moștenite mai multe clase de bază, funcțiile constructor sunt apelate în ordine de la stânga la dreaptă, funcțiile destructor fiind apelate în ordine inversă. Clasa derivată pasează claselor de bază argumentele necesare, prin utilizarea următoarei forme expandate a constructorului clasei derivate:

```
constructor_clasa_derivata (lista_argumente) : BAZA_1 (lista_argumente),
                                               BAZA_2 (lista_argumente),
                                               ..., BAZA_N (lista_argumente)
{ // Corpul constructorului din clasa derivată }
```

unde BAZA_1, ..., BAZA_N reprezintă numele constructorilor claselor de bază.
Pentru claritate, în continuare, se prezintă o serie de exemple:

1. Primul exemplu se referă la o moștenire multiplă, când o clasă derivată moștenește de la o clasă derivată din altă clasă.

```
// Program P8_7a.CPP  Mecanismul moștenirii multiple. Clase derivate
# include <iostream.h>

class B1 {                                  // Va fi utilizată pentru crearea clasei derivate D1
        int a;
public:
        B1 ( int x ) { a = x; }
        int obtine_a() { return a; }
};
class D1 : public B1 {                      // Mostenește de la clasa de bază directă
        int b;
public:
        D1(int x, int y) : B1(y)        // Constructorul clasei D1 pasează parametrul y clasei B1
        { b = x; }
        int obtine_b() { return b; }
};
class D2 : public D1 {          // Se moșteneste atât de la o clasă derivată, cât și de la una indirectă
        int c;
public:
        D2 (int x, int y, int z) : D1(y, z)       // Se pasează argumentele clasei D1
        {  c = x;   }
   // Întrucât moștenește membrii publici ai claselor de bază, clasa D2 are
   // acces la elementele de tip public, atât din clasa B1, cât și din clasa D1
```

```
        void afiseaza () {
                cout << obtine_a() << ' ' << obtine_b() << ' ' << c << endl;
        }
};
void  main (void)                       // Programul principal
{       D2 OB (10, 20, 30);             // Se construiește obiectul OB de tip D2
        OB.afiseaza();
        cout << OB.obtine_a() << ' ' << OB.obtine_b() << endl;
}
```

Apelul funcției OB.afiseaza() generează rezultatul 30 20 10. În acest exemplu, B1 este o clasă de bază *indirectă* pentru clasa D2. Se observă că D2 are acces la membrii de tip **public** atât ai clasei D1, cât și ai clasei B1. Reamintim că utilizând, în mecanismul moștenirii, specificatorul **public**, membrii publici ai clasei moștenite devin membri publici ai clasei derivate. Prin urmare, când clasa D1 a moștenit clasa B1, funcția obține_a() a devenit membră de tip **public** a clasei D1, care la rândul ei, e membră publică a clasei D2. Așa cum arată programul, fiecare clasă din ierarhia de clase trebuie să paseze toate argumentele cerute de fiecare clasă precedentă, altfel se va semnala o eroare la momentul compilării.

2. Modificăm programul anterior astfel încât o clasă derivată să moștenească de la două clase de bază.

```
// Program P8_7b.CPP  Mecanismul moștenirii multiple. Clase derivate
# include <iostream.h>

class B1 {                      // Clasa de baza B1
        int a;
public:
        B1 ( int x )      { a = x; }
        int obtine_a() { return a; }
};

class B2 {                      // Clasa de baza B2
        int b;
public:
        B2 ( int x ) { b = x; }
        int obtine_b() { return b; }
};

class D : public B1, public B2 {  // Mostenește direct de la cele două clase de bază B1 și B2
        int c;
public:
        D (int x, int y, int z) : B1(z), B2(y)
        // Variabilele z și y sunt pasate direct claselor B1 și B2
        {  c = x;  }
// Întrucât membrii claselor de bază au fost moșteniți ca publici, clasa D are
// acces la elementele de tip public, atât din clasa B1, cât și din clasa B2

        void afiseaza() {
                cout << obtine_a() << ' ' << obtine_b() << ' ' << c << endl;
        }
};
```

```
void  main (void)                          // Programul principal
{       D  OB (10, 20, 30);                // Se construiește obiectul OB de tip D
        OB.afiseaza();
        cout << OB.obtine_a() << ' ' << OB.obtine_b() << endl;
}
```

Argumentele y și z sunt pasate individual claselor B1 și B2 de către funcția constructor a clasei D.

3. Programul următor ilustrează ordinea de apel a funcțiilor constructor și destructor, când o clasă derivată moștenește, în mod direct, de la mai multe clase de bază:

```
// Program P8_7c.CPP   Mecanismul moștenirii multiple. Apelul constructorilor și destructorilor
# include <iostream.h>

class B1 {                                 // Clasa de bază B1
public:
        B1() { cout << "Se construieste B1\n"; }
         ~B1() { cout << "Se distruge B1\n"; }
};

class B2 {                                 // Clasa de bază B2
public:
        B2() { cout << "Se construieste B2\n"; }
        ~B2() { cout << "Se distruge B2\n"; }
};

class D : public B1, public B2 {       // Mostenește direct de la cele două clase de bază B1 și B2
public:
        D() { cout << "Se construieste D\n"; }
        ~D() { cout << "Se distruge D\n"; }
};
void  main (void)                          // Programul principal
{       D    OBJ;       }                  // Se creează obiectul OBJ de tip D
```

Pe ecran se va afișa:

```
        Se construieste B1
        Se construieste B2
        Se construieste D
        Se distruge D
        Se distruge B2
        Se distruge B1
```

Se observă că atunci când sunt moștenite în mod direct mai multe clase de bază, funcțiile constructor sunt apelate în ordine de la stânga la dreapta, după cum se specifică în lista de moștenitori.

8.6. Clase de bază virtuale

O problemă mai delicată apare atunci când o clasă derivată moștenește direct mai multe clase de bază, derivate, la rândul lor, dintr-o singură clasă de bază. Considerăm ierarhia de clase din Fig.8.1. Aici, clasa de bază BAZA este moștenită atât de clasa DERIVATA1, cât și de clasa DERIVATA2. Clasa DERIVATA3 moștenește direct atât clasa DERIVATA1, cât și clasa DERIVATA2. Se observă că, în această situație, clasa BAZA este moștenită de două ori de către clasa DERIVATA3 (mai întâi prin intermediul clasei DERIVATA1, iar a doua oară prin intermediul clasei DERIVATA2). Acest lucru determină o ambiguitate când clasa DERIVATA3 vrea să aibă acces la

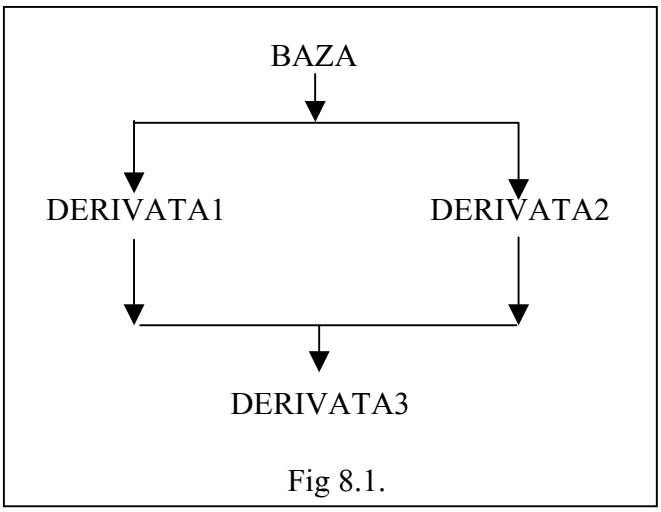

Fig 8.1.

clasa de bază BAZA. Întrucât există două copii ale clasei BAZA, în clasa DERIVATA3, o referire la un membru din clasa BAZA se referă la acela care a fost moștenit indirect prin intermediul clasei DERIVATA1, sau la membrul moștenit indirect, prin intermediul clasei DERIVATA2 ? Pentru a rezolva această ambiguitate, C++ include un mecanism prin care numai o singură copie a clasei BAZA va fi inclusă în clasa DERIVATA3. Acest mecanism se numește *clasă de bază virtuală*. Astfel, prin utilizarea cuvântului cheie **virtual**, înaintea specificatorului de acces, se previne fenomenul ca două copii ale clasei de bază să se găsească în clasa derivată. Acest aspect este prezentat în exemplul următor:

```
// Program P8_8.CPP  Mecanismul moștenirii multiple. Clase de bază virtuale
# include <iostream.h>

class BAZA {            // Clasa de baza B1
public:
        int i;
};
class DERIVATA1 : virtual public BAZA {   // Moștenește clasa de bază virtuală
public:
        int j;
};

class DERIVATA2 : virtual public BAZA {   // Moștenește clasa de bază virtuală
public:
        int k;
};

class DERIVATA3 : public DERIVATA1, public DERIVATA2
        // Moștenește atât de la clasa DERIVATA1, cât și de la clasa DERIVATA2
{
public:
        int produs() { return i*j*k; }
};

void  main (void)                   // Programul principal
{       DERIVATA3 OBJ;              // Se construiește obiectul OBJ de tip DERIVATA3
        OBJ.i = 10;
        OBJ.j = 30;
```

```
        OBJ.k = 50;
        cout << "Produsul i*j*k = " << OBJ.produs() << '\n';
        OBJ.i = 2;
        OBJ.j = 3;
        OBJ.k = 5;
        cout << "Produsul i*j*k = " << OBJ.produs() << '\n';
}
```

Observație. Dacă clasele DERIVATA1 și DERIVATA2 n-ar fi moștenit clasa de bază drept virtuală, la compilare s-ar fi semnalat o serie de erori.