1. INTRODUCERE

1.1. Scurt istoric

Multă vreme C a fost limbajul preferat de programatori, în special de cei care dezvoltau aplicații pentru sistemele MS-DOS și WINDOWS. În ultima vreme însă, popularitatea limbajului C++ a crescut datorită faptului că permite programarea orientată pe obiecte (Object-Oriented Programming) - o metodă de programare folosită în prezent pentru realizarea multor aplicații software.

Ideea programării orientate pe obiecte (POO) a apărut în anii `60, fiind pusă în practică prin intermediul limbajelor SIMULA (1967) și SMALLTALK (1975). Totuși, aceste limbaje au avut o răspândire relativ redusă, deoarece puțini programatori formați la școala limbajelor clasice procedurale din acea perioadă (FORTRAN, COBOL, PASCAL, MODULA-2, C etc.) erau dispuși să abandoneze aceste limbaje doar de dragul de a lucra obiectual. Cu toate acestea, în anii `80, în urma acceptării definitive a limbajului C, un colectiv condus de Bjarne Stroustrup, un tânăr cercetător de la Bell Lab`s, a avut ideea scrierii unui compilator care să preia simplitatea și flexibilitatea C-ului și mecanismele de "modelare" ale limbajului SIMULA 67. Bjarne a numit acest dialect "C with Classes" și, prima versiune comercială a acestuia a apărut la AT&T în 1983, cu denumirea modificată în cea actuală, C++ (sugerată de Rik Masciti, un colaborator apropit a lui B. Stroustrup). Denumirea de C++ semnifică de fapt multiplele facilități adăugate limbajului C. Profitând de multitudinea domeniilor de aplicație (de la grafica interactivă la proiectarea interfețelor utilizator și de la exploatarea rețelelor de calculatoare la tehnicile de proiectare a compilatoarelor), printre programatori în general și printre programatorii de C în particular, aproape imediat apar partizani ai POO-ului. De ce acest succes extraordinar ? În primul rând, din cauza faptului că limbajul C++ nu face nimic altceva decât sa dea un nou avânt unuia dintre cele mai la modă limbaje ale momentului (este vorba de C), iar în al doilea rând din cauza faptului că aduce o și mai mare economie de timp în procesul de dezvoltare-implementare-testare a aplicațiilor software. În cazul limbajelor tradiționale procedurale (3GL`s - 3rd Generation Languages), algoritmul materializat într-o diagramă de flux a datelor (DFD - Data Flow Diagram), ajunge să se adapteze arhitecturii calculatorului. Generația a patra de limbaje (4GL`s), cum se obișnuiește a se denumi categoria acestor limbaje orientate pe obiecte, urmărește adaptarea calculatorului la obiecte.

1.2. Avantajele POO

Avantajele POO reies din definirea principalelor concepte care stau la baza POO și anume: *abstractizarea datelor* (*data abstraction*), *moștenirea* (*inheritance*) și *polimorfismul* (*polymorphism*). *Abstractizarea datelor* se referă la procesul de definire a tipurilor abstracte de date, în timp ce *moștenirea* și *polimorfismul* se referă la mecanismele care permit programatorilor să beneficieze de caracteristicile comune ale tipurilor abstracte de date (*obiectele* din POO).

Abstractizarea datelor

Termenul *tip abstract de date* se referă la un tip de date definit de programator obținut prin încapsularea ("contopirea") datelor specifice aplicației cu setul de operații (codul) care pot fi efectuate asupra acestor date. Este numit abstract, pentru a-l diferenția de tipurile de date de bază predefinite în C, cum ar fi **int**, **char**, **float** și **double**. Deci, definirea unui tip abstract de date (abstract data type - ADT) implică specificarea reprezentării interne a datelor din acel tip, precum și a funcțiilor pe care alte module de program le vor utiliza pentru manipularea acelui tip abstract de date. Ascunderea datelor, o facilitate a abstractizării datelor, asigură posibilitatea modificării structurii interne a unui tip abstract de date fără a provoca funcționarea defectuoasă a programelor

care apelează funcțiile ce operează asupra acelui tip abstract de date. În POO, un astfel de tip abstract de date definit de utilizator, dar care se comportă la fel ca un tip predefinit, se numește *clasă* (*class*). O *clasă* poate fi considerată ca un model (șablon) din care pot fi create *obiecte* specifice. Un obiect este o instanțiere a unei clase, deci o variabilă declarată ca fiind de tipul clasă respectiv.
Funcțiile care operează asupra unui obiect sunt denumite *metode*. Metodele definesc comportarea unui obiect. În C++, metodele sunt denumite *funcții membre* (*member functions*) ale clasei.
Prin conceptul de încapsulare a datelor și codului corespunzător se ating și alte obiective: posibilitatea de localizare a erorilor (întotdeauna cauza se află în "interiorul" unei singure clase) și modularizarea problemei de rezolvat (fiecare clasă va rezolva, de regulă, o singură problemă).

Moștenirea
Abstractizarea datelor nu acoperă o caracteristică importantă a obiectelor și anume aceea că obiectele din lumea reală nu există în stare izolată. Fiecare obiect este în relație cu unul sau mai multe obiecte. De multe ori un nou obiect poate fi descris evidențiind modul în care caracteristicile și comportarea acestuia diferă față de cele ale unei clase de obiecte deja existente. Această practică de a defini noi obiecte în termeni ai unuia (unora) vechi este o parte integrantă a POO și se definește prin conceptul de *moștenire*. Prin mecanismul moștenirii, în urma definirii unei clase, cu un minim de efort și timp, se pot preciza seturi de clase asemănătoare, având totuși o trăsătură distinctivă. Moștenirea impune o relație ierarhică între clase, prin care o *clasă derivată* (*derived class*) moștenește caracteristicile unei *clase de bază* (*base class*). Trebuie precizat că, prin mecanismul moștenirii multiple, mai multe clase derivate pot moșteni o aceeași clasă de bază și, mai multe clase de bază pot fi moștenite de o aceeași clasă derivată.

Polimorfismul
În sens literal, polimorfism înseamnă calitatea de a avea mai mult de o formă. În contextul POO, polimorfismul înseamnă că într-o ierarhie de clase obținute prin moștenire, o metodă poate avea forme diferite de la un nivel la altul (specifice respectivului nivel de ierarhie) și poate funcționa diferit în obiecte diferite. De exemplu, să considerăm operația de adunare. Pentru două numere, adunarea va genera suma lor. Într-un limbaj de programare care suportă POO, operația de adunare poate fi exprimată printr-un singur operator, semnul plus (+). Considerând acest fapt, se poate utiliza expresia x+y pentru a indica suma lui x și y pentru mai multe tipuri diferite de numere x și y cum ar fi: întregi, numere în virgulă mobilă, numere complexe etc. Se poate chiar defini operația + ca însemnând concatenarea a două șiruri de caractere.

2. C++ ȘI PROGRAMAREA ORIENTATĂ PE OBIECTE

Scopul acestui capitol este de a face o scurtă trecere în revistă a celor mai importante caracteristici și concepte introduse de limbajul C++. Nu se prezintă detaliile sintactice ale limbajului C++, ci, prin câteva exemple simple, se va iniția numai un prim contact al cititorului cu noile concepte. Explicațiile furnizate aici vor fi de natură să incite la o parcurgere atentă a capitolelor următoare în care gradul de rafinare a informației va crește substanțial.

2.1. Diferența dintre clasă și structură

Listingul programului din Exemplul 2.1 evidențiază o serie de concepte din C++ cum ar fi: clasă, structură, constructor, obiect etc.

```
// Exemplul 2.1. Program  P2_1.CPP  Diferența dintre clasă și structură
# include <iostream.h>
# include <stdio.h>
class CLS {
        int  a, b;                              // a și b sunt de tip "private"
public:
        CLS (int z = 0)  { a = b = z; }         // Constructorul clasei CLS
        void Imp(char *mesaj = " ") {
                printf ("%s a si b = %d  %d\n", mesaj, a, b);
        }
};
struct STRU {
        int  a, b;                              // a și b sunt de tip "public" aici
        STRU (int z = 0) {a = b = z;}           // Constructorul structurii STRU
};
// Programul principal
void main (void)
{       CLS ob_c(1);
        STRU ob_s(10);
        ob_c.Imp("Dupa creare, datele obiectului ob_c devin: ");
        // ob_c.a = 111; ar conduce la eroarea CLS :: a is not accessible
        cout << "Datele obiectului ob_s inainte de modificare = " << ob_s.a\
                << " " << ob_s.b << endl;
        ob_s.a = 100;
        ob_s.b = 1;
        cout << "si dupa = " << ob_s.a << " " << ob_s.b << endl;
}
```

Mai întâi, se observă că în C++ spre deosebire de C, comentariile sunt precedate de "//". Dar în C++ se acceptă și forma comentariului din C: /* text comentariu */.

Construcția:

```
class CLS {
        int  a, b;                              // a și b sunt de tip "private"
public:
        CLS (int z = 0)  { a = b = z; }         / Constructorul clasei CLS
        void Imp(char *mesaj = " ") {
                printf ("%s a si b = %d  %d\n", mesaj, a, b);
        }
};
```

reprezintă *declararea clasei CLS*. Această construcție pune în evidență un șablon care va servi la crearea ulterioară a obiectelor.
Obiectul ob_c ce aparține clasei CLS este *declarat* în programul principal prin:

```
CLS ob_c(1);
```

Expresia ob_c(1) din această instrucțiune instruiește compilatorul ca să inițializeze ambele variabile de tip întreg a și b cu valoarea 1. Funcția care este automat apelată la declararea obiectului ca să realizeze această inițializare este *constructorul clasei CLS*, denumit la fel ca și clasa, CLS. În acest exemplu nu apare și funcția *destructor* al clasei. În program întâlnim și linia de comentariu

```
// ob_c.a = 111; ar conduce la eroarea CLS::a is not accessible
```

În acest stadiu să reținem doar că accesul la elementele (variabilele) a și b situate deasupra cuvântului cheie **public** este îngrădit, ele fiind de tipul **private**. Numai prin intermediul constructorului clasei CLS avem acces la variabilele a și b ale obiectului ob_c, nu și direct. Dacă în funcția main() am fi întâlnit linia sursă următoare:

```
CLS ob_c;
```

ea nu ar fi fost refuzată din punct de vedere sintactic. În acest caz, a și b ar fi fost inițializate cu valoarea asumată 0. Se vede că în interiorul constructorului CLS, z = 0. Corpul funcției CLS este delimitat, ca oricare funcție din C, de acolade, iar aici, ca unică instrucțiune se întâlnește dubla atribuire a = b = z; echivalentă cu instrucțiunile a = z; b = z;. Deci, în C++, orice linie sursă se termină prin separatorul ";" ca și în C.
Programul mai conține construcția:

```
struct STRU {
        int  a, b;                          // a și b sunt de tip "public" aici
        STRU (int z = 0) {a = b = z;}// Constructorul structurii STRU
};
```

care este șablonul unei structuri cu numele STRU ce conține două variabile a și b tot de tip întreg (ca și variabilele a și b din structura CLS) și o funcție (constructorul structurii STRU, denumit tot STRU). Obiectul ob_s creat în linia

```
STRU ob_s(10);
```

va inițializa variabilele a și b cu valoarea 10. Şi aici, în lipsa unei valori explicite, a și b vor fi inițializate cu valoarea asumată în constructor, 0. De data aceasta variabilele a și b din structura STRU fiind de tip **public** sunt accesibile din orice instrucțiune din funcția main(), fără a apela la serviciile constructorului. Deci, putem scrie:

```
ob_s.a = 100;
```

și variabila a din obiectul ob_s devine egală cu 100, în loc de 10.
Se observă de asemenea că, atât constructorul clasei CLS cât și constructorul structurii STRU sunt definiți în interiorul acestora, sau altfel spus sunt definiți în modul **inline**.
În sfârșit, acest program ne arată și modalitatea afișării datelor; este vorba de liniile în care apare cuvântul cheie **cout**, echivalentul funcției printf() din limbajul C, dar mai comod decât aceasta. Funcția printf() este acceptată și în C++ și păstrează încorsetările din C. De exemplu în C, pentru afișarea valorii variabilei întregi j, se recurge la secvența:

```
# include <stdio.h>     // Prototipul funcției printf() este definit în fișierul antet "stdio.h"
...
int j = 10;
printf ("j = %d\n", j);
```

Funcția printf() are două argumente: "j = %d\n" și j. Primul argument conține un șir de caractere și un *caracter de conversie*. Acesta este precedat de caracterul "%". Când compilatorul întâlnește %d acesta este atenționat că va fi afișat un întreg în format zecimal, în cazul nostru al doilea argument al funcției printf(), adică întregul j. Ieșirea se efectuează la fișierul logic **stdout** (de regulă atribuit ecranului videoterminalului).

Dacă j ar fi fost de tipul **double** (virgulă mobilă, dublă precizie), formatul trebuia modificat din %d în %f (sau %g). Deci, am fi avut:

```
...
double j = 10;
printf ("j = %f\n", j);
```

În acest caz, j s-ar fi transformat din reprezentarea internă în virgulă mobilă, într-un format extern zecimal propriu acestei reprezentări (de exemplu, mmmmmm.nnnnnn).
În contextul limbajului C++, **cout** înlocuiește **stdout**. Acesta este alcătuit din literalul "Datele obiectului ob_s inainte de modificare = ", urmat de variabila ob_s.a, apoi de un al doilea literal " " (adică un șir de spații), și în sfârșit de variabila ob_s.b. Dacă a și b ar fi fost de tip **double**, nu mai trebuia schimbat nimic în linia sursă în care apare obiectul **cout**, membru al clasei **ostream**; aceasta deoarece pentru fiecare tip de informație, C++ pune la dispoziție în contextul obiectului **cout** o "metodă" (funcție), care analizează și înțelege mesajul (inclusiv tipurile variabilelor care trebuie afișate) și realizează conversiile adecvate. Elementele clasei de ieșire **ostream** se află în fișierul antet **iostream.h**, inclus în exemplul de față prin linia:
include <iostream.h>.

Funcția membră a clasei
Pentru a afișa variabilele a și b ale clasei CLS, aceasta conține funcția Imp definită prin:
void Imp(**char** *mesaj = " ") { printf ("%s a si b = %d %d\n", mesaj, a, b); }
și numită *funcție de tip membru* (*member*) al clasei. Se observă că aceasta este definită în corpul clasei CLS, deci **inline**, în porțiunea publică a acestei clase, accesibilă din orice funcție inclusiv din funcția main(). Cuvântul cheie **void** din fața funcției Imp arată că aceasta nu întoarce nici un rezultat. De altfel, din corpul funcției Imp lipsește instrucțiunea **return**. Prezența sa ar fi ilegală, dacă s-a stabilit că funcția Imp nu întoarce nici un rezultat. Variabila mesaj este un pointer (notație *mesaj) la un șir de caractere. Formatul listării unui șir de caractere este **%s**. Șirul asumat, în lipsa unui text atribuit lui mesaj, este un șir de lungime nulă (șir vid). Variabilele a și b fiind de tipul întreg, fiecare vor fi asociate unui format **%d**. Notația **\n** este caracterul de tip spațiu alb (white space) numit avans la linie nouă (*line feed*, cod 0x0a în hexazecimal sau 10 în zecimal).
Funcțiile de tip membru au privilegiul de a avea acces direct la variabilele clasei, inclusiv la cele de tipul **private**. Nu a mai fost nevoie de construcții de genul ob_c.a sau ob_c.b, ci referirea s-a făcut direct la a sau b. Pentru aflarea valorilor variabilelor a și b ale obiectului ob_c, în funcția main() a fost adăugată linia următoare:

```
ob_c.Imp("Dupa creare, datele obiectului ob_c devin:");
```

plasată undeva după linia:

```
CLS ob_c(1);
```

O altă variantă a funcției Imp() este cea în care nu se mai recurge la serviciile funcției printf(), ci la cout:

```
void Imp(char *mesaj = " ") { cout <<  mesaj << a << " " << b << endl; }
```

Cuvântul **endl** este echivalentul lui \n din C.
Funcțiile "inline" au avantajul că la apelarea lor nu se mai parcurg secvențele (de regulă lungi) de salvare în stivă a parametrilor de intrare, execuția crescând în rapiditate.

2.2. Redefinirea funcțiilor și operatorilor

Pentru a prezenta alte caracteristici ale limbajului C++, considerăm listingul programului din Exemplul 2.2.

```
// Exemplul 2.2. Program P2_2.CPP
// O clasa cu constructor, destructor, o functie membru redefinita
// si o alta functie care redefineste operatorul || pentru concatenarea unor siruri
```

```
# include <string.h>
# include <stdio.h>
class SIR {
        char *text;
public:
        SIR (char *sir);                        // Constructorul clasei
        ~SIR () { delete text;}                 // Destructorul clasei
        int compar (SIR &s1, SIR &s2);
        int compar (SIR &s1, SIR &s2, unsigned int ncar);
        void operator || (SIR &s) {strcat (text, s.text);}
        void Imp (char *mesaj = " ") {
                printf ("%s = %s\n", mesaj, text);
        }
};
SIR::SIR (char *sir)                            // Definirea constructorului
{
        text = new char[strlen(sir)];
        strcpy (text, sir);
}
int SIR::compar (SIR &s1, SIR &s2)
{
        return strcmp (s1.text, s2.text);
}
int SIR::compar (SIR &s1, SIR &s2, unsigned ncar)
{
        return strncmp (s1.text, s2.text, ncar);
}
// Programul principal
void main (void)
{       SIR sir1 ("abcd"), sir2 ("abcdef");             // Se creeaza 2 obiecte de tip sir
        sir1.Imp ("Primul sir este: ");
        sir2.Imp ("Al doilea sir este: ");
        int   rez1, rez2;
        rez1 = sir1.compar (sir1, sir2);
        printf ("Rezultatul primei comparatii este %d\n", rez1);
        rez2 = sir1.compar (sir1, sir2, 4);
        printf ("Rezultatul celei de-a doua comparatii este %d\n", rez2);
        sir1 || sir2;
        sir1.Imp ("Sir1 dupa concatenare este: ");
}
```

Programul conține clasa SIR în care singura dată de tip **private** este pointerul la un șir de caractere *text. Constructorul SIR și funcția compar() sunt numai declarate în cadrul clasei, pe când destructorul ~SIR și funcția denumită **operator**||() sunt definite în mod **inline**. Faptul că denumirea compar este întâlnită de două ori nu este o eroare. Funcția compar() este redefinită. Ea are două șabloane (prototipuri). Primul șablon este utilizat când se compară două șiruri de lungimi diferite sau nu, iar al doilea când se compară numai primele ncar caractere ale acestora. Notațiile SIR &s1, SIR &s2 reprezintă referințele celor două șiruri ce trebuie comparate.
Funcția SIR este definită în afara clasei prin:

```
SIR::SIR (char *sir)                    // Definirea constructorului
{
        text = new char[strlen(sir)];
        strcpy (text, sir);
}
```

în care apare notația "::" prin care se precizează apartenența unei funcții la o anumită clasă. Instrucțiunea

```
text = new char[strlen(sir)];
```

arată că, începând de la adresa text, operatorul **new** va aloca un număr de caractere egal cu lungimea efectivă a sirului sir. Lungimea șirului este returnată de funcția strlen() (*string length*). În limbajul C pentru alocarea dinamică a unui spațiu de memorie se folosește funcția malloc() care cere specificarea numărului de octeți. Deci, în C scriem:

```
text = malloc (strlen (sir));
```

fapt acceptat și în C++. Instrucțiunea strcpy(text, sir); realizează copierea șirului sir în spațiul alocat la adresa text. Prototipurile funcțiilor strlen() și strcpy() (*string copy*) sunt declarate în fișierul antet **string.h**. Operatorul **delete** din definiția destructorului ~SIR este opusul lui **new** și are ca efect eliberarea zonei de memorie ocupată de obiect. În C, opusul funcției malloc() este funcția free(). Deci, echivalența din C a liniei **delete** sir; este linia free(sir).

Funcția compar() întoarce un rezultat de tip întreg. Celor două prototipuri declarate în clasa SIR le corespund două definiții. Primul prototip tratează cazul comparării a două șiruri de lungime oarecare:

```
int SIR :: compar (SIR &s1, SIR &s2)
{                               // Se compară d.p.d.v. lexicografic două șiruri de caractere
        return strcmp (s1.text, s2.text);
}
```

Funcția strcmp() cu prototip în fișierul antet **string.h**, întoarce rezultatul comparației a două șiruri de caractere. Pentru compararea primelor ncar caractere din două șiruri, definiția este următoarea:

```
int SIR :: compar (SIR &s1, SIR &s2, unsigned ncar)
{                               // Comparația primelor ncar caractere din două șiruri
        return strncmp (s1.text, s2.text, ncar);
}
```

Valoarea rezultatului comparației cu funcțiile strcmp() și strncmp() se obține prin scăderea nedistructivă a caracterelor celor două șiruri aflate pe aceeași poziție relativă. Dacă:

s1 < s2, rezultatul < 0
s1 == s2, rezultatul = 0
s1 > s2, rezultatul > 0

Al treilea membru al clasei SIR redefinește operatorul || (operatorul logic SAU), adică asociază operatorului || un sens nou. Prin construcția **void operator** || (SIR &s) se anunță compilatorul C++ că operatorului || i-a fost temporar asociat sensul descris în corpul funcției: { strcat (text, s.text); }. Instrucțiunea dintre acolade are ca efect concatenarea a două șiruri. Lista de parametri text, s.text ascunde două referințe la șirurile obiectului curent (text), respectiv al obiectului s (s.text). Se alipește deci șirul punctat de către s.text la șirul punctat de pointerul text. Funcția strcat() al cărui prototip se află în fișierul **string.h** nu întoarce nici un rezultat.

În programul principal se creează două obiecte sir1 și sir2 cu structura sir1 = a b c d \0, respectiv sir2 = a b c d e f \0. În linia rez1 = sir1.compar (sir1, sir2); se realizează compararea celor două șiruri de lungime diferită. Se va utiliza prima formă a funcției compar(). Șirurile nefiind identice, funcția va întoarce în variabila rez1 o valoare negativă. Se remarcă construcția:

nume_obiect.nume_metoda (lista_de_argumente).

Linia rez2 = sir1.compar (sir1, sir2, 4); conduce la o comparație conform celei de a doua metode, care se bazează pe funcția strncmp. Cum, primele 4 poziții ale celor două șiruri coincid, în variabila rez2 se va găsi valoarea 0. În urma concatenării, sir1 se va prelungi, în el regăsind acum valoarea a b c d a b c d e f `\0`, fapt atestat prin executarea liniei sir1.Imp ("Sir1 dupa concatenare este: "). Și aici se observă prezența construcției nume_obiect.nume_metoda (lista_de_argumente) în care operatorul "punct" (.) unește obiectul cu funcția apartenentă la clasa care la creat.

2.3. Despre moștenire, clase derivate și reutilizarea codului

Considerăm un program care conține clasa RAND_UNIF, prin care se generează numere aleatoare cu o distribuție uniformă.

```
// Exemplul 2.3. Program P2_3.CPP
// Generarea unor numere aleatoare cu distributie uniforma prin metoda liniar congruentiala
# include <limits.h>
class RAND_UNIF {
        long  x;
        void gen_nr () { x = x*1103515245 + 12345; }
public:
        RAND_UNIF (long n = 0) {x = n;}   // Constructorul clasei
        void seed (long n);
        unsigned int ui_unif () {
                gen_nr ();
                return x & LONG_MAX;
        }
        double d_unif () {
                gen_nr ();
                return (x & LONG_MAX) / (double) LONG_MAX;
        }
};
// Generarea a 10 numere aleatoare intregi si in virgula mobila, care sunt uniform distribuite
// in gamele [0...LONG_MAX] si [0...1]
# include <stdio.h>
void main (void)
{
        RAND_UNIF aleator;                  // Se creeaza obiectul aleator
        unsigned int   nai;
        double         nad;
        for (int i = 0; i < 10; i++) {
                nai = aleator.ui_unif ();
                nad = aleator.d_unif ();
        printf ("i - numerele aleatoare uniform distribuite = %d %u %f\n", i, nai, nad);
        }
}
```

În clasa RAND_UNIF, în secțiunea **private**, pe lângă variabila x de tip **long** este încapsulată și funcția gen_nr() de tip **inline**. Funcția membru ui_unif() este definită în modul **inline**. Ea apelează la serviciile funcției gen_nr(), cu care generează numere întregi fără semn cu relația: x = x*1103515245 + 12345. Deoarece prima dată seed() este 0, x va fi egal cu 12345. Valorii x i se aplică o mască binară LONG_MAX (o constantă definită în fișierul antet **limits.h**, a cărei valoare depinde de tipul calculatorului; de exemplu, în cazul unui calculator care permite un întreg lung pe 32 biți, LONG_MAX = 2^{31}-1). Datorită acestei "măști" se asigură tăiere bitului de semn.

În cadrul funcției main() se creează mai întâi obiectul cu numele aleator, după care, utilizând bucla **for** se generează primele 10 numere aleatoare.

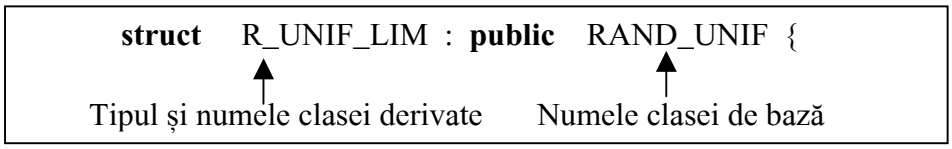
Să presupunem acum că dorim să generăm numere aleatoare uniform distribuite, însă între două limite stabilite de noi. Mai mult, nu vrem să modificăm clasa RAND_UNIF, ci să utilizăm atât structura de date, cât și codul anterior. De asemenea, se dorește realizarea unui generator de numere cu o distribuție exponențială în jurul unei valori medii.
Pentru aceasta, vom alcătui două clase "derivate" din clasa de bază RAND_UNIF. Listingul în care apar aceste clase denumite R_UNIF_LIM și R_EXP_LIM este prezentat în Exemplul 2.4.

```
// Exemplul 2.4. Program P2_4.CPP
// Declararea si definirea clasei de baza pentru generarea unor numere aleatoare cu distributie
// uniforma prin metoda liniar congruentiala
# include <limits.h>
# include <math.h>
class RAND_UNIF {
        long  x;
        void gen_nr () { x = x*1103515245 + 12345; }
public:
        RAND_UNIF (long n = 0) {x = n;}          // Constructorul clasei
        void seed (long n);
        unsigned int ui_unif () {
                gen_nr ();
                return x & LONG_MAX;
        }
        double d_unif () {
                gen_nr ();
                return (x & LONG_MAX) / (double) LONG_MAX;
        }
};
// Declararea si definirea claselor derivate
struct R_UNIF_LIM : public RAND_UNIF {
        int  inf, sup;
        R_UNIF_LIM (int i, int s) {inf = i; sup = s;}
        unsigned int  ui_unif ();
};
struct R_EXP_LIM :  public RAND_UNIF {
        int  media;
        R_EXP_LIM (int m) { media = m; }
        unsigned int  ui_unif ();
};
unsigned int R_UNIF_LIM :: ui_unif ()            // Se redefineste ui_unif()
{
    return (unsigned int) ((RAND_UNIF::ui_unif () % (sup-inf)) + inf);
}
unsigned int R_EXP_LIM :: ui_unif ()             // Se redefineste ui_unif()
{
    return (unsigned int) (-media * log(1.0-RAND_UNIF::d_unif ()) + 0.5);
}
```

```
// Generarea a 100 numere aleatoare uniform distribuite in gama [INF...SUP]
// si exponential distribuite in jurul valorii MEDIA
# define  INF             10
# define  SUP            100
# define  MEDIA          500
# include <stdio.h>
void main (void)
{
        R_UNIF_LIM    n(INF,SUP);   // Se creeaza obiectul n
        R_EXP_LIM     m( MEDIA );   // Se creeaza obiectul m
        unsigned int      nai;
        unsigned int      naexp;
        for (int i = 0; i < 100; i++) {
                nai   = n.ui_unif ();
                naexp = m.ui_unif ();
                printf ("i - numerele aleatoare uniform distribuite si exponential distribuite\
                = %3d %3u %4u \n", i, nai, naexp );
        }
}
```

În acest exemplu se evidențiază modul declarării unei clase derivate, astfel:

struct R_UNIF_LIM : **public** RAND_UNIF{



Tipul clasei derivate este **struct**, adică structură. Variabilele inf și sup ale structurii sunt publice, deci direct accesibile din program. Se observă că funcția ui_unif() va fi reutilizată de către cele două clase derivate, conform definițiilor precizate mai sus. Cele două redefiniri ale funcției ui_unif() permit determinarea distribuțiilor menționate mai sus. De fapt, pentru determinarea unor numere uniform distribuite în gama [INF...SUP] și exponențial distribuite în jurul valorii MEDIA, am *moștenit* metoda de determinare a numerelor uniform distribuite între limitele 0, ..., LONG_MAX și cu ea am creat două metode noi pentru determinarea distribuțiilor precizate.

2.4. Caracteristici ale limbajului C++ - limbaj orientat pe obiecte

1. *Clasa* este o structură de date abstractă, un șablon, care, pe lângă date, conține și funcții, fapt neobișnuit în limbajul C;
2. *Obiectul* este creat de către producătorul (constructorul) clasei conform șablonului acesteia;
3. *Mecanismul de ascundere* a datelor constă în tehnica de plasare a valorilor în variabilele de tip **private** ale obiectului;
4. *Moștenirea* este proprietatea unui obiect de a prelua anumite proprietăți ale obiectelor apartenente unei clase de bază (ierarhic superioare).
5. Obiectele comunică între ele prin mesaje. Funcțiile membre ale clasei (metodele) interpretează mesajele (valorile argumentelor) și asigură "comportarea" corespunzătoare a obiectelor. Polimorfismul, o noțiune care este corelată cu comportamentul diferit al unei metode în raport cu tipul obiectului va fi explicat în cursurile următoare.
6. *Constructorul* și *destructorul* sunt funcții speciale ale unei clase. Constructorul este automat apelat la crearea obiectelor, iar destructorul eliberează resursele de memorie ocupate de acel obiect la momentul încheierii existenței acelui obiect.