C++

limbaj creat pentru programarea orientată pe obiecte

Bibliografie

- H. Schildt: C++ manual complet, Teora, 2000
- Bjarne Stroustrup: The C++ Programming Language, Adisson-Wesley, 3nd edition, 1997
- K. Jamsa, L. Klander "Totul despre C şi C++", Editura Teora, Bucureşti, 2002 (traducere din limba engleză: C/C++ Programmer's Bible, Jamsa Press 2001)

Resurse on-line

- Object-Oriented System Development by Dennis de Champeaux, Douglas Lea, and Penelope Faure (http://g.oswego.edu/dl/oosdw3/)
- Peter Muller: Introduction to Object-Oriented Programming Using C++ (http://www.gnacademy.org/uu-gna/text/cc/material.html)
- Bruce Eckel: Thinking in C++, 2nd Edition (http://www.bruceeckel.com/)
- Online C++ tutorial (http://www.intap.net/~drw/cpp/index.htm)

Limbajul de programare C

- Învăţare prin exemple.
- Ex. 1:

```
#include <stdio.h>
main()
{
printf("Salut, cum te simti azi?
  \n");
}
```

Observaţie:

- Orice program în C începe să se execute la întâlnirea funcţiei *main*
- Instrucţiunile unei funcţii sunt închise între acolade { }
- Secvenţa \n reprezintă *rând nou;* alte secvenţe: \t (tab), \b (backspace), \\ (backslash), etc.

Limbajul de programare C

Instrucţiuni

- Instrucţiune Condiţională

  `if (`*expresie*`)` *instructiune* `[`*`else`* *instructiune*`]`

- Instrucţiuni Repetitive

  `while (`*expresie*`)` *instructiune*

Limbajul de programare C

Exemplu: căutare binară

```
/*cautbin: cauta x in
v[0]<=v[1]<=...<=v[n-1]*/

int cautbin(int x,int v[],
int n)
{
int prim, ultim, mijl;
prim=0;
ultim=n-1;
```

```
while (prim<=ultim)
   {
   mijl=(prim+ultim)/2;
   if (x<v[mijl])
      ultim=mijl-1;
      else if (x>v[mijl])

         prim=mijl+1;
      else /* s-a gasit*/
      return mijl;
      }
   return -1; /*nu s-a
   gasit*/
   }
```

Limbajul de programare C

- Instrucţiuni Repetitive

  `for` şi `while`

  Sintaxa:

  `for(`*expr1; expr2; expr3*`)` *instructiune*

  Echivalentă cu:

```
expr1;
while(expr2)
{instructiune
 expr3
 }
```

Limbajul de programare C

`for(`*initializare*`;`*conditie*`;`*incrementare*`)` *instructiune*

initializare şi *incrementare* sunt atribuiri sau apeluri de funcţie (cel mai adesea), *condiţie* este expresie relaţională

Oricare din cele 3 expresii poate lipsi (nu şi *;*)

Ce se întâmplă daca lipseşte *expr1, expr2* sau *expr3*?

Limbajul de programare C

- Instrucţiuni Repetitive

`do` *instructiune* `while (`*expresie*`);`

Exemplu:

```
void itoa (int n,
  char s[])
{
int i, semn;
if ((semn=n)<0)
  n=-n;
i=0;
do{
s[i++]=n%10+'0';
} while
  ((n/=10)>0);
if (semn<0)
   s[i++]='-';
s[i]='\0';
}
```

Limbajul de programare C

- Instrucţiune Condiţională

```
switch (expresie) {
 case expr-const: instructiuni
            ...
 case expr-const: instructiuni
[default: instructiuni]
 }
```

Limbajul de programare C

Exemplu

```
#include <stdio.h>
main() /* numara cifre,
spatii, altele*/
{
int c, i, nalb, nalte,
ncifra[10];
nalb=nalte=0;
for (i=0;i<10;i++)
ncifra[i]=0;
while
((c=getchar())!=EOF)
   {
   switch (c ) {
   case '0': case'1':
   case'2': case'3':
   case'4': case'5':
   case'6': case'7':
   case'8': case'9':
   ncifra[c-'0']++;
   break;
   case ' ':
   case '\n':
   case '\t':
   nalb++;
   break;
   default: nalte++;
   break;
   }
    }
printf("cifre =");
for (i=0; i<10;i++)
printf("%d", ncifra[i]);
printf(", spatiu alb = %d,
altul = %d\n",
nalb, nalte);
return 0;
}
```

Limbajul de programare C

`break` şi *`continue`*

- `break`: pentru a închide un *case* într-un *switch* sau pentru a ieşi imediat dintr-o buclă
- `continue`: forţează trecerea la următoarea iteraţie a buclei.

Exemplu:

```
#include <stdio.h>
void main(void)
{
char s[80], *sir;
int spatiu;
printf("introduceti un sir
");
gets(s);
sir=s;
for(spatiu=0; *sir; sir++){
   if (*sir!=' ') continue;
   spatiu++;
}
printf("%d spatii \n",
spatiu);
}
```

Limbajul de programare C

Tablouri şi şiruri

Toate tablourile au 0 ca indice pentru primul element
Exemplu:

```
void(main)
{
int x[100];
int t;
for(t=0;t<100;+
+t) x[t]=t;
}
```

- C nu controlează limitele unui tablou
- Programatorul controlează limitele acolo unde există!
- Tablourile unidimensionale sunt liste de informaţii de acelaşi tip stocate în zone contigue

Limbajul de programare C

Pointer la tablou

Un pointer la primul element al tabloului:

```
int *p
int proba[9];
p=proba;
```

Acest program atribuie lui `p` adresa primului element din `proba`
În C nu se poate transmite un tablou întreg ca argument al unei funcţii.

Următorul cod introduce adresa lui `i` în `func1()`:

```
void main(void)
{
int i[10];
func1(i);
...
}
```

`func1()` poate fi declarată astfel:

```
void func1(int *x){}
//pointer
void func1(int x[9]){}
//tablou cu dimensiune
void func1(int x[]){}
//tablou fara dimensiune
```

Limbajul de programare C

Şiruri

În C un şir este definit ca un tablou de caractere care se termină cu un caracter nul.
Un nul este specificat ca `'\0'` si are valoarea 0.
Tablourile de tip caracter trebuie declarate cu un caracter mai mult decat cel mai lung şir pe care îl vor conţine.

Limbajul de programare C

Funcţii de manevrare a şirurilor

`strcpy(s1,s2)` copiază `s1` în `s2`

`strcat(s1,s2)` concatenează `s2` la sfârşitul lui `s1`

`strlen(s1)` returnează lungimea lui `s1`

`strcmp(s1,s2)` 0 dacă `s1` si `s2` sunt identice, negativ dacă `s1<s2`, pozitiv altfel

`strchr(s1,c)` pointer la prima apariţie a lui `c` în `s1`

`strstr(s1,s2)` pointer la prima apariţie a lui `s2` în `s1`

Limbajul de programare C

Exemplu

```
#include <stdio.h>
#include <string.h>
void main(void)
{
char s1[0], s2[80];
gets(s1);
gets(s2);
printf("lungimi: %d
%d", strlen(s1),
strlen(s2));
if(!strcmp(s1,s2))
printf("siruri
egale\n");
strcat(s1,s2);
printf("%s", s1);
strcpy(s1,"acesta este
un test. \n");
printf("%s\n",s1);
if(strchr("hello",'e')
) printf("e este in
hello\n");
if(strstr("la
revedere", "la"))
printf("am gasit la");
}
```

Limbajul de programare C

Tablouri multidimensionale

Declarare, exemplu:

```
int d[10][20];
```

Accesare, exemplu

```
d[1][2];
```

Când un tablou este utilizat ca argument al unei funcţii, este transmis doar un pointer către primul element al tabloului.

Trebuie însa definit neapărat numărul de coloane ale tabloului!

Exemplu:

```
void func1(int x[][10]){ }
```

Limbajul de programare C

Tablouri de şiruri

Similar unui tablou obişnuit:

```
char matrice_siruri[10][80];
```

Pentru a accesa un şir individual:

```
gets(matrice_siruri[2]);
```

apelează funcţia gets pentru al treilea şir din tablou.

Limbajul de programare C

Iniţializarea tablourilor

C permite iniţializarea tablourilor în acelaşi timp cu declararea lor.

Exemplu:

```
int i[5]={1, 2, 3, 4, 5};
```

Tablourile de caractere care conţin şiruri permit o iniţializare prescurtată:

Exemplu:

```
char str[15]="Buna dimineata";
```

Este similar cu a scrie:

```
char sir[15]={B', 'u', 'n', 'a', ' ',
'd', 'i', ... };
```

Limbajul de programare C

Iniţializarea tablourilor multidimensionale

Similar cu cele unidimensionale.

Exemplu:

```
int patrat[5][2]=
{1,1,2,4,3,9,4,16
,5,25};
```

Iniţializarea tablourilor fără mărime:

Exemplu:

```
char e1[]="eroare
de citire\n";
char e2[]="eroare
de scriere\n";
```

Rezultat:

```
printf("%s are
marimea %d", e2,
sizeof e2);
```

Va afişa:

eroare de scriere are marimea 18

C++. Privire de ansamblu

Scopul lui C++ este acela de a stăpâni programele complexe (cu peste 100.000 de linii)

În general, când se programează în modul orientat pe obiecte, o problemă se împarte în subgrupe de secţiuni înrudite care ţin atât de codul cât şi de datele corespunzătoare din fiecare grup.

Se organizează aceste subgrupe într-o structură ierarhică

Se transforma în unităţi de sine stătătoare numite *obiecte.*

Programare în stilul C++

Stilul de programare al C++ ne ajută să gândim în C++.

```
#include <iostream.h>
main()
{
   int i;
   cout<<"iesire\n"; //comentariu
   cout<<" introduceti un numar";
   cin>>i;
   cout<<i<<"la patrat este"<<i*i<<"\n";
   return 0;
}
```

C++

Observaţii

În C++ utilizarea argumentului `void` este inutilă.

Returnarea unui 0 (dupa return) indică terminarea normală a programului. Terminarea sa anormală trebuie semnalată printr-o valoare diferită de 0.

C++
Despre I/O

```
#include <iostream.h>
main()
{
   float f;
   char sir[80];
   double d;
   cout<<"Introduceti
   doua numere in
   virgula mobila:";
   cin>>f>>d;
   cout<<"Introduceti
   un sir";
   cin>>sir;
   cout<<f<<"
   "<<d<<sir;
   return 0;
}
```

Dacă introducem *Acesta este un test* se va afişa doar *Acesta.*

Restul şirului nu este afişat deoarece operatorul >> încheie citirea şirului la primul caracter de spaţiu liber.

C++

Declararea variabilelor locale

Variabilele locale pot fi definite înainte de prima utilizare.

Declararea lor astfel ajută la evitarea unor efecte secundare accidentale.

Oponenţii susţin că împrăştierea variabilelor în tot programul face mai grea citirea.

C++
Clase

Pentru a crea un obiect în C++ trebuie definita mai întâi forma sa generală folosind cuvantul cheie **class.**

Exemplu: un tip numit stiva.

```
#define SIZE 100;
// este creata
clasa // numita
stiva
class stiva{
int stiv[SIZE];
int tos;
public:
void init();
void pune(int i);
int extrage();
```

O clasă poate conţine atât secţiuni private cât şi publice.

Implicit, toate elementele dintr-o clasă sunt private.

`stiv` şi `tos` sunt private, ceea ce înseamnă că o funcţie care nu face parte din clasa `stiva` nu are acces la ele.

Pentru a crea părţi publice într-o clasă trebuie declarate după cuvântul cheie `public`.

La toate datele sau funcţiile declarate după `public` pot avea acces toate celelalte funcţii din program.

C++

Funcţii şi date membre

Tendinţa este de a se limita sau chiar elimina utilizarea datelor publice (incapsulare).

Ar trebui ca toate datele să fie private, iar accesul la ele să se facă prin funcţii publice.

Funcţiile `init`, `pune` şi `extrage` sunt denumite funcţii membre, iar variabilele `stiv` şi `tos` sunt denumite date membre.

Doar funcţiile membre au acces la membrii privaţi ai clasei în care sunt declarate.

C++
Obiecte

O dată definită o clasă poate fi definit un obiect de acel tip.

Exemplu:

```
stiva primastiva;
```

Un obiect este o instanţiere a unei clase în acelaşi mod în care o variabilă este, de exemplu, un exemplar al tipului int.

O clasă este o abstractizare logica, utilizată de compilator la definirea obiectelor şi funcţiilor membre, iar un obiect este este **real** (i.e. un obiect există în fişierul executabil creat de compilator şi este încărcat în memoria claculatorului în momentul execuţiei).

C++

Clasă. Forma generală

Forma generală a unei clase:

```
class nume-clasa{
    date si functii private
public:
    date si functii publice
}lista de obiecte;
```

lista de obiecte poate fi goală.

C++

Declararea funcţiilor

În C++ toate funcţiile trebuie să aibă prototipuri. (ele nu sunt opţionale)

Când se scrie codul unei funcţii trebuie spus compilatorului cărei clase aparţine aceasta.

Exemplu:

```
void stiva::pune(int i)
{
   if(vis==SIZE){
      cout<<"stiva plina";
      return;
   }
   stiv[vis]=i;
   vis++;
}
```

C++

Operatorul ::

:: este numit *operatorul de rezoluţie* (sau *de specificare a domeniului*).

El spune compilatorului că versiunea aceasta a funcţiei `pune` aparţine clasei `stiva`

Mai multe clase diferite pot să folosească acelaşi nume de funcţie (polimorfism)

C++

Referire la date/funcţii membre (1)

Dintr-o secţiune de cod ce nu face parte dintr-o clasă (de ex. din corpul programului) putem sa ne referim la un membru al unei clase dar aceasta trebuie să se facă numai **în legatură cu un obiect al acelei clase.**

Pentru aceasta folosim numele obiectului urmat de **.** (operatorul de selecţie directă) şi de numele membrului respectiv.

Este valabil atât pentru funcţii cât şi pentru date membre.

Exemplu:

```
stiva stiva1,stiva2;
stiva1.init();
```

`stiva1` **şi** `stiva2` sunt două obiecte distincte.

Referire la date/funcţii membre (2)

O functie membră poate să apeleze un alt membru sau să se refere la date membre direct, fără utilizarea operatorului de selecţie directă.

Numele obiectului şi operatorul **.** trebuie folosite doar atunci când un cod care nu aparţine clasei apelează un membru.

C++
Exemplu stiva

```
#include <iostream.h>
#define SIZE 100
Class stiva{
    int stiv[SIZE];
    int vis;
public:
void init();
void pune(int i);
int scoate();
};
void stiva::init()
{
    vis=0;
}
void stiva::pune(int i)
{
    if(vis==SIZE){
        cout<<"stiva plina";
        return;
    }
stiv[vis]=i;
vis++;
}
```

```
int stiva::scoate()
{
    if(vis==0){
        cout<<"stiva vida";
        return 0;
    }
    vis--;
    return stiv[vis];
}
main()
{
stiva stiva1, stiva2; //creez doua
obiecte stiva
stiva1.init();
stiva2.init();
stiva1.pune(1);
stiva2.pune(2);
stiva1.pune(3);
stiva2.pune(4);
cout<<stiva1.scoate()<<" ";
cout<<stiva1.scoate()<<" ";
cout<<stiva2.scoate()<<" ";
cout<<stiva2.scoate()<<" \n";
return 0;
}
```

Observaţii

Elementele private ale unui obiect sunt accesibile doar *funcţiilor* care sunt membre ale acelui obiect.

Exemplu:

```
stiva1.vis=0 //eroare
```

Nu poate exista în interiorul lui `main` deoarece `vis` este privat.

C++

Clase şi Obiecte

Clasele formează baza programării pe obiecte.

O clasă este folosită pentru a defini natura unui obiect.

Clasa este unitatea de bază pentru incapsulare.

Incapsulare: mecanismul prin care sunt legate împreună cod şi date şi ambele sunt păstrate în siguranţă faţă de intervenţii din afară şi utilizări greşite.

C++

Clase

O declarare a unei clase defineşte un nou tip care leagă împreună date şi cod

Forma generală a unei clase care nu *moşteneşte* nicio altă clasă:

```
class nume-clasa{
  date si functii private
specificator de acces:
  date si functii
specificator de acces:
  date si functii
...
specificator de acces:
  date si functii
}lista de obiecte;
```

C++

Clase

Specificatori de acces:

`public`, `private`, `protected`.

`protected` este necesar când este implicată moştenirea.

Specificatorul de tip poate fi modificat într-o clasă de câte ori se doreşte.

C++

Membri de tip `static`

Faptul că declaraţia unei variabile este precedată de `static` înseamnă că va exista doar o copie a acelei variabile şi toate obiectele o vor folosi.

Nu se vor crea copii individuale ale variabilelor de tip `static`.

Toate obiectele acelei clase vor folosi aceeaşi variabilă.

Când este creat primul obiect toate variabilele de tip `static` sunt iniţializate de compilator cu 0 (sau şir vid, sau pointer NULL, sau obiect cu toate datele nule, după caz).

C++

Când se declară o variabilă de tip static, ea nu se defineşte. De aceea trebuie definită în afara clasei.

Exemplu:

```
#include<iostream.h>
class comun {
static int a;
int b;
public:
void da(int i, int
j) {a=i;b=j;}
void arata();
};
```

```
int comun::a;
//definim a
void comun::arata()
{
cout<<"a
static:"<<a;
cout<<"\n b ne-
static:"<<b;
cout<<"\n";
}
```

C++

Exemplu (continuare):

```
main()
{
comun x,y;
x.da(1,1);
x.arata();
y.da(2,2);
y.arata();
x.arata();
return 0;
}
```

Output:

```
a static: 1
b ne-static: 1
a static: 2
b ne-static: 2
a static: 2
b ne-static: 1
```

C++

Variabile `static`

O variabilă membru static există înainte de a fi creat orice obiect din acea clasă.
Aceasta înseamnă că i se poate da oricând o valoare.

Exemplu:

```
#include
<iostream.h>
class comun{
public:
static int a;
};
```

```
int comun::a;
main()
{
   comun::a=99;
   cout<<"a initial"<<comun::a;
   cout<<"\n";
   comun x;
   cout<<"x.a este"<<x.a
   return 0;
}
```

C++

Observaţii

Folosind variabilele membre de tip `static`, virtual nu mai avem nevoie de variabile globale.

Variabilele globale încalcă principiul incapsulării.

C++

Funcţii statice

Funcţiile membre pot fi de asemenea statice.

Restricţii:

- Ele au acces doar la alţi membri de tip `static` ai clasei şi la datele şi funcţiile globale
- Nu pot exista o variantă statică şi una nestatică a aceleiaşi funcţii

C++

Funcţiile membre statice au aplicaţii limitate

O bună utilizare a lor este aceea că pot preiniţializa datele private de tip static, înainte de crearea efectivă a vreunui obiect.

```
#include
<iostream.h>
class tip-
static{
static int i;
public:
static void
init(int x)
{i=x};
void arata()
{cout<<i;}
};
```

```
int  tip_static::i;
main()
{
tip-
static::init(100);
tip-static x;
x.arata();// afiseaza
100
return 0;
}
```

C++

Operatorul `::`

În general `::` este folosit la asocierea unui nume de clasă cu un nume de membru pentru a spune compilatorului cărei clase îi aparţine acel membru.

El mai poate permite accesul la un nume dintr-un domeniu închis, nume care este ascuns de o declarare locală a aceluiaşi nume.

C++

: : Exemplu

```
...
int i; //i global
void f()
{
int i; //i local
i=10;  //i local
...
}
```

Ce se întamplă dacă f are nevoie de acces la `i` global?

Soluţie:

```
int i; //i global
void f()
{
int i;  //i local
::i=10; //i global
...
}
```

C++

Clase imbricate

Este posibil să definim o clasă în interiorul alteia, creând clase imbricate.

O clasă imbricată este validă doar în interiorul clasei ce o conţine.

Datorită mecanismului de moştenire, practic nu este nevoie de clase imbricate.

C++

Constructori şi destructori

Unele părţi ale unui obiect pot fi iniţializate înainte de a fi folosite.

C++ permite obiectelor să se iniţializeze atunci când sunt create.

Această iniţializare se realizează prin intermediul unei funcţii constructor.

O funcţie constructor este o funcţie specială a unei clase şi are acelaşi nume cu acea clasă.

C++

Exemplu

```
class stiva{
   int
stiv[SIZE];
   int vis;
public:
stiva();
void pune(int
i);
int scoate();
};
```

Functia constructor `stiva()` nu are specificat un tip de returnat.

În C++ aceste funcţii nu pot să returneze valori. Functia `stiva()` are codul:

```
stiva::stiva()
{
vis=0;
cout<<"stiva
initializata";
}
```

Un constructor al unui obiect este apelat imediat atunci cand este creat acel obiect.

C++

Destructor

Obiectele locale sunt create când se intra în blocul lor şi sunt distruse când blocul este părasit.
Obiectele globale sunt distruse când se termină programul.

Exemplu:

Funcţia destructor este apelată când obiectul este distrus. Numele destructorului este numele clasei precedat de ~. Destructolul nu are argumente.

```
class stiva{
   int stiv[SIZE];
   int vis;
public:
stiva();//constructor
~stiva(); //destructor
void pune(int i);
int scoate();
};
// functia destructor
stiva::~stiva()
{
cout<<"stiva
distrusa";
}
```

C++

Exemplu `stiva`

```
#include
  <iostream.h>
#define SIZE 100
class stiva{
  int stiv[SIZE];
  int vis;
public:
stiva();
~stiva();
void pune(int i);
int scoate();
};
```

```
stiva::stiva()
{
   vis=0;
   cout<<"stiva
initializata";
}
~ stiva::stiva()
{
cout<<"stiva
distrusa\n";
}
```

C++

Exemplu `stiva` (continuare)

```
void
stiva::pune(int
i)
{
    if(vis==SIZE){
     cout<<"stiva
plina";
     return;
    }
    stiv[vis]=i;
    vis++;
}
```

```
int
  stiva::scoate()
{
  if(vis==0){
   cout<<"stiva
  vida";
   return 0;
  }
  vis--;
  return
  stiv[vis];
}
```

C++

Exemplu `stiva` (continuare)

```
main()
{
stiva stiva1, stiva2;
//creez doua obiecte stiva
stiva1.pune(1);
stiva2.pune(2);
stiva1.pune(3);
stiva2.pune(4);
cout<<stiva1.scoate()<<"
";
cout<<stiva1.scoate()<<"
";
cout<<stiva2.scoate()<<"
";
cout<<stiva2.scoate()<<"
\n";
return 0;
}
```

```
Output:
stiva initializata
stiva initializata
3 1 4 2
stiva distrusa
stiva distrusa
```

Constructori cu argumente

Este posibilă transmiterea unor argumente către funcţia constructor.

Tipic, aceste argumente sunt folosite pentru a contribui la iniţializarea unui obiect când este creat.

Pentru a crea un constructor cu argumente îi adăugăm argumente în acelaşi mod în care adaugăm oricărei alte funcţii.

C++

Exemplu

```
#include<iostream.h>
class clasamea{
int a,b;
public:
clasamea(int i, int
j) {a=i; b=j;}
void arata()
{cout<<a<<" "<<b;};
main()
{
clasamea ob(3,5);
ob.arata();
return 0;
}
```

Instrucţiunea `clasamea ob(3,5)` determină crearea unui obiect `ob`
şi se pasează argumentele `3` şi 4 către argumentele `i` şi `j` din `clasamea`
Funcţiile constructor cu argumente sunt foarte folositoare deoarece permit să evităm apelarea unei funcţii în plus doar pentru a iniţializa una sau mai multe variabile dintr-un obiect.

C++

Constructor cu un argument

Putem pasa o valoare iniţială unei funcţii constructor cu un argument astfel:

```
#include <iostream.h>
class X{
int a;
public:
X(int j){a=j;}
int daa() {return a;}
};
main()
{
X ob=99; //paseaza 99 lui j
cout<<ob.daa(); //afiseaza 99
return 0;
}
```

C++

Constructor de copiere

Un constructor al unei clase care are ca unic argument o referinţă la un obiect al clasei respecive se numeşte *constructor de copiere*. În mod normal el are ca scop crearea unui obiect care are aceleaşi date ca argumentul. Transmiterea argumentului ca *referinţă* şi nu ca valoare este obligatorie.

C++

Constructor de copiere

Dacă nu este definit de programator, compilatorul defineşte un *constructor de copiere implicit*, care efectuează copierea datelor argumentului în datele obiectului construit. Acesta poate să nu facă transmiterea datelor în mod convenabil (de exemplu, în cazul unei date pointer este copiat pointerul, dar nu se alocă o zonă separată de memorie pentru datele la care trimite pointerul, astfel încât orice schimbare ulterioară a datelor obiectului utilizat ca argument va produce şi schimbarea datelor obiectului construit). De aceea este recomandabil să se definească pentru orice clasă un astfel de constructor.

C++
Constructor de copiere

Un constructor de copiere se utilizează:

În mod *explicit* de către programator prin declarări de obiecte însoţite de iniţializări cu sintaxa:

tip `ob1=ob0;` sau *tip* `ob1(ob0);`

unde *tip* este o clasă, iar `ob1` este creat prin apelarea constructorului de copiere având ca argument efectiv obiectul `ob0` al aceleiaşi clase, creat anterior.

C++

Constructor de copiere

Un constructor de copiere se utilizează:

În mod *implicit* de către compilator când:

a) se copiază argumentele efective transmise prin valoare la apelarea unei funcţii (de aceea argumentul constructorului de copiere se transmite ca referinţă, transmiterea prin valoare producând apelare infinită)

şi

b) când se întoarce valoarea unei funcţii ca variabilă temporară în care se copiază un obiect al unei clase.

Pentru a se evita utilizarea constructorului de copiere se va utiliza întoarcerea valorilor funcţiilor ca referinţe.

C++

Clase locale

O clasă poate fi definită în interiorul unei funcţii.

Ea este cunoscută doar acelei funcţii.

Restricţii:

- Toate funcţiile membre trebuie definite în interiorul declaraţiei de clasă.
- Clasa locală nu poate să folosească sau sa aibă acces la variabilele locale ale funcţiei în care este declarată.
- În interiorul unei clase locale nu poate fi declarată nicio variabila de tip static.

Clasele locale nu sunt uzuale în C++.

C++
Clase locale. Exemplu

```
#include
<iostream.h>
void f();
main()
{
f();
return 0;
}
void f()
{
class clasa-mea{
int i;
public:
void pune-i (int n)
{i=n;}
int da-i(){return i;}
} ob;
ob.pune-i(10);
cout<<ob.da-i();
}
```

C++

Transmiterea obiectelor către funcţii

Obiectele pot fi transmise funcţiilor exact ca orice altă variabilă.

Obiectele sunt transmise prin mecanismul standard apelare prin valoare, deci prin copiere în obiecte temporare, create şi distruse automat.

Efectuarea unei copii înseamnă practic crearea unui alt obiect.

Q2. Întrebare: este executată funcţia constructor de copiere la crearea copiei şi este executat destructorul la distrugerea copiei? Motivaţi, exemplificaţi cu un program (se explică astfel faptul că argumentul constructorului de copiere este referință).

C++

Transmiterea obiectelor către funcţii

```
#include <iostream.h>
class clasamea{
int i;
public:
clasamea(int n);
~clasamea();
void pune-i(int n){i=n;}
int da-i () {return i;}
};
clasamea::clasamea(int n)
{
i=n;
cout<<"construieste"<<i<<
"\n";
}
```

```
clasamea::~clasamea()
{
cout<<"distruge"<<i<<"\n";
}
void f(clasamea ob);
main()
{
clasamea o(1);
f(o);
Cout<<"acesta este i din
main";
Cout<<o.da-i()<<"\n";
Return 0;
}
Void f(clasamea ob)
{
ob.pune-i(2);
cout<<"i local:"<<ob.da-
i()<<"\n";
}
```

C++

Moştenire

Moştenirea permite crearea clasificărilor ierarhice.

O clasă care este moştenită se numeşte clasă de bază.

Clasa care moşteneşte se numeşte clasă derivată.

O clasă derivată poate fi folosită ca una de bază pentru o altă clasă derivată.

C++

Controlul accesului la clasa de bază

Forma generală a moştenirii:

```
class clasa-derivata : acces
clasa_baza {
//corpul clasei
}
```

Membrii clasei de bază devin membri ai clasei derivate.

Accesul la membrii clasei din interiorul clasei derivate este determinat prin *acces*.

Specificatorul acces poate fi `public`, `private` sau `protected`.

Când nu este specificat, el este `private`.

C++

Specificatori de acces

Dacă specificatorul este `public`, toţi membrii `public` ai clasei de bază devin membri `public` ai clasei derivate, iar toţi membrii `protected` ai bazei devin membri `protected` ai clasei derivate.
Toţi membrii `private` ai bazei ramân `private` pentru clasa de bază, deci nu sunt accesibili membrilor clasei derivate.

C++

Exemplu

```
#include <iostream.h>
class baza{
int i,j;
public:
   void pune(int a, int
b){i=a; j=b;}
   void arata() {cout<<i<<"
"<<j<<"\n";}
};
class derivata: public
baza{
int k;
public:
derivata(int x){k=x;}
void aratak()
{cout<<k<<"\n";}
};
```

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

C++

Specificatorul `private`

Când clasa de bază este moştenită prin specificatorul `private`, toţi membrii `public` şi `protected` ai clasei de bază devin membri `private` ai clasei derivate.

Programul următor nu va fi nici măcar compilat:

C++

```
#include <iostream>
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
sunt
//private in derivata
class derivata: private baza {
int k;
public:
derivata (int x){k=x;}
void aratak(){cout <<k <<"\n";}
};
```

```
int main()
{
derivata ob(3);
ob.pune(1,2);//eroare,
nu poate avea acces
ob.arata();// eroare, nu
poate avea acces
 return 0;
}
```

C++

Moştenirea şi membrii `protected`

Când un membru este `protected`, el nu este accesibil altor elemente ale programului care nu sunt membri ai clasei.

Accesul la un membru `protected` este acelaşi ca şi la unul `private` (cu o singura excepţie) i.e. el este accesibil doar altor membri din aceeaşi clasă cu el.

Excepţie: cazul când membrul `protected` este moştenit.

C++

Comportarea membrilor `protected`

Când clasa de bază este moştenită ca `public`, membrii `protected` ai acesteia devin `protected` în clasa derivată şi, prin urmare, accesibili acesteia. Cu alte cuvinte, folosind `protected` putem crea membri ai unei clase care sunt membri privaţi ai clasei lor, dar care pot fi moşteniţi şi sunt accesibili unei clase derivate.

C++
Exemplu

```
#include <iostream>
class baza {
protected:
int i,j; //privati pt baza, dar
//accesibili in derivata
public:
void pune(int a,int b){i=a;j=b;}
void arata() {cout <<i <<" "<<j
<<"\n";}
};
class derivata : public baza {
int k;
public:
//derivata are acces la i si j
din baza
void punek {k=i*j;}
void aratak(){cout <<k <<"\n";}
};
```

```
main()
{
derivata ob;
ob.pune(2,3);//cunoscut lui
derivata
ob.arata();// cunoscut lui
derivata

ob.punek();
ob.aratak();
 return 0;
}
```

Deoarece `baza` este moştenită ca `public`, `i` şi `j` fiind `protected`, `punek()` poate avea acces la ele.

Dacă `i` şi `j` erau `private` în `baza`, `derivata` nu avea acces la ele şi nu se compila.

C++

`protected`

Când o clasă derivată este folosită ca o clasă de bază pentru o altă clasă derivată orice membru `protected` al clasei de bază iniţiale care este moştenită (ca `public`) de către prima clasă derivată poate fi moştenit ca *`protected`* şi de cea de a doua clasă derivată.

Programul din următorul exemplu este corect (`derivata2` poate avea acces la `i` şi `j`):

C++
Exemplu

```
#include <iostream>
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
// protected
class derivata1: public baza {
int k;
public:
void punek(){k =i*j;} //corect
void aratak(){cout<<k<<"\n";}
};
// i si j mosteniti indirect
// prin derivata1
class derivata2: public
derivata1 {
int m;
public:
void punem(){m =i-j;}  //corect
void aratam(){cout <<m<<"\n";}
};
int main() {
   derivata1 ob1;
   derivata2 ob2;
   ob1.pune(2,3);ob1.arata();
   ob1.punek(); ob1.aratak();
   ob2.pune(3,4); ob2.arata();
   ob2.punek(); ob2.aratak();
   ob2.aratam(); ob2.punem();
   return 0;
}
```

Dacă `baza` este moştenită ca `private`, toţi membrii din `baza` ar deveni membri `private` ai lui `derivata1`, ceea ce înseamnă că ei nu sunt accesibili lui `derivata2`. (`i` si `j` sunt în continuare accesibili lui `derivata1`).

Moştenirea `protected` a bazei

Când se moşteneşte o clasă de bază ca `protected`, toţi membrii `public` şi `protected` ai bazei devin membri `protected` ai clasei derivate.

C++

Drepturile de acces într-o subclasă ai membrilor unei supraclase în funcţie de tipul de moştenire este arătat complet mai jos

	Tip de moştenire		
	`private`	**`protected`**	**`public`**
`private`	**inaccesibil**	**inaccesibil**	**inaccesibil**
`protected`	**`private`**	**`protected`**	**`protected`**
`public`	**`private`**	**`protected`**	**`public`**

Coloana din stânga listează drepturi posibile de acces pentru membrii supraclaselor. Următoarele coloane arată drepturile de acces care rezultă pentru membrii unei supraclase în funcţie de tipul de moştenire.

C++

Constructori şi destructori

- Primul este executat constructorul pentru bază, urmat de cel pentru derivată.
- Este apelat destructorul pentru derivată, apoi cel pentru bază.
- Când este creat / eliminat un obiect al clasei derivate, ordinea constructorilor / destructorilor este cea anterioară.

C++

Când sunt moştenite clase de bază multiple, putem întâlni o ambiguitate:

```
#include <iostream>
using namespace std;
class baza {
public:
int i;
};
//derivata1 mosteneste baza.
class derivata1 :public baza {
public:
int j;
};
//derivata2 mosteneste baza.
class derivata2 :public baza {
public:
int k;
};
/* derivata3 mosteneste atat
derivata1 cat si derivata2.
Aceasta inseamna ca exista
doua copii ale  baza in
derivata3! */
class derivata3 :public derivata1,
public derivata2 {
public:
int sum;
};
int main()
{derivata3 ob;
ob.i =10;// ambiguitate, care i???
ob.j =20;
ob.k =30;
// i ambiguu si aici
ob.sum =ob.i +ob.j +ob.k;
// iar ambiguitate, care i???
cout <<ob.i <<"";
cout <<ob.j <<""<<ob.k <<"";
cout <<ob.sum;
return 0;
}
```

Ambiguitate

`derivata1` şi `derivata2` moştenesc `baza`; `derivata3` moşteneşte atât `derivata1` cât şi `derivata2`. Aceasta înseamnă ca în `derivata3` sunt 2 copii pentru `baza`.

În expresia `ob.i=10` nu e clar despre care `i` este vorba - cel din `derivata1` sau cel din `derivata2`.

Soluţie:

Aplicăm lui `i` operatorul de specificare a domeniului şi îl selectăm manual.

C++

Exemplu

```
#include <iostream>
using namespace std;
class baza {
public:
int i;
};
//derivata1 mosteneste baza.
class derivata1 :public baza {
public:
int j;
};
//derivata2 mosteneste baza.
class derivata2 :public baza {
public:
int k;
};
/* derivata3 mosteneste atat
derivata1 cat si derivata2.
Aceasta inseamna ca exista
doua copii ale  baza in
derivata3! */
class derivata3 :public derivata1,
public derivata2 {
public:
int sum;
};
int main()
{
derivata3 ob;
ob.derivata1::i =10;//domeniu
rezolvat, i din derivata1
ob.j =20;
ob.k =30;
// domeniu rezolvat
ob.sum =ob.derivata1::i +ob.j
+ob.k;
// domeniu rezolvat
cout <<ob.derivata1::i <<"";
cout <<ob.j <<""<<ob.k <<"";
cout <<ob.sum;
return 0;
}
```

Clase de bază virtuale

Problemă: ce se întâmplă când vrem efectiv doar o copie pentru bază?

Folosim clase de baza virtuale:

când două sau mai multe obiecte sunt derivate dintr-o clasă de bază comună, putem preveni prezenţa într-unul din acestea a mai multor copii ale clasei de bază, declarând baza ca `virtual` în momentul în care ea este moştenită.

C++
Exemplu

```
#include <iostream>
using namespace std;
class baza {
public:
int i;
};
// derivata1 mosteneste baza ca
// virtual.
class derivata1 :virtual public baza
{
public:
int j;
};
// derivata2 mosteneste baza ca
// virtual.
class derivata2 :virtual public baza
{
public:
int k;
};
/*derivata3 mosteneste atat
derivata1 cat si derivata2.
Acum exista numai o copie a
clasei baza*/
class derivata3 :public
derivata1, public derivata2
{
public:
int sum;
};
int main()
{
derivata3 ob;
ob.i =10;//acum neambiguu
ob.j =20;
ob.k =30;
//neambiguu
ob.sum =ob.i +ob.j +ob.k;
//neambiguu
cout <<ob.i <<"";
cout <<ob.j <<""<<ob.k
<<"";
cout <<ob.sum;
return 0;
}
```

C++

Funcţii membre virtuale

O funcţie membră virtuală este o funcţie care este declarată ca `virtual` în clasa de bază şi redefinită de o clasă derivată.

Redefinirea funcţiei în clasa derivată modifică şi are prioritate faţă de definiţia funcţiei în clasa de bază.

O funcţie virtuală declarată în clasa de bază acţionează ca un substitut pentru păstrarea datelor care specifică o clasă generală de acţiuni şi declară forma interfeţei.

O funcţie virtuală defineşte o clasă generală de acţiuni.

Redefinirea ei introduce o metodă specifică.

C++

Funcţii membre virtuale şi polimorfism

•Când sunt utilizate normal, funcţiile virtuale se comportă ca orice alta funcţie membru al clasei.

•Ceea ce le diferenţiază şi le face să admită *polimorfismul* este modul în care se comportă când sunt apelate cu pointeri.

•Un pointer al clasei de bază poate fi folosit pentru a indica spre orice altă clasă derivată din ea.

•Când un astfel de pointer indică spre un obiect derivat care conţine o funcţie virtuală, C++ determină care dintre versiunile funcţiei să fie apelată, în funcţie de tipul obiectului spre care indică acel pointer.

•Când sunt indicate obiecte diferite, sunt executate versiuni diferite ale funcţiei virtuale.

C++
Exemplu

```
#include <iostream>
using namespace std;
class baza {
public:
virtual void vfunc(){
cout <<"vfunc()din baza\n";
}
};
class derivata1 :public baza {
public:
void vfunc(){
cout <<"vfunc()din derivata1\n";
}
};
class derivata2 :public baza {
public:
void vfunc(){
cout <<"vfunc()din derivata2\n";
}
};
```

```
int main()
{
baza *p,b;
derivata1 d1;
derivata2 d2;
// indica spre baza
p =&b;
p->vfunc();// acceseaza vfunc()
            // din baza
// indica spre derivata1
p =&d1;
p->vfunc();// acceseaza vfunc()
            // din derivata1
// indica spre derivata2
p =&d2;
p->vfunc(); // acceseaza vfunc()
             // din derivata2
return 0;
}
```

C++
Rezultat

```
vfunc()din baza
vfunc()din derivata1
vfunc()din derivata2
```

Observaţii:

- Când `vfunc` este redefinit, `virtual` nu mai este necesar.
- `baza` este moştenită de `derivata1` şi `derivata2`.
- Versiunea `vfunc()` este stabilită de tipul de obiect spre care indică p.

C++

Funcţii membre virtuale

- Când o funcţie virtuală este moştenită, se moşteneşte şi natura sa virtuală.
- O funcţie ramâne virtuală indiferent de câte ori este moştenită.
- O funcţie virtuală nu trebuie neapărat suprascrisă.
- Dacă o clasă derivată nu suprascrie funcţia virtuală, atunci când un obiect din acea clasă accesează funcţia, este folosită funcţia definită de clasa de bază.

C++

Destructor virtual

La distrugerea unui obiect este apelat destructorul care corespunde tipului **declarat** al obiectului, nu tipului conţinutului acelui obiect. Aceasta înseamnă că, dacă un pointer are ca tip declarat o clasă, dar i se alocă memorie care primeşte ca valoare un obiect al unei subclase a clasei declarate, la eliberarea memoriei se va apela numai destructorul clasei de bază. Acest fapt poate să fie în multe cazuri neconvenabil, nefiind executate operaţiile necesare la distrugerea unui obiect al subclasei. Pentru a se apela şi destructorul subclasei, trebuie ca destructorul clasei de bază să fie declarat utilizându-se cuvântul cheie `virtual`.

C++

Destructor virtual

În exemplul de mai jos destructorul bazei nu este virtual, ceea ce va avea ca urmare neeliberarea memoriei ocupate de datele proprii ale clasei derivate.

C++

Destructor virtual

```
#include <iostream.h>
class Baza
{
       public:
          Baza(){ cout<<"Constructor: Baza"<<endl;}
          ~Baza(){ cout<<"Destructor : Baza"<<endl;}
};
class Derivata: public Baza
{
     // date proprii
     int n,*sir;
       public:
           Derivata(const int n){
            cout<<"Constructor: Derivata"<<endl;
         sir=new int [n]; // alocarea memoriei pentru sir
       for (int i=0; i<n; i++) sir[i]=0;
           }
           ~Derivata(){ cout<<"Destructor : Derivata"<<endl;
             delete sir; // eliberarea memoriei alocate de
// constructor
           }
};
```

C++

Destructor virtual

```
int main()
{
        Baza *Var = new Derivata(3); // constructorul
// Derivata(3) aloca memorie pentru Var->sir
        delete Var; // memoria alocata pentru Var->sir
// nu este eliberata
// deoarece se apeleaza doar destructorul ~Baza()
}
```

C++

Destructor virtual

Declararea destructorului bazei ca `virtual`, va avea ca urmare apelarea destructorului clasei derivate care eliberează memoria ocupată de datele proprii ale clasei derivate.

C++

Destructor virtual

```
#include <iostream.h>
class Baza
{
       public:
          Baza(){ cout<<"Constructor: Baza"<<endl;}
          virtual ~Baza(){ cout<<"Destructor : Baza"<<endl;}
};
class Derivata: public Baza
{
     // date proprii
     int n,*sir;
       public:
           Derivata(const int n){
            cout<<"Constructor: Derivata"<<endl;
         sir=new int [n]; // alocarea memoriei pentru sir
       for (int i=0; i<n; i++) sir[i]=0;
           }
           ~Derivata(){ cout<<"Destructor : Derivata"<<endl;
             delete sir; // eliberarea memoriei alocate de
// constructor
           }
};
```

C++

Destructor virtual

```
int main()
{
      Baza *Var = new Derivata(3); // constructorul
// Derivata(3) aloca memorie pentru Var->sir
      delete Var;// acum memoria alocata pentru Var-> sir
// este eliberata
                  // deoarece se apeleaza ambii
// destructori ~Derivata() si ~Baza()
}
```