Examen la Programarea Orientată pe Obiecte

1. Clase.

O definitie "bruta" a clasei ar fi aceea ca este un concept extins al unui tip de date abstract : in loc sa contina numai informatii – variabile – , contine si functii. Un obiect este o instantiere a unei clase. Mai precis, clasa ar fi tipul de date si obiectul ar fi variabila. La programarea obiectuala stau la baza: incapsularea (private, protected, public, published), polimorfismul si mostenirea.

Clasa reprezinta descrierea unei multimi de obiecte care au aceeasi structura si acelasi comportament. Ca urmare, intr-o clasa vom gasi definitiile datelor si ale operatiilor ce caracterizeaza obiectele clasei respective.

Clasa, intr-un anumit limbaj de programare, reprezinta definirea unui tip de obiecte abstracte sau concrete, adica descrierea proprietatilor, a datelor si a metodelor, a prelucrarilor, posibile asupra datelor. Clasa este, de fapt, o notiune abstracta, care defineste un anumit tip de obiecte, sau, altfel spus, o clasa reprezinta multimea posibila a mai multor obiecte de acelasi tip.

2. Structuri si clase.

O structură de date este un grup de elemente de date grupate împreună sub un singur nume. Aceste elemente de date, cunoscut sub numele de membri, poate avea diferite tipuri şi lungimi diferite. Primul lucru pe care trebuie să ştim este că o structură de date creează un tip nou: Odată ce o structură de date este declarată, un nou tip cu identificatorul specificat ca nume_structra este creat si poate fi folosit în restul programului ca orice alt tip de date. Implicit datele in structura sunt public pe cind in clasa private, de asemenea poate avea functii membre.

O structura este un tip de date definit de utilizator cuprinzand variabile multiple, de diferite tipuri. Structurile sunt utile pentru descrierea obiectelor. Spre deosebire de tablouri, care stocheaza mai multe valori de acelasi tip, structurile stocheaza valori de diferite tipuri.

Structura se declara folosind cuvantul rezervat struct . Continutul structurii este pus intre acolade. Dupa ultima acolada, este obligatoriu caracterul punct si virgula. Corpul structurii se compune din variabilele corelate numite variabile membru .

3. Uniuni si clase.

Uniunea aloca o porțiune corespunzătoare din memorie pentru a fi accesata ca diferite tipuri de date. Declarația sa şi folosirea este similară cu ceaa structurilor, dar funcționalitatea acesteia este cu totul diferit, toate elementele declarației de uniune ocupă acelaşi spațiu fizic în memorie, dimensiunea acesteia fiind tipul cu cea mai mare lungime din declarației. Deoarece toate dintre ele se referă la aceeaşi locație în memorie, modificarea unuia dintre elemente vor afecta valoarea tuturo, nu putem stoca valori diferite în ele independente una de cealaltă. Una dintre folosirea uniunei este de a uni un tip elementar cu un array de elemente sau structuri mai mici.

4. Uniuni anonime.

Limbajul C++ ne permite sa creem structuri anonime. Daca nu specificam un nume pentru uniune si nici nu creem un obiect la initializare, uniunea devine o uniune anonima si putem accesa membrii ei direct, folosiind numele membriilor, ca si cum uniunea nu ar exista.

Putem folosii uniuni anonime in structuri. Asta inseamna ca, putem accesa membrii uniunii ca si cum ar fii ai structuri, insa pastram caracteristicile uniunii.

```
struct carte{
string autor;
string editura;
int an_aparitie;
union {
int lei;
int dolar;
int euro; };
}o_carte;
```

5. **Functii prietene.**

Membrii private şi protected din o clasă nu pot fi accesate din afara clasei în care sunt declarate, cu toate acestea, această regulă nu afectează prietenii. Prietenii sunt funcțiile sau clasele declarate cu cuvântul cheie friend. Dacă vrem să declare o funcție externa ca prieten a clasei, această funcție va avea acces la membrii private şi protected din ea, astfel mai intii se declara prototipul functiei cu cuvintul cheie friend in corpul clase iar functia va fi externa clasei.

De aceea, o functie obisnuita poate fi utilizata pentru a prelucra obiectele unei clase, daca ea se modifica in asa fel incat sa devina functie membru. S-a facut un compromis pentru a admite accesul la elementele protejate si pentru anumite functii care nu sunt membru. Aceste functii au fost numite functii prieten (friend) pentru clasa respectiva. Ele trebuie precizate ca atare in definitia clasei. In acest scop, prototipurile lor sunt prezente in definitia clasei si sunt precedate de cuvantul cheie friend.

6. **Clase prietene.**

Aşa cum putem defini o functie prieten, putem defini si o clasă ca prieten a altei clase, astfel prima clasa va avea acces la membrii private si protected din clasa a doua.

O **clasa** este **prietena** cu o alta clasa daca ea are acces la datele membru ale acesteia.

O functie, respectiv o clasa prietena se declara utilizand cuvantul friend astfel:

a) friend Tip_functie Nume_functie (Lista_parametri_formali); // Functiefriend globala

b) friend Tip_functie Nume_clasa::Nume_functie(Lista_par._formali); // Functie friend membru

c) friend Nume_clasa; // Clasa friend

7. **Functii inline.**

Funcțiile inline este tehnica de optimizare utilizate de compilatoarelor. Se poate adauge pur şi simplu cuvântul cheie inline la prototipul functiei, astfel la fiecare apel chemarea functiei va fi inlocuita cu corpul ei.

O funcție declarată inline se schimbă la compilare cu corpul ei, şi se spune că apelul funcției se realizează prin expandare. În felul acesta se elimină operațiile suplimentare de apel şi revenire din funcție. Dezavantajul funcțiilor inline este acela căproduc creşterea dimensiunilor programului compilat, de aceea se recomandă a fi utilizate pentru funcții de dimensiuni mici(maximum 3-4 instrucțiuni). În plus, mai există şi unele restricții privind

funcțiile inline: ele nu pot fi declarate funcții externe, deci nu pot fi utilizate decât în modulul de program în care au fost definite şi nu pot conține instrucțiuni ciclice (while, for, do-while).

Sintaxa declarării unei functii inline:

inline <tip> <nume_functie>([<lp>]),

unde <tip> reprezintã tipul întors de functie, iar <lp> lista de parametri.

C++ oferă posibilitatea declarării funcțiilor inline, care combină avantajele funcțiilor propriu-zise cu cele ale macrodefinițiilor. Astfel, la fiecare apelare, corpul funcției declarate inline este inserat în codul programului de către compilator.Față de macrodefiniții (care presupun o substitutie de text într-o fază preliminară compilării), pentru funcțiile inline compilatorul inserează codul obiect al funcției la fiecare apel.Avantajul creşterii de viteza se plăteşte prin creşterea dimensiunii codului.Aşadar, funcțiile inline trebuie să fie scurte.

8. **Definirea functiilor inline într-o clasă.**

O functie membru, care este definită ca membru al clasei se numeşte o funcție de membru inline. Funcțiile membre care conțin câteva linii de cod sunt de obicei declarate inline.

```
class Y {
public:
char* f() { return 1; }
};
```

Este functie inline

```
class Y {
public:
char* f();
};
inline char* Y::f() { return 1; }
```

daca declaram functia in afara clasei atunci inline se pune la declaratie ei si nu la prototipul din clasa.

9. **Functii constructor cu parametrii.**

Constructorii sunt metode speciale care folosesc la crearea si initializarea instantelor unei clase. Constructorii au acelasi nume ca si clasa careia îi apartin si sunt apelati de fiecare data când se creaza noi instante ale clasei. Ca orice altă funcție, un constructor poate fi, supraîncărcat (funcții care au acelaşi nume dar un număr diferit de parametri). Compilatorul va apela cel a cărui parametri se potrivesc cu argumentele utilizate în apelul funcției.

```
class CRectangle {
int width, height;
public:
CRectangle ();
CRectangle (int,int);
int area (void) {return
(width*height);}
};
CRectangle::CRectangle () {
width = 5;
height = 5;
}
CRectangle::CRectangle (int a, int b) {
width = a;
height = b;
}i
nt main () {
CRectangle rect (3,4);
CRectangle rectb;
cout << "rect area: " << rect.area()
```

```
<< endl;                                    return 0;
cout << "rectb area: " << rectb.area()      }
<< endl;
```

10.**Functiile constructor cu un parametru: un caz special.**

Exista si modalitatea de a initializa membrii printr-o lista de instantiere (initializare), care apare în implementarea constructorului, între antetul si corpul acestuia. Lista contine operatorul :, urmat de numele fiecarui membru si valoarea de initializare, în ordinea în care membrii apar în definitia clasei.

```
class complex ;
complex::complex(double x, double y):real(x),imag(y).
```

11.Membrii de tip static ai claselor.

In C++, se pot defini date membre cu o comportare speciala, numite date statice. Acestea sunt alocate o singura data, existand sub forma unei singuri copii, comuna tuturor obiectelor de tipul clasa respectiv, iar crearea, initializarea si accesul la aceste date sunt independente de obiectele clasei. Sintaxa este: static DeclarareMembru.

O variabila membru static exista inainte de a fi creat orice obiect din acea clasa. De aceea el poate fi utilizatdirect in main(). Si deoarece exista inainte de crearea unui obiect, i se poate da oricand o valoare, dar nu in clasa incare este declarat, ci cand este definit.

12.Membrii statici de tip date.

O data membru al unei clase poate fi declarata static, indiferent de zona in care a fost inclusa (publica sau privata). O asemenea data este creata si initializata o singura data, in contrast cu datele ne-statice, care sint create pentru fiecare obiect al clasei. O data static este creata la inceputul programului, dar face totusi parte din clasa. Datele static declarate public sint ca variabilele normale:ele pot fi accesate de intregul cod folosindu-se numele lor, precedat de numele clasei si de operatorul scope.

13.**Functii membre statice.**

Diferențele între o funcție de membru static şi non-statice sunt după cum urmează.

a. functie membru static poate accesa doar datele statice membre, funcțiile membru statice, date şi funcții din afara clasei. O funcție membru non-statica poate accesa toate cele de mai sus, inclusiv membrii statici de date.

b. functie membru statice pot fi numite, chiar şi atunci când o clasă nu este instanțiata, o funcție non-statice nu poate

c. functie membru static nu pot fi declarate virtual, non-static poate

d. functie membru statica nu poate avea acces la "this" pointer al clasei.

Funcțiile membre statice nu sunt utilizate foarte frecvent în programare. Dar cu toate acestea, ele devin utile ori de câte ori avem nevoie de funcții care sunt accesibile chiar şi atunci când clasa nu este instantiata.

14.**Cînd sunt executati constructorii si destructorii.**

Constructorul este apelat în momentul declararii obiectelor.
Destructorul este apelat automat, la iesirea din blocul în care este recunoscut acel obiect.

15.Operatorul de specificare a domeniului.

:: (operatorul de specificare a domeniului) este folosit pentru a modifica variabilele ascunse.

```
int count = 0;
```

```
int main(void) {
int count = 0;
::count = 1; // set global count to 1
count = 2; // set local count to 2
return 0; }
```

Declarația count în main () ascunde variabila golbala int count. Cu :: count = 1 se acceseaza variabila declarata in domeniul global.

```
class X
{ public:
static int count; };
int X::count = 10; // define static data member
int main ()
{ int X = 0; // hides class type X
cout << X::count << endl; // use static member of class X }
```

În următorul exemplu, declararea variabilei int X ascunde clasa X, dar putem apela membrii
statici ai clsei prin operatorul ::

16.<u>Clase imbricate.</u>

Atunci, când o clasă se defineşte în cadrul unei alte clase, se spune că sunt clase imbricate.

O clasă imbricată este validă doar în interiorul clasei ce o conţine. Având în vedere posibilităţile oferite prin mecanismul de moştenire, astfel de clase se utilizează mai rar. O clasă imbricată este declarată în domeniul de vizibilitate al altei clase.Numele unei clase imbricate este vizibil locala in clasa de baza. Cu exceptia cazului utilizarii explicita a pointerelor, obiecte unei clase imbricate poate utiliza doar constructorii, membrii statici, enumerarile din clasa de baza ori cele declarate global.

Clasele imbricate sunt considerate ca entitati membre ale clasei unde au fost definite. Clasele interne pot accesa restul membrilor clasei externe, in corpul careia a fost definita, chiar daca acesti membri prezinta modificatorul de acces private. Clasele imbricate statice nu au acces la alti membri ai clasei externe. O clasa imbricata poate fi declarata cu orice modificator de acces. Clasele imbricate permit:

• inglobarea intr-un singur context logic a claselor care sunt utilizate intr-un singur loc
• cresterea incapsularii
• cresterea inteligibilitatii si a mentenabilitatii codului.

Accesul unei clase imbricate statice se face prin utilizarea operatorului “.” la nivelul clasei externe: SomeOuterClass.SomeStaticNestedClass.

17.<u>Clase locale.</u>

O clasă locala este declarat în definiția unei funcții. Declarațiile într-o clasă locală poate folosi doar nume de tip, enumerări, variabile statice, variabile si functii extere din domeniul vizibil.

18.<u>Transmiterea obiectelor catre functii.</u>

In C++ exista doua posibilitati de transmitere a parametrilor actuali catre o functie:

1. mecanismul de transmitere prin valoare
2. mecanismul de transmitere prin referinta

Primul este cunoscut, constituie modul standard de transmitere a parametrilor in C. 1. La apelul unei functii cu parametri de tip valoare, functiei apelate, i se transmite o copie a listei parametrilor efectivi. Ca atare, orice mofificare a acestora in corpul functiei apelate, nu afecteaza, la returnarea in functia apelanta, valoarile initiale ale acestora. Cu alte cuvinte, o functie apelata nu modifica valorile initiale ale parametrilor efectivi transmisi prin valoare; 2. daca tipul functie este **void** si daca instructiunea **return(*expresie*)** este sau nu prezenta, functia nu returneaza nici o valoare functiei apelante; 3. in cazul apelarii dependente a unei functii, in cadrul unei expresii, valoarea returnata de functie trebuie sa fie diferita de **void** si compatibila cu constructia expresiei.

Prin folosirea parametrilor formali referinta, se permite realizarea transferului prin referinta(transmiterea adresei), se elimina astfel necesitatea la utilizarea parametrilor formali pointeri, in cazul in care modificarile facute in interiorul functiei asupra parametrilor trebuie sa ramana si dupa revenirea din procedura.

Transferul prin referinta este util si atunci cand parametrul are dimensiune mare (struct, class) si crearea in stiva a unei copii a valorii lui reduce viteza de executie si incarca stiva.

- parametrii efectivi transmisi ca referinta unei functii apelate, trebuie declarati in functia apelanta ca variabile de tip referinta:
- **tip_arge1&** ***arge1***; **tip_arge2&** ***arge2***; … , **tip_argen&** ***argen***;

Daca se doreste pastrarea valorilor initiale ale parametrilor efectivi in functia apelanta, atunci se vor declara ca referinte alte variabile, carora li se vor atribui valorile parametrilor acestor variabile, si care vor fi transmise functiei apelate;

- la apelul unei functii cu parametri de tip referinta, functiei apelate, i se transmite chiar lista parametrilor efectivi. Ca atare, orice moficare a acestora in corpul functiei apelate, afecteaza, la returnarea in functia apelanta, valoarile initiale ale acestora. Cu alte cuvinte, o functie apelata prin referinte, modifica valorile initiale ale parametrilor efectivi transmisi prin referinte . Acesta proprietate, a apelului prin referinte, constituie un mijloc foarte important prin care o functie, in afara valorii returnate prin valoarea functiei furnizata de instructiunea **return(*expresie*)**, poate returna mai multe valori rezultat, prin intermediul parametrilor de tip **referinta**, ca si in cazul apelarii prin adrese, pointeri;
- daca tipul functie este **void** si daca instructiunea **return(*expresie*)** este sau nu prezenta, functia nu returneaza nici o valoare asociata numelui functiei catre functia apelanta, in acest caz functia putand returna oricate valori prin intermediul parametrilor de tip adresa;
- utilizarea apelarii functiilor prin referinta simplifica procesul de modificare a valorilor parametrilor referinta in functii, in sensul ca referirea variabilelor referinta nu mai necesita operatorul ->, ca in cazul apelarii functiilor cu parametri de tip pointer, adresa;
- in cazul apelarii dependente a unei functii, in cadrul unei expresii, valoarea returnata de functie trebuie sa fie diferita de **void** si compatibila cu constructia expresiei;

19.<u>Returnarea obiectelor.</u>

Cand un obiect este returnat de o functie, este creat un obiect temporar, care contine valoarea returnata.Acesta este de fapt obiectul returnat de functie. Daca obiectul care a fost returnat are un destructor care elibereaza memoria dinamica alocata, acea memorie va fi eliberata chiar daca obiectul care primeste valoarea returnata inca o mai foloseste. Exista cai de prevenire a acestei situatii care folosesc supraincarcarea operatorului de atribuire si definirea unui constructor de copii.

20.Atribtuirea obiectelor.

Este indicat ca implementarea operatorului de atribuire sa nu permita atribuirea catre acelasi obiect - de obicei acesta este modificat:

```
const X& X::operator= (const X& ob)
{ if ( &ob != this
{ // se asigneaza datele membru
} return *this;
} Sau constructor de copiere explicit de exemplu:
Person(int age)
{ this->age = age;}
```

Este indicat ca orice clasa sa defineasca propriul operator de atribuire, deci lucrurile sa nu fie lasate a fi tratate implicit de compilator - cu atat mai mult daca are campuri de tip pointer. Daca este vorba despre o clasa derivata, atribuirea trebuie sa realizeze operatia si pentru membrii clasei de baza.

21.**Matrice, pointeri si referinte.**

Pentru matrice trebuie obligatoriu constructori implicit (deoarece nu poate fi initializata).

Matricea bidiminsionala este pointer catre pointer.

Pointerii sunt variabile care contin adresa unei alte zone de memorie. Ei sunt utilizati pentru a date care sunt cunoscute prin adresa zonei de momorie unde sunt alocate.
Sintaxa utilizata pentru declararea lor este:

tip *variabila_pointer;

In lucrul cu pointeri se folosesc doi operatori unari:

- &: extragerea adresei unei variabile
- *: referirea continutului zonei de memorie indicate de pointer (indirectare)

Referintele, ca si pointerii, sunt variabile care contin adresa unei zone de memorie. Semantic, ele reprezinta aliasuri ale unor variabile existente.Referintele sunt legate de variabile la declaratie si nu pot fi modificate pentru a referi alte zone de memorie. Sintaxa folosita pentru declararea unei referinte este:

Tip & referinta = valoare;

Proprietatile cele mai importante ale referintelor sunt:

- referintele trebuie sa fie initializate la declaratie (spre deosebire de pointeri care pot fi initializati in orice moment);
- dupa initializare, referinta nu poate fi modificata pentru a referi o alta zona de memorie (pointerii pot fi modificati pentru a referi alta zona)
- intr-un program C++ valid nu exista referinte nule

22.Matrice de obiecte.

Pentru a declara matrici de obiecte care sa poata fi initializate trebuie definit un constructor cu parametri care sa faca posibila initializarea. Pentru a declara matrici de obiecte care sa nu fie initializate trebuie definit un constructor fara parametri. Pentru a declara matrici de obiecte care sa fie cand initializate cand neinitializate se supraincarca functia constructor.

```
#include<iostream>
using namespace std;
class C //apelare ptr. matrice neinitializate
C(int j) //apelare ptr. matrici initializate
```

```
int da()
};
void main()
C ob2[34]; //neinitializat }
```

23.**Matrice initializate/ matrice neinitializate.**

Pentru a declara matrici de obiecte care sa poata fi initializate trebuie definit un constructor cu parametri care sa faca posibila initializarea.Pentru a declara matrici de obiecte care sa nu fie initializate trebuie definit un constructor fara parametri.Pentru a declara matrici de obiecte care sa fie cand initializate cand neinitializate se supraincarca functia constructor. Matricea poate fi inițializata la definire prin precizarea constantelor de inițializare.

```
int b[2][3] = {1,2,3,4,5,6}; // echivalent cu
int b[2][3] = {{ 1,2,3},{ 4,5,6}}; // echivalent cu
int b[][ 3] = {{ 1,2,3},{ 4,5,6}}
double a[3][2]={{2},{5.9,1},{-9}}; //elementele pe linii sunt: 2 0 / 5.9 1 / -9 0
double a[3][2]={2,5.9,1,-9}; //elementele pe linii sunt: 2 5.9 / 1 -9 / 0 0
```

24.**Pointeri către obiecte.?**

Indicatori către membri vă permite să se referă la membri nonstatic de clasa de obiecte. Se poate utiliza un pointer la membru la punctul de a unui membru de clasă statice pentru adresa de membru statice nu este asociat cu orice obiect special. Pentru a indica un membru de clasă statice, trebuie să utilizați un indicatorul normal. Utilizați indicii pentru funcții de membru în acela□i mod ca □i indicii pentru funcții. Puteți compara indicii de funcții membru, asignați valori le şi utilizați-le pentru a apela membru funcții. Rețineți că o funcție membre nu are acela□i tip ca o funcție de țările care are acelaşi număr şi tip de argumente şi acelaşi reveni tip.

25.Pointerul this.

Cand este apelata o functie membru, i se paseaza un argument implicit, care este un pointer catre obiectul care a generat apelarea (obiectul care a invocat functia). Acest pointer este numit this.

La membrii unei class se poate capata acces direct dintr-o functie membru. Instructiunea b = j; ar comanda ca valoarea continuta in baza sa fie atribuita unei copii a lui b asociata obiectului care a generat apelarea. Totusi, aceeasi intructiune poate fi scrisa si astfel: this->b = j;

26.Pointeri către tipuri derivate.

In general un pointer de un anume tip nu poate indica un obiect de alt tip. Totuşi, este posibil ca un pointer de tipul unei clase de bază să pointeze (refere) către un obiect de tipul unei clase derivate. Să presupunem că avem o clasă B şi o altă clasă D, care derivă din B. Atunci este posibil ca un pointer de tip *B să indice un obiect de tip D. Reciproca nu este adevărată! Un pointer de tip *D nu poate referi un obiect de tip B. Deşi puteți referi un obiect de tip derivat cu un pointer de tipul bazei, veți putea accesa numai membrii moşteniți din tipul de bază. Nu veți putea accesa membrii particulari clasei derivate. Puteți totuşi să folosiți un cast şi să convertiți un pointer de tip bază către unul derivat, pentru a avea acces total la clasa derivată.

27.**Referinte.**

Referintele, ca si pointerii, sunt variabile care contin adresa unei zone de memorie. Semantic, ele reprezinta aliasuri ale unor variabile existente.

Referintele sunt legate de variabile la declaratie si nu pot fi modificate pentru a referi alte zone de memorie. Sintaxa folosita pentru declararea unei referinte este:
Tip & referinta = valoare;

28.**Parametri de referință.**

Functiile C + + transferă în mod normal parametrii prin valoare. Ele primesc valorile efective ale parametrilor sub forma de copii ale datelor originale. Pot modifica în interiorul lor valorile parametrilor prin pointerifără a afecta datele originale. Modificarea valorilor parametrilor se poateface mai simplu prin transferul parametrilor nu prin valoare, ci prinreferință, eliminând instrucțiunile care combină variabilele pointer cucele normale.O referința crează un nume alternativ (alias) pentru o variabilă. Sedeclară prin sintaxă:
tip& nume_alias = variabila;

unde: & (ampersand) se pune imediat dupa tip.

variabila = este cea pentru care referința este un alias

După declararea unei referințe în program putem folosi atat variabila cât şi referința. Referința nu este o variabilă. Ea nu mai poate fi modificată după ce a fostasociată unei variabile. Folosirea unei referințe ca identificator este utilăcând aceasta apare ca parametru formal al unei funcții. Spre deosebire de pointeri, referințele nu permit operațiile:

• atribuirea unui pointer o referință;
• obținerea adresei unei referințe cu operatorul adresa ;
• compararea valorilor referințelor prin operatorii relaționali;
• operații aritmetice cum ar fi adunarea unui deplasament(a uneiadrese)

Transferul parametrilor unei funcții prin referință este făcut decompilator, ceea ce simplifică scrierea funcției şi apelul ei.Transferul parametrilor de tip structură prin referință formale ca şi cel prin pointeri este mai eficient decat transferul prin valoare, deoareceelimină copierea structurii pe stivă, conducand astfel la creşterea vitezeide execuție. Rezultatul unei funcții poate fi transferat prin valoare, pointer sau referință

29.**Transmiterea referintelor către obiecte.**

Cand se face apel prin referinta, nu se face nici o copie a obiectului, asa cum se intampla cu apelul prin valoare. Aceasta inseamna ca nici un obiect folosit ca parametru nu este distrus atunci cand se termina functia, iar destructorul parametrului nu este apelat.
NOTĂ: cand parametrii sunt transmisi prin referinta, schimbarile obiectului din interiorul functiei afecteaza obiectul apelant.

30.**Returnarea referintelor.**

O functie poate sa returneze o referinta ceea ce face ca ea sa poata fi folosita in membrul stang al unei instructiuni de atribuire.

31.**Referinte independente.**

O referinta care este doar o simpla variabila este numita referinta independenta.

32.**Restrictii pentru referinte.**

Referinta independenta este de mica valoare practica deoarece ea este de fapt doar un alt nume aceeasi variabila. Avand doua nume care descriu acelasi obiect programul poate deveni confuz.

33.Operatorii de alocare dinamică din C++.

C++ defineşte doi operatori de alocare dinamică: new şi delete. Aceşti operatori alocă

şi eliberează memoria dinamic, in timpul execuției programului. Operatorul new alocă memorie şi returnează un pointer către adresa de inceput a zonei alocate. Operatorul delete eliberează zona de memorie alocată cu new.

În limbajul C++ alocarea dinamica a memoriei si eliberarea ei se pot realiza cu operatorii new si delete. Folosirea acestor operatori reprezinta o metoda superioara, adaptata programarii orientate obiect. Operatorul new este un operator unar care returneaza un pointer la zona de memorie alocata dinamic. În situatia în care nu exista suficienta memorie si alocarea nu reuseste, operatorul new returneaza pointerul NULL. Operatorul delete elibereaza zona de memorie spre care pointeaza argumentul sau.

Sintaxa:

```
tipdata_pointer = new tipdata;
tipdata_pointer = new tipdata(val_initializare);
//pentru initializarea datei pentru care se aloca memorie dinamica
tipdata_pointer = new tipdata[nr_elem]; //alocarea memoriei pentru un tablou
delete tipdata_pointer;
delete [nr_elem] tipdata_pointer;  //eliberarea memoriei pentru tablouri
```

Tipdata reprezinta tipul datei (predefinit sau obiect) pentru care se aloca dinamic memorie, iar tipdata_pointer este o variabila pointer catre tipul tipdata.

Pentru a putea afla memoria RAM disponibila la un moment dat, se poate utiliza functia coreleft: unsigned coreleft(void);

34.<u>Alocarea de memorie obiectelor.</u>

Alocarea statică a memoriei: adresele □i dimensiunile obiectelor ce fac uz de alocarea statică a memoriei sunt fixate in momentul compilării □i pot fi plasate intr-o zonă de dimensiune fixă ce corespunde unei secțiuni din cadrul fi□ierului linkedidat final. Acest tip de alocare a memoriei se nume□te statică deoarece locația □i dimensiunea lor nu variază pe durata de execuție a programului.

Alocarea automată a memoriei: obiectele temporare (variabilele locale declarate in cadrul unui bloc de cod) sunt stocate in cadrul de stivă asociat funcției apelate, iar spațiul alocat este automat eiberat □i reutilizat după ce s-a părăsit blocul in care acestea au fost declarate.

Alocarea dinamică a memoriei: blocuri de memorie de orice dimensiune pot fi alocate intr-o zonă de memorie numită heap prin intermediul funcțiilor malloc(), calloc() □i realloc().

Alocarea de memorie se face in C++ folosind operatorul new si operatorul pereche delete.

35.**<u>Supraîncărcarea functiilor si a operatorilor.</u>**

Supraincărcarea (overloading) funcțiilor şi operatorilor reflectă posibilitatea de a atribui unui simbol mai multe semnificații.

Supraincărcarea unei funcții inseamnă utilizarea aceluiaşi identificator de funcție pentru cel puțin două funcții cu condiția ca să difere intre ele prin tipuri şi/sau număr de parametri (cu prototipuri diferite). In acest fel compilatorul poate selecta, la un moment dat, funcția care trebuie apelată.

Operatorii sunt notații concise, infixate, pentru operații matematice uzuale. Limbajul C++, ca orice limbaj de programare asigură un set de operatori pentru tipurile primitive. În plus, față de limbajul C, C++ oferă posibilitatea asocierii operatorilor existenți cu tipurile definite de utilizator . Astfel, prezintă interes extinderea operatorilor în aritmetică complexă, algebra matricială, în lucrul cu şiruri de caractere, etc. Un operator poate fi privit ca o funcție, în care

termenii sunt argumentele funcției (în lipsa operatorului +, expresia a+b s-ar calcula apelând funcția aduna(a,b)).

Limbajul C++ introduce următorii operatori noi: new şi delete- pentru gestiunea memoriei dinamice, operatorul de rezoluție (::) şi operatorii de acces la membri: .* şi ->*. Functiile operator constituie un tip special de functii , care s-ar putea utiliza pentru redefinirea operatorilor de baza care apar on C. Un tip de clasa se poate defini ompreuna cu un set de operatori asociati, obtinuti prin supraoncarcarea operatorilor existenti. In acest fel, se efectueaza operatii specifice cu noul tip la fel de simplu ca on cazul tipurilor standard. Procedeul consta on definirea unei functii cu numele operator < simbol >

36.**Supraîncărcări de functii si ambiguități.**

Conversia automată a tipului, in C++, poate conduce la apariția unor ambiguități in supraincărcarea funcțiilor. Situatia in care la un apel compilatorul nu poate alege intre doua sau mai multe functii supraincarcate se numeste ambiguitate. Instructiunile ambigue sunt tratate ca erori, iar programul nu va fi compilat.

37.**Supraîncărcarea functiilor constructor.**

Dupa cum s-a invatat, functia constructor este o metoda speciala a claselor care se executa automat cand se creeaza diverse instante ale unui obiect. Supraincarcarea functiilor presupune declararea si definirea unor functii cu acelasi nume, astfel incat, in functie de parametrii transmisi, la compilare, sa se poata decida care dintre functiile cu acelasi nume sa fie adresata Ca orice functie in C++ si functiile constructor pot fi supraincarcate si adresate. Daca o functie constructor este supraincarcata, atunci, la creearea unei instante a unui obiect, se va executa totdeauna prima functie constructor, cea de-a doua executandu-se cand prima nu poate fi executata din diferite motive, ca de exemplu, transmiterea eronata a parametrilor.

```
public:
wtf() { a = 0; b = 0; c = 0; } // Constructor Implicit
wtf(int A) { a = A; b = 0; c = 0; }
wtf(int A, int B) { a = A; b = B; c = 0; }
wtf(int A, int B, int C) { a = A; b = B; c = C; }
```

Supraincărcand constructorii, clasa voastră devine mult mai flexibilă (obiectele ei pot fi inițializate in mai multe feluri). Utilizatorul clasei voastre va fi liber să aleagă din mai multe moduri de inițializare, in funcție de necesități şi circumstanțe.

38.**Crearea unei functii operator membru.**

In limbajul C++, exista posibilitatea supraincarcarii functiilor membre, prin folosirea acelorasi identificatori pentru mai multe functii, care urmeaza sa fie apelate si executate, la un anumit moment de timp. Inlaturarea ambiguitatilor legate de apelarea functiilor supraincarcate se poate face prin diferentierea parametrilor efectivi cum ar fi: numar diferit de parametri, tipuri diferite de parametri, etc..

Funcțiile operator membru au sintaxa de implementare:

```
<Tip_returnat> <Nume_clasă>:: operator # (<Lista_argumente>)
 { //Operații specifice }
```

Deseori funcțiile operator returnează un obiect din clasa asupra căreia operează, dar <Tip_returnat> poate fi orice tip valid.

este o notație pentru numele operatorului așa cum va fi folosit in program după redefinire. Deci, dacă supraîncărcăm operatorul = atunci sintaxa prototipului funcțieimembru va fi: <Tip_returnat> operator =(<Lista_argumente>);

Funcțiile operator membre au un singur parametru sau nici unul. În cazul în care au unparametru, acesta se referă la operandul din dreapta al operatorului. Celălalt operand ajunge la operator prin intermediul pointerului special this .Astfel stând lucrurile, trebuie să avem grijă de eventualitatea ca operandul din stânga să fie o constantă, ceea ce înseamnă ca nu mai avem context de apel pentru operatorulredefinit.

39.**Crearea operatorilor de incrementare si de decrementare cu prefix si cu sufix.**

Operatorii de incrementare şi decrementare pot fi folosiți atat ca prefix cat şi sufix. Cand sunt folosiți ca prefix, operație de incrementare sau decrementare se realizează inainte de evaluarea expresiei, iar atunci cand sunt folosiți ca sufix, operația de incrementare sau decrementare se realizează după evaluarea expresiei.

40.**Supraîncărcarea operatorilor prescurtati.**

Operatori prescurtați sunt +=, -= si restul de operatori care urmeaza acest Șablon (pattern). Cand se supraincarcă unul din aceşti operatori se combina o operatie cu o atribuire.

41.**Restrictii la supraîncărcarea operatorilor.**

Trebuie să precizăm faptul că precedența, aritatea (numărul de operanizi) şi asociativitatea operatorilor nu poate fi schimbată prin supraincărcare. Nu este posibil să creăm noi operatori, doar cei existenți putand fi supraincărcați. Operațiile realizate de operatorii tipurilor de date predefinite nu pot fi modificate. Programatorul nu poate, de exemplu, să schimbe modalitatea in care se adună doi intregi. Supraincărcarea operatorilor este valabilă doar pentru tipuri de date definite de programator sau pentru operații care combină tipuri de date definite de programator cu tipuri de date predefinite.

Supraincarcarea operatorilor este supusa urmatoarelor restrictii:

- Se pot supraincarca doar operatorii existenti; nu se pot crea noi operatori.
- Nu se poate modifica aritatea (numarul de operanzi) operatorilor limbajului (operatorii unari nu pot fi supraincarcati ca operatori binari, si invers).
- Nu se poate modifica precedenta si asociativitatea operatorilor.
- Nu pot fi supraincarcati operatorii . ::? si :

42.**Supraîncărcarea operatorilor folosind o functie friend.**

Se poate supraîncărca un operator relativ la o clasă folosind o funcție friend. Deoarece unprieten nu este membru al clasei, nu are disponibil un pointer de tip this.De aceea, unei funcții supraîncărcate de tip friend operator I se vor transmiteexplicit operanzii. Deci, dacă supraîncărcăm un operator unar vom avea un parametru,dacă supraîncărcăm unul binar vom avea doi parametri.Dacă supraîncărcăm un operator binar, operandul din stânga este pasat în primulparametruiar cel din stânga în al doilea parametru.

Puteți supraincărca un operator pentru o clasă folosind o funcție nonmembră, care de obicei este prietenă cu clasa. Deoarece o funcție friend nu este membră a clasei, aceasta nu are pointerul this. Aşadar, o funcție operator prietenă primeşte explicit operanzii. Asta inseamnă că o funcție prietenă ce supraincarcă un operator binar va avea doi (2) parametri, şi o funcție prietenă ce supraincarcă un operator unar, va avea un (1) parametru. In cazul supraincărcării

unui operator binar cu o funcție prietenă, operandul din stanga este tranmis primului parametru, iar cel din dreapta este transmis celui de-al doilea parametru.

43.**Folosirea unui friend pentru a supraîncărca ++ si --.**

Operatorii ++ si -- sunt operatori unari, iar supraincarcarea acestora se poate face utilizand atat functii membru non-statice, cat si functii friend. Pentru a putea distinge intre forma prefix si cea postfix a acestor operatori se aplica urmatoarea regula: O functie-membru operator++ care nu primeste nici un parametru (cu exceptia parametrului implicit this) defineste operatorul ++ postfix, in timp ce functia operator++ cu un parametru de tip int defineste operatorul ++ postfix. La apel, in cazul formei postfix, utilizatorul nu este obligat sa specifice nici un argument, valoarea transmisa implicit fiind 0. Aceeasi regula se aplica si pentru operatorul de decrementare. Astfel:

```
class X {  public:
X operator++() { ... }
X operator++(int) { ... }  };
void f(X a) {
++a; // la compilare se traduce ca a.operator++();
a++; // la compilare se traduce ca a.operator++(0); }
```

44.**Functiile friend operator adaugă flexibilitatea.**

45.**Supraîncărcarea operatorilor new si delete.**

-chiar si supraincarcat operatorul new va conlucra cu constructorul clasei in alocarea si eventual initializarea unui obiect dinamic .La supradefinire functia operator va primi dimensiunea zonei de alocat si va returna adresa memoriei alocate .

-obligatoriu functia new supraincarcata trebuie sa returneze un pointer spre memoria alocata sau zero(NULL) daca apare o eroare de alocare

-Cand new si delete sunt functii membre ale unei clase operatorii astfel supraincarcati vor fi folositi numai pentru obictele clasei , iar cand se utilizeaza operatorii new si delete pentru alte tipuri de date se va apela new si delete impliciti din C++;

Operatorii new şi delete pot fi supraîncărcați ca funcții operatori membre statice. Funcția operator new are semnătura: void* operator new(size_t); Operatorul redefinit apelează constructorul clasei după alocarea dinamică pe care o realizează. Funcția operator delete are semnătura: void operator delete(void*); unde parametrul este un pointer la obiectul eliminat. Operatorul redefinit delete apelează întotdeauna destructorul clasei înainte de a elibera memoria alocată dinamic. Dacă în clasa C se supraîncarcă operatorul new, atunci versiunea furnizată de sistem se obține prin C *p = ::new C();

46.**Supraîncărcarea operatorilor new si delete pentru matrce.**

Supraincarcarea operatorilor new si delete pentru matrice au una din formele :

```
ex.  //se aloca memorie unei matrice de obiecte
    //este apelat automat functia constructor a fiecarui obiect al matricei
void *operator new [ ](size_t dim) void *operator new [ ](unsigned dim)
{ { //efectueaza alocarea //efectueaza alocarea
return pointer_la _memorie ; return pointer_la _memorie ;  } }
```

//delete pentru o matrice de obiecte
// operatorul delete apeleaza repetat destructorul clasei pentru fiecare membru al masivului ;
-nu este permisa mixarea celor doua mecanisme de alocare si eliberare de memorie (cu functii malloc( ) si free( ),respectiv cu operatorii new si delete)adica alocare cu malloc( ) si dezolocare cu delete( ) sau alocare cu new si dezalocare cu free( ) .

47.Supraîncărcarea unor operatori speciali.

48.**Supraîncărcarea pentr [].**

Operatorul predefinit se utilizeaza pentru a face acces la elementele unui tablou, in constructii de forma tablou[expr]. Aceasta constructie poate fi privita ca o expresie formata din operanzii tablou si expr, carora li se aplica operatorul [ ]. Putem supraincarca acest operator pentru a da sens constructiilor de indexare si pentru cazul in care operanzii sunt obiecte. Operatorul [ ] este numit subscript sau indice şi este util în accesarea elementelor unui tablou (variabile cu indici).

Operatorul [ ] este un operator binar. În "teta[8]",teta reprezintă primul operand, iar "8" al doilea operand. Se are în vedere că "[ ]" este un operator de prioritate maximă. Numele tabloului este interpretat ca „primul operand“, deoarece el se poate utiliza într-un mod similar cu identificatorul unei variabile simple, adică permite accesul la o dată de un tip predefinit. Primul operand poate fi chiar o expresie cu pointeri a cărei valoare reprezintă un pointer.

Funcția operator [ ] este o funcție membră în care primul operand trebuie să fie un obiect de tipul clasei din care provine, iar al doilea operand este singurul parametru explicit al funcției de un tip oarecare. Având în vedere scopul acestui operator, de a asigura înscrierea indecşilor într-un tablou, adesea este utilizat tipul int pentru acest al doilea operand.

Nu se poate folosi funcție friend pentru supraîncărcarea operatorului [ ].

49.**Supraîncărcarea pentru ().**

Operatorul ( ) este cunoscut sub denumirea deoperator de apel de funcție. O funcție se apelează prin: id_funcție (lista_param_efectivi)

O astfel de construcție poate fi privită ca un operand. În realitate ea este formată din doi operanzi: id_funcție şilista_param_efectivi cărora li se aplică operatorul binar ( ).Cel de-al doilea parametru poate să fie şi vid. Această situație este caracteristică acestui operator, spre deosebire de ceilalți operatori binari, care nu admit operand vid.
Apelul unei funcții se poate realiza şi folosind o construcție de forma:

(*exp)(lista_param_efectivi)

în care id_funcție este înlocuit cu o expresie tip pointer spre funcția respectivă.

Un astfel de apel este util atunci când nu se cunoaşte numele funcției ci doar un pointer la ea. Supraîncărcarea operatorului ( ), nu creează o nouă cale de apel. De fapt, se creează o funcție operator, căreia i se poate transmite un număr arbitrar de parametri. Când în program se foloseşte operatorul ( ), parametrii efectivi sunt copiați în parametrii funcției de supraîncărcare a operatorului ( ). Obiectul, care generează apelul, este indicat de pointerul this.

Funcția, care supraîncarcă operatorul ( ), trebuie să fie o funcție membră nestatică. Nu poate fi funcție friend. Supraîncărcarea operatorului () se foloseşte şi la definirea unui iterator. Iteratorii se utilizează la tipuri abstracte care conțin colecții de elemente ca: liste, arbori etc. Iteratorii sunt utili în consultarea elementelor dintr-o astfel de structură de elemente protejate. O colecție de elemente se poate implementa în mai multe feluri:

a. tablou de obiecte de tip obiect cu dimensiune fixă sau variabilă;
b. listă de noduri de tip obiect simplu sau dublă înlănțuită;
c. arbore binar cu noduri de tip obiect etc.

La supraîncărcarea operatorului ( ), se pot folosi parametri de orice tip, şi se poate returna o valoare de orice tip în funcție de scopul propus.

50.**Supraîncărcarea pentru >.**

Supraîncărcarea operatorului -> se realizează printr-o funcție membră nestatică. La supraîncărcare, operatorul ->este considerat ca operator unar care se aplică operandului care îl precede. În construcția: obiect->element;
obiect este o instanțiere a clasei respective, cel care generează apelul. În ceea ce priveşte element, el trebuie să fie un element accesibil în cadrul obiectului returnat de funcția operator->( ).

51.**Supraîncărcarea pentru +.**

In limbajul C++, operatorii + şi – au diverse funcții dependente de context: operații aritmetice pentru valori intregi, reale, operații cu pointeri. Acesta este un exemplu de supraincărcare a operatorilor. Limbajul C++ permite programatorilor să supraincarce majoritatea operatorilor pentru ca aceştia să poată fi folosiți in contextul unor noi clase. Unii operatori sunt supraincărcați mai frecvent decat alții, cum ar fi de exemplu operatorul de asignare sau cei aritmetici de adunare sau scădere. Acțiunile implementate de operatorii supraincărcați pot fi realizate la fel de bine şi prin apeluri explicite de funcții, insă folosirea notației cu operatori este mai clară şi mai intuitivă. Operația de adunare + funcționează pentru variabile de tip int, float, double şi un număr de alte tipuri de dată predefinite deoarece operatorul + a fost supraincărcat chiar in limbajul de programare C++.

52.Supraîncărcarea pentru -.

```
class Complex
{  int re, im;
public:
Complex(int re=0, int im=0);
~Complex(void);
void afisare();   };
Complex::Complex(int re, int im)
{  this->re=re;
this->im=im; }
```

```
Complex::~Complex(void)
{ }
void Complex::afisare()
{  cout << re << "+" << im << "i" << "\n";  }
```

Adaugam la programul anterior prototipul functiei

```
Complex operator- (Complex);
```

Definirea functiei

```
Complex Complex::operator- (Complex c)
{
Complex tmp;
tmp.im = this->im - c.im;
tmp.re = this->re - c.re;
return tmp;  }
```

Daca avem doua variabile de tip Complex x si y apelul x-y se va transforma in x.operator-(y)

Restrictie: in acest caz tipul primului operand este mereu tipul clasei.

Proprietatile operatorilor care nu pot fi modificate:

· pluraritatea

· precedenta si asociativitatea

53.**Supraîncărcarea pentru operatorii: # (atribuire)-=.**

(atribuire) - In limbajul C++ se pot face atribuiri de obiecte care sunt instantieri ale aceleiasi clase. Daca programatorul nu defineste in mod explicit un operator de atribuire pentru o clasa, compilatorul genereaza unul implicit. Comportarea operatorului de atribuire implicit presupune ca datele membru ale obiectului din dreapta operatorului = se atribuie la datele membru corespunzatoare ale obiectului din partea stanga a operatorului, dupa care se returneaza o referinta la obiectul modificat. Aceasta copiere este de tip "membru la membru" asa cum se intampla si in cazul constructorului de copiere generat de compilator. Putem descrie deci functionarea operatorului de atribuire generat de compilator.

54.Supraîncărcarea pentru .(punct).

Urmatorii operatori nu pot fi supraincarcati: . :: .* ?: sizeof

55.Supraîncărcarea pentru sizeof.

Urmatorii operatori nu pot fi supraincarcati: . :: .* ?: sizeof

56.Supraîncărcarea pentru operatorul virgulă.

Virgula este un operator binar. Operatorul virgulă supraîncărcat, pentru a acționa într-un mod similar cu acțiunea sa normală, trebuie să renunțe la valoarea din termenul său stâng şi să atribuie valoarea operației, termenului din dreapta. Astfel că într-o listă, în care virgula este un separator, se va renunța la toți termenii prezenți, mai puțin ultimul termen din dreapta.

Permite evaluarea unei liste de obiecte şi returnează referința ultimului obiect din listă. Este recomanat să se lucreze cu pointeri constanți de conținut constant.

```
const persoana &operator,(const persoana &p) const
{ return p;}
```

57.**Mostenirea.**

Moştenirea permite crearea unei ierarhii de clase, pornind de la cea mai generală la cea mai concretă. Procesul implică definirea unei clase de bază care defineşte toate calitățile commune ale obiectelor ce vor deriva din bază. Clasele derivate din bază se numesc clase derivate. O clasă derivată include toate caracteristicile clasei de bază plus calități specifice ei (ale clasei derivate). Forma generală a moştenirii este:

class nume-clasa-derivata : acces nume-clasa-baza { // corpul clasei };

Specificatorul de acces, acces, trebuie să fie unul din următoarele cuvinte-cheie: public, private sau protected.

Acest specificator determină nivelul de acces al membrilor clasei de bază in interioul clasei derivate.

Dacă specificatorul lipseşte şi clasa este declarată cu class, atunci implicit va fi asumat private. Dacă clasa este declarată cu struct şi specificatorul lipseşte, atunci va fi asumat public. Cand acces este public, toți membrii publici din bază devin membri publici in clasa derivată, şi toți membrii protected din clasa de bază devin membri protejați in clasa derivată. Elementele private ale clasei de bază răman private şi nu sunt accesibile in clasa derivată.

58.Controlul accesului la clasa de bază.

Constructorul clasei de baza se va executa inaintea constructorului clasei derivate, iar destructorul clasei derivate se va executa inaintea destructorului clasei de baza. Accesul la membrii clasei de bază moşteniți in clasa derivată este controlat de specificatorul de acces (public, protected, private) din declarația clasei derivate.O regulă generală este că , indiferent de specificatorul de acces declarat la derivare, datele de tip private in clasa de bază nu pot fi accesate dintr-o clasă derivată. O altă Regulă generala este ca prin derivare, nu se modifică tipul datelor in clasa de bază. Un membru protected intr-o clasă se comporta ca un membru private, adică poate fi accesat numai de membrii acelei clase şi de funcțiile de tip friend ale clasei. Diferența intre tipul private şi tipul protected apare in mecanismul de derivare: un membru protected al unei clase moştenita ca public intr-o clasă derivată devine tot protected in clasa derivată , adică poate fi accesat numai de funcțiile membre şi friend ale clasei derivate şi poate fi transmis mai departe, la o nouă derivare, ca tip protected.

59.**Mostenirea si membrii protejati.**

Cand un membru al clasei este declarat protected, acel membru nu este accesibil părților de program nonmembre, dar este accesibil claselor derivate. Cand zic că este accesibil claselor derivate mă refer la faptul că poate fi accesat direct de membrii clasei derivate. In primul exemplu, clasa base are doi membri privați: a şi b, care nu pot fi accesați in clasa derivat. De exemplu, următoarea definiție a clasei derivat ar fi cauzat o eroare de compilare:

60.**Mostenirea protected a clasei de bază.**

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

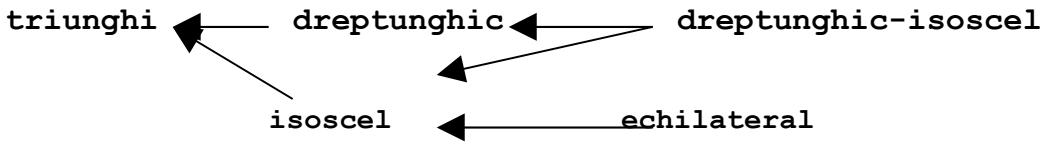
Nivelul de acces protected este un nivel intermediar între accesul public şi cel private. Membrii protected ai unei clase de bază pot fi accesați doar de membrii şi de prietenii clasei de bază şi de membrii şi prietenii claselor derivate. Membrii claselor derivate pot referi membrii public şi protected ai clasei de bază folosind numele acestor membri. Datele protected depăşesc ideea de încapsulare pentru că o schimbare a membrilor protected din clasa de bază poate influența toate clasele derivate. În general, se recomandă ca datele membre să fie declarate private, iar protected trebuie folosit numai atunci când este strict necesar.
Cand un membru al clasei este declarat protected, acel membru nu este accesibil părților de program nonmembre, dar este accesibil claselor derivate. Cand zic că este accesibil claselor derivate mă refer la faptul că poate fi accesat direct de membrii clasei derivate. In primul exemplu, clasa base are doi membri privați: a şi b, care nu pot fi accesați in clasa derivat.

Dacă specificatorul de acces din declarația clasei derivate este protected, atunci toți membrii de tip public şi protected din clasa de bază devin membri protected în clasa derivată. Bineînțeles,membrii de tip private în clasa de bază nu pot fi accesați din clasa derivată. Se reiau clasele din exemplul precedent cu moştenire protected:

class Derived : protected Base {// acelasi corp al clasei};

61.**Mostenirea din clasa de bază multiple.**

În cazul mostenirii multiple este posibila mostenirea din mai multe clase de baza:
class derivata : acces1 baza1, ..., accesn bazan { // corp clasa; };
O baza directa este mentionata în lista claselor de baza ale clasei derivate.
Prin mostenire multipla si indirecta se creaza ierarhii de clase, care sunt grafuri orientate aciclice (în cazul mostenirii simple avem un arbore orientat).
Exemplu de moştenire multiplă:



In C++ este perfect posibil ca o clasa sa mostenească campuri (variabile) si metode din mai multe clase, aceasta realizandu-se prin separarea cu virgula a diferitelor clase de baza in declaratia clasei derivate. De exemplu, fie o clasa care tipareste la consola (COutput) si se doreste ca noile clase derivate CRectangle si CTriangle sa mostenească deasemenea membrii in plus fata de cei ai clasei de baza CPolygon, se va scrie:
class CRectangle: public CPolygon, public COutput { ... }
class CTriangle: public CPolygon, public COutput { ... }

62.**Constructori, destructori si mostenire.**

Constructorul este o metodă specială a unei clase, care este membru al clasei respective şi are acelaşi nume ca şi clasa. Constructorii sunt apelați atunci când se instanțiază obiecte din clasa respectivă, ei asigurând inițializarea corectă a tuturor variabilelor membru ale unui obiect şi garantând că inițializarea unui obiect se efectuează o singură dată.

Constructorii se declară, definesc şi utilizează ca orice metodă uzuală, având următoarele proprietăți distinctive:
- poartă numele clasei căreia îi aparțin;
- nu pot returna valori; în plus (prin convenție), nici la definirea, nici la declararea lor nu poate fi specificat “void” ca tip returnat;
- adresa constructorilor nu este accesibilă utilizatorului; expresii de genul “&X :: X()” nu sunt disponibile;
- sunt apelați implicit ori de câte ori se instanțiază un obiect din clasa respectivă;
- în caz că o clasa nu are nici un constructor declarat de către programator, compilatorul va declara implicit unul. Acesta va fi public, fără nici un parametru, şi va avea o listă vidă de instrucțiuni;
- în cadrul constructorilor se pot utiliza operatorii "new" si "delete",
- constructorii pot avea parametrii.

Destructorul este complementar constructorului. Este o metodă care are acelaşi nume ca şi clasa căreia îi aparține, dar este precedat de “~”. Dacă constructorii sunt folosiți în special pentru a aloca memorie şi pentru a efectua anumite operații (de exemplu: incrementarea unui contor al numărului de obiecte), destructorii se utilizează pentru eliberarea memoriei alocate de constructori şi pentru efectuarea unor operații inverse (de exemplu: decrementarea contorului).
Destructorii au următoarele caracteristici speciale: - sunt apelați implicit în două situații:
1. când se realizează eliberarea memoriei alocate dinamic pentru memorarea unor obiecte, folosind operatorul “delete”;
2. la părăsirea domeniului de existență al unei variabile. Dacă în al doilea caz este vorba de variabile globale sau definite în “main”, distrugerea lor se face după ultima instrucțiune din “main”, dar înainte de încheierea execuției programului.

Moştenirea permite crearea unei ierarhii de clase, pornind de la cea mai generală la cea mai concretă. Procesul implică definirea unei clase de bază care defineşte toate calitățile comune ale obiectelor ce vor deriva din bază. Clasele derivate din bază se numesc clase derivate. O clasă derivată include toate caracteristicile clasei de bază plus calități specifice ei (ale clasei derivate). Forma generală a moştenirii este:

class nume-clasa-derivata : acces nume-clasa-baza { // corpul clasei };

Specificatorul de acces, acces, trebuie să fie unul din următoarele cuvinte-cheie: public, private sau protected.

63.Cind sunt executate functiile constructor si destructor.

Aşa cum vedeți, cand un obiect al clasei derivate este creat, constructorul clasei de bază este invocat primul, urmat de constructorul clasei derivate. Cand obiectul este distrus, destructorul clasei derivate este apelat primul, urmat de destructorul clasei de bază. Altfel spus, constructorii sunt executați in ordinea derivării (de la clasa din varful ierarhiei la clasa din baza ierarhiei), iar destructorii sunt executați in ordinea inversă derivării (de la baza ierarhiei la varf). Aceeaşi regulă se aplică şi moştenirii multiple.

64.Transmiterea parametrilor spre constructorii clasei de bază.

Pentru a transmite argumente constructorului clasei de bază se procedează in felul următor:

constructor-derivat(arg-list) : base1(arg-list), base2(arg-list),
// ... baseN(arg-list)
{// corpul constructorului derivate }

Trebuie să înțelegeți că dacă clasa de bază implementează un constructor cu parametri, atunci toți constructorii clasei derivate trebuie să invoce acel constructor şi să-i transmită argumente, chiar dacă constructorul clasei derivate nu ia parametri. Dacă clasa de bază supraincarcă mai mulți constructori atunci puteți alege ce constructor de bază va fi invocat. Dacă nu specificați niciun constructor de bază, atunci va fi apelat constructorul implicit (fără parametri) al clasei de bază (dacă există).

65.Permiterea accesului.

66.Clase de bază virtuale.

Într-o moştenire multiplă este posibil ca o clasă să fie moştenită indirect de mai multe ori, prinintermediul unor clase care moştenesc, fiecare în parte, clasa de bază. De exemplu:
class L { public: int x;};class A : public L { /* */};class B : public L { /* */};class D : public A, public B { /* */};

Acestă moştenire se poate reprezenta printr-un graf aciclic direcționat care indică relațiile dintre sub obiectele unui obiect din clasa D. Din graful de reprezentare a moştenirilor, se poate observa faptul că baza L este replicată în clasa D.

67.**Functii virtuale si polimorfism.**

O funcție virtuală este o funcție care este declarată de tip virtual în clasa de bază şi redefinită într-o clasă derivată. Redefinirea unei funcții virtuale într-o clasă derivată domină definiția funcției în clasa de bază. Funcția declarată virtual în clasa de bază acționează ca o descriere generică prin care se defineşte interfața comună, iar funcțiile redefinite în clasele derivate precizează acțiunile specifice fiecărei clase derivate.

In esenta, o functie virtuala declarata in clasa de baza actioneaza ca un substitut pentru pastrarea datelor care specifica o clasa generala de actiunii si declara forma interfetei. Redefinirea unui funtii virtuale intr-o clasa derivata ofera operatiile efective pe care le executa functia. Altfel spus, o functie virtuala defineste o clasa generala de actiuni. Redefinirea ei introduce o metoda specifica.

Implementarea obiectelor polimorfice se realizează prin intermediul functiilor virtuale.
Sintaxa declarării unei functii virtuale:
virtual <tip_functie> <nume_functie> ([<lp>]);<tip_functie> reprezintã tipul întors de functie, <lp> este

Când un pointer al clasei de bazã punctează la o functie virtualã din clasa derivatã si aceasta esteapelatã prin intermediul acestui pointer, compilatorul determinã care versiune a functiei trebuieapelatã, tinând cont de tipul obiectului la care punctează acel pointer. Astfel, tipul obiectului lacare punctează determinã versiunea functiei virtuale care va fi executatã.

Polimorfismul din timpul rularii este permis doar daca accesul se face print-un pointer al clasei de baza.

68.**Functiile virtuale.**

In programarea orientată pe obiecte (POO), o funcție virtuală sau metodă virtuală este o funcție al cărei comportament, in virtutea declarării acesteia ca fiind "virtuală", este

determinat de către definiția unei funcții cu aceea□i semnătură cea mai indepărtată pe linia succesorală a obiectului in care este apelată. Acest concept este o foarte importantă parte din porțiunea de polimorfism a paradigmei de programare pe obiecte (POO).

Conceptul de funcție virtuală rezolvă următoarea problemă:

In POO cand o clasă derivată mo□tene□te de la o clasă de bază, un obiect al clasei derivate poate fi considerat ca fiind (sau convertit la) o instanță a clasei de bază sau a unei clase derivate din aceasta. Dacă există funcții ale clasei de bază ce au fost redefinite in clasa derivată, apare o problemă cand un obiect derivat a fost convertit la (este referit ca fiind de) tipul clasei de bază. Cand un obiect derivat este considerat ca fiind de tipul clasei de bază, comportarea dorită a apelului de funcție este nedefinită.Distincția dintre virtual (dinamic) □i static este făcută pentru a rezolva această problemă. Dacă funcția in cauză este etichetată drept "virtuală" atunci funcția clasei derivate va fi apelată (dacă ea există). Dacă e statică, atunci va fi apelată funcția clasei de bază.

De exemplu, o clasă de bază Animal poate avea o funcție virtuală eat. Sub-clasa Fish va implementa eat() intr-un mod diferit față de sub-clasa Wolf, dar poți invoca metoda eat() in cadrul oricărei instanțe de clasă de referință Animal, și obține o comportare specifică clasei derivate pentru care această metodă a fost redefinită. Aceasta ii dă posibilitatea programatorului să proceseze o listă de obiecte din clasa Animal, spunandu-i fiecăruia pe rand să mănance (apeland funcția eat()), fără a ști ce fel de animal se poate afla pe listă. Nici măcar nu trebuie să știi cum mănancă fiecare animal, sau care ar putea fi setul complet de tipuri posibile de animale.

69.**Atributul virtual este mostenit.**

Cand o functie virtuala este mostenita, se mosteneste si natura sa virtuala. O functie ramane virtuala indiferent de cate ori este mostenita.

Atributul virtual este mostenit: cand o functie virtuala este mostenita se mosteneste si atributul virtual.

```
class D: public B1, public B2{
… void fctVirtuala(){
cout << "Acesta este o functie virtuala in clasa derivata D\n"; }} ;
void f(){
BB *pbb, bb;
B1 b1;
B2 b2;
pbb = &bb; //indica spre baza
pbb->fctVirtuala(); //acces la functia virtuala a calasei parinte
pbb = &b1; //indica spre clasa derivata B1
pbb->fctVirtuala(); //acces la functia virtuala a calasei derivate B1
pbb = &b2; //indica spre clasa derivata B1
pbb->fctVirtuala(); //acces la functia virtuala a calasei derivate B2
} void f(){ D dd;
pbb = &dd;
pbb->fctVirtuala(); //acces la functia virtuala a clasei derivate D  }
```

70.**Functiile virtuale sunt ierarhizate.**

Cand o functie este declarata ca fiind virtual intr-o clasa de baza, ea poate fi suprascrisa de o clasa derivata. Totusi, functia nu trebuie neaparat sa fie suprascrisa. Daca o clasa derivata nu

suprascrie functia virtuala, atunci, cand un obiect din acea clasa derivata are acces la functie, este folosita functia definita de clasa de baza.

```
#include<iostream.h>
class B  };
class D1 : public B  };
class D2 : public B ;
void main()
```

N.B. Atunci cand o clasa derivata nu suprascrie o functie virtuala, este folosita prima redefinire gasita in ordinea inversa a derivarii.

71.**Functii virtuale pure.**

O functie virtuala pura este o functie virtuala care nu are definitie in clasa de baza.

virtual tip nume-functie(lista-de-parametri) = 0;

Cand o functie virtuala este construita pura, orice clasa derivata trebuie sa-i asigure o definitie. In cazul in care clasa derivata nu suprascrie functia virtuala pura, va rezulta o eroare in timpul compilarii.

```
#include<iostream.h>
class NUMAR
        virtual void arata() = 0; //functie virtuala pura };
class HEX : public NUMAR };
class DEC : public NUMAR };
class OCT : public NUMAR };
void main()
```

72.Clase abstracte.

O clasa care contine cel putin o functie virtuala pura se numeste abstracta.

De cele mai multe ori, o funcție declarată de tip virtual în clasa de bază nu defineşte o acțiune semnificativă şi este neapărat necesar ca ea să fie redefinită în fiecare din clasele derivate. Pentruca programatorul să fie obligat să redefinească o funcție virtuală în toate clasele derivate în careeste folosită această funcție, se declară funcția respectivă virtuală pură. O funcție virtuală pură este o funcție care nu are definiție în clasa de bază, iar declarația ei arată în felul următor: virtual tip_returnat nume_functie(lista_argumente) = 0;

O clasă care conține cel puțin o funcție virtuală pură se numeşte clasă abstractă. Deoarece o clasă abstractă conține una sau mai multe funcții pentru care nu există definiții, nu pot fi create instanțe din acea clasă, dar pot fi creați pointerişi referințe la astfel de clase abstracte. O clasă abstractă este folosită în general ca o clasă fundamentală, din care se construiesc alte clase prin derivare.Orice clasă derivată dintr-o clasă abstractă este, la rândul ei clasă abstractă (şi deci nu se pot creainstanțe ale acesteia) dacă nu se redefinesc toate funcțiile virtuale pure moştenite. Dacă o clasă redefineşte toate funcțiile virtuale pure ale claselor ei de bază, devine clasă normală şi pot ficreate instanțe ale acesteia.

73.**Utilizarea funcțiilor virtuale.**

Una dintre cele mai puternice si mai flexibile cai de introducere a abordarii “o interfata, metode multiple” este folosirea functiilor virtuale, a claselor abstracte si a polimorfismului din timpul rularii. Folosind aceste caracteristici, creati o ierarhizare care trece de la general la specific (de la baza la derivat).

```
#include<iostream.h>
class CONVERT
        double daconv()
        double dainit()
        virtual void calcul() = 0;  };
class LITRI_GALOANE : public CONVERT
        void calcul()  };
class FAHRENHEIT_CELSIUS : public CONVERT
        void calcul()  };
void main()
```

74. **Legături initiale/ulterioare.**

Legarea inițială (Early binding) se referă la evenimentele ce au loc in timpul compilării. Legarea inițială se produce atunci cand toate informațiile necesare apelării unei funcții se cunosc in timpul compilării. Mai simplu, legarea inițială inseamnă că un obiect şi apelul funcției se leagă in timpul compilării. Apelările normale de funcții, supraincărcarea funcțiilor şi operatorilor (polimorfism in timpul compilării) sunt exemple de legare inițială. Deoarece toate informațiile necesare apelului unei funcții se ştiu din timpul compilării, aceste tipuri de apelare sunt foarte rapide.

Legarea tarzie (Late binding) se referă la apelurile de funcții determinate in timpul execuției. Funcțiile virtuale sunt folosite pentru a activa legarea tarzie. Aşadar, obiectul şi funcția vor fi legați in timpul execuției. Avantajul legării tarzii este flexibilitatea, dar deoarece apelul funcției este determinat la runtime, execuția este puțin mai inceată.

75. **Sabloanele.**

In programarea calculatoarelor, □abloanele sunt o caracteristică a limbajului de programare C++ ce permit scrierea de cod fără a lua in considerare tipul de dată ce va fi utilizat pană la urmă. □abloanele permit programare generică in C++.

□abloanele sunt foarte utile programatorilor in C++, mai ales cand sunt combinate cu tehnica mo□tenirilor multiple □i a supraancărcării operatorilor. Biblioteca Standard de □abloane (STL) a limbajului C++ aduce multe funcții utile intr-un cadru de □abloane conectate.

Există două feluri de □abloane. Un □ablon funcție se comportă ca o funcție ce poate accepta argumente de tipuri foarte diferite. De exemplu, Biblioteca Standard de □abloane a limbajului C++ conține □ablonul funcție max(x, y) ce returnează x sau y, pe cel mai mare dintre cele două argumente. max() ar putea fi declarat cam a□a:

```
template <class a>
a max(a x, a y)
{ if (x < y)
return y;
else
return x; }
```

Acest □ablon poate fi apelat intr-un mod identic cu apelul de funcție:

```
cout << max(3, 7); // afi□ează 7
```

Un □ablon clasă extinde acela□i concept peste clase. □abloanele clasă sunt folosite de

obicei pentru a face containere generice. De exemplu, biblioteca STL are un container de tip listă inlănțuită. Pentru a face o listă inlănțuită de intregi, se va scrie list<int>. O listă de șiruri de caractere este notată list<string>. O listă are un set de funcții standard asociate, ce funcționează indiferent ce vei pune intre paranteze.

76.**Functii generice.**

O funcție generică defineşte un set de operații aplicabil mai multor tipuri de dată. Tipul de dată este transmis ca parametru funcției care va opera asupra acelui tip.

Printr-o funcție generică, o singură procedură (sau set de operații) poate fi aplicată unui domeniu larg de date. De exemplu, algoritmul Quicksort este acelaşi indifferent că operează asupra unor intregi (ex: int) sau numere reale (double). Diferența constă doar in tipul datelor. După ce ați creat o funcție generică, compilatorul va genera automat codul corect pentru tipul de dată folosit atunci cand este apelată funcția. O funcție generică este practic o funcție care se supraincarcă singură.

77.**() functie cu două tipuri generice.**

Instrucțiunea template acceptă mai mult de un parametru generic. Parametrii se separă prin virgulă. De exemplu:

```
template<typename T, typename V>
void printCrap(T x, V y)
{ cout << "Printing crap: " << x << ' ' << y << '\n'; }
int main()
{ printCrap(56, "Imi place C++");
printCrap(5690L, 34.67);
return 0; }
```

Aici compilatorul inlocuieşte substituenții T şi V cu tipurile int şi char*, respectiv, long şi double, atunci cand acesta generează instanțele specifice ale printCrap() din main(). Observați că am folosit typename. Puteam la fel de bine folosi class.

78.**Supraîncărcarea explicită a unei functii generice.**

Chiar daca o functie sablon se supraincarca singura cand este necesar, puteti sa o supraincarcati si explicit. Daca supraincarcati o functie generica, atunci ea suprascrie (sau "ascunde") functia generica relativ la acea versiune specifica.

```
#include<iostream.h>
template <class X> void inloc(X &a, X &b) //aceasta surpascrie versiunea genrica a inloc()
void inloc(int &a, int &b)
main()
```

Puteți supraincărca explicit o funcție generică. Această supraincărcare se numeşte specializare explicită sau specializare template.

79.**Restrictii pentru functia generică.**

O prima restrictie legata de functiile template este aceea ca in lista de argumente formale trebuie sa apara toate tipurile generice specificate in prefixul template <class T1, . . . , Tn>. Astfel urmatoarele definitii vor genera erori de compilare:

```
template <class T>
void f1() { ... }
template <class T1, class T2>
T1* f2(T2 t2) { ... }
```

A doua restrictie legata de functiile template este aceea ca ele vor fi apelate doar in cazul in care tipurile parametrilor actuali se potrivesc perfect cu cele ale parametrilor formali. Cu alte cuvinte, la apelul functiilor template nu se fac nici un fel de conversii ale parametrilor. Acest lucru este ilustrat in exemplul de mai jos:

```
template <class T>
void f1(T t1, T t2) { ... }
template <class T1, class T2>
void f1(T1 t1, T2 t2) { ... }
main() {
f2(1,1); // corect: T1 si T2 vor fi int.
f1(12.34, 56); // eroare: cei doi parametrii trebuie sa fie exact de acelasi tip. }
```

80.**Aplicarea functiilor generice.**

81.() sortare generică.

```
#include <stdio.h>
//functia de sortare generica
int sort(void * v, int n, int size, int (*f) (void *, void*)) {
int i,j;
void *aux = malloc(size);
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
//incrementarea pointerilor void* se face cu 1
if (f(v + i*size, v + j*size) > 0) {
memcpy(aux, v + i*size, size);
memcpy(v + i*size, v + j*size, size);
memcpy(v + j*size, aux, size); } }}}
```

82.Clase generice.

Clasele colecții conțin obiecte de un tip particular (de exemplu o listă inlănțuită de intregi sau un tablou de structuri. Se pot defini familii de clase colecții, membrii acestora diferind numai prin tipul elementelor.

Considerăm clasa tablou de intregi: pentru a folosi un tablou de reali, de complec şi sau de şiruri de caractere s-ar putea copia implementarea clasei, modificand tipul datelor şi numele clasei. Astfel am avea clasele intArray, StringArray, ComplexArray, etc.

Familia de clase poate fi reprezentată printr-o clasă generică (parametrizată). Aceasta specifică modul in care pot fi construite clasele individuale, care se deosebesc numai prin tipul elementelor pe care le conțin.

Sintaxa folosită pentru furnizarea parametrilor unei clase generice este:

template <par1, par2,…>

Parametrii pot fi de două categorii: tipuri de date sau constante. Tipurile de date sunt precedate de

cuvântul cheie class, ceea ce nu înseamnă însă că pot fi doar tipuri de date declarate cu class, ci

pot fi orice tip de date. Parametrii care nu sunt precedați de cuvântul class sunt considerați

automat ca fiind constante.

83.**Un exemplu cu două tipuri de date generice.**

O clasă generică se reprezintă astfel:

template <listă_argumente_generice> declarare_clasă;

Instanțierea unei clase generice se face prin:

nume_clasă <listă_argumente_concrete> nume_obiect;

Definim clasa generică (parametrizată):

```
template <class T>
class Array{
public:
Array(int d=10): a(new T[dim]), dim(d){}
~Array(){delete a;}
private:
T *a;
int d; };
```

84.Mediul Builder.

C++Builder este un mediu de dezvoltare rapidă a aplicațiilor produs de filiala CodeGear a Embarcadero Technologies pentru scrierea programelor în limbajul de programare C++. C++Builder combină biblioteca de componente vizuale și un IDE scris în Delphi, cu un compilator C++. Ciclul de dezvoltare este în așa fel încât Delphi primește primul îmbunătățiri semnificative, urmat de C++Builder. Majoritatea componentelor dezvoltate în Delphi pot fi folosite în C++Builder fără modificări, dar nu și invers.

85.**Extinderea IDE Builder a clasei : proprietăti si evenimente.**

A. Propietăți, metode, evenimente . Dezvoltarea rapidă a aplicațiilor inseamnă suport pentru propietățile, metodele şi evenimentele obiectelor (PME). Propietățile permit setarea uşoară a caracteristicilor componentelor. Metodele execută acțiuni asupra obiectelor. Evenimentele permit ca aplicația să răspundă la mesajele Windows, sau la schimbări de stare a obiectelor.

Folosirea modelului PME furnizează un robust şi intuitiv mediu de dezvoltare pentru aplicațiile Windows.

B. C++Builder Help Mediul C++Builder oferă un ghid practic, care conține peste 3000 de pagini de documentație despre IDE, VCL, baze de date şi tehnici de programare.

C. Codurile sursă pentru VCL Mediul C++Builder pune la dispoziție codurile sursă pentru VCL - Visual Component Library, furnizand astfel o unică privire inăuntrul modului in care lucrează C++Builder. VCL furnizează peste 100 de componente reutilizabile care ajută programatorul să construiească aplicații robuste intr-un timp scurt. Aceste componente pot fi modificate pentru a corespunde necesităților din cele mai diverse. C++Builder-ul incude o suită completă de controale Windows95: TreeView, Trackbars, ProgressBars, toolbars, Rich Edit, ListViews, ImageLists, StatusBars etc. Totodată C++Builder include suport pe 32 de biți pentru numele lungi de fişiere, multi-threading şi Win95 API.

86.Paleta de obiecte Builder.

Componentele sunt elemente utilizate pentru a crea aplicații C++Builder. O componentă este de fapt un element de tip UI (user interface). Pot fi vizuale (de exemplu butoanele, cutiile de dialog), sau pot fi non-vizuale (de exemplu timer-ul). Spunem despre o componentă că este

vizuală, dacă ea este vizibilă, sau va fi vizibilă la momentul execuției, iar o componentă este non-vizuală, dacă la momentul proiectării aplicației apare pe formă ca un desen, iar in momentul execuției aplicației devine invizibilă (de exemplu TTimer din pagina System), sau este invizibilă pană in momentul in care este apelată (de exemplu TOpenDialog sau TSaveDialog din pagina Dialogs). Fiecare componentă are atribute care permit controlul aplicației. Componentele sunt grupate in pagini. In forma implicită paginile sunt: Standard, Win95, Additional, Dat Access, Data Control, Win31, Internet, Dialogs, System, QReport, ActiveX . figura 1.6 De exemplu cele mai folosite componente sunt cele din pagina Standard, care conține cutii de dialog, meniuri, butoane etc. Pentru a obține help despre fiecare dintre ele, executați click pe componenta dorită, iar apoi apăsați pe F1.

O componentă specială este şi forma, care are la randul ei ataşte propietăți, metode, evenimente etc. Aşezarea unei componente pe o formă se poate face in mai multe moduri:
- dacă dorim plasarea componentei in mijlocul formei atunci executăm dublu click pe forma respectivă.
- dacă dorim să plasăm componenta in alt loc decat centrul formei, atunci executăm un click pe componentă, iar apoi incă un click in locul dorit pe formă. Colțul din stanga sus al componentei va coincide cu locul unde am executat cel de-al doilea click. In aceste două cazuri dimensiunile componentei vor fi cele implicite. Se pot modifica aceste dimensiuni, fie din Object Inspector (vezi mai jos), fie cu ajutorul mouse-ului.

87.<u>Mediul Builder.IDE Builder.</u>

C++Builder este un mediu de dezvoltare rapidă a aplicațiilor produs de filiala CodeGear a Embarcadero Technologies pentru scrierea programelor in limbajul de programare C++. C++Builder combină biblioteca de componente vizuale şi un IDE scris in Delphi, cu un compilator C++. Ciclul de dezvoltare este in aşa fel incat Delphi primeşte primul imbunătățiri semnificative, urmat de C++Builder. Majoritatea componentelor dezvoltate in Delphi pot fi folosite in C++Builder fără modificări, dar nu şi invers. C++Builder include unelte care permit dezvoltarea vizuală bazată pe drag-and-drop,făcand programarea mai facilă prin implementarea unui GUI builder WYSIWYG in IDE.

IDE-ul C++Builder este împărțit în trei părți. Fereastra superioară poate fi considerată partea principală a IDE-ului. Pe lângă meniul principal ea conține şi bara de butoane cu cele mai folositeoperații în stânga şi selectorul de componente utilizabile în dreapta.

Pentru o mai uşoară regăsire, componentele sunt împărțite în tipuri de componente, fiecare tipavând un tab în paleta de componente. Pentru a utiliza o componentă, este suficient să o selectămîn paletă după care să dăm click pe forma pe care dorim să amplasăm componenta, la poziționadorită.O componentă este o bucată de cod software care va îndeplini în aplicație o anume funcție predefinită (cum ar fi o etichetă, un câmp de editare sau o listă).Cea de-a doua parte a IDE-ului C++Builder este inspectorul de obiecte (Object Inspector) careeste amplasat în mod implicit în stânga ecranului. Prin intermediul inspectorului de obiecte vommodifica proprietățile componentelor şi modul cum sunt tratate evenimentele referitoare la ele (interacțiunile utilizatorului cu componenta respectivă). În inspectorul de obiecte apare întotdeauna lista de proprietăți a componentei curent selectate, dublată de lista de evenimente care pot afecta componenta respectivă, dar mai poate apărea (dacă este selectată) şi ierarhia de componente a aplicației.În dreapta inspectorului de obiecte este spațiul de lucru al C++Builder

(cea de-a treia parte).Inițial în acest spațiu este afișat editorul de forme. Acesta este utilizat pentru a amplasa, muta saudimensiona diverse componente ca parte a creării formelor aplicației. În spatele editorului deforme se află parțial ascunsă fereastra editorului de cod de program (locul în care vom scrie cod pentru programele noastre).

88.Mediul Builder. Meniul, ferestrele de lucru.

MAIN MENU

File pentru a deschide, crea, salva, inchide project-uri şi fişiere;
Edit pentru prelucrare de texte şi componente
View pentru a afişa, sau ascunde elemente ale mediului;
Project pentru a compila o aplicație;
Run
Component pentru a crea sau a instala o componentă.
DataBase pentru manipulare de baze de date.
Workgroups pentru manipularea proiectelor mari.
Tools pentru a rula programele utilitare disponibile, fără a părăsi mediul C++Builder;
Options pentru a controla comportamentul mediului de dezvoltare;
Help pentru a obține ajutor in diversele faze de utilizare a mediului

Forma

Intreaga parte vizibilă a unei aplicații este construită pe un obiect special numit formă(ca cea din figura 1.2). O formă liberă este creată de fiecare dată cand este lansat in execuție mediul C++Builder. O aplicație poate avea mai multe forme. Adăugarea de noi forme unei aplicații se face selectand comanda New Form din meniul File. Pe formă se pot aşeza şi aranja componente vizulale şi non-vizuale care alcătuiesc interfața cu utilizatorul. Fiecărei forme ii sunt asociate două fişiere cu exensiile .cpp respectiv .h (in cazul formei de mai sus unit1.cpp şi unit1.h)

Editorul de cod

Mediul C++Builder are o fereastră unde programatorul poate scrie codul unei aplicații. Editorul de cod este un editor ASCII complet şi poate deschide mai multe fişiere simultan.
Bara cu instrumente Aceasta reprezintă o scurtătură la comenzile aflate in MainMenu.
Tabelul cu proprietăți ale obiectelor Acest tabel (Object Inspector) care face legătura intre interfața aplicației şi codul scris de programator are două funcții:

• setează propitetățile componentelor aflate in formă.
• creează şi ajută la navigatul prin handler-ele de evenimente. Un handler de evenimente se execută in **Object Selector** In capătul de sus al lui se află Object Selector care conține toate componentele de pe formă impreună cu tipul lor.

89.**Mediul Builder.Componente, proprietăti.**

90.Mediul Builder. Evenimente, module.

Pagina evenimentelor(Events) a inspectorului obiectelor arată lista evenimentelor determinate de component(programarea pentrru sistemele de operare cu interfață grafică a utilizatorului,in special,pentru Windows 95 sau Windows NT şi presupune descrierea reacției aplicației la anumite evenimente, pe cind insăşi sistemul de operare se ocupă de interogarea computer-ului cu scopul determinării dacă se realizează un eveniment).Fiecare component are un set propriu de procesare a evenimentelor. In C++ Builder este nevoie de a scrie funcții ce procesează evenimentele şi corelarea evenimentelor cu aceste funcții.In procesul creării a astfel de procesoare de un anumit tip se obligă programul să realizeze funcția scrisă, dacă se va realiza evenimentul dat. Fiecare componaneta are evenimentele sale posibile.

91. Mediul Builder – exemple de utilizare a componentelor: Form.

92. Proiectarea BD.

Metodologia de proiectare, consta intr-o abordare structurata, in care se utilizeaza proceduri, tehnici, instrumente si documentatii, pentru a sustine si facilita procesul de proiectare. O metodologie de proiectare consta in mai multe faze, continand etape care indruma proiectantul in alegerea tehnicilor adecvate fiecarei etape a proiectului; de asemenea il ajuta la: planificare, administrare, control si evaluarea proiectelor de dezvoltare a bazelor de date. In final are loc o abordare structurata de analiza si modelare a unui set de cerinte privind BD, intr-o maniera standardizata si organizata.

Metodologia de proiectare a BD, consta din trei faze principale:

- Proiectarea conceptuala a BD
- Proiectarea fizica a BD
- Proiectarea logica a BD

Cererea in C++ Builder este un obiect care reprezintă o colecție de date. De obicei pentru crearea unei cereri se utilizează componenta TQuery – urmaş al clasei abstracte TDataSet.

93. Utilizarea componentelor: Form, TQRBand, QuickReport, TDataSource, TTable, TField,TBDGrid, TQuery, SQL Explorer.

Form.Ea este utilizata pentru suport a obiectelor ce sunt create pe ea. Formele pot fi de diferite tipuri (parinte, copil), si create in moduri diferite. Toate C + + Builder formularele sunt definite intr-o clasă C + +, ceea ce inseamnă căacestea sunt obiecte in măsura in care cererea dumneavoastră este in cauză. Pentru ca sunt obiecte, aveți control asupra a ceea ce atributele şi metodele care le conțin.Acest lucru inseamnă că puteți trata un C + + Builder formă ca in cazul in care s-auorice alt exemplu C + + obiect, adăugand metode şi atribute pentru a se potrivinevoilor dumneavoastră. Folosind tehnicile pe care le vom acoperi, puteți creșteeficiența codul dumneavoastră prin menținerea central attribute şi obiecte care vor fi utilizate pe parcursul cererii dumneavoastră. In continuare, să ne uităm la cazul in care pentru a stoca aceste informații şi cateva modalități pentru a ajunge la ea.

TQRBand - componenta, care este parte a raportului - un container de date (de exemplu, titlul
raportului, in partea de sus sau de jos a paginii antet titlu coloane sau notele de subsol ale grupului, etc.) Componentele sunt tipărite cu TQRBand in funcție de tipul lor in locurile corespunzătoare din raport, indiferent de poziția lor relativă peformular.Caracteristica cea mai utilizată de această componentă - BandType, de tip"bar" (stranitsyili banda de subsol, "trupa" de date, etc). Valori posibile: rbTitle-raporttitlu, rbPageHeader – antetul paginii, rbColumnHeader-antetul de coloană intr-un raport multicolumn, rbDetail - polosas tabelare de date (repetată ori de cate ori există randuri vnabore baza de date avlyayuschemsya de raport), rbPageFooter - kolontitulstranitsy mai mici, rbOverlay - de fundal a paginii este tipărită in colțul dinstanga sus kazhdoystranitsy, rbGroupHeader - un antet de grup, rbSubDetail -"benzi" de date tabelare pentru Detaliu-masă, rbGroupFooter - kolontitulgruppy mai mici, rbSummary

– tipărite la sfarşitul raportului). Domeniul imobilier BandTypecomponentă sozdannogonami atribui valoarea rbTitle.

QuickReport este un set de componente şi a controalelor, care permite rapoarte să fie proiectate şi previzualizate in Delphi şi C + + Builder IDEuri. Aplicatii inclusiv funcționalitatea Quickreport pot fi apoi utilizate free.

Componenta DataSource acționează in calitate de mediator intre componentele TDataSet (TTable, TQuery, TStoredProc) şi componentele Data Controls – elemente de dirijare care asigură reprezentarea datelor in formular. Componentele TDataSet dirijează legăturile cu biblioteca Borland Database Engine (BDE), iar componente DataSource dirijează legăturile cu datele din componentele Data Controls.

Cea mai simplă metodă de adresare la tabelele bazelor de date este cu utilizarea componentei **TTable**, care permite accesul la un tabel. Pentru aceasta cel mai des utilizăm următoarele proprietăți: Active, DatabaseName, TableName, Exclusive, ReadOnly.

Componenta TDBGrid asigură metoda de afişare pe ecran a randurilor de date din componentele TTable şi TQuery sub formă de tabel. Aplicația poate să utilizeze TDBGrid pentru afişare, includere, nimicire, redactare a datelor BD. De obicei DBGrid se utilizează in combinație cu DBNavigator, deşi pot fi utilizate şi alte elemente de interfață, incluzand in procesoarele lor de evenimente metodele First, Last, Next, Ptior, Insert, Delete, Edit, Append, Post, Cancel a componentei TTable.

Ca şi in cazul componentei TTable, componenta TDataSource gestionează cu interacțiunea dintre componentele Data Controls şi componenta **TQuery**. De obicei aplicația are cate o componentă DataSource pentru fiecare componentă TQuery. Active,Eof, DatabaseName, DataSource, Fields, Params, SQL

SQL Explorer este un instrument la indemană atunci cand se lucrează cu baze de date. Pentru a incepe, selectați baza de date meniu / Exploreaza sau de a rula ca o aplicație independentă. In stanga paginii Explorer panoul de Bazele de date suntprezentate ca lista dropdown de toate proprietățile enumerate in pseudonime de configurare BDE fişier. In panoul din dreapta, aveți posibilitatea să vizualizațiconținutul tabelelor, intra şi a executa SQL-interogare de la masă şi de a obține informații despre baza de date alias (driver de baze de date, locația, limba folosităde către conducătorul auto şi alți parametri conținute in BDE fişierul de configurare),informații despre tabelele (tipul de tabela, versiune, data ultimei actualizări, etc) şi,dacă este necesar, să le modifice.