OBIECTE

-au stare si actiuni (metode/functii)
-au interfata(actiuni) si o parte ascunsa(starea)
-sunt grupate in clase, obiecte cu aceleasi proprietati

CLASE

-mentioneaza proprietatile generale ale obiectelor din clasa respectiva
-clasele nu se pot “rula”
-folositoare la encapsulare (ascunderea informatiei)
-reutilizare de cod: mostenire
-clasele au functii membru si variabile membru
    * se creeaza un tip nou de date
    * un obiect instantiaza clasa
    * functiile membru sunt date prin semnatura
    * pentru definirea fiecarei functii se foloseste ::
-:: scope resolution operator
-mai multe clase pot folosi acelasi nume de functii
-class *nume_clasa {*
private variabile si functii membru
specificator_de_acces:
variabile si functii membru
specificator_de_acces:
variabile si functii membru
// ...
specificator_de_acces:
variabile si functii membru
} lista_obiecte;

- un membru nestatic al clasei nu poate avea initializare
-nu putem avea ca membri obiecte de tipul clasei (putem avea pointeri la tipul clasei)
-nu auto, extern, register

STRUCT SI CLASS

-singura diferenta : struct care default membri ca public, iar class ca private
-struct defineste o clasa (tip de date)
-putem avea in struct si functii
-a nu se folosi struct pentru clase

UNION SI CLASS

-la fel ca struct
-toate elementele de tip data folosesc aceeasi locatie de memorie
-membrii sunt publici (by default)

UNION CA SI CLASS

-union nu poate mosteni
-nu se poate mosteni din union
-nu poate avea functii virtuale (nu avem mostenire)
-nu avem variabile de instanta statice
-nu avem referinte in union
-nu avem obiecte care fac overload pe =
-obiecte cu constructor/destructor definiti nu pot fi membri in union

UNION ANONIME

-nu au nume pentru tip
-nu se pot declara obiecte de tipul respectiv
-folosite pentru a spune compilatorului cum se tin variabilele respective in memorie
  *folosesc aceeasi locatie de memorie
-variabilele din union sunt accesibile ca si cum ar fi declarate in blocul respectiv
-nu poate avea functii
-nu poate avea private sau protected (fara functii nu avem acces la altceva)
-union-uri anonime globale trebui precizate ca static

FUNCTII PRIETEN

-cuvantul : friend
-sunt folosite pentru accesarea campurilor protected, private din alta clasa
-folositoare la overload-area operatorilor, pentru unele functii de I/O si portiuni interconectate

FUNCTII INLINE

-executie rapida
-este o sugestie/cerere pentru compilator
-pentru functii foarte mici
-pot fi si membri ai unei clase

FUNCTII CARE INTORC OBIECTE

-o functie poate intoarce obiecte
-un obiect temporar este creat automat pentru a tine informatiile din obiectul de intors
-acesta este obiectul care este intors
-dupa ce valoarea a fost intoarsa, acest obiect este distrus
-probleme cu memoria dinamica : solutie polimorfism pe = si pe constructorul de copiere

CLASE PRIETEN

-daca avem o clasa prieten, toate functiile membre ale clasei priten au acces la membrii privati ai clasei

CLASE LOCALE

-putem defini clase in clase sau functii
-class este o declaratie, deci defineste un scop
-operatorul de rezolutie de scop ajuta in aceste cazuri
-rar utilizate clase in clase

OVERLOAD DE FUNCTII

-folosirea aceluiasi nume pentru functii diferite (functii diferite, dar cu inteles apropiat)
-compilatorul foloseste numarul si tipul parametrilor pentru a diferentia apelurile
-tipul de intoarcere sau tipuri care par diferite nu sunt suficiente pentru diferentiere (eroare la compilare)

```
int myfunc(int i); // Error: differing return types are
float myfunc(int i); // insufficient when overloading.

void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

-se poate folosi si pentru functii complet diferite (nerecomandat)

POINTERI CATRE FUNCTII POLIMORFICE

-putem avea pointeri catre functii (C)
-putem avea pointeri care functii polimorfice
-cum se defineste pointerul ne spune catre ce versiune a functiei cu acelasi nume aratam

```
#include <iostream>
using namespace std;

int myfunc(int a);
int myfunc(int a, int b);

int main()
{
  int (*fp)(int a); // pointer to int f(int)
  fp = myfunc; // points to myfunc(int)
  cout << fp(5);
  return 0;
}

int myfunc(int a)
{  return a;  }

int myfunc(int a, int b)
{  return a*b;  }
```

Semnatura functiei din definitia pointerului ne spune ca mergem spre functia cu un parametru - trebuie sa existe una din variantele polimorfice care este la fel cu definitia pointerului

AMBIGUITATI PENTRU POLIMORFISM DE FUNCTII

-erori la compilare
-majoritatea datorita conversiilor implicite

```
int myfunc(double d);
// ...
cout << myfunc('c'); // not an error, conversion applied
```

```
#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);

int main()
{
  cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
  cout << myfunc(10); // ambiguous
  return 0;
}

float myfunc(float i)
{
  return i;
}

double myfunc(double i)
{
  return -i;
}
```

-problema nu e de definire a functiilor mvfunc
-problema apare la apelul functiilor

ARGUMENTE IMPLICITE PENTRU FUNCTII

-putem defini valori implicite pentru parametrii unei functii
-valorile implicite sunt folosite atunci cand acei parametri nu sunt dati la apel

```
 void myfunc(double d = 0.0)
{
// ...
}
…
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

Argumentele implicite:
-dau posibilitatea de flexibilitate
-majoritatea functiilor considera cel mai general caz, cu parametrii impliciti putem sa chemam o functie pentru cazuri particulare
-multe functii de I/O folosesc argumente implicite
-nu avem nevoie de overload

PARAMETRII IMPLICITI

-se specifica o singura data
-pot fi mai multi
-toti sunt la dreapta
-putem avea param. Impliciti in definitia constructorilor
  *nu mai facem overload pe constructor
  *nu trebuie sa ii precizam mereu la declarare
-modul corect de folosire este de a defini un asemenea parametru cand se subintelege valoarea implicita
-daca sunt mai multe posibilitati pentru valoarea implicita e mai bine sa nu se foloseasca (lizibilitate)
-cand se foloseste un parametru implicit nu trebuie sa daca probleme in program

MOSTENIREA

-incorporarea componenelor unei clase in alta
-refolosire de cod
-detalii mai subtile pentru tipuri si subtipuri
-clasa de baza, clasa derivata
-clasa derivata contine toate elementele clasei de baza, mai adauga elemente noi

-terminologie:
  * clasa de baza, clasa derivata
  * superclasa, subclasa
  *parinte, fiu

CONSTRUCTORI/DESTRUCTORI

-constructor: executat la crearea obiectului
-destructor: executat la distrugerea obiectului
-obiecte globale:
  *constructorii executati in ordinea in care sunt definite obiectele
  *destructorii :dupa ce main s-a terminat in ordinea inversa a constructorilor
-initializare automata
-obiectele nu sunt statice
-constructor: functie speciala, numele clasei
-constructorii nu pot intoarce valori (nu au tip de intoarcere)
-constructorii/destructorii sunt chemati de fiecare data cand o variabila de acel tip este creata/distrusa. Declaratii active, nu pasive.
-destructori: reversul, executa operatii cand obiectul nu mai este folositor
-memory leak

CONSTRUCTORI PARAMETRIZATI

-trimitem argumente la constructori
-putem defini mai multe variante cu mai multe numere si tipuri de parametrii
-overload de constructori

CONSTRUCTOR DE COPIERE

-apel de functie cu obiect ca parametru, apel de functie cu obiect ca variabila de intoarcere
*in aceste cazuri, un obiect temporar este creat, se copiaza prin constructorul de copiere in obiectul temporar si poi se continua
*deci vor fi din nou doua distrugeri de obiecte din clasa respectiva(una pentru parametru, una pentru obiectul temporar
-C++ il defineste pentru a face o copie identica pe date
-constructorul e folosit pentru initializare, constructorul de copiere e folosit pentru un obiect deja initializat, doar copiaza
-vrem sa folosim starea curenta a obiectului, nu starea initiala a unui obiect din clasa respectiva
OBS: Constructorul de copiere este folosit doar la initializari
Daca avem array a(10);
array b(10);
b=a;
Nu este initializare, este copiere de stare. Este posibil sa trebuiasca redefinit si operatorul mai tarziu.

DESTRUCTORI PENTRU OBIECTE TRANSMISE CATRE FUNCTII

-trebuie sa distrugem obiectul respectiv
-chemam destructorul
-putem avea probleme cu obiecte care folosesc memoria dinamic: la distrugere copia elibereaza memoria, obiectul din main este defect (nu mai are memorie alocata)
-in aceste cazuri trebuie sa redefinim operatorul de copiere (copy constructor)

POLIMORFISM PE CONSTRUCTORI

-foarte comun sa fie supraincarcati
-de ce?
*flexibilitate
-putem avea mai multe posibilitati pentru initializarea/construirea unui obiect
-definim constructori pentru toate modurile de initializare
-daca se incearca initializarea intr-un alt fel decat cele definite: eroare la compilare
*pentru a putea defini obiecte initializate si neinitializate
-important pentru array-uri dinamice de obiecte
-nu se pot initializa obiectele dintr-o lista alocata dinamic
-asadar, avem nevoie de posibilitatea de a crea obiecte neinitializate (din lista dinamica) si obiecte initializate (definite normal)
*constructori de copiere: copy constructors
-pot aparea probleme cand un obiect initializeaza un alt obiect
Myclass B=A;
-aici se copiaza toate campurile (starea) obiectului A in obiectul B

-problema apare la alocare dinamica de memorie: A si B folosesc aceeasi zona de memorie pentru ca pointerii arata in acelasi loc
-destructorul lui Myclass elibereaza aceeasi zona de memorie de doua ori (distruge A si B)

MEMBRI STATICI: de tip date

-variabila precedata de “static”
-o singura copie din acea variabila va exista pentru toata clasa
-deci fiecare oviect din clasa poate accesa campul respectiv, dar in memorie nu avem decat o singura variabila definita astfel
-variabilele initialiate cu 0 inainte de crearea primului obiect
-o variabila statica declarata in clasa nu este definita (nu s-a alocat inca spatiu pentru ea)
-deci trebuie definita in mod global in afara clasei
-aceasta definitie din afara clasei ii aloca spatiu in memorie
*sa se redefineasca variabila statica in afara clasei

Variabile statice de instanta:

-pot fi folosite inainte de a avea un obiect din clasa respectiva
-in continuare: variabila de instanta definita static si public
-ca sa folosim o asemenea variabila de instanta folosim operatorul de rezolutie de scop cu numele clasei
-folosirea variabilelor statice de instanta: semafoare (controlul asupra unei resurse pe care pot lucra mai multe obiecte), numararea obiectelor dintr-o clasa
-folosirea variabilelor statice de instanta elimina necesitatea variabilelor globale
-folosirea variabilelor globale aproape intotdeauna violeaza principiul encapsularii datelor, deci nu este in concordanta cu OOP

Functii statice pentru clase:
-au dreptul sa foloseasca doar elemente statice din clasa (sau accesibile global)
-nu au pointerul “this”
-nu putem avea variabila statica si non-statica pentru o functie
-nu pot fi declarate ca 'virtual” (legat de mostenire)
-folosire limitata (initializare de date statice)

SUPRAINCARCAREA OPERATORILOR

-majoritatea operatorilor pot fi supraincarcati
-similar ca la functii
-una din proprietatile C++ care ii confera putere
-s-a facut supraincarcarea operatorilor pentru operatii de I/O (<<. >>)
-supraincarcarea se face definind o functie operator: membru al clasei sau nu

FUNCTII OPERATOR MEMBRI AI CLASEI

- ***ret-type class-name::*operator*#(arg-list)***
  {
  // operations
  }

-# este operatorul supraincarcat (+ - * / ++ – =, etc)
-de obicei ret-type este tipul clasei, dar avem flexibilitate
-pentru operatori unari arg-list este vida

-pentru operatori binari : arg-list contine un element
-apelul la functia operator se face din obiectul din stanga la dreapta (pentru operatori binari)
-operatorul = face copiere pe variabilele de instanta, intoarce *this
-se pot face atribuiri multiple (dreapta spre stanga)

Formele postfix si prefix:

```
// Prefix increment
type operator++( ) {
// body of prefix operator
}
// Postfix increment
type operator++(int x) {
// body of postfix operator
}
```

Supraincarcarea +=, *=, etc

```
loc loc::operator+=(loc op2)
{
  longitude = op2.longitude + longitude;
  latitude = op2.latitude + latitude;
  return *this;
}
```

Restrictii:
-nu se poate redefini si precedenta operatorilor
-nu se poate redefini numarul de operanzi
*rezonabil pentru ca redefinim pentru lizibilitate
*putem ignora un operand daca vrem
-nu putem avea valori implicite; exceptie pentru ()
-nu putem fface overload pe . :: .* ?
-e bine sa facem operatiuni apropiate de intelesul operatorilor respectivi
-este posibil sa facem o decuplare completa intre intelesul initial al operatorului
*exemplu << >>
-mostenire: operatorii (mai putin =) sunt mosteniti de clasa derivata
-clasa derivata poate sa isi redefineasca operatorii

SUPRAINCARCAREA OPERATORILOR CA FUNCTII PRIETEN

-operatorii pot fi definiti si ca functie nemembra a clasei
-o facem functie prietena pentru a putea accesa rapid campurile protejate
-nu avem pointerul 'this'
-deci vom avea nevoiede toti operanzii ca param. pentru functia operator
-primul parametru este operandul din stanga, al doilea parametru este operandul din dreapta

Restrictii:
-nu se pot supraincarca = () [] sau - > ca functii prieten
-pentru ++ sau – trebuie sa folosim referinte

FUNCTII PRIETEN PENTRU OPERATORI UNARI

-pentru ++, – folosim referinta pentru a transmite operandul
*pentru ca trebuie sa se modifice si nu avem pointerul this
*apel prin valoare: primim o copie a obiectului si nu putem modifica operandul (ci doar copia)

PENTRU VARIANTA POSTFIX ++ –

-la fel ca la supraincarcarea operatorilor prin functii membru al clasei: parametru int
// friend, postfix version of ++
friend loc operator++(loc &op, int x)

DIFERENTE INTRE SUPRAINCARCAREA PRIN MEMBRII SAU PRIETENI

-de multe ori nu avem diferente, atunci e indicat sa folosim functii membru

-uneori avem insa diferente: pozitia operanzilor

*pentru functii membru operandul din stanga apeleaza functia operator supraincarcata

*daca vrem sa scriem expresie: 100+ob; probleme la compilare=> functii prieten

SUPRAINCARCAREA NEW SI DELETE

-supraincarcare op. de folosire memorie in mod dinamic pentru cazuri speciale
// Allocate an object.
void *operator new(size_t size)
{
/* Perform allocation. Throw bad_alloc on failure.
Constructor called automatically. */
return pointer_to_memory;
}

// Delete an object.
void operator delete(void *p)
{
/* Free memory pointed to by p.
Destructor called automatically. */

-size_t: predefinit

pentru new: constructorul este chemat automat

-pentru delete: destructorul este chemat automat

-supraincarcare la nivel de clasa sau globala

-daca new sau delete sunt folositi pentru alt tip de date in program, versiunile originale sunt folosite

-se poate face overload pe new si delete la nivel global

*se declara in afara oricarei clase

*pentru new/delete definiti si global si in casa, cel din clasa e folosit pentru elemente de tipul clasei si in rest e folosit cel redefinit global

-pentru array-uri fcem overload de doua ori
// Allocate an array of objects.
void *operator new[](size_t size)
{

```
/* Perform allocation. Throw bad_alloc on failure.
Constructor for each element called automatically. */
return pointer_to_memory;
}

// Delete an array of objects.
void operator delete[](void *p)
{
/* Free memory pointed to by p.
Destructor for each element called automatically.
*/
}
```

SUPRAINCARCAREA []

-trebuie sa fie functii membru, (nestatice)

-nu pot fi functii prieten

-este considerat operator binar

-o[3] se tranfsorma in o.operator[](3)

```
type class-name::operator[](int i)
{
// . . .
}
```

-operatorul [] poate fi folosit si la stanga unei atribuiri (obiectul intors este atunci referinta)

SUPRAINCARCAREA ()

-nu cream un nou fel de a chema functii
-definim un mod de a chema functii cu numar arbitrar de param.

```
double operator()(int a, float f, char *s);
O(10, 23.34, "hi");
echivalent cu O.operator()(10, 23.34, "hi");
```

SUPRAINCARCAREA - >

-operator unar
-obiect - > element
*obiect genereaza apelul
*element trebuie sa fie accesibil
*intoarce un pointer catre un obiect din clasa

```
#include <iostream>
using namespace std;
```

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

int main()
{
  myclass ob;
  ob->i = 10; // same as ob.i
  cout << ob.i << " " << ob->i;
  return 0;
}
```

SUPRAINCARCAREA ,

-operator binar
-ar trebui ignorate toate valorile mai putin a celui mai din dreapta operand

EXCEPTII IN C++

-automatizarea procesarii erorilor
-try, catch throw
-block try arunca exceptie cu throw care este prinsa cu cathc
-dupa ce este prinsa se termina executia din blocul catch si se da controlul “mai sus”, nu se revina la locul unde s-a facut throw (nu e apel de functie)

```
try {
  // try block
}
catch (type1 arg ) {
  // catch block
}
catch (type2 arg ) {
  // catch block
}
catch (type3 arg ) {
  // catch block
}...
catch (typeN arg ) {
  // catch block
}
```

→ care bloc catch este executat este dat de tipul expresiei

→ orice tip de date poate fi folosit ca argument pentru catch

→ daca nu este generata exceptie, nu se executa nici un bloc catch

→ generare de exceptie: throw exc;

-daca se face throw si nu exista un bloc try din care a fost aruncata exceptia sau o functie apelata dintr-un bloc try: eroare

-daca nu exista un catch care sa fie asociat cu throw-ul respectiv (tipuri de date egale) atunci programul se termina prin terminate()

-terminate() poate sa fie redefinita sa faca altceva

-instructiunile catch sunt verificate in ordinea in care sunt scrise, primul de tipul erorii este folosit

-aruncarea de erori din clase de baza si derivate

-un catch pentru tipul de baza va fi executat pentru un obiect aruncat de tipul derivate-sa se puna catch-ul pe tipul derivat primul si apoi catch-ul pe tipul de baza

```
// Catching derived classes.
#include <iostream>
using namespace std;

class B {};

class D: public B {};

int main()
{
  D derived;
  try {
    throw derived;
  }
  catch(B b) {
    cout << "Caught a base class.\n";
  }
  catch(D d) {
    cout << "This won't execute.\n";
  }
  return 0;
}
```

```
// This example catches all exceptions.
#include <iostream>
using namespace std;
void Xhandler(int test)
{   try{
     if(test==0) throw test; // throw int
     if(test==1) throw 'a'; // throw char
     if(test==2) throw 123.23; // throw double
    }
  catch(...) { // catch all exceptions
    cout << "Caught One!\n";
   }
}
int main()
{   cout << "Start\n";
   Xhandler(0);
```

```
    Xhandler(1);
    Xhandler(2);
    cout << "End";
    return 0;
}
```

```
Start
Caught One!
Caught One!
Caught One!
End
```

-se poate specifica ce exceptii arunca o functie
-se restrictioneaza tipurile de exceptii care se pot arunca din functie
-un alt tip nespecificat termina programul:
    *apel la unexpected() care apeleaza abort()
    *se poate redefini
void Xhandler(int test) throw(int, char, double)

Rearuncarea unei exceptii:
-throw; //fara exceptie din catch
-pentru a procesa eroarea in mai multe handlere
-evident avem try in try
-in <exception> avem
    void terminate();   //cand nu exista catch compatibil
    void unexpected(); //cand o functie nu da voie sa fie aruncat tipul respectiv de exceptie

POINTERI

- o variabila care tine o adresa din memorie
-are un tip, compilatorul stie tipul de date catre care se pointeaza
-operatiile aritmetice tin cont de tipul de date din memorie
-pointer ++==pointer+sizeof(tip)
-definitie: tip *nume_pointer;
Merge si tip* nume_pointer;

Operatori pe pointeri

- *, &, schimbare de tip
- *== “la adresa”
- &==“adresa lui”
- int i=7, *j;
- j=&i;
- *j=9;

Aritmetica pe pointeri

- pointer++; pointer--;
- pointer+7;
- pointer-4;
- pointer1-pointer2; intoarce un intreg
- comparatii: <,>,==, etc.

Pointeri si array-uri

-numele array-ului este pointer
-lista[5]==*(lista+5)
-array de pointeri, numele listei este un pointer catre pointeri (dubla indirectare)
-int **p; (dubla indirectare)

ALOCARE DINAMICA

-void *malloc(size_t bytes);

aloca in memorie dinamic bytes si intoarce pointer catre zona respectiva

```
char *p;
p=malloc(100);
```

intoarce null daca alocarea nu s-a putut face
pointer void* este convertit AUTOMAT la orice tip
diferenta la C++: trebuie sa se faca schimbare de tip dintre void* in tip*
p=(char *) malloc(100);
sizeof: a se folosi pentru portabilitate
a se verifica daca alocarea a fost fara eroare (daca se intoarce null sau nu)

```
if (!p) …
```

ALOCARE DINAMICA IN C++

-new, delete
-operatori, nu functii
-se pot folosi inca malloc() si free(), dar vor fi deprecated in viitor
-new: aloca memorie si intoarce un pointer la inceputul zonei respective
-delete: sterge zona respectiva de memorie

p= new tip;

delete p;

-la eroare se “arunca” exceptia bad_alloc din <new>

Alocare de obiecte

-cu new

-dupa creare, new intoarce un pointer catre obiect

-dupa creare se executa constructorul obiectului

-cand obiectul este sters din memorie (delete) se executa destructorul

Obiecte create dinamic cu constructori parametrizati

```
class balance {…}
…

balance *p;
// this version uses an initializer
try {
    p = new balance (12387.87, "Ralph Wilson");
} catch (bad_alloc xa) {
   cout << "Allocation Failure\n";
   return 1;
}
```

-array-uri de obiecte alocate dinamic

*nu se pot initializa

*trebuie sa existe un constructor fara parametri

*delete poate fi apelat pentru fiecare element din array

-new si delete sunt operatori

-pot fi suprascrisi pentru o anumita clasa

-pentru argumente suplimentare exista o forma speciala

p_var = new (lista_argumente) tip;

-exista forma nothrow pentru new: similar cu malloc:

p=new(nothrow) int[20]; // intoarce null la eroare

Eliberare de memorie alocata dinamic

- void free(void *p);

- unde p a fost alocat dinamic cu malloc()

- a nu se folosi cu argumentul p invalid pentru ca rezulta probleme cu lista de alocare dinamica

Array-uri de obiecte

-o clasa da un tip

-putem crea array-uri cu date de orice tip (inclusiv obiecte)

-se pot defini neinitializate sau initializate

clasa lista[10];

sau

clasa lista[10]={1,2,3,4,5,6,7,8,9,0};

pentru cazul initializat dat avem nevoie de constructor care primeste un parametru intreg

```
#include <iostream>
using namespace std;

class cl {
  int i;
public:
  cl(int j) { i=j; } // constructor
  int get_i() { return i; }
};

int main()
{
  cl ob[3] = {1, 2, 3}; // initializers
  int i;

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";
  return 0;
}
```

- initializare pentru constructori cu mai multi parametri

- clasa lista[3]={clasa(1,5), clasa(2,4), clasa(3,3)};

- pentru definirea listelor de obiecte neinitializate: constructor fara parametri

- daca in program vrem si initializare si neinitializare: overload pe constructor (cu si fara parametri)

POINTERI CATRE OBIECTE

- obiectele sunt in memorie

- putem avea pointeri catre obiecte

- &obiect;

- accesarea membrilor unei clase:

-> in loc de .

- in C++ tipurile pointerilor trebuie sa fie la fel

int *p;

float *q;

p=q; //eroare

se poate face cu schimbarea de tip (type casting) dar iesim din verificarile automate facute de C++

POINTER THIS

- orice functie membru are pointerul **this** (definit ca argument implicit) care arata catre obiectul asociat cu functia respectiva

- (pointer catre obiecte de tipul clasei)

- functiile prieten nu au pointerul this

- functiile statice nu au pointerul this

```
#include <iostream>
using namespace std;
class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  double get_pwr() { return this->val; }
};

pwr::pwr(double base, int exp)
{
  this->b = base;
  this_>e = exp;
  this_>val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) this->val = this_>val * this_>b;
}
int main()
{
    pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
    cout << x.get_pwr() << " ";
    cout << y.get_pwr() << " ";
    cout << z.get_pwr() << "\n";
    return 0;
}
```

POINTERI CATRE CLASE DERIVATE

-clasa e baza B si clasa derivata D

-un pointer catre B poate fi folosit si cu D

```
B *p, o(1);
D oo(2);
p=&o;
p=&oo;
```

```
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
};

class derived: public base {
  int j;
public:
  void set_j(int num) { j=num; }
  int get_j() { return j; }
};
int main()
{
  base *bp;
  derived d;
  bp = &d; // base pointer points to derived object
           // access derived object using base pointer
  bp->set_i(10);
  cout << bp->get_i() << " ";
/* The following won't work. You can't access elements of
a derived class using a base class pointer.

  bp->set_j(88); // error
  cout << bp->get_j(); // error
*/
return 0;
}
```

- aritmetica pe pointeri : nu functioneaza daca incrementam un pointer catre baza si suntem in clasa derivata

-se folosesc pentru polimorfism la executie (functii virtuale)

```
// This program contains an error.
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
};
```

```
class derived: public base {
  int j;
public:
  void set_j(int num) {j=num;}
  int get_j() {return j;}
};
int main()
{
  base *bp;
  derived d[2];
  bp = d;
  d[0].set_i(1);
  d[1].set_i(2);
  cout << bp->get_i() << " ";
  bp++; // relative to base, not derived
  cout << bp->get_i(); // garbage value displayed
  return 0;
}
```

POINTERI CATRE MEMBRII IN CLASE

-pointer catre membru
-nu sunt pointeri normali (catre un membru dintr-un obiect) ci specifica un offset in clasa
-nu putem aplica . si - >
-se folosesc .* si - > *

PARAMETRII REFERINTA

-nou la C++
-la apel prin valoare se adauga si apel prin referinta in C++
-nu mai e nevoie sa folosim pointeri pentru a simula apel prin referinta , limbajul ne da acest lucru
-sintaxa: in functie & inaintea parametrului formal

REFERINTE CATRE OBIECTE

-daca transmitem obiecte prin apel prin referinta la functii nu se mai creeaza noi obiecte temporare, se lucreaza direct pe obiectul transmis ca parametru

-deci copy-constructorul si destructorul nu mai sunt apelate

-la fel si la intoarcerea din functie a unei referinte

```
#include <iostream>
using namespace std;
class cl {
  int id;
public:
  int i;
  cl(int i);
  ~cl(){ cout << "Destructing " << id << "\n"; }
  void neg(cl &o) { o.i = -o.i; } // no temporary created
};

cl::cl(int num)
```

```
{      cout << "Constructing " << num << "\n";
       id = num;
}

int main()
{   cl o(1);
    o.i = 10;
    o.neg(o);
   cout << o.i << "\n";
   return 0;
}
```

```
Constructing 1
-10
Destructing 1
```

Intoarcere de referinta

```
#include <iostream>
using namespace std;

char &replace(int i); // return a reference

char s[80] = "Hello There";

int main()
{
   replace(5) = 'X';  // assign X to space after Hello
   cout << s;
   return 0;
}

char &replace(int i)
{
   return s[i];
}
```

-putem face atribuiri catre apel de functie

-replace(5) este un element din s care se schimba

-e nevoie de atentie ca obiectul referit sa nu iasa din scopul de vizibilitate

Referinte independente

-nu e asociat cu apelurile de functii

-se creeaza un alt nume pentru un obiect

-referintele independente trebuiesc initializate la definire pentru ca ele nu se schimba in timpul programului

```
#include <iostream>
using namespace std;

int main()
```

```
{
  int a;
  int &ref = a; // independent reference
  a = 10;
  cout << a << " " << ref << "\n";
  ref = 100;
  cout << a << " " << ref << "\n";
  int b = 19;
  ref = b; // this puts b's value into a
  cout << a << " " << ref << "\n";
  ref--; // this decrements a
  // it does not affect what ref refers to
  cout << a << " " << ref << "\n";
  return 0;
}
```

```
10 10
100 100
19 19
18 18
```

Referinte catre clase derivate

-putem avea referinte definte catre clasa de baza si apelata functia cu un obiect din clasa derivata

-exact ca la pointeri