# ELEMENTE FUNDAMENTALE ALE POO

Conceptele fundamentale ale POO sunt noţiunile de *clasă* şi *obiect* precum şi relaţiile dintre clase, obiecte şi dintre clase şi obiecte.

Obiectul este o entitate dinamică, caracterizată prin: -stare

-comportament

-identitate

*Starea* unui obiect semnifică temporalitatea lui. El poate fi creat şi distrus în timp de alt obiect sau de utilizator. Starea obiectului este caracterizată de atribute care au

–nume asociat

-valori asociate

*Comportarea* unui obiect exprimă metoda (modalitatea) de răspuns a sa la mesaje (acţiuni) provenite din exterior, care determină schimbarea stării sale.

*Identitatea* este proprietatea care distinge un obiect de alte obiecte. Ea se păstrează şi după ce obiectul a fost supus unor transformări sau şi-a modificat starea iniţială.

Obiectele cu stari şi comportari asemanatoare se grupează în *clase*.

O *clasă* abstractizează proprietăţile comune ale unui set de obiecte, asfel ca fiecare obiect este o **instanţă** (reprezentant sau individ) a unei clase. O clasă poate avea nici unul, unul sau mai multe obiecte instanţă. Clasa este o entitate *statică* şi nu dinamică precum obiectul, deoarece după specificare şi implementare se modifică foarte greu.

*Relaţiile* stabilesc conexiunile logice dintre diferite entităţi.

Relaţiile obiect-obiect sunt

de utilizare-între obiecte ce schimba mesaje între ele

de conţinere-obiecte complexe care conţin alte obiecte

## Relatia obiect –clasă este cea de instanţiere.

Relaţiile clasă –clasă pot fi mostenite

de asociere în utilizari

de conţinere (agregare)

Paradigmele POO sunt: abstractizarea

incapsularea

moştenirea

polimorfismul

*Abstractizarea* reprezintă procesul de ignorare intentionata a detaliilor nesemnificative şi retinerea proprietăţilor definitorii ale unei entităţi. Ea poate fi: -procedurală

-a datelor

*Abstractizarea procedurală* ignoră detaliile privitoare la desfasurare proceselor. Este specifică programarii procedurale în care datele de I/E şi funcţiile sunt distincte şi folosite separat (exemplu C++ permite programare procedurală).

*Abstractizarea datelor* ignoră detalii privind reprezentarea unui tip de date . În POO pentru entităţi de nivel inalt cele 2 abstractizari sunt combinate în cadrul claselor.

*Incapsularea* (ascunderea informaţiilor) implementează abstracţiile prin clase. Ea separă informaţiile de manipulare a unei entităţi de informaţiile de implementare.

Incapsularea informaţiei la nivel de tip se face prin împărţirea unei clase în 2 secţiuni distincte: interfaţă şi implementare.

*Interfaţa* *unei clase*, transmisa tuturor obiectelor instanţă ale clasei respective, permite accesul la clasa din exterior al potenţialilor utilizatori.

*Implementarea clasei,* care cuprinde reprezentarea abstracţiei şi algoritmii metodelor de interfaţă, nu este cunoscută şi accesibilă utilizatorilor.

*Moştenirea* permite clasificarea tipurilor prin organizarea lor în *ierarhii*. În funcţie de poziţia ocupata în ierarhie, un tip poate fi: -generalizarea altor tipuri , sau

-specificarea altor tipuri.

Astfel, o clasă este *clasă de bază* sau *superclasă* sau *supertip* pentru toate clasele care o specializează şi *clasă derivată* sau *subclasă* sau *subtip* pentru toate clasele pe care le specializează. O clasă derivată preia de la clasa de bază toate caracteristicile acesteia, la care se mai pot adăuga şi altele noi.

*Polimorfismul* permite unei valori de a avea mai multe tipuri. În C + + polimorfismul este:

* parametric
* ad-hoc
* de moştenire

Polimorfismul parametric apare când aplicam o funcţie pe argumente de tipuri diferite.

Polimorfismul ad-hoc permite supraîncarcarea funcţiilor (implementării diferite), redefinirea metodelor unei clase de bază prin redefinirea şi supraîncarcarea lor în clase derivate.

Polimorfismul de moştenire permite manipularea obiectelor de un anumit tip în situaţii ce necesită tipuri diferite de cele ale obiectelor.

# Declararea claselor

*Clasa* este un tip de date abstracte definit de utilizator, care se comportă ca un tip predefinit. O clasă C + + incapsulează datele membre, ce reprezintă atributele clasei precum şi funcţii membre (sau metode) care efectuează operaţii asupra datelor clasei.

Clasa are un *nume* ce poate fi orice identificator, unic în domeniul respectiv de existenţă. Declararea unei clase foloseşte cuvântul cheie *class* şi are sintaxa:

class nume\_clasa

{

[private:

<date membre private>

<constructori privaţi>

<funcţii membre private>]

[protected:

<date membre protejate>

<constructori protejaţi>

<funcţii membre protejate>]

[public:

<date membre publice>

<constructori publici>

<destructor public>

<funcţii membre publice>]

};

C + + asigură o limitare a accesului la membrii unei clase şi de aceea în declararea unei clase folosim *specificatori de control ai accesului*: private, protected şi public.

*Specificatorul private* precizează membrii proprii ai clasei. Ei pot fi accesaţi de orice funcţie, membră a clasei. Instanţele clasei nu au acces la membrii privaţi.

*Specificatorul protected* precizează membrii protejaţi care pot fi accesaţi nu numai de funcţii membre ale clasei, ci şi de funcţii membre ale claselor derivate din clasa respectivă.

*Specificatorul public* precizează membrii publici care pot fi accesaţi de orice funcţie din domeniul declaraţiei clasei (funcţii membre şi instanţe).

Implicit membrii unei clase au atributul de acces privat. Dacă sunt prezenţi, specificatorii de acces pot fi în orice ordine şi se pot chiar repeta.

O funcţie membru a unei clase are acces la toate datele membre ale oricărui obiect din clasa respectivă, indiferent de specificatorul de acces. Se recomandă plasarea datelor membre în secţiunea protejată pentru a facilita accesul la ele de catre toate funcţiile membre ale claselor derivate. În declaraţia unei clase pot apărea numai prototipurile funcţiilor membre. Definirea lor poate fi inclusă în acelaşi fişier sau în fişiere separate în cadrul bibliotecilor de programe. Funcţiile definite în cadrul clasei fac implicit parte din categoria inline.

Pentru a defini o funcţie membru aflată în afara declaraţiei clasei folosim operatorul de rezoluţie :: conform sintaxei:

nume\_clasa :: nume\_funcţie

unde nume\_clasa este numele clasei din care face parte funcţia.

Operatorul de rezoluţie :: spune ca funcţia are acelaşi domeniu cu declararea clasei şi este membră a clasei.

După declararea unei clase, numele ei poate fi folosit ca identificator de tip pentru a declara instanţe ale clasei, cu o sintaxă asemanatoare cu cea de la declararea variabilelor.

**Exemplu:** Clasa coordonate incapsulează 2 date membre x,y şi 2 funcţii membre(translatare şi tipărire).Cele 2 date membre sunt coordonatele unui punct din plan. Funcţia t*ranslatare* ( ) translatează coordonatele punctului considerat cu a respectiv b, iar funcţia *tipărire*() afişează coordonatele punctului.

# include<iostream.h>

class coordonate

{

private:

float x,y;

public :

void translatare (float a, float b);

void tipărire ();

};

void coordonate :: traslatare(float a, float b)

{

x + = a;

y + = b;

}

void coordonate :: tipărire ()

{

cout << “Coordonatele punctului sunt: x = “ << x << ” şi y = “ << y << endl ;

{

class coordonate M , N ; aici se declară două obiecte de tip coordonate.

Definirea funcţiilor membre translatare () şi tipărire () este făcută în afara declararii clasei *coordonate* căreia îi aparţin.

Putem opera cu obiectele dintr-o clasă ca şi cu orice tipuri de date predefinite.

**Exemplu:**

coordonate pct1, pct2; declară 2 obiecte

coordonate \*p; declară un pointer la obiect

coordonate şir [10] ; declară un şir de 10 obiecte

pct1 = pct2 ;

\*p =şir ; atribuiri

Putem accesa orice membru public prin operatorul “**.**”. Membrii privaţi nu pot fi accesaţi decât de funcţii membre ale clasei. Putem atribui o valoare unei date membre private numai printr-o funcţie membră a clasei, care poate testa validitatea respectivei valori.

**Exemplu :** Program ce declară clasa *angajat* care include2 funcţii membre publice şi 4 date membre private. Funcţia *valori ()* iniţializează datele private ale clasei şi foloseşte instrucţiunea *if* pentru a asigura lui *salariu* o valoare corectă. Funcţia de *afisare ()* afişează datele membre private. Funcţiile sunt accesate prin operatorul ”**.**’’.

# include <iostream.h>

#include <conio.h>

#include <string.h>

class angajat

{

public:

int valori (char\*, int, long, float);

void afisare (void) ;

private :

char nume [50] ;

int varsta ;

long marca ;

float salariu ;

} ;

int angajat :: valori (char\*ang\_nume, int ang\_varsta, long ang\_marca, float ang\_salariu)

{

strcpy(nume,ang\_nume);

varsta = ang\_varsta;

marca = ang\_marca;

if ( ang\_salariu < 3000000)

{salariu =ang\_salariu;

return (0);

}

else return (1);

}

void angajat :: afisare (void)

{

cout <<”Nume:” <<nume<<endl;

cout << ‘’Varsta : ‘’<<varsta<<endl ;

cout << ‘’Marca :’’<<marca<<endl ;

cout <<’’Salariu :’’<<salariu<<endl ;

}

void main (void)

{

clrsc() ;

angajat lucrator ;

if (lucrator.valori(“Popescu ‘’,55,1313,2000000) = = 0)

{

cout <<’’Date angajat :’’<<endl ;

lucrator.afisare () ;

}else cout <<’’Valoarea salariului este incorecta’’ ;

}

**Observaţia 1 :** După compilare şi rulare apare pe ecran :

Date angajat :

Nume :Popescu

Varsta :55

Marca :1313

Salariu :2000000

**Observaţia 2 :**Este posibil ca anumite clase să conţină funcţii membre private. Ele nu pot fi apelate de program prin operatorul ‘’**.**’’ , ci doar de alţi membri ai claselor de care aparţin.

**Funcţii membre inline**

Definirea unei funcţii membru în interiorul unei clase este considerată inline.

Exemplu :Aici se declară clasa *coordonate* de mai sus în care funcţia *translatare* nu este doar declarată ci şi definită ; ea devine implicit funcţie *inline.* Ca urmare, la fiecare apel, ea nu este doar apelată ci expandată. Definirea unei funcţii într-o clasă se numeşte şi *declarare inline implicită*. Funcţia membru *tipărire* este doar declarată în cadrul clasei. Definirea ei în afara declaraţiei clasei este prefixată de cuvântul cheie *inline*.Acest fapt este echivalent cu definirea funcţiei în cadrul declaraţiei clasei al cărei membru este. Acestea este o *declaraţie inline explicită.*

# include **<**iostream.h>

class coordonate

{

float x,y;

void translatare (float a, float b)

{

x + = a;

y + = b;

}

void tipărire () ;

} ;

inline void coordonate :: tipărire ()

{

cout <<’’ Coordonatele punctului sunt : x = “<<x<< “  şi y = “<<y<< endl ;

**}**

**Observaţie** : Este interzisă folosirea în funcţiile inline a instrucţiunilor repetitive for

while

do-while

Deoarece compilatorul are nevoie de întreaga definiţie a unei funcţii inline, funcţia nu poate fi folosită decât în fişierul în care a fost definită.

**Constructori**

*Constructorul* este o funcţie membru specială a unei clase, care are acelaşi nume cu clasa căreia îi aparţine. El serveşte la crearea şi iniţializarea obiectelor şi de aceea este apelat implicit de fiecare dată când se crează noi instanţieri ale clasei.

**Declarare şi caracteristici**

Într-o clasă pot exista nici unul, unul sau mai multi constructori. Dacă nu este nici un constructor, compilatorul generează automat unul care este public, fără argumente şi cu o listă vidă de instrucţiuni.

O funcţie constructor nu întoarce un tip de date, deci în corpul ei nu trebuie folosit *return.* Se declară un constructor prin :

class nume\_clasa

{

public:

nume\_clasa (); constructor prestabilit fără argumente

nume\_clasa (nume\_clasa & c); constructor de copiere

nume\_clasa ( argumente); constructorii cu argumente

} ;

La declararea şi la definirea unei funcţii constructor tipul void nu poate fi specificat ca tip returnat.

Constructorii pot avea argumente (inclusiv din cei impliciţi) şi pot fi supradefiniţi. Pointeri catre constructori nu se pot folosi, deci adresa lor nu ne este accesibiă.

Într-un constructor putem folosi operatorii new şi delete.

Un constructor prestabilit nu are argumente sau are o listă de argumente în care toţi argumente folosesc argumente prestabilite.

# Ex.1: clasa coordonate

{

protected:

float x,y;

public:

coordonate( );

coordonate(coordonate&c);

coordonate(float a, float b);

........ alte funcţii......

};

Ex.2: Program care creează clasa  **angajat**  ce conţine funcţia constructor angajat. Acesta se defineşte ca orice funcţie membră a clasei şi atribuie valori iniţiale la trei obiecte lucr1, lucr2 şi lucr3 fără a întoarce vreo valoare.

# include <iostream.h>

# include <conio.h>

# include <string.h>

class angajat

{

public: *declararea funcţiei constructor „angajat”*

angajat(char\*, int, long, float);

void afisare (void);

private :

char nume [50];

int varsta;

long marca;

float salariu;

}; definirea funcţiei constructor „angajat”

angajat :: angajat(char\*nume, int varsta,

long marca, float salariu);

{

strcpy (angajat::nume,nume);

angajat :: varsta = varsta;

angajat :: marca = marca;

if (salariu <3000000.0)

angajat :: salariu =salariu;

else

angajat :: salariu = 0.0;

}

*definirea funcţiei membru „afişare”*

void angajat :: afisare (void)

{

cout << endl;

cout << “Nume:” << nume <<endl;

cout << “Varsta:” << varsta <<endl;

cout << “Marca” << marca <<endl;

cout << “Salariu:” <<salariu <<endl;

}

void main (void )

{

clrscr( );

angajat lucr1 (“M. Pop”,50, 1313, 2000000.0);

angajat lucr2 (“A. Pop”,17, 1713, 3000000.0);

angajat lucr3 (“Marian P.”, 43, 1317, 3200000.0);

cout <<“Date angajati:” << endl;

lucr1.afisare( );

lucr2.afisare( );

lucr3.afisare( );

}

După compilare şi rulare pe ecran se vor afişa datele: Date angajati:

Nume: M.Pop

Varsta: 50

Marca: 1313

Salariu: 2000000.0

Nume: A. Pop

Varsta: 17

Marca:1713

Salariu: 3000000.0

Nume : Marian P.

Varsta: 43

Marca 1317

Salariu 3200000.0

Dacă nu precizăm valori pentru toate argumentele, funcţia constructor, ca orice funcţie, foloseşte valori implicite.

## Supradefinirea constructorilor

O funcţie constructor se supradefineşte ca orice altă funcţie prin specificarea unor funcţii alternative care corespund unor tipuri diferite de argumente.

Spre exemplu în programul de mai sus putem da o a doua definire a funcţiei constructor **angajat** înaintea definirii funcţiei membră **afişare** care solicită utilizatorului să introducă o valoare a argumentului salariu dacă aceasta nu este precizată în program, în timp ce prima definire implică specificarea în program a valorilor celor 4 argumente: nume, varsta, marca, salariu.

A doua definire are forma:

angajat :: angajat (char\* nume, int varsta, long marca)

{

strcpy (angajat :: nume,nume)

angajat :: varsta = varsta;

angajat :: marca=marca;

do{

cout <<”Introdu un salariu pt.” << nume << “mai mic decât 3000000.0”;

cin >> angajat :: salariu;

}

while (salariu>= 3000000.0)

}

În programul principal totul este identic ca mai sus, afară de lucr3, unde instrucţiunea este:

angajat lucr3 (“Marian P.”, 43, 1317);

La compilare şi rulare se cere introducerea unei valori pentru salariul lucrătorului 3 după care execuţia continua cu afişarea datelor pentru cei trei angajaţi.

Constructor de copiere

Acest constructor se foloseşte în C + + pentru a declara un obiect de tip clasă şi pentru a transfera un obiect ca argument sau ca rezultat al unei funcţii însoţite de o atribuire respectiv de iniţializarea obiectului cu datele altui obiect de acelaşi tip.

Dacă un astfel de constructor de copiere nu este declarat într-o anumită clasă, compilatorul generează automat unul care iniţializează datele unui nou obiect cu valorile corespunzătoare dintr-un obiect specificat prin copiere membru cu membru ca la atribuirea standard.

Pentru a declara un constructor de copiere pentru clasa nume\_clasa trebuie să specificăm un argument unic de tip referinţă la un obiect de acelaşi tip nume\_clasa sub forma:

nume\_clasa (const nume\_clasa & nume\_obiect);

unde modificatorul const asigură că obiectul de intrare nu poate fi modificat.

Constructorul de copiere acţionează ca orice apel de funcţie obişnuită.

Exemplu: Aici clasa dublet conţine declaraţiile unui constructor prestabilit şi a unuia de copiere care este apelat de trei ori:

Un constructor de copiere poate promova mai multe argumente, dar cele suplimentare trebuie să fie cu valoare implicită pentru a permite copierea automata a obiectelor la apelul funcţiei sau la întoarcerea din funcţie.

Constructorul de copiere generat implicit de compilator copiază doar datele membre în cazul unor structuri dinamice şi de aceea se recomandă declararea de utilizator a constructorului de copiere pentru clasele care se referă la acest tip de structuri de date.

###### Destructori

Destructorii sunt funcţii speciale ale unei clase cu efect complementar constructorului. Ele sunt automat aplelate pentru distrugerea obiectelor unei clase la sfârşitul programului.

Destructorii au acelaşi nume cu cel al clasei de care aparţin, precedati însă de caracterul ”~”. Fiecare clasă are cel mult un destructor. Dacă acesta lipseşte, compilatorul generează automat unul cu efect nul.

Funcţiile destructor nu primesc argumente. Se declară sub forma:

class nume\_clasa

{

public:

nume\_clasa( ) *constructor prestabilit*

. . . . . . . . . . . . . . . . *alţi constructori*

~ nume\_clasa( ) *destructor*

. . . . . . . . . .. . . . . . *alte funcţii membre*

};

Funcţia destructor nu întoarce nici un tip de date, nici chiar tipul void. Destructorii sunt utili mai ales pentru obiectele create prin alocarea dinamică a memoriei.

Exemplu : Un program ce are clasa **angajat** în care este definit destructorul ~angajat.

|  |  |
| --- | --- |
| # include <iostream.h>  # include <conio.h>  # include <string.h>  class angajat  {  public: *declararea funcţiei constructor*  angajat(char\*, int, long, float)  ~angajat( ); *declararea funcţiei destructor*  void afisare (void); *fct. membru*  private:  char nume [50];  int varsta;  long marca;  float salariu; *definireafuncţieiconstructor*  angajat :: angajat (char\* nume, int varsta,  long marca, float salariu)  {  strcpy(angajat :: nume,nume);  angajat :: varsta = varsta;  angajat :: marca = marca;  angajat :: salariu =salariu;  } | angajat :: ~angajat( ) *definire funcţiei*  { *destructor*  cout<<”Se distruge obiectul pentru angajatul”  <<nume<<endl;  }  void angajat :: afisare(void) *definirea funcţiei*  { *membru*  cout<<”Nume:”<<nume<<endl;  cout<<”Varsta:”<<varsta<<endl;  cout<<”Marca:”<<marca<<endl;  cout<<”Salariu”<<salariu<<endl;  }  void main ( )  {  clrscr( );  angajat lucr(”B.Pop”,90,2051,2000000.0);  cout<<”Date angajat:”<<endl;  lucr.afisare( );  } |

După compilare şi rulare pe ecran apare:

Date angajat:

Nume: B. Pop

Varsta: 90

Marca: 2051

Salariu: 2000000.0

Se distruge obiectul pentru angajatul B.Pop

###### Membri statici ai unei clase

Datele şi funcţiile unei clase declarate cu cuvântul cheie static se numesc membri statici.

### Date membre statice

Fiecare obiect dintr-o clasă are o mulţime de date membre. Cele nestatice există în copii distincte în fiecare obiect ce le conţine. Cele statice există într-o singura copie comună tuturor obiectelor ce le conţin, deci unei date membre statice i se rezervă o singură zonă de memorie comună tuturor instanţelor clasei. Acestea se declară în cadrul clasei sub forma:

private ;

public;

static tip nume\_data;

După declararea clasei orice dată membru static trebuie declarată ca variabilă globală în exteriorul clasei sub forma:

**tip\_data nume\_clasa::nume\_data\_membru**;

Obligatoriu datele membre statice trebuie iniţializate în afara declaraţiei clasei.

Datele membre statice există separat de obiectele clasei, dar orice modificare a unei date membre este sesizată de toate obiectele clasei ce o conţin.

Datele membere statice pot fi accesate înainte de crearea obiectelor clasei. Accesarea lor din interiorul funcţiilor membre ale clasei respective se face la fel ca şi pentru datele membre nestatice.

Exemplu : Program ce are clasa stoc în care membrul pret este declarat static. El este acelaşi pentru cele două obiecte ale clasei prim şi secund. Modificarea valorii membrului pret afectează instanţele celor două obiecte. Valorile celorlalte date membre ale obiectelor nu se modifică.

# include <iostream.h>

# include <string.h>

# include <conio.h>

class stoc

{

private : *declararea membrului*

static int pret; *static pret*

char denumire[50];

char producator[50];

int cantitate;

public:

stoc(char\*, char\*, int);

void afisare ( );

void val (int);

}; *declararea membrului static pret ca*

int stoc :: pret;  *variabilă globală*

void stoc :: val (int prt)

{

pret = prt;  *iniţializarea membrului static*

}

stoc :: stoc (char\* den, char\* prod,

int cant);

{

strcpy (stoc :: den, denumire);

strcpy (stoc :: prod, producator);

stoc :: cant = cantitate;

}

void stoc :: afisare ( )

{

cout<<”Denumire:”<<denumire<<endl;

cout<<”Producator:”<<producator<<endl;

cout<<”Cantitate:”<<cantitate<<”buc”<<endl;

cout<<”Pret unitar:”<<pret<<”u.m.”<<endl;

}

void main ( )

{

clrscr ( );

stoc prim (“Produs1”, “Firma A”, 200);

stoc secund (“Produs2”, “Firma B”, 300);

prim.val(100);

cout<<”Oferta iniţială:”<<endl;

prim.afisare( );

secund.afisare( );

cout<<endl<<”Oferta finala:”<<endl;

prim.val(90);

prim.afisare( );

secund.afisare( );

}

După rulare pe ecran apar afişate: Oferta iniţială:

Denumire: Produs1

Producator: Firma A

Cantitate: 200 buc.

Pret unitar: 100 u.m.

Denumire: Produs2

Producator: Firma B

Cantitate: 300 buc.

Pret unitar: 100 u.m.

Oferta finala:

Denumire: Produs1

Producator: Firma A

Cantitate: 200 buc.

Pret unitar: 90 u.m.

Denumire: Produs2

Producator: Firma B

Cantitate: 300 buc.

Pret unitar: 90 u.m.

Orice dată membru declarată ca publică şi statică se poate folosi în program chiar dacă în program nu există nici un obiect al clasei.

Pentru accesarea unui astfel de membru se foloseşte operatorul de rezoluţie globală sub forma: **nume\_clasa :: nume\_membru**

Ex:

# include<iostream.h>

# include<conio.h>

class stoc

{

public:

static int pret;

private:

char denumire [50];

char producator [50];

int cantitate;

};

int stoc :: pret; *declararea membrului stoc pret ca*

void main ( ) *variabilă globală*

{

clrscr ( );

stoc :: pret = 100;

cout<<”Pretul unitar iniţial a fost:”<<stoc :: pret <<

”u.m.”<<endl;

stoc :: pret = 90;

cout<<“Pret unitar final este:”<<stoc :: pret<<”u.m”;

}

Aici se accesează membrul pret al clasei **stoc** fără să existe un obiect al clasei stoc.

Observaţie: Prin folosirea datelor membre statice:

* reducem numarul variabilelor globale din program;
* urmărim numarul instanţelor unei clase;
* controlăm accesul;
* alocăm un bloc de memorie special pentru diferite obiecte ale unei clase.

#### Funcţii membre statice

Putem defini funcţii membre statice ca şi date membre statice. Ele fac operaţii care nu sunt asociate unor obiecte individuale, ci asociate întregii clase care le conţin.

O funcţie membru statică şi publică poate fi apelată şi când nu există nici un obiect al clasei respective.

Pentru apelare folosim operatorul de rezoluţie globală “::”.

Aceste funcţii nu primesc implicit adresa unui obiect.

Spre deosebire de funcţiile membre nestatice ale unei clase, cele statice nu au pointerul this, deci ele nu pot accesa direct decât alte elemente statice ale clasei respective (nu pot accesa elementele nestatice ale clasei!).

Se definesc ca funcţii membre statice într-o clasă acele funcţii de uz general care sunt folosite de mai multe clase, evitând definirea lor ca funcţii de sine statatoare.

|  |  |
| --- | --- |
| Ex.1:  class clasa  {  private:  static int i;  . . . . . . . . . .  public: *declararea funcţiei membre statice*  static void f(int);  . . . . . . . . . .  }; *definirea datei membre statice*  int clasa :: i=5;  void clasa :: f(int i)  { *definirea funcţiei membre statice*  ii; *accesarea directă a unui membru static*  . . . . . . . . . . .  } | Ex.2:  class cls  {  public:  int vect [15];  static int prim ( );  };  int cls :: prim ( )  {  return vect [0];  }  În acest caz compilatorul va semnala eroare la încercarea funcţiei membră statică prim de a accesa membrul nestatic vect. |

###### Funcţii prietene

C + + permite accesul la datele membre private ale unei clase nu numai funcţiilor membre ale clasei respective, ci şi altor funcţii membre clasei numite funcţii prietene. Ele pot fi independente sau membre altei clase.

Declararea unei funcţii prietene independente se face în interiorul clasei precedând prototipul funcţiei de cuvântul cheie friend astfel:

class nume\_clasa

{

public:

friend tip\_rez nume\_fct (<argumente>);

Indiferent de poziţia prototipului, funcţiile prietene sunt publice . O astfel de funcţie nu poate întoarce o referinţă la clasa prietenă.

O funcţie prietenă poate accesa membri privaţi ai clasei direct prin operatorul “ .”.

Ex.: Programul declară calsa comp şi în ea funcţia prietenă independentă: coincid(), care întoarce o valoare nenulă dacă punctele primite ca argumente sunt identice şi nulă în caz contrar.

|  |  |
| --- | --- |
| # include <iostream.h>  # include <conio.h>  class comp  {  int x,y;  public: *declararea constructorlui*  comp (int,int);  ~ comp ( );  *declararea destructorului*  void afisare ( ); *decl. funcţiei membru*  friend int coincid (comp&, comp&);  }; *declararea funcţiei prietene independente*  comp::comp(int a,int b)  {x=a; y=b;afisare( ); *definirea*  ……. *constructorului*  }  comp::~comp( ) *definirea destructorului*  {……} | void comp::afisare ( ) *definirea funcţiei*  { *membru*  cout <<”x=”<<x<<”;y=”<<y<<endl;  } *definirea funcţiei prietene*  int coincid (comp & p1,comp & p2)  {  if (p1.x == p2.x && p1.y == p2.y)  return 1;  else return 0;  }  void main ( )  {  clrscr ( );  cout <<”coordonatele punctului p1 sunt:”;  comp p1(2,3);  cout <<”coordonatele punctului p2 sunt:”;  comp p2(2,4);  cout <<”rezultat:”<<endl;  if (coincid(p1,p2))  cout <<”Punctele p1 şi p2 coincid:”<<endl;  else cout<<”Punctele p1 şi p2 sunt diferite”;  } |

După rulare pe ecarn apare:

coordonatele punctului p1 sunt: x=2; y=3

coordonatele punctului p2 sunt: x=2; y=4

Rezultat:

Punctele p1 şi p2 sunt diferite.

O funcţie prietenă poate opera asupra unui obiect transferat ca argument. Ea foloseşte numele obiectului şi un operator de rezoluţie pentru a referi date sau funcţii membre ale clasei.

Dacă un program conţine o funcţie membră a unei clase care este prietenă altei clase se impun:

* declararea corespunzătoare a funcţiilor prietene
* o anumită ordine în definiţia claselor.

Declararea funcţiilor prietene ce aparţin unei clase se face sub forma:

class nume\_cls2

{

public:

. . . . . . . . .

friend nume\_cls1 :: nume\_fct (<argumente>);

}

unde nume\_cls1= numele clasei în care este funcţia

nume\_cls2 = numele clasei cu care funcţia este prietenă.

Definiţiila celor două clase trebuie să respecte o ordine prestabilită. Pentru aceasta trebuie specificat un identificator nume\_cls1 sub forma: class nume­\_cls1. Se defineşte apoi clasa nume\_cls2. Între acestea se dă compilatorului indicaţia privind o revenire ulterioară la definirea clasei nume\_clas1. Altfel definiţia clasei nume\_cls2 ar face referire la clasa nume\_cls1, nerecunoscută pentru compilator şi aceasta ar da eroare.

Exemplu: Aici funcţia f( )clasei A şi este prietenă clasei B

class A; *definirea incompleta a clasei A*

class B *definirea clasei B*

{

. . . . . . . . . .

void f(A, . . . . )

. . . . . . . . . . .

};

class A

{ *definirea clasei A*

. . . . . . . . . .

friend void B :: f(A, . . . . );

};

void B :: f(A obiect-A, . . . . )

{

. . . . . . . . . . . *definirea funcţiei prietene*

}

Folosirea funcţiilor prietene:

* reprezintă o metodă de control a accesului la membrii unei clase
* oferă o metodă simpla de acces limitat din anumite funcţii la datele unei clase
* funcţia care în lista de argumente are cel puţin două argumente din aceeaşi clasă şi care accesează argumente privaţi nu poate fi decât prietenă.

##### Clase prietene

Clasa B este prietenă clasei A dacă toate funcţiile membre clasei B pot accesa direct membri privaţi ai clasei A.

Pentru a defini clasa B prietenă clasei A se introduce între membri publici ai clasei A o instrucţiune care să conţină cuvântul **friend** urmat de numele clasei B sub forma:

Relatia **friend** nu este nici reflexivă, nici tranzitivă.

De regulă clasele prietene în C + + sunt independente, în sensul ca nici o clasă nu mosteneşte membrii celeilalte clase.

class B;

class A

{

public:

. . . . . . . . . .

friend class B;

. . . . . . . . . .

};

class B

{. . . . . . . . };

Exemplu: Aici clasa specificare este prietenă clasei material. Programul modifică una din caracteristicile unui material şi anume funcţie modif( ) a clasei specificare modifică data membră cant care reprezintă cantitatea disponibila a materialului. Obiectul material este transmis funcţiei modif( ) prin adresă. Pentru a accesa membrul respectiv funcţia foloseşte un pointer.

# include <iostream.h>

# include <string.h>

# include <conio.h>

class specificare;

class material

{

public:

material(char\*, char\*, char\*);

void afisare(void);

friend specificare;

private:

char den [30];

char prod [30];

char cant [20];

};

material :: (char\* den, char\* prod, char\*cant)

{

strcpy(material :: den, den);

strcpy(material :: prod, prod);

strcpy(material :: cant, cant);

}

void material :: afisare(void)

{

cout<<”Denumire:”<<den<<endl;

cout<<”Producator:”<<prod<<endl;

cout<<”Cantitate:”<<cant<<endl;

}

class specificare

{

public:

void modif(material\*, char\*);

char\* atrib(material);

};

void specificare ::modif(material\* material1, char\* cant1)

{

strcpy(material1 cant, cant1);

}

char\* specificare :: atrib(material material1)

{

static char cant [20];

strcpy (cant, material1.cant);

return (cant);

}

void main ( )

{

clrscr( );

cout<<”Situaţia iniţială:”<<endl;

material car(“Inox”,”Cos Pitesti”,”5t”);

specificare actual;

car.afisare ( );

actual.modif(&car,”3t”);

cout<<endl<<”Situaţia actuala:”<<endl;

car. afisare ( );

}

După compilare şi rulare pe ecran apare:

Situaţia iniţială:

Denumire: Inox

Producator: Cos Pitesti

Cantitate: 5t

Situaţia actuala:

Denumire: Inox

Producator: Cos Pitesti

Cantitate: 3t

###### Supradefinirea operatorilor

Definirea unei clase într-un program reprezintă definirea unui nou tip de date. Folosind operatori standard putem defini diferite operaţii cu obiectele clasei. Aceşti operatori se obţin prin supradefinirea celor existenţi adica prin modificarea semnificaţiei lor.

Rolul supradefinirii operatorilor este creşterea lizibilităţii programului exprimând operaţii cu obiectele claselor definite de utilizator sub o formă cat mai sugestivă.

Supradefinirea unui operator într-o clasă se face prin specificarea unei funcţii care implementează operaţia corespunzătoare operatorului respectiv. Această funcţie este ulterior apelată atunci când clasa foloseşte operatorul supradefinit.

Declararea funcţiei foloseşte cuvântul cheie operator sub forma:

tip\_rez operator simbol\_operator (<argumente>);

La supradefinirea operatorilor trebuie respectate regulile:

* simbol\_operator = orice simbol de operator din C + + mai puţin următorii patru:

. (operator de membru al clasei)

.\* (operator de pointer la membru)

:: (operator de rezoluţie globală)

?: (operator de expresie condiţională)

* nu se poate extinde setul de operatori C + + prin desemnarea de noi simboluri (caractere sau seturi de caractere).
* aritatea unui operator nu se poate modifica.
* prioritatea şi asociativitatea operatorilor supradefiniţi se păstrează; se poate modifica doar prin folosirea parantezelor.

La supradefinirea unui operator pentru o clasă, semnificaţia lui este modificată doar în clasa respectivă, pentru restul variabilelor din program operatorul păstrându-şi semnificaţia curentă.

Funcţiile operator care supradefinesc operatorii unei clase se definesc:

* sau ca funcţii prietene
* sau ca funcţii membre ale clasei respective.

#### Funcţii operator prietene

Prototipul funcţiei operator prietene apare în cadrul declaraţiei clasei fiind precedat de cuvântul cheie friend. Această funcţie are acces la toate datele private ale clasei.

Exemplu: Aici implementăm operatorul unar minus, operatorul binar plus şi înmulţirea cu un scalar, ca funcţii prietene clasei coord, care se referă la coordonatele punctelor din plan.

Prototipurile celor trei funcţii operator sunt incluse în declaraţia clasei şi definiţiile lor sunt plasate în afara clasei.

# include <iostream.h>

# include <conio.h>

class coord

{

private:

double x; double y;

public:  *constructor*

coord (double x0=0, double y0=0)

{x=x0; y=y0;}

void afisare ( ) *fct. membru*

{

cout <<”x=”<<x;

cout <<”, y=”<<y<<endl;

} *declararea funcţieiilor operator*

friend coord operator – (coord);

friend coord operator + (coord, coord);

friend coord operator \* (int, coord);

};

coord operator - (coord a)

{ *def. fct.*

coord c; *operator minus*

c.x = -a.x;

c.x = -a.y;

return c;

}

coord operator + (coord a, coord b)

{ *def. fct.*

coord c; *operator plus*

c.x = a.x + b.x;

c.y = a.y + b.y;

return c;

}

coord operator \* (int k, coord a)

{ *def. fct. operator*

coord c;  *înmulţire cu scalar*

c.x = k \* a.x ;

c.y = k \* a.y ;

return c;

}

void main ( )

{

clrscr ( );

cout <<”Date de intrare:”<<endl;

coord p1(1,1); p1.afisare( );

coord p2(2,4); p2.afisare( );

coord p3(3,0); p3.afisare( );

coord r1, r2, r3, r4;

cout <<”Rezultate:”<<endl;

r1 =- p1;

r1.afisare ( );

r2 = p1 + p2;

r2.afisare ( );

r3 = p1 + p3 +r2;

r3.afisare ( );

r4 = - 5 \* p1;

r4.afisare ( );

}

După compilare şi rulare pe ecran apar:

Date de intrare:

x = 1, y = 1

x = 2, y = 4

x = 3, y = 0

Rezultate:

x = - 1, y = - 1

x = 3, y = 5

x = 7, y = 6

x = -5, y = - 5

###### Funcţii operator membre

O funcţie operator membră unei clase are unul din operanzi this iar ceilalţi operanzi, dacă există sunt transmişi ca argumente. Astfel implicit funcţia primeşte adresa obiectului pentru care este apelată.

Exemplu: Reluăm exemplul de mai sus pentru implementarea celor trei operatori ca funcţii membre ale clasei **coord**.

* Se înlocuiesc cele trei instrucţiuni **friend** care declară funcţiile operator prietene respective cu urmatoarele instrucţiuni care declară funcţiile operator membre:

coord operator – ( );

coord operator + (coord);

coord operator \* (int);

În definiţiile celor trei funcţii se înlocuiesc instrucţiunile subliniate mai sus respectiv cu câte una din urmatoarele:

coord coord :: operator – ( )

coord coord :: operator + (coord b)

coord coord :: operator \* (int k)

* În rest totul este nemodificat, excepţie referirea la primul argument al funcţiilor care acum se face prin referirea directă a acestora şi astfel programul va afişa aceleaşi rezultate ca şi cel de mai sus.

Directive de preprocesare

Preprocesarea este prima etapă a compilarii unui fişier sursă şi este realizată de un program special numit preprocesor care este diferit de compilator. El este automat apelat de compilator pentru a executa anumite comenzi ce există în fişierul sursă, comenzi numite directive de preprocesare şi precedate de #.

## Directiva #include

Inserează în programul sursă curent conţinutul fişierului indicat de directivă. Cu aceasta putem modulariza scrierea programelor. Această directivă are 3 sintaxe:

# include <fişier> *fişierul de aici se numeşte fişier antet*

# include “fişier” *acesta se foloseşte când fişierul antet se afla în directorul curent*

# include nume *nume este numele unei macrocomenzi*

**Fişierele antet** sunt fişiere ASCII stocate în subdirectorul INCLUDE al directorului ce conţine compilatorul C + + .

Exemplu fişierul antet iostream.h oferă funcţii de bibliotecă pentru operaţiile de I/E în flux.

Orice program C++ începe cu una sau mai multe directive # include.

Directiva #define

Pentru definirea constantelor sau a **pseudofuncţiilor** (numite şi macrodefiniţii cu argumente) sub forma: #define macro text

#define macro valoare

#define nume (lista\_program) expresie

unde: macro este identificator

text = secvenţa de caractere

valoare = valoare numerică

nume = numele pseudofuncţiei

expresie = expresia pseudofuncţiei

La procesare, orice apariţie a identificatorului macro se înlocuieşte cu definiţia sa.

Exemplu: #define EPS 0.001

#define PI 3.1415

Observaţie: Pe o linie de cod nu pot exista mai multe directive #define.

Macrodefiniţiile cu parametri permit definirea pseudofuncţiilor care sunt mai rapide decât funcţiile obişnuite dar ocupă mai multa memorie.O pseudofuncţie acceptă ca argument un parametru şi înlocuieşte orice apariţie în program a acelui parametru prin valoarea furnizată la apelul pseudofuncţiei respective.

Exemplu: #define PATRAT (x) ((x)\*(x))

#define CUB (x) (PATRAT (x)\*(x))

Aceste două pseudodefiniţii calculează x2 respectiv x3, pentru x furnizat la apel.

Directiva #define se poate folosi şi pentru:

* înlocuirea cuvintelor rezervate sau a simbolurilor cu alţi identificatori definiţi de utilizator.
* crearea de identificatori pentru tipuri de date definite de utilizator cu ajutorul tipurilor standard.
* prescurtarea unor comenzi.

# Directiva #undef

Este opusă directivei #define permiţând anularea definiţiei curente a unui identificator, dacă acesta nu mai este necesar.

Are sintaxa: #undef macro

Cu ea eliberăm spaţiul de memorie ocupat de macrodefiniţia anulată şi putem reutiliza numele identificatorului acesteia într-o alta directivă #define.

Ex.: #undef EPS

#undef PATRAT

Directive condiţionale

Acestea sunt #if, #else, #elif, #endif, #ifdef, #ifndef, în care primele 4 directive controlează compilarea unor instrucţiuni în anumite condiţii şi au sintaxele:

# if cond1

secv1

[#elif cond2

secv2

. . . . . . . .

[#elif condn

secvn]

. . . . . . .

[#else

secv0]

# endif

Aici

secv1 este o secvenţa de directive de compilare care se execută cind cond1 este indeplinita şi analog pentru celelalte;

secv0 este secvenţa de directive de compilare care se execută cind nici una din condiţiile cond1, . . . . , condn nu este indeplinita.

Ex.: # if VAR = = 1

# define ALPHA char

# elif VAR = = 2

# define BETA boolean

# else

# define GAMA float

# endif

Directiva #ifdef compilează o linie de cod dacă un anumit identificator a fost definit anterior iar #ifndef dacă identificatorul nu a fost definit anterior.

Acestea au sintaxele:

# ifdef identificator

secv1

# endif

# ifndef identificator

secv2

#endif

unde secv1 = secvenţa de directive de compilare care se execută dacă identificator a fost definit anterior.

secv2 = secvenţa de directive de compilare care se execută dacă identificator nu a fost definit anterior.

Exemplu: # ifdef MAXIM

# define ALPHA char

# endif

# ifndef MAXIM

# define BETA boolean

# endif

Directiva # error

Generează un mesaj de eroare. Are sintaxa: # error text\_mesaj\_eroare.

Aceasta semnalează incompatibilitatea versiunilor unui program sau erori de sintaxă cauzate de omiterea sau folosirea incorectă a anumitor caractere ca: paranteze, acolade, punct şi virgula etc.

Exemplu: Error directive: cannot find command procesor

Error syntax.CPP4: undefined symbol “quotes” în funcţion main ( )

Ultima eroare de sintaxă se referă la linia a 4-a a fişierului sursă şi indică neincluderea unui mesaj între ghilimele.

Directiva #line

Aceasta permite precizarea numarului liniei curente compilate într-un program.

Are sintaxa: # line numar [“nume\_fişier”]

Directiva specifică numarul liniei din fişierul cod sursă original. Cu ea putem realiza referinţe încrucişate sau identifică poziţia din codul sursă unde s-au produs erori.