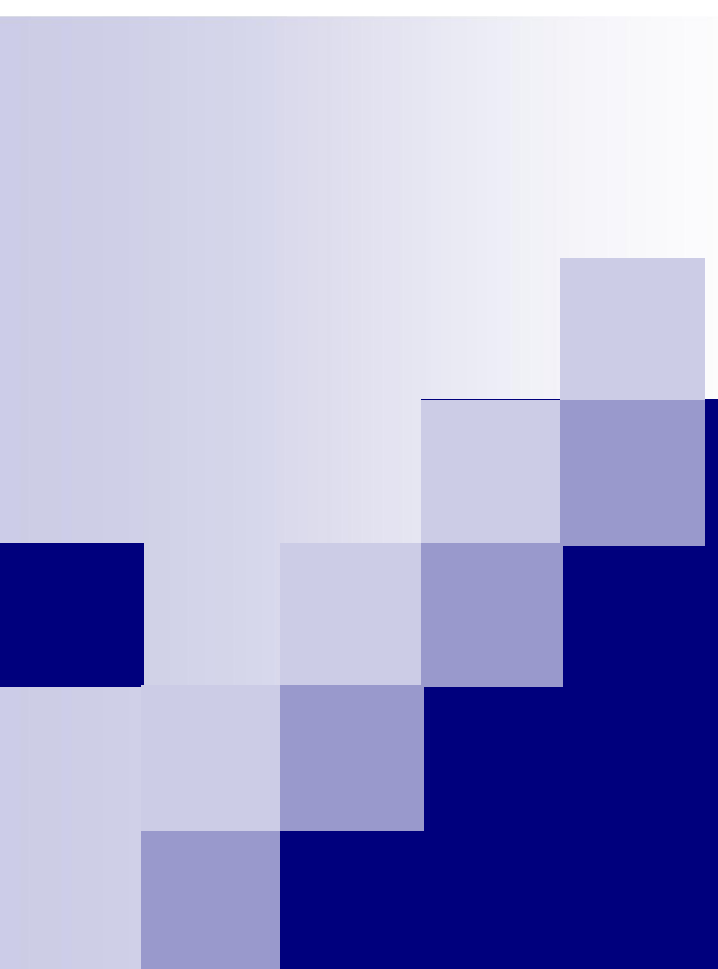




Clase și obiecte. Inițializarea și distrugerea obiectelor

Paradigme de programare

- *o paradigmă de programare este un stil sau o modalitate de programare.*

1) **programare declarativă**

- ☐ programare funcțională.
- ☐ programare logică.

2) **programare imperativă**

- ☐ programare structurată.

3) **programare orientată obiect.**

Programare declarativă, funcțională

- 1) **programare declarativă** - programatorul descrie logica unui calcul (cum ar trebui să arate rezultatul), fără să se prezinte modul de execuție (cum să-l obțină). Nu există instrucțiuni de ciclare, instrucțiuni de atribuire.
 - ❑ **programare funcțională** - procesul de calcul este descris prin evaluarea unor funcții matematice combinate în expresii, ce evită starea și datele muabile.
 - ❑ o funcție primește argumente și returnează o singură soluție; rezultatul depinde numai de datele de intrare, momentul de timp la care este apelată o funcție nu are nicio relevanță.
 - ❑ example: Scheme, Haskell, Miranda, ML.

Programare logică

1) programare declarativă

- ❑ **programare logică** - *programarea logică* și *programarea bazată de constrângeri* - programele sunt construite prin stabilirea unor relații ce specifică *fapte* și *reguli de inferență* precum și *întrebări* referitoare la faptul că dacă ceva este adevărat sau nu (adică specificarea unui scop).
- ❑ căutarea soluțiilor se realizează prin unificare și backtracking.
- ❑ se fac diverse afirmații logice despre o situație, stabilind toate faptele cunoscute.
- ❑ exemple: Prolog.

Programare imperativă

- 2) **programare imperativă** - procesul de calcul este descris printr-o secvență de pași ce modifică starea unui program (valoarea unei variabile este accesată sau modificată).
- ❑ un program constă într-o secvență de comenzi pentru acționarea calculatorului.
 - ❑ ordinea comenzilor este crucială, deoarece un anumit pas va avea consecințe diferite în funcție de valorile curente ale variabilelor.

Programare structurată

2) programare imperativă.

- ❑ **programare structurată** - fluxul de control este definit de instrucțiuni de ciclare imbricate, instrucțiuni de selecție și subrutine, mai degrabă decât prin intermediul salturilor de tip *goto*.
- ❑ variabilele sunt în general declarate local în cadrul unor blocuri.
- ❑ apărută după anul 1970 datorită complicării crescânde a programelor de calculator.
- ❑ exemple: C, Pascal, PL/I, Algol 68, Ada 83, Modula.

Programare orientată obiect

- 3) **programare orientată obiect** - un program este modelat ca o colecție de obiecte din diferite clase și se bazează pe trimiterea de mesaje către aceste obiecte.
- ❑ obiectele răspund la mesaje efectuând operații, denumite în general metode. Mesajele pot avea argumente.
 - ❑ discutăm în termeni de obiecte ce conțin date (propria memorie locală) și oferă metode (funcții ce operează pe obiecte - propriul set de operații).
 - ❑ abordarea tipică în paradigma imperativă = se operează asupra datelor prin intermediul procedurilor; de obicei datele sunt pasive, iar procedurile sunt active.
 - ❑ exemple: Java, C++, Simula, Smalltalk.

1950

A-0

IMPERATIVE

OBJECT ORIENTED

FUNCTIONAL

LOGICAL

OTHER

1960

FORTRAN 0
FORTRAN I
FORTRAN II
FORTRAN III

B-0
COBOL

ALGOL
ALGOL 60

USP

USP 1.0

USP 1.5

APL
APL/360

1970

FORTRAN IV
FORTRAN 66
ALTRAN

BASIC

PL/I

ALGOL W
PASCAL

ALGOL 68

SIMULA
SIMULA 67

BCPL

B

C

LOGO

ISWIM

APL
APL/360

1980

FORTRAN 77

ABC

Microsoft BASIC v2.0

BASIC & GWBASIC

QuickBASIC

QBASIC

Turbo PASCAL v4

Modula

Modula2

Ada

Ada ISO

Ada 95

Perl v1

Perl v2

Perl v3

Perl v4

Perl v5

Object Rexx

AWK

Rexx

Rexx 2

Rexx 3

Rexx 3.20

ANSI C

C with Classes

C++

Eiffel

SmallTalk

SmallTalk72

SmallTalk74

SmallTalk76

SmallTalk78

SmallTalk80

SmallTalk/V

ML

Scheme

Scheme MIT

Scheme 84

Scheme 1.0

Scheme 1.1

Scheme 1.2

Scheme 1.3

Scheme 1.4

Scheme 1.5

Scheme 1.6

Scheme 1.7

Scheme 1.8

Scheme 1.9

Scheme 2.0

Scheme 2.1

Scheme 2.2

Scheme 2.3

Scheme 2.4

Scheme 2.5

Scheme 2.6

Scheme 2.7

Scheme 2.8

Scheme 2.9

Scheme 3.0

Scheme 3.1

Scheme 3.2

Scheme 3.3

Scheme 3.4

Scheme 3.5

Scheme 3.6

Scheme 3.7

Scheme 3.8

Scheme 3.9

Scheme 4.0

Scheme 4.1

Scheme 4.2

Scheme 4.3

Scheme 4.4

Scheme 4.5

Scheme 4.6

Scheme 4.7

Scheme 4.8

Scheme 4.9

Scheme 5.0

Scheme 5.1

Scheme 5.2

Scheme 5.3

Scheme 5.4

Scheme 5.5

Scheme 5.6

Scheme 5.7

Scheme 5.8

Scheme 5.9

Scheme 6.0

Scheme 6.1

Scheme 6.2

Scheme 6.3

Scheme 6.4

Scheme 6.5

Scheme 6.6

Scheme 6.7

Scheme 6.8

Scheme 6.9

Scheme 7.0

Scheme 7.1

Scheme 7.2

Scheme 7.3

Scheme 7.4

Scheme 7.5

Scheme 7.6

Scheme 7.7

Scheme 7.8

Scheme 7.9

Scheme 8.0

Scheme 8.1

Scheme 8.2

Scheme 8.3

Scheme 8.4

Scheme 8.5

Scheme 8.6

Scheme 8.7

Scheme 8.8

Scheme 8.9

Scheme 9.0

Scheme 9.1

Scheme 9.2

Scheme 9.3

Scheme 9.4

Scheme 9.5

Scheme 9.6

Scheme 9.7

Scheme 9.8

Scheme 9.9

Scheme 10.0

Scheme 10.1

Scheme 10.2

Scheme 10.3

Scheme 10.4

Scheme 10.5

Scheme 10.6

Scheme 10.7

Scheme 10.8

Scheme 10.9

Scheme 11.0

Scheme 11.1

Scheme 11.2

Scheme 11.3

Scheme 11.4

Scheme 11.5

Scheme 11.6

Scheme 11.7

Scheme 11.8

Scheme 11.9

Scheme 12.0

Scheme 12.1

Scheme 12.2

Scheme 12.3

Scheme 12.4

Scheme 12.5

Scheme 12.6

Scheme 12.7

Scheme 12.8

Scheme 12.9

Scheme 13.0

Scheme 13.1

Scheme 13.2

Scheme 13.3

Scheme 13.4

Scheme 13.5

Scheme 13.6

Scheme 13.7

Scheme 13.8

Scheme 13.9

Scheme 14.0

Scheme 14.1

Scheme 14.2

Scheme 14.3

Scheme 14.4

Scheme 14.5

Scheme 14.6

Scheme 14.7

Scheme 14.8

Scheme 14.9

Scheme 15.0

Scheme 15.1

Scheme 15.2

Scheme 15.3

Scheme 15.4

Scheme 15.5

Scheme 15.6

Scheme 15.7

Scheme 15.8

Scheme 15.9

Scheme 16.0

Scheme 16.1

Scheme 16.2

Scheme 16.3

Scheme 16.4

Scheme 16.5

Scheme 16.6

Scheme 16.7

Scheme 16.8

Scheme 16.9

Scheme 17.0

Scheme 17.1

Scheme 17.2

Scheme 17.3

Scheme 17.4

Scheme 17.5

Scheme 17.6

Scheme 17.7

Scheme 17.8

Scheme 17.9

Scheme 18.0

Scheme 18.1

Scheme 18.2

Scheme 18.3

Scheme 18.4

Scheme 18.5

Scheme 18.6

Scheme 18.7

Scheme 18.8

Scheme 18.9

Scheme 19.0

Scheme 19.1

Scheme 19.2

Scheme 19.3

Scheme 19.4

Scheme 19.5

Scheme 19.6

Scheme 19.7

Scheme 19.8

Scheme 19.9

Scheme 20.0

Scheme 20.1

Scheme 20.2

Scheme 20.3

Scheme 20.4

Scheme 20.5

Scheme 20.6

Scheme 20.7

Scheme 20.8

Scheme 20.9

Scheme 21.0

Scheme 21.1

Scheme 21.2

Scheme 21.3

Scheme 21.4

Scheme 21.5

Scheme 21.6

Scheme 21.7

Scheme 21.8

Scheme 21.9

Scheme 22.0

Scheme 22.1

Scheme 22.2

Scheme 22.3

Scheme 22.4

Scheme 22.5

Scheme 22.6

Scheme 22.7

Scheme 22.8

Scheme 22.9

Scheme 23.0

Scheme 23.1

Scheme 23.2

Scheme 23.3

Scheme 23.4

Scheme 23.5

Scheme 23.6

Scheme 23.7

Scheme 23.8

Scheme 23.9

Scheme 24.0

Scheme 24.1

Scheme 24.2

Scheme 24.3

Scheme 24.4

Scheme 24.5

Scheme 24.6

Scheme 24.7

Scheme 24.8

Scheme 24.9

Scheme 25.0

Scheme 25.1

Scheme 25.2

Scheme 25.3

Scheme 25.4

Scheme 25.5

Scheme 25.6

Scheme 25.7

Scheme 25.8

Scheme 25.9

Scheme 26.0

Scheme 26.1

Scheme 26.2

Scheme 26.3

Scheme 26.4

Scheme 26.5

Scheme 26.6

Scheme 26.7

Scheme 26.8

Scheme 26.9

Scheme 27.0

Scheme 27.1

Scheme 27.2

Scheme 27.3

Scheme 27.4

Scheme 27.5

Scheme 27.6

Declararea/definirea unei funcții

Declarare:

```
<tip_returnat> <nume_functie>([<lista_param_formali>]);
```

Definire:

```
<tip_returnat> <nume_functie>([<lista_param_formali>]) {  
    <instructiuni>  
}
```

Declararea/definirea unei funcții

- la apelul unei funcții, lista parametrilor actuali trebuie să respecte lista parametrilor formali referitor la
 - tipul parametrilor,
 - numărul de parametri,
 - ordinea parametrilor.
- lista parametrilor formali trebuie să conțină cel puțin tipul parametrilor.
- dacă o funcție returnează altceva decât `void`, ea trebuie să conțină cel puțin o instrucțiune `return`.
- o funcție poate conține mai multe instrucțiuni `return`.
- instrucțiunea `return` este ultima instrucțiune executată.

Funcții inline

- la apelul unei funcții, se întrerupe execuția funcției curente (apelantă) și se execută un salt la adresa de memorie unde se găsește corpul funcției apelate; după terminarea funcției apelate, se revine la contextul anterior apelului, reluându-se execuția cu instrucțiunea următoare apelului din cadrul funcției apelante.
- înainte de declarația sau definiția unei funcții se poate folosi cuvântul cheie `inline`.
- o funcție declarată `inline` va fi expandată la compilare - compilatorul generează codul corespunzător funcției în poziția apelului, în loc de a genera secvența de apel.
 - compilatorul decide în final.
- crește performanța programelor ce folosesc multe funcții mici (3-4 instrucțiuni).
- metodele unei clase definite în interiorul clasei sunt considerate funcții `inline`.

Funcții cu valori implicite ale parametrilor

- În C++ se pot declara funcții cu valori implicite ale parametrilor.
- o funcție poate avea mai multe argumente cu valori implicite.
- argument implicit - o valoare dată la declarare pe care compilatorul o folosește dacă nu este furnizată o altă valoare.
- argumentele implicite trebuie să fie cele mai din dreapta din lista parametrilor.
- nu este permisă alternarea argumentelor cu și fără valori implicite.

```
float operatii(int a, float f = 1.6, int x = 2) {  
    if (a != 0) {  
        return a + f + x;  
    } else {  
        return f * x;  
    }  
}
```

Supraîncărcarea funcțiilor

- declararea mai multor funcții cu același nume, dar care diferă prin tipul și/sau numărul parametrilor.
- o funcție este definită de mai multe ori, diferența între definiții fiind dată de tipul parametrilor și numărul lor - niciodată de tipul returnat.
- când compilatorul întâlnește un apel la o funcție supraîncărcată, se încearcă selectarea funcției după numărul de parametri, iar dacă acest lucru nu se poate efectua în mod neambiguu, atunci selecția se va face după tipul lor.

Supraîncărcarea funcțiilor

```
int minim(int n, int a[]) {  
    int min = a[0];  
  
    for (int i = 1; i < n; i++) {  
        if (min > a[i]) {  
            min = a[i];  
        }  
    }  
  
    return min;  
}  
  
double minim(double x, double y) {  
    return (x > y) ? y : x;  
}
```

Sintaxa definirii unei clase

```
class IdNumeClasa {  
    [<IdSpecifierAcces>:] <ListaMembri>  
    [<IdSpecifierAcces>:] <ListaMembri>  
    ...  
    [<IdSpecifierAcces>:] <ListaMembri>  
};
```

unde:

<IdSpecifierAcces> := **private** | **protected** | **public**

<ListaMembri> := **<ListaDateMembre>** **<ListaFuncțiiMembre>**

Protecția datelor și funcțiilor membre

- ***private*** – datele și funcțiile aflate sub influența acestui specificator de acces NU pot fi accesate din afara clasei. *În mod implicit datele și funcțiile sunt private.*
- ***protected*** - datele și funcțiile aflate sub influența acestui specificator de acces nu pot fi accesate din afara clasei, cu excepția claselor *derivate*.
- ***public*** – datele și funcțiile aflate sub influența specificatorului *public*, pot fi accesate în afara clasei.

Obiecte

- Obiectul reprezintă ***o instanță a unei clase.***
- Sintaxa declarării unui obiect:

```
IdNumeClasă idOb1, ..., idObN;
```

- Accesarea datelor și funcțiilor membre:

```
idOb.idDataMembru
```

```
idPointerOb->idDataMembru
```

```
idOb.idMetodaMembru(lista de parametri);
```

```
idPointerOb->idMetodaMembru(lista de  
parametri);
```

Exemplu: Clasa Complex

```
class Complex {  
    private:  
        float re;  
        float im;  
    public:  
        void citire();  
        void afisare();  
        float modul();  
};
```

Exemplu: Clasa Complex

```
void Complex::citire() {  
    printf("Dati partea reala:"); scanf("%f", &re);  
    printf("Dati partea imaginara:"); scanf("%f", &im);  
}  
  
void Complex::afisare() {  
    printf("%g + %g * i", re, im);  
}  
  
float Complex::modul() {  
    return sqrt(re * re + im * im);  
}
```

Exemplu: Clasa Complex

```
int main() {  
    Complex z;  
    z.re = 2.5; // Incorect deoarece re este private  
    z.citire();  
    Complex *p = &z;  
    p->afisare();  
    return 0;  
}
```

Inițializarea obiectelor

- În cazul datelor de tipuri predefinite, este posibilă inițializarea acestora la momentul declarării.

Exemplu:

```
int x = 10;
```

```
float a[2] = {3, 8};
```

- În cazul obiectelor, inițializarea la declarare se face prin intermediul unor funcții speciale numite **constructori**.

Constructor – Definiție

- **Constructorul** este o funcție membră specială a unei clase ce se apelează în mod automat la crearea unui obiect.
- Roluri:
 - Inițializare
 - Alocare.

Constructor – Sintaxă (I)

```
class IdNumeClasa {  
    . . .  
    IdNumeClasa (<listaParametri>);  
    . . .  
};
```

```
IdNumeClasa::IdNumeClasa (<listaParametri>) {  
    //instructiuni  
}
```

Constructor – Sintaxă (II)

- Inițializarea obiectelor:

```
IdNumeClasa idObiect (<listaParametri>) ;
```

sau

```
IdNumeClasa idObiect = valParam;
```

în cazul în care lista de parametri este formată dintr-un singur parametru.

Exemplu: Clasa Complex

```
class Complex {  
    private:  
        float re;  
        float im;  
    public:  
        Complex(float r, float i);  
        void citire();  
        void afisare();  
        float modul();  
};
```

Exemplu: Clasa Complex

```
Complex::Complex(float r, float i) {  
    re = r;  
    im = i;  
}
```

```
int main() {  
    Complex z(7,3);  
    z.afisare();  
    return 0 ;  
}
```

Constructorii - Caracteristici

- Au același nume cu cel al clasei din care fac parte.
- Nu returnează nimic (nici măcar tipul **void**).
- O clasă poate avea mai mulți constructori.
- Nu pot primi ca parametri, instanțe ale clasei ce se definește, ci doar pointeri sau referințe la instanțele clasei respective.
- Constructorii nu sunt apelați explicit (în general).
- Constructorii nu se *moștenesc*.
- Constructorii nu pot fi funcții virtuale.

Tipuri de constructori (I)

■ Constructori implicați:

- *definit de utilizator* – constructor ce nu posedă niciun parametru.
- *generat de compiler* – dacă o clasă nu are niciun constructor definit atunci compilatorul generează unul automat, fără parametri, al cărui corp nu conține nicio instrucțiune.
- constructor cu toți parametrii implicați.

Exemplu: Constructor implicit

```
class Complex {  
    ...  
    public:  
        Complex() {  
            re = 0;  
            im = 0;  
            printf ("Apel constructor\n");  
        }  
    ...  
};  
...  
Complex z;  
z.afisare();  
...
```

Output

```
Apel constructor  
0+0*i
```

Tipuri de constructori (II)

■ Constructori cu parametri:

- *cu parametri ce nu iau valori implicite;*
- *cu parametri ce iau valori implicite.*

■ Funcții cu parametri implicați:

```
tip_r nume_functie(tip1 p1, ..., tipn pn, tipn+1 pn+1=vin+1, ...,  
tipm pm=vim);
```

$p_{n+1}, \dots, p_m$ = **parametri implicați**.

La apelul funcției aceștia pot să lipsească, caz în care ei au valorile implicite specificate la declarare.

Exemplu: Constructor cu parametri impliciți

```
class Complex {  
    ...  
    public:  
        Complex(float r = 0, float i = 0) {  
            re = r;  
            im = i;  
        }  
    ...  
};
```

```
...  
Complex z1(2,3), z2(4), z3 = 5, z4;  
z1.afisare();  
z2.afisare();  
z3.afisare();  
z4.afisare();  
...
```

Output

```
2+3*i  
4+0*i  
5+0*i  
0+0*i
```

Tipuri de constructori (III)

- **Constructorii de copiere** - inițializarea unui obiect cu valorile altui obiect, de același tip, deja existent.
- Constructorii de copiere pot fi:
 - *definiți de utilizator;*
 - *generați de compiler.*

Constructor de copiere - Sintaxa

```
class IdNumeClasa {  
    . . .  
    IdNumeClasa (IdNumeClasa &ob) ;  
    sau  
    IdNumeClasa (const IdNumeClasa &ob) ;  
    . . .  
};
```

Constructori de copiere - Utilizare

- Crearea de obiecte ce vor fi inițializate, pornind de la un obiect care există deja:

```
IdClasa ob2 = ob1;
```

- Apelul unei funcții ce lucrează cu obiecte transferate prin valoare, când este nevoie de crearea unei copii a obiectului pe stivă: de exemplu $f(ob)$; .

- Returnarea dintr-o funcție a unui obiect prin valoare:

```
return ob;
```

Exemplu: Constructor de copiere

```
class Complex {  
    ...  
    public:  
        Complex(const Complex &z) {  
            re = z.re;  
            im = z.im;  
            printf("Apel constructor de copiere\n") ;  
        }  
    ...  
};  
void f(Complex z) {  
    ...  
}  
...  
Complex z1(2,3);  
z1.afisare();  
Complex z2 = z1;  
z2.afisare();  
f(z1);
```

Output

```
2+3*i  
Apel constructor de copiere  
2+3*i  
Apel constructor de copiere
```

Destructori

- **Destructorul** este o funcție membră specială a unei clase ce se apelează în mod automat la distrugerea unui obiect.
- **Rol:** eliberarea zonelor alocate dinamic, resurselor, etc.
- Tipuri de destructor:
 - definit de utilizator,
 - generat de compilator.

Destructor – Sintaxă

```
class IdNumeClasa {  
    . . .  
    ~IdNumeClasa ();  
    . . .  
};
```

```
IdNumeClasa::~~IdNumeClasa () {  
    //instruțiuni  
}
```

Exemplu: Destructor

```
class Complex {
    ...
public:
    ~Complex() {
        printf("Distrugere obiect:");
        afisare();
    }
    ...
};

int main() {
    Complex z1(2,3), z2(4,7);
    z1.afisare();
    z2.afisare();
    return 0 ;
}
```

Output

```
2+3*i
4+7*i
Distrugere obiect: 4+7*i
Distrugere obiect: 2+3*i
```

Destructori - Caracteristici

- Are același nume cu numele clasei și este precedat de '~'.
- Nu are parametri.
- Nu returnează nimic (nici măcar `void`).
- O clasă poate avea un singur destructor.
- Pot fi funcții virtuale.

Temă

- Implementați clasa `NumarRational`.
- Implementați clasa `Carte`.
- Implementați clasa `Student` în care numele se va aloca dinamic.



Referințe

- <http://blog.jenkster.com/2015/12/what-is-functional-programming.html>
- <http://david.tribble.com/text/goto.html>
- <https://www.veracode.com/blog/2013/04/the-history-of-programming-languages-infographic>