Clase şi obiecte. Iniţializarea şi 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 primeşte argumente şi 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 comezilor este crucială, deoarece un anumit pas va avea consecințe diferite în funcție de valorile curente ale variabilelor.

Programare structurată

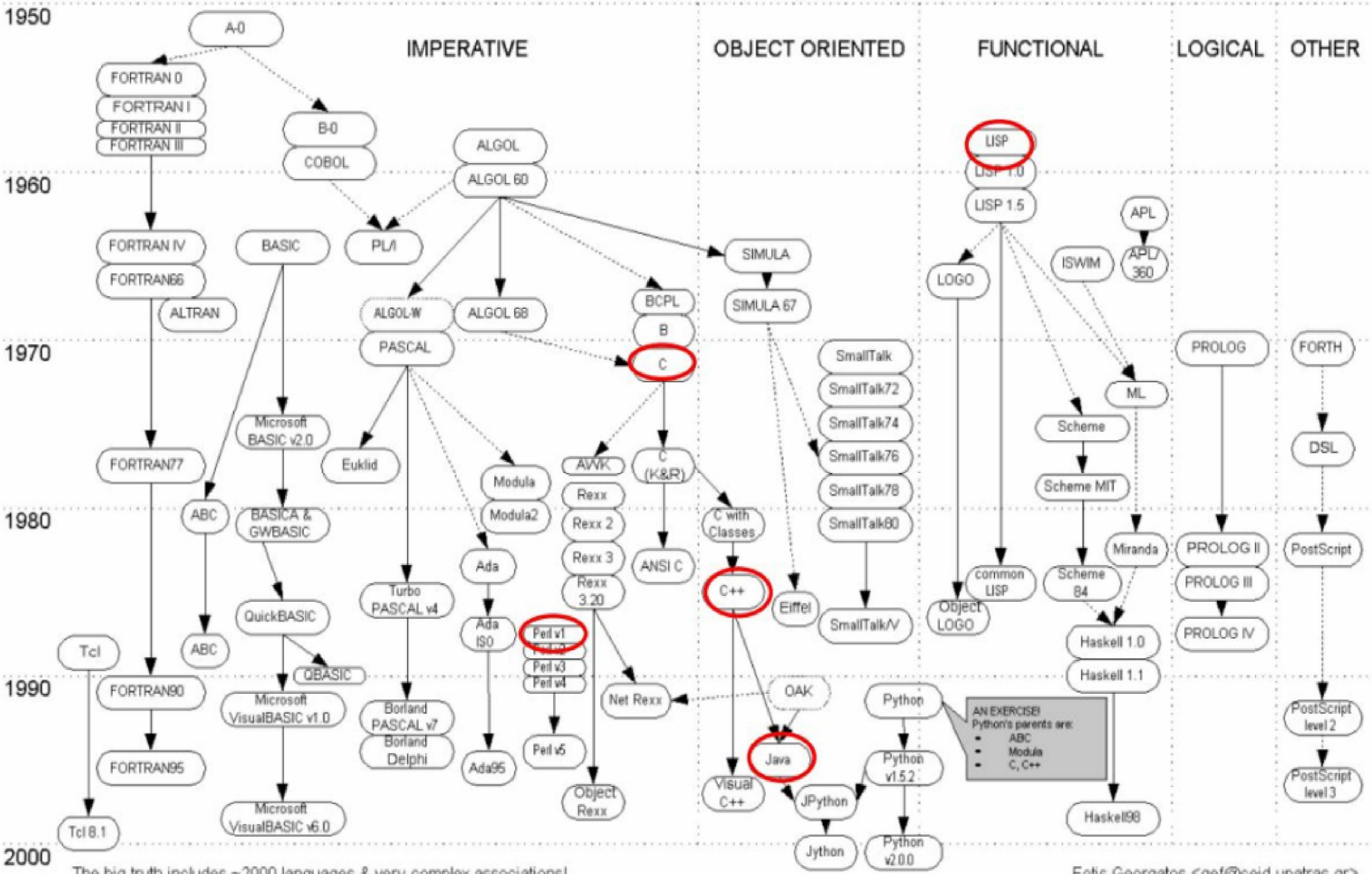
2) **programare imperativă**.

- **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.



The big truth includes ~2000 languages & very complex associations!
Read for more: *http://perso.wanadoo.fr/levenez/lang/history.html*

http://amstel.science.uva.nl/~fotisg/python

Fotis Georgatos <gef@ceid.upatras.gr>
Amstel Institute, Amsterdam, June 2002

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) şi se execută un salt la adresa de memorie unde se găseşte 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.
- creşte 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 foloseşte 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 şi 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 acelaşi nume, dar care diferă prin tipul şi/sau numărul parametrilor.
- o funcție este definită de mai multe ori, diferența între definiri fiind dată de tipul parametrilor şi numărul lor - niciodată de tipul returnat.
- când compilatorul întâlneşte 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 {
      [<IdSpecificatorAcces>:] <ListaMembri>
      [<IdSpecificatorAcces>:] <ListaMembri>
      …
      [<IdSpecificatorAcces>:] <ListaMembri>
  };
```

unde:

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

<*ListaMembri*>:=<**ListaDateMembre**><**ListaFunctiiMembre**>

Protecția datelor şi funcțiilor membre

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

Constructori - Caracteristici

- Au acelaşi 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 defineşte, ci doar pointeri sau referințe la instanțele clasei respective.
- Constructorii nu sunt apelați explicit (în general).
- Constructorii nu se *moştenesc.*
- Constructorii nu pot fi funcții virtuale.

Tipuri de constructori (I)

- **Constructori impliciți:**
  - *definit de utilizator* – constructor ce nu posedă niciun parametru.
  - *generat de compilator* – 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 impliciți.

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 impliciți:

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

p_{n+1}, …, p_m = **parametri impliciți.**

La apelul funcției aceştia 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)

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

Constructori 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() {
   //instructiuni
}
```

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 acelaşi nume cu numele clasei şi 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