Alocarea dinamică de memorie.
Date şi funcţii statice.

Separarea codului: fişiere .h şi .cpp

- Fişiere header (.h)
  - conțin macrouri, declarații de structuri de date şi enumerări, declarații de clase, prototipuri de funcții, constante, definiții de funcții statice, funcții/metode inline, declarații externe de variabile globale, definiții de şabloane (templates).
  - conțin declarația / definiția clasei.
  - permit compilatorului să recunoască clasele când sunt folosite în altă parte.
- Fişiere sursă (.cpp)
  - definiții de funcții, definiții de metode, definiții de variabile globale, inițializări.
  - conțin implementarea codului sursă.
  - programul principal.
  - funcții de test.
- Exemplul 2
  - circle.h
  - circle.cpp
  - test-circle.cpp

Directive de compilare condiționată

- `#if`, `#ifdef`, `#ifndef`: `#ifdef` și `#ifndef` sunt folosite de obicei pentru a evita incluziunea multiplă a fişierelor antet (header).
- Sintaxă

```
#ifdef nume
   text
#endif
```

```
#ifdef nume
        text1
#else
        text2
#endif
```

- ***sau***

```
#ifndef nume
     text
#endif
```

```
#ifndef nume
        text1
#else
        text2
#endif
```

- ***unde***
  - `nume` este o constantă ce este testată de către preprocesor dacă nu este definită, `text`, `text1`, `text2` sunt porțiuni de cod sursă
  - dacă `nume` nu este definită (`#ifndef`) atunci `text` respectiv `text1` sunt compilate, altfel numai `text2` este compilat și procesarea continuă după `#endif`.

Fişiere header – circle.h

```
#ifndef __CIRCLE_
#define __CIRCLE_

#include <string>

using namespace std;

#define MPI 3.1416

class Circle {
  private:
    double radius;        // raza cercului; atribut, data membru
    string color;         // culoarea cercului; data membru
  public:
    // Constructor cu parametri luand valori implicite
    Circle(double r = 1.0, string c = "red");
    Circle(Circle& c);    // Constructor de copiere

    // Functii membre: getteri si setteri
    double getRadius();
    void setRadius(double radius);

    string getColor();
    void setColor(string c);

    double getArea();
    string toString();
};
#endif // __CIRCLE_
```

Fişiere sursă – circle.cpp

```
#include <sstream>
#include "circle.h"

Circle::Circle(double r, string c) {
  radius = r;
  color = c;
}

Circle::Circle(Circle& c) {
  radius = c.getRadius();
  color = c.getColor();
}

double Circle::getRadius() {
  return radius;
}

void Circle::setRadius(double radius) {
  this->radius = radius;
}
```

```
string Circle::getColor() {
  return color;
}

void Circle::setColor(string c) {
  color = c;
}

double Circle::getArea() {
  return radius * radius * MPI;
}

string Circle::toString() {
  ostringstream oss;

  oss << "Circle[radius=" << radius
      << ", color=" << color << "]";

  return oss.str();
}
```

Fişiere sursă – test-circle.cpp

```
#include <iostream>

#include "circle.h"

using namespace std;

int main() {
  // Construim un obiect de tip Circle
  Circle c1(2.44, "yellow");

  cout << "Raza = " << c1.getRadius()
       << " Arie = " << c1.getArea()
       << " Culoare = " << c1.getColor()
       << endl;

  // Construim un alt obiect de tip Circle
  Circle c2(6.12); // culoare implicita

  cout << "Raza = " << c2.getRadius()
       << " Arie = " << c2.getArea()
       << " Culoare = " << c2.getColor()
       << endl;

  // folosim constructorul fara parametri
  Circle c3;

  // modificam raza si culoarea cercului c3
  c3.setRadius(1.33);
  c3.setColor("white");

  cout << "Raza = " << c3.getRadius()
       << " Arie = " << c3.getArea()
       << " Culoare = " << c3.getColor()
       << endl;

  // folosim metoda toString() pentru afisare
  cout << c3.toString() << '\n';

  // folosim constructorul de copiere
  Circle c4 = c3;

  cout << "Obiectul c4: " << c4.toString()
       << '\n';

  c4.setRadius(3.91);
  c4.setColor("blue");

  cout << "Obiectul c4: " << c4.toString()
       << '\n';

  return 0;
}
```

Separarea codului: fişiere .h şi .cpp

- un fișier **.h** și fişierul **.cpp** corespunzător ar trebui să corespundă unei părți clare de funcționalitate.
- într-un fișier de antet se vor folosi întotdeauna *directive de compilare condiționată*.
- variabile globale pentru un modul se declară extern în fișierul antet și se definesc în fișierul **`.cpp`**.
- declarațiile interne ale unui modul se păstrează în afara fișierului antet.
- fiecare fișier antet **`A.h`** ar trebui să includă orice alt fișier antet pe care **`A.h`** îl necesită pentru a se compila corect, dar nu mai multe.
- dacă utilizați o declarație incompletă a unui tip de dată `X`, folosiți-o în loc de a include antetul **`X.h`**.
- conținutul unui fișier antet ar trebui să fie compilat corect de la sine.
- fișierul **`A.cpp`** ar trebui să includă mai întâi fișierul `A.h`, apoi orice alte fişiere antet necesare pentru compilarea codului său.
- nu se vor include niciodată fișiere **`.cpp`** .

Alocare dinamică de memorie – limbajul C

- Alocarea/eliberarea de memorie în limbajul C:

```
void* malloc(size_t size);
void free(void* ptr);
```

- Dezavantaje:
  - necesită conversia explicită a pointerului returnat şi specificarea numărului de octeți.

```
int* p = (int *) malloc(sizeof(int));
```

  - nu permite inițializarea / eliberarea obiectelor.
    - `malloc()` nu apelează constructorul.
    - `free()` nu apelează destructorul.
- În programele C++ nu vom utiliza `malloc()` şi `free()` pentru a aloca / elibera **obiecte** în mod dinamic.

Operatorul **new** pentru alocare dinamică (I)

- operatorul **new** pentru *alocare dinamică*

```
idPointer = new tip;
```

alocă un spațiu de memorie în **heap** pentru a reține o dată de tipul tip.

- Exemplu:

```
float* p;
p = new float;
*p = 2.67;
```

Operatorul **new** pentru alocare dinamică (II)

- operatorul **new** pentru *alocare dinamică* şi *inițializare*

```
idPointer = new tip(expresie);
```

  alocă un spațiu de memorie în **heap** pentru a reține o dată de tipul tip şi inițializează data din zona alocată cu valoarea expresiei.

- Exemplu:

```
float* p;
p = new float(2.67);
```

Operatorul **new** pentru alocare dinamică (III)

- operatorul **new** pentru *alocare dinamică de tablouri*

```
idPointer = new tip[dim_max];
```

  alocă un spațiu de memorie în **heap** pentru a reține un tablou de dim_max elemente de tipul tip.

- Exemplu:

```
float *t;
t = new float[10];
t[0] = 2.3;
t[1] = 7.2;
```

Operatorul **delete**

- operatorul **delete** eliberează zonele alocate dinamic:

```
delete idPointer;
delete []idPointerTablou;
```

- Exemplu:

```
int *p = new int;
float *t = new float[10];
. . .
delete p;
delete []t;
```

Alocare dinamică de obiecte (I)

- Alocare spațiu pentru un singur obiect:

```
idPointerObiect = new IdClasa(lista_de_parametri);
```

Alocă un spațiu de memorie în **heap** pentru a reține un obiect de tipul IdClasa, pentru inițializarea acestuia apelându-se constructorul clasei cu parametrii din lista_de_parametri.

Dacă lista de parametri este vidă atunci se apelează constructorul implicit.

Alocare dinamică de obiecte (II)

- Alocare spațiu pentru un tablou de obiecte

```
idPointerObiect = new IdClasa[dim_max];
```

Alocă un spațiu de memorie în **heap** pentru a reține un tablou de dim_max obiecte de tipul IdClasa, pentru inițializarea acestora apelându-se de dim_max ori constructorul implict.

Eliberarea obiectelor alocate dinamic (I)

- Eliberare obiect:

```
delete idPointerObiect;
```

Se eliberează spațiul de memorie indicat de idPointerObiect. Înainte de eliberare se apelează automat destructorul obiectului referit.

Eliberarea obiectelor alocate dinamic (II)

- Eliberare tablou:

```
delete []idPointerTablouDeObiecte;
```

  Se eliberează spațiul de memorie indicat de `idPointerTablouDeObiecte`. Înainte de eliberare se apelează automat destructorul fiecărui obiect din tablou.

Exemplu

```
class Punct {
  private:
    float x;
    float y;
  public:
    Punct() {
      x = 0;
      y = 0;
      printf("Apel constructor implicit Punct(0,0)\n");
    }
    Punct(float x, float y) {
      this->x = x;
      this->y = y;
      printf("Apel constructor Punct(%g,%g)\n", x, y);
    }
    ~Punct() {
      printf("Apel destructor (%g, %g)\n", x, y);
    }
};

int main() {
  Punct *p1 = new Punct;
  Punct *p2 = new Punct(3,4);
  Punct *t = new Punct[3];
  delete p1;
  delete p2;
  printf("Eliberez tabloul:\n");
  delete[] t;
  return 0;
}
```

Output

```
Apel constructor implicit Punct(0,0)
Apel constructor Punct(3,4)
Apel constructor implicit Punct(0,0)
Apel constructor implicit Punct(0,0)
Apel constructor implicit Punct(0,0)
Apel destructor pentru obiectul (0, 0)
Apel destructor pentru obiectul (3, 4)
Eliberez tabloul:
Apel destructor pentru obiectul (0, 0)
Apel destructor pentru obiectul (0, 0)
Apel destructor pentru obiectul (0, 0)
```

Pointerul **this**

- Este un *parametru introdus de compilator* în mod implicit la apelului unei funcții membru.
- El *indică către obiectul curent* pentru care a fost apelată funcția membru.

Referințe vs pointeri (I)

- Referințele şi pointerii pot fi inițializați astfel:

```
int a = 10;
int *pa = &a;
int &refa = a;
```

- Un pointer poate să rămână neinițializat.
- *O referință nu poate să rămână neinițializată*.
- Astfel avem garanția că o referință duce la o zonă de memorie validă.

```
float &refp;  // Eroare la compilare
```

Referințe vs pointeri (II)

- Pointerii îşi pot schimba valoarea (pot să puncteze la mai multe variabile) pe parcursul execuției unui program.

```
int a = 10;
int b = 5;
int *pa = &a;
pa = &b;
```

- Referințele pot puncta doar la o singură variabilă: cea cu care au fost inițializați.

```
int a = 10;
int b = 5;
int &refa = a;
&refa = b;      // Eroare la compilare
```

- Un pointer poate avea o valoare `NULL`. O referință trebuie să puncteze la o zonă de memorie validă.

Referințe vs pointeri (III)

- Pointerii acceptă anumite operații aritmetice: ‘+’, ‘–’, ‘++’ etc.

```
int a = 10;
int b = 5;
int *pa = &a;
pa++;
(*pi) = 30;
```

- Referințele nu acceptă operații aritmetice.

```
int a = 10;
int &refa = a;
refa++;
(&refa)++;    // Eroare la compilare
```

Referințe vs pointeri (IV)

- Pointerii acceptă cast-uri între ei. Orice pointer acceptă implicit un cast la un pointer de tipul `void` (`void*`).

```
int a = 10;
char *pa = (char*)&a;
pa++;
(*pa) = 30;
```

- Referințele nu acceptă cast-uri.

```
int a = 10;
char &refa = a; // Eroare la compilare
```

- Se garantează astfel că referința punctează la o variabilă de un anumit tip.

Referințe vs pointeri (V)

- Pointerii acceptă multiple indirectări (un pointer poate puncta către un alt pointer).

```
int a = 10;
int *pa = &a;
int **ppa = &pa;
**ppa = 30;
```

- O referință punctează tot timpul către un singur obiect de un anumit tip.

```
int a = 10;
int &refa = a;
int & &refrefa = refa; // Eroare la compilare
```

Referințe vs pointeri (VI)

- Pointerii pot fi utilizați în array-uri (şi inițializați dinamic în acestea).

```
int *a[1000];
```

- Referințele nu pot fi legate de un array (nu se poate crea un array de referințe).

```
int &ar[1000]; // Eroare la compilare
```

- Se pot crea pointeri către o referință ce punctează la o valoare temporară.

```
const int &refa = int(7);
int *pa = (int*)&refa;
```

Date membre statice (I)

- Datele membre statice sunt *date membre ale unei clase ce sunt partajate de toate obiectele*.
- Spre deosebire de datele membre obişnuite, care există în câte un exemplar pentru fiecare obiect, *datele membre statice nu aparţin fiecărui obiect al clasei*, ci există într-un *singur exemplar comun tuturor obiectelor clasei respective*.
- Se iniţializează automat cu valoarea `0`.

Date membre statice (II)

- Se declară în interiorul clasei cu ajutorul cuvântului cheie `static`:

```
class IdClasa {
   …
   static tip idDataMembraStatica;
   …
};
```

- Se definesc în afara clasei.

```
tip IdClasa::idDataMembraStatica = val_initala;
```

- Referirea datelor membre statice se poate face astfel:

```
IdClasa::idDataMembraStatica        // Recomandat
IdObiect.idDataMembraStatica
IdPointerObiect->idDataMembraStatica
```

Date membre statice. Exemplu

```
class Produs {
      private:
          char nume[100];
          float pret; //fara TVA
          static float PROCENT_TVA;
      public:
          Produs(char *nume, float pret) {
             strcpy(this->nume, nume);
             this->pret = pret;
          }

          void afisare() {
               printf("Nume: %s\n", nume);
               printf("Pret fara TVA: %g\n", pret);
               printf("Pret cu TVA: %g\n", getPretCuTVA());
          }

          float getPretCuTVA() {
              return pret * ( 1 + PROCENT_TVA);
          }
};

float Produs::PROCENT_TVA = 0.21;
```

Declarare

Definire

```
int main(){
    Produs p1("CPU", 100);
    Produs p2("HDD", 200);
    p1.afisare();
    p2.afisare();

    return 0;
}
```

OUTPUT

```
Nume: CPU
Pret fara TVA: 100
Pret cu TVA: 121
Nume: HDD
Pret fara TVA: 200
Pret cu TVA: 242
```

Funcții membre statice (I)

- **Metodele statice** se utilizează pentru a sugera faptul că acționează asupra instanțelor clasei *văzute ca un întreg* şi nu asupra instanțelor clasei în mod individual.
- Pot acționa doar asupra atributelor *statice* (ale *claselor*).
- Nu pot acționa asupra atributelor obiectului curent (deoarece nu li se transmite pointerul `this`).
- Nu pot apela decât funcții membre statice.

Funcții membre statice (II)

- Sintaxa declarării/definirii:

```
class IdClasa {
  static tip idMetodaStatica(lista_de_parametri);
};
tip IdClasa::idMetodaStatica(lista_de_parametri) {
…
}
```

- Utilizare:

```
IdClasa::idMetodaStatica(lista_de_parametri); //Recomandat
IdObiect.idMetodaStatica(lista_de_parametri);
IdPointerObiect->idMetodaStatica(lista_de_parametri);
```

Funcții membre statice. Exemplu

```
class Produs {
      . . .
          static float PROCENT_TVA;
      public:
          static float getTVA();
          static void setTVA(float TVA);
};

float Produs::PROCENT_TVA = 0.21;

float Produs::getTVA() {
    return PROCENT_TVA;
}

void Produs::setTVA(float TVA) {
    PROCENT_TVA = TVA;
}

int main() {
    Produs p1("CPU", 100);
    Produs p2("HDD", 200);
    p1.afisare();
    p2.afisare();
    Produs:: PROCENT_TVA = 0.25; Inaccesibil!
    Produs::setTVA(0.25);
    p1.afisare();
    p2.afisare();
}
```

OUTPUT

```
Nume: CPU
Pret fara TVA: 100
Pret cu TVA: 121
Nume: HDD
Pret fara TVA: 200
Pret cu TVA: 242
Nume: CPU
Pret fara TVA: 100
Pret cu TVA: 125
Nume: HDD
Pret fara TVA: 200
Pret cu TVA: 250
```

Temă

- Să se implementeze clasa `Matrice` în care elementele sunt reprezentate sub forma unui tabel unidimensional alocat dinamic.
- Să se implementeze clasa `DataCalendaristica` care să permită afişarea datei sub mai multe formate (*25.07.2013*, *25 iulie 2013*).

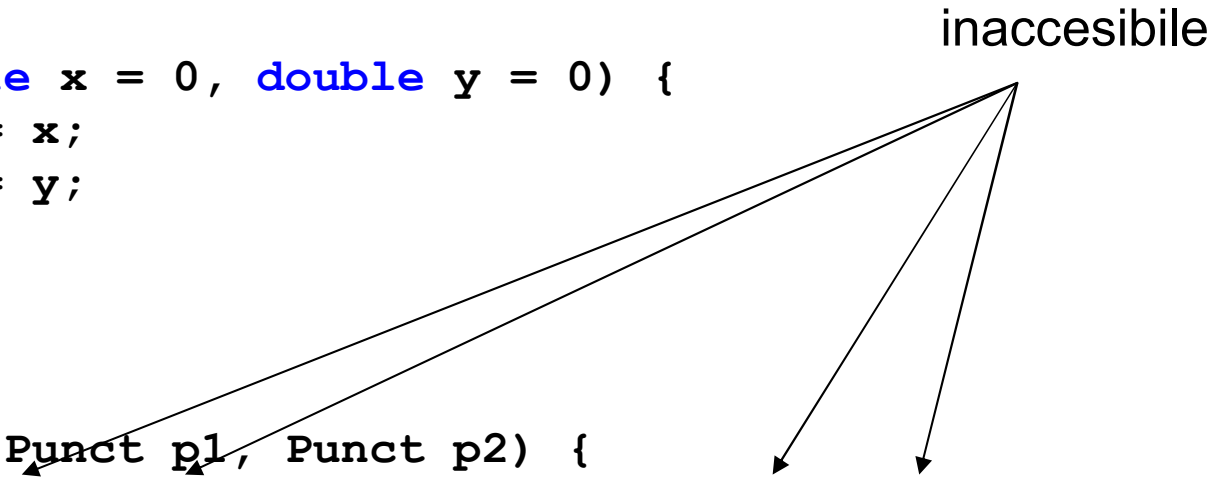
Clase şi funcţii friend.

Funcții friend (I)

```
class Punct {
    private:
      double x, y;
    public:
     Punct(double x = 0, double y = 0) {
       this->x = x;
       this->y = y;
     }

};

double distanta(Punct p1, Punct p2) {
  return sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y));
}
```



Funcții friend (II)

- Funcțiile **`friend`** (prietene) sunt funcții asociate unor clase care au acces la datele şi metodele protejate ale acelor clase, deşi nu sunt funcții membre ale acelei clase.
- Tipuri de *funcții prieten*:
  - funcții globale;
  - funcții membre ale altor clase.

Funcții friend globale

- Declararea unei funcții **`friend`** se face incluzând prototipul ei, precedat de cuvântul cheie **`friend`**, în acea clasă:

```
class IdClasa {
  friend tip_ret id_functie_prieten(lista_de_parametri);
};
```

- Definiția funcției se face în afara clasei:

```
tip_ret id_functie_prieten(lista_de_parametri) {
  // corpul de instructiuni în care avem acces la datele
  // protejate ale obiectelor clasei IdClasa
}
```

Funcții friend globale

```
class Punct {
    private:
      double x, y;
    public:
     Punct(double x = 0, double y = 0) {
       this->x = x;
       this->y = y;
     }
     friend double distanta(Punct p1, Punct p2);
};

double distanta(Punct p1, Punct p2) {
  return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}

int main() {
    Punct p1(1, 0), p2(4, 4);
    cout << "distanta=" << distanta(p1, p2);

    return 0;
}
```

Funcția distanta este declarată funcție prieten a clasei `Punct`

Definiția funcției distanta. Avem acces asupra datelor private `x` şi `y` din clasa `Punct`

Funcții friend, membre ale altor clase

- Sunt funcții membre ale unei clase ce au acces la datele membru protejate ale unei alte clase.
- Declararea unei funcții **`friend`** se face incluzând prototipul ei, precedat de cuvântul cheie **`friend`**, în clasa în care se doreşte accesul:

```
class IdClasaA; //declarare a clasei IdClasaA inaintea definirii clasei IdClasaB
class IdClasaB {
   tip_ret id_functie_prieten(lista_de_parametri); //declararea functiei
};

class IdClasaA {
   friend tip_ret IdClasaB::id_functie_prieten(lista_de_parametri);
};

tip_ret IdClasaB::id_functie_prieten(lista_de_parametri) {
  // corpul de instructiuni în care avem acces la datele protejate ale
  // obiectelor clasei IdClasaA
}
```

Funcții friend membre ale altor clase

```
class Punct;

class PoligonConvex {
      Punct *varfuri;
      int n;
   public:
      PoligonConvex (int n, Punct v[]);
      ~PoligonConvex();
      void afiseaza();
};

class Punct {
    private:
      double x, y;
    public:
     Punct(double x = 0, double y = 0);
     friend void PoligonConvex::afiseaza();
};

Punct::Punct(double x, double y) {
       this->x = x;
       this->y = y;
}
```

```
PoligonConvex::PoligonConvex(int n, Punct v[]) {
    this->n = n;
    varfuri = new Punct[n];
    for (int i = 0; i < n; i++) {
        varfuri[i] = v[i];
    }
}

PoligonConvex::~PoligonConvex() {
   delete []varfuri;
}

void PoligonConvex::afiseaza() {
     cout << "[";
     for (int i = 0; i < n; i++) {
       cout << "(" << varfuri[i].x << ","
            << varfuri[i].y << ")";
     }
     cout<<"]";
}

int main() {
    Punct t[3] = {Punct(0,0), Punct(3,0),
Punct(3,4)};
    PoligonConvex p(3, t);
    p.afiseaza();
    return 0 ;
}
```

Clase friend (I)

- Dacă dorim ca toate metodele dintr-o clasă **`IdClasaB`** să aibă acces asupra tuturor datelor membre protejate ale unei alte clase **`IdClasaA`**, atunci declarăm clasa **`IdClasaB`** ca fiind clasă **`friend`** (*prietenă*) pentru clasa **`IdClasaA`**.
- Sintaxa declarării claselor prietene este următoarea:

```
class IdClasaB; // declarare a clasei IdClasaB inaintea
                // definirii clasei IdClasaA
class IdClasaA {
   friend class IdClasaB;
};
```

Clase friend (II)

- Relația de *prietenie* dintre două clase **nu este reflexivă:** dacă clasa **`IdClasaA`** este clasă prieten a clasei **`IdClasaB`**, aceasta nu înseamnă că şi clasa **`IdClasaB`** este clasă prietenă a clasei **`IdClasaA`**.
- Relația de prietenie **nu este tranzitivă**: dacă clasa **`IdClasaA`** este clasă prietenă clasei **`IdClasaB`**, iar **`IdClasaB`** este clasă prietenă clasei **`IdClasaC`**, aceasta *nu implică* faptul că şi **`IdClasaA`** este clasă prietenă a clasei **`IdClasaC`**.
- Relația de prietenie **nu se moşteneşte** în clasele *derivate*.

Clase Friend. Exemplu

```
class Segment;//Declarare clasa Segment

class Punct {
    private:
     double x, y;
     void afisare();
    public:
     Punct(double x = 0, double y = 0);
     //Declaram clasa Segment ca fiind
     //functie prietena a clasei Punct
     friend class Segment;
};

class Segment {
      Punct o;
      Punct v;
   public:
      Segment(Punct o, Punct v);
      double lungime();
      void afisare();
};

Punct::Punct(double x, double y) {
      this->x = x;
      this->y = y;
}
```

```
void Punct::afisare() {
  cout << "(" << x << "," << y <<")";
}

Segment::Segment (Punct o, Punct v) {
  this->o = o;
  this->v = v;
}

double Segment::lungime() {
  return sqrt((o.x-v.x)*(o.x-v.x)+
              (o.y-v.y)*(o.y-v.y));
}
void Segment::afisare() {
  cout << "[";
  o.afisare();
  cout << ",";
  v.afisare();
  cout << "]";
}
int main() {
  Punct o(1,0), v(4,4);
  Segment s(o, v);
  s.afisare();
  cout << "\nLungime =" << s.lungime();
  return 0;
}
```

Avem acces la date şi metode private din clasa Punct

Referințe

- https://www.slideshare.net/rosedu/rosedu-tech-talks-prezentarea-01-preprocesorul-c
- http://users.utcluj.ro/~igiosan/Resources/PC/Curs/C05.pdf
- https://sites.google.com/site/razvanaciu/tehnici-de-programare/tehnici-de-programare---06---stepwise-refinement-preprocesorul