Programare II
Programare Orientată Obiect

Curs 1

Scop și obiective

- Scop
  - Acumularea de cunoștințe și aptitudini necesare realizări unor aplicații orientate obiect
- Obiective
  - Prezentarea și învătarea conceptelor programării orientate obiect
  - Abilitatea de a proiecta aplicații orienate obiect
- Limbaj de programare folosit pentru ilustrarea conceptelor de programare orientată obiect C++
  - Crearea de aplicații consolă
  - Biblioteca STL

Informații administrative

- Curs – Flavia Micota
  - Cabinet 046B
  - E-mail: flavia.micota@e-uvt.ro
  - web.info.uvt.ro/~zflavia -> Programare II
- Laboratoare
  - Flavia Micota
  - Adriana Diniș - adriana.dinis@e-uvt.ro
  - Florin Roșu - florin.rosu@e-uvt.ro
  - Sebastian Ștefănigă - sebastian.stefaniga@e-uvt.ro
  - Todor Ivașcu - todor.ivascu@e-uvt.ro
- Consultații
  - Marți: 8:00 – 9:00
  - Joi: 13:00 – 14:00

Informații administrative

- Laborator
  - Schimbarea grupelor la laborator se va face cu acordul cadrului didactic la care doriți sa mergeți
  - Recuperarea unui laborator se realizează în timpul săptămânei în care ați lipsit, cu acordul cadrului didactic la care doriți să mergeți
  - Pentru a putea susține examenul în prima sesiune trebuie să aveți minim 10 prezențe la laborator
  - Dacă numărul de prezente este mai mic decat 7 și nu promovați examenul după primele două sesiuni de restanțe va trebui să recontractați materia în anul II

Informații administrative

- Teme
  - Temele vor fi submise pe platforma elearning.e-uvt.ro la cursul de Progamare II

Informații administrative

- Regli de evaluare
  - Curs
    - 0.25 pct Prezență curs
    - 0.25 pct Corectitudine răspunsuri la testele de la curs
  - Laborator
    - 0.25 pct Prezență laborator
    - 0.5 pct Activitate laborator
    - 1 pct Teme
    - 2 pct Test de laborator în timpul semestrului (aproximativ saptamâna 7)
  - Examen
    - 4.5 pct Proba scrisa examen
      - ! **Nota de la examenul scris va fi luată în considerare la calcul medie dacă depășește 4.5**
    - 2 pct Proba laborator după examenul scris

Curriculă curs

- Paradigme de programare. Scurt istoric OOP și C++
- Concepte de bază a programării orientate obiect
- Clase (I). Declararea claselor. Istanțierea obiectelor. Membri clasei
- Clase(II). Controlul accesului. Constructori. Destructor. Autoreferentiere
- Exemplu de implementare a unui aplicații orienate obiect
- Supraîncărcarea operatorilor (I)
- Supraîncărcarea operatorilor (II). Excepții

Curriculă curs

- Moștenire simplă
- Moștenire multiplă
- Templete-uri
- Standard Template Library (I)
- Standard Template Library (II)
- Dezvoltarea de aplicații orientate obiect. Princupii POO
- Noutăți introduse de C++11, C++14

Bibliografie

- Bjarne Stroustrup – The C++ Programming Language 3rd Edition. Addison Wesley, 1997.
- Kris Jamsa, Lars Klander – Totul despre C şi C++. Manualul fundamental de programare în C şi C++, Teora
- Muşlea Ionuţ – Iniţiere în C++. Programare orientată pe obiecte. Cluj-Napoca, Microinformatica, 1993
- Andrei Alexandrescu - Programarea moderna in C++, Teora, 2002
- Octavian Catrina, Iuliana Cojocaru – Turbo C++, Teora, 1992
- N. A. Solter, S. J. Kleper - Proffesional C++, Wiley 2005 (http://edc.tversu.ru/elib/inf/0146.pdf)
- Bruce Eckel - Thinking in C++ Vol I (http://www.lib.ru.ac.th/download/ebooks/TIC2Vone.pdf)
- Bruce Eckel - Thinking in C++ Vol II (http://www.lib.ru.ac.th/download/ebooks/Tic2Vtwo.pdf)

Paradigme de programare. Scurt istoric OOP și C++

Curs 1

Cuprins

- Tehnici de programare
- Programarea orientată obiect
- Istoric C++

Cuprins

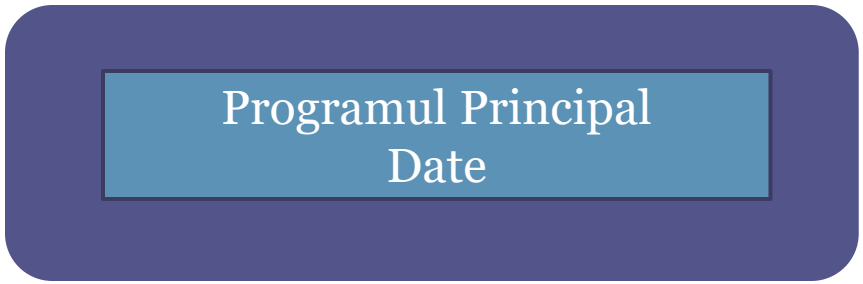
- Tehnici de programare
- Programarea orientată obiect
- Istoric C++

Tehnici de programare

- Programarea nestructurată
- Programarea procedurală
- Programarea modulară
- Abstractizarea datelor
- Programarea orientată obiect
- Programarea generică
- Programarea orientată pe aspecte

Programarea nestructurată

- Programe simple / mici ca dimensiune care conţin doar o singură metodă
- Program = succesiune de comenzi care modifică date globale
- Dezavantaje
  - Greu de întreţinut cu când codul devine mai lung
  - Mult cod duplicat (copy/paste)
- Exemple: programe scrise în: asamblare, limbajul C, limbajul Pascal

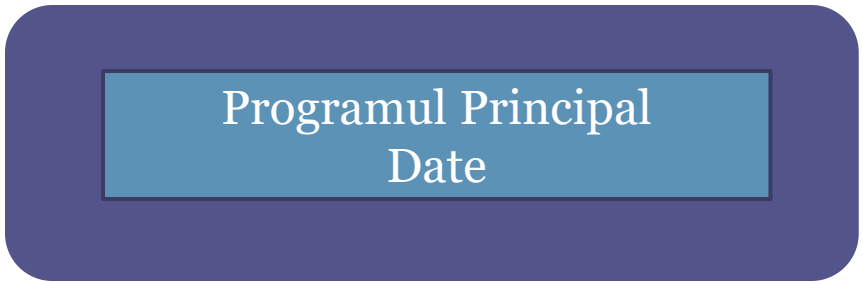



```
test.c
//declaţii date
int main (int argc, char* argv[] ) {
  // declaraţii date locale
  // instrucţiuni
}
```

Programarea nestructurată

- Programe simple / mici ca dimensiune care conţin doar o singură metodă
- Program = succesiune de comenzi care modifică date globale
- Dezavantaje
  - Greu de întreţinut cu când codul devine mai lung
  - Mult cod duplicat (copy/paste)
- Exemple: programe scrise în: asamblare, limbajul C, limbajul Pascal

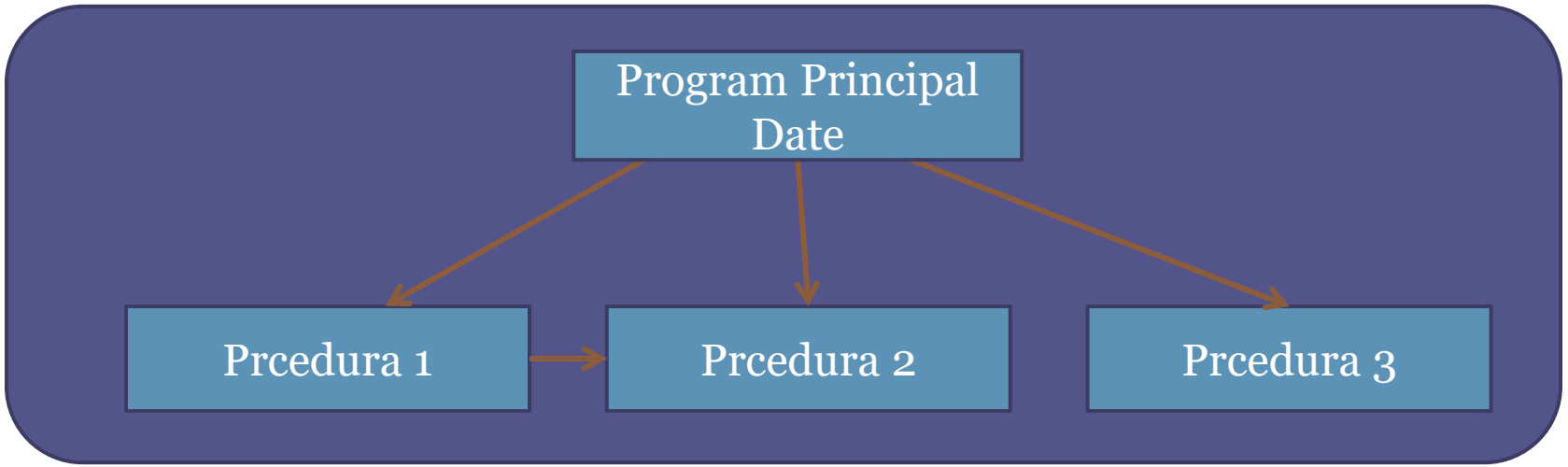
Care ar fi soluţia?




```
test.c
//declaţii date
int main (int argc, char* argv[] ) {
  // declaraţii date locale
  // instrucţiuni
}
```

Programarea procedurală

- Se bazează pe noțiunea de procedură (funcție)
- Procedura stochează algoritmul pe care dorin sa îl (re)folosim
- Dezavantaje
  - Menţinerea de diferite stucturi de date şi algoritmi care prelucrează datele
- Exemple: programe scrise în C, Pascal, Fortran, Algol



Programarea procedurală

- Se bazează pe noțiunea de procedură (funcție)
- Dezavantaje
  - Menţinerea de diferite stucturi de date şi algoritmi care prelucrează datele

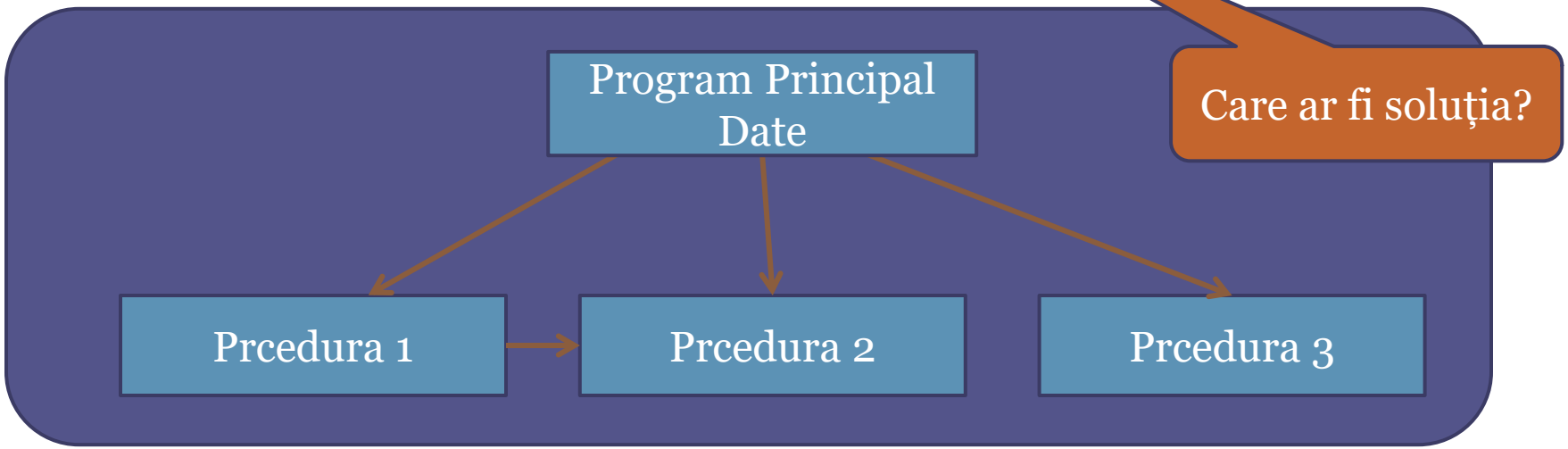
test.c

```
double sqrt(double arg)  { ... }
void f(double x, double y) {  ... sqrt(y); ... }

int main (int argc, char* argv[] ) {
    ...
    sqrt (123); f(7,8);
}
```

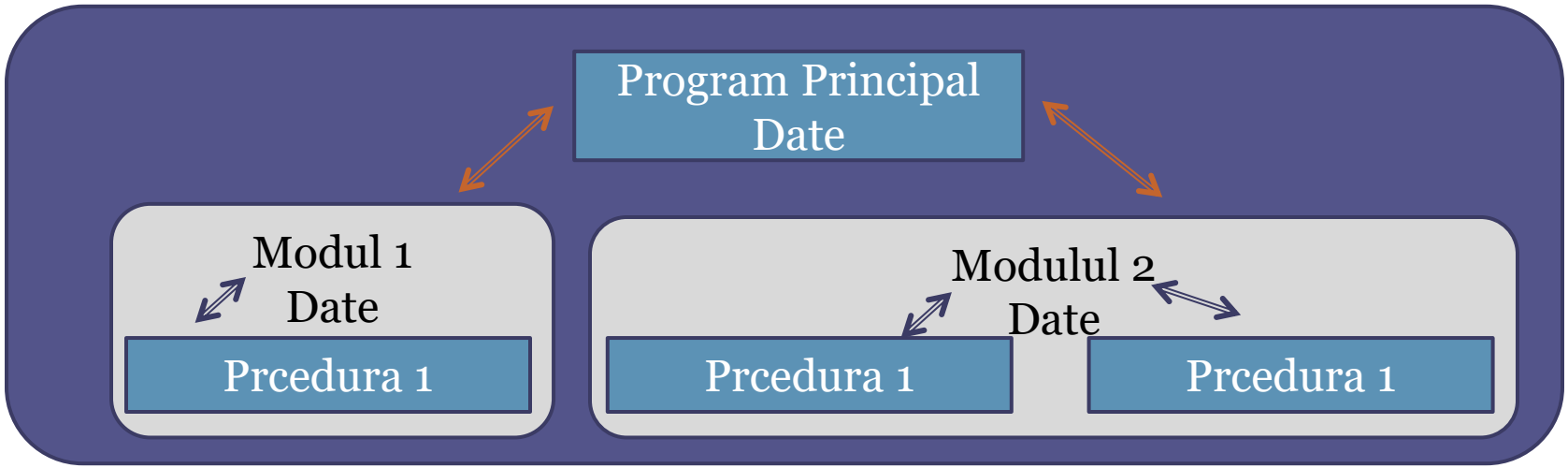
Programarea procedurală

- Se bazează pe noțiunea de procedură (funcție)
- Procedura stochează algoritmul pe care dorin sa îl (re)folosim
- Dezavantaje
  - Menţinerea de diferite stucturi de date şi algoritmi care prelucrează datele
- Exemple: programe scrise în C, Pascal, Fortran, Algol



Programarea modulară

- Dimensiunea programului creşte → Organizarea datelor
- Ce module dorim; partiţionarea programelor astfel încât datele să fie ascunse în module (data hiding principle)
- Dezavantaje
  - Doar un singur modul există o dată într-un program
- Exemple: programe scrise în C, Modula-2



Programarea modulară

- Dimensiunea programului creşte → Organizarea datelor
- Dezavantaje
  - Doar un singur modul există o dată într-un program

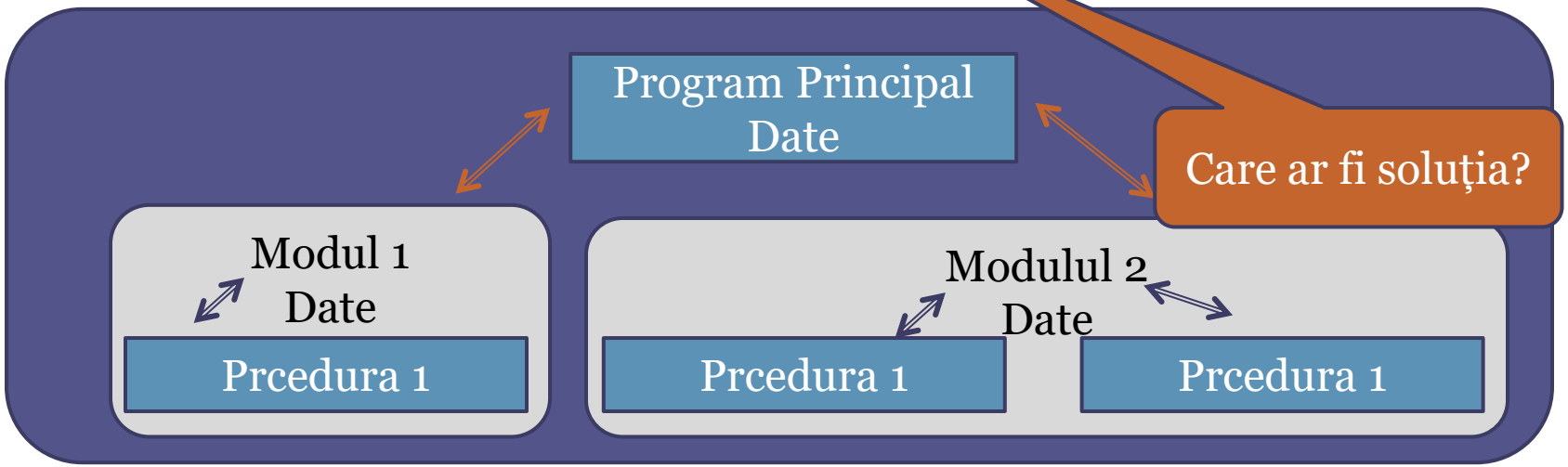
stiva.h
```
// declarea interfeţei modulului
char pop();
void push(char);
const dim_stiva= 100;
```

main.c
```
#include „stiva.h"
void functie() {
  push('c');
  char c = pop();
  if (c != 'c') error("imposibil");
}
```

stiva.c
```
#include "siva.h"
// "static" – local acestui fişier / modul
static char v[dim_stiva];
static char* p = v; // stiva este iniţial goală
char pop() {
 //  extrage element
}
void push(char c) {
 // adaugă element
}
```

Programarea modulară

- Dimensiunea programului creşte → Organizarea datelor
- Ce module dorim; partiţionarea programelor astfel încât datele să fie ascunse în module (data hiding principle)
- Dezavantaje
  - Doar un singur modul există o dată într-un program
- Exemple: programe scrise în C, Modula-2



Abstractizarea datelor

- Realizarea de tipuri de date definite de utilizator care se comportă ca şi tipurile default (build-in) (Abstract Data Types)
- Ce tipuri de date avem nevoie; implementarea unui set de operaţii pentru ele
- Dezavantaje
  - Imposibilitatea de a adapta abstractizările la noile tipuri, fără a modifica definiţia (are nevoie de „câmpuri de tip” pentru a face diferenţa între diferite instanţe)

complex.h

```
class complex {
   double re, im;
public:
   complex(double r, double i) { re=r; im=i; }
   // float->complex conversie
   complex(double r) { re=r; im=0; }
   friend complex operator+(complex, complex);
   // binar minus
   friend complex operator-(complex, complex);
};
```

main.c

```
void f() {
   int ia = 2,ib = 1/a;
   complex a = 2.3;
   complex b = 1/a;
   complex c = a+b*complex(1,2.3);

 c = -(a/b)+2;
}
```

Abstractizarea datelor

- Dezavantaje
  - Imposibilitatea de a adapta abstractizările la noile tipuri, fără a modifica definiţia (are nevoie de „câmpuri de tip” pentru a face diferenţa între diferite instanţe)

figura.h

```
enum tip{ cerc, triunghi, patrat};
class figura {
  punct centru;
  culoare col;
   tip k;
 // reprezentarea figurii
public:
    punct unde() { return centru; }
    void muta(punct to) { centru= to; deseneza(); }
    void deseneaza();
};
```

figura.cpp

```
void figura::deseneaza() {
   switch (k) {
      case cerc: // deseneaza cerc
          break;
      case triunghi: // deseneaza    triunghi
          break;
      case dreptunghi:
          // deseneaza dreptunghi
         break;
      default: // figura nedefinita
   }
}
```

Abstractizarea datelor

Care ar fi soluţia?

- Dezavantaje
  - Imposibilitatea de a adapta abstractizările la noile tipuri, fără a modifica definiţia (are nevoie de „câmpuri de tip” pentru a face diferenţa între diferite instanţe)

figura.h

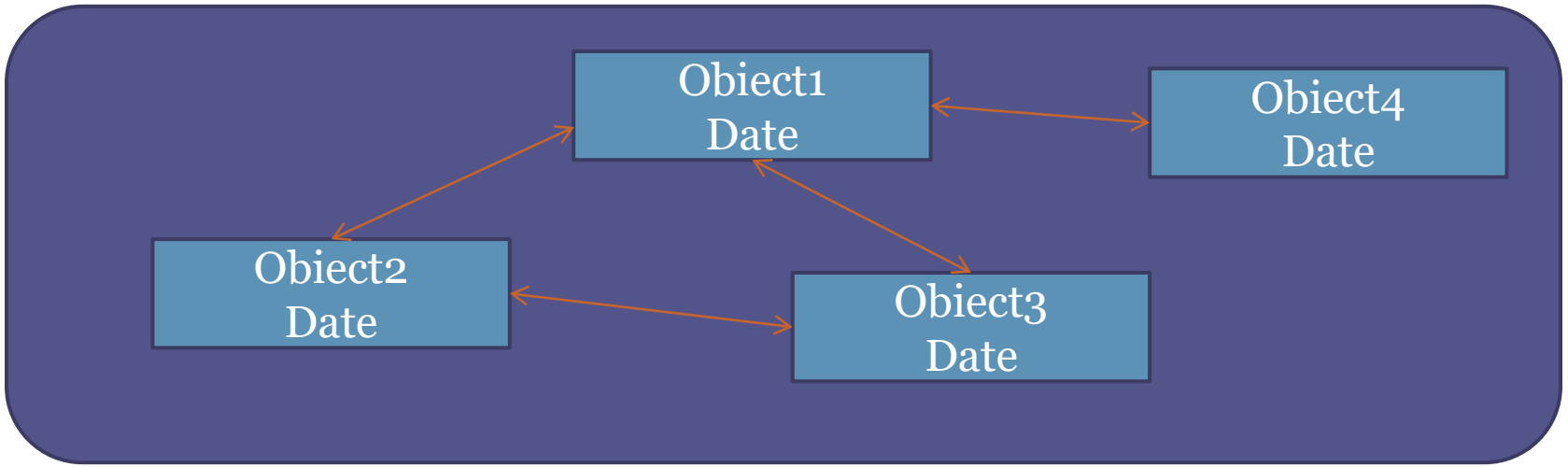
```
enum tip{ cerc, triunghi, patrat};
class figura {
   punct centru;
   culoare col;
    tip k;
 // reprezentarea figurii
public:
     punct unde() { return centru; }
     void muta(punct to) { centru= to; deseneza(); }
     void deseneaza();
};
```

figura.cpp

```
void figura::deseneaza() {
    switch (k) {
       case cerc: // deseneaza cerc
            break;
       case triunghi: // deseneaza    triunghi
            break;
       case dreptunghi:
            // deseneaza dreptunghi
           break;
       default: // figura nedefinita
    }
}
```

Programarea orientată obiect

- Obiecte care interacţionează, fiecare gestionând starea proprie
- Ce clase avem nevoie; definirea unei mulţimi de operaţii, utilizarea moştenirii pentru extragerea comportamentului comun
- Exemple: programe scrise în Simula, C++, Java, Eiffel, Smalltack, etc



Programarea orientată obiect

- Obiecte care interacţionează, fiecare gestionând starea proprie
- Ce clase avem nevoie; definirea unei mulţimi de operaţii, utilizarea moştenirii pentru extragerea comportamentului comun
- Exemple: programe scrise în Simula, C++, Java, Eiffel, Smalltack, etc

figura.h

```
class figura{
    punct centru;
    culoare col;
   // reprezentarea figurii
public:
    punct unde() { return centru; }
    void muta(punct to) { centru= to; deseneza(); }
    void deseneaza();
};
```

rectangle.h

```
class dreptungi: public figura{
     double înălţime, lungime;
      // reprezentarea dreptunghiului
public:
      void deseneaza() {
        // deseneaza dreptunghi
     }
};
```

Programarea generică

- Algoritmii independenţi de detaliile de reprezentare
- Ce algoritm se vrea; parametrizare astfel încât să funcţioneze cu o mulţime de date şi structuri de date potrivite
- Exemple: programe scrise în C++, Java (≥1.5)

stiva.h

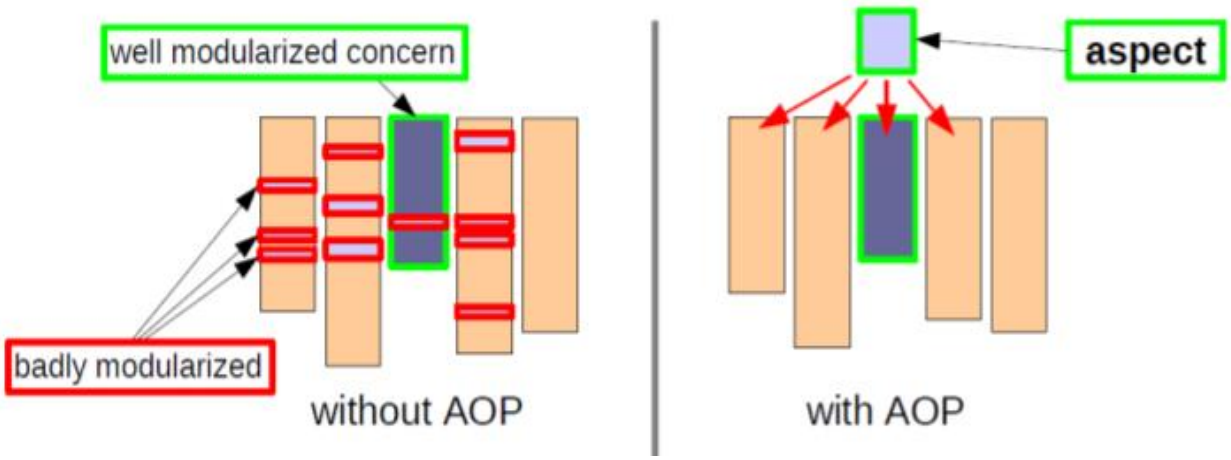
```
template<class T> class stiva {
        T* v;
        int dim_max, top;
public:
       stiva(int s);
      ~stiva();
      void push(T v);
      T pop();
};
```

fisier.cpp

```
void f() {
      stiva<char> schar;
      stiva<complex> scomplex;
      stiva<list<int>> slistint;
      schar.push('c');
      if(schar.pop()!='c') throw Impossible();
      scomplex.push(complex(3, 2));
}
```

Programarea orientată spre aspecte

- Programarea orientată spre aspecte (Aspect Oriented Programming)
- Obiective: separarea problemelor comune diferitelor module
- Este un utilitar care ajută la rezolvarea unor probleme a programării orientate obiect (nu un înlocuitor a cesteia)
- Izolează funcţiile sau logica secundară de logica de business a programului principal (modele de logging, autentificare, managementul erorilor, ...)



Cuprins

- Tehnici de programare
- Programarea orientată obiect
- Istoric C++

Programarea orientată obiect

- Ce este programare orientată obiect (POO)?
- Istoric programare orientată obiect
- Ce este C++?

POO - Definiţii

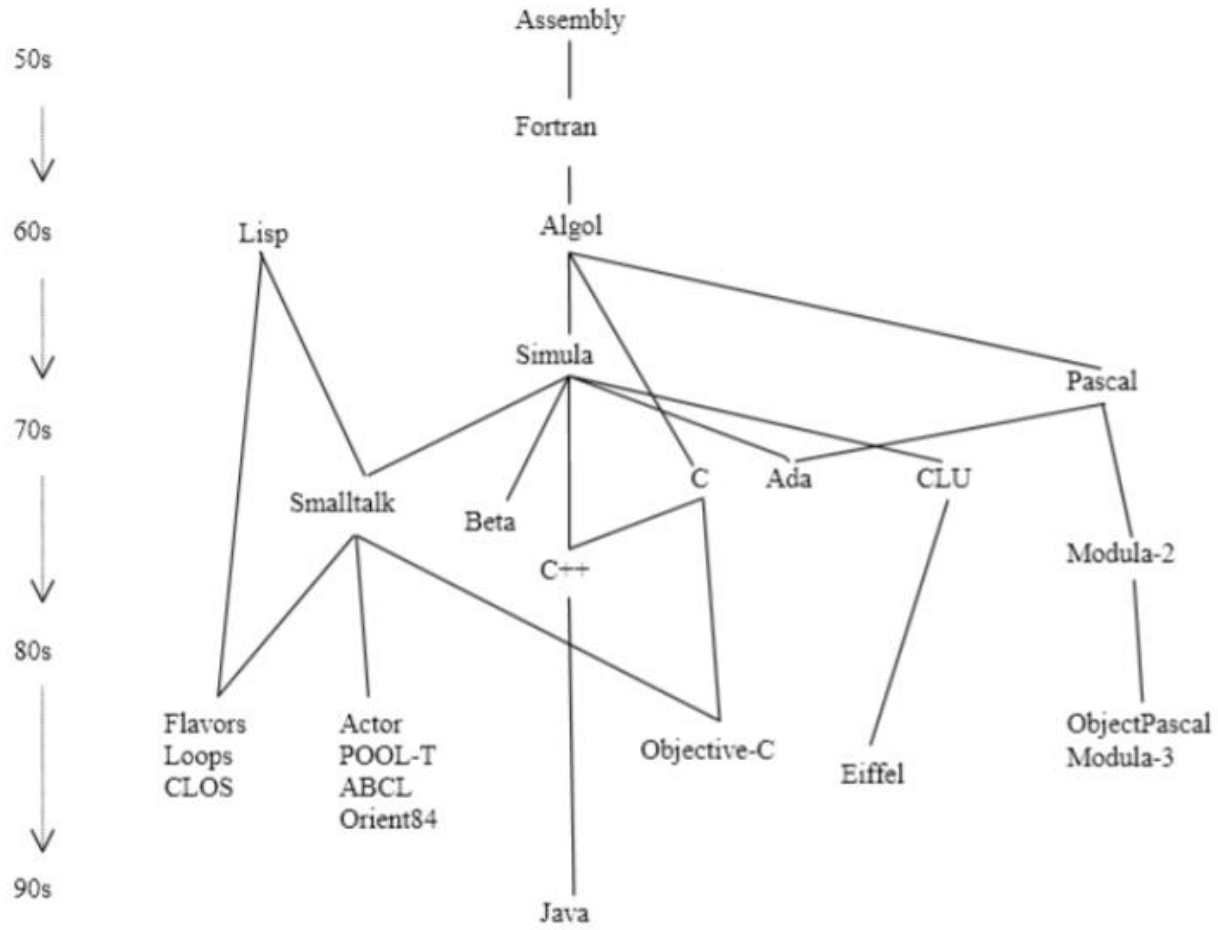
- Definiţie programare orientată obiect (Object OrientedProgramming)
  - Un limbaj sau o tehnică este orientat obiect dacă şi numai dacă îndeplineşte următoarele condiţiile [Stroustrup, 1995]
    - Abstractizare – oferă clase şi obiecte
    - Moştenirea – posibilitatea de a construi noi abstractizări peste cele existente
    - Polimorfism la execuţie – oferirea unui mecanism de legare dinamică (runtime binding)

- Definiţe generală pentru limbajele care sunt caracterizate ca fiind orientate obiet: Ada95, Beta, C++, Java, CLOS, Eiffel, Simula, Smalltalk şi altele care se încadrează în această definiţie.
  - Limbaje care nu suportă clase ca C, Fortran4, Pascal sunt excluse.
  - Limbaje care nu au suport pentru moştenire sau legare dinamică ca Ada88 sau ML sunt excluse.

Istoric POO

- Simula 67 –primul limbaj de programare orientat obiect; extensie a limbajului ALGOL60
- Smalltalk – realizat de Alan Kay (Smalltalk-72, Smalltalk-80); legarea dinamică; Strongtalk (1993) – Smalltalk + type system
- ani 80 – multe limbaje au adăugat suport pentru programarea OO: Objective C, C++, Object Pascal, Modula 3, Oberon, Objective CAML,CLOS.
- Eiffel – Bertrand Meyer (1988) – sintaxă asemănătoare cu Pascal, modelarea prin contract (design-by-contract)
- Alte limbaje “exotice” OO: Sather, Trellis/Owl, Emerald, Beta (versiune evoluata de Simula), Self
- Java – James Gosling (1995); Java 1.5 (2004) – oferă suport pentru programarea generică

Evoluția limbajelor



Ce este C++?

- Definiţie 1
  - C + + este un limbaj de programare cu scop general, cu o înclinație spre sisteme de programare care suportă eficient calcule de nivel scăzut, abstractizare a datelor, programarea orientată pe obiecte și programarea generică. [Stroustrup, 1999]

- Definiţie 2
  - C + + este un limbaj static (statically-typed) cu scop general bazându-se pe clase și funcții virtuale pentru a sprijini programarea orientată pe obiecte, template-uri pentru a sprijini programarea generice, precum și furnizarea de facilități de nivel scăzut pentru a sprijini sistemele de programare detaliate. [Stroustrup, 1996]

Idei de Proiectare C++

C++ a fost conceput pentru a accepta o gamă largă de stiluri bune şi folositoare. Indiferent dacă acestea au fost orientate-obiect [Stroustrup, 1995]:

1. Abstractizarea - capacitatea de a reprezenta concepte direct într-un program şi ascunde detaliile din spate (interfeţe bine definite) - este cheia pentru fiecare sistem flexibil şi uşor de înţeles de orice dimensiuni considerabile.
2. Încapsulare - capacitatea de a oferi garanţii că o abstractizare este utilizată numai în conformitate cu specificaţiile sale - este esenţial să se apere împotriva coruperi abstractizări.
3. Polimorfism - capacitatea de a oferi aceeaşi interfaţă pentru obiecte cu implementari diferite - este crucială pentru a simplifica codul folosind abstractizări.
4. Moştenirea - abilitatea de a compune abstracţiuni noi pornind de la una deja existent - este unul dintre cele mai puternice moduri de construcţie de abstractizări utile.
5. Genericele - capacitatea de parametrizare a tipurilor şi funcţiilor prin tipuri şi valori - este esenţială pentru crearea de tipuri sigure şi este un instrument puternic pentru a scrie algoritmi generali.
6. Coexistenţa cu alte limbi şi sisteme - esenţială pentru funcţionarea în medii de execuţie reale.
7. Compactitatea şi viteza la execuţie - esenţială pentru programare pe sisteme clasice.
8. Limbaj static – o familie de limbaje din care face parte şi C++ care asigură eficienţă de spaţiu şi la rulare a programelor

Cuprins

- Tehnici de programare
- Programarea orientată obiect
- Istoric C++

Istoric C++

- 1979 - C with Classes: Bjarne Stroustrup (AT&T Bell Labs) transpune conceptele (precum clase, moştenire) din Simula67 în C
- 1982 - From C with Classes to C++: prima variantă de C++ şi publicarea cărţii care defineşte limbajul C++ în Octombrie 1985
- 1985 - Versiunea 2.0: Evoluţia primei versiunii (publicarea cărţi The C++ Programming Language - Bjarne Stroustrup)
- 1988 – Standardele actuale: ISO şi ANSI
- 1994 - Standard Template Library
- 1998 - International C++ standard
- 2011 - un nou standard pentru C++11
- 2013 The C++ Programming Language, 4th edition
- 2014 C++14 mici modificări aduse standardului

Viitor C++

- 2017 C++17 un nou release al standardelor

Cursul următor

- Tipuri de date.
  - Tipuri de date concrete
  - Tipuri de date abstracte
  - Tipuri de date generice.
- Modelarea aplicaţiilor orientate obiect
  - Încapsulare
  - UML
- Diferențe/îmbunătățiri C++ față de C