Programare II
Programare Orientată Obiect

Cuprins

- Tehnici de programare
- C++ 11

Tehnici de programare

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

Cuprins

- Tehnici de programare
- C++ 11

Programarea nestructurată

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

Programul Principal
Date

```
test.c
//declații date
int main (int argc, char* argv[] ) {
  // declarații date locale
  // instrucțiuni
}
```

Programarea nestructurată

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

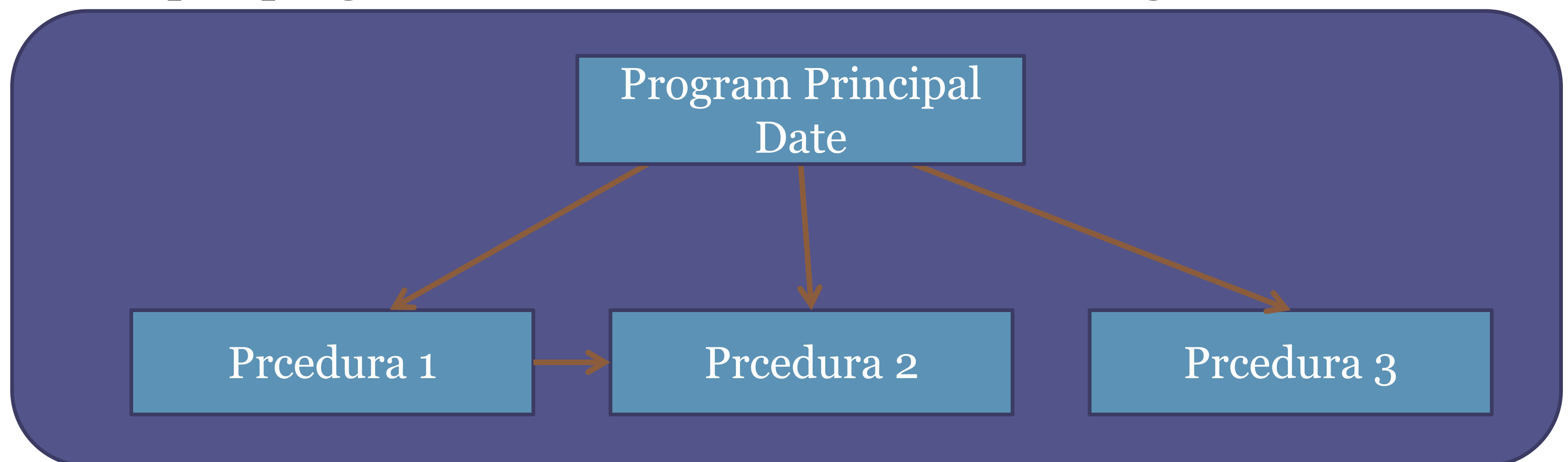
Care ar fi soluția?

Programul Principal
Date

```
test.c
//declații date
int main (int argc, char* argv[] ) {
  // declarații date locale
  // instrucțiuni
}
```

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

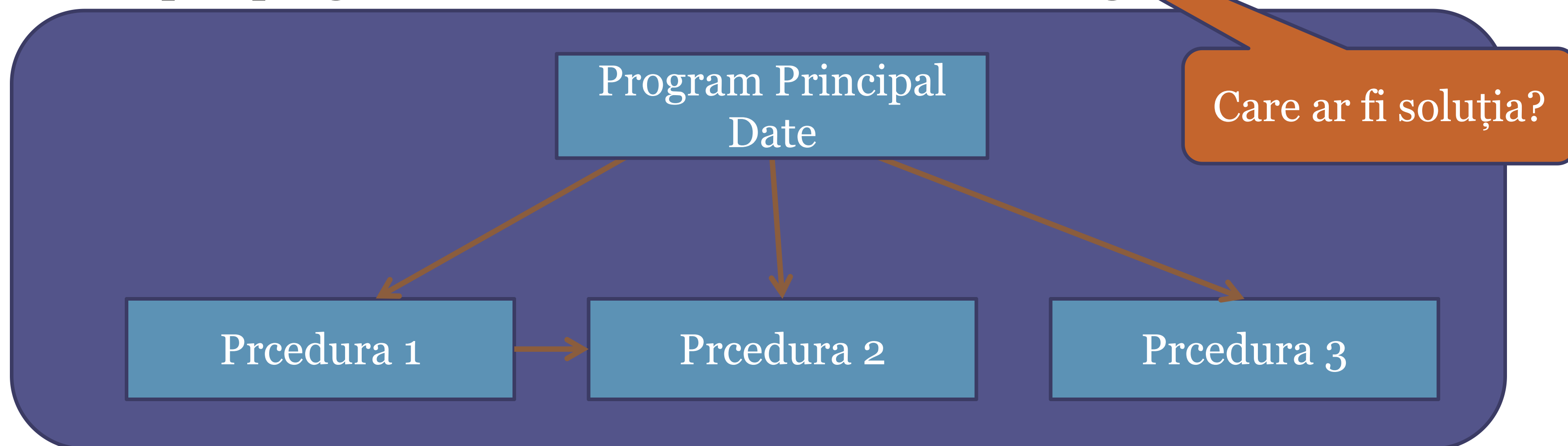
test.c

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

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

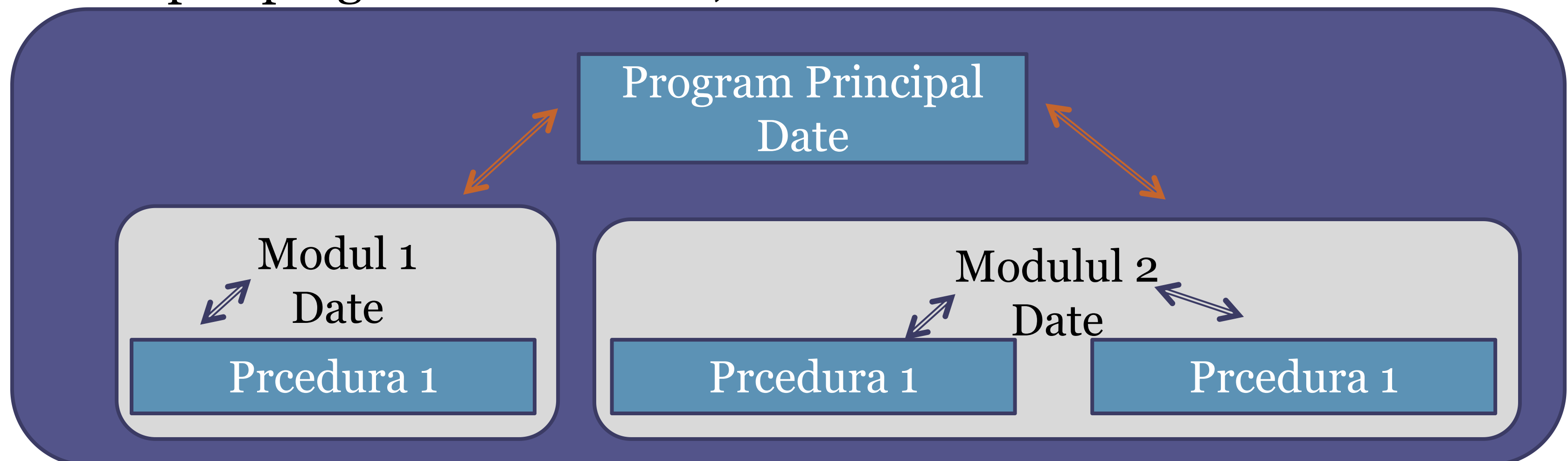
Programarea procedurală

- 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

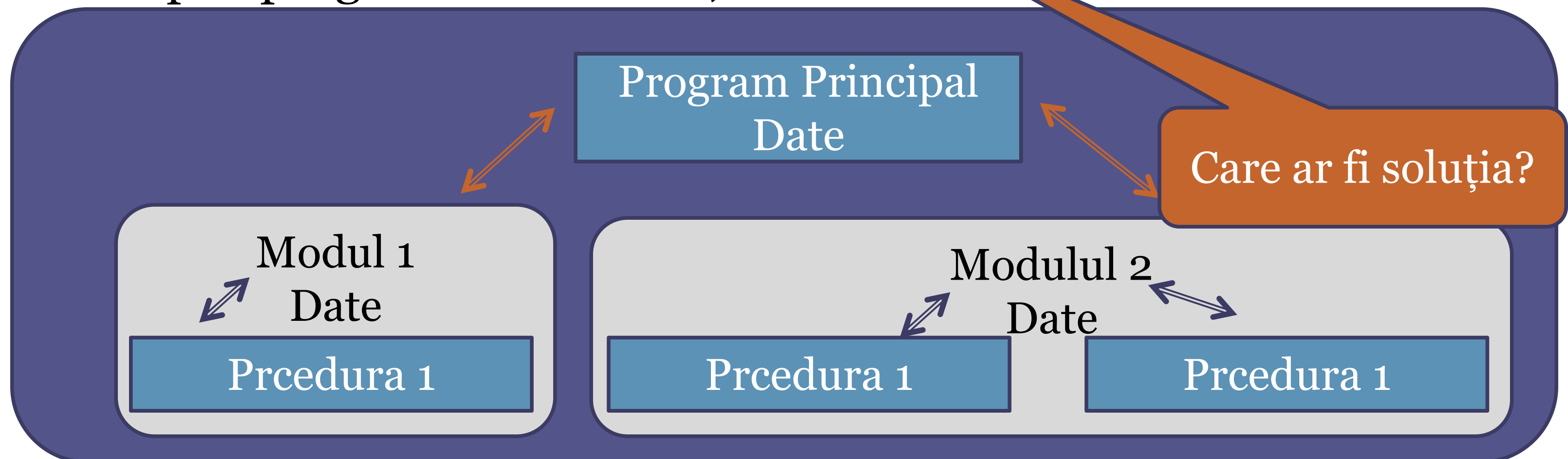
```
stiva.h
// declarea interfeţei modulului
char pop();
void push(char);
const dim_stiva= 100;
```

```
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; deseneaza(); }
    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 soluția?

- 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

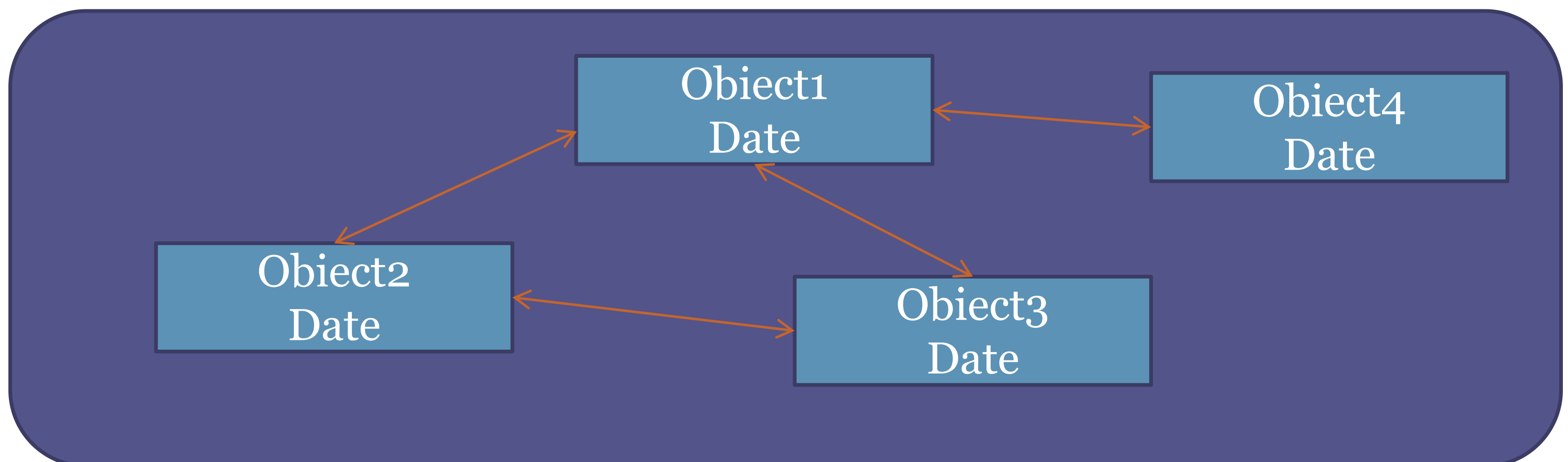
```
enum tip{ cerc, triunghi, patrat};
class figura {
  punct centru;
  culoare col;
   tip k;
 // reprezentarea figurii
public:
    punct unde() { return centru; }
    void muta(punct to) { centru= to; deseneaza(); }
    void deseneaza();
};
```

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; deseneaza(); }
   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

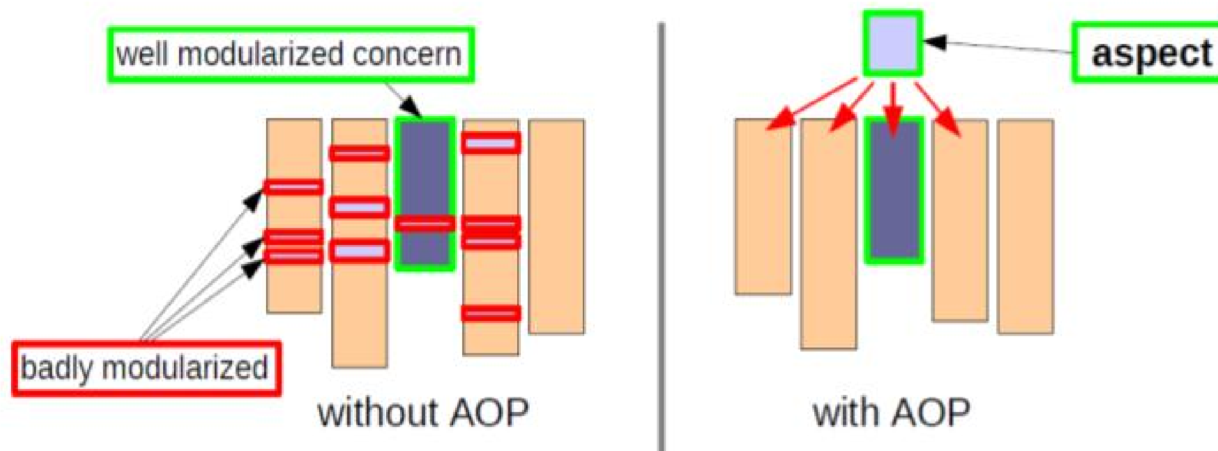
```
template<class T> class stiva {
        T* v;
        int dim_max, top;
public:
       stiva(int s);
      ~stiva();
      void push(T v);
      T pop();
};
```

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

- C++ 11
  - auto
  - for
  - Initializarea datelor
  - Specificatori override și final
  - Smart pointers

Deducerea tipului - auto

- Compilatorul determina tipul
  - auto myString = “my string”;
  - auto myInt = 6;
  - auto p1 = 3.14;
- Obținerea unui iterator la primul element al unui vector
  - vector v<int>;
  - vector<int>::iterator it1= v.begin(); // C++98
  - auto it2= v.begin(); //C++11

Deducerea tipului folosing - decltype

- Compilatorul determină tipul unei expresii
  - decltype("str") myString= "str"; // C++11
  - decltype(5) myInt= 5; // C++11
  - decltype(3.14) myFloat= 3.14; // C++11
  - decltype(myInt) myNewInt= 2011; // C++11

  - int add(int a,int b){ return a+b; };
  - decltype(add) myAdd= add; // (int)(*)(int, int)
  - myAdd(2,3)

Deducerea tipului de return a unei funcții

- Declarera unei funcții
  - template auto add(T1 first, T2 second) -> decltype(first + second){ return first + second; }
  - add(1,1);
  - add(1,1.1);
  - add(1000L,5);

- Rezultatul este de tip
  - int
  - double
  - long

Cicluri Range-Based

- Iterarea unui container se poate realiza astfel
  - std::vector<int> v;
  - ...
  - for (int i : v) std::cout << i;

- Se pot folosi și variabile referință
  - for (int& i : v) std::cout << ++i;

- Se poate folosi și auto
  - for (auto i : v) std::cout << i; // same as above for (auto& i : v) std::cout << ++i;

Inițializarea obiectelor – (), {}

- Inițializarea obiectelor se poate face cu paranteze rotunde, acolade sau egal
  - int x(0);
  - int x=0;
  - int x{0};
  - int x = {0}

Inițializarea obiectelor – (), {}

- Inițializarea containărelor
  - std::vector<int> v{ 1, 3, 5 };
  - vector<string> vec= {"Scott",st,"Sutter"};
  - unordered_map<string, int> um= {{"C++98",1998},{"C++11",i}};
- Inițializarea membrilor clasei

```
class Foo{
    ...
    private:
        int x{ 0 }; // OK, valoare implicita a lui x este zero
        int y = 0; // OK
        int z(0); // eroare!
};
```

Inițializarea obiectelor – (), {}

```
Class Foo{
public:
    Foo(int I, bool b);
    Foo(int I, double d);
};
```

- Creare obiecte
  - Foo w1; //apel constructor default
  - Foo w2=w1; //apel constructor de copiere
  - w1 = w2; //apel operator egal
  - Foo f1(10, true); //apel primul constructor
  - Foo f2{10, true}; //apel primul constructor
  - Foo f3{10, 5.5}; //apel al doilea constructor

Specificatori override si final

- Override – pentru a specifica că o metodă suprascrie o metodă a supraclasei
  - Este o indicație care spune compilatorului să verifice dacă există în supraclasă o metodă cu aceelași prototip

- Exemplu

```
class B {
    public: virtual void f(int) {std::cout << "B::f" << std::endl;}
};
class D : public B {
    public: virtual void f(int) const override {std::cout << "D::f" <<
       std::endl;}
};
```

Specificatori override si final

- Final – face metoda imposibil de suprascris

- Exemplu

```
class B {
    public: virtual void f(int) {std::cout << "B::f" << std::endl;}
};
class D : public B {
    public: virtual void f(int) override final {std::cout << "D::f" <<
       std::endl;}
};
class F : public D {
    public: virtual void f(int) override {std::cout << "F::f" << std::endl;}
};
```

Smart pointers

- Se comportă ca poineri default, dar gestionează obiectele pe care le conțin astfel încât nu mai trebuie să ne facem griji în privința dealocării
- Arată și se comportă ca poineri din C
  - Prin supraîncărcarea operatorilor *,->,[],...
- C++98
  - auto_ptr
    - Probleme cu transferul propritarului
    - Nu pot fi folosiți cu tablouri
    - Nu pot fi folosiți cu containăre partajate ca list, vector, map

Smart Pointers

- C++11
  - shared_ptr
    - Mai mulți pointeri pot pointa spre accelași obiect
  - unique_ptr
    - Un pointer deține exclusiv un obiect
  - weak_ptr
    - Pot partaja resurse dar nu le dețin