Programare orientată obiect

Cursul 1

rev 1.4/PP/20131020

Conținut

- Recapitulare, extensii C++
- Concepte POO, clase, obiecte
- Constructori, destructor și constructor de copiere
- Intrări/Ieșiri C++
- Supraîncărcarea operatorilor
- Moștenire și derivare
- Clase generice (template)
- Biblioteca standard de clase generice: STL (Standard Template Library)

Referințe bibliografice

- I. Smeureanu, M. Dârdală, Programarea orientată obiect în limbajul C++, Editura CISON, 2004
- Bjarne Stroustroup , The C++ Programming Language, 4th Edition, Addison-Wesley, 2013
- C++ Language Reference, MSDN
- Cursuri și seminarii
- Materiale disponibile online

Sumar

- Recapitulare
  - Apelul funcțiilor
  - Pointeri la funcții
  - Pointeri constanți
  - Preprocesare
- Elemente noi de limbaj (C++)
  - Alocarea memoriei și referințe
  - Domenii de nume
  - Sistemul de intrare-ieșire
  - Tratarea excepțiilor

Recapitulare

Apelul funcțiilor

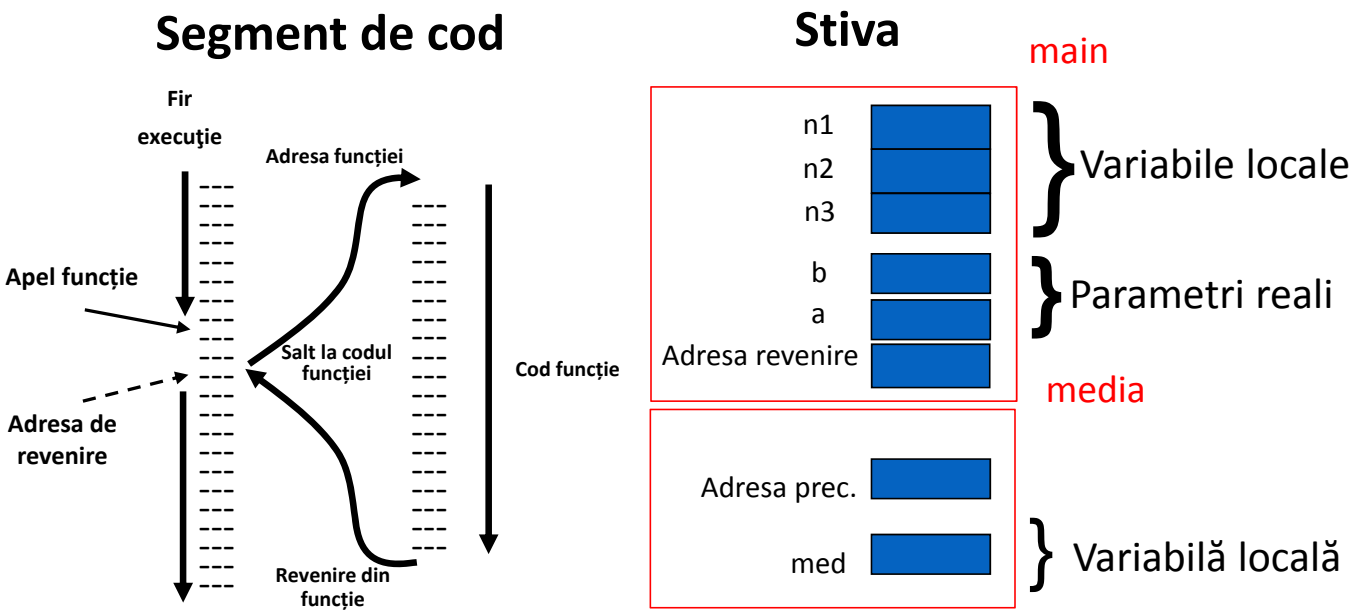
```
#include <stdio.h>

// prototipul funcției
float media(float, float);

void main()
{
	float n1, n2, n3;
	//initializeaza n1 si n2
	n3 = media(n1, n2);
	printf("media este: %f", n3);
}
```

```
//definirea functiei
float media(float a, float b)
{
	float med = (a + b)/2;
	return med;
}
```

Apelul funcțiilor



main() media(a, b)

Pointeri la funcții

- Numele unei funcții: pointer constant
  - adresa din memorie asociată codului executabil
- Se utilizează pentru transmiterea funcțiilor ca parametri altor funcții
- Componente:
  - Tipul parametrilor
  - Tipul returnat
- Se inițializează cu numele unei funcții
- Exemple de utilizare întîlnite:
  - Sortare: modalitatea de ordonare
  - Funcții cu apel invers (callback)
  - Diferite calcule/operații (implementate prin funcții cu același prototip) aplicate repetat pe aceleași seturi de date

Pointeri la funcții

- void (*__pf2__)()
  - Pointer la funcție fără parametri care nu returnează valoare
- bool (*__pf1__)(int, int)
  - Pointer la funcție cu doi parametri de tip int care returnează bool
- int (*__pf3__)(int)
  - Pointer la funcție cu un parametru de tip int care returnează int
- double (*__pf4[N]__)(double, double)
  - Vector de N pointeri la funcții cu doi parametri de tip double care returnează double

Constante de tip pointer

- Pointer constant:
  - Tip * **const** p = adresa;
- Pointer la o zonă de memorie constantă
  - Tip **const** * p;
  - **const** Tip * p;
- Pointer constant la o zonă de memorie constantă
  - **const** Tip * **const** p = adresa;

Directive preprocesor

- Includere fișiere (#include)
- Constante simbolice/macrodefiniții (#define, #undef)
- Compilare condiționată (#ifdef, #ifndef, #else, #endif)
- Erori (#error)
- Opțiuni compilare (#pragma)
- Controlul liniilor de cod la compilare (#line)

Operatori preprocesare

- # - transformare în șir de caractere
- ## concatenare parametri

Elemente noi de limbaj (C++)

Alocarea memoriei

- Operatorii **new** și **delete**
- Referințe

Alocarea memoriei

- Operatorul **new**
- Alocare fără inițializare
  - int *pi = new int;
- Alocarea cu inițializare
  - int *pi = new int(3);
- Alocare masiv unidimensional
  - int *pv = new int[10];
- De verificat rezultatul alocării!

Eliberarea memoriei

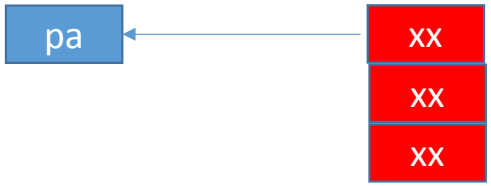
- Operatorul **delete**
- Eliberare memorie element
  - delete pi;
- Eliberare memorie masiv;
  - delete [] pv;
- **malloc()/free()** și **new/delete**!

Alocare dinamică - Probleme posibile

- Referirea unor zone de memorie protejate
- Referirea unor zone de memorie eliberate (*dangling pointers*)
- Neeliberarea memoriei (*memory leaks*)
- Încercarea de eliberare a memoriei eliberată anterior

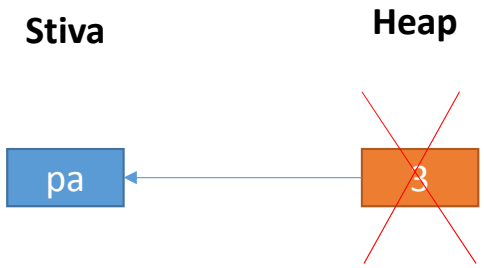
Alocarea memoriei

```
…
int *pa = NULL;
…
*pa = 20;
```



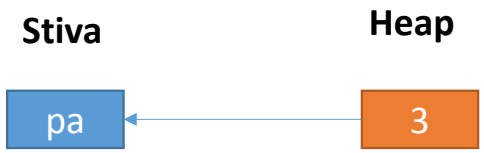
Alocarea memoriei

```
…
int *pa = new int(3);
…
delete pa;
…
*pa = 20;
```



Alocarea memoriei

```
…
{
        int *pa = new int(3);
        …
}
```

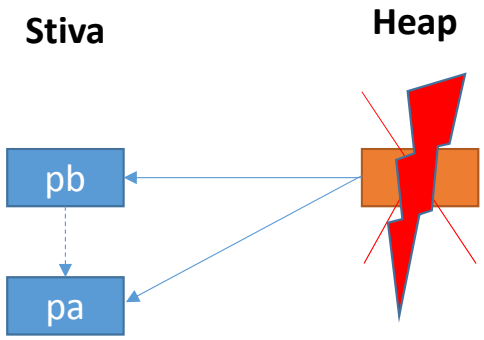


Alocarea memoriei

```
...
int *pa, *pb = new int(3);

pa = pb;

delete pa;
delete pb;
```



Referințe

- Stochează adresa unei variabile
  - Alias pentru variabila asociată
- Se inițializează la definire
- Nu pot fi modificate ulterior
- Se dereferențiază automat
- Utile în transmiterea parametrilor funcțiilor

Referințe

- Tip & ref = varDefinitaAnterior;
- Exemplu:

```
int x = 20;
int & rx = x;
rx = 15;
```

Referințe

- Funcție

```
void comuta(int &x, int &y)
{
        int aux = x;
        x = y;
        y = aux;
}
```

- Apel:

```
int a = 3, b = 10;
comuta(a, b);
```

Referințe

- Funcție:

```
int calcRadical(double x, double &rez)
{
        int codRet = 0; //in regula

        if (x < 0)
                codRet = -1;//eroare
        else
                rez = sqrt(x);
        return codRet;
}
```

- Apel:

```
double a = 3, rad;

int cod = calcRadical(a, rad);
```

Referințe

- Funcție:

```
int & inc(int &x)
{
        return ++x;
}
```

- Apel:

```
int a = 3;
inc(a)+=10;
```

Domenii de nume

- Posibilitatea de grupare a tipurilor definite de utilizator
- Evitarea conflictelor de nume
- Definirea unui domeniu de nume:

  namespace nume_domeniu
  {
  }
- Referirea membrilor dintr-un domeniu:
  - **using namespace** nume_domeniu;

  sau
  - nume_domeniu**::**membru_domeniu;

Sistemul de intrare-ieșire C++

- Orientat de fluxuri de date (stream)
- Variabile inițializate global
  - **cin** – intrare
  - **cout** – ieșire
  - **cerr** – eroare (ieșire)
  - etc.
- Operatori
  - **>>** (intrare) – extracție din flux
  - **<<** (ieșire) – inserție în flux
- Se utilizează:
  - fișierul antet *iostream*: **include <iostream>**
  - domeniul de nume *std*: **using namespace std;**

  sau
  - domeniul de nume *std* și operatorul de rezoluție: **std::cout**, **std::cin** etc.

Tratarea excepțiilor

- Excepții – situații anormale la execuția programului
  - Împărțire la zero
  - Depășire
  - Accesarea unui element în afara limitelor unui masiv
  - Referirea ilegală a unei adrese de memorie
- Întreruperea execuției programului – dacă nu sînt tratate!
- **try/catch/throw:**
  - la apariția unei situații anormale în blocul **try** este transmisă (lansată) o variabilă (*excepție*) prin instrucțiunea **throw** (direct în blocul **try** sau de funcțiile apelate în acesta)
  - situațiile excepționale sînt tratate în blocul **catch** corespunzător excepției lansate

Tratarea excepțiilor

```
try
{
        //cod care poate lansa excepții (prin throw):
                - apeluri de funcții care pot lansa excepții
                - direct, prin utilizarea instrucțiunii throw
}
catch(tipExceptie varExceptie)
{
        //tratare excepție de tip tipExceptie
}
catch(tipExceptie1 varExceptie1)
{
        //tratare excepție de tip tipExceptie1
}
catch(…)
{
        //tratare excepție de orice alt tip
}
```

Exemplu excepție