

Seminarul 1

Introducere in C++.

Clase. Encapsulare. Constructori. Destructori

1 Introducere in C++

Un program in C++ are urmatoarea structura:

```
1 <directive-include>;
2
3 int main (<parameterii>) { // functia principala main
4     <instructiuni-program>
5     return <valoare>;
6 }
```

Executia oricarui program incepe cu functia **main**, deci fiecare program trebuie sa aiba o functie **main**. Fiecare instructiune a unui program scris in C++ trebuie sa se incheie cu **;**.

Functia principala **main** este o functie care intoarce un intreg, care reprezinta codul de iesire al procesului in urma unei executii. Daca nu se intoarce o valoare din main, implicit compilatorul intoarce valoarea **0**.

Exemplu:

```
1 #include <iostream>
2
3 int main () {
4     std::cout << "Hello world";
5     return 0;
6 }
```

Tipuri de date si variabile

In C++ avem multiple tipuri de date predefinite pentru manipularea datelor usuale (Figura 1). Tipurile de date sunt folosite pentru a declara variabile:

```
1 <tip-date> <nume-variabile> [= <valoare>];
```

Un nume de variabile din C++ este o secventa de una sau mai multe litere, cifre sau **_**. Un nume de variabila incepe intotdeauna cu o litera (sau **_**). Spatiile sau alte caractere speciale nu pot fi

folosite in numele unei variabile. Numele de variabile sunt case sensitive (e.g. variabila `var` este diferita de variabila `Var`)

Exemplu:

```
1 int x = 2;
2 float f = 22.4453;
3 char c1, c2, c3 = 'C';
```

short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	8	-2,147,483,648 to 2,147,483,647
unsigned long int	8	0 to 4,294,967,295
long long int	8	-(2 ⁶³) to (2 ⁶³)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	

Figure 1: Tipuri de date primitive in C++

Citirea si afisarea datelor

Pentru citirea si afisarea datelor se folosesc operatorii `<<` si `>>` . Citirea si afisarea in C++ se fac prin intermediul fluxurilor de date (stream). In C++ avem fluxul standard de intrare `cin` si fluxul standard de iesire `cout` pentru citirea si afisarea la tastatura.

```
1 int i;
2
3 cin >> i; // citeste i de la tastatura
4 cout << "i = " << i;
5
```

```

6 float f1, f2, f3;
7
8 cin >> f1 >> f2 >> f3; // se citesc 3 numere de tip float de la tastatura
9 cout << "suma = " << f1 + f2 + f3;

```

Pentru citirea si afisarea in/din fisier se folosesc fluxurile de tip `ifstream` si `ofstream`.

```

1 #include <iostream> // fluxurile ofstream si ifstream se gasesc in headerul iostream
2
3
4 ifstream fin("data.in");
5 ofstream out("data.out");
6
7 int main () {
8     int i;
9
10    fin >> i; // citeste i din fisier
11    fout << "i = " << i;
12
13    float f1, f2, f3;
14
15    fin >> f1 >> f2 >> f3; // se citesc 3 numere de tip float din fisier
16    fout << "suma = " << f1 + f2 + f3;
17
18    return 0;
19 }

```

Instructiunea conditionala

Instructiunea conditionala `if-else` ne ajuta sa executam niste instructiuni doar daca o anumita conditie este indeplinita

```

1 if (<conditie>) {
2     <instructiuni executate conditie adevarata>
3 } [ else {
4     <instructiuni executate conditie falsa>
5 }]

```

Putem ca in blocul de instructiuni executat atunci cand conditia este indeplinita/respinsa sa avem alte instructiuni `if-else`. In acest caz asocierile intre dintre blocurile `if` si `else` se face dupa regula: fiecare `else` se asociaza cu primul `if` neasociat inca.

```

1 if (i != 0)
2     if (i > 0)
3         cout << "pozitiv";
4     else
5         cout << "negativ"
6 else
7     cout << "nul";

```

Instructiunea de selectie multipla

In cazul in care avem de verificat o valoare intr-o lista de valori posibile, folosirea instructiunii `if-else` s-ar putea sa duca la un cod nu foarte usor de folosit. Pentru o mai buna gestionare se

poate folosi instructiunea de selectie multipla:

```
1 switch (<expresie>)
2 {
3     case <val-constatant-1>:
4         <instructiuni-1>;
5         break;
6     case <val-constatant-2>:
7         <instructiuni-2>;
8         break;
9     .
10    .
11    .
12    [ default :
13        <instructiuni-caz-default >;]
14 }
```

Odata evaluata expresia, se va executa grupul de instructiuni asociat cu clauza **case** in care este mentionata valoare expresiei. Daca nici o clauza **case** nu este executata atunci se executa grupul de instructiuni asociat cu clauza **default**, daca exista

```
1 switch (x) {
2     case 1:
3         cout << "x este 1";
4         break;
5     case 2:
6         cout << "x este 2";
7         break;
8     default:
9         cout << "nu se cunoaste x";
10 }
```

Cicluri

In C++ avem mai multe tipuri de cicluri pentru a putea executa operatii iterativ:

Ciclul **while** :

```
1 while (<conditie-continuation>) {
2     <instructiuni>
3 }
```

Executa **<instructiuni>** cat timp **<conditie-continuation>** este evaluata la o valoare true. Pentru ca **<conditie-continuation>** este evaluata inainte de a executa **<instructiuni>** e posibil ca blocul **<instructiuni>** sa nu se execute niciodata.

Ciclul **do-while** :

```
1 do {
2     <instructiuni>
3 } while (<conditie-continuation>);
```

Executa **<instructiuni>** cat timp **<conditie-continuation>** este evaluata la o valoare true. Blocul **instructiuni** este evaluat inainte evaluarii **<conditie oprire>**, deci **instructiuni** este executat cel putin o data.

Ciclul **for** :

```
1 for (<initializari>; <conditie-continuatione>; <incrementari>) {  
2     <instructiuni>  
3 }
```

Executa **<initializari>** , dupa executa **instructiuni** cat timp **<conditie-continuatione>** este evaluata la o valoare true. La sfarsitul fiecarei iterarii se executa **<incremenetari>** . Similar cu ciclul **while** , **<conditie-continuatione** este evaluata inainte de **instructiuni** .

Una sau mai multe componente pot lipsi, inasa **;** trebuie sa fie prezente de fiecare data.

Funcții

Funcțiile sunt grupuri de instructini care au asociat un nume si care pot fi apelate (executate) de oriunde din program. Funcțiile (prodeduri) ne ajuta sa ne structuram codul.

```
1 <tip-retur> <nume> ([<tip-1> <param-1>, <tip-2> <param-2>, ... <tip-n> <param-n>]) {  
2     <instructiuni>;  
3 }
```

Lista de parametrii a unei functii poate contine un numar arbitrar de parametri. Fiecare parametru se specifica prin tipul de date si numele parametrului.

```
1 int suma (int a, int b) {  
2     int s = a + b;  
3     return s;  
4 }  
5  
6 int main () {  
7     int i = 52, j = 44;  
8     int k = suma(i, j);  
9 }
```

Daca o functie intoarce orice tip de date diferit de **void** , atunci fiecare ramura de executie a functiei trebuie sa intoarca o valoare. Funcțiile care intorc **void** nu necesita folosirea cuvântului **return** .

Parametrii pot fi transmisi catre o functie in doua moduri:

- prin valoare – intainte de apel se face o copie a parameterului care este folosita pe toata durata executiei functiei. La intoarcerea din apel valoarea parametrului ramane neschimbata: orice modificare efectuata pe parametru in corpul functiei nu este vazuta dupa finalizarea apelului.
- prin referinta – functia este apelata direct cu variabila pasata ca parametru si orice modificare facuta in corpul functiei pe acel parametru este vizibil dupa revenirea din apel. Valorile constante nu pot fi pasate prin referinta catre o functie. Un parametru este trimis prin referinta daca are **&** inainte de numele parameterului.

```
1 void swapVal (int a, int b) {  
2     int c = a;  
3     a = b;  
4     b = c;  
5 }
```

```

6
7 void swapRef (int &a, int &b) {
8     int c = a;
9     a = b;
10    b = c;
11 }
12
13 int main () {
14     int i = 5, j = 10;
15     swapVal(i, j);
16     cout << i << " " << j; // afiseaza '5 10'
17     swapRef(i, j);
18     cout << i << " " << j; // afiseaza '10 5'
19 }

```

Tablouri

Un tablou (array) reprezinta o serie de elemente de acelasi tip, plasate in consecutiv in memorie (un bloc de memorie). Un tablou se declara in felul urmatoar:

```

1 <tip de date> <nume>[<dimensiune>];

```

Pentru a initializa un tablou avem:

```

1 int v1[5] = {1, 2, 3, 4, 5};
2 int v2[] = {1, 2, 3, 4, 5}; // dimensiunea este dedusa din lista de valori.
3 int v3[] {1, 2, 3, 4, 5};

```

Intr-un tabou fiecare element are un index. Primul element din tablou are indexul 0 iar ultimul element din tablou are indexul **<dimensiune> - 1**. Pentru a putea accesa elementul cu indexul **i** trebuie sa scriem **<nume-tablou>[i]**.

```

1 int v[5] = {10, 20, 30, 40, 50};
2 cout << v[3]; // afiseaza 40
3 v[1] = 22; // am modificat elementul de pe pozitia a doua din vector la valoarea 22

```

Pentru a putea pasa un tablou ca parametru trebuie sa declaram trebuie sa procedam o functie cu antetul **<tip-return> <nume functie> (<tip-date> <nume>[]);** iar la apelare sa scriem **<nume-functie>(<nume-tablou>);**. Exemplu:

```

1 #include <iostream>
2 using namespace std;
3
4 void print (int v[], int n) {
5     for (int i = 0; i < n; ++i) {
6         cout << v[i] << ' ';
7     }
8     cout << '\n';
9 }
10
11 int main ()
12 {
13     int w[3] = {5, 10, 15};
14     print (w,3);
15     return 0;
16 }

```

Pointeri

În C++ orice variabilă este declarată static, i.e. memoria este prealocată pentru acea variabilă la începutul execuției programului și este dealocată la sfârșitul zonei de vizibilitate a variabilei.

În cazul în care ne dorim să nu se întâmple acest lucru, putem folosi *pointeri* pentru a putea alocă memoria când avem nevoie de ea. *Un pointer este o adresă către o zonă din memorie.* Un pointer se poate declara în următorul fel:

```
1 <tip-data> * <nume-pointer>;
```

Când declarăm un pointer, compilatorul alocă doar 4 octeți necesari pentru a putea salva valoarea adresei de memorie. Pentru a putea alocă un pointer, folosim operatorul **new** :

```
1 <nume-pointer> = new <tip-date>;
```

sau, pentru a alocă un bloc de **<dimensiune>** locații de memorie consecutive:

```
1 <nume-pointer> = new <tip-date>[<dimensiune>];
```

Pentru a accesa valoarea de la adresa către care indică un pointer trebuie să scriem ***<nume-pointer>** . Pentru a accesa valoarea de pe poziția **i** din blocul de memorie adresă către care indică un pointer trebuie să scriem ***(<nume-pointer> + i)** sau **<nume-pointer>[i]** .

Pentru a dealoca o zonă de memorie folosim operatorul **delete** :

```
1 delete <nume-pointer>;
```

sau pentru a dealoca un bloc de memorie (locații consecutive):

```
1 delete [] <nume-pointer>;
```

Dacă folosim **delete** pe un pointer alocat cu **new[]** se va dealoca doar prima locație din blocul de memorie, restul locațiilor rămânând alocate.

Exemplu:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(int argc, const char * argv[]) {
6     int *n = new int;
7     int *v;
8
9     cin >> *n;
10
11     v = new int[*n];
12
13     for (int i = 0; i < *n; i++) {
14         cin >> *(v+i);
15     }
16
17     for (int i = 0; i < *n; i++) {
18         cout << v[i] << " ";
19     }
```

```

20
21     delete [] v;
22
23     return 0;
24 }

```

Structuri

În dezvoltarea unui program sunt situații în care tipurile de date predefinite nu sunt suficiente și trebuie să ne definim propriile tipuri de date. Pentru asta avem conceptul de structuri prin care putem defini noi tipuri de date:

```

1 struct [<nume-structura>] {
2     <tip-prop-1> <nume-prop-1>;
3     <tip-prop-2> <nume-prop-2>;
4     <tip-prop-3> <nume-prop-3>;
5     ...
6 } [<lista-variabile>];

```

Pentru a putea accesa o proprietate a unei structuri trebuie să se folosească operatorul `.`:

```

1 <nume-var-structura>.<nume-prop>;

```

Exemplu:

```

1 struct student {
2     char *nume, *prenume;
3     int varsta;
4 } s1, s2;
5
6 s1.prenume = "Florin"

```

Putem alocă variabile de tip structură dinamic:

```

1 student *s = new student;
2 *(s).prenume = "Florin";
3 s->prenume = "Florin";

```

Exercitii

1. Implementați un program care citește două numere întregi de la tastatură și afișează maximumul.
2. Implementați un program care citește de la tastatură un $n \geq 3$ și afișează al n -lea număr al lui Fibonacci.
3. Implementați un program care citește de la tastatură un vector de numere în virgulă mobilă și îl sortează în ordine crescătoare.
4. Implementați un program care citește de la tastatură un număr întreg și determină dacă este palindrom.

2 Clase

O clasa reprezinta extensie a structura de date, unde, pe langa membrii care pastreaza starea/ datele (*proprietati*), exista si functii ca membrii (*metode*).

```
1  class <nume_clasa> {
2      <specificator_access>:
3          /* definitii proprietati & metode */
4      <specificator_access>:
5          /* definitii proprietati & metode */
6
7      <numeClasa> (); // constructor
8      ~<nume_clasa> (); // destructor
9  } <list_obiecte>;
```

Exemplu:

```
1  class Point {
2      int x = 3, y = -5;
3
4      public:
5          void show () {
6              std::cout << "(" << x << ", " << y << ")";
7          }
8  } p;
```

Se pot declara clase folosind si **struct** :

```
1  struct <nume_structura> {
2      <specificator_access>:
3          /* definitii proprietati & metode */
4      <specificator_access>:
5          /* definitii proprietati & metode */
6
7      <nume_structura> (); // constructor
8      ~<nume_structura> (); // destructor
9  } <list_obiecte>;
```

Exemplu:

```
1  struct Point {
2      int x = 3, y = -5;
3  } p;
```

3 Specificatori de access

Un concept de baza in POO este **encapsularea informatiei**: ascunderea informatiei si oferirea unei interfete prin care se pot executa un set de operatii pe acea informatie.

Q: De ce encapsulare? A: Pentru ca, de cele mai multe ori, operarea directa pe datele unei structuri poate duce la rezultate neprevazute atunci cand modificarea datelor presupune anumiti algoritmi. Prin encapsulare se poate forta ca orice operatie definita pe un set de date sa fie facute doar prin mecanismele definite, care modifica datele intr-o maniera consistenta si sigura.

In C++ avem 3 specificatori:

- **private** - access permis doar in interiorul clasei
- **protected** - access permis in interiorul clasei si in clasele derivate
- **public** - access permis peste tot.

Este recomandat ca poate proprietatile unei clase sa fie declarate cu specificatorul **private** sau **protected**

In cazul claselor declarate folosind cuvantul cheie **class**, specificatorul de access default este **private**. In cazul claselor declarate folosind cuvantul cheie **struct**, specificatorul default de access este **public**.

4 Constructori

Un constructor reprezinta "reteta" dupa care se creaza obiectul. Constructorul spune compilatorului cum anume trebuie construit un obiect dintr-o clasa. Constructorul se declara similar cu metodele, cu urmatoarele precizari: numele este identic cu numele clasei si nu exista tip de retur:

```
1  class Point {  
2      int x = 3, y = -5;  
3  
4      public:  
5          Point (int a, int b) {  
6              x = a;  
7              y = b;  
8          }  
9          void show () {  
10             std::cout << "(" << x << ", " << y << ")";  
11         }  
12     };
```

Daca o clasa nu are definit un constructor, compilatorul va atribui un constructor implicit pentru acea clasa.

Exista mai multe tipuri:

- constructor fara parameterii
- constructor cu parameterii
- constructor de copiere

Constructorul cu parametrii si constructorul fara parametrii sunt apelati la declararea obiectului. Aceste doua tipuri de constructor spun cum anume se creaza un obiect atunci cand nu avem informatii despre obiect sau cand avem informatii despre obiect.

Exemplu:

```
1 class Point {
2     int x = 3, y = -5;
3
4     public:
5         Point () {
6             x = 0;
7             y = 0;
8         }
9         Point (int a, int b) {
10            x = a;
11            y = b;
12        }
13 };
14
15 int main () {
16     Point p; // se apeleaza constructorul fara parameterii
17     Point o(1, -10); // se apeleaza constructorul cu parameterii
18     return 0;
19 }
20
```

Constructorul de copiere este apelat la apelul unei functii care primeste ca parametru un obiect de tipul clasei. Scopul este sa se creeze o copie a obiectului curent, care sa fie folosita in apelul functiei, a.i. dupa terminarea apelului obiectul curent sa ramana nemodificat.

```
1 class Point {
2     int x = 3, y = -5;
3
4     public:
5         Point (Point& p) {
6             x = p.x;
7             y = p.y;
8         }
9 };
10
11 void foo (Point o) {
12     /* corpul functiei */
13 }
14
15 int main () {
16     Point p;
17     Point o(p); // se apeleaza explicit constructorul de copiere
18     f(p); // se apeleaza implicit constructorul de copiere
19     return 0;
20 }
```

Este absolut necesar ca parameterul pasat catre constructorul de copiere sa fie pasat prin referinta. Altfel se ajunge la un lant infinit de apeluri recursive.

5 Destructori

Destructorul este opusul constructorului. Scopul unui destructor este sa spuna compilatorului cum anume se distruge obiectul. Destructorul este apelat automat de compilator atunci cand durata de viata a unui obiect ajunge la sfarsit.

La fel ca in cazul constructorilor, destructori se declara ca orice alta metoda, avand numele clasei cu `~` in fata si niciun tip de retur

```
1  class Point {
2      int x, y;
3
4      public:
5          Point () {
6              x = 3;
7              y = 44;
8          }
9
10         ~Point () {
11             std::cout << "Obiect distrus";
12         }
13     };
14
15     int main () {
16         Point p;
17         return 0;
18     } // se apeleaza destructorul
19
```

Daca nu este declarat un destructor compilatorul creaza un destructor implicit.

Exercitii

1. Implementati clasa **Stiva** avand urmatoarele functionalitati:

- memorie alocata dinamic
- constructor (de toate tipurile) si destructor
- metoda de adaugare element in Stiva
- metoda de eliminare element din Stiva care returneaza elementul eliminat;

2. Implementati clasa **Vector** avand urmatoarele functionalitati:

- memorie alocata dinamic
- constructor (de toate tipurile) si destructor
- metoda de adaugare element la sfarsitul vectorului

- metoda de eliminare element de la sfarsitul vectorului
- metoda ce intoarce un element de pe pozitia **i** primita ca parameteru
- metoda ce intoarce numarul de elemente din vector