Seminarul 1

1 Introducere in C++

Un program in C++ are urmatoarea structura:

```
<directive-include>;

int main (<parameterii>) { // functia principala main
    <instructiuni-program>;
    return <valoare>;
}
```

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 principla `main` este of functie care intoarce un intreg, care reprezinta codul de iesire al procesului in urma unei executiei. Daca nu se intoarce o valoare din main, implicit compilatorul intoarce valoarea `0` .

Exemplu:

```
#include <iostream>

int main () {
    std::cout << "Hello world";
    return 0;
}
```

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:

```
<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 varialbila incepe intotdeauna cu o litera (sau `_`). Spatiile sau alte caractere speciale nu pot fi folosite in numele unei variablile. Numele de variabile sunt case sensitive (e.g. variabila `var` este diferita de variabila `Var`)

Type Name	Bytes	Other Names	Range of Values
`int`	4	`signed`	-2,147,483,648 to 2,147,483,647
`unsigned int`	4	`unsigned`	0 to 4,294,967,295
`__int8`	1	`char`	-128 to 127
`unsigned __int8`	1	`unsigned char`	0 to 255
`__int16`	2	`short`, `short int`, `signed short int`	-32,768 to 32,767
`unsigned __int16`	2	`unsigned short`, `unsigned short int`	0 to 65,535
`__int32`	4	`signed`, `signed int`, `int`	-2,147,483,648 to 2,147,483,647
`unsigned __int32`	4	`unsigned`, `unsigned int`	0 to 4,294,967,295
`__int64`	8	`long long`, `signed long long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned __int64`	8	`unsigned long long`	0 to 18,446,744,073,709,551,615
`bool`	1	none	`false` or `true`
`char`	1	none	-128 to 127 by default 0 to 255 when compiled by using /J
`signed char`	1	none	-128 to 127
`unsigned char`	1	none	0 to 255
`short`	2	`short int`, `signed short int`	-32,768 to 32,767
`unsigned short`	2	`unsigned short int`	0 to 65,535
`long`	4	`long int`, `signed long int`	-2,147,483,648 to 2,147,483,647
`unsigned long`	4	`unsigned long int`	0 to 4,294,967,295
`long long`	8	none (but equivalent to `__int64`)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long`	8	none (but equivalent to `unsigned __int64`)	0 to 18,446,744,073,709,551,615
`enum`	varies	none	
`float`	4	none	3.4E +/- 38 (7 digits)
`double`	8	none	1.7E +/- 308 (15 digits)
`long double`	same as `double`	none	Same as `double`
`wchar_t`	2	`__wchar_t`	0 to 65,535

Figure 1: Tipuri de date primitive in C++

Exemplu:

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

Citirea si afisarea datelor

Pentru citirea si afisarea datelor se folosesc operatorii `<<` si `>>` (operatorii de shift-are pe biti din C). 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 iesier `cout` pentru citirea si afisarea la tastatura.

```
int i;

cin >> i; // citeste i de la tastatura
cout << "i = " << i;

float f1, f2, f3;

cin >> f1 >> f2 >> f3; // se citesc 3 numere de tip float de la tastatura
cout << "suma = " << f1 + f2 + f3;
```

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

```
#include <iostream> // fluxurile ofstream si ifstream se gasesc in headerul iostream


ifstream fin("data.in");
ofstream out("data.out");

int main () {
    int i;

    fin >> i; // citeste i din fisier
    fout << "i = " << i;

    float f1, f2, f3;

    fin >> f1 >> f2 >> f3; // se citesc 3 numere de tip float din fisier
    fout << "suma = " << f1 + f2 + f3;

    return 0;
}
```

Instructiunea conditionala

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

```
if (<conditie>) {
    <instructiuni executate conditie adevarata>;
} [else {
    <instructiuni executate conditie falsa>;
}]
```

Putem ca in blocul de instruciuni executat atunci cand conditia este indeplinita/respinsa sa avem alte instruciuni `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. Aceasta regula poate fi alterata folosind acolade (in masura respectarii sintaxei).

```
if (i != 0)
    if (i > 0)
        cout << "pozitiv";
    else
        cout << "negativ";
else
    cout << "nul";
```

Instructiunea de selectie multipla

In cazul in care avem de verificat o valoare intr-o lista de valori posibile, folosirea instructiuni `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:

```
switch (<expresie>)
{
    case <val-constatant-1>:
        <bloc-instructiuni-caz-1>;
        break;
    case <val-constatant-2>:
        <bloc-instructiuni-caz-2>;
        break;
    .
    .
    .
    [default:
        <instructiuni-caz-default>;]
}
```

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

```
switch (x) {
    case 1:
        cout << "x este 1";
        break;
    case 2:
        cout << "x este 2";
        break;
    default:
        cout << "nu se cunoaste x";
}
```

Cicluri

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

Ciclul `while` :

```
while (<conditie-continuare>) {
    <instructiuni>;
}
```

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

Ciclul `do-while` :

```
do {
    <instructiuni>;
} while (<conditie-continuare>);
```

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

Ciclul `for` :

```
for (<initializari>; <conditie-continuare>; <incrementari>) {
    <instructiuni>
}
```

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

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

Functii

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

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

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

```
int suma (int a, int b) {
    int s = a + b;
    return s;
}

int main () {
    int i = 52, j = 44;
    int k = suma(i, j);
}
```

Daca o functie intoarce orice tip de date diferit de `void` , atunci fiecare ramura de executie a functiei trebuie sa intoarca o valoare. Functiile care intorc `void` nu necesita folosirea cuvantului `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.

```
void swapVal (int a, int b) {
    int c = a;
    a = b;
    b = c;
}

void swapRef (int &a, int &b) {
    int c = a;
    a = b;
    b = c;
}

int main () {
    int i = 5, j = 10;
    swapVal(i, j);
    cout << i << " " << j; // afiseaza '5 10'
    swapRef(i, j);
    cout << i << " " << j; // afiseaza '10 5'
}
```

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 urmator:

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

Pentru a initializa un tablou avem:

```
int v1[5] = {1, 2, 3, 4, 5};
int v2[] = {1, 2, 3, 4, 5}; // dimensiunea este dedusa din lista de valori.
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 acessa elementul cu indexul `i` trebuie sa scriem `<nume-tablou>[i]` .

```
int v[5] = {10, 20, 30, 40, 50};
cout << v[3]; // afiseaza 40
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-taboul>);` . Exemplu:

```
#include <iostream>
using namespace std;

void print (int v[], int n) {
    for (int i = 0; i < n; ++i) {
        cout << v[i] << ' ';
    }
    cout << '\n';
}

int main ()
{
    int w[3] = {5, 10, 15};
    print (w,3);
    return 0;
}
```

Pointeri

In C++ orice variabila este declarata static, i.e. memoria este prealoca pentru acea variabila la inceputul executiei programului si este dezalocata la sfarsitul zonei de vizibilitate a variabilei.

In cazul in care ne dorim sa nu se intample acest lucru, putem folosi *pointeri* pentru a putea aloca memoria cand avem nevoie de ea. *Un pointer este o adresa catre o zona din memorie.* Un pointer se poate declara in urmatorul fel:

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

Cand declaram un pointer, compilatorul aloca doar 4 octeti necesari pentru a putea salva valoarea adresei de memorie. Pentru a putea aloca un pointer, folosim operatorul `new` :

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

sau, pentru a aloca un bloc de `<dimensiune>` locatii de memorie consecutive:

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

Pentru a accesa valoarea de la adresa catre care indica un pointer trebuie sa scriem `*<nume-pointer>` . Pentru a access valoarea de pe pozitia `i` din blocul de memorie adresa catre care indica un pointer trebuie sa scriem `*(<nume-pointer> + i)` sau `<nume-pointer>[i]` .

Pentru a dezaloca o zona de memorie folosim operatorul `delete` :

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

sau pentru a dezaloca un bloc de memorie (locatii consecutive):

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

Daca folosim `delete` pe un un pointer alocat cu `new[]` se va dezaloca doar prima locatie din blocul de memorie, restul locatiilor ramanand alocate.

Exemplu:

```
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {
    int *n = new int;
    int *v;

    cin >> *n;

    v = new int[*n];

    for (int i = 0; i < *n; i++) {
        cin >> *(v+i);
    }

    for (int i = 0; i < *n; i++) {
        cout << v[i] << " ";
    }

    delete [] v;

    return 0;
}
```

Structuri

In dezvoltarea unui program sunt situatii in care tipurile de date predefinite nu sunt suficiente si trebuie sa ne definim propriile tipuri de date. Pentru asta avem conceptul de structuri prin care putem defini noi tipuri de date:

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

Pentru a putea accessa o proprietate a unei structuri trebuie se foloseste operatorul `.` :

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

Exemplu:

```
struct student {
    char *nume, *prenume;
    int varsta;
} s1, s2;

s1.prenume = "Florin"
```

Putem aloca variabile de tip structura dinamic:

```
student *s = new student;
*(s).prenume = "Florin";
s->prenume = "Florin";
```

Exercitii

1. Implementati un program care citeste doua numere intregi de la tastatura si afiseaza maximul.

2. Implementati un program care citeste de la tastatura un $n \geq 3$ si afiseaza al n-lea numar al lui Fibbonaci.

3. Implementati un program care citeste de la tastatura un vector de numere in virgula mobila si il sorteaza in ordine crescatoare.

4. Implementati un program care citeste de la tastatura un numar intreg si determina daca 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*).

```
    class <nume_clasa> {
        <specificator_access>:
            /* definitii proprietati & metode */
        <specificator_access>:
            /* definitii proprietati & metode */

        <numeClasa> (); // constructor
        ~<nume clasa> (); // destructor
    } <list_obiecte>;
```

Exemplu:

```
    class Point {
        int x = 3, y = -5;

        public:
            void show () {
                std::cout << "(" << x << ", " << y << ")";
            }
    } p;
```

Se pot declara clase folsind si `struct` :

```
    struct <nume_structura> {
        <specificator_access>:
            /* definitii proprietati & metode */
        <specificator_access>:
            /* definitii proprietati & metode */

```

```
        <nume_structura> (); // constructor
        ~<nume_structura> (); // destructor
    } <list_obiecte >;
```

Exemplu:

```
    struct Point {
        int x = 3, y = -5;
    } 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 rezulate 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 mechanismele 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 folsind cuvantul cheie `class` , specificatorul de access default este `private` . In cazul claselor declarare folosing 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 object dintr-o clasa. Constructorul se declara similar cu metodele, cu urmatoarele precizari: numele este identic cu numele clasei si nu exista tip de retur:

```
    class Point {
        int x = 3, y = -5;

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

```
};

void foo (Point o) {
    /* corpul functiei */
}

int main () {
    Point p;
    Point o(p); // se apeleaza explicit constructorul de copiere
    f(p);       // se apeleaza implicit constructorul de copiere
    return 0;
}
```

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

```
    class Point {
        int x, y;

        public:
            Point () {
                x = 3;
                y = 44;
            }

            ~Point () {
                std::cout << "Obiect distrus";
            }
    };

    int main () {
        Point p;
        return 0;
    } // se apeleaza destructorul

```

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