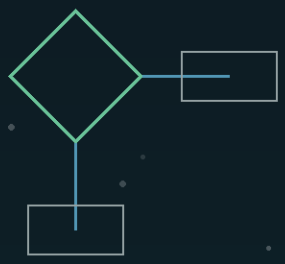
```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Salut, lume!";
    return 0;
}
```



1011
0110
1001
0011

MANUAL DE PROGRAMARE

C++

pentru liceu · profil informatică · clasele IX–XII

ALGORITMI · STRUCTURI · RECURSIVITATE · STL · GIT & DOCKER

Cuprins

Anexă — Glosar de cuvinte-cheie C++

INTRODUCERE

Cum să folosești această carte

De ce învățăm programare, cum este organizat manualul și cum lucrezi cu el.

Introducere

Cum să folosești această carte

Trăiești într-o lume construită din programe. Telefonul din buzunar, harta care îți arată drumul, jocul la care te uiți seara, motorul de căutare care îți răspunde în jumătate de secundă — toate sunt scrise de cineva, linie cu linie. A învăța să programezi înseamnă să treci de partea cealaltă a ecranului: din om care folosește tehnologia devii om care o creează.

Dar programarea nu este doar despre calculatoare. Este, înainte de toate, un mod de a gândi limpede. Înveți să descompui o problemă mare în pași mici, să te întrebi ce date ai și ce rezultat vrei, să verifici dacă raționamentul tău chiar funcționează. Aceste deprinderi îți folosesc la matematică, la fizică, la un proiect de școală sau pur și simplu atunci când vrei să iei o decizie bună.

De ce C++

Există multe limbaje de programare, fiecare bun la ceva. Noi am ales C++ din trei motive. Întâi, este limbajul folosit la olimpiadele de informatică și la examenul de Bacalaureat pe profil informatică din România — așa că tot ce înveți aici îți este util direct la școală și la concursuri. Apoi, C++ este rapid și aproape de modul în care funcționează cu adevărat un calculator, deci înțelegi nu doar „ce” se întâmplă, ci și „de ce”. În sfârșit, este folosit în lumea reală: în jocuri video, în motoare de browser, în programe științifice și în sisteme care trebuie să fie foarte eficiente.

Nu te lăsa intimidat de reputația lui de limbaj „greu”. Vom porni de la zero, pas cu pas, și fiecare idee nouă se va sprijini pe ceea ce ai înțeles deja. Nimic din această carte nu presupune că ai mai scris vreun rând de cod până acum.

Ideea de bază:

Programarea nu se învață citind, ci scriind cod. Citește, apoi scrie tu programul. Greșește, corectează, încearcă din nou. Fiecare eroare pe care o repari te învață mai mult decât o pagină citită fără greșeli.

După ce vei termina acest manual, vei putea scrie programe care citesc date, iau decizii, repetă acțiuni și prelucrează șiruri și tabele de numere. Vei ști să cauți și să sortezi, să descompui un program în funcții, să folosești recursivitatea și tehnici precum backtracking sau divide et impera. Vei lucra cu structuri de date și cu biblioteca standard STL, vei înțelege ce înseamnă complexitatea unui algoritm și vei face primii pași în lumea grafurilor. Pe scurt: vei avea tot ce îți trebuie pentru școală, pentru concursuri și pentru a continua singur mai departe.

Cum este organizată cartea

Manualul este împărțit în șase părți, care urcă treptat în dificultate. În prima parte faci primii pași: ce este un algoritm, primul tău program, tipuri de date, variabile, operatori, citire și afișare. A doua parte aduce controlul programului — decizii, cicluri și tehnici elementare. A treia se ocupă de date organizate: vectori, matrice, șiruri de caractere, căutare și sortare. A patra parte introduce funcțiile, recursivitatea, backtracking-ul și divide et impera. A cincea parte te duce mai în profunzime: pointeri, structuri, structuri de date, STL și complexitate. Iar a șasea parte deschide drumul mai departe, către grafuri și uneltele moderne ale unui programator.

Fiecare capitol se încheie cu o secțiune numită „Teste de verificare”. Acolo găsești programe scurte însoțite de rezultatul lor așteptat și câteva exerciții propuse. Rostul lor este simplu: să poți verifica singur dacă ai înțeles cu adevărat, înainte de a trece mai departe. Nu sări peste ele — sunt cea mai bună oglindă a progresului tău.

Cum să lucrezi cu acest manual

Citește mai întâi explicația și urmărește exemplul cu atenție. Apoi — și acesta este pasul cel mai important — scrie codul de mână tu însuți, nu doar copia-l. Compilează-l, rulează-l și uită-te la ce afișează. Modifică-l puțin și încearcă să prezici ce se va întâmpla înainte de a apăsa „Run”. La final, rezolvă testele de verificare. Învățarea programării seamănă cu învățarea unui instrument muzical: se câștigă prin exercițiu, nu prin privit.

De ce ai nevoie ca să rulezi programele

Ca să transformi codul C++ într-un program care chiar funcționează, ai nevoie de un compilator — un instrument care traduce ceea ce scrii tu în limbajul pe care îl înțelege calculatorul. Ultimul capitol, „Compilare cu GCC, Git și Docker pe Windows”, îți arată exact cum să îți pregătești calculatorul, pas cu pas. Dar nu trebuie să aștepți până acolo: la început poți folosi un compilator online direct din browser, fără să instalezi nimic. Deschizi pagina, scrii codul, apeși un buton — și vezi imediat rezultatul. Important este să rulezi cod cât mai devreme și cât mai des.

Acum ești pregătit să începi. Vei avea momente în care ceva nu merge și nu înțelegi de ce — toți programatorii din lume trec prin asta, în fiecare zi. Răbdarea, curiozitatea și obiceiul de a încerca din nou contează mai mult decât talentul. Deschide editorul, scrie primul tău rând de cod și hai să construim împreună. Succes!

PARTEA I

Primii pași

Ce este un algoritm, cum scriem și rulăm primul program C++, și cum lucrăm cu date.

PARTEA I

1.1 Algoritmi și pseudocod

Ce înseamnă să rezolvi o problemă pas cu pas

Înainte să scrii vreun rând de C++, merită să înțelegi un lucru: calculatorul nu gândește. El execută, ascultător, niște pași pe care i-i dai tu. Arta programării începe deci cu arta de a descrie limpede pașii unei rezolvări.

Un algoritm este o succesiune finită de pași clari prin care, pornind de la niște date de intrare, ajungem la un rezultat dorit. Cuvântul vine de la numele matematicianului persan al-Khwarizmi, care a trăit acum mai bine de o mie de ani.

Algoritmi din viața de zi cu zi

Folosești algoritmi tot timpul, fără să le spui așa. O rețetă de prăjitură este un algoritm: ingrediente (intrare), pași în ordine, prăjitura gata (ieșire). La fel, indicațiile „mergi drept, apoi la a doua stradă la stânga” sau pașii prin care îți legi șireturile.

De reținut:

Un algoritm transformă o INTRARE într-o IEȘIRE, prin pași bine definiți.

Proprietățile unui algoritm bun

Nu orice șir de instrucțiuni este un algoritm corect. Pentru a merita acest nume, el trebuie să respecte câteva proprietăți:

Claritate: fiecare pas este descris fără ambiguitate, astfel încât oricine îl execută la fel. „Pune sare după gust” nu e clar pentru un calculator; „adaugă 5 grame de sare” este.

Determinism: pentru aceleași date de intrare, algoritmul produce mereu același rezultat și urmează aceeași ordine de pași.

Finitudine: algoritmul se termină după un număr finit de pași. Unul care s-ar învârti la nesfârșit nu rezolvă nimic.

Intrare și ieșire: are zero sau mai multe date de intrare și produce cel puțin un rezultat.

Generalitate: rezolvă nu un singur caz, ci o întreagă clasă de probleme. Un algoritm de adunare funcționează pentru orice două numere, nu doar pentru 2 plus 3.

Cum reprezentăm un algoritm

Avem trei moduri uzuale de a descrie un algoritm, de la cel mai liber la cel mai formal:

1) În limbaj natural: cu propoziții obișnuite. E ușor de citit, dar uneori imprecis.

2) Prin pseudocod: un limbaj de mijloc, între vorbirea noastră și limbajul de programare. Folosește cuvinte ca CITEȘTE, SCRIE, DACĂ, CÂT TIMP. Noi îl vom folosi des în acest capitol.

3) Prin schemă logică (organigramă): un desen cu forme legate prin săgeți. Un oval marchează Start și Stop, un dreptunghi o prelucrare (un calcul, o atribuire), un paralelogram o operație de citire sau scriere, iar un romb o decizie, din care pleacă două ramuri: DA și NU. Săgețile arată ordinea în care parcurgem blocurile.

Primul nostru pseudocod: media a două numere

Citim două numere, le adunăm, împărțim suma la 2 și afișăm rezultatul.

```
PSEUDOCOD
┌ START
│ CITEȘTE a, b
│ media ← (a + b) / 2
│ SCRIE media
└ STOP
```

Săgeata ← înseamnă „primește valoarea”.
Pentru a = 6 și b = 10, media iese 8.

Decizii: maximul a două numere

Uneori algoritmul trebuie să aleagă. Folosim structura DACĂ ... ATUNCI ... ALTFEL, care corespunde rombului din schema logică.

```
PSEUDOCOD
┌ START
│ CITEȘTE a, b
│ DACĂ a > b ATUNCI
│      max ← a
│ ALTFEL
│      max ← b
│ SCRIE max
└ STOP
```

Rombul testează condiția a > b.
Pe ramura DA luăm a, pe ramura NU luăm b.

Maximul a trei numere

Extindem ideea: aflăm întâi maximul dintre primele două, apoi îl comparăm cu al treilea.

```
PSEUDOCOD
┌ START
│ CITEȘTE a, b, c
│ max ← a
│ DACĂ b > max ATUNCI max ← b
│ DACĂ c > max ATUNCI max ← c
│ SCRIE max
└ STOP
```

Pornim presupunând că a e cel mai mare,
apoi corectăm dacă găsim ceva mai mare.

Repetiție: suma primelor N numere

Când un pas trebuie repetat, folosim CÂT TIMP. Adunăm 1 + 2 + ... + N folosind o variabilă care ține suma și un contor care crește.

```
PSEUDOCOD
┌ START
│ CITEȘTE N
│ suma ← 0
│ i ← 1
│ CÂT TIMP i ≤ N EXECUTĂ
│     suma ← suma + i
│     i ← i + 1
│ SCRIE suma
└ STOP
```

Contorul i merge de la 1 la N.
La fiecare pas adăugăm i la sumă.

Idee-cheie:

Atribuire, decizie și repetiție sunt cărămizile din care construim orice algoritm.

Teste de verificare

Execută „pe hârtie” fiecare pseudocod de mai jos, urmărind valorile, și compară cu rezultatul dat.

Testul 1 — media pentru a = 7, b = 3:

```
PSEUDOCOD
CITEȘTE a, b
media ← (a + b) / 2
SCRIE media
```

```
REZULTAT AȘTEPTAT
media = 5
```

Testul 2 — maximul pentru a = 4, b = 9:

```
PSEUDOCOD
CITEȘTE a, b
DACĂ a > b ATUNCI max ← a
ALTFEL max ← b
SCRIE max
```

```
REZULTAT AȘTEPTAT
max = 9
```

Testul 3 — suma primelor numere pentru N = 5:

```
PSEUDOCOD
CITEȘTE N
suma ← 0; i ← 1
CÂT TIMP i ≤ N EXECUTĂ
    suma ← suma + i; i ← i + 1
SCRIE suma
```

```
REZULTAT AȘTEPTAT
suma = 15
```

Exerciții propuse

1) Scrie pseudocodul care citește un număr și afișează dublul lui.

2) Scrie pseudocodul care citește două numere și afișează minimul dintre ele.

3) Desenează schema logică (descrie blocurile start/stop, prelucrare, decizie) pentru algoritmul care decide dacă un număr este par sau impar.

4) Scrie pseudocodul care citește N și afișează produsul primelor N numere naturale (factorialul lui N).

Provocare:

Scrie pseudocodul care citește 3 numere și le afișează ordonate crescător.

PARTEA I

1.2 Primul program C++

Structura unui program și funcția main

Fiecare programator începe la fel: scrie un program care afișează un scurt mesaj pe ecran. Este un ritual vechi și, pe lângă faptul că este simplu, ne arată întreaga anatomie a unui program C++. Hai să îl scriem, apoi să îl disecăm linie cu linie.

Programul „Salut, lume!”

CPP

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

int main() {
    cout << "Salut, lume!" << endl;
    return 0;
}
```

Cele șapte linii de mai sus formează un program C++ complet, care poate fi compilat și rulat.

REZULTAT

```
Salut, lume!
```

Anatomia liniei cu linie

La prima vedere par multe simboluri, dar fiecare are un rol precis. Le luăm pe rând.

#include <iostream>

Aduce în program „uneltele” de intrare/ieșire. Fără el, nu am putea folosi cout pentru afișare.

Cuvântul iostream vine de la input/output stream (flux de intrare/ieșire). Linia care începe cu # este o directivă preprocesor: ea se execută înainte de compilarea propriu-zisă și lipește conținutul bibliotecii în programul nostru.

using namespace std;

Ne permite să scriem cout în loc de std::cout. Numele standard din C++ stau în „spațiul” std.

int main() { ... }

Funcția main este punctul de pornire. Execuția programului începe mereu de aici.

Calculatorul caută întotdeauna funcția main și începe să execute instrucțiunile dintre acoladele { și }. Acoladele marchează începutul și sfârșitul corpului funcției; tot ce vrem să facă programul scriem între ele. Cuvântul int din față spune că funcția returnează un număr întreg.

cout << "Salut, lume!" << endl;

cout trimite text către ecran (consolă). Operatorul << „împinge" valoarea spre ieșire. endl trece pe linia următoare.

Textul dintre ghilimele se numește șir de caractere și este afișat exact așa cum l-am scris. La final apare ; (punct și virgulă), care încheie orice instrucțiune în C++. Lipsa lui este cea mai frecventă eroare a începătorului.

return 0;

Semnalează că programul s-a terminat cu bine. Valoarea 0 înseamnă „fără erori".

Cum se compilează și se rulează

Calculatorul nu înțelege direct codul C++; mai întâi un compilator îl traduce într-un program executabil. Pe scurt, salvăm codul într-un fișier (de exemplu salut.cpp), îl compilăm, apoi rulăm executabilul rezultat.

```text
g++ salut.cpp -o salut
./salut
```

Prima comandă compilează, a doua rulează. Detaliile complete (instalare, GCC, Git, Docker) sunt în capitolul „Compilare cu GCC, Git și Docker pe Windows". La început, cel mai simplu este să folosești un compilator online din browser.

Comentarii: notițe pentru oameni

Comentariile sunt texte ignorate de compilator, scrise pentru a explica codul. Ai două forme: // pentru o singură linie și /* ... */ pentru mai multe linii.

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

int main() {
    // afișăm un salut pe ecran
    cout << "Buna ziua!" << endl;
    /* Tot ce este aici,
       pe mai multe linii,
       este ignorat. */
    return 0;
}
```

Folosește comentariile ca să explici de ce faci ceva, nu doar ce faci.

```
Buna ziua!
```

Mai multe linii cu endl

Putem afișa oricâte linii dorim. De fiecare dată când vrem să trecem pe rândul următor, folosim endl (sau caracterul special \n).

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

int main() {
    cout << "Linia 1" << endl;
    cout << "Linia 2" << endl;
    cout << "Gata!" << endl;
    return 0;
}
```

Fiecare instrucțiune cout afișează un text, iar endl mută cursorul pe linia de jos.

```
Linia 1
Linia 2
Gata!
```

Erori frecvente ale începătorului

De evitat:

Lipsa lui ; la finalul instrucțiunii. Majuscule greșite: Cout sau MAIN nu există. Ghilimele neînchise în jurul textului afișat. Uitarea acoladei } care închide funcția main.

C++ ține cont de litere mari și mici: cout este corect, dar Cout sau COUT dau eroare. Dacă programul nu compilează, citește cu atenție prima eroare raportată; de obicei, restul dispar după ce o rezolvi pe prima.

Teste de verificare

Scrie, compilează și rulează programele de mai jos. Compară rezultatul tău cu cel așteptat.

CPP

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

int main() {
    cout << "Am invatat C++!" << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
Am invatat C++!
```

CPP

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

int main() {
    cout << "Nume: Ana" << endl;
    cout << "Clasa: a IX-a" << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
Nume: Ana
Clasa: a IX-a
```

CPP

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

int main() {
    // comentariul nu apare in rezultat
    cout << "*****" << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
*****
```

Exerciții propuse

Rezolvă singur:

1. Afișează numele și prenumele tău pe ecran. 2. Afișează pe trei linii: orașul, județul, țara. 3. Afișează un dreptunghi din caractere * (3 linii). 4. Adaugă un comentariu // care explică programul 1. 5. Strică intenționat un ; și citește mesajul de eroare al compilatorului.

PARTEA I

1.3 Tipuri de date și variabile

Cum păstrăm și folosim valori

Un program lucrează aproape mereu cu valori: un număr, o literă, un răspuns de tip da/nu. Ca să le ținem minte și să le refolosim, le punem în variabile.

Idee de reținut:

O variabilă este o cutie cu nume în memorie. În cutie pui o valoare; numele te ajută s-o regăsești.

Declarare și inițializare

A declara o variabilă înseamnă a spune ce tip de valoare va păstra și ce nume are. A o inițializa înseamnă a-i pune o valoare din primul moment. Fără inițializare, cutia conține o valoare necunoscută, gunoi rămas în memorie.

```cpp
int varsta;        // declarare, fara valoare
varsta = 16;       // atribuire ulterioara
int an = 2026;     // declarare + initializare
```

Recomandare: inițializează variabila chiar când o declari, ca să eviți valori surpriză.

Tipurile fundamentale

C++ are câteva tipuri de bază. Fiecare ocupă un spațiu în memorie și acoperă un anumit domeniu de valori. Iată cele mai folosite:

Tipuri uzuale:

int - numere intregi (ex. -5, 0, 42) long long - numere intregi foarte mari float - numere zecimale, precizie mica double - numere zecimale, precizie buna char - un singur caracter (ex. 'A') bool - adevarat / fals (true / false)

Un int obișnuit acoperă numere până la cam două miliarde. Dacă lucrezi cu numere mai mari (de exemplu populația lumii sau distanțe în kilometri spre stele), folosește long long, care merge până la valori uriașe.

```cpp
int populatie_oras = 350000;
long long populatie_glob = 8000000000LL;
```

Sufixul LL spune compilatorului că literalul este de tip long long.

Literali: valorile scrise direct

Un literal este o valoare scrisă chiar în cod. Forma lui îi spune compilatorului ce tip are.

```cpp
int n = 10;          // literal intreg
double pi = 3.14;    // literal zecimal
char litera = 'A';   // literal caracter, apostrof
bool ploua = true;   // literal logic
```

Atenție: caracterul folosește apostrof ('A'), iar un text (string) folosește ghilimele ("Ana").

Constante cu const

Uneori o valoare nu trebuie să se mai schimbe în timpul programului, de exemplu numărul Pi. O marcăm cu cuvântul const; dacă încerci s-o modifici, compilatorul dă eroare.

```cpp
const double PI = 3.14159;
// PI = 4;  // EROARE: nu poti schimba un const
```

Reguli de denumire

Numele unei variabile spune cititorului la ce folosește. Câteva reguli simple te țin pe drumul drept:

Cum denumim corect:

- incepe cu litera sau _ (nu cu cifra) - doar litere, cifre si _ (fara spatii, fara ar.) - conteaza majusculele: varsta != Varsta - nu folosi cuvinte rezervate (int, return...) - alege nume clare: raza, nrElevi, esteValid

Atribuirea cu =

Semnul = nu înseamnă egalitate matematică, ci atribuire: ia valoarea din dreapta și o pune în cutia din stânga. Poți reține rezultatul unui calcul tot printr-o atribuire.

```cpp
int a = 5;
int b = 3;
int suma = a + b;   // suma primeste 8
a = a + 1;          // a devine 6
```

Conversii între tipuri

Când amesteci int și double, C++ transformă automat valorile. Trecerea de la double la int taie partea zecimală (nu rotunjește). Cea mai vicleană capcană este împărțirea: dacă ambele numere sunt întregi, rezultatul este tot întreg.

```cpp
int x = 7, y = 2;
cout << x / y << endl;      // 3 (impartire int)
cout << x / 2.0 << endl;    // 3.5 (apare double)
double z = x;               // 7 devine 7.0
int w = 3.9;                // w devine 3 (taie)
```

Dacă vrei rezultat zecimal, fă măcar un operand double, de exemplu împarte la 2.0, nu la 2.

```
3
3.5
```

Exemplu complet: aria cercului

Punem totul la lucru: o constantă, un double pentru rază, un calcul și afișarea rezultatului.

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

int main() {
    const double PI = 3.14159;
    double raza = 2.0;
    double arie = PI * raza * raza;
    int varsta = 16;
    char initiala = 'M';
    bool elev = true;
    cout << "Aria: " << arie << endl;
    cout << "Varsta: " << varsta << endl;
    cout << "Initiala: " << initiala << endl;
    cout << "Este elev: " << elev << endl;
    return 0;
}
```

Un bool afișat apare ca 1 (true) sau 0 (false).

```
REZULTAT
Aria: 12.5664
Varsta: 16
Initiala: M
Este elev: 1
```

Teste de verificare

Compilează și rulează fiecare program; compară-ți ieșirea cu rezultatul așteptat.

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

int main() {
    int mere = 10, cosuri = 3;
    cout << mere / cosuri << endl;
    cout << mere % cosuri << endl;
    return 0;
}
```

Împărțire întreagă și restul ei.

```
REZULTAT AȘTEPTAT
3
1
```

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

int main() {
    double a = 5, b = 2;
    cout << a / b << endl;
    char c = 'B';
    cout << c << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
2.5
B
```

Exerciții propuse

Rezolvă singur:

1. Declara raza unui cerc (double) si afiseaza circumferinta: 2 * PI * raza. 2. Citeste de la tastatura doua numere intregi si afiseaza catul si restul impartirii lor. 3. Pune intr-un long long numarul 5000000000 si afiseaza-l; incearca si cu int - ce observi? 4. Declara un char cu prima litera a numelui tau si un bool care spune daca esti la liceu. 5. Calculeaza media a trei note (double) si afiseaza-o cu doua zecimale.

PARTEA I

1.4 Operatori și expresii

Aritmetici, relaționali, logici

O expresie este o combinație de valori, variabile și operatori pe care calculatorul o evaluează pentru a obține un rezultat. De exemplu, 3 + 4 * 2 este o expresie a cărei valoare este 11. Operatorii sunt simbolurile care spun ce operație se face, iar operanzii sunt valorile asupra cărora se lucrează.

Operatori aritmetici

Cei cinci operatori aritmetici de bază sunt: + (adunare), - (scădere), * (înmulțire), / (împărțire) și % (rest, numit modulo). Primii patru funcționează cum te aștepți, dar împărțirea / ascunde o capcană importantă.

Împărțire întreagă vs reală

Dacă ambii operanzi sunt numere întregi, / face împărțire ÎNTREAGĂ: rezultatul este partea întreagă, fără zecimale. Dacă măcar unul este real (double), / face împărțire obișnuită, cu zecimale.

CPP

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

int main() {
    cout << 7 / 2 << endl;     // intreg: 3
    cout << 7.0 / 2 << endl;   // real: 3.5
    cout << 7 % 2 << endl;     // rest: 1
    return 0;
}
```

7 / 2 dă 3 fiindcă ambele sunt întregi. Punând 7.0 forțăm calculul real.

REZULTAT

```
3
3.5
1
```

Operatorul modulo %

Operatorul % dă restul împărțirii întregi. El este foarte util: a % 2 spune dacă un număr e par (rest 0) sau impar (rest 1), iar n % 10 dă ultima cifră a unui număr. Atenție: % se folosește doar cu numere întregi.

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

int main() {
    int n = 2374;
    cout << n % 10 << endl;  // ultima cifra: 4
    cout << n % 2 << endl;   // paritate: 0 (par)
    return 0;
}
```

n % 10 izolează ultima cifră, iar n % 2 verifică paritatea.

```
REZULTAT
4
0
```

Atribuire compusă

În loc să scriem x = x + 5, putem prescurta cu x += 5. La fel există -=, *=, /= și %=. Toate iau valoarea curentă, aplică operația cu valoarea din dreapta și pun rezultatul înapoi în variabilă.

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

int main() {
    int x = 10;
    x += 5;   // x devine 15
    x *= 2;   // x devine 30
    x %= 7;   // restul lui 30 la 7 -> 2
    cout << x << endl;
    return 0;
}
```

Fiecare linie modifică x pe loc, pas cu pas.

```
REZULTAT
2
```

Incrementare și decrementare

Operatorii ++ și -- cresc sau scad o variabilă cu 1. Forma pre (++i) modifică variabila și apoi folosește noua valoare; forma post (i++) folosește mai întâi valoarea veche și abia apoi crește. Pe scurt: dacă rezultatul nu e folosit în aceeași expresie, ++i și i++ fac același lucru.

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

int main() {
    int i = 5;
    cout << i++ << endl;  // afiseaza 5, apoi i=6
    cout << ++i << endl;  // i=7, apoi afiseaza 7
    return 0;
}
```

i++ dă valoarea veche (5), ++i dă valoarea nouă (7).

```
REZULTAT
5
7
```

Operatori relaționali

Operatorii ==, !=, <, <=, > și >= compară două valori și dau un rezultat de tip bool: true (1) sau false (0). Atenție la == (compară) față de = (atribuie) — o greșeală foarte frecventă!

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

int main() {
    cout << (3 < 5) << endl;    // true -> 1
    cout << (4 == 4) << endl;   // true -> 1
    cout << (2 >= 9) << endl;   // false -> 0
    return 0;
}
```

Rezultatul unei comparații se afișează ca 1 (true) sau 0 (false).

```
REZULTAT
1
1
0
```

Operatori logici și tabele de adevăr

Operatorii logici combină condiții: && (ȘI) e true doar dacă ambele părți sunt true; || (SAU) e true dacă măcar una e true; ! (NU) inversează valoarea.

Tabele de adevăr (1=true, 0=false):

A B | A && B | A || B 0 0 | 0 | 0 0 1 | 0 | 1 1 0 | 0 | 1 1 1 | 1

1 !0 = 1 !1 = 0

CPP

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

int main() {
    int v = 16;
    // par SI mai mare ca 10?
    bool ok = (v % 2 == 0) && (v > 10);
    cout << ok << endl;  // 1
    return 0;
}
```

Condiția compusă e true doar dacă v e par ȘI peste 10.

REZULTAT

```
1
```

Precedență și asociativitate

Când într-o expresie apar mai mulți operatori, contează ordinea în care se aplică. Precedența spune cine se evaluează primul; asociativitatea spune ce facem la egalitate (de obicei de la stânga la dreapta).

De la mare la mică precedență:

1. ! ++ -- (unari) 2. * / % 3. + - 4. < <= > >= 5. == != 6. && 7. || 8. = += -= ...

Așa, în 3 + 4 * 2 se face întâi 4 * 2 = 8, apoi 3 + 8 = 11. Dacă vrei altă ordine, folosește paranteze: (3 + 4) * 2 dă 14. Sfat: când nu ești sigur, pune paranteze — codul devine clar și corect.

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

int main() {
    cout << 3 + 4 * 2 << endl;      // 11
    cout << (3 + 4) * 2 << endl;    // 14
    return 0;
}
```

Parantezele schimbă ordinea de evaluare.

```
11
14
```

Teste de verificare

Compilează și rulează programele de mai jos, apoi compară ieșirea cu rezultatul așteptat.

Test 1 — Conversie din lei în bani (1 leu = 100 bani):

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

int main() {
    double ron = 12.50;
    int bani = (int)(ron * 100);
    cout << bani << endl;
    return 0;
}
```

```
1250
```

Test 2 — Suma cifrelor unui număr de două cifre:

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

int main() {
    int n = 47;
    int s = n / 10 + n % 10;
    cout << s << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
11
```

Test 3 — Verifică dacă un an e bisect:

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

int main() {
    int an = 2024;
    bool bisect = (an % 4 == 0 && an % 100 != 0)
                  || (an % 400 == 0);
    cout << bisect << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
1
```

Exerciții propuse

1. Citește un număr întreg de trei cifre și afișează ultima sa cifră folosind % 10.

2. Citește un întreg și afișează 1 dacă este par și 0 dacă este impar (folosind % 2).

3. Citește o sumă în bani (întreg) și afișează câți lei întregi și câți bani sunt (cu / și %).

4. Citește două note întregi și afișează 1 dacă media lor este cel puțin 5, altfel 0 (folosind operatori logici).

5. Fără să rulezi, calculează pe hârtie valoarea expresiei 10 - 2 * 3 + 8 / 4, apoi verifică în C++.

PARTEA I

1.5 Citire și afișare

Dialogul programului cu utilizatorul

Un program devine cu adevărat util abia atunci când comunică: primește date de la utilizator și îi arată rezultate. În C++ acest dialog se poartă prin două obiecte din biblioteca <iostream>: cout (pentru afișare) și cin (pentru citire).

Afișarea cu cout

cout (citește „si-aut”, de la console output) este fluxul de ieșire standard. Trimitem text sau valori către el folosind operatorul <<, numit operator de inserție. Imaginează-ți <<-ul ca pe o săgeată care împinge datele spre ecran.

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

int main() {
    cout << "Salut, lume!";
    return 0;
}
```

Textul dintre ghilimele se numește șir de caractere și apare exact așa pe ecran.

```
Salut, lume!
```

Înlănțuirea și trecerea la linie nouă

Putem pune mai multe << pe același rând, unul după altul. Datele apar în ordinea în care le scriem. Pentru a trece pe o linie nouă folosim endl sau secvența "\n" pusă în interiorul unui șir.

```cpp
int a = 5, b = 7;
cout << "Suma este " << a + b << endl;
cout << a << " + " << b << "\n";
```

endl și "\n" fac același lucru: mută cursorul pe rândul următor. Observă spațiile puse în șiruri ca să nu se lipească valorile.

```
REZULTAT
Suma este 12
5 + 7
```

Citirea cu cin

cin (console input) este fluxul de intrare standard, de obicei tastatura. Citim valori cu operatorul >>, numit operator de extracție. Săgeata arată acum spre variabilă: datele intră în ea.

```cpp
int n;
cin >> n;
cout << "Ai introdus " << n << endl;
```

Programul așteaptă să tastezi un număr și să apeși Enter; valoarea ajunge în variabila n.

Citirea mai multor valori și ordinea lor

Putem citi mai multe valori cu un singur cin, înlănțuind >>. Le poți tasta despărțite prin spațiu pe același rând sau câte una pe rând (Enter între ele) — pentru C++ spațiul și Enter sunt la fel. Important este că valorile se citesc în ordinea scrierii: prima valoare intră în prima variabilă.

```cpp
int a, b;
cin >> a >> b;
```

Dacă tastezi 5 7, atunci a devine 5 și b devine 7.

Un program tipic: citește, calculează, afișează

Cele mai multe programe respectă același tipar în trei pași: întâi citesc datele de intrare, apoi fac un calcul, iar la final afișează rezultatul. Iată suma a două numere.

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

int main() {
    int a, b;
    cin >> a >> b;        // citește
    int s = a + b;        // calculează
    cout << s << endl;    // afișează
    return 0;
}
```

Pentru intrarea 5 7 programul scrie 12.

```
REZULTAT
5 7
12
```

Formatarea numerelor reale

Numerele de tip double pot avea multe zecimale sau pot fi afișate în notație științifică. Ca să controlăm forma lor, includem <iomanip> și folosim fixed (afișare cu virgulă fixă) împreună cu setprecision(k), unde k este numărul de zecimale dorit.

```cpp
CPP
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    double raza;
    cin >> raza;
    double aria = 3.14159 * raza * raza;
    cout << fixed << setprecision(2);
    cout << aria << endl;
    return 0;
}
```

fixed și setprecision(2) cer afișarea cu exact 2 zecimale. Pentru raza 2 obținem aria 12.57 (rotunjit).

```
REZULTAT
2
12.57
```

Sfat pentru problemele de concurs

La problemele de concurs și la examen, formatul ieșirii trebuie respectat exact. NU afișa mesaje suplimentare de genul "Introduceți un număr:" sau "Rezultatul este:". Acele mesaje sunt utile când te joci acasă, dar verificatorul automat compară ieșirea caracter cu caracter și le-ar considera greșeli. Scrie doar ce ți se cere, în ordinea cerută.

Teste de verificare

Compilează fiecare program, introdu datele indicate și verifică dacă obții rezultatul așteptat.

Testul 1 — media a trei note. Citește trei note întregi și afișează media lor cu 2 zecimale.

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

int main() {
    int n1, n2, n3;
    cin >> n1 >> n2 >> n3;
    double media = (n1 + n2 + n3) / 3.0;
    cout << fixed << setprecision(2);
    cout << media << endl;
    return 0;
}
```

Împărțim la 3.0 (real), nu la 3, ca să nu pierdem zecimalele. Intrare: 9 8 10.

```
9 8 10
9.00
```

Testul 2 — schimbă două valori la afișare. Citește două numere și afișează-le în ordine inversă, despărțite printr-un spațiu.

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

int main() {
    int a, b;
    cin >> a >> b;
    cout << b << " " << a << endl;
    return 0;
}
```

Intrare: 3 8.

```
3 8
8 3
```

Testul 3 — perimetrul unui dreptunghi. Citește lungimea și lățimea (numere întregi) și afișează perimetrul.

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

int main() {
    int L, l;
    cin >> L >> l;
    cout << 2 * (L + l) << endl;
    return 0;
}
```

Intrare: 5 3.

```
5 3
16
```

Exerciții propuse

1. Citește un număr întreg și afișează dublul și triplul lui, despărțite printr-un spațiu.

2. Citește două numere reale și afișează suma lor cu 3 zecimale (folosește fixed și setprecision).

3. Citește raza unui cerc (număr real) și afișează circumferința (2 * 3.14159 * raza) cu 2 zecimale.

4. Citește patru numere întregi pe același rând și afișează-le câte unul pe linie, în ordinea citirii.

5. Citește un preț (real) și o cantitate (întreg) și afișează costul total cu 2 zecimale, fără niciun mesaj suplimentar.

PARTEA II

Controlul execuției

Cum ia programul decizii și cum repetă acțiuni — fundamentul oricărui algoritm.

PARTEA II

2.1 Instrucțiuni de decizie

if, if-else și switch

Până acum programele noastre au mers drept înainte: fiecare instrucțiune se executa după cea de dinainte. În realitate, însă, deciziile sunt peste tot. „Dacă plouă, iau umbrela.” „Dacă nota este peste 5, am promovat.” Calculatorul ia astfel de decizii cu ajutorul instrucțiunilor de decizie.

Instrucțiunea if

Cea mai simplă decizie verifică o condiție. Dacă aceasta este adevărată, se execută un bloc de instrucțiuni; dacă este falsă, blocul este sărit. Condiția este o expresie de tip bool (adevărat / fals).

```cpp
if (varsta >= 18) {
    cout << "Esti major." << endl;
}
```

Dacă varsta este 18 sau mai mult, se afișează mesajul; altfel programul continuă fără să afișeze nimic.

Instrucțiunea if-else

De multe ori vrem să facem un lucru când condiția este adevărată și altceva când este falsă. Pentru asta folosim if-else: exact una dintre cele două ramuri se execută.

```cpp
if (nota >= 5) {
    cout << "Promovat" << endl;
} else {
    cout << "Restanta" << endl;
}
```

Ramura else se execută numai când condiția din if este falsă. Nu se pot executa amândouă.

Exemplu: paritatea unui număr

Un număr este par dacă restul împărțirii la 2 este 0. Folosim operatorul modulo (%) și un if-else.

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

int main() {
    int n;
    cin >> n;
    if (n % 2 == 0) {
        cout << "par" << endl;
    } else {
        cout << "impar" << endl;
    }
    return 0;
}
```

Atenție: folosim == pentru comparație, nu = (care înseamnă atribuire).

```
REZULTAT (pentru n = 7)
impar
```

Importanța acoladelor

Acoladele {} grupează mai multe instrucțiuni într-un singur bloc. Dacă le uităm, if controlează doar PRIMA instrucțiune care urmează, iar restul se execută mereu. Sfatul nostru: pune întotdeauna acolade, chiar și pentru o singură linie. Așa eviți greșeli ascunse și greu de găsit.

Scara if - else if - else

Când avem mai mult de două cazuri, le înlănțuim. Se verifică condițiile pe rând, de sus în jos, iar prima care este adevărată își execută ramura; restul sunt ignorate. Ramura finală else prinde toate cazurile rămase.

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

int main() {
    int x;
    cin >> x;
    if (x < 0) {
        cout << "negativ" << endl;
    } else if (x == 0) {
        cout << "zero" << endl;
    } else {
        cout << "pozitiv" << endl;
    }
    return 0;
}
```

Determinăm semnul unui număr: negativ, zero sau pozitiv.

```
REZULTAT (pentru x = 0)
zero
```

Condiții cu operatori relaționali și logici

Condițiile se construiesc cu operatori relaționali (<, <=, >, >=, ==, !=) și se combină cu operatori logici: && (ȘI, ambele adevărate), || (SAU, măcar una adevărată) și ! (NU, inversează valoarea). Astfel exprimăm cerințe complexe într-o singură condiție.

```cpp
if (nota >= 5 && nota <= 10) {
    cout << "Nota valida." << endl;
}
```

Condiția este adevărată doar când nota este între 5 și 10 inclusiv.

Exemplu: maximul a trei numere

Combinăm operatorul logic && pentru a verifica dacă un număr este mai mare sau egal cu celelalte două.

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

int main() {
    int a, b, c;
    cin >> a >> b >> c;
    if (a >= b && a >= c) {
        cout << a << endl;
    } else if (b >= c) {
        cout << b << endl;
    } else {
        cout << c << endl;
    }
    return 0;
}
```

Dacă a nu e cel mai mare, alegem între b și c.

```
REZULTAT (pentru 3 9 5)
9
```

Exemplu complet: ecuația de gradul II

Ecuația $a \cdot x^2 + b \cdot x + c = 0$ se rezolvă cu discriminantul delta = $b^2 - 4 \cdot a \cdot c$. Dacă delta > 0 avem două soluții reale, dacă delta == 0 una singură, iar dacă delta < 0 nu există soluții reale. Avem nevoie de <cmath> pentru rădăcina pătrată sqrt.

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

int main() {
    double a, b, c;
    cin >> a >> b >> c;
    double delta = b * b - 4 * a * c;
    if (delta > 0) {
        double x1 = (-b - sqrt(delta)) / (2 * a);
        double x2 = (-b + sqrt(delta)) / (2 * a);
        cout << x1 << " " << x2 << endl;
    } else if (delta == 0) {
        cout << -b / (2 * a) << endl;
    } else {
        cout << "fara solutii reale" << endl;
    }
    return 0;
}
```

Tratăm toate cele trei cazuri ale discriminantului.

```
REZULTAT (pentru 1 -3 2)
1 2
```

Instrucțiunea switch

Când comparăm aceeași variabilă (de tip întreg sau char) cu mai multe valori fixe, switch este mai clar decât o scară lungă de if-uri. Fiecare case verifică o valoare; break oprește execuția ca să nu „curgă” în cazul următor; default prinde valorile neacoperite.

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

int main() {
    int optiune;
    cin >> optiune;
    switch (optiune) {
        case 1:
            cout << "Adunare" << endl;
            break;
        case 2:
            cout << "Scadere" << endl;
            break;
        default:
            cout << "Optiune invalida" << endl;
    }
    return 0;
}
```

Un meniu simplu. Fără break, după case 1 s-ar executa și case 2 - de aceea break este esențial.

```
REZULTAT (pentru optiune = 2)
Scadere
```

Operatorul ternar

Pentru o decizie scurtă care alege între două valori există operatorul ternar: conditie ? a : b. Dacă condiția este adevărată, rezultatul este a, altfel b. Este un if-else comprimat într-o expresie.

```cpp
int a = 7, b = 12;
int maxim = (a > b) ? a : b;
cout << maxim << endl;
```

Echivalent cu un if-else, dar pe o singură linie. Folosește-l doar pentru cazuri simple, ca să rămână lizibil.

```
REZULTAT
12
```

Teste de verificare

Compilează și rulează aceste programe; compară ieșirea cu rezultatul așteptat.

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

int main() {
    int n = 10;
    if (n % 2 == 0)
        cout << "par" << endl;
    else
        cout << "impar" << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
par
```

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

int main() {
    int nota = 8;
    string m = (nota >= 5) ? "trecut" : "picat";
    cout << m << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
trecut
```

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

int main() {
    char c = 'B';
    switch (c) {
        case 'A': cout << "unu"; break;
        case 'B': cout << "doi"; break;
        default:  cout << "altul";
    }
    cout << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
doi
```

Exerciții propuse

1. Citește un an și afișează „bisect” dacă este divizibil cu 4 și nu cu 100, sau divizibil cu 400; altfel „nebisect”.

2. Citește trei lungimi și afișează dacă pot forma laturile unui triunghi (fiecare latură mai mică decât suma celorlalte două).

3. Citește o cifră de la 1 la 7 și afișează ziua săptămânii corespunzătoare, folosind switch.

4. Citește două numere și un operator (+, -, *, /) și afișează rezultatul operației, tratând împărțirea la zero.

5. Rescrie exemplul cu maximul a trei numere folosind doar operatorul ternar, fără if.

PARTEA II

2.2 Instrucțiuni repetitive

while, do-while și for

Calculatoarele strălucesc atunci când trebuie să facă același lucru de multe ori, fără să obosească și fără să greșească. Un ciclu (sau buclă) este o instrucțiune care repetă un bloc de cod câtă vreme o condiție rămâne adevărată. În loc să scriem de o sută de ori aceeași linie, o scriem o dată și lăsăm ciclul să o repete.

while: verifică, apoi repetă

Cel mai simplu ciclu este while. El verifică o condiție la început: dacă este adevărată, execută corpul, apoi verifică din nou; dacă este falsă, iese. Programul de mai jos numără de la 1 la 5.

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

int main() {
    int i = 1;          // contor
    while (i <= 5) {    // condiția
        cout << i << " ";
        i = i + 1;      // pasul
    }
    cout << endl;
    return 0;
}
```

Atenție: dacă uităm linia i = i + 1, condiția rămâne mereu adevărată și ciclul nu se mai oprește niciodată. Acesta este un ciclu infinit.

```
REZULTAT
1 2 3 4 5
```

do-while: repetă, apoi verifică

Ciclul do-while pune condiția la sfârșit. De aceea corpul se execută cel puțin o dată, chiar dacă condiția este falsă de la început. Este potrivit când vrem să facem ceva o dată și apoi să decidem dacă repetăm, de exemplu la un meniu.

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

int main() {
    int n;
    do {
        cout << "Da un numar pozitiv: ";
        cin >> n;
    } while (n <= 0);   // se cere din nou
    cout << "Multumesc: " << n << endl;
    return 0;
}
```

Programul cere un număr până când utilizatorul introduce o valoare pozitivă. Observă punctul și virgulă obligatoriu după while(...).

```
Da un numar pozitiv: -3
Da un numar pozitiv: 0
Da un numar pozitiv: 7
Multumesc: 7
```

for: ciclul cu trei părți

Când știm de câte ori vrem să repetăm, cel mai comod este for. El strânge la un loc trei lucruri, despărțite prin ; : inițializarea (pregătim contorul), condiția (cât timp continuăm) și pasul (cum se schimbă contorul după fiecare repetare).

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

int main() {
    // initializare; conditie; pas
    for (int i = 1; i <= 5; i = i + 1) {
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

Acest for face exact ce făcea ciclul while de mai devreme, dar mai pe scurt. În practică se scrie i++ în loc de i = i + 1: înseamnă același lucru.

```
1 2 3 4 5
```

Cele trei cicluri sunt înrudite: orice for poate fi rescris ca while, iar un while poate deveni for. Alegem for când numărul de pași e cunoscut dinainte, while când repetăm „cât timp”

ceva e adevărat, și do-while când vrem măcar o execuție garantată.

Suma primelor N numere

Un model clasic: adunăm într-un acumulator. Pornim suma de la 0 și adăugăm pe rând fiecare număr de la 1 la N.

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

int main() {
    int n, suma = 0;
    cout << "N = ";
    cin >> n;
    for (int i = 1; i <= n; i++) {
        suma = suma + i;
    }
    cout << "Suma = " << suma << endl;
    return 0;
}
```

Pentru N = 5 se calculează 1+2+3+4+5 = 15.

```
N = 5
Suma = 15
```

Factorialul lui N

Factorialul (notat N!) este produsul 1·2·3·...·N. De data aceasta acumulatorul pornește de la 1, fiindcă înmulțim, nu adunăm.

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

int main() {
    int n;
    long long f = 1;    // pentru valori mari
    cout << "N = ";
    cin >> n;
    for (int i = 2; i <= n; i++) {
        f = f * i;
    }
    cout << n << "! = " << f << endl;
    return 0;
}
```

Folosim long long fiindcă factorialul crește foarte repede: 13! depășește deja un int obișnuit.

```
REZULTAT
N = 5
5! = 120
```

Numărarea cifrelor unui număr

Ca să aflăm câte cifre are un număr, îl împărțim repetat la 10, numărând câte împărțiri facem până ajungem la 0. Aici while este alegerea naturală, fiindcă nu știm dinainte numărul de pași.

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

int main() {
    int n, cifre = 0;
    cin >> n;
    if (n == 0) cifre = 1;
    while (n != 0) {
        n = n / 10;     // taie ultima cifra
        cifre = cifre + 1;
    }
    cout << "Cifre: " << cifre << endl;
    return 0;
}
```

Împărțirea întreagă 3457/10 dă 345, apoi 34, apoi 3, apoi 0 — adică patru pași, patru cifre.

```
REZULTAT
3457
Cifre: 4
```

Divizorii unui număr

Pentru a afișa divizorii lui N, încercăm pe rând fiecare număr de la 1 la N și verificăm dacă împărțirea este fără rest (restul, dat de operatorul %, este 0).

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

int main() {
    int n;
    cin >> n;
    cout << "Divizori: ";
    for (int d = 1; d <= n; d++) {
        if (n % d == 0)
            cout << d << " ";
    }
    cout << endl;
    return 0;
}
```

Pentru N = 12 divizorii sunt 1, 2, 3, 4, 6, 12.

```
12
Divizori: 1 2 3 4 6 12
```

Cicluri imbricate: tabla înmulțirii

Putem pune un ciclu în interiorul altuia. Spunem că sunt imbricate. Ciclul exterior alege rândul, iar cel interior parcurge coloanele. Pentru fiecare valoare a lui i, ciclul cu j rulează complet.

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

int main() {
    for (int i = 1; i <= 3; i++) {
        for (int j = 1; j <= 3; j++) {
            cout << i * j << " ";
        }
        cout << endl;  // rand nou
    }
    return 0;
}
```

Ciclul interior se execută complet de fiecare dată când ciclul exterior face un pas: 3 × 3 = 9 înmulțiri în total.

```
1 2 3
2 4 6
3 6 9
```

break și continue

Două instrucțiuni ne ajută să controlăm un ciclu din interior. break oprește ciclul imediat și sare după el. continue sare peste restul corpului și trece direct la următoarea repetare.

CPP

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

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 7) break;     // ne oprim
        if (i % 2 == 0) continue; // sarim
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

Afișăm numerele impare, dar ne oprim complet când ajungem la 7. continue sare peste numerele pare.

REZULTAT

```
1 3 5
```

Citirea până la o santinelă

Adesea nu știm dinainte câte valori vom citi. Atunci convenim asupra unei valori speciale, numită santinelă, care marchează sfârșitul. Aici citim numere și le adunăm până când utilizatorul tastează 0.

CPP

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

int main() {
    int x, suma = 0;
    cout << "Numere (0 = stop):" << endl;
    cin >> x;
    while (x != 0) {       // 0 e santinela
        suma = suma + x;
        cin >> x;
    }
    cout << "Total: " << suma << endl;
    return 0;
}
```

Citim prima valoare înainte de while, apoi pe fiecare următoare la sfârșitul corpului. Santinela 0 nu se adună în sumă.

```
REZULTAT
Numere (0 = stop):
4 10 6 0
Total: 20
```

Pericolul ciclului infinit

De evitat:

Uitarea pasului care schimbă contorul (i++). O condiție care rămâne mereu adevărată. ; pus din greșeală imediat după for(...) sau while(...), ceea ce lasă corpul gol.

Dacă un program pare „blocat” și nu se mai termină, cel mai probabil ai un ciclu infinit. Verifică dacă valoarea din condiție chiar se modifică în corp, astfel încât condiția să devină la un moment dat falsă. Poți opri programul cu Ctrl+C.

Teste de verificare

Scrie, compilează și rulează programele de mai jos. Compară rezultatul tău cu cel așteptat.

CPP

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

int main() {
    for (int i = 10; i >= 1; i--) {
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
10 9 8 7 6 5 4 3 2 1
```

CPP

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

int main() {
    int s = 0;
    for (int i = 1; i <= 4; i++)
        s = s + i * i;   // patrate
    cout << "Suma patratelor: " << s << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
Suma patratelor: 30
```

CPP

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

int main() {
    for (int i = 1; i <= 4; i++) {
        for (int j = 1; j <= i; j++)
            cout << "*";
        cout << endl;
    }
    return 0;
}
```

REZULTAT AȘTEPTAT

```
*
**
***
****
```

Exerciții propuse

Rezolvă singur:

1. Afișează numerele pare de la 2 la 20. 2. Citește N și calculează 1+2+...+N folosind while. 3. Citește un număr și afișează suma cifrelor lui. 4. Afișează tabla înmulțirii completă (1..10). 5. Citește numere până la -1 și afișează cel mai mare dintre ele.

PARTEA II

2.3 Tehnici elementare de calcul

Cifre, divizori, numere prime

Multe probleme de informatică pornesc de la un singur număr întreg și cer ceva despre el: câte cifre are, dacă este prim, care sunt divizorii lui. În acest capitol învățăm tehnicile clasice care rezolvă astfel de cerințe folosind doar cicluri și două operații-cheie: împărțirea întreagă și restul.

Prelucrarea cifrelor unui număr

Trucul fundamental: ultima cifră a unui număr este restul împărțirii la 10, iar dacă împărțim numărul (întreg) la 10 îi ștergem ultima cifră. Repetând până numărul devine 0, parcurgem toate cifrele, de la coadă spre cap.

```cpp
int n = 5273;
while (n > 0) {
    int cifra = n % 10;  // ultima cifra
    n = n / 10;          // o stergem
}
```

La pașii succesivi cifra ia valorile 3, 7, 2, 5, iar n devine 527, 52, 5, 0.

Suma cifrelor și numărul de cifre

Folosind același ciclu, adunăm cifrele într-un acumulator și numărăm de câte ori se repetă bucla.

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

int main() {
    long long n;
    cin >> n;
    int suma = 0, nrCifre = 0;
    while (n > 0) {
        suma += n % 10;
        nrCifre++;
        n /= 10;
    }
    cout << suma << ' ' << nrCifre;
    return 0;
}
```

Atenție: dacă n poate fi 0, tratează-l separat, fiindcă bucla nu pornește pentru n = 0.

```
REZULTAT (intrare: 5273)
17 4
```

Oglinditul unui număr și palindrom

Oglinditul (răsturnatul) se construiește mutând cifra extrasă în noul număr: îl înmulțim cu 10 și adăugăm cifra curentă. Un număr este palindrom dacă este egal cu oglinditul său (ex. 1221).

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

int main() {
    long long n;
    cin >> n;
    long long copie = n, ogl = 0;
    while (copie > 0) {
        ogl = ogl * 10 + copie % 10;
        copie /= 10;
    }
    cout << ogl << '\n';
    if (ogl == n)
        cout << "palindrom";
    else
        cout << "nu este palindrom";
    return 0;
}
```

Salvăm n într-o copie, pentru că bucla îl distruge, iar la final mai avem nevoie de valoarea inițială.

```
REZULTAT (intrare: 1221)
1221
palindrom
```

Divizibilitate și divizori

Spunem că d divide pe n dacă restul împărțirii este zero: n % d == 0. Pentru a afișa toți divizorii lui n, încercăm pe rând fiecare candidat de la 1 la n.

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

int main() {
    int n;
    cin >> n;
    for (int d = 1; d <= n; d++)
        if (n % d == 0)
            cout << d << ' ';
    return 0;
}
```

Simplu, dar lent pentru numere mari: facem n pași.

```
REZULTAT (intrare: 12)
1 2 3 4 6 12
```

Optimizare: până la radical

Divizorii vin în perechi: dacă d divide pe n, atunci și n/d îl divide. Unul dintre ei este mereu cel mult egal cu radicalul lui n. Deci e suficient să căutăm până la sqrt(n) și să folosim perechea. Astfel numărăm divizorii și le aflăm suma mult mai repede.

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

int main() {
    long long n;
    cin >> n;
    int nrDiv = 0;
    long long suma = 0;
    for (long long d = 1; d * d <= n; d++) {
        if (n % d == 0) {
            nrDiv += 2;
            suma += d + n / d;
            if (d == n / d) {  // patrat perfect
                nrDiv--;       // d numarat o data
                suma -= d;
            }
        }
    }
    cout << nrDiv << ' ' << suma;
    return 0;
}
```

*Condiția d*d <= n evită sqrt și erorile de rotunjire. Dacă d și n/d coincid (n e pătrat perfect), corectăm ca să nu numărăm divizorul de două ori.*

```
REZULTAT (intrare: 36)
9 91
```

Testul de primalitate

Un număr n ≥ 2 este prim dacă nu are niciun divizor între 2 și n-1. Folosind aceeași idee a radicalului, verificăm divizorii doar până la sqrt(n): dacă n nu se împarte la niciunul, este prim.

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

bool estePrim(long long n) {
    if (n < 2) return false;
    for (long long d = 2; d * d <= n; d++)
        if (n % d == 0)
            return false;
    return true;
}

int main() {
    long long n;
    cin >> n;
    if (estePrim(n))
        cout << "prim";
    else
        cout << "nu este prim";
    return 0;
}
```

Numerele 0 și 1 nu sunt prime, de aceea le respingem din start. Pentru n = 2 bucla nu pornește și răspunsul rămâne corect: prim.

```
REZULTAT (intrare: 97)
prim
```

Ciurul lui Eratostene

Când avem nevoie de toate numerele prime până la o limită, în loc să testăm fiecare separat folosim ciurul. Pornim presupunând că toate sunt prime, apoi tăiem multiplii fiecărui număr prim găsit. Ce rămâne netăiat este prim.

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

int main() {
    const int N = 30;
    bool prim[N + 1];
    for (int i = 0; i <= N; i++)
        prim[i] = true;
    prim[0] = prim[1] = false;
    for (int i = 2; i * i <= N; i++)
        if (prim[i])
            for (int j = i * i; j <= N; j += i)
                prim[j] = false;
    for (int i = 2; i <= N; i++)
        if (prim[i])
            cout << i << ' ';
    return 0;
}
```

*Tăierea pornește de la i*i, fiindcă multiplii mai mici (2*i, 3*i, ...) au fost deja eliminați de numerele prime anterioare.*

```
2 3 5 7 11 13 17 19 23 29
```

Cel mai mare divizor comun (cmmdc)

Algoritmul lui Euclid găsește cmmdc fără să caute divizori. Varianta cu scădere: cât timp numerele sunt diferite, scădem pe cel mic din cel mare. Când devin egale, aceea este valoarea căutată.

```cpp
int cmmdcScadere(int a, int b) {
    while (a != b) {
        if (a > b) a -= b;
        else       b -= a;
    }
    return a;
}
```

Funcționează pentru numere strict pozitive.

Varianta cu modulo este mult mai rapidă: înlocuim perechea (a, b) cu (b, a % b) până când al doilea număr devine 0. Atunci primul este cmmdc.

```cpp
int cmmdc(int a, int b) {
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

Aceasta este forma folosită aproape mereu la concurs.

Cel mai mic multiplu comun (cmmmc)

Din cmmdc obținem imediat cmmmc cu formula cmmmc = a * b / cmmdc(a, b). Ca să evităm depășirea, împărțim întâi: a / cmmdc * b.

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

int cmmdc(int a, int b) {
    while (b != 0) {
        int r = a % b; a = b; b = r;
    }
    return a;
}

int main() {
    int a, b;
    cin >> a >> b;
    int d = cmmdc(a, b);
    long long m = (long long)a / d * b;
    cout << d << ' ' << m;
    return 0;
}
```

*Conversia la long long protejează produsul a*b de depășire când numerele sunt mari.*

```
REZULTAT (intrare: 12 18)
6 36
```

Teste de verificare

Compilează fiecare program, rulează-l cu intrarea indicată și compară rezultatul cu cel așteptat.

Testul 1 — suma cifrelor și produsul lor:

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

int main() {
    long long n = 1234;
    int suma = 0;
    long long produs = 1;
    while (n > 0) {
        int c = n % 10;
        suma += c;
        produs *= c;
        n /= 10;
    }
    cout << suma << ' ' << produs;
    return 0;
}
```

```
10 24
```

Testul 2 — câte numere prime sunt sub 20:

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

bool estePrim(int n) {
    if (n < 2) return false;
    for (int d = 2; d * d <= n; d++)
        if (n % d == 0) return false;
    return true;
}

int main() {
    int cnt = 0;
    for (int i = 2; i < 20; i++)
        if (estePrim(i)) cnt++;
    cout << cnt;
    return 0;
}
```

```
8
```

Testul 3 — cmmdc și cmmmc pentru 24 și 36:

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

int cmmdc(int a, int b) {
    while (b != 0) {
        int r = a % b; a = b; b = r;
    }
    return a;
}

int main() {
    int a = 24, b = 36;
    int d = cmmdc(a, b);
    cout << d << ' ' << a / d * b;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
12 72
```

Exerciții propuse

1. Citește un număr natural și afișează prima și ultima sa cifră, apoi spune dacă ele sunt egale.

2. Verifică dacă un număr citit este palindrom, fără a-i construi explicit oglinditul (compară cifrele de la capete folosind puteri ale lui 10).

3. Afișează toți divizorii proprii ai unui număr (toți în afară de el însuși) folosind căutarea până la radical.

4. Un număr se numește perfect dacă este egal cu suma divizorilor săi proprii (ex. 6 = 1 + 2 + 3). Verifică dacă numărul citit este perfect.

5. Citește două numere și afișează cmmdc-ul lor folosind ambele variante (cu scădere și cu modulo), verificând că dau același rezultat.

PARTEA III

Date structurate

Cum păstrăm și prelucrăm colecții de date: vectori, matrice, șiruri, căutare și sortare.

PARTEA III

3.1 Tablouri unidimensionale

Vectori: declarare, parcurgere, prelucrare

Până acum, fiecare valoare avea propria ei variabilă. Dar dacă vrei să lucrezi cu notele a 30 de elevi? Ai declara 30 de variabile? Ar fi un coșmar. Soluția se numește tablou (sau vector): un singur nume sub care păstrezi multe valori de același tip, numerotate.

Idee de reținut:

Un tablou e un sir de cutii, una langa alta. Toate au acelasi nume, dar un numar de ordine (indice). a[0], a[1], a[2], ... sunt elementele tabloului a.

Declararea unui tablou

Declari un tablou spunând tipul elementelor, numele și, în paranteze drepte, câte locuri rezervi. La concursuri și la bacalaureat se folosesc tablouri statice clasice, cu o dimensiune fixă, generoasă.

```cpp
int a[100];        // 100 de intregi: a[0]..a[99]
double note[50];   // 50 de numere zecimale
int n;             // cate elemente folosim de fapt
```

Rezervi mai mult loc decât îți trebuie (de exemplu 100), dar folosești doar primele n poziții.

Există și std::vector, un tablou care își schimbă singur mărimea. Este foarte util și îl studiem în detaliu la capitolul despre STL; aici rămânem la tablourile statice, mai apropiate de stilul de examen.

Indexarea începe de la 0

Aceasta este regula care încurcă cel mai mult la început. Primul element are indicele 0, nu 1. Deci un tablou cu n elemente le ține pe pozițiile de la 0 până la n-1. Indicele n nu mai există: a-l folosi înseamnă depășirea limitelor, o greșeală gravă.

```cpp
int a[5];
a[0] = 10;   // primul element
a[1] = 20;
a[4] = 50;   // ultimul (indice n-1 = 4)
// a[5] = 99; // GRESIT: iese din tablou!
```

Indicii valizi pentru a[5] sunt 0, 1, 2, 3, 4. Accesul în afara lor poate strica alte date sau bloca programul.

Citirea unui vector de n elemente

Tiparul standard: citești mai întâi câte elemente sunt (n), apoi citești pe rând valorile, folosind o buclă for care plimbă indicele de la 0 la n-1.

```cpp
int n;
int a[100];
cin >> n;                  // cate numere
for (int i = 0; i < n; i++)
    cin >> a[i];           // citeste elementul i
```

Condiția i < n este cheia: oprim exact înainte de indicele n, care nu mai e valid.

Afișarea elementelor

La fel parcurgem tabloul ca să-l afișăm: tot cu un for de la 0 la n-1. Punem un spațiu între valori ca să se citească frumos.

```cpp
for (int i = 0; i < n; i++)
    cout << a[i] << " ";
cout << endl;
```

În toate exemplele care urmează presupunem că am citit deja acest vector cu n = 6 elemente:

```
VECTOR EXEMPLU
n = 6
a = 4 8 15 16 23 42
```

Suma elementelor și media

Pornim de la o sumă egală cu 0 și adăugăm pe rând fiecare element. Media e suma împărțită la n; ca să iasă zecimală, împărțim la (double)n.

```cpp
int suma = 0;
for (int i = 0; i < n; i++)
    suma += a[i];
double media = (double)suma / n;
cout << "Suma: " << suma << endl;
cout << "Media: " << media << endl;
```

Conversia (double)suma forțează o împărțire zecimală, altfel am pierde partea fracționară.

```
Suma: 108
Media: 18
```

Minimul, maximul și pozițiile lor

Presupunem la început că primul element este și minimul, și maximul. Apoi parcurgem restul: dacă găsim ceva mai mic, actualizăm minimul; la fel pentru maxim. Reținem și poziția (indicele) lor.

```cpp
int minim = a[0], pmin = 0;
int maxim = a[0], pmax = 0;
for (int i = 1; i < n; i++) {
    if (a[i] < minim) { minim = a[i]; pmin = i; }
    if (a[i] > maxim) { maxim = a[i]; pmax = i; }
}
cout << "Min " << minim << " pe " << pmin;
cout << endl;
cout << "Max " << maxim << " pe " << pmax;
cout << endl;
```

Pornim bucla de la i = 1, fiindcă elementul 0 e deja presupus minim și maxim.

```
Min 4 pe 0
Max 42 pe 5
```

Numărarea elementelor cu o proprietate

Vrem, de exemplu, câte numere pare sunt în vector. Folosim un contor pornit de la 0 și îl creștem ori de câte ori elementul curent respectă condiția.

```cpp
int cate = 0;
for (int i = 0; i < n; i++)
    if (a[i] % 2 == 0)   // numar par
        cate++;
cout << "Numere pare: " << cate << endl;
```

```
Numere pare: 4
```

Căutarea unei valori

Parcurgem vectorul și comparăm fiecare element cu valoarea căutată. Dacă o găsim, reținem poziția și ne oprim. Dacă bucla se termină fără succes, poziția rămâne -1, un semn că valoarea lipsește.

```cpp
int x = 16;     // valoarea cautata
int gasit = -1;
for (int i = 0; i < n; i++)
    if (a[i] == x) { gasit = i; break; }
if (gasit != -1)
    cout << "Gasit pe " << gasit << endl;
else
    cout << "Negasit" << endl;
```

break oprește căutarea de îndată ce am dat de valoare; nu mai are rost să continuăm.

```
Gasit pe 3
```

Afișarea inversată

Ca să afișăm vectorul de la coadă la cap, plimbăm indicele invers: pornim de la n-1 și coborâm până la 0. Tabloul în sine nu se schimbă, doar ordinea de afișare.

```cpp
for (int i = n - 1; i >= 0; i--)
    cout << a[i] << " ";
cout << endl;
```

Condiția i >= 0 și pasul i-- ne duc invers prin toate pozițiile valide.

```
42 23 16 15 8 4
```

Exemplu complet

Punem laolaltă citirea, suma, media, maximul și afișarea inversată într-un singur program.

CPP

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

int main() {
    int n, a[100];
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int suma = 0, maxim = a[0];
    for (int i = 0; i < n; i++) {
        suma += a[i];
        if (a[i] > maxim) maxim = a[i];
    }
    cout << "Suma: " << suma << endl;
    cout << "Media: " << (double)suma / n;
    cout << endl;
    cout << "Maxim: " << maxim << endl;
    cout << "Invers: ";
    for (int i = n - 1; i >= 0; i--)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

Date de intrare: 6 apoi 4 8 15 16 23 42

REZULTAT

```
Suma: 108
Media: 18
Maxim: 42
Invers: 42 23 16 15 8 4
```

Teste de verificare

Compilează și rulează fiecare program; compară-ți ieșirea cu rezultatul așteptat.

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

int main() {
    int a[5] = {3, 1, 4, 1, 5};
    int cate = 0;
    for (int i = 0; i < 5; i++)
        if (a[i] == 1) cate++;
    cout << cate << endl;
    return 0;
}
```

Numără de câte ori apare valoarea 1.

```
REZULTAT AȘTEPTAT
2
```

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

int main() {
    int a[4] = {7, 2, 9, 5};
    int minim = a[0];
    for (int i = 1; i < 4; i++)
        if (a[i] < minim) minim = a[i];
    cout << minim << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
2
```

Exerciții propuse

Rezolvă singur:

1. Citeste n numere intr-un vector si afiseaza suma elementelor de pe pozitii pare (0, 2, 4...). 2. Citeste un vector si numara cate elemente sunt mai mari decat media tuturor elementelor. 3. Citeste un vector si afiseaza pozitia primului element negativ (sau -1 daca nu exista). 4. Citeste un vector si afiseaza valorile lui in ordine inversa, fara sa modifici tabloul. 5. Citeste un vector si gaseste a doua cea mai mare valoare din el (presupune valori distincte).

PARTEA III

3.2 Tablouri bidimensionale

Matrice și prelucrarea lor

Un vector așază numerele într-un singur rând. Dar multe date din lumea reală sunt organizate ca un tabel: catalogul clasei (elevi pe linii, materii pe coloane), tabla de șah, o imagine alb-negru. Pentru ele folosim un tablou bidimensional, numit matrice.

Idee de reținut:

O matrice este un tabel cu linii si coloane. Un element are doua adrese: linia i si coloana j. Il scriem a[i][j] (intai linia, apoi coloana).

Declararea unei matrice

Declarăm o matrice spunând tipul, numele și două dimensiuni în paranteze drepte separate: numărul de linii și numărul de coloane. La bacalaureat se rezervă static loc mai mult decât e nevoie, ca să încapă orice date primite.

```cpp
int a[100][100];   // pana la 100 linii, 100 col.
int n, m;          // dimensiunile reale folosite
```

Liniile sunt numerotate 0..n-1, iar coloanele 0..m-1. Folosim doar colțul din stânga-sus, de dimensiune n pe m.

Cei doi indici

Fiecare element are nevoie de doi indici: primul alege linia, al doilea alege coloana. Astfel a[1][2] este elementul de pe linia 1, coloana 2. Schimbi un indice, te muți cu un pas în tabel.

```cpp
a[0][0] = 5;       // coltul stanga-sus
a[2][3] = 8;       // linia 2, coloana 3
cout << a[0][0] << ' ' << a[2][3];
```

Citirea unei matrice

Ca să umplem tabelul, avem nevoie de două cicluri unul în altul (imbricate). Ciclul exterior trece prin linii, iar cel interior, pentru fiecare linie, trece prin toate coloanele. Citim element cu element, în ordinea liniilor.

```cpp
cin >> n >> m;
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        cin >> a[i][j];
```

Variabila i numără liniile, j numără coloanele. Numele i și j sunt o convenție foarte răspândită.

Afișarea frumoasă, aliniată

Afișăm tot cu cicluri imbricate, dar după fiecare linie completă trecem la rândul următor cu endl. Punem un spațiu între numere ca să nu se lipească. Pentru aliniere putem folosi setw din <iomanip>, care rezervă o lățime fixă pentru fiecare număr.

```cpp
#include <iomanip>   // pentru setw
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
        cout << setw(4) << a[i][j];
    cout << endl;   // rand nou dupa o linie
```

setw(4) face fiecare număr să ocupe 4 poziții, deci coloanele ies drepte chiar dacă numerele au câte cifre diferite.

Exemplu complet: citire și afișare

Punem cap la cap citirea și afișarea unei matrice. Programul primește dimensiunile și elementele, apoi le tipărește înapoi sub formă de tabel.

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

int a[100][100], n, m;

int main() {
    cin >> n >> m;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> a[i][j];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            cout << setw(4) << a[i][j];
        cout << endl;
    }
    return 0;
}
```

Pentru intrarea: 2 3 / 1 2 3 / 4 5 6

```
REZULTAT
   1   2   3
   4   5   6
```

Parcurgeri: pe linii și pe coloane

Aceeași matrice poate fi străbătută în două feluri. Pe linii: fixezi o linie și parcurgi toate coloanele ei, apoi treci la linia următoare. Pe coloane: fixezi o coloană și cobori prin toate liniile. Diferența e doar care ciclu stă în exterior.

```cpp
// pe coloane: j fix in exterior, i in interior
for (int j = 0; j < m; j++)
    for (int i = 0; i < n; i++)
        cout << a[i][j] << ' ';
```

Acum afișăm întâi toată coloana 0, apoi coloana 1 și așa mai departe.

Suma pe fiecare linie

Pentru fiecare linie vrem suma elementelor ei. Ținem un acumulator s pe care îl punem la zero la începutul fiecărei linii, adunăm toate coloanele, apoi afișăm suma liniei respective.

```cpp
for (int i = 0; i < n; i++) {
    int s = 0;
    for (int j = 0; j < m; j++)
        s += a[i][j];
    cout << "Linia " << i << ": " << s << endl;
}
```

Suma pe fiecare coloană

Analog, pentru suma fiecărei coloane punem coloana în ciclul exterior. Acumulatorul se resetează la începutul fiecărei coloane, iar înăuntru coborâm prin toate liniile.

```cpp
for (int j = 0; j < m; j++) {
    int s = 0;
    for (int i = 0; i < n; i++)
        s += a[i][j];
    cout << "Coloana " << j << ": " << s << endl;
}
```

Matrice pătratică și diagonalele

O matrice este pătratică dacă are tot atâtea linii câte coloane (n = m). Numai la ea au sens cele două diagonale. Diagonala principală merge din colțul stânga-sus spre dreapta-jos: pe ea linia este egală cu coloana, deci i == j. Diagonala secundară merge din dreapta-sus spre stânga-jos: pe ea i + j == n-1.

Condițiile de reținut:

diagonala principala: i == j diagonala secundara: i + j == n - 1

```cpp
int dp = 0, ds = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
        if (i == j) dp += a[i][j];
        if (i + j == n - 1) ds += a[i][j];
    }
cout << dp << ' ' << ds << endl;
```

dp adună diagonala principală, ds pe cea secundară. Pe o matrice pătratică folosim n și pentru linii, și pentru coloane.

Transpusa unei matrice

Transpusa schimbă liniile cu coloanele: elementul de pe linia i, coloana j ajunge pe linia j, coloana i. Dacă matricea de pornire are n linii și m coloane, transpusa are m linii și n coloane. O construim într-o a doua matrice, t.

```cpp
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        t[j][i] = a[i][j];   // schimba indicii
```

Observă cum indicii sunt inversați la scriere: ceea ce era a[i][j] devine t[j][i].

Maximul și poziția lui

Ca să găsim cel mai mare element, presupunem la început că maximul e primul element, a[0][0], și îi reținem poziția. Parcurgem apoi toată matricea; ori de câte ori găsim o valoare mai mare, actualizăm maximul și poziția (linia și coloana lui).

```cpp
int maxim = a[0][0], li = 0, co = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        if (a[i][j] > maxim) {
            maxim = a[i][j];
            li = i; co = j;
        }
cout << maxim << " la (" << li
     << "," << co << ")" << endl;
```

li și co rețin linia și coloana unde se află maximul găsit până în acel moment.

Teste de verificare

Compilează și rulează fiecare program; compară-ți ieșirea cu rezultatul așteptat. La citire, dă mai întâi dimensiunile, apoi elementele pe linii.

```cpp
#include <iostream>
using namespace std;
int a[100][100], n, m;
int main() {
    cin >> n >> m;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> a[i][j];
    for (int i = 0; i < n; i++) {
        int s = 0;
        for (int j = 0; j < m; j++)
            s += a[i][j];
        cout << s << endl;
    }
    return 0;
}
```

Suma fiecărei linii. Intrare: 2 3 / 1 2 3 / 4 5 6

```
REZULTAT AȘTEPTAT
6
15
```

CPP

```cpp
#include <iostream>
using namespace std;
int a[100][100], n;
int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> a[i][j];
    int dp = 0;
    for (int i = 0; i < n; i++)
        dp += a[i][i];   // i == j
    cout << dp << endl;
    return 0;
}
```

Suma diagonalei principale. Intrare: 3 / 1 2 3 / 4 5 6 / 7 8 9 (adica 1+5+9)

REZULTAT AȘTEPTAT

```
15
```

CPP

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int a[100][100], t[100][100], n, m;
int main() {
    cin >> n >> m;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++) {
            cin >> a[i][j];
            t[j][i] = a[i][j];
        }
    for (int j = 0; j < m; j++) {
        for (int i = 0; i < n; i++)
            cout << setw(3) << t[j][i];
        cout << endl;
    }
    return 0;
}
```

Transpusa. Intrare: 2 3 / 1 2 3 / 4 5 6

REZULTAT AȘTEPTAT

```
  1  4
  2  5
  3  6
```

Exerciții propuse

Rezolvă singur:

1. Citeste o matrice n x m si afiseaza media aritmetica a tuturor elementelor ei. 2. Citeste o matrice patratica si afiseaza suma elementelor de pe diagonala secundara. 3. Gaseste elementul minim dintr-o matrice si afiseaza valoarea lui impreuna cu linia si coloana pe care se afla. 4. Numara cate elemente pare are matricea, apoi cate sunt pe fiecare linie in parte. 5. Verifica daca o matrice patratica este simetrica fata de diagonala principala (adica a[i][j] == a[j][i] pentru orice i, j).

PARTEA III

3.3 Șiruri de caractere

char[] și std::string

Până acum am lucrat cu numere. Dar programele prelucrează adesea text: nume, cuvinte, propoziții. Unitatea de bază a textului este caracterul, iar un text întreg este un șir de caractere. În acest capitol vedem cum reprezintă C++ caracterele și cum lucrăm cu șiruri în două moduri: stilul clasic char[] și stilul modern, recomandat, std::string.

Caracterul: tipul char

Un caracter individual se păstrează în tipul char. Scriem o constantă caracter între apostrofuri: 'A', 'z', '5', ' '. În spate, fiecare caracter este de fapt un număr mic: codul lui ASCII. De aceea un char poate fi folosit și ca un întreg.

CPP

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

int main() {
    char c = 'A';
    cout << c << ' ' << (int)c << endl;
    return 0;
}
```

(int)c forțează afișarea codului numeric. Litera 'A' are codul ASCII 65.

REZULTAT

```
A 65
```

Codul ASCII și aritmetica pe caractere

Pentru că un char este un număr, putem face calcule cu el. Literele mari 'A'..'Z' au coduri consecutive (65..90), literele mici 'a'..'z' tot consecutive (97..122), iar cifrele '0'..'9' au codurile 48..57. Aceste regularități ne dau conversii simple.

CPP

```cpp
char litera = 'a' + 3;     // a patra literă mică
char majuscula = 'd' - 32; // 'd' devine 'D'
int valoare = '7' - '0';   // cifra ca număr
cout << litera << ' ' << majuscula
     << ' ' << valoare << endl;
```

Diferența dintre o literă mică și corespondenta mare este mereu 32. Iar '7' - '0' transformă caracterul cifrei în valoarea ei întreagă, 7.

```
REZULTAT
d D 7
```

Șirul clasic: char s[100]

Un șir în stil C este un vector de caractere care se termină cu caracterul special '\0' (caracterul nul). Acel '\0' marchează sfârșitul textului, așa că tabloul trebuie să aibă mereu un loc în plus față de numărul de litere. Declarăm de obicei generos, ex. char s[100].

```cpp
char s[100];
cin >> s;          // citește un cuvânt
cout << s << endl; // afișează ce a citit
```

Atenție: cin >> s se oprește la primul spațiu! Dacă tastezi „Ana are mere”, în s ajunge doar „Ana”.

```
REZULTAT
Ana are mere
Ana
```

Citirea unei linii întregi cu getline

Ca să citim o propoziție cu tot cu spații într-un char[], folosim cin.getline(s, dimensiune). Aceasta citește până la apăsarea tastei Enter.

```cpp
char s[100];
cin.getline(s, 100);
cout << s << endl;
```

Acum tot textul de pe linie, inclusiv spațiile, ajunge în s.

```
REZULTAT
Ana are mere
Ana are mere
```

Funcții din <cstring>

Pentru șirurile clasice, biblioteca <cstring> oferă funcții utile. Le menționăm pe cele mai folosite: strlen(s) dă lungimea (numărul de caractere până la '\0'), strcpy(dest, sursa) copiază un șir în altul, iar strcmp(a, b) compară două șiruri (întoarce 0 dacă sunt egale). Sunt utile, dar incomode; de aceea trecem la varianta modernă.

```cpp
#include <cstring>
char s[100] = "informatica";
cout << strlen(s) << endl;  // 11 caractere
```

strlen numără literele, fără să includă '\0'.

```
REZULTAT
11
```

std::string: varianta recomandată

Tipul std::string (din <string>) este un șir „deștept”: își gestionează singur memoria, crește cât e nevoie și are operații comode. Pentru tot ce scrii de aici înainte, folosește string în loc de char[].

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

int main() {
    string nume;
    cin >> nume;        // citește un cuvânt
    cout << "Salut, " << nume << "!" << endl;
    return 0;
}
```

La fel ca la char[], cin >> nume citește un singur cuvânt (se oprește la spațiu).

```
REZULTAT
Maria
Salut, Maria!
```

Citirea unei linii cu getline

Pentru a citi o propoziție întreagă într-un string, folosim funcția getline(cin, s). Ea citește toată linia până la Enter, spațiile incluse.

```cpp
string propozitie;
getline(cin, propozitie);
cout << propozitie << endl;
```

getline(cin, ...) este pentru string; cin.getline(...) este pentru char[]. Nu le confunda.

```
REZULTAT
Ana are mere
Ana are mere
```

Operațiile pe string

Un string oferă tot ce ne trebuie. Lungimea o aflăm cu .length() sau .size() (sunt identice). Accesăm caracterul de pe poziția i cu s[i] (numerotare de la 0). Lipim șiruri cu operatorul +. Comparăm cu == sau cu < (ordine alfabetică).

```cpp
string a = "abc", b = "def";
string c = a + b;        // concatenare
cout << c << endl;
cout << c.length() << endl;
cout << c[0] << endl;    // primul caracter
cout << (a == b) << endl;// 0 = diferite
```

+ produce „abcdef”; c[0] este 'a'; a == b dă fals, adică 0.

```
REZULTAT
abcdef
6
a
0
```

Subșiruri și căutare: substr și find

.substr(poz, lung) extrage o porțiune din șir, începând de la poziția poz, de lungime lung. .find(text) caută text în șir și întoarce poziția primei apariții (sau valoarea specială string::npos dacă nu îl găsește).

```cpp
string s = "programare";
cout << s.substr(0, 4) << endl; // primele 4
cout << s.find("gram") << endl;// poziția
```

substr(0, 4) ia „prog”; find("gram") întoarce 3, fiindcă „gram” începe pe poziția 3.

```
REZULTAT
prog
3
```

Parcurgerea caracterelor

Cel mai des prelucrăm un șir caracter cu caracter, într-un ciclu de la 0 la lungime - 1. Aceasta este baza majorității algoritmilor pe text.

```cpp
string s = "abc";
for (int i = 0; i < s.length(); i++)
    cout << s[i] << '-';
cout << endl;
```

Afișăm fiecare caracter urmat de o liniuță.

```
REZULTAT
a-b-c-
```

Exemplu: numără vocalele

Parcurgem șirul și creștem un contor de fiecare dată când întâlnim o vocală (mare sau mică).

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

int main() {
    string s;
    getline(cin, s);
    int nr = 0;
    string voc = "aeiouAEIOU";
    for (int i = 0; i < s.length(); i++)
        if (voc.find(s[i]) != string::npos)
            nr++;
    cout << nr << endl;
    return 0;
}
```

Folosim find în șirul de vocale: dacă litera curentă se găsește acolo, e vocală.

```
REZULTAT
Ana are mere
5
```

Exemplu: transformă în majuscule

Pentru fiecare literă mică (între 'a' și 'z'), scădem 32 din codul ei ca să obținem majuscula. Restul caracterelor rămân neschimbate.

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

int main() {
    string s;
    getline(cin, s);
    for (int i = 0; i < s.length(); i++)
        if (s[i] >= 'a' && s[i] <= 'z')
            s[i] = s[i] - 32;
    cout << s << endl;
    return 0;
}
```

s[i] = s[i] - 32 ridică litera la majusculă, folosind aritmetica pe coduri ASCII.

```
Salut, lume!
SALUT, LUME!
```

Exemplu: verifică palindrom

Un palindrom se citește la fel de la stânga la dreapta și invers (ex. „cojoc”). Comparăm caracterul de pe poziția i cu cel simetric de la capăt, s[lung-1-i].

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

int main() {
    string s;
    cin >> s;
    int n = s.length();
    bool palindrom = true;
    for (int i = 0; i < n / 2; i++)
        if (s[i] != s[n - 1 - i])
            palindrom = false;
    cout << (palindrom ? "DA" : "NU");
    cout << endl;
    return 0;
}
```

Verificăm doar prima jumătate; dacă vreo pereche diferă, nu este palindrom.

```
cojoc
DA
```

Exemplu: numără cuvintele dintr-o propoziție

Un cuvânt începe acolo unde o literă urmează după un spațiu (sau la începutul șirului). Numărăm fiecare astfel de început.

CPP

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

int main() {
    string s;
    getline(cin, s);
    int nr = 0;
    for (int i = 0; i < s.length(); i++)
        if (s[i] != ' ' &&
            (i == 0 || s[i - 1] == ' '))
            nr++;
    cout << nr << endl;
    return 0;
}
```

Numărăm tranzițiile de la spațiu la literă, deci fiecare cuvânt este numărat exact o dată.

REZULTAT

```
Ana are mere coapte
4
```

Teste de verificare

Compilează fiecare program, introdu datele indicate și verifică rezultatul așteptat.

Testul 1 — lungimea unui cuvânt. Citește un cuvânt și afișează câte caractere are.

CPP

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

int main() {
    string s;
    cin >> s;
    cout << s.length() << endl;
    return 0;
}
```

Intrare: informatica.

REZULTAT AȘTEPTAT

```
informatica
11
```

Testul 2 — numără literele mari. Citește o linie și afișează câte litere mari ('A'..'Z') conține.

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

int main() {
    string s;
    getline(cin, s);
    int nr = 0;
    for (int i = 0; i < s.length(); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            nr++;
    cout << nr << endl;
    return 0;
}
```

Intrare: Ana Maria Pop.

```
Ana Maria Pop
3
```

Testul 3 — inversează un cuvânt. Citește un cuvânt și afișează-l scris invers.

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

int main() {
    string s;
    cin >> s;
    for (int i = s.length() - 1; i >= 0; i--)
        cout << s[i];
    cout << endl;
    return 0;
}
```

Intrare: liceu.

```
liceu
uecil
```

Exerciții propuse

1. Citește un caracter și afișează codul lui ASCII (folosește (int)c).

2. Citește un cuvânt și afișează câte consoane are (literele care nu sunt vocale).

3. Citește o linie și afișează-o cu toate literele mari transformate în litere mici.

4. Citește două cuvinte și afișează DA dacă sunt egale (==), altfel NU.

5. Citește o linie și un caracter, apoi afișează de câte ori apare acel caracter în linie.

PARTEA III

3.4 Căutare și sortare

Algoritmi fundamentali pe vectori

Două dintre cele mai des întâlnite operații pe vectori sunt căutarea (unde se află un element?) și sortarea (aranjarea elementelor în ordine). Aproape orice program serios le folosește, de la o agendă telefonică la un motor de căutare. În acest capitol învățăm cum funcționează și le scriem de la zero, ca să înțelegem ce se petrece „pe dedesubt".

Căutarea secvențială (liniară)

Cea mai simplă căutare parcurge vectorul element cu element, de la stânga la dreapta, și compară fiecare valoare cu cea căutată. Dacă o găsește, returnează poziția; dacă a ajuns la capăt fără succes, returnează -1. Funcționează pe orice vector, sortat sau nu.

```cpp
int cautaLiniar(int v[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (v[i] == x) {
            return i;  // gasit pe pozitia i
        }
    }
    return -1;  // nu exista in vector
}
```

Returnăm indicele primei apariții, sau -1 dacă valoarea nu se găsește.

În cel mai rău caz (elementul lipsește sau e ultimul) facem n comparații. Pentru un vector cu un milion de elemente, asta înseamnă un milion de pași — mult.

Căutarea binară (într-un vector SORTAT)

Dacă vectorul este deja sortat crescător, putem căuta mult mai inteligent. Ne uităm la elementul din mijloc. Dacă este exact cel căutat, am terminat. Dacă este mai mare, valoarea căutată poate fi doar în jumătatea stângă; dacă este mai mic, doar în jumătatea dreaptă. La fiecare pas eliminăm jumătate din candidați.

Pre-condiție esențială:

Căutarea binară funcționează DOAR pe un vector sortat. Pe un vector nesortat dă rezultate greșite.

```cpp
int cautaBinar(int v[], int n, int x) {
    int st = 0, dr = n - 1;
    while (st <= dr) {
        int mij = (st + dr) / 2;
        if (v[mij] == x) {
            return mij;       // gasit
        } else if (v[mij] < x) {
            st = mij + 1;     // caut la dreapta
        } else {
            dr = mij - 1;     // caut la stanga
        }
    }
    return -1;                // negasit
}
```

st și dr delimitează zona în care poate fi elementul. Când st depășește dr, zona e goală: elementul lipsește.

De ce e atât de rapidă? Pentru că la fiecare pas înjumătățim căutarea. Un milion de elemente se reduce la 500.000, apoi 250.000, și tot așa: în doar vreo 20 de pași am terminat. Față de cele un milion de pași ai căutării liniare, diferența este uriașă. (Vom măsura riguros aceste viteze în capitolul despre complexitate.)

Schimbarea a două elemente (swap)

Toate sortările de mai jos au nevoie să interschimbe valori din vector. Nu putem face direct a = b și b = a, pentru că am pierde una dintre valori. Folosim o variabilă temporară.

```cpp
void schimba(int &a, int &b) {
    int aux = a;  // salvam a
    a = b;
    b = aux;
}
```

Parametrii prin referință (&) permit funcției să modifice chiar variabilele primite. Există și std::swap în STL.

Sortarea prin selecție

Ideea: caut cel mai mic element din vector și îl aduc pe prima poziție. Apoi caut cel mai mic din rest și îl aduc pe a doua poziție. Și tot așa, până la capăt. La fiecare pas „selectăm” minimul rămas.

```cpp
void sortSelectie(int v[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int poz = i;  // pozitia minimului
        for (int j = i + 1; j < n; j++) {
            if (v[j] < v[poz]) {
                poz = j;
            }
        }
        schimba(v[i], v[poz]);
    }
}
```

Bucla interioară găsește minimul; apoi îl mutăm pe poziția i printr-o singură schimbare.

Sortarea prin inserție

Este modul în care mulți oameni își aranjează cărțile de joc în mână: iei fiecare carte nouă și o strecori la locul ei printre cele deja ordonate. Pornim de la al doilea element și îl deplasăm spre stânga peste toate elementele mai mari decât el.

```cpp
void sortInsertie(int v[], int n) {
    for (int i = 1; i < n; i++) {
        int val = v[i];
        int j = i - 1;
        while (j >= 0 && v[j] > val) {
            v[j + 1] = v[j];  // mutam la dreapta
            j--;
        }
        v[j + 1] = val;  // inseram pe locul gasit
    }
}
```

Partea din stânga lui i rămâne mereu sortată. „val” se strecoară până la poziția potrivită.

Metoda bulelor (bubble sort)

Parcurgem vectorul și comparăm fiecare pereche de elemente vecine; dacă sunt în ordine greșită, le schimbăm. După o trecere completă, cel mai mare element a „urcat” ca o bulă la capăt. Repetăm trecerile până nu mai e nevoie de nicio schimbare.

```cpp
void sortBule(int v[], int n) {
    bool schimbat = true;
    while (schimbat) {
        schimbat = false;
        for (int i = 0; i < n - 1; i++) {
            if (v[i] > v[i + 1]) {
                schimba(v[i], v[i + 1]);
                schimbat = true;
            }
        }
    }
}
```

Steagul „schimbat” ne lasă să ne oprim devreme dacă vectorul a devenit deja sortat.

Să trasăm bubble sort pas cu pas

Urmărim vectorul [5, 1, 4, 2] în prima trecere completă, comparând perechile vecine de la stânga la dreapta:

Prima trecere:

[5, 1, 4, 2] -> 5 > 1, schimba -> [1, 5, 4, 2] [1, 5, 4, 2] -> 5 > 4, schimba -> [1, 4, 5, 2] [1, 4, 5, 2] -> 5 > 2, schimba -> [1, 4, 2, 5]

După prima trecere, 5 (cel mai mare) a ajuns la capăt. A doua trecere aranjează [1, 4, 2] în [1, 2, 4], iar a treia confirmă că totul e ordonat. Rezultat final: [1, 2, 4, 5].

Un program complet de testare

Sortăm un vector și afișăm conținutul înainte și după, ca să vedem efectul algoritmului.

CPP

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

void schimba(int &a, int &b) {
    int aux = a; a = b; b = aux;
}

void sortSelectie(int v[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int poz = i;
        for (int j = i + 1; j < n; j++)
            if (v[j] < v[poz]) poz = j;
        schimba(v[i], v[poz]);
    }
}

int main() {
    int v[] = {5, 1, 4, 2, 8};
    int n = 5;
    cout << "Inainte:";
    for (int i = 0; i < n; i++)
        cout << " " << v[i];
    cout << endl;
    sortSelectie(v, n);
    cout << "Dupa:";
    for (int i = 0; i < n; i++)
        cout << " " << v[i];
    cout << endl;
    return 0;
}
```

Vectorul este modificat direct în memorie de către funcție, fiindcă tablourile se transmit prin adresă.

REZULTAT

```
Inainte: 5 1 4 2 8
Dupa: 1 2 4 5 8
```

Cât de rapide sunt aceste sortări?

Selecția, inserția și metoda bulelor fac, în esență, două bucle imbricate: pentru fiecare element mai parcurg, în medie, jumătate din vector. Asta înseamnă aproximativ n × n operații — spunem că sunt algoritmi de ordinul O(n²). Pentru 100 de elemente, vreo 10.000 de pași; pentru un milion, devin nepractice. Există sortări mult mai rapide, O(n·log n), studiate în capitolul despre complexitate.

Funcția sort() din STL

În practică nu îți scrii propria sortare: biblioteca standard (STL) oferă funcția sort(), foarte rapidă și deja testată. O folosești dând începutul și sfârșitul zonei de sortat. Capitolul dedicat STL o detaliază.

```cpp
#include <algorithm>  // pentru sort
// ...
int v[] = {5, 1, 4, 2, 8};
sort(v, v + 5);  // sorteaza crescator
```

v + 5 indică „dincolo de ultimul element”. Rezultatul este {1, 2, 4, 5, 8}. Scriem sortări de mână mai ales ca să înțelegem cum funcționează.

Teste de verificare

Compilează și rulează aceste programe; compară ieșirea cu rezultatul așteptat.

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

int cautaLiniar(int v[], int n, int x) {
    for (int i = 0; i < n; i++)
        if (v[i] == x) return i;
    return -1;
}

int main() {
    int v[] = {7, 3, 9, 1, 4};
    cout << cautaLiniar(v, 5, 9) << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
2
```

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

int cautaBinar(int v[], int n, int x) {
    int st = 0, dr = n - 1;
    while (st <= dr) {
        int m = (st + dr) / 2;
        if (v[m] == x) return m;
        else if (v[m] < x) st = m + 1;
        else dr = m - 1;
    }
    return -1;
}

int main() {
    int v[] = {1, 3, 5, 7, 9, 11};
    cout << cautaBinar(v, 6, 7) << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
3
```

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

int main() {
    int v[] = {4, 2, 5, 1, 3};
    int n = 5;
    for (int i = 1; i < n; i++) {
        int val = v[i], j = i - 1;
        while (j >= 0 && v[j] > val) {
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = val;
    }
    for (int i = 0; i < n; i++)
        cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
1 2 3 4 5
```

Exerciții propuse

1. Scrie o funcție care numără de câte ori apare o valoare x într-un vector, folosind căutarea secvențială (parcurge tot vectorul, nu te opri la prima apariție).

2. Modifică sortarea prin selecție astfel încât să ordoneze vectorul descrescător (de la cel mai mare la cel mai mic).

3. Citește un vector sortat crescător și o valoare x, apoi folosește căutarea binară pentru a spune dacă x există. Afișează „DA” sau „NU”.

4. Trasează pe hârtie pașii sortării prin inserție pentru vectorul [3, 1, 4, 1, 5], notând conținutul după fiecare inserare.

5. Sortează un vector cu metoda bulelor și afișează câte treceri complete au fost necesare până când nu s-a mai făcut nicio schimbare.

PARTEA IV

Subprograme și recursivitate

Cum descompunem problemele în subprograme și cum gândim recursiv.

PARTEA IV

4.1 Subprograme (funcții)

Parametri prin valoare și prin referință

Până acum am scris totul în interiorul lui main. Funcționează pentru programe mici, dar când problema crește, codul devine o îngrămădire greu de citit și de corectat. Soluția este să descompunem problema în bucăți mai mici, fiecare rezolvând o singură sarcină clară. Aceste bucăți se numesc subprograme — în C++, funcții.

Avantajele sunt trei: descompunerea (împărțim o problemă grea în pași simpli), reutilizarea (scriem o dată o funcție și o apelăm de câte ori vrem) și claritatea (un nume bun, ca maxim sau prim, spune imediat ce face codul).

Cum se definește o funcție

O funcție are un tip returnat, un nume, o listă de parametri între paranteze și un corp între acolade. Forma generală este:

Structura unei funcții:

tip_returnat nume(parametri) { // instructiuni return valoare; }

Să scriem o funcție care întoarce maximul dintre două numere întregi. Tipul returnat este int, iar valoarea o trimitem înapoi cu instrucțiunea return.

```cpp
int maxim(int a, int b) {
    if (a > b) {
        return a;  // ies si trimit a
    }
    return b;
}
```

return termină imediat funcția și trimite valoarea înapoi celui care a apelat-o.

Apelul unei funcții

A defini o funcție nu o execută. Ea rulează doar când o apelăm, scriindu-i numele urmat de valorile concrete. Rezultatul lui return poate fi folosit într-o expresie sau afișat direct.

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

int maxim(int a, int b) {
    if (a > b) return a;
    return b;
}

int main() {
    int x = maxim(7, 3);  // apel
    cout << x << endl;
    cout << maxim(10, 25) << endl;
    return 0;
}
```

Funcția maxim este apelată de două ori, cu argumente diferite.

```
REZULTAT
7
25
```

Parametri formali și parametri efectivi

În definiția int maxim(int a, int b), numele a și b sunt parametrii formali — sunt doar niște „cutii” în care vor intra valorile. La apelul maxim(7, 3), valorile 7 și 3 sunt parametrii efectivi (sau argumente): ele se copiază în a și b.

Funcții void (proceduri)

Uneori o funcție face ceva (afișează, modifică) dar nu are nimic de întors. Atunci tipul returnat este void, iar return nu este obligatoriu. O astfel de funcție se mai numește procedură. Iată una care afișează un vector:

```cpp
void afiseazaVector(int v[], int n) {
    for (int i = 0; i < n; i++) {
        cout << v[i] << ' ';
    }
    cout << endl;
}
```

Tipul void înseamnă „nu întoarce nicio valoare”. O apelăm simplu: afiseazaVector(v, n);

Transmiterea prin valoare

În mod implicit, parametrii se transmit prin valoare: funcția primește o COPIE a argumentului. Orice modificare din interiorul funcției afectează doar copia, nu și variabila originală din main.

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

void incearca(int x) {
    x = 100;  // modific doar copia
}

int main() {
    int a = 5;
    incearca(a);
    cout << a << endl;  // a ramane 5
    return 0;
}
```

x este o copie a lui a; schimbarea lui x nu atinge a.

```
REZULTAT
5
```

Transmiterea prin referință (&)

Dacă vrem ca funcția să modifice chiar variabila originală, punem & după tipul parametrului. Atunci parametrul devine un alt nume (un alias) pentru variabila trimisă — nu o copie. Exemplul clasic este interschimbarea (swap) a două valori: ea funcționează DOAR cu referință.

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

void interschimba(int &a, int &b) {
    int aux = a;
    a = b;
    b = aux;
}

int main() {
    int x = 3, y = 8;
    interschimba(x, y);
    cout << x << ' ' << y << endl;
    return 0;
}
```

Cu & lângă a și b, funcția lucrează direct pe x și y. Fără &, x și y ar rămâne 3 și 8.

```
REZULTAT
8 3
```

Regula de aur:

Fără & -> funcția primește o COPIE (nu schimbă originalul). Cu & -> funcția lucrează pe ORIGINAL (îl poate schimba).

Prototipuri (declarare înainte de main)

Compilatorul citește fișierul de sus în jos: o funcție trebuie cunoscută înainte de a fi apelată. Putem fie defini funcția înainte de main, fie pune la început doar prototipul ei (antetul terminat cu punct și virgulă), iar definiția mai jos.

CPP

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

int patrat(int n);  // prototip

int main() {
    cout << patrat(6) << endl;
    return 0;
}

int patrat(int n) {  // definitie
    return n * n;
}
```

Prototipul îi spune compilatorului că funcția există; definiția poate veni după main.

REZULTAT

```
36
```

Variabile locale și globale. Domeniul de vizibilitate

O variabilă declarată în interiorul unei funcții este locală: există doar cât timp rulează acea funcție și nu poate fi văzută din afară. O variabilă declarată în afara tuturor funcțiilor este globală și e vizibilă peste tot. Recomandarea de bază: folosește variabile locale și trimite datele prin parametri; variabilele globale, în exces, fac programul greu de urmărit.

O funcție poate apela altă funcție

Funcțiile se pot combina: o funcție poate apela alte funcții. Scriem cmmdc (cel mai mare divizor comun) prin algoritmul lui Euclid și o funcție prim care verifică dacă un număr este prim, apoi le folosim din main.

```cpp
int cmmdc(int a, int b) {
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}

bool prim(int n) {
    if (n < 2) return false;
    for (int d = 2; d * d <= n; d++) {
        if (n % d == 0) return false;
    }
    return true;
}
```

prim întoarce bool (true sau false). cmmdc folosește împărțiri succesive până când b devine 0.

Teste de verificare

Compilează și rulează aceste programe; compară ieșirea cu rezultatul așteptat.

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

int maxim(int a, int b) {
    return (a > b) ? a : b;
}

int main() {
    cout << maxim(4, 9) << endl;
    cout << maxim(15, 2) << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
9
15
```

CPP

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

int cmmdc(int a, int b) {
    while (b != 0) {
        int r = a % b; a = b; b = r;
    }
    return a;
}

int main() {
    cout << cmmdc(12, 18) << endl;
    cout << cmmdc(7, 5) << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
6
1
```

CPP

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

bool prim(int n) {
    if (n < 2) return false;
    for (int d = 2; d * d <= n; d++)
        if (n % d == 0) return false;
    return true;
}

int main() {
    cout << prim(7) << ' ';
    cout << prim(10) << endl;
    return 0;
}
```

În C++, true se afișează ca 1 și false ca 0.

REZULTAT AȘTEPTAT

```
1 0
```

Exerciții propuse

1. Scrie o funcție int suma(int a, int b) care întoarce suma a două numere și testeaz-o din main.

2. Scrie o funcție void afiseazaVector(int v[], int n) care afișează elementele unui vector, separate prin spațiu.

3. Scrie o funcție bool parDivizibil(int n) care întoarce true dacă n este par, altfel false.

4. Modifică funcția de swap astfel încât să fie de tip double și verifică, schimbând două variabile reale, că valorile chiar se interschimbă (folosește &).

5. Folosind funcția prim, scrie un program care afișează toate numerele prime de la 2 la n, unde n se citește de la tastatură.

PARTEA IV

4.2 Recursivitate

Funcții care se autoapelează

Recursivitatea este una dintre cele mai elegante idei din programare: o funcție își rezolvă problema chemându-se pe ea însăși, dar pentru un caz mai mic. Pare un truc, însă este doar un mod de a gândi: „dacă aș ști să rezolv problema pentru ceva puțin mai simplu, aș putea rezolva și cazul de față?”. Multe probleme se descriu natural așa.

O funcție recursivă este, pur și simplu, o funcție care se apelează pe sine. În capitolul despre funcții am văzut cum o funcție poate chema altă funcție; acum funcția se cheamă chiar pe ea, dar cu un argument mai mic decât cel primit.

Cele două ingrediente obligatorii

Orice funcție recursivă corectă are exact două părți. Lipsa oricăreia dintre ele duce la un program greșit.

Rețeta recursivității:

1. CAZUL DE BAZĂ — situația cea mai simplă, rezolvată direct, FĂRĂ autoapel (oprirea). 2. CAZUL GENERAL — reduce problema la un subcaz mai mic și se autoapelează pentru acel subcaz.

Cazul de bază este frâna: el oprește lanțul de apeluri. Cazul general este motorul: el apropie problema, pas cu pas, de cazul de bază. Dacă fiecare autoapel folosește un argument strict mai mic, mai devreme sau mai târziu ajungem la caz de bază și ne oprim.

Primul exemplu: factorialul

Factorialul lui n, notat n!, este produsul 1·2·3·…·n. De exemplu, 5! = 120. Definiția matematică este deja recursivă: 0! = 1 (cazul de bază), iar n! = n · (n−1)! pentru n ≥ 1 (cazul general).

CPP

```cpp
long long factorial(int n) {
    if (n == 0) {
        return 1;          // cazul de baza
    }
    return n * factorial(n - 1);  // caz general
}
```

Când n este 0 ne oprim și întoarcem 1. Altfel, înmulțim n cu factorialul unui număr mai mic cu o unitate.

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

long long factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int main() {
    cout << factorial(5) << endl;
    return 0;
}
```

```
120
```

Stiva apelurilor — cum se așază și se desfac apelurile

Ca să înțelegem cu adevărat recursivitatea, trebuie să vedem stiva apelurilor. Când o funcție cheamă altă funcție, calculatorul „pune deoparte” apelul curent (cu valorile lui) și începe noul apel. Apelurile se stivuiesc unul peste altul; fiecare așteaptă rezultatul celui de deasupra. Abia când ajungem la cazul de bază, stiva începe să se desfacă, întorcând rezultatele înapoi, de sus în jos.

factorial(3) — așezarea apelurilor (coborâm):

factorial(3) = 3 * factorial(2) -- asteapta factorial(2) = 2 * factorial(1) -- asteapta factorial(1) = 1 * factorial(0) -- asteapta factorial(0) = 1 -- CAZ DE BAZA

factorial(3) — desfacerea apelurilor (urcăm):

factorial(0) intoarce 1 factorial(1) intoarce 1 * 1 = 1 factorial(2) intoarce 2 * 1 = 2 factorial(3) intoarce 3 * 2 = 6

Fiecare apel ocupă un loc pe stivă cât timp așteaptă rezultatul de mai sus. De aceea recursivitatea consumă memorie pe stivă proporțional cu adâncimea apelurilor.

Ce se întâmplă FĂRĂ caz de bază

Dacă uităm cazul de bază (sau dacă argumentul nu se micșorează), funcția se cheamă la nesfârșit: recursivitate infinită. Stiva se umple până când programul rămâne fără memorie și se oprește brusc cu eroarea „stack overflow”. Este echivalentul recursiv al unui while care nu se termină niciodată.

```cpp
int gresit(int n) {
    // LIPSESTE cazul de baza!
    return gresit(n - 1);  // nu se opreste
}
```

Acest cod compilează, dar la rulare provoacă stack overflow. Întotdeauna verifică mai întâi cazul de bază.

Suma primelor n numere naturale

Suma 1 + 2 + … + n se gândește recursiv astfel: suma până la n este n plus suma până la n−1. Cazul de bază este suma până la 0, care este 0.

```cpp
int suma(int n) {
    if (n == 0) return 0;   // caz de baza
    return n + suma(n - 1); // caz general
}
```

```
REZULTAT
suma(5) = 15
```

Ridicarea la putere

Vrem să calculăm a la puterea b (b ≥ 0). Recursiv: a^0 = 1 (caz de bază), iar a^b = a · a^(b−1) (caz general).

```cpp
long long putere(int a, int b) {
    if (b == 0) return 1;       // a^0 = 1
    return a * putere(a, b - 1);
}
```

De fiecare dată coborâm exponentul cu 1, până ajunge la 0.

```
REZULTAT
putere(2, 10) = 1024
```

Suma cifrelor unui număr

Putem aduna cifrele unui număr recursiv. Ultima cifră este n % 10, iar restul numărului este n / 10. Suma cifrelor lui n este (n % 10) plus suma cifrelor lui n / 10. Cazul de bază este n == 0, când nu mai avem cifre de adunat.

```cpp
int sumaCifrelor(int n) {
    if (n == 0) return 0;     // caz de baza
    return n % 10 + sumaCifrelor(n / 10);
}
```

```
REZULTAT
sumaCifrelor(2025) = 9
```

Afișarea unui număr cifră cu cifră

Dacă afișăm întâi ultima cifră, numărul iese pe dos. Trucul recursiv: mai întâi afișăm partea din față (n / 10) și abia apoi ultima cifră (n % 10). Astfel apelurile mai adânci tipăresc cifrele din stânga primele, în ordinea corectă.

```cpp
void afiseazaCifre(int n) {
    if (n < 10) {
        cout << n << ' ';  // o singura cifra
        return;
    }
    afiseazaCifre(n / 10);  // intai partea din fata
    cout << n % 10 << ' '; // apoi ultima cifra
}
```

Pentru n = 503, ordinea apelurilor face ca afișarea să fie 5 0 3, nu invers.

```
REZULTAT
5 0 3
```

Șirul lui Fibonacci (și o capcană)

În șirul lui Fibonacci fiecare termen este suma celor doi dinaintea lui: 0, 1, 1, 2, 3, 5, 8, 13, … Definiția are două cazuri de bază (F(0) = 0 și F(1) = 1) și un caz general (F(n) = F(n−1) + F(n−2)).

```cpp
long long fib(int n) {
    if (n < 2) return n;    // F(0)=0, F(1)=1
    return fib(n - 1) + fib(n - 2);
}
```

```
REZULTAT
fib(10) = 55
```

Atenție: această variantă naivă este foarte ineficientă! Pentru a calcula fib(n) reface aceleași subprobleme de nenumărate ori — fib(5) calculează fib(3) de două ori, fib(2) de trei ori și așa mai departe. Numărul de apeluri crește aproape la fel de repede ca șirul însuși, așa că fib(50) ar dura un timp uriaș. În capitolele următoare vom vedea soluții mult mai rapide (iterativă sau cu memorare).

Recursivitate vs iterație

Aproape orice se poate scrie și recursiv, și iterativ (cu bucle). Cum alegem? Recursivitatea este adesea mai scurtă și mai apropiată de definiția matematică a problemei — codul „se citește" ca enunțul. Iterația, în schimb, nu încarcă stiva și de obicei este mai rapidă, fiindcă nu plătește costul apelurilor de funcție.

Reține echilibrul:

RECURSIV: cod clar, aproape de definitie; dar consuma stiva si poate fi mai lent. ITERATIV: rapid, fara risc de stack overflow; dar uneori mai greu de citit.

Pentru probleme cu structură natural recursivă (parcurgerea unui arbore, backtracking, divide et impera) recursivitatea strălucește. Pentru un simplu factorial sau o sumă, o buclă este perfect potrivită. Important este să înțelegi ambele și să alegi unealta corectă.

Teste de verificare

Compilează și rulează fiecare program; compară ieșirea cu rezultatul așteptat de mai jos.

CPP

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

long long factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int main() {
    cout << factorial(6) << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
720
```

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

int sumaCifrelor(int n) {
    if (n == 0) return 0;
    return n % 10 + sumaCifrelor(n / 10);
}

int main() {
    cout << sumaCifrelor(1234) << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
10
```

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

long long fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}

int main() {
    cout << fib(7) << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
13
```

Exerciții propuse

1. Scrie o funcție recursivă produs(n) care întoarce produsul 1·2·…·n (adică factorialul), fără să folosești bucle. Testeaz-o pentru n = 4 (rezultat 24).

2. Scrie o funcție recursivă numaraCifre(n) care întoarce câte cifre are un număr natural n. Indiciu: o cifră dacă n < 10, altfel 1 + numaraCifre(n / 10).

3. Scrie o funcție recursivă putereDoi(n) care întoarce 2 la puterea n. Verifică: putereDoi(8) trebuie să dea 256.

4. Scrie o funcție recursivă afiseazaInvers(n) care afișează cifrele numărului n în ordine inversă (ultima cifră prima). Pentru 503 trebuie să apară 3 0 5.

5. Scrie o funcție recursivă cmmdc(a, b) care calculează cel mai mare divizor comun după

algoritmul lui Euclid: dacă b == 0 rezultatul este a, altfel cmmdc(b, a % b). Testeaz-o pentru (48, 36) — rezultat 12.

PARTEA IV

4.3 Metoda backtracking

Generarea sistematică a soluțiilor

Sunt probleme la care nu ne ajunge o singură soluție: vrem toate aranjamentele posibile, toate submulțimile, toate modurile de a așeza piese pe o tablă. Backtracking este metoda prin care construim aceste soluții pas cu pas și le explorăm pe toate, fără să uităm vreuna și fără să repetăm.

Ideea de bază este simplă: completăm soluția poziție cu poziție. Pe poziția curentă încercăm, pe rând, fiecare valoare care pare validă. Dacă o valoare e bună, mergem mai departe recursiv pe poziția următoare. Când ne întoarcem din recursivitate, „revenim” (engl. backtrack): anulăm alegerea și încercăm valoarea următoare.

Numele vine tocmai de aici: când o cale nu mai duce nicăieri, ne întoarcem (back) pe urmele noastre (track) și alegem altfel. Metoda presupune cunoscută recursivitatea, fiindcă fiecare poziție este tratată printr-un nou apel al funcției.

Schema generală

Aproape orice problemă de backtracking respectă același tipar. Avem o funcție back(k) care încearcă să completeze poziția k a soluției:

Tiparul back(k):

dacă soluția e completă -> o folosim (tipărim) altfel, pentru fiecare candidat de pe poziția k: dacă e valid: îl marcăm, apelăm back(k+1), apoi îl demarcăm (revenim).

Avem nevoie de două tipuri de condiții. Condiția de soluție ne spune când o soluție este completă (de obicei, când am umplut toate pozițiile). Condiția de validare (sau de continuare) ne spune dacă un candidat poate fi pus pe poziția curentă fără a încălca regulile problemei.

Exemplul 1: permutările lui {1, 2, ..., n}

Vrem toate ordonările numerelor de la 1 la n. Soluția are n poziții; pe fiecare punem un număr care nu a mai fost folosit. Marcăm numerele folosite într-un vector folosit[].

CPP

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

int n;
int sol[20];        // soluția curentă
bool folosit[20];   // ce numere sunt deja puse

void back(int k) {
    if (k == n) {           // soluție completă
        for (int i = 0; i < n; i++)
            cout << sol[i] << ' ';
        cout << '\n';
        return;
    }
    for (int v = 1; v <= n; v++)
        if (!folosit[v]) {  // candidat valid
            folosit[v] = true;
            sol[k] = v;
            back(k + 1);
            folosit[v] = false;  // revenim
        }
}

int main() {
    n = 3;
    back(0);
    return 0;
}
```

Pe poziția k punem orice număr nefolosit, îl marcăm, coborâm recursiv, apoi îl eliberăm pentru a încerca altul. Când k == n, am umplut toate pozițiile.

REZULTAT (n = 3)

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Observă ordinea: numerele apar în ordine lexicografică, fiindcă pe fiecare poziție încercăm candidații de la cel mai mic la cel mai mare. Pentru n = 3 obținem 3! = 6 permutări.

Exemplul 2: produsul cartezian (numere de n cifre)

Acum poziția k nu mai depinde de alegerile anterioare: pe fiecare dintre cele n poziții putem pune orice cifră din mulțimea {1, 2, ..., m}. Asta generează toate „cuvintele” de lungime n peste un alfabet de m simboluri.

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

int n = 2, m = 3;   // lungime n, simboluri 1..m
int sol[20];

void back(int k) {
    if (k == n) {
        for (int i = 0; i < n; i++)
            cout << sol[i];
        cout << '\n';
        return;
    }
    for (int v = 1; v <= m; v++) {
        sol[k] = v;     // orice simbol e valid
        back(k + 1);
    }
}

int main() {
    back(0);
    return 0;
}
```

Nu mai avem vector folosit: pe fiecare poziție orice simbol este permis, deci validarea este banală.

```
REZULTAT (n = 2, m = 3)
11
12
13
21
22
23
31
32
33
```

Aceeași schemă, doar condiția de validare se schimbă. Dacă, în plus, am cere ca v să fie strict mai mare decât valoarea de pe poziția anterioară, am genera submulțimile (combinările) în ordine crescătoare. Backtracking este o singură idee, multe probleme.

Exemplul 3: problema celor n dame

Pe o tablă de șah n×n trebuie să așezăm n dame astfel încât nicio damă să nu fie atacată de alta. Punem câte o damă pe fiecare linie; poziția k este linia k, iar valoarea aleasă este coloana. Două dame se atacă dacă sunt pe aceeași coloană sau pe aceeași diagonală (diferența liniilor egală cu diferența coloanelor, în modul).

```cpp
#include <iostream>
#include <cstdlib>   // abs
using namespace std;

int n = 4;
int col[20];    // col[i] = coloana damei de pe linia i
int nrSol = 0;

bool valid(int k, int c) {
    for (int i = 0; i < k; i++) {
        if (col[i] == c) return false;        // coloană
        if (abs(col[i] - c) == abs(i - k))
            return false;                     // diagonală
    }
    return true;
}

void back(int k) {
    if (k == n) { nrSol++; return; }
    for (int c = 0; c < n; c++)
        if (valid(k, c)) {
            col[k] = c;
            back(k + 1);
        }
}

int main() {
    back(0);
    cout << nrSol << '\n';
    return 0;
}
```

Funcția valid verifică doar liniile deja completate (0..k-1): coloana ocupată și ambele diagonale. Aici doar numărăm soluțiile.

```
REZULTAT (n = 4)
2
```

Pentru n = 4 există exact 2 așezări corecte. Pentru n = 8 (tabla clasică) sunt 92. Verificarea „pe scurt” doar pentru liniile de deasupra este suficientă, fiindcă punem o singură damă pe fiecare linie.

Teste de verificare

Test 1. Câte permutări tipărește programul din Exemplul 1 pentru n = 4? Modifică n = 4 și numără liniile afișate.

```
REZULTAT AȘTEPTAT
24 linii (adică 4! permutări)
```

Test 2. În programul produsului cartezian pune n = 3 și m = 2 și rulează. Câte „cuvinte”

obții?

```
REZULTAT AȘTEPTAT
8 cuvinte (2 la puterea 3), de la 111 la 222
```

Test 3. În problema damelor pune n = 5 și rulează.

```
REZULTAT AȘTEPTAT
10
```

Exerciții propuse

1. Modifică Exemplul 1 ca să afișeze și numărul total de permutări la final, pe o linie separată.

2. Generează toate submulțimile mulțimii {1, 2, ..., n}: pe fiecare poziție alege fie să incluzi numărul, fie să nu îl incluzi (două candidaturi: 0 și 1).

3. Generează aranjamentele de n luate câte p: ca la permutări, dar oprește-te când ai completat p poziții (k == p).

4. Pentru produsul cartezian, adaugă o condiție de validare care interzice două cifre egale alăturate (sol[k] diferit de sol[k-1]).

5. Modifică problema damelor să tipărească prima soluție găsită (coloanele), nu doar numărul lor, apoi oprește căutarea.

PARTEA IV

4.4 Divide et impera

Împarte, rezolvă, combină

Numele vine din latină: „împarte și stăpânește”. Ideea e veche și foarte simplă: o problemă mare și grea devine ușoară dacă o spargi în bucăți mici, le rezolvi pe fiecare separat, apoi lipești rezultatele la loc. În informatică, divide et impera este o strategie de proiectare a algoritmilor care, aplicată cu cap, ne aduce câștiguri uriașe de viteză.

Cei trei pași

Orice algoritm de tip divide et impera urmează aceeași schemă în trei pași:

Schema generală:

1. ÎMPARTE problema în subprobleme mai mici, de același tip cu cea inițială. 2. REZOLVĂ fiecare subproblemă, recursiv, până ajungi la cazuri atât de simple încât răspunsul e imediat (cazul de bază). 3. COMBINĂ răspunsurile subproblemelor într-un răspuns pentru problema mare.

Observă cât de natural se leagă acest tipar de recursivitate: pasul „rezolvă subproblemele” înseamnă, de fapt, că funcția se apelează pe ea însăși pe date mai mici. Cazul de bază al recursiei este chiar subproblema atât de mică încât nu mai are rost s-o împărțim.

Căutarea binară, văzută ca divide et impera

Cunoaștem deja căutarea binară într-un vector sortat. Ea este, de fapt, divide et impera în forma cea mai pură: ÎMPĂRȚIM vectorul în două jumătăți față de mijloc, ne uităm în care jumătate poate fi valoarea și REZOLVĂM căutarea doar acolo. Aici „combinarea” e banală — răspunsul subproblemei este direct răspunsul nostru.

```cpp
int cautaBinar(int v[], int st, int dr, int x) {
    if (st > dr) {
        return -1;        // caz de baza: gol
    }
    int mij = (st + dr) / 2;
    if (v[mij] == x) {
        return mij;       // l-am gasit
    } else if (x < v[mij]) {
        // cauta in jumatatea stanga
        return cautaBinar(v, st, mij - 1, x);
    } else {
        // cauta in jumatatea dreapta
        return cautaBinar(v, mij + 1, dr, x);
    }
}
```

Aceeași căutare binară pe care o știam, dar scrisă recursiv. La fiecare apel renunțăm la jumătate din vector, deci ajungem la răspuns în circa $\log_2(n)$ pași.

Maximul dintr-un vector prin împărțire la jumătate

Cum aflăm cel mai mare element dintr-un vector folosind divide et impera? ÎMPĂRȚIM vectorul în două jumătăți, REZOLVĂM recursiv găsind maximul fiecărei jumătăți, apoi COMBINĂM: maximul întregului vector este cel mai mare dintre cele două maxime parțiale.

```cpp
int maximDinValori(int a, int b) {
    return (a > b) ? a : b;
}

int maxim(int v[], int st, int dr) {
    if (st == dr) {
        return v[st];     // caz de baza: 1 element
    }
    int mij = (st + dr) / 2;
    int stMax = maxim(v, st, mij);
    int drMax = maxim(v, mij + 1, dr);
    return maximDinValori(stMax, drMax);  // combina
}
```

Cazul de bază e un singur element: el este propriul maxim. Pentru suma elementelor, schema e identică — doar înlocuiești alegerea maximului cu adunarea celor două sume parțiale.

Ridicarea rapidă la putere

Vrem să calculăm a la puterea n. Metoda naivă înmulțește pe a cu el însuși de n ori — n înmulțiri. Divide et impera face mult mai bine, pornind de la o observație simplă despre exponenți:

Ideea cheie:

a^n = (a^(n/2))^2, daca n este par a^n = a * (a^(n/2))^2, daca n este impar a^0 = 1 (cazul de baza)

Calculăm o singură dată a^(n/2), apoi îl ridicăm la pătrat. Astfel ÎMPĂRȚIM exponentul la jumătate la fiecare pas, deci facem doar circa $\log_2(n)$ înmulțiri în loc de n. Pentru n = 1.000.000, asta înseamnă vreo 20 de înmulțiri în loc de un milion!

```cpp
long long putere(long long a, int n) {
    if (n == 0) {
        return 1;          // caz de baza
    }
    long long jum = putere(a, n / 2);
    long long patrat = jum * jum;  // combina
    if (n % 2 == 1) {
        return a * patrat;  // n impar
    } else {
        return patrat;      // n par
    }
```

Folosim long long fiindcă puterile cresc foarte repede. Spunem că algoritmul lucrează în O(log n) — exponențiere logaritmică. Atenție: lipsește acolada de închidere; o adăugăm la final.

```cpp
}
```

Acolada care închide funcția putere. (Am despărțit codul doar ca să încapă comentariile.)

Sortarea prin interclasare (merge sort)

Cea mai elegantă aplicație a strategiei este o sortare rapidă, numită sortare prin interclasare. ÎMPĂRȚIM vectorul în două jumătăți, le SORTĂM recursiv pe fiecare, apoi le COMBINĂM („interclasăm”) într-un singur vector sortat. Pasul de combinare este partea ingenioasă, așa că hai să-l vedem întâi separat.

Interclasarea a doi subvectori sortați

Avem două bucăți deja sortate, una lângă alta în același vector: v[st..mij] și v[mij+1..dr]. Le îmbinăm comparând primul element rămas din fiecare bucată și luându-l mereu pe cel mai mic. Rezultatul îl construim într-un vector auxiliar, apoi îl copiem înapoi.

```cpp
void interclaseaza(int v[], int st,
                   int mij, int dr) {
    int aux[100];
    int i = st, j = mij + 1, k = 0;
    while (i <= mij && j <= dr) {
        if (v[i] <= v[j]) {
            aux[k++] = v[i++];
        } else {
            aux[k++] = v[j++];
        }
    }
    while (i <= mij) aux[k++] = v[i++];
    while (j <= dr)  aux[k++] = v[j++];
    for (int t = 0; t < k; t++) {
        v[st + t] = aux[t];  // copiaza inapoi
    }
}
```

Primele două bucle iau pe rând minimul din capul fiecărei jumătăți. După ce una se golește, ultimele două bucle copiază restul celeilalte, deja sortat.

Acum structura recursivă este scurtă: împărțim, sortăm fiecare jumătate, interclasăm.

```cpp
void mergeSort(int v[], int st, int dr) {
    if (st >= dr) {
        return;           // 0 sau 1 element
    }
    int mij = (st + dr) / 2;
    mergeSort(v, st, mij);      // sorteaza stanga
    mergeSort(v, mij + 1, dr);  // sorteaza dreapta
    interclaseaza(v, st, mij, dr);  // combina
}
```

Cazul de bază: un vector cu cel mult un element e deja sortat. Toată munca de ordonare se face în interclasare.

De ce e merge sort important? Pentru că face doar circa $n \cdot \log_2(n)$ operații, mult mai puține decât cele n^2 ale sortărilor simple. Pentru un milion de elemente, diferența este între câteva zeci de milioane de pași și un trilion — imens. Vom măsura riguros aceste viteze în capitolul despre complexitate.

Când este utilă strategia?

Divide et impera dă roade atunci când problema se poate sparge în subprobleme independente, de același tip, iar combinarea rezultatelor este ieftină. Câștigul tipic apare fiindcă, înjumătățind datele la fiecare pas, numărul de niveluri de recursie rămâne mic (de ordinul lui log n). Dacă subproblemele se suprapun sau combinarea e scumpă, alte tehnici pot fi mai potrivite.

Teste de verificare

Compilează și rulează aceste programe; compară ieșirea cu rezultatul așteptat.

CPP

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

long long putere(long long a, int n) {
    if (n == 0) return 1;
    long long j = putere(a, n / 2);
    long long p = j * j;
    return (n % 2) ? a * p : p;
}

int main() {
    cout << putere(2, 10) << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
1024
```

CPP

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

int maxim(int v[], int st, int dr) {
    if (st == dr) return v[st];
    int m = (st + dr) / 2;
    int a = maxim(v, st, m);
    int b = maxim(v, m + 1, dr);
    return (a > b) ? a : b;
}

int main() {
    int v[] = {3, 9, 1, 7, 5};
    cout << maxim(v, 0, 4) << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
9
```

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

int cautaBinar(int v[], int st, int dr, int x) {
    if (st > dr) return -1;
    int m = (st + dr) / 2;
    if (v[m] == x) return m;
    if (x < v[m]) return cautaBinar(v, st, m-1, x);
    return cautaBinar(v, m + 1, dr, x);
}

int main() {
    int v[] = {2, 4, 6, 8, 10, 12};
    cout << cautaBinar(v, 0, 5, 10) << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
4
```

Exerciții propuse

1. Scrie, după modelul funcției maxim, o funcție recursivă suma(v, st, dr) care calculează suma elementelor unui vector împărțindu-l mereu în două jumătăți.

2. Modifică funcția maxim astfel încât să întoarcă valoarea minimă din vector în loc de cea maximă.

3. Folosește ridicarea rapidă la putere pentru a calcula 3^15 și verifică pe hârtie numărul de înmulțiri făcute.

4. Completează programul cu funcțiile interclaseaza și mergeSort, sortează vectorul [5, 2, 8, 1, 9, 3] și afișează rezultatul.

5. Trasează pe hârtie apelurile lui mergeSort pentru vectorul [4, 1, 3, 2]: desenează arborele de împărțiri și notează ce întoarce fiecare interclasare.

PARTEA V

Concepte avansate

Pointeri, structuri proprii, structuri de date, biblioteca standard și eficiența.

PARTEA V

5.1 Pointeri și memorie dinamică

Adrese, new/delete, tablouri dinamice

Până acum am lucrat cu variabile fără să ne întrebăm „unde anume" trăiesc ele. Acest capitol ridică tocmai capacul: vom vedea că fiecare variabilă ocupă un loc bine precizat în memorie și că putem reține și folosi acel loc. Unealta care face asta se numește pointer.

Memoria și adresele

Memoria calculatorului este ca o stradă lungă cu case numerotate. Fiecare „casă" este o celulă care poate păstra o valoare, iar numărul ei se numește adresă. Când scrii int x = 7;, calculatorul rezervă o casă pentru x și pune înăuntru valoarea 7.

Adresa unei variabile se obține cu operatorul & (citit „adresa lui"). Adresele se afișează de obicei în hexazecimal și diferă de la o rulare la alta — nu conta pe o valoare anume, ci pe ideea că există.

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

int main() {
    int x = 7;
    cout << "valoarea: " << x << '\n';
    cout << "adresa:  " << &x << '\n';
    return 0;
}
```

& înaintea unei variabile dă adresa ei în memorie.

```
valoarea: 7
adresa:  0x7ffe5c3a9b4c
```

Adresa afișată la tine va arăta altfel; important este că x are o adresă proprie.

Ce este un pointer

Un pointer este o variabilă specială: în loc să rețină un număr obișnuit, ea reține o adresă. Spunem că pointerul „arată spre" celula de la acea adresă. Îl declarăm punând un asterisc * după tipul valorii spre care va arăta.

```cpp
int x = 7;
int *p;       // p poate reține adresa unui int
p = &x;       // p arată acum spre x
```

*int *p; — p este pointer spre int. p = &x; îl leagă de x.*

Citește declarația int *p; ca „p este de tip pointer spre int". Un pointer spre int nu poate reține adresa unui double — tipul contează.

Dereferențierea: valoarea de la adresă

Dacă p arată spre x, atunci *p (citit „valoarea de la adresa din p") ne dă chiar valoarea lui x. Operatorul * folosit astfel se numește dereferențiere. Atenție: * are două roluri — la declarare arată că variabila e pointer, iar la folosire dereferențiază.

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

int main() {
    int x = 7;
    int *p = &x;     // p arată spre x
    cout << *p << '\n';   // citește prin pointer
    *p = 20;         // modifică x prin pointer
    cout << x << '\n';    // x s-a schimbat!
    return 0;
}
```

**p citește și *p = ... scrie chiar în x, fiindcă p arată spre x.*

```
REZULTAT
7
20
```

Iată legătura esențială: cât timp p = &x, expresia *p și variabila x sunt două nume pentru aceeași celulă de memorie.

Pointerul nul

Un pointer care nu arată (încă) spre nimic ar trebui setat pe nullptr. Este valoarea „goală" a unui pointer. Dereferențierea unui pointer nul este o eroare gravă, așa că verificăm înainte.

```cpp
int *p = nullptr;     // nu arată spre nimic
if (p != nullptr) {
    cout << *p;       // sigur doar dacă p != nullptr
}
```

nullptr marchează un pointer „fără țintă"; nu îl dereferenția.

Alocare dinamică: new și delete

Până acum dimensiunile erau fixate la scriere. Uneori abia la rulare aflăm de câtă memorie avem nevoie. Operatorul new cere sistemului o celulă nouă și ne dă adresa ei; când terminăm, o eliberăm cu delete.

CPP

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

int main() {
    int *p = new int;   // o celulă nouă
    *p = 42;
    cout << *p << '\n';
    delete p;           // eliberăm memoria
    p = nullptr;        // bună practică
    return 0;
}
```

new int alocă; delete p eliberează. Pune p = nullptr după delete.

REZULTAT

```
42
```

Tablouri alocate dinamic

Marele avantaj: putem aloca un tablou a cărui dimensiune o citim de la tastatură. Folosim new tip[n] pentru a aloca și, obligatoriu, delete[] pentru a elibera (cu paranteze pătrate!).

CPP

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

int main() {
    int n;
    cin >> n;
    int *v = new int[n];   // tablou de n int
    for (int i = 0; i < n; i++)
        v[i] = (i + 1) * (i + 1);
    for (int i = 0; i < n; i++)
        cout << v[i] << ' ';
    cout << '\n';
    delete[] v;            // eliberează tot tabloul
    return 0;
}
```

v[i] funcționează la fel ca la vectori. delete[] eliberează totul.

```
REZULTAT
(intrare: 4)
1 4 9 16
```

Pointeri și funcții

Transmițând adresa unei variabile unei funcții, funcția poate modifica originalul. Așa schimbăm două valori între ele fără a le copia înapoi.

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

void schimba(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int main() {
    int x = 3, y = 8;
    schimba(&x, &y);   // trimitem adresele
    cout << x << ' ' << y << '\n';
    return 0;
}
```

Funcția primește adrese și modifică direct variabilele apelantului.

```
REZULTAT
8 3
```

Capcane: memory leaks și pointeri atârnați

Două greșeli frecvente. Pierderea de memorie (memory leak) apare când aloci cu new dar uiți delete: memoria rămâne ocupată inutil. Pointerul „atârnat" apare când dereferențiezi un pointer după ce ai eliberat memoria — el arată spre o celulă care nu-ți mai aparține.

Regula de aur:

Fiecare new are exact un delete. Fiecare new[] are exact un delete[]. După delete, pune pointerul pe nullptr.

Teste de verificare

Compilează și rulează aceste programe; compară cu rezultatul așteptat.

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

int main() {
    int a = 5;
    int *p = &a;
    *p = *p + 10;
    cout << a << '\n';
    return 0;
}
```

*Modificarea prin *p schimbă chiar pe a.*

```
15
```

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

int main() {
    int n = 5;
    int *v = new int[n];
    int s = 0;
    for (int i = 0; i < n; i++) {
        v[i] = i;
        s += v[i];
    }
    cout << s << '\n';
    delete[] v;
    return 0;
}
```

Sumă 0+1+2+3+4 dintr-un tablou dinamic.

```
10
```

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

void dubleaza(int *x) {
    *x = *x * 2;
}

int main() {
    int k = 9;
    dubleaza(&k);
    cout << k << '\n';
    return 0;
}
```

Funcția dublează valoarea prin adresa primită.

```
REZULTAT AȘTEPTAT
18
```

Exerciții propuse

1. Declară int x = 100; și un pointer p spre x. Afișează valoarea lui x atât direct, cât și prin *p, apoi scade 1 din x folosind *p și afișează din nou.

2. Citește n de la tastatură, alocă dinamic un tablou de n întregi, citește elementele și afișează valoarea maximă. Eliberează memoria cu delete[].

3. Scrie o funcție void increment(int *p) care adună 1 la valoarea indicată de p. Apeleaz-o de trei ori asupra aceleiași variabile și afișează rezultatul.

4. Scrie o funcție void minmax(int *v, int n, int *pmin, int *pmax) care pune în *pmin minimul și în *pmax maximul unui tablou alocat dinamic.

5. Pornind de la un pointer p = nullptr, scrie un program care îi alocă o valoare cu new, o folosește, apoi o eliberează corect și pune p pe nullptr. Explică în comentariu de ce setarea pe nullptr previne pointerii atârnați.

PARTEA V

5.2 Structuri (struct)

Tipuri de date proprii

Până acum, fiecare informație stătea într-o variabilă separată. Dar gândește-te la un elev: el are un nume, o vârstă și o medie. Sunt trei date care merg împreună, descriu același elev. Dacă ai 30 de elevi, ai vrea trei vectori paraleli? E incomod și ușor de greșit. Soluția se numește structură.

Idee de reținut:

Un struct grupeaza mai multe date legate intre ele sub un singur nume, ca un dosar cu mai multe rubrici. Astfel iti creezi propriul tip de date.

Definirea unui struct

O structură se definește o singură dată, de obicei înaintea funcției main. Spui ce câmpuri (rubrici) are. Nu uita punctul și virgula de la final, după acolada de închidere!

```cpp
struct Punct {
    int x;   // coordonata pe orizontala
    int y;   // coordonata pe verticala
};           // ATENTIE: ; obligatoriu aici
```

Punct devine acum un tip nou, la fel ca int sau double. Câmpurile x și y trăiesc împreună în el.

Declararea unei variabile și accesul cu .

După ce ai tipul, declari variabile de tipul lui. La câmpuri ajungi cu operatorul punct: nume_variabilă urmat de punct și numele câmpului.

```cpp
Punct p;        // o variabila de tip Punct
p.x = 3;        // scriem in campul x
p.y = 4;        // scriem in campul y
cout << p.x << ", " << p.y << endl;
```

p.x și p.y se comportă exact ca niște variabile int obișnuite, doar că aparțin lui p.

```
REZULTAT
3, 4
```

Inițializarea pe scurt

Poți da valori câmpurilor chiar la declarare, în ordinea în care apar în struct, folosind acolade.

```cpp
Punct a = {3, 4};   // a.x = 3, a.y = 4
Punct b = {0, 0};   // originea
```

Valorile se potrivesc pe rând cu câmpurile: prima merge la x, a doua la y.

Funcții cu structuri ca parametru

Putem trimite o structură unei funcții, exact ca pe orice altă valoare. Iată o funcție care calculează distanța dintre două puncte.

```cpp
double distanta(Punct a, Punct b) {
    int dx = a.x - b.x;
    int dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
```

Avem nevoie de #include <cmath> pentru funcția sqrt (rădăcina pătrată).

Apelăm funcția dând două puncte; pentru (0,0) și (3,4) distanța iese 5 (triunghiul 3-4-5).

```cpp
Punct a = {0, 0};
Punct b = {3, 4};
cout << distanta(a, b) << endl;
```

```
5
```

Prin valoare vs. prin referință

Trimisă prin valoare, structura se copiază: funcția lucrează pe o copie și nu poate schimba originalul. Dacă pui & după tip, transmiți prin referință și funcția modifică chiar structura primită.

```cpp
void muta(Punct &p, int dx, int dy) {
    p.x += dx;   // schimba originalul
    p.y += dy;
}
```

& înseamnă „aceeași variabilă, nu o copie”. Util când structura e mare: eviți copierea inutilă.

Vector de structuri

Cel mai puternic lucru: poți avea un tablou de structuri. De exemplu, o clasă întreagă de elevi. Definim un struct Elev cu nume și medie, apoi un vector de Elev.

```cpp
struct Elev {
    string nume;
    double media;
};

Elev clasa[100];   // pana la 100 de elevi
int n;             // cati elevi avem
```

clasa[i] e un Elev întreg; clasa[i].nume e numele lui, clasa[i].media e media lui.

Citirea unui vector de structuri

Parcurgem cu un for de la 0 la n-1 și citim, pentru fiecare elev, numele și media. Reține: clasa[i].nume se citește ca orice string.

```cpp
cin >> n;
for (int i = 0; i < n; i++) {
    cin >> clasa[i].nume;
    cin >> clasa[i].media;
}
```

Elevul cu media maximă

Presupunem că primul elev are media maximă, apoi parcurgem restul și actualizăm dacă găsim o medie mai mare. Reținem poziția lui.

```cpp
int best = 0;
for (int i = 1; i < n; i++)
    if (clasa[i].media > clasa[best].media)
        best = i;
cout << "Premiant: " << clasa[best].nume;
cout << " cu " << clasa[best].media << endl;
```

best ține indicele celui mai bun de până acum; la final el arată spre elevul căutat.

Pentru intrarea: 3 elevi — Ana 9.50, Bogdan 8.20, Carla 9.80 — programul afișează:

```rezultat
Premiant: Carla cu 9.8
```

Sortarea după un câmp

Adesea vrem clasamentul: elevii ordonați descrescător după medie. Putem folosi std::sort (din capitolul de STL) dând o regulă de comparare care privește câmpul media.

CPP

```cpp
bool dupaMedie(Elev a, Elev b) {
    return a.media > b.media;  // mai mare intai
}
// in main, dupa citire:
sort(clasa, clasa + n, dupaMedie);
```

Funcția dupaMedie spune lui sort cine vine primul. Pentru sort ai nevoie de #include <algorithm>.

Structuri imbricate

Un câmp poate fi, la rândul lui, o structură. De exemplu, un Cerc are un centru (un Punct) și o rază.

CPP

```cpp
struct Cerc {
    Punct centru;   // un struct in alt struct
    double raza;
};
Cerc c = {{2, 3}, 5.0};
cout << c.centru.x << endl;  // ajungem prin doua .
```

c.centru e un Punct, deci c.centru.x e coordonata x a centrului. Punctele se înlănțuie.

REZULTAT

```
2
```

Exemplu complet

Punem totul cap la cap: citim elevii și afișăm premiantul.

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

struct Elev {
    string nume;
    double media;
};

int main() {
    int n;
    Elev clasa[100];
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> clasa[i].nume >> clasa[i].media;
    int best = 0;
    for (int i = 1; i < n; i++)
        if (clasa[i].media > clasa[best].media)
            best = i;
    cout << clasa[best].nume << " ";
    cout << clasa[best].media << endl;
    return 0;
}
```

Intrare: 3 Ana 9.50 Bogdan 8.20 Carla 9.80

```
Carla 9.8
```

Teste de verificare

Compilează și rulează fiecare program; compară-ți ieșirea cu rezultatul așteptat.

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

struct Punct { int x, y; };

int main() {
    Punct p = {2, 5};
    p.x += 3;
    cout << p.x + p.y << endl;
    return 0;
}
```

După p.x += 3, avem p.x = 5 și p.y = 5.

```
10
```

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

struct Elev { string nume; int v; };

int main() {
    Elev e[3] = {
        {"Ana", 14},
        {"Dan", 16},
        {"Eva", 15}
    };
    int s = 0;
    for (int i = 0; i < 3; i++)
        s += e[i].v;
    cout << s << endl;
    return 0;
}
```

Adună vârstele celor trei elevi.

```
REZULTAT AȘTEPTAT
45
```

Exerciții propuse

Rezolvă singur:

1. Defineste struct Punct si scrie o functie care spune daca un punct se afla pe axa Ox (y == 0). 2. Citeste n elevi (nume, medie) si afiseaza media intregii clase. 3. Citeste n elevi si afiseaza numele celor cu media peste 9. 4. Defineste struct Dreptunghi (lungime, latime) si o functie care ii calculeaza aria si perimetrul. 5. Citeste n elevi si afiseaza clasamentul lor ordonat descrescator dupa medie.

PARTEA V

5.3 Liste, stive și cozi

Structuri de date liniare

Un vector ține elementele așezate unul lângă altul, în ordine. Dar de multe ori nu contează doar UNDE stau datele, ci REGULA după care le adăugăm și le scoatem. O structură de date liniară este o colecție în care elementele sunt aranjate în șir, unul după altul, iar accesul la ele urmează o anumită ordine. În acest capitol studiem trei structuri liniare clasice: stiva, coada și lista înlănțuită.

Le vom construi de la zero, folosind ce știm deja despre vectori, struct și pointeri. La final, vei vedea că biblioteca standard (STL) le oferă gata făcute — dar a le înțelege „pe dedesubt” te ajută să le folosești corect.

Stiva (LIFO)

O stivă funcționează ca un teanc de farfurii: pui o farfurie nouă deasupra și tot de deasupra o iei. Ultimul element pus este primul scos. De aceea i se spune structură LIFO — Last In, First Out (ultimul intrat, primul ieșit).

Operațiile unei stive:

***push* — adaugă un element în vârf *pop* — scoate elementul din vârf *top* — citește vârful (fără a-l scoate) *vid* — verifică dacă stiva este goală**

O putem implementa simplu cu un vector și un indice numit vârf, care arată câte elemente sunt în stivă (și, totodată, unde se află poziția liberă următoare).

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

const int MAX = 100;
int stiva[MAX];
int varf = 0;  // numarul de elemente

void push(int x) {
    stiva[varf] = x;  // pun deasupra
    varf++;
}

int pop() {
    varf--;
    return stiva[varf];  // scot din varf
}

bool vida() {
    return varf == 0;
}
```

Indicele varf marchează vârful stivei. push crește varf, pop îl scade. Elementele „scoase” rămân fizic în vector, dar sunt ignorate.

Să verificăm comportamentul LIFO punând trei valori și scoțându-le pe rând:

```cpp
int main() {
    push(10);
    push(20);
    push(30);
    while (!vida()) {
        cout << pop() << " ";
    }
    cout << endl;
    return 0;
}
```

```
30 20 10
```

Aplicație: paranteze echilibrate

O folosire clasică a stivei este verificarea parantezelor dintr-un șir. Un șir are parantezele echilibrate dacă fiecare paranteză deschisă „(” se închide corect cu „)”, în ordinea potrivită. Ideea: parcurgem șirul; la fiecare „(” facem push, la fiecare „)” trebuie să avem ce scoate (pop). Dacă la final stiva e goală, parantezele sunt echilibrate.

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

bool echilibrat(string s) {
    int varf = 0;  // stiva simpla de '('
    for (char c : s) {
        if (c == '(') {
            varf++;        // push
        } else if (c == ')') {
            if (varf == 0) return false;
            varf--;        // pop
        }
    }
    return varf == 0;
}

int main() {
    cout << echilibrat("(a+b)*(c-d)") << endl;
    cout << echilibrat("(()") << endl;
    cout << echilibrat(")(") << endl;
    return 0;
}
```

Aici avem doar un tip de paranteză, așa că ne ajunge un contor (o stivă „logică”). varf == 0 la final înseamnă că toate s-au închis. Un „)” care vine pe stivă goală face rezultatul fals imediat.

```
REZULTAT
1
0
0
```

Coada (FIFO)

O coadă funcționează ca un rând la magazin: cine vine primul este servit primul. Adăugăm la spate și scoatem din față. De aceea i se spune structură FIFO — First In, First Out (primul intrat, primul ieșit).

Operațiile unei cozi:

enqueue — adaugă un element la spate dequeue — scoate elementul din față vid — verifică dacă coada este goală

O implementăm cu un vector și DOI indici: fata (de unde scoatem) și spate (unde adăugăm). enqueue scrie la spate și avansează spate; dequeue citește din fata și avansează fata.

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

const int MAX = 100;
int coada[MAX];
int fata = 0, spate = 0;

void enqueue(int x) {
    coada[spate] = x;  // adaug la spate
    spate++;
}

int dequeue() {
    int x = coada[fata];  // scot din fata
    fata++;
    return x;
}

bool vida() {
    return fata == spate;
}
```

Când fata ajunge din urmă pe spate, coada este goală. Această variantă simplă „consumă” pozițiile din vector; o coadă circulară refolosește spațiul revenind la indicele 0 când ajunge la MAX.

Verificăm comportamentul FIFO:

```cpp
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    while (!vida()) {
        cout << dequeue() << " ";
    }
    cout << endl;
    return 0;
}
```

```
10 20 30
```

Observă diferența: la stivă valorile au ieșit invers (30 20 10), la coadă în aceeași ordine (10 20 30).

Lista înlănțuită (introducere)

La vector, elementele stau lipite în memorie, iar mărimea e fixă. Lista înlănțuită rezolvă altfel problema: fiecare element este un nod separat, care păstrează informația lui ȘI un pointer către nodul următor. Nodurile pot fi împrăștiate prin memorie; legăturile (pointerii) le țin într-un șir.

```cpp
struct Nod {
    int info;     // valoarea din nod
    Nod* urm;     // pointer la nodul urmator
};
```

Ultimul nod are urm = nullptr, semn că lista s-a terminat. Un singur pointer, numit de obicei cap, reține primul nod.

Inserarea la început este foarte ieftină: cream un nod nou, îl legăm de fostul cap și mutăm cap pe el. Pentru parcurgere pornim de la cap și urmăm urm până la nullptr.

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

struct Nod {
    int info;
    Nod* urm;
};

Nod* cap = nullptr;

void insereazaInceput(int x) {
    Nod* nou = new Nod;
    nou->info = x;
    nou->urm = cap;  // leg de fostul prim
    cap = nou;       // noul cap
}

void afiseaza() {
    Nod* p = cap;
    while (p != nullptr) {
        cout << p->info << " ";
        p = p->urm;  // trec la urmatorul
    }
    cout << endl;
}

int main() {
    insereazaInceput(10);
    insereazaInceput(20);
    insereazaInceput(30);
    afiseaza();
    return 0;
}
```

Inserăm 10, 20, 30 la început, deci ultimul inserat ajunge primul în listă. p = p->urm avansează parcurgerea nod cu nod.

```
REZULTAT
30 20 10
```

STL: structurile gata făcute

Acum că înțelegi mecanismul, vestea bună: nu trebuie să le rescrii de fiecare dată. Biblioteca standard C++ oferă stack (stivă), queue (coadă) și list (listă înlănțuită), deja testate și optimizate. Le studiem pe larg în capitolul despre STL; aici e suficient să știi că există și că se sprijină exact pe ideile de mai sus.

Teste de verificare

Rulează programele de mai jos și compară cu rezultatul așteptat.

1) Stiva inversează ordinea — punem 1, 2, 3, 4 și le scoatem:

```cpp
int main() {
    for (int i = 1; i <= 4; i++) push(i);
    while (!vida())
        cout << pop() << " ";
    cout << endl;
    return 0;
}
```

```
REZULTAT
4 3 2 1
```

2) Coada păstrează ordinea — aceleași valori, altă structură:

```cpp
int main() {
    for (int i = 1; i <= 4; i++) enqueue(i);
    while (!vida())
        cout << dequeue() << " ";
    cout << endl;
    return 0;
}
```

```
REZULTAT
1 2 3 4
```

3) Verificarea parantezelor pe câteva șiruri:

```cpp
int main() {
    cout << echilibrat("(((()))") << endl;
    cout << echilibrat("((()(()))") << endl;
    cout << echilibrat("(()") << endl;
    return 0;
}
```

```
REZULTAT
1
1
0
```

Exerciții propuse

1) Adaugă funcția top() la stivă, care întoarce vârful fără a-l scoate. Testează că, după push(5), top() dă 5, iar stiva rămâne nevidă.

2) Scrie o funcție care, folosind o stivă, inversează cifrele unui număr citit de la tastatură (ex.: 1234 -> 4321).

3) Modifică verificarea parantezelor ca să accepte și paranteze drepte [ ] și acolade { }, folosind o stivă reală care reține caracterul deschis (nu doar un contor).

4) În lista înlănțuită, scrie o funcție care numără câte noduri are lista, parcurgând-o de la cap până la nullptr.

5) Adaugă în listă o funcție care caută o valoare x și returnează true dacă există vreun nod cu info == x, altfel false.

PARTEA V

5.4 Introducere în STL

vector, string, sort, map, set

Până acum am scris singuri vectori, sortări și căutări. A fost util ca să înțelegem ce se petrece „pe dedesubt”, dar la un concurs sau într-un proiect serios nu ai timp să rescrii de fiecare dată aceleași lucruri. Aici intervine STL — Standard Template Library, biblioteca standard de șabloane a limbajului C++. Ea îți oferă containere (structuri de date gata făcute) și algoritmi (sortare, căutare, minim, maxim) deja scriși, testați și foarte rapizi.

Avantajul este uriaș: cu o singură linie faci ceea ce altfel ar lua zeci de linii. În acest capitol vedem cele mai folosite unelte: vector, string, sort, pair, map și set. Pentru comoditate vom include direct toată biblioteca.

```cpp
#include <bits/stdc++.h>
using namespace std;
```

<bits/stdc++.h> aduce dintr-o singură linie tot ce avem nevoie: vector, string, algorithm, map, set. Este foarte comod la concursuri.

Containerul vector

Un std::vector<T> este un tablou care își poate schimba dimensiunea în timpul execuției. T este tipul elementelor: vector<int> pentru numere întregi, vector<string> pentru cuvinte și așa mai departe. Nu trebuie să-i dai dinainte o dimensiune fixă — crește singur când adaugi elemente.

```cpp
vector<int> v;        // vector gol
v.push_back(10);      // adauga 10 la final
v.push_back(20);
v.push_back(30);
cout << v.size() << endl;  // cate elemente
cout << v[0] << endl;      // primul element
```

push_back adaugă la sfârșit, size() spune câte elemente are, iar v[i] accesează elementul de pe poziția i, exact ca la tablourile obișnuite (indexarea începe de la 0).

```
REZULTAT

3
10
```

Parcurgerea unui vector

Putem parcurge un vector în două feluri. Cu un for clasic, folosind indici, sau cu un for „pe interval” (range-based for), care e mai scurt și mai clar când nu ai nevoie de poziție, ci doar de valori.

CPP

```cpp
vector<int> v = {10, 20, 30};

// for clasic, cu indici
for (int i = 0; i < v.size(); i++)
    cout << v[i] << " ";
cout << endl;

// range-based for: citeste fiecare element
for (auto x : v)
    cout << x << " ";
cout << endl;
```

auto lasă compilatorul să deducă tipul lui x (aici int). Cele două bucle afișează exact același lucru.

REZULTAT

```
10 20 30
10 20 30
```

Recapitulare: string

Tipul std::string este, de fapt, un container ca un vector de caractere. Îl putem concatena cu +, îi aflăm lungimea cu .size() (sau .length()) și îi accesăm caracterele cu [i]. Putem chiar să-l parcurgem cu range-based for.

CPP

```cpp
string s = "liceu";
cout << s.size() << endl;   // 5
cout << s[0] << endl;       // l
string t = s + " 2026";    // concatenare
cout << t << endl;
```

Adunarea a două string-uri le lipește unul după altul. Caracterul s[0] este primul din șir.

REZULTAT

```
5
l
liceu 2026
```

Algoritmul sort()

Funcția sort() din STL ordonează un container. Îi dai începutul și sfârșitul zonei: pentru un vector v scriem sort(v.begin(), v.end()). Implicit, sortează crescător. Această singură linie înlocuiește toate sortările pe care le-am scris de mână în capitolul anterior.

CPP

```cpp
vector<int> v = {5, 1, 4, 2, 8};
sort(v.begin(), v.end());  // crescator
for (auto x : v) cout << x << " ";
cout << endl;
```

v.begin() arată spre primul element, v.end() „dincolo” de ultimul. Rezultatul este vectorul ordonat crescător.

REZULTAT

```
1 2 4 5 8
```

Sortare descrescătoare

Ca să sortăm descrescător, dăm lui sort() un al treilea argument: regula de comparare. Cea mai simplă este greater<int>(), care înseamnă „de la mare la mic”. Alternativ, putem folosi o funcție lambda — o mini-funcție scrisă pe loc, între acolade.

CPP

```cpp
vector<int> v = {5, 1, 4, 2, 8};

// varianta 1: greater
sort(v.begin(), v.end(), greater<int>());

// varianta 2: lambda (acelasi efect)
sort(v.begin(), v.end(),
     [](int a, int b) { return a > b; });

for (auto x : v) cout << x << " ";
cout << endl;
```

Lambda [](int a, int b){ return a > b; } spune „a vine înaintea lui b dacă a este mai mare”. Asta dă ordine descrescătoare.

REZULTAT

```
8 5 4 2 1
```

Minim, maxim și reverse

STL are și alți algoritmi scurți și utili. Cu min și max comparăm două valori. Cu min_element și max_element găsim cel mai mic, respectiv cel mai mare element dintr-un vector (ne dau poziția, deci punem * în față ca să luăm valoarea). Cu reverse inversăm ordinea elementelor.

```cpp
vector<int> v = {5, 1, 4, 2, 8};
cout << min(3, 7) << endl;   // 3
cout << max(3, 7) << endl;   // 7
cout << *min_element(v.begin(), v.end())
     << endl;                // 1
cout << *max_element(v.begin(), v.end())
     << endl;                // 8
reverse(v.begin(), v.end()); // inverseaza
for (auto x : v) cout << x << " ";
cout << endl;
```

Steluța (*) ia valoarea din locul indicat de min_element / max_element. reverse răstoarnă vectorul pe loc.

```
3
7
1
8
8 2 4 1 5
```

Perechea pair

Un std::pair<A, B> ține împreună două valori, posibil de tipuri diferite. Le accesăm cu .first și .second. Este util când vrei să legi două informații, de exemplu o notă și numele elevului, sau coordonatele x și y ale unui punct.

```cpp
pair<string, int> elev = {"Ana", 10};
cout << elev.first << " are nota "
     << elev.second << endl;
```

.first este primul element al perechii (string-ul), .second al doilea (int-ul).

```
Ana are nota 10
```

Dicționarul map

Un std::map<cheie, valoare> asociază fiecărei chei o valoare, ca un dicționar: cauți după cheie și primești valoarea. Cheile sunt unice și păstrate ordonate. Un map este perfect pentru a număra de câte ori apare ceva — de exemplu frecvența numerelor sau a cuvintelor.

```cpp
map<int, int> frecv;
int x;
vector<int> date = {3, 1, 3, 2, 3, 1};
for (auto v : date)
    frecv[v]++;  // creste numaratorul

// parcurgem perechile (cheie, valoare)
for (auto p : frecv)
    cout << p.first << " -> "
         << p.second << endl;
```

frecv[v]++ creează automat cheia v cu valoare 0 dacă nu exista, apoi o crește. La parcurgere, p.first e numărul, p.second e de câte ori apare. Cheile ies în ordine crescătoare.

```
1 -> 2
2 -> 1
3 -> 3
```

Mulțimea set

Un std::set este o mulțime: păstrează doar elemente unice și le ține automat ordonate crescător. Dacă inserezi de două ori aceeași valoare, ea apare o singură dată. E ideal când vrei să elimini duplicatele dintr-un șir de date.

```cpp
set<int> s;
vector<int> date = {4, 2, 4, 1, 2, 4};
for (auto v : date)
    s.insert(v);  // duplicatele se ignora

cout << "Distincte: " << s.size() << endl;
for (auto x : s) cout << x << " ";
cout << endl;
```

insert adaugă valoarea doar dacă nu există deja. size() ne dă numărul de elemente distincte, iar parcurgerea le scoate ordonate crescător.

```
Distincte: 3
1 2 4
```

Cât cod economisim

Gândește-te puțin: sortarea de mână era o funcție de vreo zece linii, eliminarea duplicatelor cerea bucle imbricate, iar numărarea frecvențelor presupunea un tablou indexat cu grijă. Cu STL, fiecare dintre aceste sarcini devine una sau două linii. Mai puțin cod înseamnă mai puține greșeli, rezolvare mai rapidă la concurs și programe mai ușor de

citit. Tocmai de aceea STL este unealta preferată a olimpicilor.

Teste de verificare

Compilează și rulează aceste programe; compară ieșirea cu rezultatul așteptat.

CPP

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v = {7, 3, 9, 1, 4};
    sort(v.begin(), v.end());
    for (auto x : v) cout << x << " ";
    cout << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
1 3 4 7 9
```

CPP

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> s;
    int date[] = {5, 5, 2, 8, 2, 5};
    for (int x : date) s.insert(x);
    cout << s.size() << endl;
    return 0;
}
```

REZULTAT AȘTEPTAT

```
3
```

CPP

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    map<string, int> m;
    vector<string> w =
        {"a", "b", "a", "a"};
    for (auto x : w) m[x]++;
    cout << m["a"] << " " << m["b"]
         << endl;
    return 0;
}
```

```
REZULTAT AȘTEPTAT
3 1
```

Exerciții propuse

1. Citește n numere într-un vector, sortează-le descrescător cu sort() și greater<int>(), apoi afișează-le.

2. Citește un șir de numere și folosește un set ca să afișezi câte valori distincte conține și care sunt ele, în ordine crescătoare.

3. Citește un text format din cuvinte și folosește un map<string, int> ca să afișezi de câte ori apare fiecare cuvânt, în ordine alfabetică.

4. Citește un vector și afișează cel mai mic și cel mai mare element folosind min_element și max_element (cu * în față).

5. Creează un vector de pair<string, int> cu nume de elevi și notele lor, apoi sortează-l după notă descrescător, folosind o lambda care compară p.second.

PARTEA V

5.5 Complexitatea algoritmilor

Notația O și eficiența

Doi algoritmi pot rezolva aceeași problemă, dar unul termină într-o clipită, iar celălalt te face să aștepți minute întregi sau chiar ore. La concursuri și la bacalaureat fiecare problemă are o limită de timp (de obicei 1 secundă) și o limită de memorie. Un program corect, dar prea lent, ia zero puncte. De aceea nu ne interesează doar DACĂ un algoritm funcționează, ci și CÂT de repede funcționează când datele cresc.

De ce contează eficiența

Un calculator obișnuit execută aproximativ 100 de milioane (adică 10^8) de operații simple într-o secundă. Pentru date mici, orice algoritm pare instant. Dar dacă numărul de date n ajunge la un milion, diferența dintre un algoritm bun și unul slab devine uriașă. Pe lângă timp contează și memoria: un vector cu zece milioane de întregi ocupă deja zeci de megaocteți.

Întrebarea cheie:

Cum crește timpul de execuție atunci când dimensiunea datelor n se mărește?

Numărăm operațiile în funcție de n

Ideea de bază este să numărăm câte operații „importante” face algoritmul, exprimat printr-o formulă în funcție de n (numărul de date). Nu cronometrăm cu ceasul — timpul depinde de calculator. În schimb numărăm pași: comparații, adunări, atribuiri. Dacă parcurgem un vector de n elemente o singură dată, facem cam n pași.

```cpp
long long suma = 0;
for (int i = 0; i < n; i++) {
    suma += v[i];   // se executa de n ori
}
```

Bucla rulează de n ori, deci numărul de operații crește proporțional cu n.

Notația O mare

Pentru a compara algoritmi folosim notația O mare (citită „o de...”), care descrie ORDINUL de creștere. Ne interesează doar cum se comportă timpul pentru n foarte mare, așa că ignorăm constantele și termenii mici. De exemplu, dacă un algoritm face 3*n + 7 pași,

spunem că are complexitatea O(n): pentru n mare, termenul dominant este n, iar factorul 3 și constanta 7 devin neglijabile.

Reguli de simplificare:

1. Păstrăm doar termenul care crește cel mai repede. 2. Eliminăm constantele multiplicative (5n -> n). 3. Exemplu: n^2 + 100n + 50 => O(n^2).

Clasele uzuale de complexitate

O(1) — constant: timpul nu depinde de n. Accesul la un element de vector prin indice se face în același timp, indiferent cât de mare e vectorul.

```cpp
int x = v[0];       // O(1)
int y = v[n - 1];   // tot O(1)
```

O(log n) — logaritmic: la fiecare pas eliminăm jumătate din candidați. Tipic căutării binare. Pentru un milion de elemente sunt suficienți cam 20 de pași.

```cpp
int st = 0, dr = n - 1;
while (st <= dr) {        // O(log n)
    int m = (st + dr) / 2;
    if (v[m] == x) return m;
    else if (v[m] < x) st = m + 1;
    else dr = m - 1;
}
```

O(n) — liniar: parcurgem datele o dată. Suma elementelor, căutarea secvențială, găsirea maximului.

O(n log n): sortările eficiente (sort din STL, merge sort, quick sort). Este cel mai bun ordin posibil pentru o sortare bazată pe comparații.

O(n^2) — pătratic: două cicluri imbricate peste aceleași date. Sortările simple (bule, selecție, inserție) sau compararea fiecărei perechi de elemente.

```cpp
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        // se executa de n*n ori  -> O(n^2)
        cout << i * j << " ";
```

*Ciclu în ciclu: înmulțim numerele de repetări, deci n * n = n^2.*

O(2^n) și O(n!) — exponențial și factorial: tipice backtracking-ului care încearcă toate variantele (toate submulțimile, toate permutările). Cresc atât de repede încât devin imposibil de rulat chiar și pentru n=30 sau n=15. Le folosim doar când n este foarte mic.

Cum estimezi complexitatea unui cod

Regulă practică:

Instrucțiuni una după alta -> aduni complexitățile (păstrezi cea mai mare). Cicluri imbricate -> înmulțești numărul de repetări. Înjumătățire repetată -> apare un log n.

Un singur ciclu de la 0 la n dă O(n). Două cicluri imbricate, fiecare până la n, dau O(n^2). Trei imbricate dau O(n^3). Dacă în interiorul unui ciclu de n pași faci o căutare binară (log n), complexitatea totală este O(n log n).

Cât de mare poate fi n într-o secundă

Pornind de la ~10^8 operații pe secundă, putem estima cam ce dimensiune de date „încape” în limita de timp pentru fiecare clasă de complexitate:

Orientativ (limită ~1 secundă):

O(log n) -> n practic nelimitat O(n) -> n pana la ~100.000.000 O(n log n) -> n pana la ~5.000.000 O(n^2) -> n pana la ~10.000 O(n^3) -> n pana la ~500 O(2^n) -> n pana la ~25 O(n!) -> n pana la ~11

Liniar vs binar: o comparație concretă

Să presupunem că avem un vector sortat cu n = 1.000.000 de elemente. Căutarea secvențială face în cel mai rău caz un milion de comparații. Căutarea binară, înjumătățind de fiecare dată, face cel mult 20 de comparații, fiindcă 2^20 este puțin peste un milion. Aceasta este diferența dintre O(n) și O(log n).

n = 1.000.000:

Căutare liniară (O(n)) -> ~1.000.000 pași Căutare binară (O(log n)) -> ~20 pași

Teste de verificare

Pentru fiecare cod, estimează complexitatea în notația O. Răspunsurile sunt date după programe.

```cpp
// Cod A
int s = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        s += v[i] * v[j];
```

Răspuns A: două cicluri imbricate, fiecare n -> O(n^2).

```cpp
// Cod B
int p = 1;
while (p < n)
    p = p * 2;
```

Răspuns B: p se dublează la fiecare pas (1, 2, 4, 8, ...) până depășește n -> O(log n).

```cpp
// Cod C ruleaza si afiseaza rezultatul
#include <iostream>
using namespace std;

int main() {
    int n = 5, pasi = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            pasi++;
    cout << pasi << endl;
    return 0;
}
```

*Răspuns C: bucla interioară rulează de n*n ori, deci complexitate O(n^2); pentru n=5 se fac 25 de pași.*

```
REZULTAT AȘTEPTAT
25
```

Exerciții propuse

1. Estimează complexitatea unui algoritm care, pentru un vector de n elemente, afișează toate perechile (i, j) cu i < j. Câte perechi sunt și ce ordin de creștere are?

2. Scrie un program care numără câți pași face o căutare binară pentru a găsi o valoare într-un vector sortat de n = 1.000.000 elemente și verifică experimental că sunt cel mult 20.

3. Ai de rezolvat o problemă cu n = 100.000 și limita de 1 secundă. Un algoritm O(n^2) este suficient? Dar unul O(n log n)? Justifică folosind tabelul din capitol.

4. Pentru codul: trei cicluri imbricate, fiecare de la 0 la n, scrie complexitatea în notația O și calculează numărul de pași pentru n = 100.

5. Explică de ce un algoritm de backtracking care generează toate permutările a n elemente are complexitatea O(n!) și de ce devine inutilizabil deja pentru n = 15.

PARTEA V

5.6 Introducere în grafuri

Reprezentare și parcurgeri BFS/DFS

Imaginează-ți harta unui oraș: niște intersecții legate prin străzi. Sau o rețea de prieteni pe o platformă online: persoane legate prin relații de prietenie. Sau hărțile de zboruri dintre aeroporturi. Toate aceste situații au aceeași structură: niște obiecte și niște legături între ele. Această structură matematică se numește graf.

Un graf este format din două lucruri: o mulțime de noduri (numite și vârfuri) și o mulțime de muchii. Fiecare muchie unește două noduri. Nodurile le numerotăm de obicei cu 0, 1, 2, ..., n-1 (sau de la 1 la n), iar o muchie o scriem ca o pereche, de exemplu {1, 3}, adică „nodul 1 este legat de nodul 3”.

Orientat sau neorientat

Dacă o muchie poate fi parcursă în ambele sensuri, graful este neorientat: muchia {1, 3} înseamnă că de la 1 ajungi la 3 și de la 3 ajungi la 1 (ca o stradă cu două sensuri sau o prietenie reciprocă). Dacă legăturile au un sens, graful este orientat: muchia (1, 3) merge doar de la 1 la 3 (ca o stradă cu sens unic sau relația „X îl urmărește pe Y”). În acest caz vorbim de arce, nu de muchii.

Gradul unui nod, într-un graf neorientat, este numărul de muchii care ies din el. De exemplu, dacă nodul 1 e legat de 0, 2 și 3, atunci gradul lui este 3. Suma gradelor tuturor nodurilor este mereu dublul numărului de muchii (fiecare muchie se numără la ambele capete).

Un drum este o succesiune de noduri în care fiecare două noduri consecutive sunt legate printr-o muchie, de exemplu 0 - 1 - 3. Dacă un drum se închide, ajungând înapoi la nodul de plecare, avem un ciclu. Spunem că graful este conex dacă între oricare două noduri există un drum, adică „totul e legat de tot”. Dacă nu, graful se rupe în mai multe componente conexe (insule separate).

Reprezentarea 1: matricea de adiacență

Pentru ca un program să lucreze cu un graf, trebuie să-l ținem cumva în memorie. Prima variantă este o matrice pătratică a[n][n], unde a[i][j] = 1 dacă există muchie între i și j, altfel 0. Pentru un graf neorientat matricea este simetrică: a[i][j] = a[j][i].

```cpp
int a[100][100];   // matricea de adiacenta
int n, m;          // noduri si muchii

// citim m muchii (graf neorientat)
cin >> n >> m;
for (int k = 0; k < m; k++) {
    int x, y;
    cin >> x >> y;
    a[x][y] = 1;
    a[y][x] = 1;   // simetric
}
```

Pentru un graf orientat punem doar a[x][y] = 1, fără linia simetrică.

Avantajul matricei: verificăm instant dacă două noduri sunt legate (citim a[i][j]). Dezavantajul: ocupă n×n celule, oricâte muchii ar fi. La 10 000 de noduri matricea ar avea 100 de milioane de celule, prea mult. Matricea e bună doar pentru grafuri mici.

Reprezentarea 2: listele de adiacență

A doua variantă reține, pentru fiecare nod, lista vecinilor lui. Folosim un vector de vectori: adia[i] conține toate nodurile legate de i. Astfel ocupăm spațiu proporțional cu numărul de muchii, nu cu n×n.

```cpp
#include <vector>
using namespace std;

vector<int> adia[100];   // listele de vecini
int n, m;

cin >> n >> m;
for (int k = 0; k < m; k++) {
    int x, y;
    cin >> x >> y;
    adia[x].push_back(y);
    adia[y].push_back(x);   // neorientat
}
```

Avantaj: economic la memorie și parcurgem ușor toți vecinii. Dezavantaj: ca să verificăm o muchie anume trebuie să căutăm în listă.

Parcurgerea în adâncime (DFS)

A parcurge un graf înseamnă a vizita toate nodurile la care putem ajunge, pornind dintr-un nod dat. Parcurgerea în adâncime (DFS, Depth-First Search) merge cât de departe poate pe un drum, apoi se întoarce și încearcă alt drum, exact ca la backtracking. Pentru a nu vizita un nod de două ori ținem un vector vizitat[].

```cpp
bool vizitat[100];

void dfs(int nod) {
    vizitat[nod] = true;
    cout << nod << ' ';
    // mergem la fiecare vecin nevizitat
    for (int vecin : adia[nod])
        if (!vizitat[vecin])
            dfs(vecin);
}
```

DFS este recursiv: fiecare nod cheamă dfs pentru vecinii săi nevizitați.

Parcurgerea în lățime (BFS)

Parcurgerea în lățime (BFS, Breadth-First Search) vizitează nodurile în „valuri”: întâi nodul de start, apoi toți vecinii lui, apoi vecinii vecinilor și așa mai departe. Pentru asta folosim o coadă (std::queue): scoatem un nod, îi punem în coadă toți vecinii nevizitați, repetăm până se golește coada.

```cpp
#include <queue>

void bfs(int start) {
    queue<int> q;
    vizitat[start] = true;
    q.push(start);
    while (!q.empty()) {
        int nod = q.front();
        q.pop();
        cout << nod << ' ';
        for (int vecin : adia[nod])
            if (!vizitat[vecin]) {
                vizitat[vecin] = true;
                q.push(vecin);
            }
    }
}
```

Marcăm nodul ca vizitat în momentul în care îl punem în coadă, ca să nu intre de mai multe ori.

Un exemplu complet

Să luăm un graf neorientat cu 6 noduri (0..5) și muchiile: 0-1, 0-2, 1-3, 2-3, 4-5. Observă că nodurile 0,1,2,3 formează o componentă, iar 4,5 alta separată. Pornim ambele parcurgeri din nodul 0.

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

vector<int> adia[6];
bool viz[6];

void dfs(int nod) {
    viz[nod] = true;
    cout << nod << ' ';
    for (int v : adia[nod])
        if (!viz[v]) dfs(v);
}

int main() {
    int m = 5;
    int mu[5][2] = {{0,1},{0,2},{1,3},
                    {2,3},{4,5}};
    for (int k = 0; k < m; k++) {
        int x = mu[k][0], y = mu[k][1];
        adia[x].push_back(y);
        adia[y].push_back(x);
    }
    cout << "DFS: ";
    dfs(0);
    cout << '\n';
    return 0;
}
```

Vecinii sunt parcurși în ordinea în care au fost adăugați în listă.

```
REZULTAT
DFS: 0 1 3 2
```

DFS pleacă din 0, intră la primul vecin 1, de acolo la 3, din 3 vede 2 (nevizitat) și îl vizitează. Nodurile 4 și 5 nu apar: sunt în altă componentă. Dacă în main folosim bfs(0) în loc de dfs(0), valurile dau altă ordine:

```
REZULTAT (BFS)
BFS: 0 1 2 3
```

Aplicație: numărul de componente conexe

Câte „insule” separate are graful? Pornim o parcurgere din fiecare nod încă nevizitat; fiecare pornire nouă descoperă exact o componentă întreagă. Numărăm câte porniri am făcut.

```cpp
int componente() {
    int nr = 0;
    for (int i = 0; i < n; i++)
        if (!viz[i]) {
            nr++;       // o componenta noua
            dfs(i);     // o marcam toata
        }
    return nr;
}
```

Pentru graful exemplu rezultatul este 2: {0,1,2,3} și {4,5}.

Verificarea dacă există drum între două noduri u și v este și mai simplă: facem o parcurgere din u (DFS sau BFS) și la final ne uităm la viz[v]. Dacă v a fost vizitat, există drum; altfel nu.

Teste de verificare

Test 1. Pe graful exemplu (muchiile 0-1, 0-2, 1-3, 2-3, 4-5), apelează componente() după ce ai golit vectorul viz. Programul trebuie să afișeze:

```cpp
    cout << componente() << '\n';
```

```
REZULTAT AȘTEPTAT
2
```

Test 2. Verifică dacă există drum de la 0 la 5 pe același graf:

```cpp
    for (int i = 0; i < n; i++) viz[i] = false;
    dfs(0);   // marcheaza componenta lui 0
    if (viz[5]) cout << "DA\n";
    else        cout << "NU\n";
```

```
REZULTAT AȘTEPTAT
NU
```

Test 3. Adaugă muchia 3-4 (leagă cele două insule) și rerulează numărul de componente. Acum totul e conex:

```
REZULTAT AȘTEPTAT
1
```

Exerciții propuse

1. Citește un graf neorientat (n noduri, m muchii) în liste de adiacență și afișează gradul fiecărui nod.

2. Scrie un program care citește un graf și afișează ordinea de vizitare BFS pornind dintr-un nod dat de la tastatură.

3. Determină numărul de componente conexe ale unui graf citit de la tastatură, folosind o parcurgere.

4. Date două noduri u și v, decide (DA/NU) dacă există drum între ele, fără a afișa parcurgerea.

5. Reprezintă același graf și prin matrice de adiacență, și prin liste; afișează matricea și verifică, pentru câteva perechi, că a[i][j] = 1 exact atunci când j apare în adia[i].

PARTEA VI

Instrumente moderne ale programatorului

Cum compilezi cu GCC pe Windows, cum versionezi codul cu Git și cum folosești Docker.

PARTEA VI

6.1 Compilare cu GCC, Git și Docker pe Windows

Mediul de lucru al programatorului modern

Până acum ai scris programe într-un mediu gata pregătit, care apăsa „Run" în locul tău. Dar un programator adevărat își stăpânește uneltele: știe să compileze din linia de comandă, să-și păstreze istoricul codului și să-l ruleze identic pe orice calculator. Acest capitol final îți pune în mână trei unelte esențiale — GCC, Git și Docker — toate pe Windows.

Nu trebuie să le memorezi pe toate dintr-o dată. Citește, încearcă pe calculatorul tău și revino aici când ai nevoie. Acestea sunt unelte pe care le vei folosi ani de zile.

Compilarea cu GCC pe Windows

Calculatorul nu înțelege direct C++. Un compilator este un program care traduce codul-sursă (textul scris de tine) într-un fișier executabil pe care procesorul îl poate rula. GCC (GNU Compiler Collection) este cel mai cunoscut compilator liber; comanda lui pentru C++ se numește g++.

Pe Windows, GCC nu vine instalat implicit. Cea mai simplă cale este pachetul MinGW-w64, pe care îl poți obține fie prin MSYS2 (un mediu de pachete pentru Windows), fie descărcând arhiva gata făcută WinLibs. După instalare, trebuie să adaugi folderul care conține g++.exe (de exemplu C:\mingw64\bin) în variabila de mediu PATH, ca să-l poți chema din orice folder.

De reținut:

Cod-sursă (.cpp) --g++--> Executabil (.exe) Compilatorul traduce o singură dată; executabilul rulează apoi de câte ori vrei.

După instalare, deschide PowerShell și verifică dacă g++ a fost găsit în PATH:

```
POWERSHELL
g++ --version
```

Dacă apare versiunea, GCC e instalat și vizibil în PATH.

```
REZULTAT
g++ (MinGW-w64) 13.2.0
Copyright (C) 2023 Free Software Foundation
```

Dacă în loc de versiune primești „g++ : The term 'g++' is not recognized", înseamnă că folderul bin nu e în PATH. Verifică din nou pasul de instalare.

Acum compilează un program. Presupunem că ai un fișier program.cpp în folderul curent. Comanda de bază este simplă:

```
POWERSHELL
g++ program.cpp -o program.exe
.\program.exe
```

-o dă numele executabilului; .\program.exe îl rulează.

Pe Windows, în PowerShell, rulezi executabilul scriind .\program.exe (punctul și bara indică „din folderul curent"). Dacă omiți -o, GCC creează implicit a.exe.

În practică adaugi câteva flag-uri foarte utile: -Wall pornește toate avertismentele (te ajută să prinzi greșeli ascunse), -std=c++17 cere standardul modern al limbajului, iar -O2 cere optimizări care fac programul mai rapid. O comandă completă, recomandată, arată așa:

```
POWERSHELL
g++ -Wall -std=c++17 -O2 program.cpp -o program.exe
```

-Wall = avertismente, -std=c++17 = standard, -O2 = optimizare.

Când codul are o greșeală, GCC nu produce executabilul, ci afișează o eroare de compilare. Învață să o citești: ea îți spune fișierul, linia și natura problemei. De pildă, dacă uiți punctul și virgula după o instrucțiune:

```
CPP
int main() {
    int x = 5    // lipsește ;
    return 0;
}
```

Programul de mai sus nu compilează — lipsește ; după 5.

```
EROARE DE COMPILARE
program.cpp:2:14: error: expected ';'
before 'return'
    int x = 5
             ^
```

Citește mereu PRIMA eroare: deseori, o singură greșeală generează un lanț de mesaje. „2:14" înseamnă linia 2, coloana 14. Corectează, recompilează, repetă.

Git — versionarea codului

Git este un sistem de versionare: ține minte istoricul fișierelor tale. Cu Git poți reveni la o versiune de ieri, poți vedea exact ce ai schimbat și poți colabora cu alții fără să vă suprascrieți munca. Este, fără exagerare, unealta pe care o folosesc aproape toți programatorii din lume.

Unitatea de bază în Git este commit-ul: o „fotografie" a stării proiectului la un moment dat, însoțită de un mesaj care explică ce ai făcut. Șirul de commit-uri formează istoricul.

Ce este un commit:

O fotografie salvată a proiectului, cu un mesaj care spune CE ai schimbat. Poți reveni oricând la orice commit.

Înainte de prima folosire, spune-i lui Git cine ești (numele apare lângă fiecare commit). Faci asta o singură dată, global:

```
POWERSHELL
git config --global user.name "Ana Pop"
git config --global user.email "ana@exemplu.ro"
```

--global = setare valabilă pentru toate proiectele tale.

Acum fluxul de bază într-un folder cu cod. Intri în folderul proiectului și pornești un depozit (repository) Git, apoi verifici starea:

```
POWERSHELL
git init
git status
```

git init creează depozitul; git status arată ce s-a schimbat.

```
REZULTAT
On branch main
No commits yet
Untracked files:
  program.cpp
```

„Untracked" înseamnă că Git vede fișierul, dar nu-l urmărește încă. Îl adaugi în zona de pregătire cu git add, apoi creezi commit-ul cu un mesaj clar:

```powershell
git add .
git commit -m "Primul meu program"
git log
```

add . adaugă tot; -m dă mesajul; git log arată istoricul.

git add . adaugă toate fișierele modificate; punctul înseamnă „folderul curent". git log îți arată lista commit-urilor, fiecare cu un cod (hash), autorul și mesajul.

Nu vrei să salvezi în Git fișierele generate automat, cum sunt executabilele .exe — ele se pot recrea oricând prin compilare. Creează un fișier numit .gitignore care le exclude:

```bash
# .gitignore
*.exe
*.o
a.exe
```

Liniile spun lui Git ce să NU urmărească.

Până acum codul stă doar pe calculatorul tău. Ca să-l salvezi online și să colaborezi, îl trimiți pe GitHub. Întâi creezi un repo gol pe github.com, apoi îl legi de proiectul tău local și trimiți codul prima dată:

```powershell
git remote add origin <url-ul-repo-ului>
git push -u origin main
```

remote add leagă repo-ul; push -u trimite și reține legătura.

„origin" este numele scurt al depozitului de pe GitHub, iar „main" este ramura principală. Opțiunea -u o folosești o singură dată; după aceea, pentru orice modificare ulterioară e suficient git push. Ca să aduci pe calculatorul tău schimbările făcute de alții, folosești git pull.

Iată un scenariu complet, exact cum lucrezi zi de zi: modifici programul, îl adaugi, faci commit cu un mesaj și îl trimiți pe GitHub:

```powershell
# după ce ai editat program.cpp
git add program.cpp
git commit -m "Adaug citirea de la tastatura"
git push
```

Ciclul de bază: editezi -> add -> commit -> push.

Docker — imagini și containere

Ai întâlnit vreodată replica „dar la mine merge"? Programul tău funcționează pe calculatorul tău, dar pe altul nu — pentru că au versiuni diferite de compilator, biblioteci sau setări. Docker rezolvă exact asta: împachetează programul împreună cu tot mediul de care are nevoie, astfel încât să ruleze identic peste tot.

Două noțiuni de bază trebuie deosebite. O imagine este un șablon înghețat — conține un sistem, uneltele și codul. Un container este o instanță vie care rulează, pornită dintr-o imagine.

Imagine vs container:

Imaginea = rețeta de prăjitură (șablon). Containerul = prăjitura gata făcută (rulează). Dintr-o rețetă poți face oricâte prăjituri.

Pe Windows, instalezi Docker Desktop (de pe docker.com), îl pornești și aștepți să apară starea „Engine running". Atât — restul se face din linia de comandă.

Comenzile de bază. Descarci o imagine de pe internet cu pull, vezi imaginile descărcate cu images și containerele care rulează cu ps:

```
POWERSHELL
docker pull gcc
docker images
docker ps
```

pull descarcă imaginea oficială gcc; images/ps le listează.

Un prim test simpatic: imaginea hello-world doar confirmă că Docker funcționează.

```
POWERSHELL
docker run hello-world
```

run pornește un container dintr-o imagine.

```
REZULTAT
Hello from Docker!
This message shows that your
installation is working correctly.
```

Acum partea utilă: poți compila și rula un program C++ FĂRĂ să ai g++ instalat pe Windows, folosind imaginea oficială gcc. Trucul este să „montezi" folderul tău curent în interiorul containerului:

```powershell
docker run --rm -v ${PWD}:/app -w /app gcc \
  bash -c "g++ program.cpp -o program && ./program"
```

Compilează și rulează program.cpp folosind imaginea gcc.

Să descâlcim opțiunile: -v ${PWD}:/app montează folderul curent (${PWD} = calea curentă în PowerShell) la /app în container, astfel încât containerul vede fișierele tale; -w /app stabilește /app ca director de lucru; --rm șterge automat containerul după ce termină, ca să nu rămână gunoaie. În interior rulează g++ și apoi programul rezultat.

Pentru proiecte serioase scrii instrucțiunile într-un fișier numit Dockerfile, ca să nu mai tastezi comenzi lungi. Iată unul care pleacă de la imaginea gcc, copiază programul, îl compilează și îl rulează:

```dockerfile
FROM gcc:latest
WORKDIR /app
COPY program.cpp .
RUN g++ -O2 program.cpp -o program
CMD ["./program"]
```

FROM = imaginea de bază; COPY/RUN = construcția; CMD = ce rulează.

Din folderul cu Dockerfile, construiești o imagine proprie (îi dai un nume cu -t) și o rulezi:

```powershell
docker build -t programul-meu .
docker run --rm programul-meu
```

build creează imaginea (punctul = folderul curent); run o pornește.

De ce merită:

Cu un Dockerfile, oricine clonează proiectul îl poate compila și rula cu DOUĂ comenzi, fără să-și instaleze nimic în plus.

Teste de verificare

1. Salvează programul de mai jos ca salut.cpp, compilează-l cu g++ folosind -Wall și -std=c++17, apoi rulează executabilul și verifică ieșirea.

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

int main() {
    cout << "Salut din executabil!" << '\n';
    return 0;
}
```

Compilează cu: g++ -Wall -std=c++17 salut.cpp -o salut.exe

```
Salut din executabil!
```

2. Creează un folder nou, copiază salut.cpp în el, apoi inițializează un depozit Git (git init), adaugă fișierul (git add .) și fă un commit cu mesajul „Programul de salut". Verifică istoricul cu git log.

3. Adaugă un .gitignore care ignoră *.exe, recompilează programul (ca să apară salut.exe) și verifică cu git status că executabilul NU mai este propus pentru commit.

4. Rulează același salut.cpp într-un container Docker cu imaginea oficială gcc, montând folderul curent, și confirmă că ieșirea este identică cu cea de la punctul 1.

```powershell
docker run --rm -v ${PWD}:/app -w /app gcc \
  bash -c "g++ salut.cpp -o salut && ./salut"
```

Aceeași ieșire ca la compilarea locală, dar din container.

Exerciții propuse

1. Scrie un program care cere un număr de la tastatură și afișează tabla înmulțirii lui. Compilează-l cu toate cele trei flag-uri (-Wall -std=c++17 -O2) și rezolvă orice avertisment apare.

2. Introdu intenționat o greșeală (șterge un ; sau o acoladă), recompilează și citește mesajul de eroare. Notează linia și coloana indicate, apoi corectează.

3. Pune un proiect mic (2–3 fișiere) sub Git, fă cel puțin trei commit-uri cu mesaje clare și creează un repo pe GitHub în care să-l trimiți cu git push.

4. Scrie un Dockerfile care compilează și rulează unul dintre programele tale; construiește imaginea cu un nume la alegere și rulează-o. Explică în comentarii ce face fiecare linie.

5. Cercetează diferența dintre git push și git pull și descrie, în câteva fraze, un scenariu de colaborare cu un coleg în care folosiți amândouă comenzile pe același repo.

ÎNCHEIERE

Încotro mergi mai departe

Olimpiade, bacalaureat, proiecte personale — și cum continui să înveți.

Încheiere

Încotro mergi mai departe

Cât drum ai parcurs

Îți mai amintești primul tău program? Acel „Salut, lume!" care a apărut, timid, pe ecranul negru al consolei? Pe atunci nu știai ce este o variabilă, iar punctul și virgula ți se părea o capcană așezată dinadins la capătul fiecărei linii. Acum, dacă privești în urmă, vezi cât de departe ai ajuns: de la prima instrucțiune de afișare ai trecut prin decizii și repetiții, prin funcții și tablouri, prin șiruri de caractere și fișiere, prin recursivitate, structuri, pointeri și liste înlănțuite, până la sortări, căutări, stive, cozi, arbori și grafuri, și până la uneltele moderne ale limbajului C++.

Fiecare capitol pe care l-ai parcurs nu a fost doar o lecție de sintaxă, ci un mod nou de a gândi. Ai învățat să descompui o problemă mare în pași mici, să cauți tiparul ascuns dintr-un șir de date, să verifici dacă soluția ta chiar funcționează. Aceste deprinderi rămân cu tine indiferent de limbajul pe care îl vei folosi mai departe.

Ține minte:

Programezi învățând, înveți programând. Niciun programator nu s-a născut știind — fiecare a greșit, a căutat, a încercat din nou. Perseverența scrie cod mai bun decât talentul grăbit.

Concursuri și olimpiade

Dacă ți-a plăcut să rezolvi probleme și simți fiorul plăcut atunci când un algoritm „iese", lumea concursurilor de informatică te așteaptă. În România, Olimpiada de Informatică are mai multe etape: începi cu etapa locală, urcă apoi la etapa județeană (OJI — Olimpiada Județeană de Informatică), iar cei mai buni ajung la etapa națională (ONI — Olimpiada Națională de Informatică). De acolo, lotul național selectează echipa care reprezintă țara la olimpiadele internaționale. Pe lângă acestea există numeroase concursuri online, deschise tuturor, unde poți concura oricând de acasă.

Ca să te antrenezi, ai la dispoziție platforme excelente, multe în limba română: pbinfo.ro are sute de probleme organizate exact pe materia de liceu și îți verifică automat soluțiile; infoarena.ro adună o arhivă uriașă de probleme de algoritmică și o comunitate activă; Kilonova este o platformă modernă, prietenoasă, cu probleme pentru toate nivelurile; iar codeforces.com te pune față în față cu programatori din toată lumea în concursuri cronometrate. Începe cu probleme ușoare, rezolvă constant câte puțin în fiecare zi și vei vedea cum progresul vine de la sine.

Bacalaureatul la informatică

Pentru mulți dintre voi urmează și o probă concretă: bacalaureatul la informatică, susținut ca probă la alegere a profilului. Aici ți se cere să stăpânești noțiunile fundamentale ale limbajului — tipuri de date, structuri de decizie și repetitive, tablouri, subprograme, șiruri de caractere, recursivitate, fișiere și algoritmi clasici de prelucrare. Tocmai acestea sunt subiectele pe care le-ai studiat capitol cu capitol în această carte. Reia exemplele, refă exercițiile de la finalul fiecărui capitol și rezolvă variante din anii trecuți: vei observa că ai deja, în mare parte, tot ce îți trebuie.

Învață construind

Cel mai plăcut mod de a învăța programare este să construiești ceva ce îți place. Scrie un joc simplu — „ghicește numărul", X și 0, un mic joc de tip șarpe în consolă. Fă o aplicație utilă pentru tine: un program care îți organizează temele, un calculator de medii, un mic catalog de cărți sau filme. Automatizează o sarcină plictisitoare: redenumirea unor fișiere, ordonarea unei liste, calcule repetitive. Fiecare proiect personal te învață mai mult decât zece exerciții făcute fără chef, pentru că rezolvi o problemă care chiar contează pentru tine.

Încotro mergi mai departe

Drumul nu se oprește la ultima pagină a acestei cărți. În C++ te așteaptă teme mai avansate: programarea pe obiecte la nivel profund, șabloanele (template-uri), biblioteca standard de containere și algoritmi (STL), gestionarea memoriei și performanța. Dincolo de C++, vei descoperi că un al doilea limbaj se învață mult mai ușor — Python, de pildă, îți va părea aproape relaxant și îți deschide ușa către prelucrarea datelor, inteligența artificială și dezvoltarea web. Iar dacă te atrage competiția, programarea competitivă și structurile de date avansate îți vor oferi provocări pe măsură. Oricare ar fi direcția — dezvoltare de software, jocuri, securitate, robotică — fundamentele pe care le-ai pus aici rămân solide.

Unde cauți răspunsuri

Niciun programator, oricât de experimentat, nu ține totul minte. Arta nu stă în a memora, ci în a ști unde să cauți. Pentru detaliile exacte ale limbajului, cppreference.com este referința pe care o vei deschide cel mai des. Citește documentația oficială atunci când vrei să înțelegi cum funcționează cu adevărat o funcție sau o bibliotecă. Iar când te blochezi, alătură-te comunităților — forumuri, grupuri ale olimpicilor, secțiunile de discuții ale platformelor de antrenament: vei descoperi că aproape orice problemă a mai fost întâlnită de cineva, iar oamenii sunt, de obicei, bucuroși să te ajute.

Închizi această carte, dar nu termini de învățat — abia acum începe partea cu adevărat interesantă. Ai în mâini un limbaj puternic și, mai important, un mod de a gândi care îți va folosi toată viața. Scrie cod, greșește, repară, întreabă, construiește. Lumea are nevoie de oameni care știu să transforme idei în programe. Succes — și spor la compilat!

ANEXĂ

Glosar de cuvinte-cheie C++

Fiecare cuvânt-cheie și noțiune importantă din carte, explicat pe scurt.

Anexă — Glosar de cuvinte-cheie C++

`int` *tip întreg*

Tip de date pentru numere întregi (ex. -3, 0, 42).

Apare în: peste tot

`cout` *flux de ieșire*

Obiectul care afișează date pe ecran, folosit cu <<.

Apare în: cap. 2, 5

`cin` *flux de intrare*

Obiectul care citește date de la tastatură, folosit cu >>.

Apare în: cap. 5

`if` *instrucțiune de decizie*

Execută un bloc doar dacă o condiție este adevărată.

Apare în: cap. 6

`for` *ciclu cu contor*

Repetă un bloc de un număr cunoscut de ori.

Apare în: cap. 7

`algoritm` *algoritm*

succesiune finită de pași clari care duce de la datele de intrare la rezultatul dorit

Apare în: Cap. 1 — Algoritmi și pseudocod

`pseudocod` *pseudocod*

mod de a descrie un algoritm într-un limbaj de mijloc, între limba vorbită și limbajul de programare

Apare în: Cap. 1 — Algoritmi și pseudocod

`schema_logica` *schemă logică*

reprezentare grafică a unui algoritm prin blocuri (start/stop, prelucrare, decizie) legate prin săgeți

Apare în: Cap. 1 — Algoritmi și pseudocod

`main` — *funcția main*

Funcția de la care pornește execuția oricărui program C++; corpul ei stă între acolade.

Apare în: Primul program C++

`include` — *#include*

Directivă care adaugă în program o bibliotecă, de exemplu <iostream> pentru intrare/ieșire.

Apare în: Primul program C++

`comentariu` — *comentariu*

Text ignorat de compilator, scris pentru explicații; // pentru o linie, /* */ pentru mai multe.

Apare în: Primul program C++

`double` — *zecimal*

Tip pentru numere cu virgula, cu precizie buna.

Apare în: Tipurile fundamentale

`char` — *caracter*

Tip pentru un singur caracter, scris intre apostrofuri.

Apare în: Literali: valorile scrise direct

`bool` — *logic*

Tip care retine doar true (adevarat) sau false (fals).

Apare în: Tipurile fundamentale

`%` — *modulo*

Operatorul rest al împărțirii întregi; n % 2 dă paritatea, n % 10 dă ultima cifră.

Apare în: cap. Operatori — Operatorul modulo %

`&&` — *ȘI logic*

Dă true doar dacă ambele condiții sunt true.

Apare în: cap. Operatori — Operatori logici

`||` — *SAU logic*

Dă true dacă măcar una dintre condiții este true.

Apare în: cap. Operatori — Operatori logici

`endl` — *endl*

Manipulator care trece afișarea pe o linie nouă; echivalent cu "\n".

Apare în: Citire și afișare

`setprecision` — *setprecision*

Funcție din <iomanip> care fixează numărul de zecimale afișate; folosită cu fixed.

Apare în: Citire și afișare

`switch` — *instrucțiunea switch*

Alege un caz dintr-o listă comparând o valoare întreagă sau char; folosește case, break și default.

Apare în: Capitolul „Instrucțiuni de decizie”.

`?:` — *operatorul ternar*

Expresia conditie ? a : b returnează a dacă e adevărat, altfel b; un if-else comprimat.

Apare în: Capitolul „Instrucțiuni de decizie”.

`while` — *ciclul while*

Ciclu care verifică o condiție la început și repetă corpul cât timp aceasta este adevărată.

Apare în: Instrucțiuni repetitive

`break` — *break și continue*

break oprește imediat ciclul; continue sare peste restul corpului și trece la repetarea următoare.

Apare în: Instrucțiuni repetitive

`cmmdc` — *cmmdc*

Cel mai mare divizor comun al două numere; se calculează eficient cu algoritmul lui Euclid.

Apare în: Cel mai mare divizor comun (cmmdc)

prim — *număr prim*

Număr natural ≥ 2 care are exact doi divizori: 1 și el însuși; se testează verificând divizorii până la radical.

Apare în: Testul de primalitate

ciur — *ciurul lui Eratostene*

Metodă care găsește toate numerele prime până la o limită, tăind multiplii fiecărui prim găsit.

Apare în: Ciurul lui Eratostene

tablou — *vector*

Sir de valori de acelasi tip, sub un singur nume, numerotate prin indici.

Apare în: Declararea unui tablou

indice — *pozitie*

Numarul de ordine al unui element; primul are indicele 0.

Apare în: Indexarea începe de la 0

parcurgere — *traversare*

Trecerea cu un for prin toate elementele tabloului, de la 0 la n-1.

Apare în: Citirea unui vector de n elemente

matrice — *tablou 2D*

Tabel de valori organizat pe linii si coloane, accesat cu doi indici: a[i][j].

Apare în: Declararea unei matrice

diagonala — *diagonale*

Principala are i == j; secundara are i + j == n-1, doar la matrice patratica.

Apare în: Matrice pătratică și diagonalele

string — *std::string*

Tip din <string> pentru șiruri de caractere care își gestionează singur memoria; oferă +, ==, .length(), s[i], .substr(), .find().

Apare în: Șiruri de caractere

getline — *getline*

Funcție care citește o linie întreagă (cu spații) până la Enter; getline(cin, s) pentru string, cin.getline(s, n) pentru char[].

Apare în: Șiruri de caractere

strlen — *strlen*

Funcție din <cstring> care întoarce lungimea unui șir clasic char[], numărând caracterele până la '\0'.

Apare în: Șiruri de caractere

cautare binara — *căutarea binară*

Caută o valoare într-un vector SORTAT, înjumătățind la fiecare pas zona de căutare; foarte rapidă, O(log n).

Apare în: Capitolul „Căutare și sortare”.

swap — *schimbarea a două elemente*

Interschimbarea valorilor a două variabile folosind o variabilă temporară; pas de bază în orice sortare.

Apare în: Capitolul „Căutare și sortare”.

bubble sort — *metoda bulelor*

Sortare care compară perechi vecine și le schimbă dacă sunt în ordine greșită, repetat până la ordonare; este O(n²).

Apare în: Capitolul „Căutare și sortare”.

return — *return*

Instrucțiunea care încheie o funcție și trimite înapoi o valoare celui care a apelat-o.

Apare în: Subprograme (funcții)

void — *void*

Tip returnat care arată că funcția nu întoarce nicio valoare (o procedură).

Apare în: Subprograme (funcții)

`&` *referință*

Simbolul pus după tipul unui parametru pentru a-l transmite prin referință: funcția lucrează pe variabila originală, nu pe o copie.

Apare în: Subprograme (funcții)

`caz_baza` *caz de bază*

Situația cea mai simplă a unei funcții recursive, rezolvată direct, fără autoapel; ea oprește recursivitatea.

Apare în: ch14 — Recursivitate

`stiva_apeluri` *stiva apelurilor*

Zona de memorie unde se stivuiesc apelurile de funcții aflate în curs; recursivitatea infinită o umple și provoacă stack overflow.

Apare în: ch14 — Recursivitate

`backtracking` *backtracking*

metodă de a construi toate soluțiile pas cu pas, încercând candidați valizi și revenind când o cale nu duce la o soluție

Apare în: Metoda backtracking

`revenire` *revenire (backtrack)*

anularea ultimei alegeri (demarcarea ei) pentru a încerca un alt candidat pe aceeași poziție

Apare în: schema generală back(k)

`divide et impera` *divide et impera*

Strategie în trei pași — împarte problema în subprobleme de același tip, rezolvă-le recursiv până la cazuri simple, combină rezultatele.

Apare în: Capitolul „Divide et impera”.

`merge sort` *sortarea prin interclasare*

Sortare divide et impera: împarte vectorul în două, sortează recursiv fiecare jumătate, apoi le interclasează; este O(n·log n).

Apare în: Capitolul „Divide et impera”.

---

pointer — *pointer*

Variabilă care reține adresa altei variabile; *p dă valoarea de la acea adresă.

Apare în: Pointeri și memorie dinamică

new — *new*

Operator care alocă memorie dinamic la rulare și întoarce adresa ei.

Apare în: Pointeri și memorie dinamică

delete — *delete*

Operator care eliberează memoria alocată cu new (delete[] pentru tablouri).

Apare în: Pointeri și memorie dinamică

struct — *structura*

Tip de date propriu care grupeaza mai multe campuri legate sub un singur nume.

Apare în: Definirea unui struct

camp — *membru*

O rubrica din interiorul unei structuri; se acceseaza cu operatorul punct (.).

Apare în: Declararea unei variabile și accesul cu .

stiva — *stivă (LIFO)*

Structură liniară în care ultimul element adăugat (push) este primul scos (pop) — Last In, First Out.

Apare în: cap. Liste, stive și cozi

coada — *coadă (FIFO)*

Structură liniară în care primul element adăugat (enqueue) este primul scos (dequeue) — First In, First Out.

Apare în: cap. Liste, stive și cozi

lista_inlantuita — *listă înlănțuită*

Șir de noduri în care fiecare nod reține o informație și un pointer (urm) către nodul următor; ultimul are urm = nullptr.

Apare în: cap. Liste, stive și cozi

vector — *std::vector*

Tablou care își poate schimba dimensiunea; adaugi elemente cu push_back, afli câte are cu size() și le accesezi cu [i].

Apare în: Capitolul „Introducere în STL”.

sort — *algoritmul sort()*

Funcție STL care ordonează un container; implicit crescător, descrescător cu greater<>() sau cu o funcție lambda.

Apare în: Capitolul „Introducere în STL”.

map — *std::map*

Dicționar care asociază fiecărei chei unice o valoare; util pentru frecvențe; cheile sunt păstrate ordonate.

Apare în: Capitolul „Introducere în STL”.

set — *std::set*

Mulțime de elemente unice, păstrate ordonate crescător; ideală pentru a elimina duplicatele.

Apare în: Capitolul „Introducere în STL”.

notatia O — *notația O mare*

Mod de a descrie ordinul de creștere al timpului unui algoritm în funcție de n, ignorând constantele și termenii mici; ex. O(n), O(log n), O(n^2).

Apare în: Capitolul „Complexitatea algoritmilor”.

complexitate — *complexitatea unui algoritm*

Estimarea numărului de operații executate de un algoritm în funcție de dimensiunea datelor n; arată cât de bine „scalează” algoritmul.

Apare în: Capitolul „Complexitatea algoritmilor”.

graf — *graf*

Structură formată din noduri (vârfuri) și muchii care leagă perechi de noduri; modelează rețele, hărți, relații.

Apare în: cap. Introducere în grafuri

bfs — *BFS*

Parcurgere în lățime: vizitează nodurile în valuri, folosind o coadă; util pentru drumuri minime în pași.

Apare în: cap. Introducere în grafuri

dfs — *DFS*

Parcurgere în adâncime: merge cât poate pe un drum apoi revine; se implementează recursiv cu vector vizitat[].

Apare în: cap. Introducere în grafuri

gcc — *g++ (GCC)*

Compilatorul de C++ care traduce codul-sursă în executabil; se cheamă din linia de comandă.

Apare în: Compilare cu GCC, Git și Docker pe Windows

git — *Git*

Sistem de versionare care păstrează istoricul codului și permite colaborarea.

Apare în: Compilare cu GCC, Git și Docker pe Windows

commit — *commit*

O fotografie salvată a proiectului, însoțită de un mesaj; unitatea de bază a istoricului Git.

Apare în: Compilare cu GCC, Git și Docker pe Windows

docker-imagine — *imagine (Docker)*

Șablon înghețat cu sistem, unelte și cod, din care se pornesc containere.

Apare în: Compilare cu GCC, Git și Docker pe Windows

docker-container — *container (Docker)*

Instanță vie, în execuție, pornită dintr-o imagine Docker.

Apare în: Compilare cu GCC, Git și Docker pe Windows