SILVIA GRECU
LUCIA MIRON
MIRELA ȚIBU

BACALAUREAT INFORMATICĂ
LIMBAJUL C++

Ghid complet de pregătire a examenului de Bacalaureat


```
{ Matematică-Informatică
  Științe ale Naturii
}
```

Editura Paralela 45

Lucrarea este realizată în conformitate cu programa examenului național de Bacalaureat la disciplina Informatică.

Redactare: Iuliana Voicu
Tehnoredactare: Mariana Dumitru
Pregătire de tipar: Marius Badea
Design copertă: Mirona Pintilie

Descrierea CIP a Bibliotecii Naționale a României
GRECU, SILVIA
Bacalaureat - Informatică : limbajul C++ : ghid complet de pregătire a examenului de Bacalaureat : specializările Matematică-Informatică, Științe ale Naturii / Silvia Grecu, Lucia Miron, Mirela Țibu. - Pitești : Paralela 45, 2019
Conține bibliografie
ISBN 978-973-47-3115-2
I. Miron, Lucia
II. Țibu, Mirela-Anca
004

COMENZI – CARTEA PRIN POȘTĂ
EDITURA PARALELA 45
Bulevardul Republicii, nr. 148, Clădirea C1, etaj 4, Pitești,
jud. Argeș, cod 110177
tel.: 0248 633 130; 0753 040 444; 0721 247 918
tel./fax: 0248 214 533; 0248 631 439; 0248 631 492
e-mail: comenzi@edituraparalela45.ro

www.edituraparalela45.ro

Tiparul executat la tipografia Editurii Paralela 45
E-mail: tipografie@edituraparalela45.ro

Cuvânt-înainte

Lucrarea de față se dorește a fi un ghid de pregătire individuală a elevilor de liceu pentru proba de Informatică a examenului de Bacalaureat – varianta C++. În acest sens, cele unsprezece capitole de teorie prezintă un breviar al noțiunilor studiate la disciplina Informatică, prezente în programa de Bacalaureat, urmat de modele de teste propuse prin a căror rezolvare se poate realiza aprofundarea acestor noțiuni.

După parcurgerea noțiunilor de teorie, cartea se continuă cu teze întocmite după modelul examenului de Bacalaureat, atât pentru specializarea Matematică-Informatică, cât și pentru specializarea Științe ale Naturii.

Pentru a veni în sprijinul celor care vor folosi această carte, problemele și tezele au fost rezolvate integral, iar programele au fost verificate în Code::Blocks.

Ghidul poate fi utilizat în aceeași măsură și ca auxiliar didactic la orele de Informatică.

Sperăm că acest ghid se va dovedi un instrument util elevilor, dar și profesorilor care îl folosesc în pregătirea examenului de Bacalaureat.

Autoarele

CAPITOLUL 1

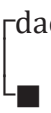
Reprezentarea algoritmilor în pseudocod. Elemente de bază ale limbajului C++

Teorie

Principiul programării structurate: Orice algoritm poate fi descris utilizând trei tipuri de structuri de control: liniară, alternativă și repetitivă.

Structuri de control – reprezentare în pseudocod	Implementare C++
• **Structura liniară** – conține operații de citire/scriere/atribuire. citește var_1, var_2, ..., var_n var_1, var_2, ... sunt identificatori de variabile scrie $expresie_1$, $expresie_2$, ..., $expresie_n$ $expresie_1$, $expresie_2$, var_2, ... sunt expresii de orice tip (numeric, caracter sau șir de caractere) *variabila* ← *expresie* Se evaluează expresia din partea dreaptă, iar valoarea obținută i se atribuie variabilei cu identificatorul *variabila.*	Citirea valorilor variabilelor de la tastatură: cin ≫ var_1 ≫ var_2 ≫ ⋯ ≫ var_n; //valorile variabilelor se citesc în ordinea din listă Afișarea valorilor expresiilor pe ecran: cout ≪ $expresie_1$ ≪ ⋯ ≪ $expresie_n$; //expresiile se vor evalua și afișa în ordinea din listă variabila = expresie; variabila < *op* > = expresie; unde *op* ∈ {+, −, ∗, /, %, ≫, ≪ , &, |, ^}, este echivalentă cu *variabila = variabila <op> expresie;*
Structura alternativă – conține operații de decizie: dacă <*expresie*> atunci *secvența_A* altfel *secvența_B* ■	Instrucțiunile *if* și *switch* implementează structura alternativă. if (*expresie*) instrucțiuni_A else instrucțiuni_B **Efect:** Se evaluează *expresie*. Dacă valoarea este diferită de 0, se execută secvența de

Sau



Efect: Se evaluează *expresie*. Dacă este adevărată, se execută *secvența_A*, altfel se execută *secvența_B* și se continuă algoritmul cu următoarele structuri.

Secvențele A și/sau B pot conține orice alte structuri liniare, alternative sau repetitive.

Ramura altfel poate lipsi.

instrucțiuni_A, altfel se execută secvența de instrucțiuni_B.
În cazul în care o secvență conține mai mult de o instrucțiune, se va delimita prin { } grupul de instrucțiuni care definesc secvența.

Instrucțiunea *switch* pentru alternative multiple, simplifică modul de scriere pentru if-urile imbricate.

```
switch (expresie) {
    case constantă_1: instrucţiuni_1
                      break;
    case constantă_2: instrucţiuni_2
                      break;
    .....
    case constantă_n: instrucţiuni_n
                      break;
    default: instrucţiuni
    }
```

Efect: Valoarea *expresie* este evaluată la un tip întreg, apoi această valoare este comparată cu fiecare constantă. Este rulat blocul de instrucţiuni al valorii găsite.
Dacă este prezentă instrucțiunea *break* după blocul executat, se va părăsi instrucțiunea *switch*, altfel se vor executa în continuare toate blocurile de instrucțiuni ale constantelor care urmează, până la întâlnirea primului *break* sau până la sfârșitul instrucțiunii *switch*.
În caz că numărul nu este egal cu niciuna dintre constante, este executat blocul aflat după *default*.

Structura repetitivă

- **Cu condiție inițială**

Efect:
Pas 1. Se evaluează *expresie*.
Pas 2. Dacă este diferită de 0, se execută *secvența_A* și se reia Pas 1. Dacă este 0 se părăsește structura repetitivă și se continuă algoritmul cu următoarele structuri.

Instrucțiunea **while** implementează structura repetitivă cu condiție inițială.

```
while (expresie)
{
        //declarări de date locale
        Instrucţiuni
}
```

Variabilele declarate în blocul instrucțiunii *while* sunt vizibile doar în acest bloc și sunt valabile doar pe durata execuției instrucțiunii.

Obs. 1) Dacă la prima evaluare, valoarea *expresiei* este 0, *secvența_A* nu se execută deloc.

2) În *secventa_A* trebuie să se modifice valoarea cel puțin a unei variabile care apare în *expresie* astfel încât, după un număr finit de pași, execuția structurii *cât timp* să se încheie.

- **Cu condiție finală**

repetă
    secvența_A
■ până când *<expresie>*

Efect:
Pas 1. Se execută *secvența_A*.
Pas 2. Se evaluează *expresie*.
Pas 3. Dacă valoarea este diferită de 0 (este adevărată), se părăsește structura repetitivă și se continuă algoritmul cu următoarele structuri. Dacă valoarea este egală cu 0, se reia Pasul 1.
Obs. 1) *secvența_A* se execută cel puțin o dată, indiferent de valoarea inițială a *expresiei*.

2) În *secvența_A* trebuie să se modifice valoarea cel puțin a unei variabile care apare în *expresie* astfel încât, după un număr finit de pași, execuția structurii *cât timp* să se încheie.

- **Cu număr cunoscut de pași**

pentru contor ← start, stop, pas execută
    secvența_A
■

Obs. Dacă *pas* nu este precizat, implicit va avea valoarea 1.
Efect:
Pas 1. Se atribuie variabilei *contor* valoarea de start (evaluată ca număr întreg)
Pas 2. Se compară valoarea *contor* cu valoarea stop, în funcție de semnul pasului. Dacă pasul este 1 (parcurgere crescătoare) atunci compararea va fi *contor≤stop*, iar dacă pasul este –1 (parcurgere descrescătoare), atunci compararea va fi *contor≥stop*

Instrucțiuni pot fi orice instrucțiuni C++, inclusiv dintre cele care implementează structuri repetitive.

Instrucțiunea **do-while** implementează structura repetitivă cu condiție finală.

```
do {
        //declarări de date locale
        Instrucțiuni
} while (not expresie);
```

Atenție! Condiția de oprire a buclei, exprimată de *expresie* din descrierea în pseudocod, se transformă în condiție de repetare a buclei, în instrucțiunea din C++. De aceea apare negația (not *expresie*).

Variabilele declarate în blocul instrucțiunii *do-while* sunt vizibile doar în acest bloc și sunt valabile doar pe durata execuției instrucțiunii.

Instrucțiuni pot fi orice instrucțiuni C++, inclusiv dintre cele care implementează structuri repetitive.

Instrucțiunea *for* implementează structura repetitivă cu număr cunoscut de pași *pentru*.

Dacă *pas* este 1

```
for (contor=start; contor<=stop;
contor++)
{
    Instrucțiuni
}
```

CAPITOLUL 2

Algoritmi elementari

Teorie

I. Algoritmi care prelucrează cifrele unui număr

- spargerea în cifre a numărului *n* și prelucrarea cifrelor de la dreapta la stânga:

```
dacă n=0 atunci
    prelucrare_cifra(0)
■
cât timp n ≠ 0 execută
    cif ← n%10
    prelucrare_cifră(cif)
    n ← [n/10]
■
```

```
repetă
    cif ← n%10
    prelucrare_cifră (cif)
    n ← [n/10]
■ până când n=0
```

- spargerea în cifre a numărului *n* și prelucrarea cifrelor de la stânga la dreapta:

```
p ← 1
cât timp p * 10 ≤ n atunci
    p ← p * 10
■
cât timp p ≠ 0 execută
    cif ← [n/p]
    prelucrare_cifră(cif)
    n ← [n%p]
    p ← [p/10]
■
```

Formarea numărului cu cifrele din *n* de pe poziții impare, considerând numerotarea de la stânga la dreapta, prima cifră fiind considerată pe poziția 1.

```
p ← 1
cât timp p * 10 ≤ n atunci
    p ← p * 10
■
nrNou ← 0
cât timp p ≠ 0 execută
    cif ← [n/p]
    nrNou← nrNou * 10 + cif
    n ← [n%(p*10)]
    p ← [p/100]
■
```

prelucrare_cifră(cif) – descrie operațiile specifice problemei în care apare această prelucrare. De exemplu: suma cifrelor/numărarea cifrelor cu o anumită proprietate, verificarea proprietății de palindrom, eliminarea/inserarea cifrelor cu o anumită proprietate etc.

Atenție! Spargerea în cifre a unui număr distruge valoarea inițială a acestuia! Este necesară crearea unei copii a lui *n*, anterior spargerii, pentru a putea procesa ulterior valoarea inițială.

Exemple

Construirea oglinditului și verificare **n** *dacă este palindrom*	*Media aritmetică a cifrelor nenule*	*Cifra maximă din* **n**
copie ← n oglindit ← 0 cât timp *n ≠ 0* execută cif ← n%10 oglindit ← oglindit * 10 + cif n ← [n/10] ■ dacă *copie=oglindit* atunci prelucrare_cifră(palindrom) ■	sum ← 0; nrcif ← 0; medie ← 0; cât timp *n ≠ 0* execută cif ← n%10 dacă *cif>0* atunci sum ← sum + cif nrcif ← nrcif + 1 ■ n ← [n/10] ■ dacă *nrcif>0* atunci medie ← sum/nrcif ■ prelucrare(medie)	cifmax ← 0 cât timp *n ≠ 0* execută cif ← n%10 dacă *cif>cifmax* atunci cifmax ← cif ■ n ← [n/10] ■ prelucrare(cifmax)

Eliminarea cifrelor impare din **n**	*Dublarea aparițiilor cifrelor pare din* **n**	*Numărarea cifrelor pare existente în* **n**
nrNou ← 0 p ← 1 cât timp *n ≠ 0* execută cif ← n%10 dacă *cif % 2 = 0* atunci nrNou ← nrNou + cif *p p ← p * 10 ■ n ← [n/10] ■ prelucrare(nrNou)	nrNou ← 0 p ← 1 cât timp *n ≠ 0* execută cif ← n%10 dacă *cif % 2 = 0* atunci nrNou ← nrNou + cif *p p ← p * 10 ■ nrNou ← nrNou + cif *p p ← p * 10 n ← [n/10] ■ prelucrare(nrNou)	nrPare ← 0 repetă cif ← n%10 dacă *cif % 2 = 0* atunci nrPare ← nrPare + 1 ■ n ← [n/10] ■ până când *n=0* prelucrare(nrPare)

Cifra de control a unui număr *n* se obține calculând suma cifrelor lui *n* apoi repetând procesul cu cifrele sumei obținute anterior până când se obține un număr format dintr-o singură cifră, numită cifră de control. De exemplu, pentru n=7912 se obțin pe rând sumele 7 + 9 + 1 + 2 = 19, 1 + 9 = 10, 1 + 0 = 1, iar 1 este cifra de control a lui 7912.

```
cât timp n > 9 execută
    sumcif ← 0
    cât timp n ≠ 0 atunci
        cif ← n % 10
        sumcif ← sumcif + cif
        n ← [n/10]
    n ← sumcif
prelucrare(n)
```

Algoritmul eficient ca timp de execuție se bazează pe observația că cifra de control aunui număr este periodică și respectă relația:

$$\text{cifControl}(n) = \begin{cases} 0, \text{ dacă } n=0 \\ 9, \text{ dacă n\%9=0 și } n \neq 0 \\ n\%9, \text{ altfel} \end{cases}$$

II. Divizibilitate. Algoritmi care prelucrează divizorii proprii/improprii/primi ai unui număr

Fie *n* număr natural. Definim mulțimile:

Divizori **proprii**: d ∈ {2,3,4,...,$[\frac{n}{2}]$}, $n \vdots d$ Divizorii **primi**: d ∈ {2,3,5,7 ... $[\sqrt{n}]$}, $n \vdots d$

Divizorii **improprii**: d ∈ {1,n}

Divizorii proprii ai lui ***n***

```
p ← 1
pentru d←2,[n/2],1 execută
    dacă n % d = 0 atunci
        prelucrare (d)
```

Divizorii proprii ai lui ***n*** *– optimizat*

```
p ← 1
pentru d←2,[√n]– 1,1 execută
    dacă n % d = 0 atunci
        prelucrare (d)
        prelucrare (n/d)
dacă d * d = n atunci
    prelucrare (d)
```

Divizorii primi

```
d ← 2
cât timp n > 1 execută
    p ← 0
    cât timp n% d = 0 execută
        n ← [n/d]
        p ← p + 1
    dacă p ≠0 atunci
        prelucrare (d, p)
    d ← d+1
```

Divizorii primi ai lui ***n*** *– optimizat*

```
d ← 2
cât timp d*d ≤ n execută
    p ← 0
    cât timp n% d = 0 execută
        n ← [n/d]
        p ← p + 1
    dacă p ≠ 0 atunci
        prelucrare (d, p)
    d ← d+1
dacă n ≠ 1 atunci
    prelucrare (n, 1)
```

CUPRINS