Cosmina Corcheș
Marius Misaroș
Alexandru Ciobotaru
Liviu Cristian Miclea

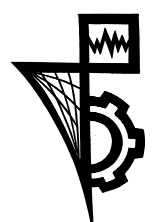
Fundamentele programării în C: de la bazele teoretice la implementare

UTPRESS
Cluj-Napoca, 2023
ISBN 978-606-737-682-1

Cosmina Corcheș
Marius Misaroş
Alexandru Ciobotaru
Liviu Cristian Miclea

Fundamentele programării în C: de la bazele teoretice la implementare

UTPRESS
Cluj-Napoca, 2023
ISBN 978-606-737-682-1



Editura UTPRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director: ing. Dan COLȚEA

Recenzia: Conf.dr.ing. Enyedi Szilárd
Conf.dr.ing. Stan Ovidiu Petru

Pregătire format electronic on-line: Gabriela Groza

Copyright © 2023 Editura UTPRESS
Reproducerea integrală sau parţială a textului sau ilustraţiilor din această carte este posibilă numai cu acordul prealabil scris al editurii UTPRESS.

ISBN 978-606-737-682-1

Cuprins

Capitolul 1 – Noțiuni fundamentale..1

1.1. Noțiuni privind reprezentarea informației în calculator 2

1.2. Exemplu reprezentarea informației în calculator 7

1.3. Scheme logice și pseudocod 9

1.3.1 Scheme logice 9

1.3.2 Pseudocod...11

1.4. Exemple de scheme logice și pseudocod 12

1.4.1 Exemple de scheme logice 12

1.4.2 Exemple pseudocod 13

1.5. Exerciții...14

Capitolul 2 – Programarea aplicațiilor simple în C 15

2.1. Aspecte generale 15

2.2. Medii de programare 16

2.3. Dezvoltarea aplicațiilor tip Consolă în Visual Studio 2022 17

2.4. Exemplu de scriere a unor programe simple în C /C++ 23

2.5. Utilizarea debugger-ului în Visual Studio 2022..25

2.5.1 Adăugare/Ștergere breakpoint 25

2.5.2 Rularea pas cu pas a programului...27

2.5.3 Watch window 27

2.6. Exemplu utilizare utilizarea debugger-ului în Visual Studio 2022...............28

2.7. Exerciții...30

Capitolul 3 – Variabile și tipuri de date fundamentale 31

3.1. Noțiunea de variabilă 31

3.2. Tipuri de date elementare 32

3.3. Operatorii limbajului C (C++) 33

3.3.1 Operatori matematici 34

3.3.2 Operatorii relaționali 34

3.3.3 Operatori logici 34
3.3.4 Operatori pe biți 35
3.3.5 Operatorii de incrementare și de decrementare 35
3.3.6 Operatorii și expresiile de atribuire 37
3.4. Conversiile de tip (cast sau transtipaj) 38
3.5. Exerciții 40
Capitolul 4 – Directive preprocesor 42
4.1. Directivele #define și #undef 43
4.2. Directiva #include 45
4.3. Directivele #if , #ifdef și #ifndef 46
4.4. Directiva #error 49
4.5. Directiva #pragma 50
4.6. Exerciții utilizare directive preprocesor 51
Capitolul 5 – Instrucțiuni de decizie 52
5.1. Instrucțiunea if 52
5.2. Instrucțiunea switch 53
5.3. Exerciții 54
Capitolul 6 – Instrucțiuni de ciclare (repetitive) 56
6.1. Instrucțiunea for 56
6.2. Instrucțiunea while 59
6.3. Instrucțiunea do-while 62
6.4. Exemple suplimentare instrucțiuni de ciclare 63
6.5. Exerciții 66
Capitolul 7 – Tablouri de elemente și pointeri 68
7.1. Noțiuni introductive tablouri (arrays) și șiruri de caractere 68
7.2. Concepte de bază legate de pointeri 71
7.3. Exemple tablouri de elemente și pointeri 72
7.4. Exerciții 75

Capitolul 8 - Structura programelor mari 77
8.1. Lucrul cu Fișiere și Operații I/O în C/C++..77
8.2. Operații de intrare-ieșire în C/C++ 78
8.2.1 Citirea dintr-un fișier înregistrat pe un hard-disk 79
8.3. Exemple lucru cu fișiere 80
8.4. Funcții din biblioteca limbajului C..81
8.5. Crearea și utilizarea propriilor funcții..84
8.6. Conceptul de recursivitate...89
8.7. Exerciții..90
Capitolul 9 - Introducere în algoritmi fundamentali 94
9.1. Algoritmi de inițializare 95
9.2. Algoritmul “Divide et Impera” 97
9.3. Algoritmi de sortare – Metoda ”Bubble Sort” 101
9.4. Algoritmul de interclasare...104
9.5. Algoritmi de calcul cu mulțimi 106
Bibliografie..110

Capitolul 1

Noțiuni fundamentale

Dezvoltarea tehnologiei are la bază două ramuri principale: hardware și software. Elementele hardware a unui sistem informatic fac referire mai exact la componentele fizice ale unui sistem. Toate aceste componente electronice sau mecanice sunt interconectate pentru a putea a facilita funcționarea dispozitivului [1].

În Figura 1-1. Schema bloc a unei plăci de bază. este exemplificată arhitectura unei plăci de bază folosită în diferite dispozitive pentru a asigura funcționarea unui sistem: circuit de control, circuit de date, circuit de adrese, memorie RAM, microprocesor, port pentru intrări, port pentru ieșiri și circuite auxiliare, [2]. Sistemul software sau numit și sistemul de calcul se referă la programele care rulează pe un dispozitiv și îndeplinesc anumite condiții.

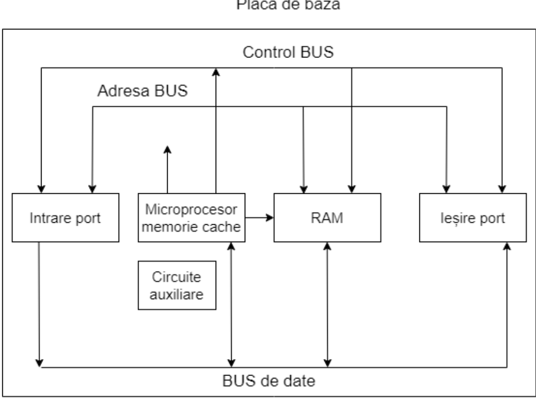


Figura 1-1. Schema bloc a unei plăci de bază.

Acesta este format dintr-un ansamblu de date, programe și instrucțiuni care funcționează împreună. Un sistem software clasic

este alcătuit din următoarele componente principale: Sistemul de operare, software-ul de aplicație, drivere de dispozitiv, utilitare de sistem, biblioteci software, interfața cu utilizatorul, servicii de rețea și sisteme de gestionare a bazelor de date.

1.1. Noțiuni privind reprezentarea informației în calculator

Reprezentarea datelor în calculator este un concept fundamental, care se referă la modul în care datele utilizate se pot reprezenta în mod digital [3], [4].

Datele și instrucțiunile nu pot fi introduse direct, în calculator, folosind limbajul uman. Astfel toate tipurile de date (e.g., numere, caracter, imagini, video, audio etc.) necesită convertirea lor într-o formă ce poate fi interpretată de mașină, mai exact într-o formă binară. Datorită acestui aspect este necesar să înțelegem modul în care un calculator comunică cu dispozitivele periferice prin intermediul circuitelor electronice, mediilor magnetice și prin dispozitivele optice [4].

Circuitele digitale sunt caracterizate de următoarele două valori logice: “1” logic și „0” logic. Pentru a înțelege principiul de reprezentare “1” logic și “0” logic, se poate face o analogie cu funcționarea circuitului circuit electric din Figura 1-2.

În momentul în care întrerupătorul este închis (“1” logic) led-ul se va aprinde, iar când întrerupătorul este deschis („0” logic) led-ul se va stinge. Similar, reprezentarea binară (doar două valori) este folosită pentru a reprezenta datele (starea biților) stocate în memoria calculatoarelor.

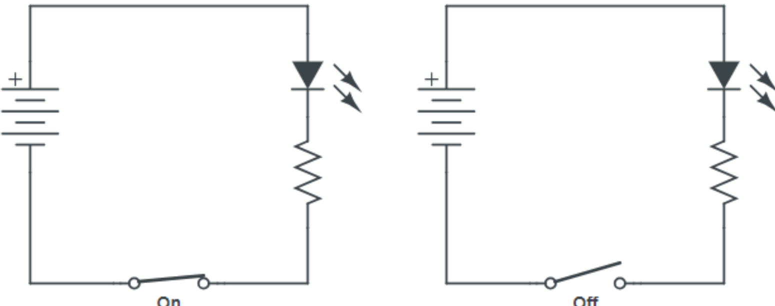


Figura 1-2. Analogie între principiul de funcționare a circuitelor digitale și a circuitelor electrice

Mediile magnetice sunt folosite pentru unitățile de stocare de tip hard disk. Modul de reprezentare a informației este similară cu cel al circuitelor digitale, doar că modul de citire și scriere se face printr-o abordare total diferită.

Pentru reprezentarea lui „1” logic și a lui „0” logic este utilizat un cap de scriere electromagnetic care polarizează secțiuni minuscule ale hard disk-ului, astfel încât acestea să fie orientate în sus sau în jos (“Nord” sau ”Sud”). Interpretarea codului scris se face prin utilizarea capului de citire care detectează tipul de polarizare și generează un cod binar, Figura 1-3.

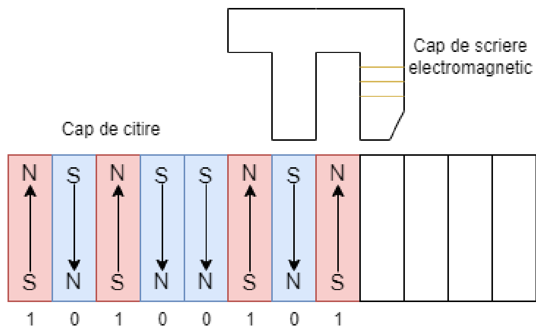


Figura 1-3. Principiul de citire a unui hard disk.

În limbajele de programare (e.g., C, C++, Java, Python) se acceptă utilizarea directă a reprezentărilor binare, acest lucru fiind semnalat prin intermediul prefixului “0b” sau “0B” urmat de codul în

binar [4]–[6]. Precum s-a subliniat în paragrafele anterioare, sistemele de calcul utilizează doar sistemul binar. Cu toate acestea, sistemul binar nu este la îndemâna oamenilor. Sistemul de reprezentare a valorilor numerice utilizat de către oameni este sistemul zecimal care cuprinde 10 cifre distincte, de la 0 la 9. În sistemul binar, dispunem doar de două valori și anume 0 și 1. Spre exemplu, reprezentarea $(111\ 1101)_2$, corespunde în baza 10 reprezentării 125_{10}. Pentru convertirea unei reprezentări din baza 2 în baza 10 este necesară înmulțirea fiecărei valori din reprezentare, cu 2 la puterea corespunzătoare poziției pe care se află, începând cu prima valoare din partea dreapta care corespunde primei poziții și care este numerotată cu 0. Să presupunem că un număr binar are n cifre:

$$B = a_{n-1} \cdot a_{n-2} \cdot \cdots a_2 \cdot a_1 \cdot a_0 \tag{1}$$

corespondentul său în decimal se va scrie sub forma:

$$D = (a_0 \times 2^0) + (a_1 \times 2^1) + (a_2 \times 2^2) + \cdots + (a_{n-1} \times 2^{n-1}) \tag{2}$$

În vederea convertirii unui număr din baza zece în baza doi se face împărțirea continuă a numărului la 2, până în momentul în care se va obține câtul egal cu zero.

$$\left.\begin{array}{l} A \div 2 = B \times R1 \\ B \div 2 = C \times R2 \\ C \div 2 = D \times R3 \\ D \div 2 = E \times R4 \end{array}\right| \tag{3}$$

Realizarea reprezentării în sistemul binar se face prin ordonarea de la stânga la dreapta a restului (Rn) obținut din împărțire, începând cu ultimul rest primit.

$$Y = (R_4 R_3 R_2 R_1) \tag{4}$$

Un octet sau byte este o reprezentare a informației în vederea procesării sau stocării pe un calculator. Acesta este format din 8 biți, fiecare având posibilitatea să fie una din cele doua valori asociate sistemului binar. Prin intermediul unui octet se poate reprezenta un caracter ASCII precum litera A, simbolul pentru virgulă sau simbolurile pentru operatorii matematici [7]. Codul ASCII (American Standard

Code for Information Interchange) standard permite codificarea de până la 2^7 = 128 caractere, mai exact sunt codificate literele din alfabetul englez, cifrele cuprinse între 0 și 9, operatorii semnelor de punctuație și simbolurile. Se poate folosi codul ASCII Extins între 128 și 256 care cuprinde mai multe simboluri grafice și diacritice.

În momentul de față, memoria unui calculator este de ordinul giga-octeților sau tera-octeților, care reprezintă unități de ordinul miliardelor și trilioanelor de octeți. Tabel 1.1 prezintă multiplii ai octetului pentru sistemele de calcul [2], [5], [8].

Tabel 1.1. Multiplii octet

Denumire	Număr de octeți	Putere a lui 2
Octet (Byte), B	1	2^0
Kilo-octet (Kilobyte), KB	$\sim10^3$	2^{10}
Mega-octet (Megabyte), MB	$\sim10^6$	2^{20}
Giga-octet (Gigabyte), GB	$\sim10^9$	2^{30}
Tera-octet (Terabyte), TB	$\sim10^{12}$	2^{40}
Peta-octet (Petabyte), PB	$\sim10^{15}$	2^{50}
Exa-octet (Exabyte), EB	$\sim10^{18}$	2^{60}
Zetta-octet (Zettabyte),ZB	$\sim10^{21}$	2^{70}
Yotta-octet (Yottabyte),YB	$\sim10^{24}$	2^{80}

Dacă pentru memorarea unei informații sunt necesari mai mult de 8 biți, se va folosi o succesiune de mai mulți octeți. Fiecare celulă de memorie (de 1 octet) are și o adresă care permite accesarea ei [9], [10]. La pornirea calculatorului fiecare celulă de memorie din memoria RAM are o valoare arbitrară așa cum este prezentat în Figura 1-4. Adresa unei celule și conținutul ei sunt mărimi care nu au nimic în comun [1], [2].

În timpul funcționării calculatorului, memoria acestuia reține în celulele sale reprezentări binare specifice informației stocate [2], [5], [11]. Se poate preciza de asemenea, că fiecare octet conține un număr întreg între 0 și 255 în baza 10 deoarece 255 este cel mai mare număr

întreg care se poate reprezenta folosind 8 biți. Reprezentarea numărului $(255)_{10}$ în baza 2 este astfel $(1111\ 1111)_2$.

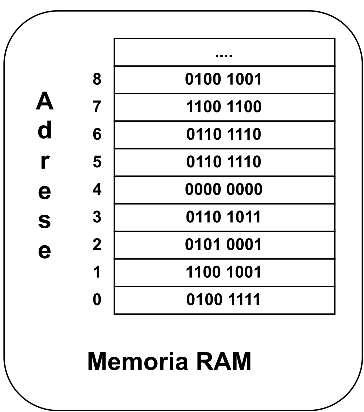


Figura 1-4. Modul de reprezentare a informației în memoria RAM.

Procesorul calculatorului poate accesa o celulă de memorie (un octet) de la o adresă dată pentru a realiza două funcții fundamentale:

- poate scrie în celulă (conținutul anterior al celulei este înlocuit cu o nouă valoare);
- poate citi valoarea din celulă (procesorul obține o copie a conținutului celulei, conținutul celulei rămâne neschimbat).

În timpul executării unui program, o serie de valori din memorie rămân neschimbate în timp ce altele se modifică pe măsură ce execuția programului progresează. De obicei rămân fixe celulele care conțin date de intrare sau instrucțiunile ce urmează a fi executate de către program. La momentul inițial, orice zonă de memorie va fi caracterizată de o valoare arbitrară, reprezentată în binar. Pe parcursul execuției programului, conținutul zonei de memorie se va modifica conform calculelor efectuate, Figura 1-5.

La scrierea unui program într-un limbaj de programare de nivel înalt nu este necesară ținerea evidenței adreselor locațiilor de memorie care conțin date sau informații despre tipul acestora La fel ca și reprezentarea binară, lucrul cu adrese de memorie nu este la îndemâna oamenilor. Tocmai din acest motiv limbajele de programare

de nivel înalt permit asocierea unor denumiri pentru zonele de memorie, denumiri care sunt mai ușor de citit și asociat cu funcționalitatea lor în cadrul programului. Despre acest aspect se va discuta în capitolul următor.

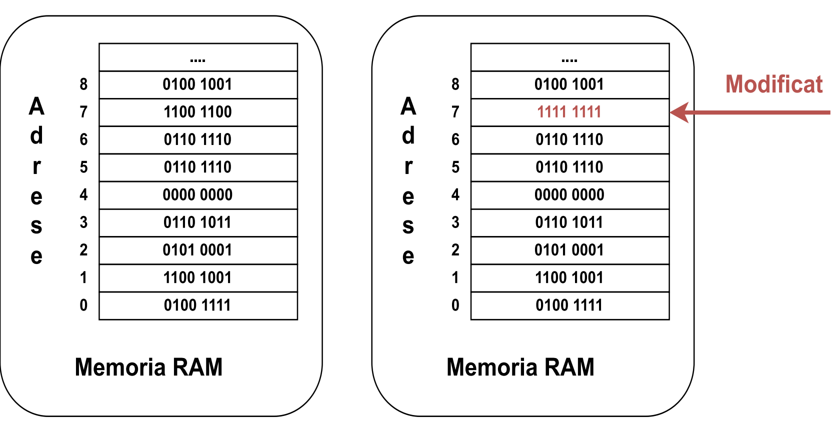


Figura 1-5. Scrierea unei celule de memorie RAM

1.2. Exemplu reprezentarea informației în calculator

1. Reprezentarea numărului întreg „6" se face în calculator folosind sistemul binar. Numărul 6 în zecimal se poate reprezenta în binar folosind minim trei biți. După am menționat în paragraful anterior zonele de memorie sunt caracterizate de un număr de biți. Prin urmare informația se reprezintă pe numărul respectiv. În capitolul 3 se va discuta cum se poate specifica numărul de biți pe care se face reprezentarea datelor. De exemplu, dacă se specifică reprezentarea pe 16 biți, atunci numărul 6 în zecimal va avea următoarea reprezentare binară pe 16 biți: 0000000000000110.

2. În ceea ce privește reprezentarea caracterelor în sistemul binar, se va realiza tot ca succesiuni de 0 și 1. Pentru a simplifica procesul, există o tabelă prin care se corelează caracterul reprezentat de un cod binar pe 7 biți. Tabela în cauză poartă denumirea de tabelă ASCII.

Pentru fiecare caracter din cuvântul „salut” avem în codul ASCII următoarele coduri:

- „s” este reprezentat de 115 în codul ASCII;
- „a” este reprezentat de 97 în codul ASCII;
- „l” este reprezentat de 108 în codul ASCII;
- „u” este reprezentat de 117 în codul ASCII;
- „t” este reprezentat de 116 în codul ASCII.

Primul pas în reprezentarea în cod ASCII a cuvântului "salut" constă în identificarea codurilor numerice asociate caracterelor conform tabelei ASCII. A doua etapă constă în convertirea valorilor numerice în reprezentare binară după cum urmează: 1110011 1100001 1101100 1110101 1110100.

3. Pentru reprezentarea unei imagini trebuie ținut cont de faptul că acestea sunt formate din pixeli reprezentați de biți, fiecare având propria culoare. În majoritatea cazurilor imaginile sunt reprezentate prin model RGB, model care definește culorile pixelilor folosind trei culori roșu, verde și galben.

Modul de reprezentare a acestor culori este realizat prin stocarea intensității acestora în sistemul binar pe 8 biți, având o valoare cuprinsă între 0 $(00000000)_2$ până la 255 $(111111111)_2$ care stabilește intensitatea culorii, 0 reprezentând absența culorii și 255 reprezentând intensitatea maximă.

Spre exemplu pentru obținerea culorii galben se fac următoarele setări: roșu și verde setat la maxim, iar albastru este setat pe valoarea minimă. Similar se întâmplă și pentru stabilirea celorlalte culori necesare realizării imaginii dorite. Automat modul de modificare a unei poze implică modificarea directă și a pixelilor, fapt care provoacă în cele din urmă schimbarea dorită.

1.3. Scheme logice și pseudocod

1.3.1 Scheme logice

Schemele logice vin în ajutorul programatorului sub o formă grafică cu scopul de a sintetiza pașii necesari în rezolvarea unui algoritm, Tabel 1.2. Modalitatea de interpretare a acestora se realizează printr-un set de simboluri standard care etapizează parcurgerea algoritmului în pași clar specificați.

Tabel 1.2 Simbolurile folosite pentru descrierea unui program folosind pseudocod.

Tipologie Simbol	Simbol	Descriere
Linii de flux	→ ←	Indicatori pentru evidențierea sensului de parcurgere.
Simboluri terminale	START	Marchează începutul algoritmului.
	STOP	Marchează sfârșitul algoritmului.
Simboluri de intrare-ieșire	CITIRE	Realizează citirea datelor.
	SCRIERE	Realizează scrierea datelor.

Tipologie Simbol	Simbol	Descriere
Bloc de calcul	VARIABILA=EXPRESIE	Se evaluează expresia și se atribuie rezultatul variabilei.
Bloc logic	DA / CONDIȚIE LOGICĂ / NU	Se evaluează condiția logică: dacă expresia este adevărată secvența se continuă pe calea DA, iar dacă condiția este falsă, secvența se continuă pe calea NU.
Conector		Conectivitatea a doua linii de flux între ele.
Bloc de procedură		Reprezintă o succesiune de calcule similar unui algoritm independent, calculele realizate nu se exemplifică în cadrul schemei.

Aceste tipuri de scheme sunt utilizate de către programatori în vederea structurării pașilor necesari pentru rezolvarea unor cerințe de proiect. Din punct de vedere a structurării și așezării acestor blocuri la realizarea unei scheme logice este permisă amplasarea în lateral a unui bloc, înaintea sau în spatele acestora, dar nu și suprapunerea lor.

Avantajele utilizării schemelor logice [12]:

- vizualizarea structurală;
- identificarea rapidă a erorilor;
- îmbunătățirea colaborării într-o echipă;

- identificarea rapidă a unui fragment din algoritm;
- oferă programatorului posibilitatea de a analiza vizual cerințele de proiect.

1.3.2 Pseudocod

Este modalitatea prin care programatorii au posibilitatea de a descrie printr-o reprezentare de tip text pașii unui algoritm [13]. Se preferă reprezentarea algoritmului prin intermediul pseudocodului deoarece acesta poate să fie interpretat indiferent de limbajul de programare utilizat. Modul de implementare a unei soluții pe baza unui pseudocod trebuie să fie mai degrabă văzut ca o traducere a acesteia în sintaxa de cod a limbajului cerut. Cu toate acestea modul de scriere a pseudocodului trebuie să conțină o descriere completă a logicii algoritmului. În vederea dezvoltării unui pseudocod care să permită o interpretare unitară este necesar respectarea unui set de reguli bine definite:

1. scrierea doar a unei afirmații pe linie pentru delegarea doar a unei acțiuni în lucru;
2. cuvintele cheie trebuie scrise cu majuscule (READ, WRITE, IF, ELSE, ENDIF, WHILE, ENDWHILE, REPEAT, UNTIL);
3. alinierea conformă pentru a păstra ierarhia în parcurgerea etapelor;
4. închiderea corecta a structurilor multilinie;
5. păstrarea expresiilor coincise, simple și lizibile;
6. Scrierea etapelor în funcție de cerințele problemei, nu în funcție de etapele de implementare.

Avantajele pseudocodului:

- poate să fie conceput pentru documentarea codului sau a unei parți specifice din cod;
- flexibilitatea de care beneficiază îi conferă posibilitatea de a putea fi implementat în orice limbaj de programare;

- ușurează munca programatorilor prin explicațiile complete și ample în momentul scrierii codului;
- face parte din cele mai bune tipuri de abordări a unei probleme;
- oferă o colaborare mai bună atunci când este vorba de munca în echipă;
- permite programatorului să pună accent pe logica algoritmului, fără a interveni în problemele de sintaxă a unui limbaj de programare.

1.4. Exemple de scheme logice și pseudocod

1.4.1 Exemple de scheme logice

1. Se cere o schemă logică pentru suma a două cifre (X, Y) introduse de utilizator, iar înainte de afișarea acesteia rezultatul (S) obținut din adunare să fie comparat cu 10, iar în cazul în care suma este mai mică, să se afișeze mesajul „Au fost alese numere mici”, Figura 1-6.

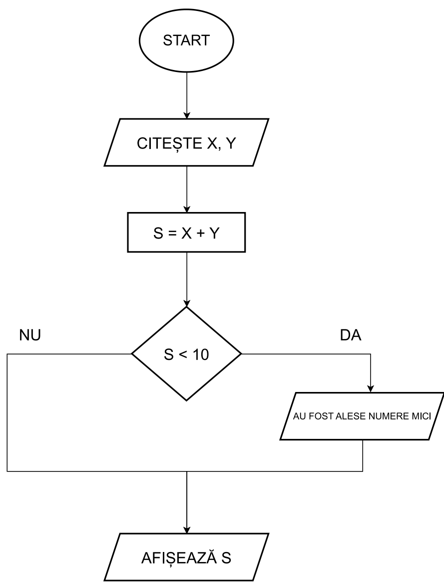


Figura 1-6. Schema logică calcul sumă

2. Se cere o schemă logică a unui program care determină soluțiile reale ale unei ecuații de gradul doi cu coeficienții reali a, b și c introdusi de la tastatură de către utilizator, Figura 1-7.

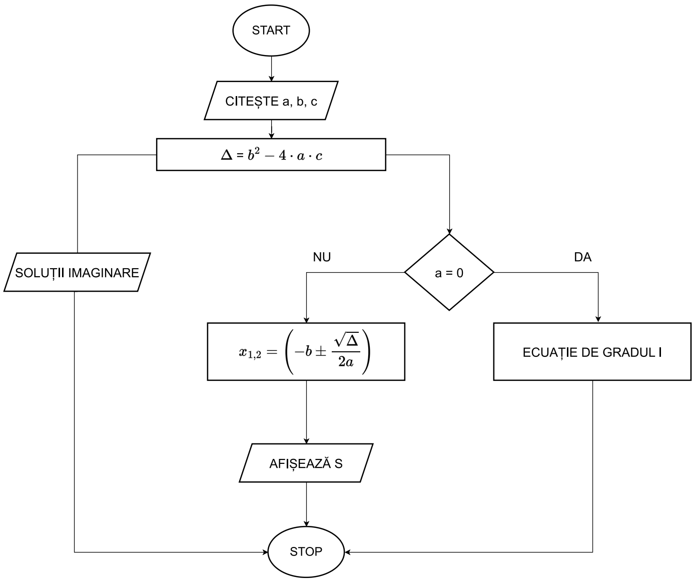


Figura 1-7. Schema logică pentru rezolvare ecuație de gradul doi

1.4.2 Exemple pseudocod

1. Pentru toate numerele cuprinse între 1 și 100 să se afișeze Fizz dacă numărul este divizibil cu 3, iar dacă numărul este divizibil cu 5 să se afișeze Buzz. În momentul în care numărul este divizibil cu ambele numere 3 și 5 atunci se va afișa FizzBuzz. În cazul în care nici una din variantele anterioare nu se îndeplinește se va afișa numărul curent.

```
ÎNCEPUT
PENTRU i începând de la 1 la 100 SE FACE
        DACĂ i este divizibil cu 3 și cu 5 ATUNCI
        IEȘIREA "FizzBuzz"
        ALTFEL DACĂ i este divizibil cu 3 ATUNCI
        IEȘIREA "Fizz"
        ALTFEL DACĂ i este divizibil cu 5 ATUNCI
        IEȘIREA "Buzz"
        ALTFEL
```

IEȘIREA i

SFÂRȘIT

2. Scrieți sub formă de pseudocod un algoritm care afișează factorialul unui număr introdus de utilizator.

```
ÎNCEPUT
OBȚINE n
INIȚIALIZARE i = 1, f = 1
CÂT TIMP (i<=n) SE FACE
        f = f * i
        i = i + 1
SE INCHIDE BUCLA CÂT TIMP
AFIȘARE f
SFÂRȘIT
```

1.5. Exerciții

1. Să se realizeze o schemă logică a unui program care rezolvă o ecuație de gradul întâi având coeficienții X, Y dați de către utilizator, ca la final să se afișeze soluția.

2. Să se realizeze o schemă logică a unui program care calculează factorialul unui număr n introdus de către utilizator, ca la final să se afișeze rezultatul.

3. Să se scrie un algoritm sub formă de pseudocod prin care să se calculeze numărul de picioare dintr-o curte, ținând cont că se află x găini, y oameni și z câini. Numărul de câini, oameni și găini este stabilit de către utilizator.

4. Să se scrie un algoritm sub formă de pseudocod care calculează soluțiile pentru o ecuație de gradul doi cu coeficienții x, y și z dați de utilizator.

Capitolul 2

Programarea aplicațiilor simple în C

2.1. Aspecte generale

După cum s-a menționat în capitolul 1, informația în calculator este reprezentată în sistem binar, prin urmare, limbajul mașină este limbajul fundamental al unui calculator. Acesta transmite sau utilizează instrucțiunile folosind secvențe de 0 și 1. Aceste secvențe sau instrucțiuni codificate pot părea lipsite de semnificație pentru noi, astfel că primele limbaje de programare, respectiv limbajele de asamblare au fost dezvoltate pentru a simplifica munca programatorului. Totuși, un calculator nu poate executa direct instrucțiunile în limbaj de asamblare. Aceste instrucțiuni trebuie întâi traduse în limbaj mașină.

Un program numit asamblor convertește instrucțiunile limbajului de asamblare în limbaj mașină. Trecerea de la limbajul mașină la limbajul de asamblare a simplificat programarea, însă programatorul era în continuare obligat să gândească în termeni de instrucțiuni individuale ale mașinii. Următorul pas în simplificarea programării a implicat dezvoltarea limbajelor de programare de nivel înalt, acestea fiind mai apropiate de limbajul natural în care gândim și comunicăm. Basic (Beginner's All-purpose Symbolic Instruction Code), FORTRAN (FORmula TRANslation), COBOL(Common Business - Oriented Language), C, C++, C#, Java și Python sunt toate limbaje de nivel înalt. În cadrul acestei cărți, veți învăța limbajul de programare de nivel înalt C/C++.

După scrierea și salvarea unui program în limbajul C++, utilizând extensia *.cpp*, etapa următoare este traducerea codului sursă din limbaj de programare în limbaj mașină. Acest proces se numește *compilare*, iar programul (sau aplicația) utilizat pentru a realiza acest proces este denumit *compilator*.

Programele scrise în limbaje de nivel înalt nu pot fi, însă, executate direct de către procesor. O soluție frecvent utilizată este apelarea la un interpretor, care traduce și execută pe rând instrucțiunile limbajului de nivel înalt. *Interpretorul* constă într-un set de instrucțiuni în limbaj mașină pe care procesorul le poate executa imediat, iar *programul sursă* reprezintă un set de instrucțiuni pe care interpretorul le cunoaște și le poate procesa.

Ca rezultat al compilării, se obține un *program executabil* cu extensia *.exe*, Figura 2-1. Pentru a înțelege mai bine conceptele prezentate, în subcapitolul 2.3 se vor descrie toți pașii ce trebuie urmați pentru a crea un proiect în cadrul mediului de dezvoltare Visual Studio.

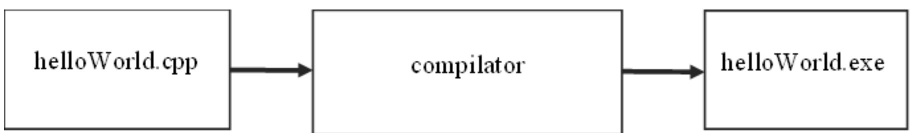


Figura 2-1. Diagrama procesului de compilare a aplicației HelloWorld

2.2. Medii de programare

Pentru dezvoltarea aplicațiilor de tip consolă (Console Application), se pot utiliza medii de dezvoltare integrate (IDE) cum ar fi: *Visual Studio* (pentru Windows), *Visual Studio Code* (pentru Windows, Mac sau Linux), *CLion, CodeLite* etc. De asemenea, se poate folosi și un compilator online, precum: *Online GDB*, disponibil la adresa https://www.onlinegdb.com/online_c++_compiler, *Compiler Explorer*, disponibil la adresa: https://godbolt.org/, *Wandbox*, disponibil la

adresa https://wandbox.org/, *Rextester* disponibil la https://rextester.com/ etc.

Compilatoarele online pot fi folosite, de exemplu, pentru a testa aplicații sau exemple simple prezentate în cadrul cărții. Cu toate acestea, în cazul aplicațiilor complexe, se recomandată utilizarea unui mediu de dezvoltare integrat pentru a beneficia de toate funcționalitățile pe care acesta le oferă.

În cadrul acestei cărți se va utiliza mediul de dezvoltare *Visual Studio 2022*, pe care îl puteți descărca de la adresa: https://visualstudio.microsoft.com/downloads/, ediția gratuită oferită de către Microsoft fiind *Visual Studio Community.* Microsoft oferă de asemenea edițiile Professional și Enterprise, o comparație sumară a acestor trei ediții este disponibilă la adresa: https://visualstudio.microsoft.com/vs/compare/.

Pașii necesari pentru instalarea mediului de dezvoltare sunt menționați în documentația oficială pusă la dispoziție utilizatorilor de către Microsoft la adresa: https://learn.microsoft.com/en-us/cpp/build/vscpp-step-0-installation?view=msvc-170#visual-studio-2022-installation. Pentru a dezvolta aplicații în C/C++ trebuie să verificați că ați instalat pachetul "*Desktop development with C++*".

2.3. Dezvoltarea aplicațiilor tip Consolă în Visual Studio 2022

1. Pentru a crea primul proiect de tip Console Application se lansează mediul de dezvoltare, Visual Studio, după care se parcurg următorii pași: după deschiderea mediului de dezvoltare, Visual Studio, din meniul principal, selectați File | New | Project, Figura 2-2.

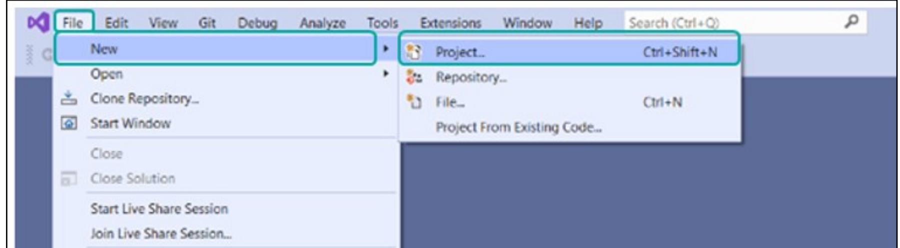


Figura 2-2. Pasul 1 - Crearea unui nou proiect în Visual Studio

2. În fereastra "*Create a new project*" se va selecta "*Console App*" din lista de șabloane disponibile pentru limbajul C++ și apoi click pe butonul "*Next*", Figura 2-3.

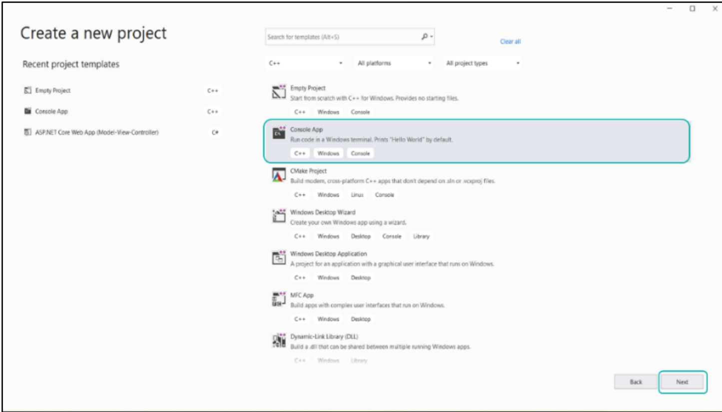


Figura 2-3. Pasul al-2 lea- Alegerea modelului de proiect

3. Alegeți un nume sugestiv pentru proiect și specificați locația sau calea către directorul în care doriți să salvați proiectul, Figura 2-4.

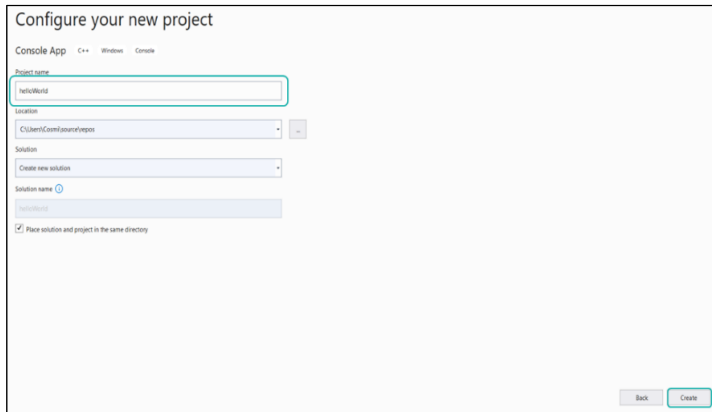


Figura 2-4. Pasul al-3-lea – Configurarea locației pentru proiect

4. Visual Studio va genera un șablon tip consolă, cu o funcție principală, funcția "main()" unde se va introduce codul sursă al aplicației, Figura 2-5.

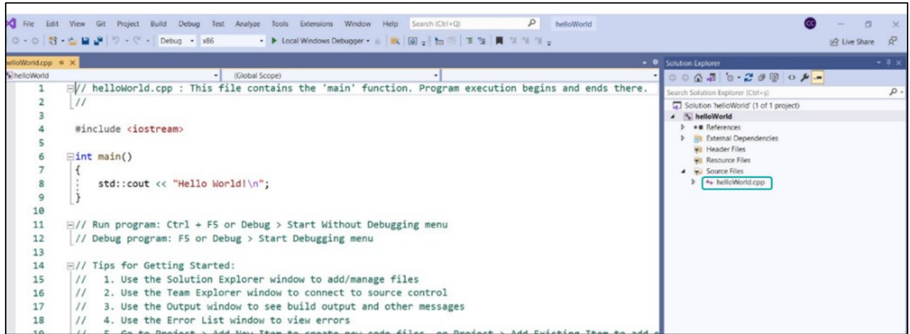


Figura 2-5. Pasul al-4-lea – Generarea proiectului în Visual Studio

Pentru a adăuga proiectului creat anterior un fișier sursă suplimentar celui existent, creat automat ("*helloWorld.cpp*"), se va selecta în "*Solution Explorer*" Source File | Add| New Item, Figura 2-6. În fereastra "*Add New Item*" selectați "*C++ File (.cpp)*". Ulterior în câmpul "*Name*", introduceți numele pentru fișierul sursă creat, verificând că acesta are extensia "*.cpp*" și dați click pe butonul "*Add*".

În cazul în care la pasul al-2-lea se optează pentru "*Empty project*", va trebui să se adauge obligatoriu un prim fișier sursă în cadrul proiectului creat.

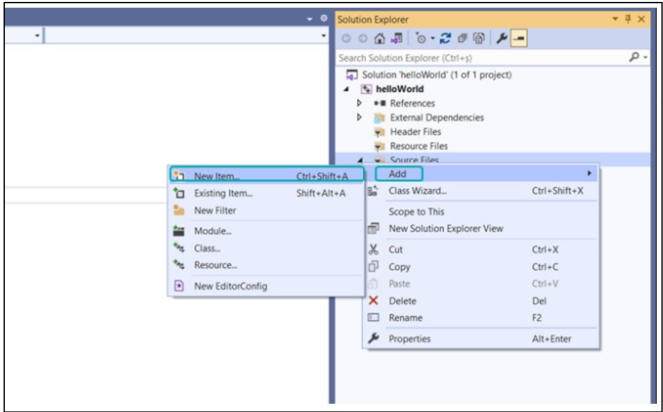


Figura 2-6. Pasul al-4-lea – Adăugarea unui nou fișier în proiect

5. Programul sursă se introduce în fereastra de editare. Ulterior, acesta se salvează fie utilizând butonul "*Save*" situat în bara de instrumente, fie accesând opțiunea "*Save*" din meniul "File", Figura 2-7.

Figura 2-7. Pasul al-5-lea – Salvarea modificărilor din proiect

6. Pentru compilarea proiectul creat la pașii anteriori se selectează în meniul Debug opțiunea Build solution (tastele Ctrl + Shift + B), Figura 2-8.

În fereastra „*Output*”, se poate observa rezultatul procesului de compilare: "*Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped*".

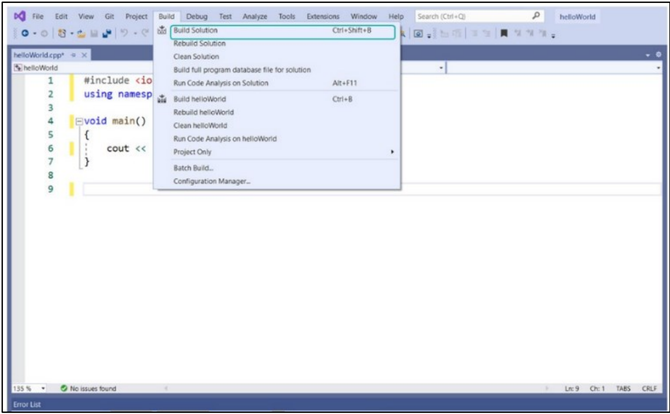


Figura 2-8. Pasul al-6-lea – Compilarea proiectului

Astfel, compilatorul traduce codul sursă în cod obiect, verificând codul din punct de vedere lexical, sintactic și semantic. Dacă

sunt identificate erori de compilare, acestea vor fi afișate utilizatorului în fereastra "*Error List*", Figura 2-9.

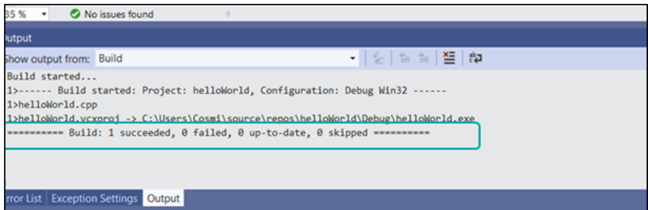


Figura 2-9. Pasul al-6-lea – Statistica acțiunii de compilare

Conform exemplului prezentat în procesul de compilare nu s-a finalizat cu succes datorită unei erori de sintaxă "*syntax error: missing ';' before '}'*", Figura 2-10. Prin urmare, procesul de rebuild (meniul Build | Rebuild Solution) se va putea efectua doar după identificarea și corectarea erorii menționate. După corectarea erorii/erorilor, mediul de dezvoltare va iniția cele două procese succesive: procesul de compilare și cel de link-editare, unde modulele de cod obiect sunt unite și bibliotecile necesare integrate, rezultând astfel în final aplicația executabilă.

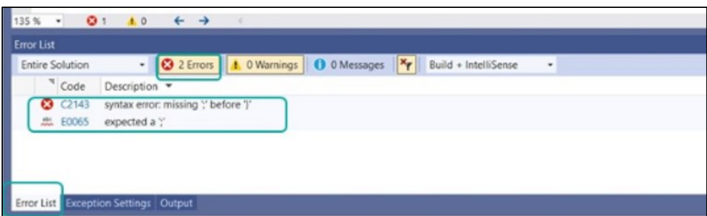


Figura 2-10. Pasul al-6-lea – Identificarea erorilor din procesul de compilare

7. Programul executabil generat poate fi rulat în Visual Studio, folosind opțiunile "*Start Debugging*" sau "*Start Without Debugging*" accesibile direct din meniul "*Debug*" Figura 2-11, sau din bara de instrumente Figura 2-12. Tehnicile de debugging în Visual Studio vor fi prezentate în subcapitolul 2.5.

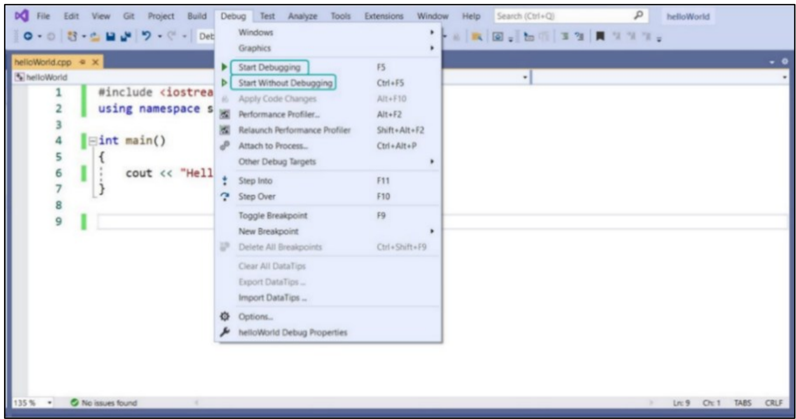


Figura 2-11. Pasul al-7-lea – Rularea proiectului din meniul Debug

Figura 2-12. Pasul al-7-lea – Variantă alternativă pentru rularea proiectului

După rularea programul executabil în Visual Studio, aplicația poate fi vizualizată într-o fereastră separată sau în consolă, depinzând de tipul aplicației dezvoltate. La final pentru a închide aplicația, fie se închide fereastra în care aceasta rulează, fie se oprește execuția din Visual Studio, dacă este rulată în modul debugging.

Pentru a relua sau deschide aplicația, se va accesa folder-ul proiectului unde este salvat fișierul cu extensia *.sln* și se va realiza dublu-click pe acesta pentru a-l deschide în Visual Studio. Ulterior, se rulează din nou programul utilizând opțiunile '*Start Debugging*' sau '*Start Without Debugging*' accesibile din meniul '*Build*' al mediului de dezvoltare, Visual Studio, Figura 2-13.

De asemenea, asigurați-vă că salvați toate modificările efectuate în codul sursă înainte de a închide Visual Studio, pentru a

preveni pierderea datelor și a asigura coerența și corectitudinea codului la re-deschidere.

Name	Date modified	Type
.vs	9/21/2023 11:19 PM	File folder
Debug	9/23/2023 1:05 AM	File folder
helloWorld.cpp	9/23/2023 1:05 AM	C++ source file
helloWorld.sln	9/21/2023 11:19 PM	Microsoft Visual Studio ...
helloWorld.vcxproj	9/21/2023 11:30 PM	VC++ Project
helloWorld.vcxproj.filters	9/21/2023 11:30 PM	VC++ Project Filters File
helloWorld.vcxproj.user	9/21/2023 11:19 PM	Adobe Acrobat Docume...
main.cpp	9/29/2021 9:13 PM	C++ source file

Figura 2-13. Deschiderea/redeschiderea unui proiect în Visual Studio

2.4. Exemplu de scriere a unor programe simple în C /C++

Atunci când dezvoltăm sau scriem un program, parcurgem mai multe etape esențiale. În continuare, sunt reprezentate schematic toate aceste etape pentru a facilita înțelegerea, încă de la primele capitole, a procesului de dezvoltare a unui program în limbajul de programare C/C++. Ulterior, se va prezenta un exemplu de aplicație simplă, pentru a sublinia concret etapele reprezentate schematic, Figura 2-14.

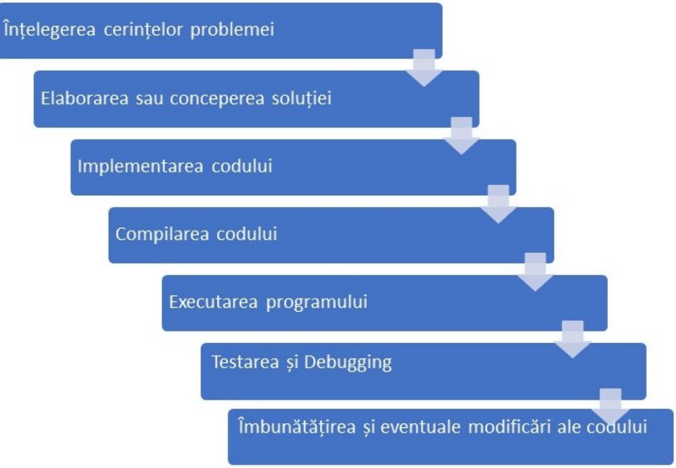


Figura 2-14. Etapele de rezolvarea a unei probleme/ dezvoltare a unui program în limbajul de programare C/C++

În cele ce urmează se dorește scrierea unui program care să ne ofere posibilitatea de a calcula suma totală a caloriilor consumate într-o zi, luând în considerare cele trei mese principale: mic dejun, prânz și cină.

Cerințe:

1.Input:

- Numărul de calorii pentru micul dejun.
- Numărul de calorii pentru prânz.
- Numărul de calorii pentru cină.

2.Output: Totalul caloriilor consumate într-o zi, obținut prin adunarea caloriilor de la fiecare masă.

3.Soluție:

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5    int breakfast_calories;
6    int lunch_calories;
7    int dinner_calories;
8
9    cout << "Introduceti numarul de calorii pentru micul dejun: ";
10   cin >> breakfast_calories;
11
12   cout << "Introduceti numarul de calorii pentru pranz: ";
13   cin >> lunch_calories;
14
15   cout << "Introduceti numarul de calorii pentru cina: ";
16   cin >> dinner_calories;
17
18   int total_calories = breakfast_calories + lunch_calories + dinner_calories;
19   cout << "Numarul total de calorii consumate in aceasta zi este: " <<
20 total_calories << endl;
21
22 return 0;
23 }
```

2.5. Utilizarea debugger-ului în Visual Studio 2022

2.5.1 Adăugare/Ștergere breakpoint

Procesul de debugging este extrem de important pentru dezvoltarea de software și esențial pentru orice programator [13]. Printre avantajele acestuia am putea menționa faptul că permite înțelegerea logicii și funcționalității codului mai ales dacă se lucrează cu aplicații complexe. Al doilea avantaj major este că facilitează refactoring-ul (îmbunătățește structura codului fără a-i schimba funcționalitatea).De asemenea permite monitorizarea stării variabilelor oferind informații despre valorile atribute variabilelor sau obiectelor în timpul execuției, facilitând identificarea și corectarea posibilelor erori. Cu toate acestea, trebuie să reținem că rolul debugger-ului este de a ajuta programatorul să identifice erorile în cod, dar interpretarea și soluționarea acestor erori revin programatorului.

Mediul de dezvoltare Visual Studio facilitează semnificativ procesul de debugging, oferind o interfață intuitivă și accesibilă. Astfel în Visual Studio se poate începe debugging-ul apăsând tasta F5 sau din meniul "*Debug*" | "*Start Debugging*". Programul va fi executat până ajunge la primul *breakpoint* , termenul echivalent în română ar fi întrerupere, astfel execuția programului va fi oprită intenționat.

Pentru a adăuga un breakpoint avem mai multe opțiuni, cea mai simplă modalitate este să faceți click pe bara verticală din partea stângă, lângă numărul liniei de cod unde doriți să opriți execuția, astfel se va seta un breakpoint reprezentat sub forma unui cerc roșu. O altă modalitate de a seta breakpoint-ul este să plasați cursorul la linia de cod unde doriți să setați breakpoint-ul și din meniul "*Debug*" | "*Toggle Breakpoint*" (tasta F9), Figura 2-15.

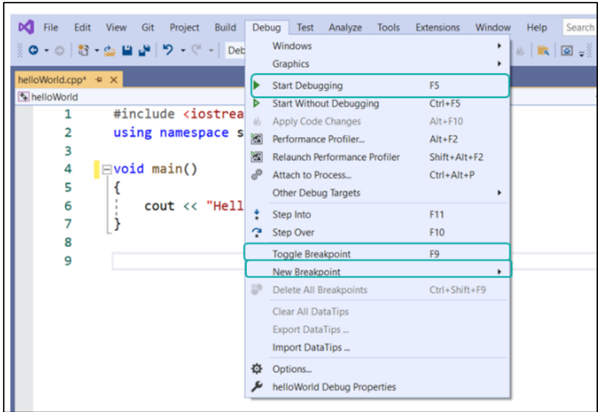


Figura 2-15. Procesul de debugging și setare a breakpoint-urilor în Visual Studio

În acest moment puteți plasa cursorul peste variabilele din cod pentru a vizualiza valorile lor curente. De asemenea, este posibil să utilizați fereastra "*Watch*" pentru a adăuga variabilele pe care doriți să le inspectați mai detaliat.

Pentru a dezactiva un breakpoint, există mai multe opțiuni: putetți apăsa din nou tasta F9 atunci când cursorul este poziționat pe linia de cod corespunzătoare, puteți da click din nou pe cercul roșu, sau ca a treia opțiune, puteți da click dreapta pe cercul roșu și să selectați opțiune "Delete Breakpoint", Figura 2-16.

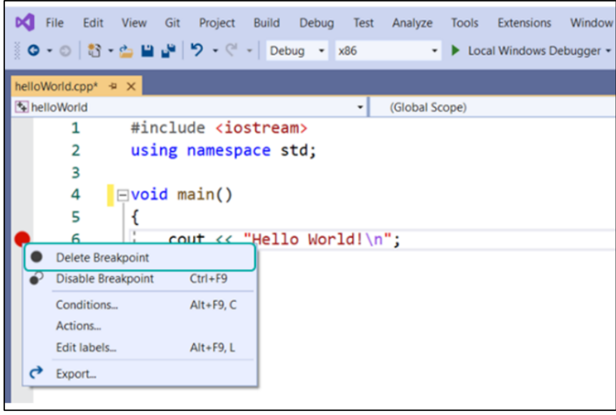


Figura 2-16. Procesul de ștergere a breakpoint-urilor în Visual Studio

2.5.2 Rularea pas cu pas a programului

Pentru a rula programul pas cu pas, respectiv până la următorul breakpoint sau până la terminarea acestuia, există opțiunea de a selecta butonul "*Continue*" (de obicei tasta F5).Pentru a controla execuția codului în timpul procesului de debugging, avem la dispoziție următoarele trei opțiuni:

- "*Step Over*" (*tasta F10*), permite executarea următoarei linii de cod, evitând saltul la codul funcțiilor apelate ce s-ar putea afla pe linia executată;
- "*Step Into*" *(tasta F11)*, permite executarea următoarei linie de cod, realizând în timpul execuției inclus salt la liniile de cod asociate codului pentru funcțiile apelate ce se află pe linia de cod executată;
- "*Step Out*" *(tastele Shift + F11)*vă permite să finalizați execuția funcției curente și să reveniți la funcția de unde a fost apelată.

2.5.3 Watch window

"Watch window" a fost menționată inițial la adăugarea și ștergerea unui breakpoint. Prin adăugarea unei variabile în această fereastră se poate observa cum valoarea acesteia se modifică pe parcursul realizării procesului de debug, Figura 2-17.

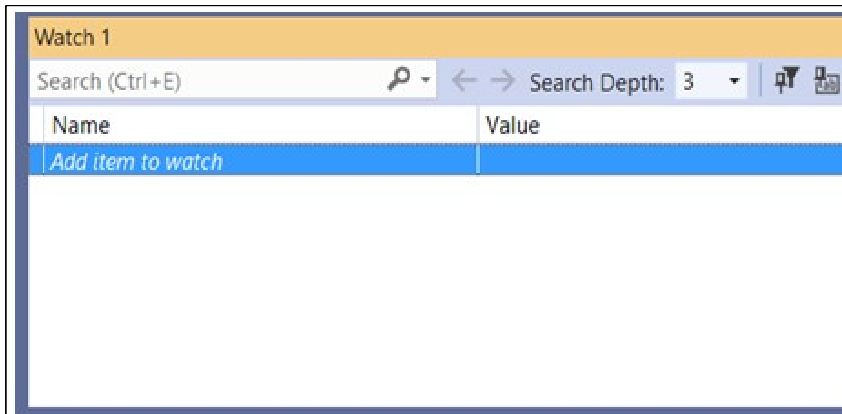


Figura 2-17. Fereastra "Watch" în Visual Studio

2.6. Exemplu utilizare utilizarea debugger-ului în Visual Studio 2022

Revenind la exemplul furnizat în cadrul subcapitolului 2.4, Exemplu de scriere a unor programe simple în C /C++, se va prezenta procesul de debugging și utilizarea ferestrei '*Watch*' în mediul de dezvoltare Visual Studio. Înainte de a începe debugging-ul se va plasa breakpoint-ul la linia de cod 18, Figura 2-18.

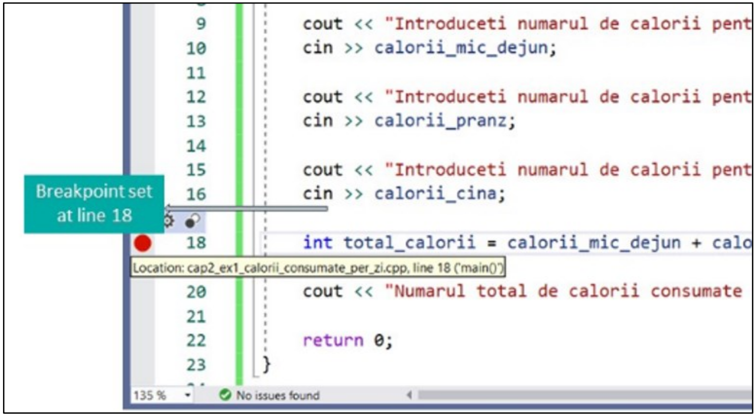


Figura 2-18. Setare breakpoint în mediul de dezvoltare Visual Studio

După setarea breakpoint-ului/breakpoint-urilor, prin intermediul tastei F5 (în majoritatea IDE-urilor) vom rula programul în modul debugging, Figura 2-19.

Figura 2-19. Inițierea procesului de Debugging în mediul de dezvoltare Visual Studio

Când execuția ajunge la breakpoint-ul setat, variabilele pot fi inspectate fie prin plasarea cursorului deasupra numelui variabilei, fie adăugând variabilele în fereastra "*Watch*", Figura 2-20.

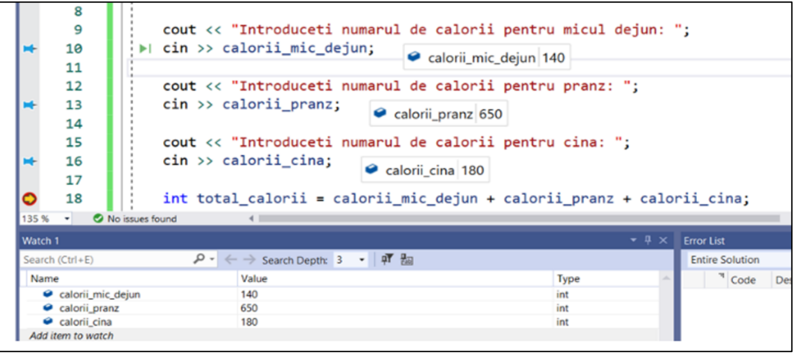


Figura 2-20. Verificarea valorii variabilelor în mediul de dezvoltare Visual Studio

Continuând se pot utiliza opțiunile "*Step Over*" (F10), "*Step Into*" (F11), și "*Step Out*" (Shift + F11) pentru a controla execuția programului pas cu pas și a observa modul în care se modifică valorile variabilelor, Figura 2-21.

Dacă sunt identificate erori sau comportamente neașteptate, este posibilă oprirea debugger-ului pentru a efectua eventuale modificări necesare codului și ulterior se poate relua procesul de debugging. Astfel, după identificarea și corectarea erorilor, este necesar să verificăm dacă aplicația funcționează corespunzător cerințelor sau specificațiilor.

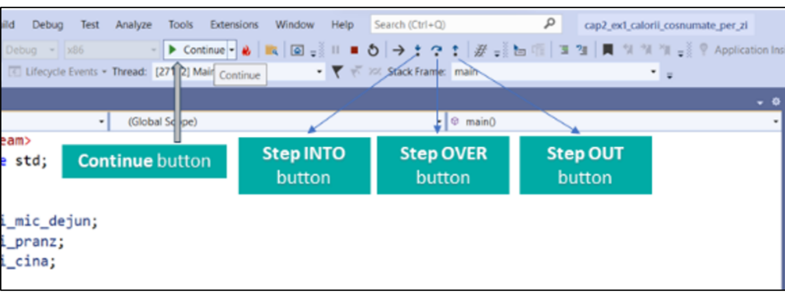


Figura 2-21. Controlul execuției programului în cadrul procesului de Debugging în mediul de dezvoltare Visual Studio

2.7. Exerciții

1.Instalați mediul de dezvoltare Visual Studio.

Instrucțiuni:

a) Accesați site-ul oficial Visual Studio și descărcați ediția gratuită, Community.
b) Urmăriți pașii de instalare și asigurați-vă că selectați pachetele necesare pentru dezvoltarea aplicațiilor în C++.
c) După instalare, deschideți mediul de dezvoltare Visual Studio și creați un proiect pentru a verifica dacă instalarea a fost realizată cu succes.

2. Testați exemplele de cod furnizate în acest capitol.

Instrucțiuni:

a) Deschideți mediul de dezvoltare Visual Studio și creați un proiect nou în C++.
b) Copiați cel de al -2-lea exemplu de aplicație prezentat în acest capitol în proiectul creat.
c) Compilați și executați programul pentru a vedea rezultatele.

3. Familiarizarea cu procesul de debugging în Visual Studio compilarea, rularea și debugging-ul exemplului furnizat.

Instrucțiuni:

a) Utilizând exemplul de la exercițiul precedent setați un breakpoint la linia 18 de cod.
b) Rulați programul în modul debugging (F5) și observați valorile variabilelor în fereastra "Watch".
c) Utilizați opțiunile "Step Over" (F10) și "Continue" pentru a parcurge codul pas cu pas, și observați cum se modifică valorile variabilelor.

Capitolul 3

Variabile și tipuri de date fundamentale

3.1. Noțiunea de variabilă

O variabilă are ca scop rezervarea unui anumit spațiu în memoria calculatorului, în vederea stocării datelor sub un anumit nume definit de utilizator [11], [14], [15]. Memoria alocată unei variabile este disponibilă pentru a fi accesată și modificată ulterior de către programator.

Declararea unei variabile trebuie realizată înaintea de utilizarea acesteia într-un program, constând în precizarea tipului și numelui variabilei. După declararea variabilei aceasta poate fi folosită în program ținând cont de tipul de date folosit.

În vederea stabilirii numelui variabilei există un set de reguli bine definite [14]:

a) fiecare variabilă trebuie să aibă un nume sugestiv;
b) în numele unei variabile se pot utiliza litere, cifre si caracterul de subliniere.
c) numele unei variabile trebuie să înceapă cu literă mică,
d) pentru variabile formate din doua cuvinte se recomandă convenția *camelCase*;
e) cuvintele cheie nu pot fi utilizate ca nume de variabile.
f) Variabilele care au același tip de date pot fi scrise pe aceeași linie.

Inițializarea unei variabile este procesul prin care se face atribuirea unei anumite valori variabilei declarate, folosind operatorul

„=”. De asemenea atribuirea poate să aibă loc inclusiv în momentul declarării variabilei. Sintaxa utilizată este următoarea „tip nume_variabilă = valoare”, iar dacă atribuirea unei valori nu se face în momentul declarării variabilei, sintaxa va fi următoarea „nume_variabilă = valoare”.

3.2. Tipuri de date elementare

În limbajul de programare C/C++ lucrăm cu următoarele tipuri de date elementare: char, int, float, double și boolean.

Tipurile de date întregi reprezintă toate numerele întregi, indiferent de semnul acestora [8]:

a) Interval cuprins între -2.147.483.648 și 2.147.483.647.
b) Dimensiune de 4 octeți.
c) Specificator de formă “%d”.
d) Pentru a declara o variabilă de tipul întreg se folosește cuvântul cheie “int” înaintea variabilei (int nume_variabilă).
e) Tipul de date întreg poate fi utilizat cu următorii modificatori de acces: unsigned int (numere întregi fără semn), short int, long int, unsigned short int.

Char este un tip de date elementar din C, ce permite stocarea unui singur caracter (un octet/byte). Acest tip de date permite programatorului să lucreze cu date sub formă de text și să efectueze operații pentru manipularea șirurilor:

a) Interval cuprins intre -128 la 127 sau de la 0 la 255 (depinde de compilator).
b) Dimensiunea de 1 octet.
c) Specificator de formă “%c”.
d) Pentru a declara o variabila se folosește cuvântul cheie “char” înaintea variabilei (char nume_variabila).

Tipul de date “float” este utilizat cu scopul de a reprezenta numere reale, și este reprezentat pe 32 de biți:

a) interval cuprins între -3.4 x 10^38 la 3.4 x 10^38.
b) dimensiunea de 4 octeți.
c) precizie 7 zecimale;
d) specificator de formă “%f”.
e) pentru a declara o variabilă se folosește cuvântul cheie “float” înaintea variabilei (float nume_variabila).

Double este un tip de date elementar folosit pentru a reprezenta numere reale, fiind este reprezentat pe 32 de biți:

a) interval cuprins intre -1.7 x 10^308 la 1.7 x 10^308.
b) dimensiunea de 8 octeți.
c) precizie 16 zecimale;
d) specificator de formă “%lf”.
e) Pentru a declara o variabila se folosește cuvântul cheie “double” înaintea variabilei (double *nume_variabila*).

Tipul de date boolean este folosit în scopul reprezentării valorilor logice și poate avea doar una din cele două valori de adevărat: “true” sau "fals" [16]. Aceste tipuri de date sunt utilizate de obicei pentru condiționare și operații logice în vederea controlării fluxului unui program.

3.3. Operatorii limbajului C (C++)

Limbajele de programare de nivel înalt folosesc o gamă variată de operatori. În C (și C++), există patru categorii principale de operatori: operatori matematici, operatori relaționali, operatori logici și respectiv operatori pe biți [17], [18]. Termenul "*operanzi*" se referă la valorile sau variabilele cu care operatorii efectuează operații.

De exemplu, în expresia a + b, variabila "a" și variabila "b" sunt considerate operanzi, în timp ce semnul + (plus) reprezintă un operator. C/C++ oferă o varietate de tipuri de operatori, și aceștia sunt explicați pe scurt în acest subcapitol.

3.3.1 Operatori matematici

În tabelul de mai jos, regăsiți operatorii matematici din limbajul C, împreună cu o scurtă descriere și exemple, Tabel 3.1. Operatorii matematici, [19].

Tabel 3.1. Operatorii matematici

Operator	Semnificație	Exemplu
+	Adunare	int suma = a + b;
-	Scădere	int diferența = x – y;
*	Înmulțire	int produs = m * n;
/	Împărțire	double rezultat = x / y;
%	Modulo (restul împărțirii)	double rest = x % y;

3.3.2 Operatorii relaționali

În tabelul de mai jos, veți regăsiți operatorii relaționali din limbajul C, Tabel 3.2, [20].

Tabel 3.2. Operatorii relaționali

Operator	Semnificație	Exemplu
= =	Egalitate	a = = b
!=	Diferit de	(x != y)
>	Mai mare	a > (lim + 1)
<	Mai mic	p < (lim + 1)
>=	Mai mare sau egal	a >= b
<=	Mai mic sau egal	x <= y

3.3.3 Operatori logici

În limbajul de programare C++, avem trei operatori logici principali, Tabel 3.3.

Tabel 3.3. Operatori logici

Operator	Semnificație	Exemplu
&&	Operatorul logic "și" – True dacă ambele expresii sunt adevărate	if (expr1 && expr2) { /* Bloc de instrucțiuni*/ }
||	Operatorul logic "sau" – True dacă cel puțin una dintre expresii este adevărată.	if (expr1 || expr2) { /* Bloc de instrucțiuni*/ }
!	Operatorul logic de "negare"	if (!expr) { /* Bloc de instrucțiuni*/ }

3.3.4 Operatori pe biți

În limbajul de programare C, avem și operatori specializați care ne permit să efectuăm operații la nivel de biți. În tabelul următor, se vor prezenta operatorii pe biți [17].

Tabel 3.4. Operatori pe biți

Operator	Semnificație	Exemplu
&	Operatorul "ȘI pe biți"	number1 & number2;
|	Operatorul „SAU pe biți"	number1 | number2;
^	Operatorul "SAU exclusiv pe biți"	number1 ^ number2;
~	Operatorul "Complement pe biți"	~number1;
<<	Operatorul "Operatorul "Shiftare la stânga" "	number1 << n;
>>	Operatorul "Shiftare la dreapta"	number1 >> n;

3.3.5 Operatorii de incrementare și de decrementare

Operatorul de *incrementare* "++", de exemplu a++ sau ++a, este echivalent în matematică cu "adună 1 la a", ceea ce înseamnă a = a + 1. În mod similar, operatorul de *decrementare* "--", de exemplu a-- sau --a, este echivalent cu "scăderea lui 1 din a", ceea ce înseamnă a = a - 1 [11].

Operatorul de *incrementare* "++" și respectiv operatorul de decrementare "--" după cum s-a putut observa, pot fi plasați fie înainte, fie după numele variabilei. Când sunt plasați înainte, operația se numește *preincrementare* (pentru "++") sau *predecrementare*

(pentru "--"). Când sunt plasați după, operația se numește *postincrementare* (pentru "++") sau *postdecrementare* (pentru "--").

Preincrementarea sau *predecrementarea* înseamnă că valoarea variabilei este modificată înainte de a fi utilizată în orice altă operație în aceeași expresie. *Postincrementarea* sau *postdecrementarea* înseamnă că valoarea originală a variabilei este utilizată în expresie înainte de a fi modificată. Mai jos se regăsesc câteva exemple a operatorilor menționați.

Exemplu preincrementare:

```
1 int a = 22;
2
3 int b = ++a;
```

Exemplu postincrementare:

```
1 int a = 22;
2
3 int b = a++;
```

Și pentru operatorul de decrementare, exemplu predecrementare:

```
1 int a = 22;
2 int b = --a; /*Predecrementare: a este mai întâi decrementat
3                la valoarea 21, apoi valoarea lui a (care este acum
4                21) este asignată lui b.*/
```

Exemplu postdecrementare:

```
1 int a = 22;
2 int b = a--; /*Postdecrementare: Valoarea originală a lui
3                (care este 22) este mai întâi asignată lui b
4               apoi a este decrementat la 21.*/
```

Iată și un ultim exemplu, unde operatorii "++ "și " -- "sunt combinați în aceeași expresie:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a = 1, b = 2, c = 3;
6     cout << (++a) - (b--) + (--c) << endl;
7
8     return 0;
9 }
```

Când utilizăm operatorii de incrementare și decrementare în expresii complexe, este esențial să înțelegem ordinea de evaluare și impactul acestora asupra valorii variabilelor. În exemplul anterior, *cout << (++a) - (b--) + (--c) << endl;*, modul în care fiecare operator modifică conținutul variabilei:

1. *(++a)*: valoarea variabilei *a* este mai întâi incrementată de la valoarea 1 la 2, iar valoarea 2 este utilizată în expresie.
2. *(b--)*: valoarea variabilei *b* (care este 2) este utilizată în expresie, iar apoi aceasta (valoarea variabilei *b*) este decrementată la 1.
3. *(--c)*: valoarea variabilei *c* este mai întâi decrementată de la 3 la 2, iar valoarea 2 este utilizată în expresie.

Expresia devine acum: 2 - 2 + 2, ceea ce se evaluează la 2. Valoarea finală afișată va fi 2, iar valorile variabilelor vor fi: *a = 2*, *b = 1* și *c = 2*. Este important să înțelegem aceste tipuri de expresii pentru a evita eventuale erori introduse în codul nostru.

3.3.6 Operatorii și expresiile de atribuire

Operatorul de atribuire este reprezentat de simbolul egal (=). După ce expresia situată în dreapta simbolului de atribuire este evaluată, aceasta este asignată variabilei din stânga.

Operatorii de atribuire compusă sunt folosiți pentru a scurta instrucțiunile de forma: *exp1 op = exp2*. Această instrucțiune este echivalentă cu *exp1 = exp1 op (exp2)*. Observați că expresia *exp2* este

între paranteze din motive de prioritate. De obicei, operatorul *op* este unul dintre operatorii aritmetici: +, -, *, %, /, deși poate fi și oricare dintre operatorii de bit: &, ^, |, <<, >>. Mai jos regăsiți un exemplu de utilizare al operatorului de atribuire:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5   int a = 22, b = 2;
6   a += 6; //echivalent cu a = a + 6, variabila a devine 28
7   a *= b + 3; /*echivalent cu a = a * (b + 3), variabila a
8                devine 140 (28 * (2 + 3))*/
9   a -= b + 8; /*echivalent cu a = a - (b + 8), variabila a
10               devine 130 (140 - (2 + 8))*/
11  a /= b; //echivalent cu a = a / b, variabila a devine 65
12          //(130 / 2)
13  a %= b + 1; /*echivalent cu a = a % (b + 1), variabila a
14               devine 1 (65 % (2 + 1))*/
15  cout << "Num = " << a << endl;
16
17  return 0;
18 }
```

3.4. Conversiile de tip (cast sau transtipaj)

Când într-o expresie intervin operanzi de tipuri diferite, operanzii corespunzând unui tip mai restrictiv se convertesc automat spre tipul mai general astfel încât să nu se producă pierdere de informație [2].

Un exemplu simplu de conversie implicită este conversia unui caracter (*char*) într-un întreg (*int*). Char este reprezentat pe 8 poziții binare și poate fi convertit direct într-un întreg.

```
1  #include <iostream>
2  using namespace std;
3
```

```
4  int main() {
5    char c = '9'; // caracterul '9'
6    int n = c; // Conversie implicită din char în int
7    cout << "Valoarea lui n este: " << n << endl;
8            // Afișează:
9            //Valoarea lui n este: 57 (valoarea ASCII pentru  //'9')
10   return 0;
11 }
```

În acest exemplul de mai sus, s-a folosit o conversie implicită, iar caracterul '9' a fost convertit la valoarea sa ASCII, care este 57.

Un alt exemplu ar fi următorul program care convertește un șir *s* de cifre într-o valoare întreagă n.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    // Definim un șir de caractere care conține cifre
6    char s[] = "9876";
7    // Declarăm variabilele n și respectiv i de tip întreg
8    int n = 0;
9    int i = 0;
10   // Parcurgem șirul de caractere
11   while (s[i] >= '0' && s[i] <= '9') {
12     n = 10 * n + (s[i] - '0'); // Convertim cifrele din șir
13                 //în valoare întreagă și le adăugăm la n
14     i++;
15   }
16
17   cout << "Valoarea întreagă obținută din șir este: " << n
18        << endl; // Afișăm rezultatul
19   return 0;
20 }
```

Pe de altă parte, conversiile de tip explicit se realizează prin intermediul operatorilor de cast. Aceștia permit specificarea în mod explicit a tipului de date în care se face conversia, utilizând următoarea

sintaxă: *(nume-tip) expresie*, unde "*nume-tip*" reprezintă noul tip de date în care se dorește să se facă conversia, iar "*expresie*" este valoarea sau variabila care urmează să fie convertită.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    char c = '9'; // caracterul '9'
6    int n = (int) c; // Conversie explicită din char în int
7    cout << "Valoarea lui n este: " << n << endl; // Afișează:
8     //Valoarea lui n este: 57 (valoarea ASCII pentru '9')
9    return 0;
10 }
```

Regăsiți în cele ce urmează un alt exemplu, în care se utilizează conversia explicită (double) a variabilei "*number*" la tipul double. Rezultatul este apoi afișat pe consolă.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int number;
6    cout << "Introduceti un numar intreg: ";
7    cin >> number;
8    // Realizare conversie de tip
9    double valoare = (double) number;
10   // Afișarea rezultatului
11   cout << "Valoarea este: " << valoare << endl;
12   return 0;
13 }
```

3.5. Exerciții

1. Realizați un program care citește o serie de numere introduse de utilizator. Citirea numerelor se va încheia atunci când utilizatorul introduce valoarea 0. După terminarea introducerii numerelor,

programul va calcula și va afișa media aritmetică a tuturor numerelor introduse, fără a include valoarea 0 în calculul mediei.

2. Se cere realizarea unui program care calculează produsul: $P = \frac{2}{1} \times \frac{4}{3} \times \dots \frac{2n}{2n-1}$, unde n este un număr întreg pozitiv introdus de utilizator. Programul va afișa valoarea finală a produsului.
3. Să se implementeze un program care realizează conversia temperaturii din grade Celsius în grade Kelvin si Fahrenheit folosind următoarele formule: $K = C + 273.15$ și $F = C \times 9/5 + 32$. Temperatura în grade Celsius va fi citită de la tastatură, iar temperaturile în Kelvin și Fahrenheit vor fi afișate pe linii separate, cu o precizie de 3 zecimale.
4. Calculați media a două numere întregi. Instrucțiuni:
   a) Cere utilizatorului să introducă doua numere întregi *nr1* și *nr2*.

   b) Utilizați formula $(nr1 + nr2) / 2$ pentru a calcula rezultatul.

   c) Rezultatul să fie de tip „double”.

Capitolul 4

Directive preprocesor

Directivele preprocesor sunt comenzi date unei componente software specifică limbajului C denumită preprocesor și care acționează asupra programului sursă înaintea compilării acestuia. Caracteristicile principale ale directivelor preprocesor includ [11]:

1. **Simbolul '#'**: Toate directivele preprocesor încep cu simbolul *'#'*.
2. **Plasare**: Directivele preprocesor sunt plasate de obicei la începutul fișierelor cod sursă, dar pot fi situate oriunde în cod.
3. **Lipsa punctului și virgulei**: Directivele preprocesor nu se încheie cu ';' (punct și virgulă).
4. **Exemple de directive**: Printre cele mai comune directive preprocesor se numără #include (pentru includerea fișierelor sursă sau header), *#define* (pentru definirea constantelor și macro-urilor), *#if*, *#else*, *#endif* (pentru condiționarea compilării) și *#undef* (pentru anularea definiției unui macro).
5. **Scop și utilizare**: Directivele preprocesor sunt utilizate pentru a influența procesul de compilare, a gestiona dependențele, a defini constante și macro-uri, a condiționa compilarea unor părți de cod și a elimina codul care nu este necesar.
6. **Optimizare și gestionare**: Prin utilizarea directivelor preprocesor, programatorii pot optimiza și gestiona mai eficient codul sursă.

În concluzie, directivele preprocesor sunt un instrument vital în dezvoltarea de software în C și C++, oferind un nivel suplimentar de

control asupra codului sursă și contribuind la crearea de programe mai robuste, flexibile și ușor de întreținut.

4.1. Directivele #define și #undef

Directiva #define este utilizată pentru definirea unor constante sau macrouri. Sintaxa sa este: *#define* nume_macro valoare.

Rolul ei este de a defini o constantă sau o macroinstrucțiune numită *nume_macro* şi care este înlocuită în fișierul în care este inclusă, prin valoare. În exemplul de mai jos: #define LIMITA 10, LIMITA este un macro. Oriunde în codul se utilizează LIMITA, preprocesorul va înlocui LIMITA cu valoarea 10 înainte de compilare. Avantajele principale sunt că facilitează modificarea valorilor care se repetă în cod și este mai sigur și rapid decât varianta schimbării manuale a valorilor, în special în programele mari.

```
1  #include <iostream>
2  using namespace std;
3
4  #define LIMITA 10
5
6  int main() {
7   int x = 20 - LIMITA;
8   int y = 20 + LIMITA;
9   int z = 3 * LIMITA;
10  cout << "x = " << x << ", y = " << y << ", z = " << z
11       << endl;
12  return 0;
13 }
```

Prin urmare programul va afișa *x = 10*, *y = 30*, *z = 30*.

Un alt exemplu pentru definirea unei constante și utilizarea acesteia pentru a stabili dimensiunea unui șir de caractere:

```
1  #include <iostream>
2  using namespace std;
3
```

```
4  #define getmax(a, b)((a) > (b) ? (a) : (b)) // Calcularea
5                      //valorii maxime dintre a și b
6
7  int main() {
8    int x = 5, y;
9    y = getmax(x, 2); // getmax(x, 2) -> ((5) > (2) ? (5) : ( //))
10   cout << "Valoarea maxima dintre " << x << " si 2 este: "
11        << y << endl; // După execuție, y va avea valoarea 5
12   return 0;
13 }
```

Directiva #undef în C++ are rolul de a elimina o definiție preexistentă a unui macro, opus funcției directivei *#define* care creează sau definește un macro. Prin utilizarea directivei *#undef*, o constantă sau macro-instrucțiune specificată ca parametru este înlăturată din lista de constante predefinite.

Exemplu:

```
1  #include <iostream>
2  using namespace std;
3
4  #define LIMITA 10 // Definirea macro-ului LIMITA cu
5                  //valoarea 10
6
7  int main() {
8    // Calculul valorilor folosind valoarea inițială a macr
9    //-ului LIMITA
10   int x = 20 - LIMITA;
11   int y = 20 + LIMITA;
12   int z = 3 * LIMITA;
13   cout << "x = " << x << ", y = " << y << ", z = " << z
14        << endl;
15
16   #undef LIMITA // Eliminarea definiției macro-ului LIMITA
17   #define LIMITA 20 // Redefinirea macro-ului LIMITA cu o
18                 //nouă valoare, 20
19
20   // Utilizarea macro-ului LIMITA cu noua sa valoare în calcule
```

```
21  x = 20 - LIMITA;
22  y = 20 + LIMITA;
23  z = 3 * LIMITA;
24  cout << "Dupa redefinire LIMITA:" << endl;
25  cout << "x = " << x << ", y = " << y << ", z = " << z
26      << endl;
27
28  return 0;
29 }
```

Convenția adoptată în programare este aceea de a denumi macrourile folosind exclusiv litere majuscule. Aceasta facilitează identificarea rapidă a macrourilor în codul sursă. De asemenea, este important de observat că, atunci când se folosesc directivele *#define* sau *#undefine*, nu se adaugă punct și virgulă la finalul instrucțiunii.

Utilizarea macrourilor pentru valorile care apar în mod repetat într-un program este o practică recomandată. Această abordare facilitează procesul de modificare ulterioară a codului, permițând modificarea valorii într-un singur loc, eliminând astfel necesitatea de a căuta și de a schimba valoarea în multiple locuri din program contribuind la eficientizarea dezvoltării software și reducerea riscului de erori.

4.2. Directiva #include

Directiva *#include* în limbajul C/C++ este folosită pentru a include conținutul unui alt fișier în programul sursă înainte de compilare. Acest mecanism este esențial pentru separarea codului în fișiere diferite, facilitând gestionarea codului și respectiv reutilizarea codului sursă. Există două moduri de a folosi directiva #include pentru a specifica fișierul ce urmează a fi inclus în proiect:

- ***#include*** *"file":* când se utilizează ghilimele, preprocesorul va căuta mai întâi fișierul în directorul curent (directorul în care se află fișierul sursă care conține directiva #include). Dacă fișierul

nu este găsit, preprocesorul va căuta apoi în directoarele standard în care sunt stocate fișierele de antet (header files).

- ***#include*** *<file>:* când se utilizează parantezele unghiulare, preprocesorul va căuta fișierul direct în directoarele standard în care sunt stocate fișierele de antet.

Exemple utilizare directiva #include:

#include <cmath> // Include biblioteca *cmath*, oferind acces la funcții matematice precum sqrt(), pow(), sin(), cos() etc.

#include <fstream> // Include biblioteca pentru lucru cu fișiere, permițând crearea, citirea, și scrierea în fișiere.

#include "myHeader.h" // Include fișierul header definit de utilizator, "myHeader.h".

#include "IErrorHandler_cq22cc.h" // Include fișierul header definit de utilizator, " IErrorHandler_cq22cc.h".

4.3. Directivele #if , #ifdef și #ifndef

Compilarea condiționată este un instrument puternic în limbajul C++ care permite includerea sau excluderea porțiunilor de cod din procesul de compilare, folosind anumite directive speciale. Printre directivele cele mai utilizate pentru compilarea condiționată se numără *#if, #else, #elif și #endif*. Aceste directive sunt evaluate în timpul fazei de preprocesare a compilării unui program C++ și vă permit să compilați selectiv anumite părți din codul sursă al programului dumneavoastră. *Directiva #if* permite includerea condiționată a codului. Dacă expresia constantă care urmează după *#if* este evaluată ca adevărată, codul dintre *#if* și *#endif* va fi compilat, în caz contrar, blocul de instrucțiuni va fi ignorat. Directiva *#endif* indică sfârșitul blocului condiționat. Forma generală a unei directive *#if* este următoarea:

#if *expresie-constantă*

secvență de instrucțiuni

#endif

Se va lua următorul exemplu:

```
1  #include <iostream>
2  using namespace std;
3
4  #define NUMAR_MAXIM_UTILIZATORI 64
5
6  int main() {
7    #if NUMAR_MAXIM_UTILIZATORI > 18
8    cout << "Sunt permisi mai mult de 18 utilizatori.\n";
9    #endif
10
11   // ...
12
13   return 0;
14 }
```

În exemplul de mai sus, expresia *#if NUMAR_MAXIM_UTILIZATORI > 18* va fi evaluată ca fiind adevărată, pentru că valoarea definită pentru *NUMAR_MAXIM_UTILIZATORI* este 64, care este mai mare decât 18. Astfel, mesajul "*Sunt permisi mai mult de 18 utilizatori*." va fi afișat când programul va rula. Dacă condiția #if nu este îndeplinită, se poate utiliza #else pentru a specifica un bloc de cod alternativ care să fie compilat, sau #elif pentru a verifica o altă condiție.

Directiva #else și varianta sa condiționată #elif oferă o structură similară instrucțiunilor decizionale *if* și *else*, diferența principală fiind că acestea sunt evaluate în etapa de preprocesare, înainte de compilarea propriu-zisă a codului. Pentru fiecare #if este necesară o directivă #endif corespunzătoare, însă pentru #else și #elif nu este necesar o directivă #endif separată.

Directivele #ifdef și #ifndef, care înseamnă "*dacă este definit*" și respectiv "*dacă nu este definit*", și se referă la numele macroinstrucțiunii.

Sintaxa generală a directivei #ifdef este:

#ifdef nume_macro

secvență de instrucțiuni

#endif

Dacă numele macroinstrucțiunii a fost definit anterior cu ajutorul directivei #define, atunci secvența de instrucțiuni dintre #ifdef și #endif va fi compilată.

Sintaxa generală a directivei #ifndef este:

#ifndef nume_macro

secvență de instrucțiuni

#endif

Dacă un macro nu este definit anterior printr-o directivă #define, atunci blocul de instrucțiuni delimitate de #ifndef și #endif va fi inclus la compilare. Directivele #ifdef și #ifndef pot fi utilizate împreună cu #else sau #elif. De asemenea, directivele #ifdef și #ifndef pot fi imbricate în același mod ca directivele #if.

În exemplul de mai jos, folosim directiva #ifdef pentru a verifica dacă NUMAR_MAXIM_UTILIZATORI este definit prin intermediul directivei #define. Dacă este, blocul de cod corespunzător va fi compilat și executat. Dacă NUMAR_MAXIM_UTILIZATORI nu este definit, se va executa codul din blocul #else. Pentru a testa comportamentul directivei #ifndef, puteți comenta linia #define NUMAR_MAXIM_UTILIZATORI 64 și să utilizați linia de cod #undef NUMAR_MAXIM_UTILIZATORI, ceea ce va face ca

NUMAR_MAXIM_UTILIZATORI să nu fie definit și astfel codul din blocul #else va fi executat.

```
1  #include <iostream>
2  using namespace std;
3
4  #define NUMAR_MAXIM_UTILIZATORI 64
5
6  //De asemenea, puteti comenta linia de mai sus si
7  //utilizati urmatoarea linie pentru a testa comportamentul
8  //cu #ifndef
9  //#undef NUMAR_MAXIM_UTILIZATORI
10
11 int main() {
12   #ifdef NUMAR_MAXIM_UTILIZATORI
13   // Codul aici va fi compilat doar daca NUMAR_MAXIM_UTILIZATORI
14 este definit
15   if (NUMAR_MAXIM_UTILIZATORI > 18) {
16     cout << "Sunt permisi mai mult de 18 utilizatori.\n";
17   } else {
18     cout << "Nu sunt permisi mai mult de 18 utilizator.\n";
19   }
20   #else
21   // Codul aici va fi compilat doar daca
22   //NUMAR_MAXIM_UTILIZATORI nu este definit
23   cout << "Numarul maxim de utilizatori nu este defini.\n";
24   #endif
25
26   // Codul aici va fi compilat indiferent de definitiile anterioare
27
28   return 0;
29 }
```

4.4. Directiva #error

Această directivă oprește procesul de compilare dacă se produce eroarea specificată ca parametru. În cadrului exemplului de mai jos, compilarea se oprește dacă constanta __cplusplus nu este definită. Sintaxa directivei #error este: #error *message*.

Exemplu:

```
1 #ifndef __cplusplus
2 #error Un compilator C++ este necesar
3 #endif
```

Directiva #error este folosită, de obicei, în cadrul unei instrucțiuni preprocesor condiționale de tip #if. În următorul exemplu dacă constanta *MY_FUNCTION* este definită, atunci mesajul de eroare definit de directiva #error va fi afișat în timpul procesului de compilare.

Exemplu:

```
1 #if defined MY_FUNCTION
2 #error MY_FUNCTION este deja definită
3 #endif
```

4.5. Directiva #pragma

În C++, sintaxa acestei directive este #pragma *name*, unde *name* reprezintă comanda specifică sau acțiunea dorită ce trebuie executată de către compilator.

Directiva #pragma în C++ este definită de implementare, conform standardelor ANSI. Însemnând că comportamentul și utilizarea sa pot varia în funcție de compilatorul specific utilizat. Scopul principal al directivei #pragma este de a oferi un mod standardizat de a transmite comenzi speciale compilatorului, care pot influența compilarea programului în diverse moduri.

Exemplu:

```
1  #include <iostream>
2  using namespace std;
3
4  int display();
5
6  #pragma startup display
7  #pragma exit display
8
```

```
9  int main() {
10     cout << "\nI am in main function";
11     return 0;
12 }
13
14 int display() {
15     cout << "\nI am in display function";
16     return 0;
17 }
```

4.6. Exerciții utilizare directive preprocesor

1. Creați un program care solicită un număr întreg de la utilizator. Definiți un macro care calculează valoarea absolută a acestui număr. Folosiți macro-ul creat pentru a afișa valoarea absolută a numărului citit.

2. Dezvoltați un program care citește valori de tip double și în funcție de macroinstrucțiunea definită, se numără fie valorile pozitive, fie pe cele negative. Definiți macroinstrucțiunea COUNTER_POSITIVE astfel încât programul să numere valorile pozitive. Dacă COUNTER_POSITIVE nu este definită, programul va număra valorile negative.

Capitolul 5

Instrucțiuni de decizie

5.1. Instrucțiunea if

Instrucțiunea decizională if permite executarea selectivă a unei instrucțiuni sau a unui bloc de instrucțiuni bazată pe evaluarea unei expresii logice. Dacă rezultatul expresiei este adevărat (true), atunci instrucțiunea sau blocul de instrucțiuni definit se execută [11], [17]. Dacă rezultatul este fals (false), atunci instrucțiunea sau blocul de instrucțiuni nu se execută.

În tabelul de mai jos este prezentată sintaxa pentru variantele de scriere a instrucțiunii *if*, Tabel 5.1:

Tabel 5.1. Instrucțiunea decizională if

Sintaxă instrucțiune	Descriere	Pseudocod
if (*expresieLogică*) instrucţiune;	Execută instrucțiunea dacă expresia este adevărată.	DACĂ *expresieLogică* ADEVĂRATĂ ATUNCI execută instrucțiune; SFÂRȘIT DACĂ

if (*expresieLogică*) {
bloc de instrucțiuni
}
	Execută blocul de instrucțiuni dacă expresia este adevărată.	DACĂ *expresieLogică* ADEVĂRATĂ ATUNCI
		execută blocul de instrucțiuni;
		SFÂRȘIT DACĂ

if (*expresieLogică*) instrucțiune; else instrucțiune;	Execută prima instrucțiune dacă expresia este adevărată, altfel execută a doua instrucțiune.	DACĂ *expresieLogică* ADEVĂRATĂ ATUNCI execută prima instrucțiune; ALTFEL execută a doua instrucțiune; SFÂRȘIT DACĂ

Sintaxă instrucțiune	Descriere	Pseudocod

if (*expresieLogică*) {
    bloc de instrucțiuni
} else {
    bloc de instrucțiuni
}
	Execută blocul de instrucțiuni dacă expresia este adevărată, altfel execută al doilea bloc de instrucțiuni.	DACĂ *expresieLogică* ADEVĂRATĂ ATUNCI
		  execută primul bloc de instrucțiuni;
		ALTFEL
		  execută al doilea bloc de instrucțiuni;
		SFÂRȘIT DACĂ

if (*expresieLogică*)
    instrucțiune;
else {
    bloc de instrucțiuni
}
	Dacă expresia este adevărată, execută instrucțiunea, altfel execută blocul de instrucțiuni.	DACĂ *expresieLogică* ADEVĂRATĂ ATUNCI
		  execută instrucțiune;
		ALTFEL
		  execută blocul de instrucțiuni;
		SFÂRȘIT DACĂ

if (*expresieLogică*) {
    bloc de instrucțiuni
} else instrucțiune;
	Execută blocul de instrucțiuni dacă expresia este adevărată, altfel execută instrucțiunea.	DACĂ *expresieLogică* ADEVĂRATĂ ATUNCI
		  execută primul bloc de instrucțiuni;
		ALTFEL execută instrucțiune;
		SFÂRȘIT DACĂ

5.2. Instrucțiunea switch

Instrucțiunea switch este utilizată pentru a construi un bloc de instrucțiuni, iar punctul de unde începe execuția depinde de valoarea unei expresii având valori întregi.

Sintaxa instrucțiunii este următoarea:

```
1  switch (expresie) {
2  case constanta_1:
3   // Bloc de cod pentru valoare1
4   break;
5
6  case constanta_2:
7   // Bloc de cod pentru valoare2
8   break;
```

```
9
10  // Alte cazuri...
11 default:
12  // Bloc de cod care se execută dacă niciunul din  cazurile de mai
13 sus nu este adevărat
   }
```

În cadrul sintaxei instrucțiunii *switch* prezentate anterior:

- ***expresie*** este evaluată o singură dată la începutul instrucțiunii *switch*;
- ***case*** specifică valorile constante pe care expresie le poate avea, și blocul de cod corespunzător fiecăreia;
- ***break*** este folosit pentru a ieși din instrucțiunea switch, prevenind astfel executarea cazurilor următoare;
- **default** este un caz opțional care se execută atunci când niciun alt caz (case) nu corespunde valorii expresiei evaluate.

5.3. Exerciții

1. Să se scrie un program care verifică dacă un număr introdus de la tastatură este par sau impar.
2. Să se scrie un program care citește de la tastatură trei numere întregi: x, y și z. Programul trebuie să determine și să afișeze valoarea maximă dintre acestea.
3. Să se scrie un program care citește de la tastatură un număr întreg. Programul trebuie să calculeze și să afișeze dacă scriind numărul în oglindă, valoarea obținută este un număr care conține cel puțin două cifre. (De exemplu, nrIntreg = 340, scris în oglindă este 43, și se poate observa că este format din două cifre).
4. Creați un program care realizează un calculator simplu, capabil să efectueze doar patru operații de bază: adunare, scădere, înmulțire și împărțire. Programul va citi două valori de tip double și un operator (+, -, *, /) de la utilizator. În funcție de operatorul introdus, programul va calcula și afișa rezultatul corespunzător.

Asigurați-vă că programul gestionează cazurile speciale, cum ar fi împărțirea la zero.

5. Realizați o aplicație care determină calificativul unui student la examen. Aplicația acceptă ca date de intrare 3 note citite de la tastatură: nota la colocviul de laborator (în intervalul: 1 - 10), nota la parțial (în intervalul: 1 - 10), nota la activitatea de curs (în intervalul: 1 - 10). La final programul va determină nota finală pe baza următoarelor reguli:

    a) dacă media dintre cele 3 note este mai mare ca 9, atunci calificativul este A.

    b) dacă media dintre cele 3 note este >= 7 și < 9, atunci calificativul este B.

    c) dacă media dintre cele 3 note este >= 5 și < 7, atunci calificativul este C.

    d) dacă media dintre cele 3 note este < 5, atunci calificativul este F.

Capitolul 6

Instrucțiuni de ciclare (repetitive)

Procesul de repetiție, cunoscut și sub numele de ciclu în programare, este un element fundamental în dezvoltarea software. De exemplu, într-un sistem ce are ca scop monitorizarea și reglarea temperaturii, se impune verificarea la anumite momente de timp (ciclic) a temperaturii locale și comandarea proceselor de răcire/încălzire pentru a menține temperatura în anumite valori stabilite. Instrucțiunile repetitive sunt esențiale pentru a executa un bloc de instrucțiuni de mai multe ori, atâta timp cât o condiție logică de control rămâne adevărată. Condiția este evaluată prin intermediul unei variabile, cunoscută ca variabilă de control al ciclului. Ciclul se încheie atunci când condiția nu mai este îndeplinită, și este crucial ca programatorul să asigure existența unei condiții de ieșire pentru a evita crearea unei buclele infinite.

Limbajele de programare C/C++ oferă următoarele instrucțiuni repetitive: *for, while și do while*. Fiecare dintre aceste instrucțiuni are caracteristici specifice și se pot utiliza în funcție de cerințele programului pentru a facilita implementarea ciclurilor [16].

6.1. Instrucțiunea for

Forma generală a instrucțiunii *for* în C/C++ este următoarea:

for(*exp1*; *exp2*; *exp3*)

{

/* un bloc de instrucțiuni, care este executat repetitiv atâta timp cât valoarea lui *exp2* este adevărată. */ }

Cele trei expresii, *exp1, exp2 și exp3*, pot fi orice expresii valide în C/C++. Execuția instrucțiunii *for* implică următorii pași:

1. *exp1* este executată doar o singură dată. De obicei, *exp1* inițializează o variabilă folosită în celelalte două expresii.
2. valoarea *exp2* este condiția de control a ciclului care se evaluează înainte de executarea blocului de instrucțiuni. De obicei, este o condiție relațională. Dacă este falsă, ciclul se încheie și execuția programului continuă cu instrucțiunea/instrucțiunile ce preced instrucțiunea *for*. Dacă este adevărată, blocul de instrucțiuni din ciclu este executat.
3. *exp3* este o expresie ce se executată la finalul execuției blocului de interacțiuni din corpul *for*-ului. De obicei, aceasta modifică valoarea unei variabile folosite în *exp2*.
4. *Pașii 2 și 3* sunt repetați până când *exp2* devine falsă.

Dacă corpul instrucțiunii *for* are o singură instrucțiune, acoladele {} pot fi omise. Următorul program folosește instrucțiunea *for* pentru a afișa numerele de la 1 la 10:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5  for (int i = 1; i < 11; i++)
6    cout << i << endl;
7
8  return 0;
9 }
```

În exemplul precedent, *exp1* corespunde procesului de declarare și inițializare a variabilei contor *i*: *int i = 0*. În pasul al doilea, se verifică condiția de reluare a ciclului (*exp2*), *i < 11*. Dacă această condiție este îndeplinită (adevărată), ciclul continuă cu executarea blocului de instrucțiuni din corpul instrucțiunii *for*. În pasul al treilea, după executarea blocului de instrucțiuni, variabila contor (*exp3*),

reprezentată în acest exemplu de variabila *i*, este incrementată înainte de o nouă verificare a condiției de ciclare (*exp2*. Astfel, ciclul *for* se va executa de 10 ori, afișând numerele de la 1 la 10.

Instrucțiunea *for* este fundamentală în limbajele C/C++, permițând programatorilor să execute un set de instrucțiuni în mod repetitiv și controlat, oferind eficiență (mai puțină memorie ocupată) și claritate în cod, Figura 6-1.

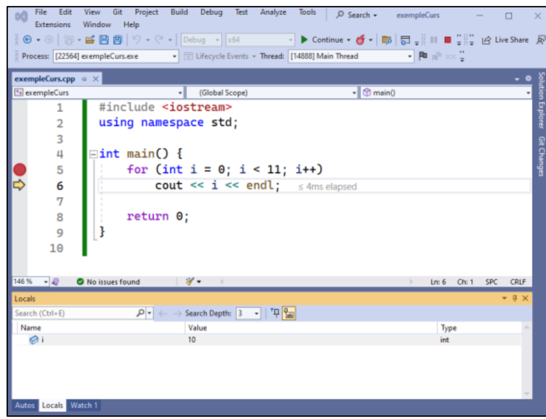


Figura 6-1. Exemplu utilizare instrucțiune for

Cele trei părți ale instrucțiunii *for* sunt opționale. De fapt, C/C++ ne permite să omitem oricare dintre acestea sau chiar pe toate. În următorul exemplu, deoarece variabila *a* este inițializată înainte de instrucțiunea *for*, prima expresie poate fi omisă:

```
1  int a = 0;
2  for (; a < 5; a++)
```

După cum se poate observa, chiar dacă prima expresie lipsește, caracterul punct și virgulă trebuie să fie prezent înaintea celei de a doua expresie, acționând ca delimitator între expresii De fapt, cele două caractere de tip punct și virgulă trebuie să fie întotdeauna prezente chiar dacă expresiile lipsesc.

Dacă condiția de reluare a ciclului lipsește (*exp2*), ciclul se va executa la infinit, deoarece nu există o condiție care să oprească ciclul.

În exemplul următor, prin intermediul instrucțiunii for se creează un ciclu infinit: `for (a = 0;  ; a++).`

În programare, dacă se omit cele trei expresii în definirea instrucțiunii for se generează un ciclu infinit. Iată un exemplu: for (; ;). Programatorii utilizează astfel de bucle infinite atunci când doresc, ca o porțiune de cod să ruleze până când o condiție sau un eveniment extern determină terminarea buclei Cu toate acestea, este esențial să includem un mecanism (cum ar fi o instrucțiune break) în interiorul buclei, în caz contrar, codul va continua să ruleze la infinit.

6.2. Instrucțiunea while

Instrucțiunea *while* permite crearea unei structuri repetitive (ciclice) condiționate anterior. Corpul ciclului poate fi executat o dată, de mai multe ori sau niciodată. Sintaxa sa este următoarea:

```
1  while (expresie) {
2    // bloc de instrucțiuni (corpul buclei)
3  }
```

Similar instrucțiunilor *if* și *for*, dacă corpul buclei are o singură instrucțiune, acoladele pot fi omise. Atunci când se execută o instrucțiune *while*, condiția de control al ciclului (e.g. expresie) este evaluată inițial. Dacă condiția este falsă, bucla *while* nu este executată. Dacă condiția este adevărată, corpul buclei este executat, ca apoi să se verifice din nou condiția de ciclu. Dacă condiția devine falsă, bucla *while* se termină.

În caz contrar, corpul buclei este executat din nou. Acest proces se repetă până când condiția de control (expresie) devine falsă. Comparativ cu instrucțiunea ciclică *for*, în cazul instrucțiunii ciclice *while*, programatorul trebuie să aibă grijă să modifice variabila ce controlează condiția de control în corpul buclei *while*, pentru a preveni generarea unei bucle infinite. De exemplu, următorul program utilizează instrucțiunea *while* pentru a afișa numerele între 10 și 1:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int i = 10;
6    while (i >= 1) {
7      cout << i << endl;
8      i--;
9    }
10   return 0;
11 }
```

Dacă valoarea expresiei de control (condiției) este întotdeauna adevărată, bucla devine infinită. Exemplu:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5   while (1) {
6     cout << "Bucla infinita" << endl;
7   }
8   return 0;
9 }
```

În cele ce urmează este prezentat un alt exemplu pentru a sublinia conceptele de bază ale utilizării instrucțiunii *while* în dezvoltarea aplicațiilor:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int contor = 1;
6    while (contor <= 3) {
7      cout << "Valoarea contorului: " << contor << endl;
8      contor++;
9    }
10   cout << "Programul s-a terminat." << endl;
11
12   return 0; }
```

Modul de funcționare al programului este următorul:

1. Variabila 'contor' primește inițial valoarea 1.
2. Condiția 'contor <= 3' este evaluată și considerată adevărată.
3. Deoarece condiția este adevărată, se execută corpul ciclului:

   a. Valoarea curentă a variabilei '*contor*' este afișată: "Valoarea contorului: 1"

   b. Variabila '*contor*' este incrementată cu o unitate folosind operatorul '*contor++*', astfel încât devine 2.
4. Condiția '*contor <= 3*' este din nou evaluată și considerată adevărată.
5. Corpul ciclului este din nou executat:

   a. Valoarea curentă a variabilei '*contor*' este afișată: "Valoarea contorului: 2"

   b. Variabila '*contor*' este incrementată cu o unitate și devine 3.
6. Condiția '*contor <= 3*' este evaluată din nou ca adevărată.
7. Corpul ciclului se execută din nou:

   a. Valoarea curentă a variabilei '*contor*' este afișată: "Valoarea contorului: 3"

   b. Variabila '*contor*' este incrementată cu o unitate și devine 4.
8. Condiția '*contor <= 3*' este acum evaluată ca falsă.
9. Deoarece condiția este falsă, programul sare peste corpul ciclului.
10. Se afișează "Programul s-a terminat."

Pentru a utiliza un ciclu *while*, programatorii trebuie să fie atenți la următoarele aspecte:

1. Inițializarea variabilelor utilizate în ciclu: Este important să inițializăm variabilele utilizate în ciclu înainte de intrarea în corpul ciclului. În exemplul de mai sus, inițializăm variabila '*contor*' cu 1.
2. Condiția de oprire a repetării corpului ciclului: Condiția specificată în instrucțiunea *while* determină când ciclul se va încheia. În exemplul nostru, ciclul continuă atât timp cât variabila '*contor*' este mai mare sau egală cu 3. Este important să alegeți o condiție care să permită întreruperea ciclului în momentul potrivit.

3. Modificarea variabilei în corpul ciclului: În interiorul buclei, trebuie să modificăm variabila (sau variabilele) care sunt folosite în condiția de oprire. În exemplul nostru, utilizăm operatorul '++' pentru a incrementa variabila '*contor*' cu 1 la fiecare iterație. Această modificare este crucială pentru a evita un ciclu infinit.

Controlul ciclului poate fi realizat în mai multe moduri, inclusiv prin folosirea unei variabile contor (vezi exemplu de mai sus), prin intermediul unei variabile de tip fanion care poate întrerupe ciclul, printr-o variabilă calculată în ciclu sau prin includerea unui apel de funcție care poate returna valoarea true (adevărat) sau false (fals) în condiția testată.

6.3. Instrucțiunea do-while

Spre deosebire de instrucțiunile *for* și *while*, în cazul cărora condiția (expresia testată) este testată înainte de executarea corpului buclei, instrucțiunea *do-while* testează condiția după fiecare executare a corpului buclei. Prin urmare, o buclă *do-while* este executată cel puțin o dată.

Sintaxa instrucțiunii *do-while* este următoarea:

```
1 do {
2   /* bloc de instrucțiuni (corpul buclei) care este executat cel puțin o
3 dată și apoi repetat atâta timp cât condiția este adevărată. */
4 } while (expresie);
```

Mai întâi se execută corpul buclei, apoi se evaluează condiția testată (e.g. *expresie*). Dacă condiția testată este falsă, bucla se termină. În caz contrar, corpul buclei este executat din nou, iar condiția este retestată. Dacă condiția devine falsă, bucla se termină. Contrar, corpul buclei se execută din nou. Acest proces se repetă până când condiția devine falsă. Instrucțiunea *do-while* trebuie să se termine cu caracterul punct și virgulă.

În cadrul exemplului de mai jos, se utilizează instrucțiunea *do-while* pentru a afișa numerele întregi de la 1 la 10 în ordine crescătoare:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5   int i = 1;
6   do {
7     cout << i << endl;
8     i++;
9   } while (i <= 10);
10
11  return 0;
12 }
```

6.4. Exemple suplimentare instrucțiuni de ciclare

Primul exemplu constă într-un program ce calculează factorialul unui număr pozitiv n. Factorialul n (notat n!) este produsul tuturor numerelor întregi pozitive de la 1 la n.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5   long n, fact = 1;
6
7   cout << "Introduceti n: ";
8   cin >> n;
9
10  if (n >= 0) {
11   if (n == 0) {
12     cout << "n! = 1" << endl;
13   } else { // Calculam factorialul
14     while (n > 1) {
15       fact = fact * n;
16       n = n - 1;
17     }
```

```
18      cout << "n! = " << fact << endl; // Afisam rezultatul
19    }
20  } else {
21    cout << "n trebuie să fie >= 0." << endl; // Mesaj de eroare pentru n
22 negativ
23  }
24
25  return 0;
26 }
```

Regăsiți mai jos un alt exemplu în care se utilizează cicluri suprapuse. În programare, uneori este necesar să avem bucle (cicluri) în interiorul altor bucle. Această structură de cod, în care un ciclu este inclus în corpul altui ciclu, este cunoscută sub numele de cicluri suprapuse.

Programul solicită utilizatorului să introducă numărul de linii și numărul de caractere pe linie. Apoi, utilizează două bucle *for* pentru a afișa caracterele ''pe numărul de linii specificat, fiecare linie conținând numărul specificat de caractere''.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int nrLinii, nrCaracterePeLinie;
6    // Cerem utilizatorului sa introduca numarul de linii si numarul de
7  caractere pe linie
8    cout << "Introduceti numarul de linii: ";
9    cin >> nrLinii;
10   cout << "Introduceti numarul de caractere pe linie: ";
11   cin >> nrCaracterePeLinie;
12   // Utilizam doua bucle "for" pentru a afisa caracterele "*" pe numarul
13 de linii specificate de utilizator
14   for (int i = 0; i < nrLinii; i++) {
15     for (int j = 0; j < nrCaracterePeLinie; j++) {
16       cout << "*";
17     }
```

```
18    cout << endl;
19   }
20
21   return 0;
22 }
```

Cel de al treilea exemplu calculează și afișează valorile funcțiilor sinus (sin), și respectiv cosinus pentru unghiul introdus de utilizator și permite continuarea introducerii altor unghiuri în grade până când utilizatorul răspunde cu "n" sau "N".

```
1  #include <iostream>
2  #include <cmath>
3  #include <cctype>
4  using namespace std;
5
6  int main() {
7    char rasp;
8    double x;
9
10   do {
11     cout << "Introduceti x (grade) : ";
12     cin >> x;
13
14     // Calculam si afisam valorile sin si cos pentru unghiul dat
15     double radians = 3.14159 * x / 180.0;
16     double sinValue = sin(radians);
17     double cosValue = cos(radians);
18
19     cout << "sin(x)  = " << sinValue << endl;
20     cout << "cos(x)  = " << cosValue << endl;
21
22     cout << "Mai continuati? (d/n) ";
23     cin >> rasp;
24   } while (toupper(rasp) == 'D');
25
26   return 0;
27 }
```

6.5. Exerciții

1. Care este output-ul următorului program când este executat?

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5  int countdown = 10;
6  int sum = 0;
7
8  cout << "Countdown from " << countdown << " to 1:" << endl;
9  for (int i = countdown; i >= 1; i--) {
10   cout << i << " ";
11   sum += i;
12  }
13
14  cout << "\nSum of numbers from " << countdown << " to 1: "
15       << sum << endl;
16  return 0;
17 }
```

2. Analizați și corectați programul de mai jos știind că funcționalitatea acestuia este să numere câte valori introduse de utilizator sunt pozitive și respectiv câte valori sunt negative. Intervalul de valori este cuprins între [300, 900]. Programul se va termina când utilizatorul introduce valoarea zero.

```
1 #include <cctype>
2
3 int main() {
4  int num, positive = 0, negative = 0, inRangeCount;
5
6  do {
7   cout << "Enter a number: ";
8   cin >> num;
9
10   if (num > 0) {
```

```
11       positive++;
12       if (num >= 300 && num <= 800)
13         inRangeCount++;
14     } else if (num < 0)
15       negative--;
16   } while (num != 0)
17
18   cout << "Positive numbers = " << positive << " Negative numbers = "
19 << negative << " Numbers in range [300, 900] = " << inRangeCount <<
20 endl;
21
22   return 0;
23 }
```

3. Scrieți un program care citește numărul de studenți dintr-o grupă și notele lor la un test. Programul trebuie să afișeze media notelor, nota cea mai mică și respectiv nota cea mai mare obținută, precum și câți studenți au obținut nota cea mai mare. Se presupune că notele sunt numere întregi de la 1 până la 10.

4. Se citesc de la tastatură două numere întregi, a și b (unde a < b, a > 1 și b < 100). Programul trebuie să afișeze toate numerele din intervalul [a, b] care au exclusiv divizori proprii pari.

Capitolul 7

Tablouri de elemente și pointeri

7.1. Noțiuni introductive tablouri (arrays) și șiruri de caractere

Un tablou unidimensional, sau vector, este o colecție de date de același tip, grupate sub un singur nume. Fiecare valoare din tablou este un element al tabloului și este accesată folosind un index. Elementele unui tablou sunt stocate în memorie unul după celălalt, în ordine. Indexarea în C/C++ ***începe de la 0***, astfel dacă considerăm un vector format din n elemente, primul element are indexul 0, al doilea indexul 1, și așa mai departe până la ultimul element care va avea indexul n-1 [16].

Declararea unui tablou implică specificarea tipului de date al elementelor, urmată de numele tabloului și numărul de elemente între paranteze drepte. Exemplu: int v[22] pentru un vector de numere întregi, float v[22]; pentru un vector de numere reale, char sir[22] pentru un șir de caractere. Elementele tabloului pot fi inițializate în momentul declarării sau ulterior, folosind instrucțiuni ciclice pentru a parcurge tabloul.

Exemplu de inițializare tablou la declarare: int v[10] = {22, 14, 17, 12, 25, 3, 2, 1, 0, 9};.

Este important de reținut că elementele unui array sunt accesate specificând numele tabloului și între paranteze drepte indexul elementului.

Exemplu:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    // Declararea și inițializarea vectorului cu 10 elemente
6    int v[10] = {22, 14, 17, 12, 25, 3, 2, 1, 0, 9};
7
8    // Accesarea și afișarea primului element (index 0)
9    cout << "Primul element (v[0]): " << v[0] << endl;
10
11   // Accesarea și afișarea celui de-al cincilea element (index 4)
12   cout << "Al cincilea element (v[4]): " << v[4] << endl;
13
14   return 0;
15 }
```

Un șir de caractere este un tip special de tablou unidimensional, unde fiecare element este un caracter și ultimul caracter este întotdeauna nul (‚\0'), semnalând sfârșitul șirului [11]. De exemplu, dacă declarați un șir de caractere astfel: char sir[6] = „array" acesta va fi stocat în memorie ca {‚a', ‚r', ‚r', ‚a', ‚y', ‚\0'}. Caracterul nul la sfârșitul șirului este esențial pentru a indica sfârșitul șirului de caractere în limbajul C. În exemplul de mai jos putem observa modul de declarare și inițializare a unui șir de caractere care poate conține 10 caractere. Figura 7-1 prezintă modul de stocare în memorie a șirului de caractere.

Exemplu - Declararea și inițializarea unui șir de caractere:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
6    // Declararea și inițializarea unui șir de caractere cu 10 elemente
7    char cuvant[10] = "array";
8    return 0;}
```

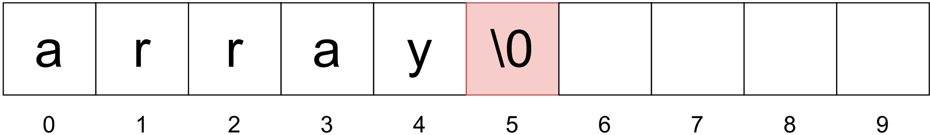


Figura 7-1. Modul de reprezentare al unul șir de caractere în memorie

După cum se poate observa, șirul de caractere declarat anterior poate stoca până la 10 octeți care sunt indexați de la 0 la 9. Cu toate acestea, doar primii 5 octeți sunt alocați cu caracterele cuvântului “array”, iar cel de-al șaselea octet va fi mereu ocupat de către caracterul nul (\0). Astfel, atunci când dorim să inițializăm un șir de caractere, trebuie mereu să avem grijă să îl includem si pe acesta.

Pentru lucrul cu șiruri de caractere, limbajul C++ dispune de biblioteca <cstring>. Această bibliotecă oferă o mare varietate de funcții ce permit manipularea șirurilor de caractere. Pentru a folosi aceste funcții, trebuie inclusă biblioteca menționată anterior folosind directiva #include <cstring>. În cele ce urmează, vom explica câteva astfel de funcții împreună cu un exemplu sugestiv.

Funcția strlen returnează lungimea efectivă a șirului de caractere fără a lua în considerare caracterul nul de la finalul șirului.

Exemplu pentru funcția strlen:

```
1 char sir_caractere[50] = "array";
2 int lungime_sir = strlen(sir_caractere);
3 cout << lungime_sir; //programul va afișa 5
```

Funcția strcpy primește ca parametrii două șiruri de caractere, primul fiind șirul destinație, iar cel de-al doilea șirul sursă. Funcția copiază șirul sursă în șirul destinație sau cu alte cuvinte funcția simulează operația de atribuire clasică.

Exemplu pentru funcția strcpy:

```
1 char sir_sursa[50] = "primul sir de caractere";
2 char sir_destinatie[50] = "al doilea sir de caractere";
```

```
3 strcpy(sir_destinatie, sir_sursa);
4 cout << sir_sursa << endl; //programul va afișa primul sir de caractere
5 cout << sir_destinatie; //programul va afișa primul sir de caractere
```

Funcția strcat acceptă doi parametrii, un șir de caractere sursă și un șir de caractere destinație. Această funcție concatenează cele două șiruri de caractere.

Exemplu pentru funcția strcat:

```
1 char sir_sursa[50] = "sir1";
2 char sir_destinatie[50] = "sir2";
3 string rezultat = strcat(sir_destinatie, sir_sursa);
4 cout << rezultat << endl; //programul va afișa sir2sir1
```

7.2. Concepte de bază legate de pointeri

Un pointer este o variabilă specială care nu stochează o valoare, ci o adresă de memorie [9], [10]. Declararea unui pointer nu înseamnă alocarea unei zone de memorie în care pot fi stocate date. Un pointer este tot un tip de date, a cărui valoare este un număr ce reprezintă o adresă de memorie. Spre exemplu, o adresa este de forma 0x7ffeed5749d0. Sintaxa pentru declararea unui pointer este: *tip *nume_pointer*.

Exemple de declarare și utilizare a pointerilor:

```
1 int i = 17, j = 3; // declararea variabilelor
2
3 int * p; // declararea unui pointer la int
4
5 p = & i; // atribuirea adresei variabilei 'i' pointerului 'p'
```

Operatorii unari utilizați cu pointeri:

- & - Operator de adresă: returnează adresa de memorie a unei variabile.
- * - Operator de dereferențiere: accesează conținutul zonei de memorie la care pointerul indică.

Operații cu pointeri în C/C++

1. **Accesarea valorilor:**
    - Utilizând ***nume_pointer*** pentru a accesa valoarea la care pointerul ***nume_pointer*** indică. Acest proces este cunoscut ca dereferențierea unui pointer.
    - Folosind ***nume_pointer[index]*** pentru a accesa un element specific într-un array la care ***nume_pointer*** indică zona de început pentru array.
2. **Obținerea adresei unui pointer:**
    - Folosind ***&nume_pointer*** se va obține adresa de memorie a pointerului ***nume_pointer*** însuși, nu a valorii la care acesta indică.
3. **Operațiile cu pointeri:**
    - **Incrementarea/decrementarea:** schimbarea adresei la care pointerul ***nume_pointer*** indică cu o valoare egală cu dimensiunea tipului de date referit.

Pointerii manipulează adrese de memorie, ceea ce necesită atenție sporită în utilizare. Utilizarea incorectă a pointerilor poate duce la erori grave, cum ar fi accesarea memoriei nealocate sau coruperea datelor.

7.3. Exemple tablouri de elemente și pointeri

În continuare vom prezenta un exemplu de lucru cu pointeri și vectori. Secvența de cod este prezentată mai jos, iar în Figura 7-2 este prezentat raționamentul utilizat atunci când se folosesc pointeri pentru a modifica.

Exemplu:

```
1 #include <iostream>
2 using namespace std;
3
```

```
4  int main() {
5    int vector[5]; // declararea unui vector cu 5 elemente
6    int * p; // declararea unui pointer de tip int
7    p = vector; // atribuim pointerului p adresa primului element din vector
8    * p = 17; // atribuim valoarea 17 pointerului p
9    p++; // mutăm pointerul p la următoarea adresă a vectorului
10   * p = 6; // atribuim valoarea 6 pointerului p
11   p = & vector[2]; // mutăm pointerul p la adresa cu indexul 2 din vector
12   * p = 3; // atribuim valoarea 3 pointerului p
13   p = vector + 3; // mutăm pointerul p la adresa cu indexul 3 din vector
14   * p = 47; // atribuim valoarea 47 pointerului p
15   p = vector; // atribuim pointerului p adresa primului element din vector
16   *(p + 4) = 79; // atribuim valoare 79 la adresa cu indexul 4 din vector
17   * p = 123; // atribuim valoarea 123 pointerului p
18
19   return 0;
20
21 }
```

Primul pas ilustrat în Figura 7-2 este declararea unui vector gol care poate stoca 5 elemente. După cum se poate observa din comentarii, indexarea elementelor din vector începe de la 0 până la 4. La pasul al doilea declarăm un pointer de tip întreg (eng. pointer to integer), îi atribuim valoarea primei adrese din vector, apoi îi asignăm valoarea 17 folosind operatorul de dereferențiere. De reținut este faptul că prin definiție, numele unui vector este un pointer la primul său element.

La pasul al treilea, mutăm adresa pointerului p la adresa corespunzătoare indexului următor și atribuim acestuia valoarea 6. La pasul 4 atribuim adresa de memorie de la poziția a doua din vectorul p și îi asignăm valoarea 3. La pasul 5, mutăm pointerul p la adresa celui de-al patrulea element din vector (adresa primului element din vector + 3) și îi atribuim valoarea 47. Asemănător procedăm și la pasul al cincilea, doar că de data aceasta pointerul p pointează inițial către

adresa primului element din vector, apoi atribuim valoarea 79 ultimului element din vector (adresa primului element + 4).

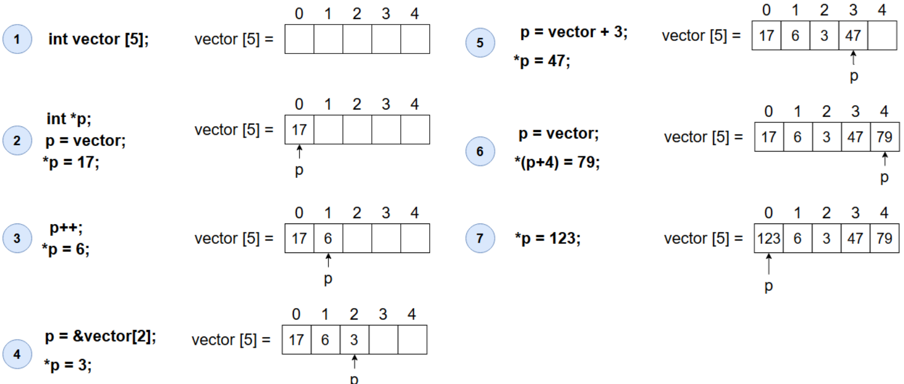


Figura 7-2. Manipularea vectorilor folosind pointeri

În ultimul pas se alocă valoare 123 pe poziția primului element din vector. Astfel, după executarea codului prezentat în exemplul de mai sus, vectorul va avea următoarea formă: vector[5] = {123, 6, 3, 47, 79}.

În continuare, vom prezenta un exemplu de interschimbare a doua variabile de tip întreg folosind pointeri.

Exemplu - Interschimbarea a două variabile folosind pointeri:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5
6    int a = 3, b = 4, aux;
7    int * ptr_a = & a;
8    int * ptr_b = & b;
9
10   cout << "Inainte de interschimbare" << endl;
11   cout << a << " " << b << endl;
12
13   aux = * ptr_a;
14   * ptr_a = * ptr_b;
15   * ptr_b = aux;
```

```
16
17   cout << "După de interschimbare" << endl;
18   cout << a << " " << b << endl;
19
20   return 0;
21 }
```

Aspectul care merită menționat în exemplul de mai sus este faptul că atunci când dorim să concepem algoritmi ce utilizează pointeri, trebuie să îi inițializăm cu adresa variabilelor cu care dorim să lucrăm. Astfel, *prt_a* și *ptr_b* stochează adresa variabilelor *a* și *b* folosind operatorul &. Apoi, algoritmul de interschimbare folosește operatorul de dereferențiere pentru a accesa valoarea acestora. Pentru a observa mai bine fiecare pas al exemplelor menționate anterior, este indicat ca acestea să fie rulate pas cu pas utilizând debugger-ul.

7.4. Exerciții

1. Realizați o aplicație care citește de la tastatură un șir de caractere și afișează vocalele: (A, E, I, O, U) conținute în șir despărțite printr-un spațiu. Dacă șirul de caractere nu are vocale, se va afișa un mesaj.

2. Realizați o aplicație care să inverseze un șir de caractere:
   a) fără a folosi pointeri.
   b) folosind pointeri.

3) Creați o aplicație care sortează un vector de numere întregi în ordine crescătoare folosind pointeri. Metoda de sortare folosită va fi Bubble Sort.

4) Creați un program care realizează suma elementelor unui vector folosind pointeri.

5) Ce afișează programul următor?

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5   int v[4] = {5, 2, 89, 1};
6   int * p = v;
7   p++;
8   p++;
9   * p = 16;
10  p = & v[0];
11  p = v + 2;
12  * p = 83;
13  * p = 123;
14
15  for (int i: v)
16    cout << i << ' ';
17
18  return 0;
19 }
```

Capitolul 8

Structura programelor mari

8.1. Lucrul cu Fișiere și Operații I/O în C/C++

Operațiile de intrare-ieșire (I/O) sunt esențiale pentru a permite unui program de calculator să interacționeze cu mediul extern [7], [8]. Aceste operații se efectuează prin utilizarea fișierelor care sunt stocate pe diverse suporturi și pot fi transferate utilizând echipamente periferice. Operațiile de I/O sunt variate și complexe. Ele implică atât componentele hardware precum memoria și dispozitivele periferice (de exemplu tastatura și ecranul), cât și elementele software, incluzând secvențe de cod dedicate conversiei și transferului de date.

Limbajele de programare C, C++ sau Java nu au instrucțiuni dedicate pentru gestionarea intrărilor și ieșirilor. Motivul fiind complexitatea și asigurarea compatibilității în contextul diversității echipamentelor și componentelor software utilizate.

Un fișier se poate defini ca un șir de octeți localizați pe un suport de stocare. Octeții pot fi transferați prin intermediul echipamentelor periferice. Fișierele de intrare conțin seturi de date ce urmează a fi prelucrate. Prin prelucrarea acestor date, se generează un fișier de ieșire populat cu seturi de rezultate. Fișierele de ieșire stochează rezultatele obținute în urma prelucrării datelor. Aceste rezultate pot fi utilizate ulterior ca date de intrare pentru alte programe.

Programele dezvoltate au capacitatea de a citi sau de a scrie secvențe de octeți de la sau către periferice, cum ar fi tastatura și

ecranul, sau de a gestiona fișiere stocate pe unitățile de stocare de tip hard-disk.

8.2. Operații de intrare-ieșire în C/C++

Pentru a utiliza funcțiile disponibile din bibliotecile standard ale limbajelor C și C++, acestea trebuie inițial incluse. În limbajul C, pentru realizarea de operații standard de intrare-ieșire, se include biblioteca "stdio.h" prin directiva "#include <stdio.h>". Această bibliotecă conține declarații pentru funcții esențiale, precum *printf* pentru afișare și *scanf* pentru citire. Pe de altă parte, în C++, utilizăm bibliotecile "*iostream*" și "*fstream*" pentru operațiuni similare. Biblioteca "*iostream*" conține declarații pentru funcții esențiale, precum 'std::cin' pentru citire și 'std::cout' pentru afișare, în timp ce '#include <fstream>' este folosită pentru lucrul cu fișiere.

Datorită simplității sale, în exemplele din această carte vom folosi funcțiile de intrare și ieșire standard, definite în C++. Aceste funcții sunt definite în biblioteca "*iostream*" și sunt accesibile fără declarații suplimentare, facilitând interacțiunea cu utilizatorul prin intermediul tastaturii și ecranului [5].

Citirea de la tastatură: Tastatura este tratată ca fișierul standard de intrare în C++. Identificatorul asociat acestui fișier este "*cin*", care este o prescurtare de la "*console in*". "*cin*" este utilizat în combinație cu operatorul de extracție ">>" pentru a prelua datele introduse de utilizator. Nu este necesară declararea explicită a fișierului standard de intrare "*cin*", deoarece este definit automat în cadrul spațiului de nume "*std*".

Scrierea pe ecran: Analog, ecranul este tratat ca fișierul standard de ieșire. Identificatorul asociat este *'cout'*, prescurtat de la "*console out*". *'cout'* este utilizat cu operatorul de inserție '<<' pentru a afișa date pe ecran. La fel ca '*cin*', '*cout*' este disponibil implicit și nu necesită declarare separată.

Exemplu:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5   int a, b, c;
6   cout << "Introduceti valorile a, b si c:" << endl;
7   cin >> a >> b >> c;
8   cout << "a+b+c = " << a + b + c << endl;
9   return 0;
10 }
```

Citirea de la tastatură este adesea realizată valoare cu valoare. Totuși, este posibil să citim simultan mai multe valori într-o singură instrucțiune. Pentru a separa valorile introduse în timpul execuției programului, ne putem folosi de următorii separatori: spațiu, ENTER sau tab.

8.2.1 Citirea dintr-un fișier înregistrat pe un hard-disk

Citirea dintr-un fișier stocat pe un mediu de stocare de tip hard-disk în C++ se realizează cu ajutorul bibliotecii *"<fstream>"*. Pentru a asocia un fișier cu un identificator, se folosește declarația: "*fstream fisier*". Pentru a deschide un fișier, se apelează metoda *"open()",* specificând calea fișierului și modul de accesare. După finalizarea lucrului cu fișierul, acesta se închide utilizând metoda "*close()*". Se pot utiliza operatorii >> și << pentru a citi din sau a scrie în fișiere, similar cu "*cin*" și "*cout*".

Exemplu:

```
1 // Deschide fișierul pentru citire datelor
2 fisier.open("calea/fisierului.txt", ios::in);
3 // ... lucrul cu fișierul
4 fisier.close(); // Închide fișierul
```

Funcţia open() are doi parametri, primul este numele fișierului, iar al doilea este modul de exploatare, care poate fi:

- ios::in: Deschide fișierul pentru citire;
- ios::out: Deschide fișierul pentru scriere (șterge conținutul fișierului dacă acesta există);
- ios::app: Deschide fișierul pentru scriere, dar adaugă la sfârșitul acestuia (păstrează conținutul existent).

Exemplu:

```
1 fstream fisier;
2 fisier.open("calea/fisierului.txt", ios::in);
3 int nr;
4 double sold;
5 fisier >> nr >> sold;
6 fisier.close();
```

Pentru a verifica că s-a ajuns la capăt (sfârșit) de fișier , avem la dispoziție următoarele variante:

- funcţia *eof()*: returnează 1 (true) dacă s-a ajuns la sfârșitul fișierului;
- funcţia *good()*: returnează 1 (true) dacă se poate continua citirea.

Este important de menționat că funcțiile "*eof()*" și "*good()*" semnalează atingerea sfârșitului fișierului doar după efectuarea unei ultime citiri. De aceea, poate fi utilă apelarea funcției "*ipfx()*", care are prototipul *int ipfx(int nrCaractereMinim)*. Această funcție returnează 1 dacă s-au citit din fișier cel puțin *nrCaractereMinim* câmpuri sau 0 în caz contrar.

8.3. Exemple lucru cu fișiere

În folder-ul proiectului, există un fișier numit clienti.txt. Acest fișier conține numele, prenumele și vârsta clienților unei biblioteci online. Se solicită determinarea și afișarea numărului și listei abonaților majori, precum și numărul abonaților minori.

```
1  #include <fstream>
2  #include <iostream>
```

```
3  using namespace std;
4
5  int main() {
6   char nume[20], prenume[30];
7   int varsta, nrmaj = 0, nrmin = 0;
8   fstream cit;
9
10  // Deschide fișierul clienti.txt pentru citire
11  cit.open("clienti.txt", ios::in);
12
13  // Parcurge fișierul și citește datele clienților
14  while (cit.good()) {
15    cit >> nume >> prenume >> varsta;
16
17    // Verifică dacă s-au citit minim 3 câmpuri din fișier
18    if (cit.ipfx(3)) {
19      // Dacă citirea este validă, procesează datele
20      if (varsta >= 18) {
21        // Afișează numele clienților majori
22        cout << nume << " " << prenume << endl;
23        nrmaj++;
24      } else {
25        // Numără clienții minori
26        nrmin++;
27      }
28    }
29  }
30
31  // Închide fișierul
32  cit.close();
33
34  // Afișează numărul clienților majori și minori
35  cout << "Numarul clientilor majori: " << nrmaj << endl;
36  cout << "Numarul clientilor minori: " << nrmin << endl;
37
38  return 0;
39 }
```

8.4. Funcții din biblioteca limbajului C

Limbajul C include un set extins de funcții standard, esențiale. Funcțiile din C sunt organizate în biblioteci, fiecare cu fișiere header

(.h) corespunzătoare. În continuare se va prezenta o selecție a celor mai utilizate funcții din biblioteca standard a limbajului C:

- **Funcții matematice** (<math.h>): *sin*, *cos*, *tan*, *exp* etc.
- **Funcții de testare/modificare caractere** (<ctype.h>): *isalpha*, *isdigit*, *tolower*, *toupper*.
- **Funcții pentru șiruri de caractere** (<string.h>): *strlen*, *strcpy*, *strcat*, *strcmp*.
- **Funcții utilitare** (<stdlib.h>): *atof*, *atoi*, *rand*, *exit*, *system*.

1.Funcţii matematice (fișierul header *<math.h>*)

Aceste funcții efectuează calcule matematice de bază până la cele mai complexe:

Funcții trigonometrice:

- double sin(double x);
- double cos(double x);
- double tan(double x);
- double asin(double x);
- double acos(double x);
- double atan(double x);
- double atan2(double y, double x);
- double sinh(double x);

Funcţii exponenţiale şi logaritmice:

- double exp(double x);
- double log(double x);
- double log10(double x);
- double pow(double x, double y);
- double sqrt(double x);

Funcţii de rotunjire:

- double floor(double x);
- double ceil(double x);

Valori absolute:

- int abs(int x);
- long int labs(long int x);
- double fabs(double x);

2. Funcții de clasificare (testare) a clasei caracterelor (<*ctype.h*>)

Biblioteca "<*ctype.h*>" oferă funcții esențiale pentru testarea proprietăților caracterelor (cum ar fi dacă sunt litere, cifre, litere mică sau literă mare) și pentru conversia între majuscule și litere mici:

- int isalpha(int c);
- int isdigit(int c)
- int islower(int c);
- int isupper(int c);

Funcţii de conversie a caracterelor:

- int tolower(int c);
- int toupper(int c);

3. Funcţii de tratare a șirurilor de caractere (<*string.h*>):

- int strlen(char sir[]); // Returnează lungimea șirului "sir"
- strcpy(char dest[], char sursa[]); // Copiază șirul "sursa" în "dest"
  //Copiază primele n caractere din șirul "sursa" în șirul "dest"
- strncpy(char dest[], char sursa[ ], int n);
  //Concatenează șirul "sursa" la sfârșitul lui șirului "dest"
- strcat(char dest[], char sursa[]);
  /*Asemănătoare funcţiei strcat(), dar adaugă la sfârșitul șirului destinație cel mult n caractere din șirul "sursa".*/
- strncat(char dest[], char sursa[ ], int n);
  /* Compară şirurile de caractere sir1 şi sir2 şi returnează 0 dacă sir1 == sir2, o valoare negativă dacă sir1 < sir2 şi o valoare pozitivă dacă sir1 > sir2 */
- int strcmp(char sir1[], char sir2[]);
  /* Compară primele n caractere a șirului "sir1" și "sir2", furnizând o valoare similară funcției strcmp */
- int strncmp(char sir1[], char sir2[], int n);
  /* Caută prima apariție a caracterului "c" în sir și returnează un pointer la locația acestuia sau NULL dacă nu este găsit */
- char [] strchr(char sir[], char c);
  /* Caută ultima apariție a caracterului "c" în **"sir"** sau NULL dacă nu este găsit*/

- char [] strrchr(char sir[], char c);

  /* Caută subșirul "sir2" în șirul "sir1". Funcția returnează subșirul format prin preluarea caracterelor din "sir1" pornind din poziția în care apare "şir 2" în "şir 1"sau NULL dacă "şir 2" nu se regăsește în "şir 1". */
- char [] strstr(char sir1[], char sir2[]);

4. Funcţii utilitare (<*stdlib.h*>)

// Convertește șirul de caractere "s" într-un număr de tip double.

- double atof(char s[]);

// Convertește șirul de caractere "s" într-un număr întreg de tip int.

- int atoi(char s[]);

// Convertește șirul de caractere "s" într-un număr întreg de tip long.

- long atol(char s[]);

// Returnează un număr aleator între 0 și RAND_MAX.

- int rand(void);

// Întrerupe execuția programului, oprirea este anormală.

- void abort();

// Încheie execuția programului.

- void exit(int stare);

8.5. Crearea și utilizarea propriilor funcții

Funcțiile sunt esențiale în C/C++, prin intermediul acestora putem structura programele în secvențe mai mici, fiind astfel și mai ușor gestionat [21]. Să luăm un exemplu simplificat și anume procesul de fabricare a unei biciclete. Să presupunem că avem un atelier unde construim biciclete. Pentru fiecare bicicletă, avem nevoie de un cadru, două roți și o șa. Dacă avem un număr total de cadre, roți și șei, câte biciclete complete putem construi?

```
1 // Exemplu de funcție care calculează numărul de biciclete ce pot fi
2 asamblate
3 int biciclete(int roti, int cadre, int sa) {
```

```
4  // Calculează numărul de biciclete fabricate pe baza componentelor
5 disponibile
6 }
7
8 // Exemplu de apel al funcției
9 biciclete(4, 2, 2);
```

În C, o funcție este definită folosind următoarea sintaxă:

```
type functionName(parameter1, parameter2, ...) {
   // Corpul funcției }
```

- *type*: reprezintă tipul valorii returnate de funcție.
- *functionName*: reprezintă numele prin care funcția poate fi apelată.
- *parameters*: reprezintă parametrii funcției, fiecare având un tip și un nume.

Atunci când lucrăm cu funcții este esențial să înțelegem structura lor:

- **Antetul funcției:** Se specifică numele funcției, tipul valorii pe care o returnează funcția și parametrii pe care îi primește.
- **Declarații de variabile locale**: Aceste variabile sunt definite în cadrul funcției și sunt accesibile doar în interiorul acesteia (local).
- **Instrucțiuni:** Acestea reprezintă corpul funcției. Funcția se termină, de obicei, cu instrucțiunea "*return*", care termină execuția funcției și, opțional, returnează o valoare.

Să considerăm funcția "*adunare*", prin intermediul căreia vom ilustra conceptele menționate în subcapitolul de mai sus *Crearea și utilizarea propriilor funcții*.

```
1  #include <iostream>
2  using namespace std;
3
4  // Antetul funcției
5  int adunare(int a, int b) {
6    int r = a + b;
7    return r;
```

```
8 }
9
10 int main() {
11  // Apelarea funcției și afișarea rezultatului
12  int rezultat = adunare(5, 3);
13  cout << "Rezultatul este " << rezultat;
14 }
```

Funcția "*adunare*" este definită pentru a calcula suma a două numere întregi. Antetul funcției include doi parametri formali: *int a* și *int b*, care sunt numerele ce vor fi adunate. În acest caz, nu este necesar să adăugăm prototipul funcției, deoarece este definită înainte de funcția *main()*. Dacă funcția "*adunare*" ar fi definită ulterior utilizării funcției principale *main*, atunci ar fi necesar să adăugăm prototipul funcției la începutul programului, înainte de funcția *main*. Funcția adunare este apelată în funcția *main*. Aici, parametrii efectivi 5 și 3 sunt transmiși funcției adunare:

```
1 int main() {
2    // Apelarea funcției și afișarea rezultatului
3    int rezultat = adunare(5, 3);
4    cout << "Rezultatul este " << rezultat;
5
6    return 0;
7 }
```

La apelul funcției, valorile 5 și 3 sunt transmise prin valoare funcției adunare. Acest lucru înseamnă că funcția "*adunare*" lucrează cu copii ale valorilor 5 și respectiv 3

În C/C++, parametrii unei funcții pot fi transmiși fie prin valoare, fie prin referință. Transmiterea prin valoare (ca în cazul nostru) înseamnă că funcția primește copii ale valorilor parametrilor, iar modificările acestor copii nu afectează valorile originale. Pe de altă parte, transmiterea prin referință înseamnă că funcția lucrează direct cu adresele de memorie ale parametrilor, permițând modificarea valorilor originale.

Se va modifica exemplul de mai sus, astfel funcția "*adunare*" va primi în acest caz trei parametrii prin referință: void adunare(int& a, int& b, int& rezultat).

```
1  #include <iostream>
2  using namespace std;
3
4  // Modificarea antetului funcției pentru a primi parametrii prin referință
5  void adunare(int & a, int & b, int & rezultat) {
6    rezultat = a + b;
7    // Se vor modifica stocate de a și b pentru a ilustra efectul transmiterii
8  parametrilor prin referință
9    a += 10;
10   b += 20;
11 }
12
13 int main() {
14   int numar1 = 5;
15   int numar2 = 3;
16   int rezultat = 0;
17
18   // Apelarea funcției și afișarea rezultatului
19   adunare(numar1, numar2, rezultat);
20   cout << "Rezultatul adunarii este: " << rezultat << endl;
21   cout << "Valoarea modificata a lui numar1: " << numar1 << endl;
22   cout << "Valoarea modificata a lui numar2: " << numar2 << endl;
23 }
```

Adăugarea funcțiilor definite în fișiere de tip header, separate de fișierul în care avem programul principal, în mediul de dezvoltare Visual Studio, ajută la organizarea și modularizarea codului. Iată pașii ce trebuie parcurși pentru a adăuga funcțiile create în fișiere de tip header:

1. Crearea fișierului de tip header

a. În Visual Studio, se deschide proiectul în care doriți să adăugați fișierele de tip header.

b. Pentru a adăuga un fișier de tip header parcurgeți următorii pași: "Solution Explorer" -> Click dreapta pe numele proiectului sau a folderului corespunzător -> "Add" -> "New Item".
c. În fereastra "Add New Item" -> selectați "Header File (.h)" -> Denumiți fișierul header (de exemplu, **adunare.h**).
d. Adăugați fișierul creat în proiectului dvs. utilizând opțiunea "Add".

2. Definirea funcției în fișierul header

a. Deschide fișierul header creat anterior
b. Scrieți prototipul funcției sau funcțiilor create în fișierul header. De exemplu: "void adunare(int& a, int& b, int& rezultat)";

3. Definirea funcțiilor în fișierul sursă (.cpp)

a. Creați sau deschideți fișierul sursă: dacă nu aveți deja un fișier sursă (.cpp) pentru implementarea funcției, creați un nou fișier sursă.
b. Implementați funcția în fișierul sursă.

Exemplu:

```
1 #include "functii.h"
2
3 int adunare(int a, int b) {
4   return a + b;
5 }
```

4. Utilizarea funcției în cadrul programului

a. Se va include fișierul header în cadrul fișierul sursă (de exemplu, main.cpp sau alte fișiere funcție de context) unde se apelează funcția: `#include "functii.h"`.
b. Acum se va putea apela funcția "*adunare*" din fișierul "main.cpp" sau din orice alt fișier sursă unde s-a inclus fișierul header.

5. Compilarea și Testarea

a. Se compilează proiectul pentru a verifica că totul funcționează corect.

b. Se rulează aplicația și se testează funcționalitatea pentru a ne asigura că programul îndeplinește cerințele date.

Prin urmare parcurgând pașii de mai sus, se pot adăuga și utiliza funcții definite în fișiere de tip header în cadrul proiectele dvs utilizând mediul Visual Studio.

8.6. Conceptul de recursivitate

Recursivitatea este o tehnică fundamentală în programare, permițând unei funcții să se apeleze pe sine însăși. Această tehnică este deosebit de utilă pentru că problemele pot fi descompuse în subprobleme similare sau mai mici, furnizând frecvent o soluție mai elegantă și mai intuitivă decât buclele tradiționale [7], [8]. Câteva exemple de probleme care se pot rezolva prin intermediul recursivității ar fi: calcularea Celui Mai Mare Divizor Comun, generarea șirul lui Fibonacci.

Pentru a ilustra conceptul de recursivitate se propune calcularea factorialului unui număr *n*, $n! = n \times (n-1)!$. În C, putem calcula simplu și eficient factorialul utilizând recursivitatea:

```
1  #include <iostream>
2  using namespace std;
3
4  long factorial(int n);
5  double suma(int n, double a[]);
6
7  int main() {
8    long f;
9    double s;
10   double a[5] = {10., 33., -12., 1., 10.};
11   f = factorial(6);
12   s = suma(5, a);
13
14   return 0;
15 }
16
```

```
17 long factorial(int n) {
18   if (n == 1)
19     return 1;
20   else
21     return n * factorial(n - 1);
22 }
23
24 double suma(int n, double a[]) {
25   if (n == 0)
26     return a[0];
27   else
28     return a[n] + suma(n - 1, a);
29 }
```

Exemplul anterior ilustrează principiul recursivității prin apelul funcției "*factorial*" de către ea însăși. În momentul implementării apelului recursiv, este importantă stabilirea unei condiții de oprire, în cazul de față când *n* este 1.

8.7. Exerciții

1. Pentru un sistem de gestionare a cărților împrumutate din cadrul unei biblioteci, trebuie dezvoltată o aplicație care să permită calcularea taxei de penalizare aplicată de bibliotecă unui abonat dacă cartea sau cărțile împrumutate nu sunt returnate la termenul prevăzut.

Notă: Pentru a simplifica problema, aplicația se va dezvolta pentru un singur abonat/pentru o singură fișă de împrumut. Nu este necesar să precizăm datele abonatului.

Descriere: Cărțile împrumutate de un abonat se vor introduce prin intermediul unei funcții care va inițializa structura de cărți împrumutate (fișa de împrumut). Pentru cărțile pentru care se depășește perioada maximă de împrumut de 14 zile, se vor aplica penalizări. Penalizarea este de 5 Ron pe zi per carte.

Cerințe:

- Adăugarea cărților împrumutate de abonat: Pentru calcularea penalizărilor este necesar inițializarea fișei de împrumut asociată abonatului (detalii carte împrumutată, nr. de zile în care cartea a fost împrumutată). O carte este caracterizată de următoarele câmpuri: Titlu, Autor, Cotă Carte (valoare unică), Nr zile carte împrumutată. Inițializarea fișei de împrumut se face în cadrul funcției principale main().
- Calculul penalizărilor: Sistemul va include o funcție pentru calculul penalizărilor dacă se depășește perioada maximă de împrumut. Penalizarea este de 5 Ron pe zi per carte.
- Returnare carte: Sistemul va include o funcție pentru returnarea carte împrumutată. Cartea se va returna pe baza cotei asociate fiecărei cărți.
- Interfața utilizatorului: Dezvoltați o interfață simplă și intuitivă pentru utilizatori, care va include opțiuni pentru verificare penalizare, returnare carte.

Indicații:

- Codul ar trebui să fie bine organizat, folosind de exemplu o structură Carte pentru a reprezenta cărțile de pe fișa de împrumut. Aceasta va include câmpuri pentru titlul cărții, autor, cotă carte (val unică), precum și număr zile carte împrumutată. Pentru a crea fișa de împrumut este nevoie să folosiți tablou de structuri:
    - *Carte fisaImprumut[nrCartiImprumutate];*
    - *Carte* este structura descrisă în enunțul problemei.
- Asigurați-vă că gestionați corect cărțile returnate. Abonatul poate returna doar o carte pe care a împrumutat-o (identificată prin cota de carte). În caz contrar trebuie afișat un mesaj corespunzător.

Exemplu rulare:

Meniu:
1. Calcul penalizare
2. Returnare carte
3. Ieșire
Alegeți o opțiune: 1
Fișa abonatului include următoarele cărți împrumutate:
Matematica pentru toti, Ion Petrescu, COT002, 21 zile
Curs fizica, Ion Pop, COT003, 9 zile
Penalizare pentru întârziere este: 35 Ron
Meniu:
1. Calcul penalizare
2. Returnare carte
3. Ieșire
Alegeți o opțiune: 2
Introduceți cota aferenta cartii returnata: COT001
Cartea cu cota COT001 nu se regaseste in fisa abonatului.
Meniu:
1. Calcul penalizare
2. Returnare carte
3. Ieșire
Alegeți o opțiune: 2
Introduceți cota aferenta cartii returnata: COT002
Cartea cu cota COT002 a fost returnată.
Meniu:
1. Calcul penalizare
2. Returnare carte
3. Ieșire
Alegeți o opțiune: 1
Fișa abonatului include următoarele cărți împrumutate:
Curs fizica, Ion Pop, COT003, 9 zile
Penalizare pentru întârziere este: 0 Ron
Meniu:

1. Calcul penalizare
2. Returnare carte
3. Ieșire
Alegeți o opțiune: 3
Sesiunea de lucru s-a încheiat

Capitolul 9

Introducere în algoritmi fundamentali

În viața de zi cu zi, efectuăm diverse activități urmând niște pași bine definiți, asemenea unor algoritmi. Aceste activități pot include pregătirea pentru facultate, prepararea micului dejun, conducerea unei mașini etc. Pentru a realiza cu succes fiecare dintre aceste activități, urmăm o secvență de pași precisă.

Pentru a ilustra conceptul de algoritm, să luăm ca exemplu activitatea de conducere a unei mașini:

1. **Pregătirea mașinii pentru utilizare:** Verificați nivelul de combustibil, uleiul și starea anvelopelor.
2. **Ajustarea setărilor:** Așezați-vă în scaunul șoferului și reglați oglinzile și scaunul în poziția corectă.
3. **Pornirea mașinii:** În funcție de sistemul mașinii (cheie sau buton de pornire), porniți motorul.
4. **Selectarea treptei de viteză:** Dacă mașina are o transmisie manuală, selectați treapta de viteză potrivită. Pentru mașinile automate, puneți mașina în "Drive" sau "D" pentru a merge înainte.
5. **Conducerea mașinii:** Utilizați pedala de accelerație la pornire și pedala de frână pentru a reduce viteza sau pentru a opri mașina.
6. **Semnalizare și verificarea oglinzilor:** Pentru a efectua manevrele de schimbare a direcției sau de depășire, utilizați semnalele de direcție și verificați oglinzile pentru siguranță.
7. **Conducerea în siguranță:** Respectați regulile de circulație și semnalele de circulație pe tot parcursul călătoriei.
8. **Oprirea mașinii:** Când ajungeți la destinație, respectați procedura specifică mașinii dumneavoastră pentru a opri motorul.

Conducerea unei mașini poate fi comparată cu un algoritm În acest context, algoritmul reprezintă un set bine definit de pași pentru a realiza o sarcină specifică. În programare, algoritmii reprezintă pașii logici pe care calculatorul îi urmează pentru a rezolva anumite probleme și pentru a produce rezultatele dorite [3], [12], [13]. Algoritmul are un început și un sfârșit bine definit și constă dintr-un număr finit de pași.

Un algoritm este caracterizat de următoarele aspecte:

- Precizie - pașii sunt precis definiți, fără ambiguități.
- Unicitate - rezultatele fiecărui pas sunt clare și definite în mod unic.
- Finitudine - algoritmul se încheie întotdeauna după un număr finit de pași, evitând buclele infinite.
- Generalitate - algoritmul trebuie să fie capabil să rezolve probleme dintr-o întreagă clasă de probleme.
- Input (date de intrare) - algoritmul primește o valoare sau o mulțime de valori ca date de intrare necesare pentru procesare.
- Output (date de ieșire) - algoritmul produce un rezultat (valoare sau mulțime de valori) ca date de ieșire.

În continuare, vor fi prezentați următorii algoritmi: algoritmii de inițializare, algoritmul "Divide et Impera", algoritmi de sortare - Metoda "Bubble Sort", algoritmul de interclasare și respectiv diverși algoritmi de calcul cu mulțimi [12].

9.1. Algoritmi de inițializare

Realizarea unui set de date de test sub forma unui șir de valori, de obicei aleatoare, sau a unei matrice se întâlnește frecvent în aplicații.

Exemple algoritmi de inițializare:

a. Exemplu inițializare șir cu valori aleatoare produse prin apelul funcției rand().

```
1  #include <stdlib.h>
2  #include <iostream>
3  #include <time.h>
4  using namespace std;
5
6  int main(void) {
7   int i;
8   int a[20];
9
10   // Pornesc generatorul de numere aleatoare cu o valoare returnata de
11 time().
12   srand((unsigned) time(NULL));
13
14   // Creez cele 20 de valori aleatoare.
15   for (i = 0; i < 20; i++) {
16     a[i] = rand();
17     cout << a[i] << endl;
18   }
19
20   return 0;
21 }
```

b. Exemplu inițializare matrice unitară.

O astfel de matrice are toate elementele nule cu excepția celor de pe diagonala principală. Elementele de pe diagonala principală au proprietatea de a avea indicele de linie egal cu cel al coloanei.

```
1  #include <iostream>
2  using namespace std;
3
4  void unitate(double a[][10], int n);
5
6  int main(void) {
7   int n, i, j;
8   double a[10][10];
9   cout << "n= ";
10  cin >> n;
11
```

```
12  // Initializare matricea unitate
13  unitate(a, n);
14  for (i = 0; i < n; i++) {
15    for (j = 0; j < n; j++)
16      cout << a[i][j] << " ";
17    cout << endl;
18  }
19  return 0;
20 }
21
22 void unitate(double a[][10], int n) {
23  int i, j;
24  for (i = 0; i < n; i++)
25    for (j = 0; j < n; j++)
26      if (i == j) // elementele de pe diagonala principala
27        a[i][j] = 1.0;
28      else
29        a[i][j] = 0.0;
30 }
```

9.2. Algoritmul “Divide et Impera”

Metoda "Divide et Impera" denumită și "*Divide-and-conquer*" reprezintă o tehnică de rezolvare a problemelor complexe prin care problema inițială se împarte în subprobleme mai mici cu o structură similară problemei originale sau de aceeași natură. Subproblemele sunt rezolvate independent una față de alta, iar apoi soluțiile lor sunt combinate pentru a obține soluția finală.

Această tehnică aduce un beneficiu semnificativ în rezolvarea problemelor complexe, deoarece permite abordarea lor pas cu pas, reducând astfel complexitatea generală și facilitând dezvoltarea algoritmilor. Un exemplu practic de aplicare a acestei metode este algoritmul de căutare binară, Figura 9-1.

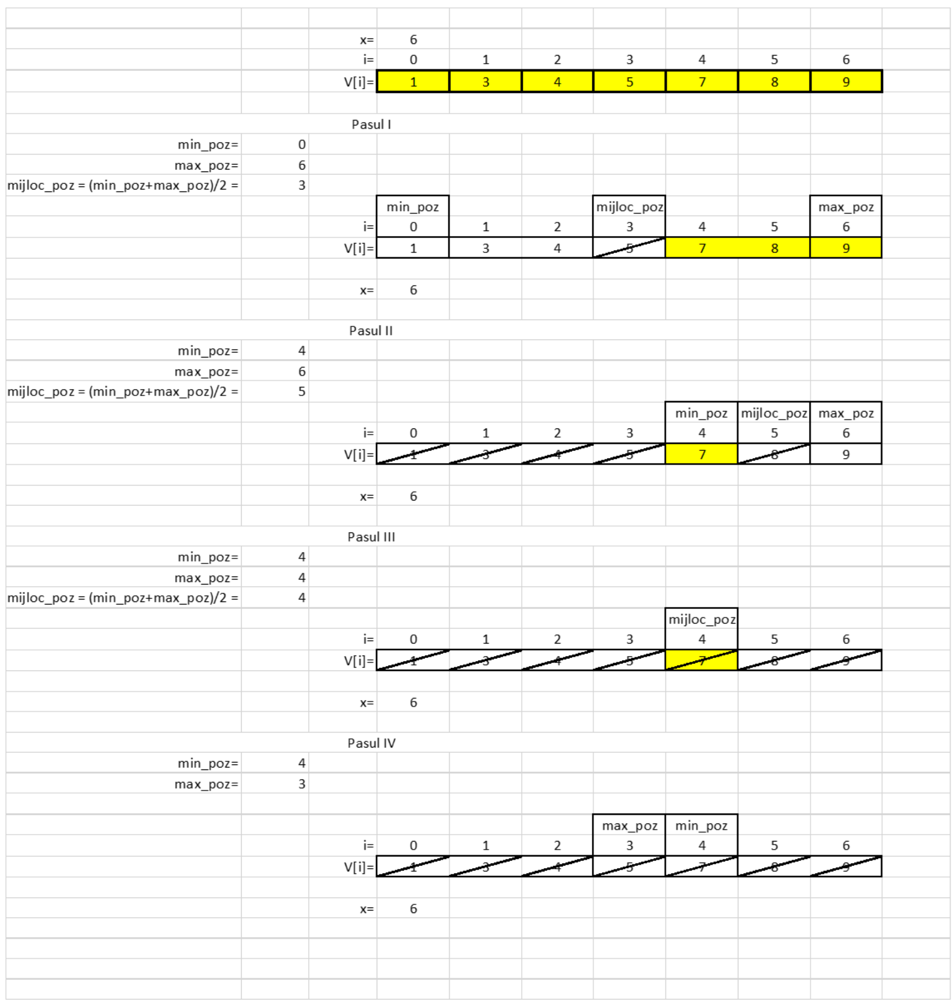


Figura 9-1. Illustrarea pașilor din cadrul algoritmului de căutare binară folosind metoda Divide et Impera

În exemplul următor, vom utiliza funcția *prezent()*. Scopul acestei funcții este să verifice dacă o valoare *x* se găsește într-un șir *v[]* ordonat crescător. Atunci când valoarea *x* este găsită în șir, funcția va returna poziția sa în cadrul șirului. Pe de altă parte, dacă valoarea *x* nu se regăsește în șir, funcția va returna -1.

```
1  #include <iostream>
2  using namespace std;
3
4  int prezent(int x, int v[], int n) {
5    int mini, maxi, mijloc;
```

```
6   mini = 0;
7   maxi = n - 1;
8
9   while (mini <= maxi) {
10    mijloc = (mini + maxi) / 2;
11    if (x < v[mijloc])
12      maxi = mijloc - 1;
13    else if (x > v[mijloc])
14      mini = mijloc + 1;
15    else
16      return mijloc;
17  }
18  return -1;
19 }
20
21 int main() {
22  const int n = 6;
23  int v[n] = {1, 3, 5, 7, 9, 11};
24  int x;
25
26  cout << "Introduceti elementul cautat: ";
27  cin >> x;
28
29  int rezultat = prezent(x, v, n);
30
31  if (rezultat == -1)
32    cout << "Elementul nu a fost gasit in vector." << endl;
33  else
34    cout << "Elementul a fost gasit la pozitia: " << rezultat << endl;
35
36  return 0;}
```

Explicații:

1. Algoritmul de căutare binară:
    - Algoritmul compară elementul căutat *x* cu valoarea de la mijlocul array-ului. Dacă valorile sunt egale, returnează poziția acestuia. Dacă *x* este mai mic decât elementul din mijloc, continuă căutarea în jumătatea inferioară a array-ului.

Dacă *x* este mai mare, căutarea continuă în jumătatea superioară.

2. Procesul de împărțire
    - La fiecare pas, algoritmul împarte intervalul de căutare în două, reducând astfel semnificativ numărul de comparații necesare. Aceasta este esența tehnicii "Divide et Impera", care descompune o problemă mare în subprobleme mai mici și mai ușor de gestionat.
3. Efectuarea căutării
    - Algoritmul se bazează pe comparații succesive pentru a determina în ce jumătate a intervalului curent ar trebui să continue căutarea.
4. Testarea algoritmului:
    - În funcția principală ***main()***, se definește un array sortat și se solicită utilizatorului să introducă un element pentru căutare.
    - Se apelează funcția ***prezent()*** pentru a determina dacă elementul este prezent în array și, dacă da, la ce poziție.
    - Rezultatul căutării este apoi afișat, indicând fie poziția elementului, fie că elementul nu a fost găsit.

9.3. Algoritmi de sortare – Metoda ”Bubble Sort”

Metoda bulelor, cunoscută și sub numele de *Bubble Sort*, este unul dintre cei mai simpli algoritmi de sortare [12]. Principiul său de bază este de a compara elemente adiacente într-un șir și de a le inversa dacă nu sunt în ordinea corectă, Figura 9-2. Procesul se repetă până când șirul devine complet sortat, Figura 9-3.

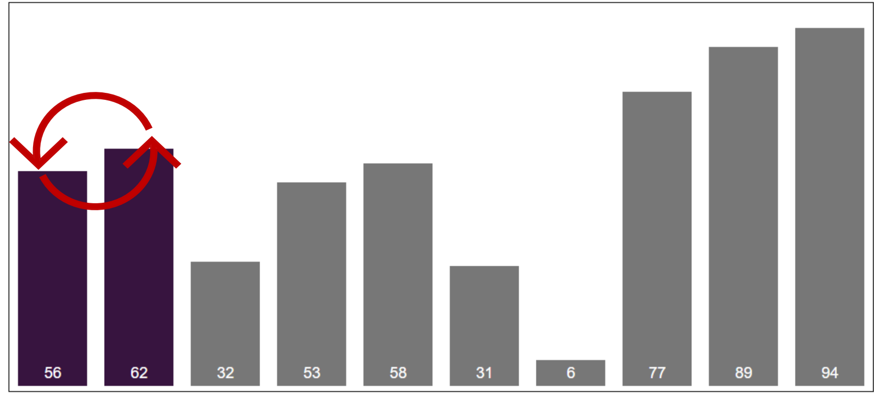


Figura 9-2. Principiu de sortare a metodei Bubble Sort

Funcționarea algoritmului Bubble Sort:

1. Parcurgerea șirului: Se parcurge șirul de la început până la sfârșit.
2. Compararea și inversarea: La fiecare pas, se compară fiecare pereche de elemente adiacente, două câte două, acestea fiind inversate dacă nu respectă ordinea corectă (ordine crescătoare sau descrescătoare).
3. Finalizarea unei iterații: Odată ce se ajunge la sfârșitul șirului, se finalizează o iterație/parcurgere completă.
4. Optimizarea iterațiilor: Procesul se repetă pentru fiecare element al șirului, dar cu fiecare iterație, numărul de elemente care trebuie verificate se reduce cu unu, deoarece "cel mai mare element" / "cel mai mic element" din șir ajunge pe poziția corectă.

5. Repetarea procesului: Acest proces se repetă până când nu mai este necesară inversarea elementelor (pe parcursul iterație), ceea ce indică faptul că șirul este complet sortat în ordinea corectă.

n = 5	0	1	2	3	4
	8	**5**	**7**	**6**	**2**
	5	8	7	6	2
	5	7	8	6	2
	5	7	6	8	2
	5	7	6	2	8
	5	7	6	2	8
	5	7	6	2	8
	5	6	7	2	8
	5	6	2	7	8
	5	6	2	7	8
	5	6	2	7	8
	5	2	6	7	8
	5	2	6	7	8
	2	5	6	7	8

Figura 9-3. Ilustrarea pașilor pentru sortarea în ordine crescătoare a șirului de numere: 8, 5, 7, 6, 2

În cele ce urmează se va prezenta un program care citește un șir numeric de *n* valori întregi și le ordonează crescător folosind metoda Bubble Sort.

```
1  #include <iostream>
2  using namespace std;
3
4  void ordon(int[], int);
5
6  int main() {
7   int n, a[20], i;
8   cout << "n = ";
9   cin >> n;
10  for (i = 0; i < n; i++) {
11    cout << "a[" << i << "]= ";
12    cin >> a[i];
13  }
14
15  ordon(a, n);
16  cout << "Sirul ordonat:" << endl;
17  for (i = 0; i < n; i++)
18    cout << "a[" << i << "]= " << a[i] << endl;
```

```
19
20   return 0;
21 }
22
23 void ordon(int a[], int n) {
24   int i, aux, fanion, k;
25   k = 0;
26   do {
27     fanion = 0;
28     for (i = 0; i < n - 1 - k; i++) {
29       if (a[i] > a[i + 1]) {
30         aux = a[i];
31         a[i] = a[i + 1];
32         a[i + 1] = aux;
33         fanion = 1;
34       }
35     }
36     k++;
37   } while (fanion);
38 }
```

Explicații:

1. Apelul funcției de sortare:
    - După citirea datelor, se apelează funcția *ordon()*.
2. Implementarea metodei Bubble Sort (funcția *ordon()*):
    - În cadrul funcției *ordon()*, se inițializează variabilele *i*, *aux*, *fanion* și *k*.
    - Variabila *fanion* este de tip logic și indică dacă s-au efectuat interschimbări după parcurgerea array-ului.
    - Se utilizează o buclă *do-while* care se repetă cât timp *fanion* este adevărat (1). La începutul fiecărei iterații, *fanion* este setat la 0 (fals).
3. Procesul de sortare:
    - Numărul de elemente verificate se reduce la fiecare iterație completă a buclei externe (*n - 1 - k*) pentru a optimiza procesul de sortare.

- Dacă un element este mai mare decât următorul (*a[i] > a[i + 1]*), acesta se interschimbă folosind variabila auxiliară *aux*. După o interschimbare, *fanion* este setat la 1 (adevărat) pentru a indica că a avut loc cel puțin o interschimbare.

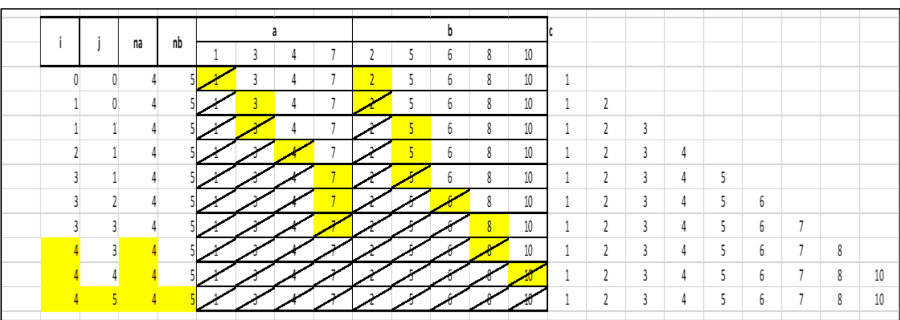
4. Optimizarea procesului de sortare:
    - După fiecare parcurgere completă a array-ului, elementul cel mai mare ajunge pe poziția corectă. Astfel, după prima parcurgere, elementul maxim ajunge la ultima poziție, iar după a doua parcurgere, maximul dintre primele n-1 elemente ajunge la penultima poziție, și așa mai departe. Deci, nu este necesar să se verifice aceste elemente la parcurgerile ulterioare.
    - Variabila *k* este incrementată după fiecare iterație a buclei *do-while*, ceea ce reduce numărul de elemente verificate în parcurgerile ulterioare.
5. Afișarea șirului sortat:
    - După finalizarea sortării (când *fanion* rămâne 0, semnalizând că nu mai sunt necesare interschimbări), șirul ordonat este afișat.

9.4. Algoritmul de interclasare

Pentru a înțelege algoritmul de interclasare a doi vectori, să luăm exemplu din Figura 9-4. În cadrul exemplului sunt dați doi vectori, "*a*" și "*b*", care sunt deja sortați în ordine crescătoare. Numărul de elemente din vectorul "*a*" este reprezentat prin "*na*", iar numărul de elemente din vectorul "*b*" este reprezentat prin "*nb*". Scopul este de a combina acești doi vectori într-un singur vector, "*c*", păstrând ordinea crescătoare a elementelor. Inițial se compară primul element din vectorul "*a*" cu elementul de pe poziția întâi din vectorul "*b*". Dacă primul element al vectorului "*a*" este mai mic, îl copiem în vectorul "*c*"

și apoi incrementăm indicele "*i*", pentru a parcurge următorul element din "*a*".

În cazul în care primul element din vectorul "*b*" este mai mic, se va copia în vectorul *"c"*, și incrementăm indicele "*j*", pentru a parcurge următorul element din "*b*". Astfel, prin comparații succesive, se construiește vectorul "*c*", până când toate elementele din vectorul "*a*" sau "*b*" sunt parcurse. Elementele rămase în vectorul "*a*" sau "*b*", în funcție de situație, sunt copiate în ordine în vectorul "*c*".



i	j	na	nb	a				b					c								
				1	3	4	7	2	5	6	8	10									
0	0	4	5	1	3	4	7	2	5	6	8	10	1								
1	0	4	5	1	3	4	7	2	5	6	8	10	1	2							
1	1	4	5	1	3	4	7	2	5	6	8	10	1	2	3						
2	1	4	5	1	3	4	7	2	5	6	8	10	1	2	3	4					
3	1	4	5	1	3	4	7	2	5	6	8	10	1	2	3	4	5				
3	2	4	5	1	3	4	7	2	5	6	8	10	1	2	3	4	5	6			
3	3	4	5	1	3	4	7	2	5	6	8	10	1	2	3	4	5	6	7		
4	3	4	5	1	3	4	7	2	5	6	8	10	1	2	3	4	5	6	7	8	
4	4	4	5	1	3	4	7	2	5	6	8	10	1	2	3	4	5	6	7	8	10
4	5	4	5	1	3	4	7	2	5	6	8	10	1	2	3	4	5	6	7	8	10

Figura 9-4. Ilustrarea pașilor din cadrul algoritmului de interclasare

Exemplu algoritm de interclasare:

```
1  #include <iostream>
2  using namespace std;
3
4  void inter(int a[], int b[], int rez[], int na, int nb);
5
6  int main() {
7   int na, nb, a[20], b[20], c[40], i;
8
9   cout << "na = ";
10  cin >> na;
11
12  for (i = 0; i < na; i++) {
13    cout << "a[" << i << "]= ";
14    cin >> a[i];
15  }
16
17  cout << "nb = ";
18  cin >> nb;
19  for (i = 0; i < nb; i++) {
```

```
20     cout << "b[" << i << "]= ";
21     cin >> b[i];
22   }
23   inter(a, b, c, na, nb);
24
25   cout << "Sirul rezultat:" << endl;
26   for (i = 0; i < na + nb; i++)
27     cout << "c[" << i << "]= " << c[i] << endl;
28
29   return 0;
30 }
31
32 void inter(int a[], int b[], int rez[], int na, int nb) {
33   int i = 0, j = 0;
34
35   do {
36     if (i < na && j < nb)
37       if (a[i] <= b[j]) {
38         rez[i + j] = a[i];
39         i++;
40       }
41     else {
42       rez[i + j] = b[j];
43       j++;
44     } else
45     if (i < na) {
46       rez[i + j] = a[i];
47       i++;
48     } else {
49       rez[i + j] = b[j];
50       j++;
51     }
52   } while (i + j < na + nb);
53 }
```

9.5. Algoritmi de calcul cu mulțimi

În cele ce urmează se urmărește exemplificarea unui algoritm de verificare a relației de incluziune, respectiv exemplificarea unui algoritm de verificare a relației de intersecție a două mulțimi date.

a. Algoritmul de verificare a relației de incluziune:

În exemplul de mai jos se consideră doi vectori, "*t1*" și "*t2*", reprezentând două mulțimi de numere. Algoritmul verifică dacă vectorul "*t2*" este un subset al lui vectorului "*t1*". Se definește funcția includ ce are ca parametrii cei doi vectori precizați, precum și numărul de elemente din fiecare vector (*na* pentru "*t1*" și *nb* pentru "*t2*"). În cadrul algoritmului, se parcurg ambii vectori, comparând fiecare element din "*t1*" cu toate elementele din "*t2*". Astfel, se determină dacă toate elementele din vectorul "*t2*" se găsesc în vectorul "*t1*".

```
1  #include <iostream>
2  using namespace std;
3
4  int includ(int a[], int b[], int na, int nb);
5
6  int main(void) {
7   int t1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
8   int t2[3] = {4, 5, 6};
9
10  if (includ(t1, t2, 10, 3))
11    cout << "t1 contine t2" << endl;
12
13  return 0;
14 }
15
16 int includ(int a[], int b[], int na, int nb) {
17  int i, j;
18  i = 0;
19  j = 0;
20  do {
21    if (a[i] == b[0])
22      for (j = 1; j < nb; j++) {
23        if (a[i + j] != b[j])
24          break;
25      };
26    if (j == nb)
27      return 1;
```

```
28
29     i++;
30   } while (i <= na - nb);
31
32   return 0;
33 }
```

b. Algoritmul de calcul a intersecției a două mulțimi:

În exemplul prezentat mai jos se consideră doi vectori, "*t1*" și "*t2*", care reprezintă două mulțimi de numere, și un al treilea vector, "*t3*", ce conține toate elementele comune a celor două mulțimi. Precondiție în cadrul exemplului: vectorii "*t1*" și "*t2*", să fie sortați în ordine crescătoare. Algoritmul compară elementele din vectorul "*t1*" și "*t2*" și adaugă în vectorul "*t3*" acele elemente care se găsesc în ambele mulțimi. Acest proces continuă până când toate elementele din vectorul "*t1*" sau "*t2*" sunt comparate, astfel vectorul "*t3*" va conține doar elementele comune a celor doi vectori menționați,"*t1*" și "*t2*".

```
1  #include <iostream>
2  using namespace std;
3
4  void inters(int a[], int b[], int rez[], int na, int nb, int & nrez);
5
6  int main() {
7    int t1[20] = {2, 4, 6, 8, 9, 10, 15, 15, 15, 15, 17};
8    int t2[20] = {1, 4, 5, 8, 11, 15, 15, 16, 18, 19};
9    int t[20], n, i;
10
11   inters(t1, t2, t, 10, 10, n);
12
13   for (i = 0; i < n; i++) {
14     cout << t[i] << " ";
15   }
16   cout << endl;
17
18   return 0; }
```

```
19
20 void inters(int a[], int b[], int rez[], int na, int nb, int & nrez) {
21   int i, j;
22   i = j = nrez = 0;
23
24   do {
25     if (a[i] < b[j])
26       i++;
27     else if (a[i] > b[j])
28       j++;
29     else {
30       rez[nrez] = a[i];
31       i++;
32       j++;
33       nrez++;
34     }
35   } while (i < na && j < nb);
36 }
```

Bibliografie

[1] P. Van der Linden, *Expert C programming : Deep C Secrets*. Pearson, 1994.

[2] D. S. Malik, *C++ Programming : Program Design Including Data Structures*. Boston: Cengage Learning, 2017.

[3] E. W. Dijkstra, *A discipline of programming*. New Jersey: Prentice-Hall, INC., 1976.

[4] M. Stevanovic, *Advanced C and C++ Compiling*. Apress, 2014.

[5] B. Stroustrup, *The C ++ Programming*, 4th ed. Pearson Education, Inc., 2013.

[6] H. Schildt, *C++: A Beginner's Guide*, 2nd ed. McGraw Hill Education, 2004.

[7] Y. Kanetkar, *Let Us C: Authentic Guide to C Programming Language*, 19th ed. BPB Publications, 2022.

[8] H. Schildt, *C++: The complete reference*, 4th ed. McGraw Hill Education, 2003.

[9] K. A. Reek, *Pointers on C*. Pearson, 1997.

[10] Y. P. Kanetkar, *Understanding pointers in C*, 3rd ed. BPB Publications, 2003.

[11] G. S. Tselikis and N. D. Tselikas, *C: From Theory to Practice* , 2nd ed. Boca Raton: Taylor & Francis Group LLC, 2017.

[12] R. Sedgewick, *Algorithms in C*. Addison-Wesley Pub. Co, 1990.

[13] M. T. Goodrich, R. Tamassia, and D. M. Mount, *Data Structures and Algorithms in C++*, 2nd ed. John Wiley & Sons, Inc., 2011.

[14] M. Dawson, *Beginning C++ through game programming*. Boston: Cengage Learning PTR., 2015.

[15] J. Raynard, *Update to Modern C++*. Independently published, 2023.

[16] Z. A. Shaw, *Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)*. Addison-Wesley Professional, 2016.

[17] H. Schildt, *Teach yourself C*, 3rd ed. McGraw-Hill Osborne Media, 1997.

[18] H. Schildt, *Teach Yourself C++*, 3rd ed. McGraw-Hill Osborne Media, 1998.

[19] D. Griffiths and D. Griffiths, *Head First C*. O'Reilly Media, Inc., 2012.

[20] Code Quickly, *Learn C++ Quickly*. Drip Digital, 2020.

[21] R. Grimm, *C++ Core Guidelines explained: Best Practices for Modern C++*. Addison-Wesley Professional, 2022.