Andreea-Mihaela COMȘIȚ

Cristina-Ioana CĂLIN

ELEMENTE INTRODUCTIVE ÎN ÎNVĂȚAREA LIMBAJULUI C++

Îndrumar de laborator

Editura Universității Transilvania din Brașov

2025

EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAȘOV

Adresa: Str. Iuliu Maniu nr. 41A
500091 Brașov
Tel.: 0268 476 050
Fax: 0268 476 051
E-mail: editura@unitbv.ro

Editură recunoscută CNCSIS, cod 81

ISBN 978-606-19-1810-2 (e-book)

Lucrarea a fost avizată de Consiliul Departamentului de Electronică și Calculatoare, Facultatea de Inginerie electrică și știința calculatoarelor a Universității Transilvania din Brașov.

Cuprins

Introducere

Într-o eră digitală în continuă expansiune, cunoașterea unui limbaj de programare devine o abilitate esențială. Limbajul C++ ocupă un loc important în acest context, fiind utilizat pe scară largă în dezvoltarea de aplicații software, jocuri video, sisteme embedded și multe alte domenii ale informaticii.

Această carte s-a născut din dorința de a oferi un ghid clar și structurat celor care doresc să învețe C++ de la zero sau să-și consolideze noțiunile deja acumulate. Fiecare capitol a fost conceput pentru a introduce gradual conceptele fundamentale ale limbajului C++, punând accent pe înțelegerea logicii de programare și pe aplicabilitatea practică.

Cartea se adresează în special elevilor, studenților și autodidacților, dar poate fi deopotrivă utilă și profesorilor care doresc un material bine organizat pentru predare. Am inclus exemple relevante, exerciții rezolvate și teme de lucru pentru a încuraja învățarea activă.

Această carte își propune să fie un ghid complet pentru învățarea limbajului de programare C++, de la elementele de bază până la concepte avansate. Scopul principal este să ofere o abordare logică, accesibilă și practică a programării în C++, astfel încât cititorul să poată construi programe funcționale și eficiente.

Structura cărții este următoarea:

- **Capitol 1:** Noțiuni introductive despre calculatoare
- **Capitol 2:** Limbajul de programare C++
- **Capitol 3:** Expresii aritmetice, apeluri de funcții și ieșiri

- **Capitol 4:** Condiții și expresii logice
- **Capitol 5:** Instrucțiuni
- **Capitol 6:** Subprograme (funcții)
- **Capitol 7:** Tablouri
- **Capitol 8:** Pointeri
- **Capitol 9:** Structuri
- **Capitol 10:** Programare orientată pe obiecte

Fiecare capitol include:

- Explicații teoretice clare
- Exemple de cod comentate
- Exerciții propuse și rezolvate
- Exerciții recapitulative

La finalul cărții, cititorul va avea nu doar o bază solidă în C++, ci și capacitatea de a scrie programe complexe, optimizate și corect structurate.

1. Noțiuni introductive despre calculatoare

Programarea unui calculator presupune ordonarea și structurarea pașilor necesari pentru a obține un anumit rezultat, folosind atât hardware-ul, cât și software-ul. Hardware-ul se referă la componentele fizice ale calculatorului, precum: procesorul, memoria, hard disk-ul, placa de bază și altele, care sunt esențiale pentru funcționarea sistemului. Software-ul este format din programele și aplicațiile care rulează pe calculator și care permit utilizatorului să interacționeze cu dispozitivul, să proceseze informații și să execute diverse sarcini.

Programarea este procesul prin care se creează software-ul, utilizând un limbaj de programare pentru a scrie instrucțiuni precise, care să controleze comportamentul calculatorului. Aceste instrucțiuni sunt ordonate într-o secvență logică și executate de procesor. Un program bine structurat poate rezolva probleme complexe, de la procesarea datelor și automatizarea sarcinilor până la crearea de jocuri video sau dezvoltarea de aplicații mobile.

Un alt aspect important este faptul că ordinea de execuție a instrucțiunilor poate afecta eficiența și corectitudinea programului. De exemplu, alegerea unei anumite metode de sortare a datelor sau de calculare a unui rezultat poate face ca programul să fie mai rapid sau mai lent, în funcție de algoritmii folosiți.

Programarea calculatorului se concentrează pe crearea unui set de instrucțiuni care să permită unui calculator să rezolve o problemă specifică. Fiecare program reprezintă o descriere detaliată a pașilor ce trebuie urmați

pentru a atinge un obiectiv, fie că este vorba despre efectuarea unor calcule matematice, procesarea unor date sau interacțiunea cu utilizatorii.

În esență, un program poate fi văzut ca un proces care definește ordinea și modul în care trebuie să se execute diverse acțiuni. De exemplu, atunci când se scrie un program care sortează o listă de numere, trebuie să se aleagă un algoritm (precum „algoritmul de sortare prin selecție" sau „algoritmul de sortare rapidă") și să se descrie pașii în codul sursă. Acesta va fi tradus într-un limbaj pe care calculatorul îl înțelege și îl poate executa pas cu pas.

Programarea calculatorului este un proces care implică mai multe etape:

- *analiza cerințelor:* se definește scopul programului, se identifică funcționalitățile necesare, se stabilesc constrângerile și specificațiile tehnice;
- *proiectarea (design):* se alege arhitectura software-ului, se creează diagrame de flux și modele de date, se alege limbajul de programare și tehnicile folosite;
- *scrierea codului (implementarea):* se scrie codul conform specificațiilor din etapa de proiectare, se respectă principiile de programare, se folosește controlul versiunilor (exemplu: Git);
- *testarea:* se testează unitățile individuale ale codului, se testează integrarea componentelor, se verifică performanța și securitatea;
- *depanare și optimizare:* se identifică și corectează erorile (debugging), se optimizează codul pentru eficiență și scalabilitate;

- *implementare și lansare:* se instalează și configurează aplicația pe sistemele utilizatorilor, se monitorizează performanța și se asigură că programul funcționează corect;

- *mentenanță și actualizare:* se corectează eventualele bug-uri apărute după lansare, se adaugă noi funcționalități și îmbunătățiri, se optimizează performanța în funcție de utilizare.

Aceste etape sunt aplicabile atât programării de aplicații simple, cât și dezvoltării de software complex. În plus, programarea poate fi realizată în diverse limbaje de programare (de exemplu: Python, Java, C++, C#) care au sintaxe și paradigme proprii, dar toate urmează aceleași principii logice fundamentale pentru a rezolva problemele.

1.1 Calculatorul

Un calculator este un dispozitiv electronic capabil să îndeplinească diverse sarcini prin executarea instrucțiunilor care îi sunt date sub formă de programe. Acesta constă în componente hardware și software care lucrează împreună pentru a procesa date, a stoca informații și a efectua calcule.

Un calculator este alcătuit din mai multe componente hardware și software care lucrează împreună pentru a permite realizarea diferitelor sarcini. Componentele unui calculator sunt:

- **unitatea centrală de procesare** (CPU) - cunoscută și ca “creierul calculatorului”. Aceasta execută instrucțiunile stocate în memoria calculatorului. Efectuează operații aritmetice, logice și de control, coordonând activitățile altor componente hardware.

- **memoria** - stochează date și instrucțiuni de programe pe care unitatea centrală trebuie să le acceseze rapid. Există mai multe tipuri de memorii:
    a. volatile - RAM
    b. nevolatile - hard disk (HDD), solid state drives (SSD) și flash drives.
- **dispozitive de intrare** - permite utilizatorilor să interacționeze cu calculatorul și să introducă date sau comenzi. Dispozitivele de intrare obișnuite includ tastaturi, mouse-uri, touchpad-uri, scanere și microfoane.
- **dispozitive de ieșire** - afișează informația procesată. Exemple: monitor, imprimantă, boxe.
- **sistem de operare** (OS) - este un program software care gestionează resursele hardware ale calculatorului și oferă o interfață cu utilizatorul pentru interacțiunea cu calculatorul.

  Exemple de sisteme de operare: Microsoft Windows, macOS, Linux și Android.

- **aplicații software** - cunoscute ca programe sau aplicații. Acestea sunt instrucțiuni care execută anumite sarcini pe calculator.

  Exemple: browser web, jocuri, media (muzică, poze)

- **placa de bază** - este componenta care conectează toate celelalte componente hardware ale calculatorului și le permite să comunice între ele. Ea găzduiește CPU-ul, memoria RAM și alte periferice. De asemenea, include și sloturi pentru plăci suplimentare.
- **placă grafică** (GPU - Graphics Processing Unit) - este responsabilă pentru procesarea și redarea imaginilor și grafică pe ecran. În timp ce

unele calculatoare au procesoare grafice integrate (care sunt parte din CPU), altele dispun de plăci grafice dedicate pentru performanțe superioare, esențiale în jocuri sau aplicații de editare grafică.

- **rețea și carduri de expansiune** - cardurile de rețea permit calculatorului să se conecteze la internet sau la alte dispozitive prin rețea (Wi-Fi, Ethernet). Cardurile de expansiune sunt adăugate pentru a extinde capacitățile sistemului, cum ar fi plăci suplimentare de sunet sau de stocare.
- **sistem de răcire** - este esențial pentru menținerea temperaturii calculatorului la un nivel sigur, prevenind supraîncălzirea componentelor. Acestea includ ventilatoare, radiatoare și, uneori, sisteme de răcire pe lichid.

Aceste componente hardware lucrează împreună pentru a permite calculatorului să execute programele și să rezolve problemele pentru care au fost configurate. Pe lângă hardware, există și software-ul care controlează și coordonează funcționarea acestora, creând o interfață între utilizator și mașină.

1.2 Limbaje de programare

Un limbaj de programare reprezintă un sistem formal de instrucțiuni și reguli utilizat pentru a interacționa cu un sistem de calcul și pentru a dezvolta programe software. Acesta permite programatorilor să elaboreze aplicații informatice, pagini web, jocuri digitale, etc.

Limbajele de programare sunt concepute pentru a permite programatorilor să scrie cod pe care calculatoarele îl pot înțelege și executa. Ele oferă o sintaxă și o gramatică care definește regulile de scriere a codului valid, precum și o semantică care definește semnificația codului.

Există numeroase limbaje de programare, fiecare cu propria sintaxă, caracteristici și aplicații. Unele limbaje de programare sunt de uz general și pot fi utilizate pentru o gamă largă de sarcini, în timp ce altele sunt concepute pentru scopuri specifice, cum ar fi dezvoltarea web, analiza datelor sau programarea sistemului.

Există mai multe tipuri de limbaje de programare, fiecare având scopuri și caracteristici diferite:

1. Limbaje de programare de nivel înalt:
    - sunt concepute pentru a fi ușor de înțeles de către oameni;
    - exemple: Python, Java, C++, JavaScript;
    - limbaje utilizate pentru majoritatea aplicațiilor, eficiente pentru dezvoltare rapidă.
2. Limbaje de programare de nivel scăzut:
    - sunt mai aproape de limbajul „nativ” al procesorului (codul mașină), iar programatorii trebuie să gestioneze direct resursele hardware;
    - exemple: Assembly;
    - acestea sunt mai greu de utilizat, dar pot oferi un control mai bun asupra performanței;
3. Limbaje de programare de nivel mediu:
    - combină elemente din limbajele de nivel înalt și cele de nivel scăzut, permițând atât accesul rapid la resursele hardware, cât și o sintaxă mai ușor de înțeles;
    - Exemple: C, C++.
4. Limbaje declarative și funcționale:

- în limbajele declarative, programatorul specifică ce vrea să facă, nu cum să facă;
- exemplu: SQL pentru baze de date;
- limbajele funcționale sunt bazate pe conceptul de funcții și evită schimbarea stării sau a datelor;
- Exemple: Haskell, Lisp.

Exemple de limbaje de programare:

- C++: un limbaj de programare, orientat pe obiecte, cunoscut pentru performanță și versatilitate;
- Python: un limbaj interpretat de nivel înalt, cunoscut pentru simplitatea și lizibilitatea sa, adesea folosit pentru dezvoltarea web, analiza datelor și inteligența artificială;
- Java: un limbaj de uz general, orientat pe obiecte, cunoscut pentru independența și robustețea platformei sale, adesea folosit pentru construirea de aplicații pentru întreprinderi la scară largă;
- JavaScript: un limbaj interpretat de nivel înalt, utilizat în principal pentru dezvoltarea web la nivelul clientului, care permite pagini web interactive și dinamice;
- Ruby: un limbaj dinamic, orientat pe obiecte, cunoscut pentru simplitatea și productivitatea sa, adesea folosit pentru dezvoltarea web;
- Swift: un limbaj compilat, dezvoltat de Apple pentru a construi aplicații iOS, macOS, watchOS.

Un program scris într-un limbaj de programare este de obicei transformat într-un cod care poate fi înțeles și executat de procesorul unui calculator printr-un compilator sau interpretor:

- *Compilatorul* este un program care transformă codul sursă scris într-un limbaj de programare de nivel înalt (C, C++, Java) într-un cod executabil pe calculator (cod mașină sau bytecode). Procesul de compilare are mai multe etape: analiza lexicală, analiza sintactică, analiza semantică, optimizarea codului și generarea codului mașină.
- *Interpretorul* este un program care citește și execută codul sursă linie cu linie, fără a-l transforma într-un fișier executabil înainte de rulare. Acest lucru îl face diferit de un compilator, care traduce întregul cod într-un format executabil înainte de a-l rula. Limbajele de programare joacă un rol important în dezvoltarea software-ului, permițând programatorilor să creeze o varietate de aplicații și sisteme pentru a satisface diferite nevoi și cerințe.

Limbajele de programare joacă un rol important în dezvoltarea software-ului, permițând programatorilor să creeze o varietate de aplicații și sisteme pentru a satisface diferite nevoi și cerințe.

În cadrul acestei lucrări se va studia limbajul de programare C++. Orice program C++ începe cu declararea fișierelor antet. În C++, instrucțiunile sunt separate prin punct și virgulă, această sintaxă fiind interpretată ca și instrucțiune vidă.

Exemplu:

```
#include<iostream.h>

#include<math.h>
```

2. Limbajul de programare C++

C++ este un limbaj de programare de nivel înalt care extinde limbajul C prin adăugarea de caracteristici orientate pe obiect (OOP), ceea ce permite programatorilor să creeze programe complexe și modulare. Este un limbaj puternic, cu o mare flexibilitate și control asupra resurselor hardware, și este utilizat pe scară largă pentru dezvoltarea aplicațiilor de performanță înaltă, sisteme de operare, jocuri video, software de inginerie și multe altele.

Caracteristici esențiale ale C++:

- programare orientată pe obiecte (OOP): C++ permite definirea de clase și obiecte, moștenire, polimorfism și încapsulare;
- performanță înaltă: C++ oferă un control foarte detaliat asupra resurselor hardware, ceea ce îl face potrivit pentru aplicații care necesită o performanță ridicată, cum ar fi jocurile, aplicațiile grafice sau sistemele în timp real;
- compilare eficientă: C++ este un limbaj compilat, iar programele C++ sunt transformate în cod mașină, ceea ce le face foarte rapide în execuție;
- suport pentru manipularea directă a memoriei: C++ permite utilizarea pointerilor pentru a manipula direct memoria, oferind control total asupra alocării și eliberării memoriei;
- bibliotecă standard extinsă (STL): C++ include o bibliotecă standard foarte puternică, care include structuri de date și algoritmi eficienți, precum vectori, liste, seturi și map-uri.

2.1 Sintaxa și semantica C++

Sintaxa este setul de reguli care definesc structura și formatarea corectă a unui limbaj de programare. Este similară cu gramatica într-un limbaj natural,

asigurându-se că instrucțiunile sunt scrise corect pentru a fi înțelese de calculator. Nu se acceptă ambiguități. Încălcarea oricărei reguli a limbajului de programare poate genera erori de sintaxă și codul sursă nu poate fi compilat până la corectarea lor.

Exemple de erori de sintaxă:

- scrierea incorectă a unui cuvânt
- uitarea unei virgule

Importanța sintaxei:

- ajută programatorii să scrie codul clar și ușor de citit;
- permite compilatoarelor și interpretoarelor să execute corect programele;
- previne erorile care ar putea duce la blocarea programului.

Semantica unui limbaj de programare definește semnificația și comportamentul corect al instrucțiunilor scrise în acel limbaj. Dacă sintaxa se referă la regulile de scriere a codului, semantica stabilește ce face codul respectiv.

Dacă un cod este scris corect din punct de vedere sintactic, dar nu produce rezultatul dorit, atunci are o eroare semantică.

Importanța semanticii:

- ne ajută să înțelegem ce face programul, nu doar cum trebuie scris;
- previne erori logice care pot cauza comportamente neașteptate;
- face diferența între un cod care doar "merge" și unul care face ceea ce trebuie.

În C++, un **identificator** este denumirea folosită pentru a reprezenta variabile, funcții, clase, constante, array-uri și alte elemente definite de utilizator. Identificatorii permit programatorilor să dea nume semnificative elementelor din cod, făcând programul mai clar și mai ușor de înțeles.

Reguli pentru identificatori în C++:

- trebuie să înceapă cu o literă (A-Z, a-z) sau un underscore _ ;
- poate conține litere, cifre (0-9) și underscore _ ;
- nu poate fi un cuvânt-cheie rezervat (exemplu: int, return, for);
- sensibil la majuscule și minuscule (nume și Nume sunt diferite);
- fără spații (folosește ” _”).

Exemple de identificatori corecți:

- A7;
- suma_nr_pare;
- GetData.

Exemple de identificatori incorecți:

- 6ore - nu poate începe cu o cifră;
- get data - nu poate conține spațiu;
- x - 5 - nu poate conține ‘-’ deoarece este simbol matematic;
- int - cuvântul int predefinit în C++.

2.2 Structura programului în C++

Structura unui program constă în:

1) directive de preprocesare - sunt comenzi pentru compilator care încep cu simbolul '#'. Sunt procesate înainte de compilarea codului. Directivele de preprocesare obișnuite includ (*#include*) fișiere antet sau definesc macros (*#define)*;
2) declarații namespace - se folosesc pentru organizarea codului în grupuri logice pentru a preveni conflictele de notație. În programele C++, de obicei se folosește "using namespace std;" care permite folosirea elementelor din librăria C++ standard fără a specifica namespace de fiecare dată;
3) funcția main - toate programele C++ trebuie să conțină funcția *main()*, aceasta fiind partea centrală a programului și executarea programului începând cu această funcție;
4) definirea funcțiilor - există și alte funcții în afară de funcția *main()*. Scopul acestora este de a organiza codul în părți mai mici pentru a fi mai ușor de înțeles. Funcțiile sunt formate din: numele funcției, parametri, corpul funcției și return;
5) declararea variabilelor - se declară variabile pentru a se stoca datele în program. Acestea trebuie declarate înainte de a fi folosite;
6) declarații și expresii - acestea sunt elementele de bază ale programului. Declarațiile sunt instrucțiuni care efectuează expresii, iar expresiile sunt combinații de operatori, constante sau variabile care au ca rezultat o valoare;
7) comentarii - se folosesc pentru a documenta codul. Acestea oferă informații despre scopul codului sau alte detalii semnificative. În C++ se pot scrie comentarii pe o singură linie folosindu-se *" //"* sau pe mai multe linii folosindu-se *"/* ... */"*.

Exemplu:

```
#include <iostream>
using namespace std;
// Prototipul functiei
void greet(string name);
// Functia main()
int main() {
    string userName;
    cout << "Introduceti numele ";
    cin >> userName;
    greet(userName);
    return 0;
}
// Definirea functiei
void greet(string name) {
    cout << "Hello, " << name << "!" << endl;
}
```

Fiecare funcție conține acolade { } care marchează începutul și sfârșitul instrucțiunilor care trebuiesc executate. Codul dintre acolade se numește *corpul funcției* sau *bloc de instrucțiuni*.

2.3 Date și tipuri de date

În C++, tipurile de date stabilesc ce fel de valori pot fi stocate și manipulate de variabile, influențând dimensiunea și modul în care acestea sunt reprezentate în memorie. Informațiile sunt păstrate în memoria calculatorului, unde fiecare locație are o adresă unică exprimată în format binar. Această adresă este utilizată pentru a accesa sau modifica datele stocate. Limbajele de programare de nivel înalt, precum C++, ne scapă de necesitatea de a gestiona direct aceste adrese de memorie. În acest context, tipul de dată indică natura valorii pe care o poate conține o variabilă.

Mai jos sunt descrise tipurile de date din C++:

a. tipuri numerice:

- int
- long
- enum
- short
- char

Se referă la valori întregi. Numerele întregi sunt secvențe de una sau mai multe cifre și nu se admite virgula. Semnul '-' precedă un număr întreg.

Exemplu: 20, -58, 0, 30, -570

Se poate folosi cuvântul rezervat "unsigned" precedat de tipul datei. Valoarea întreagă poate fi 0 sau doar pozitivă.

Exemplu: unsigned intreg;

O dată de tip *int* se poate găsi în intervale diferite: pe un calculator poate fi intervalul -32768 ... +32767, iar pe altul între -2147483648 ...

+2147483647. Pentru a afla valorile maximă și minimă din interval pentru un anumit calculator, se poate folosi librăria *"limits"* din C++ Standard Library. Dacă calculatorul încearcă să calculeze o valoare mai mare decât valoarea sa maximă, atunci rezultatul va fi un *integer overflow*.

Exemplu pentru a obține valoarea minimă și maximă pentru un întreg:

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

int main() {
    cout << "Valoarea minima este: " << numeric_limits<int>::min()
<< endl;
    cout << "Valoarea maxima este " << numeric_limits<int>::max()
<< endl;

    return 0;
}
```

Un întreg dacă începe cu cifra 0 este considerat ca fiind scris în baza 8.

Exemplu: 013 este 13_8

Tipul de dată *char* este cel mai mic tip de dată care poate fi folosit pentru valorile întregi. Acesta este recomandat pentru situațiile când se dorește o economie de memorie și se folosesc numere întregi mici. De obicei, tipul *char* se folosește pentru a descrie date care constau într-un caracter alfanumeric din setul de caractere ASCII (literă, cifră sau simbol special).

Exemplu: 'A' 'a' '9' '1' '+' '-' '*' ' '

Fiecare caracter este cuprins între apostroafe, astfel încât limbajul de programare C++ să poată face diferența dintre data de tip caracter '9' și valoarea întreagă 9.

Nu se obișnuiesc operații de adunare sau scădere a caracterului *'a'* cu caracterul *'b'*, dar aceste caractere se pot compara, astfel că caracterul *'a'* este întotdeauna mai mic decât caracterul *'b'*.

b. tipuri reale (virgulă flotantă):

- float
- double
- long double

Se folosesc pentru a reprezenta numere reale. Numerele cu virgulă mobilă au o parte reală și o parte fracționară, separate de un punct.

Exemplu: 40.0 478.97 0.7 .4 6859.795

Numerele cu virgulă mobilă pot avea un exponent. În loc de $1.304x10^{15}$, în C++ scriem 1.304e15, unde 'e' înseamnă exponent al bazei 10. Numărul dinaintea lui *e* nu trebuie să includă obligatoriu punctul zecimal.

Cel mai folosit tip de dată este tipul *float* deoarece este suficient. Valoarea maximă oferită de acest tip de dată este în general în jur de 3.4e+38.

Pot exista erori deoarece unele calculatoare nu pot reprezenta întotdeauna numerele în virgulă mobilă. Multe valori reale pot fi aproximate în sistem datorită faptului că memorarea se face în formă binară.

Exemplu: 6.9 poate fi afișat pe unele calculatoare ca 6.89999998 fără să fie vreo eroare de programare

c. **Tipuri adresă:**
   - pointer
   - referință

d. **Tipuri structurate/definite de utilizator:**
   - tablou (array)
   - struct
   - union
   - class

2.4 Declarații

O declarație în C++ este o instrucțiune prin care se introduce un identificator (nume de variabilă, funcție, clasă etc.) în program, specificând tipul său de date, dar fără a-i atribui în mod obligatoriu o valoare.

Sintaxa pentru a declara o variabilă în C++ este: tip_dată nume_variabilă;

Exemplu:

```
int nr1;
double nr2;
char c;
```

În exemplu de mai sus, *nr1* este numele unei variabile al cărei conținut este de tip *int*. Când declarăm o variabilă, compilatorul alege o locație pentru aceasta și o asociază cu identificatorul păstrând această asociere la dispoziția programului. Fiecare identificator trebuie să fie unic în domeniul lui într-un program.

O variabilă se poate inițializa în timpul declarării: tip_dată nume_variabilă = valoare_inițială;

Exemplu:

```
int nr1 = 10;

double nr2 =  20.4;

char c = 'a';
```

În C++ un identificator trebuie mai întâi declarat și apoi folosit. Dacă definim un identificator ca fiind constantă și mai târziu încercăm să îi modificăm valoarea, compilatorul detectează această incompatibilitate și semnalează eroarea.

2.5 Variabile

O variabilă în C++ este o locație de memorie care are un nume și care poate stoca o valoare. Fiecare variabilă este asociată cu un tip de date, care definește ce tip de valoare poate stoca și câtă memorie este necesară pentru stocarea acelei valori. Variabilele permit programatorilor să stocheze și să manipuleze date în timpul execuției unui program.

Caracteristicile unei variabile în C++ sunt:

- numele variabilei (identificator) - fiecare variabilă trebuie să aibă un nume unic care o identifică în program. Numele poate conține litere, cifre și caractere de subliniere, dar trebuie să înceapă cu o literă sau un underscore _;

- tipul de date - fiecare variabilă are un tip de date (de exemplu: int, float, char), care definește tipul de valoare pe care o poate stoca;
- valoarea - poate stoca o valoare (de exemplu: un număr întreg sau un caracter) care poate fi citită, modificată sau manipulată.;
- memoria - o variabilă ocupă un anumit spațiu de memorie, în funcție de tipul său (de exemplu: un int ocupă de obicei 4 bytes).

Declarația variabilei presupune specificarea tipului de dată și a numelui variabilei. Declarația de variabile se termină întotdeauna cu caracterul ';'.

Exemplu:

```
int nr;
```

În C++, se pot declara variabile multiple într-o singură declarație, acestea având același tip de dată: tip_dată variabila_1, variabila_2, variabila_3;

Exemplu:

```
int nr1, nr2, nr3;
```

Această declarație este echivalentă cu:

```
int nr1;

int nr2;

int nr3;
```

Declararea fiecărei variabile pe o linie separată ne permite adăugarea comentariilor care ne vor ajuta pe parcursul programului. Comentariile sunt precedate de simbolul "*//*" și sunt ignorate de compilator.

Exemplu:

```
int nr1; //declarea unui număr întreg
```

Se pot adăuga comentarii pe mai multe linii folosindu-se "/* ... */"

2.6 Constante

În C++, o constantă este o valoarea a unei variabile ce nu poate fi schimbată pe parcursul execuției programului. Constantele pot fi: numere întregi, numere reale, caractere (cuprinse între ' ') și secvențe de caractere sau string-uri (cuprinse între " ").

Exemplu:

49 37.5 'A' "casa"

Caracteristicile constantelor în C++:

- valoare neschimbabilă: odată ce valoarea este atribuită unei constante, nu se poate schimba în restul programului.
- definirea constantelor: folosim cuvântul-cheie const pentru a declara o constantă.
- tip de date: constantelor le putem atribui orice tip de date (int, double, char, etc.).

Se folosesc constante ca părți ale unor expresii matematice. Se pot scrie instrucțiuni care adună, scad, înmulțesc, împărțesc constante și rezultatul obținut se plasează într-o altă variabilă. Numim *valoare literară* orice constantă din program.

O alternativă sunt constantele simbolice (constante cu nume). Acestea reprezintă locații de memorie referite printr-un identificator care păstrează date ce pot fi modificate. De exemplu: folosim constanta simbolică PI în loc de literalul 3.14159. Astfel programul este mai ușor de citit și modificat.

Declararea constantelor în C++ se face cu cuvântul cheie *const*, iar semnul "=" se pune între identificator și valoarea literară.

Exemplu:

```
const int test = ' ';
const float PI = 3.14159;
const int = 10;
```

De obicei, constantele simbolice se scriu cu litere mari pentru a le distinge mai ușor.

2.7 Asignare

Asignarea presupune schimbarea valorii variabilei.

Exemplu:

```
nota = 10;
```

În exemplul de mai sus, valoarea *10* este asignată variabilei *nota*, adică valoarea 10 va fi stocată în locația de memorie numită *nota.*Orice valoare anterioară stocată se pierde, fiind înlocuită de noua valoare.

Într-o instrucțiune de asignare doar o valoare poate apărea în stânga operației.

Notă: a=10; → variabila *a* primește valoarea *10*

```
if (a==10)
```

→ se verifică dacă valoarea variabilei *a* este egală cu *10* (echivalent cu egalul din matematică)

Dacă avem următoarele declarații:

```
int a;
float b;
double c;
char ch;
int d;
```

Se pot face următoarele asignări:

```
a = 50;
b = 59.8;
c = 47.46790;
ch = 'K';
d = a ;
```

Asignarea *ch = "Test"*; nu este corectă deoarece *ch* este o variabilă de tip *char*, iar *"Test"* este un string.

Pe parcursul acestui laborator se vor respecta următoarele reguli pentru o mai bună lizibilitate:

1. folosim inițială mică pentru numele variabilelor.

Exemplu: num nr_intreg

2. folosim inițială mare pentru numele de funcții și clase.

   Exemplu: Calcul_mediaaritmetica() Angajat

3. folosim majuscule pentru constante simbolice.

   Exemplu: PI

O expresie este formată din constante, variabile și operatori. Operatorii admiși într-o expresie depind de tipurile de date ale constantelor și ale variabilelor din expresie. Operatorii matematici care pot fi folosiți în expresii sunt:

a. plus unar +
b. minus unar -
c. adunare +
d. scădere -
e. înmulțire *
f. împărțire reală (rezultatul este de tip real) sau împărțire întreagă (rezultatul este de tip întreg)
g. modulo % - reprezintă restul împărțirii

Notă: Operatori unari folosesc un singur operand, iar operatorii binari folosesc 2 operanzi.

O variabilă fără semn este considerată pozitivă.

Câtul împărțirii a 2 numere întregi se numește împărțire întreagă, iar modulo reprezintă restul împărțirii și se aplică doar numerelor întregi.

Exemplu:

8 / 2 → 4 câtul împărțirii lui 8 la 2 este 4.

8 % 2 → 0 restul împărțirii lui 8 la 2 este 0.

9 / 2 → 4 câtul împărțirii lui 9 la 2 este 4.

9 % 2 → 1 restul împărțirii lui 9 la 2 este 1.

Rezultatul unei împărțirii reale este un număr real, iar valoarea operanzilor trebuie să fie reală.

Exemplu împărțire reală:

8.0 / 2.0 → 4.0

9.0 / 2.0 → 4.5

Expresie	**Rezultat**
4 + 8	12
2.2+4.9	7.1
2 * 7	14
10.0 / -5	-2
7 % 2.4	eroare
5.0 / 0.0	eroare
9 % 0	eroare

În expresii pot apărea și variabile.

Exemplu:

```
num1  = num2 + 7;
```

num1 = num2 + num3;

Dacă avem instrucțiunea *num1 = num2 + num3* înseamnă că variabila num1 primește rezultatul adunării dintre num2 și num3, astfel variabila *num1* își pierde vechea valoare.

2.8 Incrementare și decrementare

C++ pune la dispoziția programatorilor operatori de incrementare și decrementare. Aceștia sunt operatori unari și pot fi folosiți pentru numere întregi sau reale. Incrementarea și decrementarea sunt operații aritmetice fundamentale în C++, care permit modificarea valorii unei variabile prin creșterea sau scăderea acesteia cu o unitate. Aceste operații sunt folosite frecvent în bucle și în manipularea variabilelor numerice.

a. Incrementare ++
b. Decrementare --

Exemplu:

```
int a=15;
```

a++; → după executarea acestei instrucțiuni variabila *a* va avea valoarea *9* (echivalent cu a = a + 1;)

Forme de incrementare:

- incrementarea prefixată (++variabilă) - incrementarea se face înainte de utilizarea variabilei. Aceasta înseamnă că valoarea variabilei va fi crescută înainte de a fi utilizată în expresie.
- incrementarea sufixată (variabilă++): incrementarea se face după ce variabila a fost utilizată în expresie. Valoarea variabilei este folosită înainte de a fi crescută.

Forme de decrementare:

- decrementare prefixată (--variabilă): decrementarea se face înainte de utilizarea variabilei. Aceasta înseamnă că valoarea variabilei va fi scăzută înainte de a fi utilizată în expresie.
- decrementare sufixată (variabilă--): decrementarea se face după ce variabila a fost utilizată în expresie. Valoarea variabilei este folosită înainte de a fi scăzută.

Aceste operații sunt deosebit de utile atunci când se lucrează cu bucle (de exemplu: bucle for, while), unde se modifică o variabilă de fiecare dată.

2.9 Afișare

În C++, afișarea se referă la ieșirea de date pe ecran sau într-un fișier pentru a le face vizibile utilizatorului sau pentru a utiliza datele în scopuri de depanare (debugging). C++ oferă mai multe metode pentru a afișa informații pe consolă (ecran), iar cele mai frecvent utilizate sunt *cout* din biblioteca standard *iostream* și alte funcții din biblioteci externe sau personalizate.

Exemplu:

```
cout << "hello";
```

Această instrucțiune afișează caracterele hello pe dispozitivul de ieșire, de obicei este reprezentat de ecran. Variabila *cout* este predefinită în C++ și semnifică un *flux de ieșire*.

Operatorul de inserție << (“put to”) folosește 2 operanzi. Operandul din stânga reprezintă o expresi flux (exemplu *cout*), iar cel din dreapta conține șirul de caractere sau expresia al cărei rezultat trebuie afișat.

Exemplu:

```
cout << "Rezultatul este ";
cout << 2*4;
```

Se poate folosi de mai multe ori operatorul într-o singură instrucțiune de afișare și rezultatul fiind același.

Exemplu:

```
cout << "Rezultatul este "<< 2*4;
```

Dacă dorim să afişăm un şir care conţine caracterul ” trebuie să plasăm semnul \ înaintea ”:

Exemplu:

```
cout << ”Ion \”Anna\” Vasile”;
```

În mod normal, mai multe instrucțiuni de ieșire succesive afișează rezultatele pe aceeași linie.

Exemplu:

```
cout << ”Hello";
cout <<"Ana!";
```

Se afișează: Hello Anna!

Pentru a se afișa text pe linii separate se folosește identificatorul *endl.* Acesta este un manipulator și permite terminarea unei linii și continuarea scrierii pe linia următoare.

Exemplu:

```
cout << ”Hello"<<endl;

cout <<"Ana!";
```

Se afișează:

```
Hello

Ana!
```

Nota: Pentru a folosi obiectul cout trebuie inclusă biblioteca standard “iostream” la începutul codului.

2.10 Elaborarea unui program C++

Elaborarea unui program în C++ presupune mai multe etape, de la definirea problemei și a cerințelor, la scrierea codului și testarea acestuia. În continuare se va prezenta cum se scriu diferite elemente prezentate până acum pentru a forma un program C++. Un program C++ este format din clase și funcții, neapărat una din funcții numindu-se *main*.

Modelul unui program este:

- crearea unui fișier sursă
- includerea librăriilor necesare
- scrierea funcției *main()*
- scrierea codului

```cpp
#include <iostream>

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

Figura 2.10-1 Program C++

- compilarea programului - se folosește un compilator C++ precum "g++" (GNU Compiler Collection), unde hello.cpp este numele fișierului.

```
g++ hello.cpp -o hello
```

Figura 2.10-2 Compilare cod C++

- rularea programului

```
./hello
```

Figura 2.10-3 Rularea programului după compilare

2.11 Exerciții

1. Scrieți 4 instrucțiuni care să declare: un număr întreg, un număr real, un caracter și o constantă de tip float.
2. La exercițiul precedent adăugați comentarii în 2 moduri pentru fiecare variabilă definită.
3. Se dau următoarele nume de variabile. Stabiliți care sunt corecte și care sunt greșite. Explicați și corectați numele de variabile greșite.

   nr 1

numar_intreg

float

nr_real

char1

&variabila-7

a@fgsfe

4. Declarați 3 variabile și asignați-le cu valori: una de tip întreg, una de tip real și una de tip char.

   Variabila de tip întreg să se incrementeze și variabila de tip real să se decrementeze.

5. Afișați pe ecran (în consolă) mesajul: “Primul meu program în C++”
6. Scrieți o instrucțiune C++ care afișează un text pe o singură linie.
7. Scrieți instrucțiunii C++ care afișează mesajul: “Bun venit la laborator!”
   a. pe o singură linie
   b. pe 2 linii
   c. pe 3 linii
8. Scrieți un program C++ care afișează suma a 2 numere întregi.
9. Scrieți un program C++ care afișează produsul a n numere naturale. Valoarea variabilei n se va citi de la tastatură.

3. Expresii aritmetice, apeluri de funcții și ieșiri

O **expresie** în programare este o combinație de **operanzi** și **operatori** care produce un rezultat. În funcție de complexitatea expresiei, aceasta poate include doar câțiva operanzi și operatori simpli sau poate deveni mai complexă, incluzând funcții, paranteze și operatori suplimentari. Acestea sunt esențiale în algoritmi deoarece permit efectuarea unor calcule sau manipulări de date. În contextul programării, expresiile pot include numere, variabile și operatori, iar evaluarea lor poate duce la obținerea unui rezultat care va fi folosit mai departe în algoritm.

În timpul execuției unui algoritm, expresiile sunt evaluate pas cu pas, iar variabilele sunt înlocuite cu valorile lor actuale. Astfel, procesul de evaluare a expresiilor presupune înlocuirea variabilelor cu datele stocate în memorie și apoi efectuarea calculelor sau evaluărilor, în funcție de tipul expresiei.

Există două categorii principale de expresii: expresiile aritmetice și expresiile logice, iar fiecare dintre acestea are un rol specific în algoritmi și programe.

3.1 Expresii aritmetice

În C++, o expresie este o combinație de variabile, constante, operatori și funcții care sunt evaluate pentru a obține un rezultat. Expresiile sunt fundamentale în orice program deoarece ele reprezintă modul prin care sunt efectuate calcule, comparații și alte tipuri de operații.

Tipuri de expresii în C++:

- expresii aritmetice: sunt folosite pentru a efectua operații matematice (adunare, scădere, înmulțire, împărțire, etc.). Exemplu: int x = 5 + 3;
- expresii de atribuire: sunt folosite pentru a atribui valori variabilelor. Exemplu: int a = 10;
- expresii de comparații: exemplu: bool b = (x > 3);
- expresii logice: sunt folosite pentru a face comparații între valori, folosind operatori logici (exemplu: &&, ||, !). Exemplu: bool result = (a > 5) && (b < 10);
- expresii de incrementare și decrementare. Exemplu: count++;
- expresii de condiție (ternare): o expresie ternară este o formă compactă de a scrie o instrucțiune *if-else*. Exemplu: int x = (a > b) ? a : b;

Exemplu:

nr = nr1 + nr2 / nr3

În acest exemplu, mai întâi se efectuează împărțirea *nr2 / nr3,* iar apoi rezultatul este adunat cu *nr1*.

Exemplu:

nr = (nr1 + nr2) / nr3

Folosindu-se parantezele, ordinea efectuării operațiilor se schimbă astfel: mai întâi se evaluează subexpresia *(nr1+nr2)* apoi se face împărțirea cu *nr3*

Atunci când există mai mulți operatori cu aceeași precedență, asociativitatea acestora se face de la stânga la dreapta.

Exemplu:

f1 - f2 + f3 este echivalentă cu (f1 - f2) + f3

și nu cu: f1 - (f2 + f3)

Se pot folosi funcții matematice disponibile în librăria standard. Aceste funcții includ funcțiile trigonometrice (sin, cos, tan), exponențială și logaritmică (exp, log, log10), dar și alte funcții. Pentru a fi folosite, este necesară includerea librărie *cmath*.

Exemplu - operații matematice în C++:

```
#include <iostream>
#include <cmath>  // pentru sqrt() și pow()
int main() {
    double x = 2.0;
    double y = 3.0;
    // calculează: √x + y^2
    double result = sqrt(x) + pow(y, 2);
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

Valorile întregi și valorile reale sunt stocate diferit în memoria calculatorului. Cu alte cuvinte modelul de memorie al biților care reprezintă

numărul natural 2 este diferit față de numărul real 2.0. Apare o problemă atunci când se întâlnesc un întreg și un real într-o expresie sau într-o asignare.

Dacă se fac declarațiile:

```
int i;

float j;
```

atunci variabila *i* poate păstra doar valori întregi, iar variabila *j* doar valori în virgulă mobilă. Instrucțiunea de asignare `j = 5;` pare că încarcă valoarea întreagă 5 în variabila j, dar calculatorul refuză să stocheze altceva decât valori de tip float în variabila *j*. Compilatorul inserează, două noi instrucțiuni care mai întâi convertește valoarea *5* în *5.0* şi apoi stochează 5.0 în variabila *j*. Această transformare automată a unei valori dintr-un tip de dată în alt tip de dată se numește *conversie implicită*.

Instrucțiunea `int i = 2.5;` reprezintă de asemenea o forțare de tip. Când un număr real este asignat unei variabile întregi, partea fracționară este trunchiată. Ca rezultat, variabilei *i* i se va asigna valoarea *2*.

Deseori, în conversiile implicite sunt implicate expresii. Păstrarea rezultatului unei expresii reale într-o variabilă de tip *float* nu duce la pierderea de informații, dar stocarea rezultatului unei expresii reale într-o variabilă de tip întreagă (*int*), duce la trunchierea părții fracționare. Pentru a se evita astfel de pierderi, se folosește *conversia explicită.* În C++ această operație constă în precizarea tipului de dată pe care dorim să îl aibă rezultatul expresiei urmat, între paranteze rotunde, de expresia pe care dorim să o convertim.

Exemplu:

```
a = int (6 * 5.8 + 2.4)
```

b = float (2 / 7 + 10.8)

c = 5.3 + float (7.9 + 57 / 6.3)

Este posibilă combinarea datelor de diferite tipuri în expresii.

3.2 Intrările în program

În C++, intrările se referă la datele sau informațiile pe care un program le primește de la utilizator sau dintr-o sursă externă (de exemplu: un fișier, un dispozitiv hardware, etc.). Intrările sunt esențiale pentru interactivitatea unui program deoarece permit utilizatorilor să interacționeze cu programul, să introducă valori și să obțină un rezultat.

Cel mai des întâlnit tip de intrare într-un program C++ este cel care vine de la utilizator prin intermediul tastaturii. C++ oferă cin pentru a citi aceste intrări din fluxul de intrare. Acesta face parte din C++ Standard Library's Input/Output și este folosit împreună cu operatorul de extracție *">>"*.

Exemplu

cin **>>** a**;**

Se poate folosi operatorul de extracție de mai multe ori într-o instrucțiune.

Exemplu

cin >> a >> b >> c;

Se pot face următoarele declarații:

istream cin**;**

ostream cout**;**

Cele 2 declarații de mai sus arată că *cin* este un obiect de tip *istream*, iar *cout* este un obiect de tip *ostream*. Cu alte cuvinte, *cin* este asociat cu tastatura, iar *cout* cu afișarea (display-ul).

Notă: cin poate fi folosit doar în combinație cu >>, iar cout doar cu <<

Când se introduce o dată de la tastatură, trebuie să ne asigurăm că tipul introdus și cel așteptat de program se potrivesc. Un număr întreg este forțat automat la un număr real. Operația inversă, poate duce la rezultate eronate. Atunci când se extrag valori dintr-un stream, operatorul ">>" ignoră orice spațiu de la început. De asemenea, ignoră caracterul care marchează sfârşitul liniei. Apoi, operatorul ">>" procedează la extragerea valorilor din stream-ul de intrare. Dacă data așteptată este un *char*, intrarea se întrerupe după primul caracter. Dacă este vorba de un *int* sau *double*, intrarea se întrerupe la primul caracter care nu se potrivește ca tip de dată, cum ar fi un spațiu.

Exemplu de citirea datelor de la tastatură folosind instrucțiunea cin

```
#include <iostream>

int main() {

int number;

std::cout << "Introduceti un numar ";

std::cin >> number;  // citire numar de la tastatura

std::cout << "Ati introdus numarul " << number << std::endl;

return 0;
```

}

În exemplul de mai sus, *cin* este folosit pentru a citi de la tastatură un număr întreg care va fi stocat în variabila numită *number*. Operatorul ">>" extrage datele din stream-ul de intrare și le salvează în variabilă.

Cin este doar unul din obiectele predefinite de intrare de tip stream. În C++ există posibilitatea de a citi o linie folosind "std::getline()" sau alte stringuri mai complexe folosindu-se "std::stringstream"

Citirea datelor de tip caracter folosind instrucțiunea get

În C++, instrucțiunea get() este folosită pentru a citi un caracter din fluxul de intrare, incluzând spațiile și caracterele speciale. Spre deosebire de cin, care citește doar până la primul spațiu sau linie nouă, get() permite citirea caracterelor pe întreaga linie, inclusiv spațiile.

Metoda *std::cin.get()* se folosește pentru a citi un singur caracter de la tastatură.

Exemplu de citire a unui caracter de la tastatură folosind funcția get()

```
#include <iostream>

int main() {

   char ch;

   std::cout << "Introduceti un caracter ";

   ch = std::cin.get();  // Citirea unui singur caracter de la tastatura

   std::cout << "Ati introdus caracterul " << ch << std::endl;
```

```
    return 0;
}
```

Metoda *std::getline()* se folosește pentru a citi o linie cu text (mai multe caractere) și o salvează într-o variabilă de tip string.

Exemplu de citirea a unei linii de la tastatură folosind funcția getline()

```
#include <iostream>
#include <string>
int main() {
    std::string line;
    std::cout << "Introduceti o linie cu text: ";
    std::getline(std::cin, line);  // se citeste o linie de text introdusa de utilizator
    std::cout << "Ati introdus: " << line << std::endl;
    return 0;
}
```

Ignorarea caracterelor folosind funcția *ignore*

În C++, funcția ignore() este utilizată pentru a ignora (sări) caractere din fluxul de intrare. Aceasta este foarte utilă atunci când vrem să curățăm bufferul de intrare (de exemplu: pentru a ignora caracterele care sunt deja în

buffer și care nu ne sunt necesare), sau atunci când dorim să sărim peste anumite caractere (cum ar fi caracterele de linie nouă sau spațiile).

În C++, funcția *ignore* face parte din clasa std::istream care este folosită pentru operațiile de intrare. Funcția *ignore* are 2 parametri.

Exemplu de folosire a funcției ignore

```
#include <iostream>
int main() {
   char ch;
   // Elimina primele trei caractere din fluxul de intrare
   std::cin.ignore(3);
   // Citeste si afiseaza caracterul urmator
   std::cin.get(ch);
   std::cout << " Următorul caracter după ignorare este: " << ch << std::endl;
   return 0;
}
```

În exemplul de mai sus, se ignoră primele 3 caractere de la începutul stream-ului - std::cin.ignore(3). Apoi se citește și se afișează următorul caracter.

3.3 Citirea și scrierea în fișiere

În C++, un fișier reprezintă o locație de stocare a datelor pe disc sau într-un alt dispozitiv de stocare permanentă. Fișierele pot conține informații care sunt citite sau modificate de către programele C++ în timpul execuției lor. C++ oferă un set de funcționalități pentru a citi din fișiere și a scrie în fișiere,

folosind clasele și funcțiile din biblioteca <fstream>.Un program C++ poate citi datele scrise dintr-un fișier, dar poate și scrie în fișier. Cele 3 clase principale oferite de librăria <fstream> sunt:

1. *ifstream* - se folosește pentru citirea datelor din fișiere
2. *ofstream* - se folosește pentru scrierea în fișiere
3. *fstream* - se folosește atât pentru citirea din fișier, cât și pentru scrierea în fișier.

Exemplu de lucru cu fișiere

```
#include <iostream>

#include <fstream> // Includerea bibliotecii de fișiere I/O

int main() {

 // Crearea unui flux de fisiere de iesire pentru a scrie într-un fisier

std::ofstream outFile("example.txt");

// se verifica daca fluxul de fisiere este deschis

if (outFile.is_open()) {

        // scrierea datelor in fisier

        outFile << "Hello, World!" << std::endl;

        outFile << 42 << std::endl;

        outFile << 3.14 << std::endl;

        // inchiderea fluxului de fișiere

        outFile.close();
```

```
        std::cout << "Datele au fost scrise in fisier." << std::endl
} else {
        std::cerr << "Eroare: Imposibilitatea de a deschide fișierul." << std::endl;
            }
return 0;
        }
```

În exemplul de mai sus, se fac următoarele acțiuni:

- se include header-ul <*fstream*>
- se creează obiectul de tip *ofstream* numit *"outFile"* care reprezintă fișierul de ieșire
- se deschide fișierul numit *"example.txt"* folosit pentru a se scrie în el.
- se scrie în fișier folosindu-se operatorul de inserție "<<"
- se se închide fișierul folosindu-se funcția *close()*
- dacă deschiderea fișierului nu se poate realiza, atunci se va afișa un mesaj de eroare (*std::cerr*)

Citirea din fișier este necesară pentru a declara un fișier din care citim/scriem. Există 2 cazuri clasice de citire din fișier:

1. când se citesc n elemente din fișier

```
f >> n;
for (int i =1; i <=n; i++){
```

```
    f>>x;
    se prelucreaza x;
}
```

2. când nu se cunoaște numărul elementelor din fișier

```
while (f >> x) {
    se prelucreaza x;
}
```

Variabila end of file (eof) verifică dacă pe poziția curentă s-a întâlnit sau nu sfârșitul de fișier. Pornind de aici se poate utiliza formula:

```
while (!f.eof()) {
    f>>x;
    se prelucreaza x;
}
```

Când deschidem fișierul pentru citire cursorul se află la începutul acestuia și după fiecare citire se deplasează automat la sfârșitul zonei pe care a citit-o.

ofstream variabila (nume fisier)- în acest caz se crează un fișier nou ce nu conține informații. Pentru a scrie în fișier utilizăm variabila "<< expresie".

Fișierele trebuie închise după ce se termină lucrul cu acestea: variabila.close.

3.4 Ieșiri ale programelor C++

Ieșirea unui program C++ constă în valorile sau mesajele generate de program în urma execuției, furnizate prin intermediul fluxului de ieșire standard (std::cout) sau al altor mecanisme de ieșire, precum scrierea în fișiere sau transmiterea către dispozitive externe. Aceste ieșiri reflectă rezultatul procesării datelor și sunt esențiale pentru interacțiunea dintre program și utilizator ori alte componente software.

Exemplu

```
cout << "Formatarea "<<endl;

cout<<endl;

cout << "iesirilor"<<endl;
```

În exemplul de mai sus, prima linie afișează șirul de caractere "*Formatarea*", iar cu *endl* se trece la un rând nou. Următoarea instrucțiune produce o nouă trecere pe rândul următor, iar a treia instrucțiune tipărește caracterele "*iesirilor"*. Astfel rezultatul este:

```
Formatarea

iesirilor
```

Instrucțiunile de mai sus sunt echivalente cu:

```
cout << "Formatarea "<<endl<<endl<< "iesirilor"<<endl;
```

O instrucțiune C++ poate fi scrisă pe mai multe linii și compilatorul urmărește apariția semnului *;* pentru sfârșitul instrucțiunii, ci nu sfârșitul fizic al liniei.

Pentru spațierea orizontală se obișnuiește introducerea unor spații suplimentare.

Exemplu

```
cout << "Formatarea "<<' '<< "iesirilor"<<' ';
```

Această instrucțiune va afișa mesajul: *Formatarea iesirilor*.

Dacă se doresc spații mai mari, se poate opta pentru folosirea șirurilor constante care conțin spații.

Exemplu

```
cout << "Formatarea "<<”          “<< "iesirilor"<<”       ”;
```

Pentru a se afișa:

* * *

* * * *

* * * * *

se pot folosi următoarele instrucțiuni:

```
cout <<"* * *"<<endl;
cout <<"* * * * "<<endl;
cout <<"* * * * *"<<endl;
```

Spațiile situate înafara apostroafelor sau ghilimelelor nu vor fi afișate pe ecran.

Exemplu

```
cout << ‘*’ << ‘*’;
```

Se va afișa: **

3.5 Exerciții

1. Scrieți un program C++ care adună 20 de numere consecutive și calculează suma lor. Rezultatul obținut să fie de tip întreg.

2. Scrieți un program C++ care afișează pe ecran rezultatul expresiei de mai jos ca număr întreg și ca număr real.

   Expresia: 4.7 % 3 + 8.2
3. Care este rezultatul expresiei 10 / 5.0 + int (9 % 4) ?
4. Variabila x este de tip real. Care dintre următoarele expresii C/C++ are valoarea 1 dacă și numai dacă numărul real memorat în x nu aparține intervalului (2, 9] ?

   (x > 2) && (x <= 9)

   (x <= 2) && (x > 9)

   (x <= 2) || (x > 9)

   (x < 2) || (x > 9)

5. Fiecare dintre variabilele întregi x și y memorează câte un număr natural. Care dintre expresiile C/C++ de mai jos are valoarea 1 dacă și numai dacă numărul memorat în x este strict mai mare decât 0 și numărul memorat în y este strict mai mare decât 5?

   x * y - 5 != 0

   x *(y - 5) != 0

   x *(y - 5) >= 0

! (x * (y - 5) <= 0)

6. Adaugați paranteze astfel încât rezultatul expresiilor să fie de tip int:

   float 50.5 / 6 + int (4+8.2)

   int 2 - 5.0

   39.6 % 3 + 20

   40 * 2 / 5+4.5

7. Care dintre următoarele expresii C/C++ are ca rezultat cel mai mic dintre numerele naturale nenule, cu cel mult 4 cifre fiecare, memorate în variabilele întregi x și y.

   (x + y - abs (x - y)) / 2

   x + y - abs(x - y) / 2

   (x + y +abs (x - y)) / 2

   (x + y +abs (x + y)) / 2

8. Scrieți un program C++ care citește de la tastatură un număr întreg și un număr real.
9. Scrieți un program C++ care citește de la tastatură în 2 moduri diferite (folosind *cin* și *get*) 2 litere.
10. Să se deschidă un fișier și să se scrie în acesta toate numerele impare de la 4 la 100.
11. Scrieți un program C++ care citește de la tastatură o linie care conține text (mai multe caractere) folosind funcția *getline()*.
12. Se citește un șir de maxim 30 de litere mici. Să se afișeze numărul vocalelor și numărul consoanelor care apar în șir.

13. Se citește de la tastatură un șir de maxim 50 caractere care reprezintă numele și prenumele dumneavoastră, separate prin spațiu. Programul construiește în memorie și afișează un al doilea șir care conține prenumele urmat de spațiu și apoi prenumele.

    Exemplu:

    sir 1: Popescu Adrian

    sir 2: Adrian Popescu

14. Se citesc de la tastatură 10 caractere. Folosind funcția ignore, să se ignore primele 4 caractere și apoi să se afișeze caracterele rămase.
15. Să se scrie un program C++ care citește datele din fișierul "*date.in*" și scrie în fișierul *"date.out"* doar valorile pare din fișierul citit.

    Datele din fisierul *date.in* sunt:

    5 6 400 89 38 75 1000 6 27 25 90 97

16. Scrieți un program C++ care tipărește inițialele dumneavoastră.

    Exemplu: Luca Ionescu se va afișa:

    ```
    L          I
    L          I
    L          I
    L L L L L  I
    ```

17. Scrieți un program C++ care afișează rezultatul operației 3^4, folosind biblioteca *cmath*.

18. Scrieți un program C++ care are următoarea ieșire:

```
*       *       *       *       *

        *       *       *

                *
```

19. Scrieți un program C++ care copiază conținutul unui fișier sursa.txt într-un alt fișier destinație.txt. Fișierul sursă.txt conține vârsta și data dumneavoastră de naștere.
20. Scrieți un program C++ care citește de la tastatură numele și prenumele dumneavoastră și le salvează într-un fișier date.txt.
21. Scrieți un program C++ care citește de la tastatură n numere și salvează în fișierul data.txt numerele pare. Dacă numărul citit de la tastatură este un număr impar, atunci acesta nu va fi scris în fișier.

    Exemplu:

    Date citite: 2 4 84 9 91 48 36 100 46 66 58

    data.txt: 2 4 84 48 36 100 46 66 58
22. Scrieți un program C++ care citește numerele naturale din fișierul numere.txt și afișează pe ecran media aritmetică a acestora.

    Exemplu:

    numere.txt: 10 30 50

    media arithmetică este: 30
23. Scrieți un program C++ care citește de la tastatură un număr n și afișează pe prima linie a fișierului data.in primele n numere pare, pe a doua linie primele n-1 numere pare. Numerele vor fi scrise pe fiecare linie separate de spații.

    Exemplu:

n = 3

linia 1: 0 2 4

linia 2: 0 2

linia 3: 0

24. În fișierul date.in se găsesc separate printr-un spațiu mai multe valori. Să se scrie în fișierul date.out numerele din date.in fără cifrele impare. Dacă toate cifrele sunt impare, se vor elimina toate valorile.

    Exemplu:

    date.in

        25 167 395 22 43 53

    date.out

        2 6 22 4

4. Condiții și expresii logice

În exemplele prezentate anterior, instrucțiunile programului au fost executate în succesiunea în care au fost definite. Să considerăm acum un scenariu în care se impune verificarea unei anumite condiții asupra datelor de intrare, efectuarea unui calcul sau afișarea unui mesaj de eroare, fiecare dintre aceste acțiuni fiind executată în funcție de circumstanțe distincte.

Ordinea de execuție a instrucțiunilor

În limbajul C++, ordinea de execuție a instrucțiunilor este determinată de structura secvențială a codului sursă, completată de mecanismele de control al fluxului, cum ar fi instrucțiunile condiționale (if, switch), buclele de control

(for, while, do-while) și apelurile de funcții. Executarea instrucțiunilor se realizează în mod determinist, conform regulilor semantice ale limbajului, de la prima instrucțiune validă până la ultima, cu respectarea oricărei ramificații sau cicluri definite explicit în cod.

Exemplu: dacă dorim să achiziționăm o mașină cu suma de 2000$, iar suma strânsă de noi este sub această valoare atunci mașina nu poate fi achiziționată. Dacă suma de bani strânsă depășește valoarea mașinii atunci mașina poate fi cumpărată.

Condiții și expresii logice

Expresiile logice sunt folosite pentru a evalua condițiile și de a se lua o decizie pe baza rezultatului obținut. De obicei implică valori de tip bool (true sau false) și operatori logici. Operatorii logici folosiți în C++ sunt:

a. **AND logic (&&)** - acest operator returnează *"true"* dacă ambii operanzi au valoarea *"true"*, în caz contrar se returnează *"false"* - 0 este *false* și 1 este *true.*

```
if (condition1 && condition2) {
    // Code to execute if both condition1 and condition2 are true
}
```

Figura 4 -1 Exemplu AND logic

Tabel 1 AND logic

a	b	a && b
0	0	0
0	1	0
1	0	0
1	1	1

b. **OR logic (||)** - acest operator returnează *"true"* dacă unul din operanzi este *"true"* - 0 este *false* și 1 este *true.*

```
if (condition1 || condition2) {
    // Code to execute if either condition1 or condition2 is true
}
```

Figura 4-1 Exemplu OR logic

Tabel 2 OR logic

a	b	a || b
0	0	0
0	1	1
1	0	1
1	1	1

c. **NOT logic (!)** - este un operator unar care returnează valoarea *true* dacă operandul este *false*, iar dacă operandul are valoarea *false* returnează *true.*

```
if (!condition) {
    // Code to execute if condition is false
}
```

Figura 4-3 Exemplu NOT logic

Tabel 3 NOT logic

a	**!a**
0	1
1	0

d. operatori aritmetici: +, -, / , %, *

Operatorul "%" poate fi aplicat doar valorilor întregi.

Operatorul "/" - dacă este utilizat pentru numere întregi, produce un rezultat întreg, iar dacă unul dintre operanzi este real se produce un rezultat real.

e. operatorul de atribuire "="

Are forma: variabilă = expresie

Se pot realiza atribuiri multiple sub forma: variabila_1 = variabila_2= … =expresie

f. operatori relaționali - aceștia compară 2 valori și returnează un rezultat de tip *boolean*. Aceștia includ operatorii:
   - egal (==)
   - diferit (!=)
   - mai mare (>)
   - mai mic (<)
   - mai mare sau egal (>=)
   - mai mic sau egal (<=)

```
if (a == b) {
    // Code to execute if a is equal to b
}
```

Figura 4-4 Exemplu de folosire a operandului egal

g. operatori pe biți - se pot folosi în expresiile logice și include:
   - & (AND pe biți)
   - | (OR pe biți)
   - ^ (XOR pe biți)
   - ~ (NOT pe biți)

Expresiile logice sunt folosite în condiții, în bucle sau în alte structuri de control.

4.1 Exerciții

1. Ce rezultat au următoarele expresii logice (true sau false) unde x = 1, y = 0 și z = 0 ?
    a. x && y || z
    b. x || z
    c. !x && (y && z)
    d. ! (x || z)
    e. !y || y

2. Adăugați paranteze (dacă este cazul) astfel încât următoarele expresii logice să fie adevărate pentru a = 0, b = 1, c = 0.
    - a || b
    - !a && b
    - a || b && c
    - c && !b

3. Cum sunt următoarele expresii logice adevărate sau false? Unde a = 3, b = 10, c = 5
    - ! (a == b)
    - ! (c > 8)
    - ! (a < c)
    - a >= b
    - a <= (b + 26)

5. Instrucțiuni

O instrucțiune în C++ reprezintă o unitate fundamentală de execuție a unui program, constând într-o comandă care indică procesorului o acțiune de realizat, cum ar fi: declararea unei variabile, efectuarea unui calcul, controlul fluxului de execuție sau apelul unei funcții. Instrucțiunile sunt evaluate secvențial sau în funcție de structuri de control și se încheie, în general, cu punct și virgulă (;).

5.1 Instrucțiunea if

Instrucțiunea *IF* se folosește pentru a se executa una sau mai multe instrucțiuni pe baza rezultatului unei condiții logice. În C++ instrucțiunea *if* poate fi folosită în 2 variante: IF - THEN sau IF - THEN - ELSE. Instrucțiunea *if* folosește 2 cuvinte cheie: **if** și **else**.

Uneori se dorește să se execute anumite acțiuni doar când condiția este îndeplinită, dar să nu se întâmple nimic când aceasta este falsă.

Exemplu:

```
if (x < y )
 z = 20;
```

Ramura IF-THEN poate fi reprezentată de un bloc de instrucțiuni.

Exemplu:

```
if (x < y) {
    cout<< “conditia este adevarata”;
    z = 20;
    y = x;
```

```
}
```

În cazul în care se uită folosirea acoladelor, compilatorul va executa doar prima instrucțiune dacă condiția este adevărată. Pentru exemplul de mai sus, se va executa doar instrucțiunea cout<< “conditia este adevarata”; dacă lipsesc acoladele.

Sintaxa în C++ pentru IF - THEN: *if (conditie)* { // se va executa dacă condiția este adevărată }

Dacă condiția este adevărată atunci se vor executa instrucțiunile dintre acolade {}, iar dacă condiția este falsă, codul dintre acolade nu se va executa și se va trece la următoarea instrucțiune.

Sintaxa în C++ pentru IF - THEN - ELSE:

if (expresie)

Instrucțiune A

else

Instrucțiune B

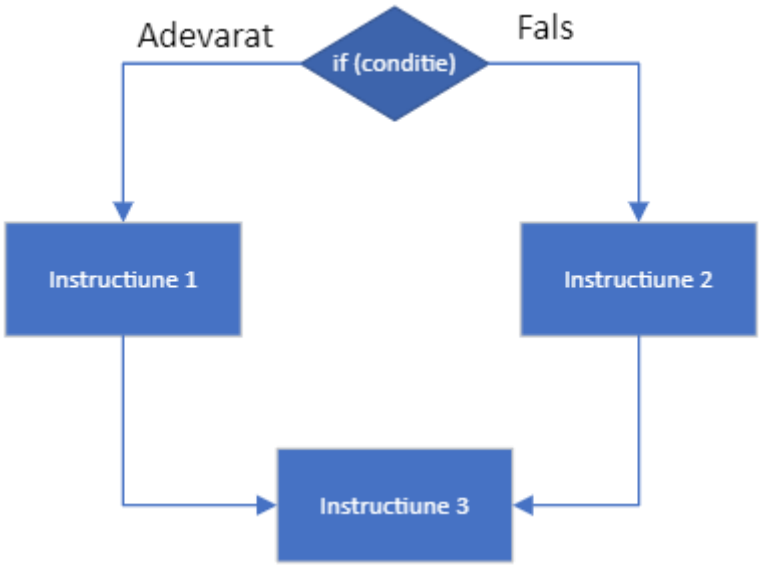

Figura 5.1-1 Schema logică pentru instrucțiunea IF - THEN - ELSE

Dacă expresia este adevărată (*true*) atunci se va executa Instrucțiune 1, iar dacă expresia este falsă (*false*) atunci se va executa Instrucțiune 2.

Se poate folosi instrucțiunea *if* imbricată (*if* în *if*).

Exemplu:

```
float a = 4;

float b = 5;

if ( a<12)  a= a + 2;

        else if (a == 0) a = a+4;

                else a = a-3;
```

5.2 Instrucțiunea switch

În C++, instrucțiunea *switch* este folosită pentru a executa un bloc de cod. Această instrucțiune presupune implementarea structurilor de control cu ramnificare multiplă. Valoarea expresiei determină ramura care se va executa.

Instrucțiunea *switch* este formată din una sau mai multe ramuri *case* și o ramură *default*.

Sintaxa instrucțiunii *switch* este prezentată în imaginea de mai jos, unde:

- expresia - valoarea care este comparată cu cazurile predefinite.
- *case constant 1, case constant 2* : etichetele care indică blocurile de cod care se vor executa dacă expresia are aceeași valoare cu constantele.
- breack - după execuția fiecărui bloc de cod, trebuie să se iasă din instrucțiunea switch. Dacă acesta lipsește, se va continua cu execuția următorului caz (*en.* case).
- default - este o opțiune opțională și se va executa dacă niciunul din cazurile predefinite nu este îndeplinit și executat. Este echivalent cu funcția “else” din instrucțiunea “if-else”.

```
switch(expression) {
    case constant1:
        // code to be executed if expression matches constant1
        break;
    case constant2:
        // code to be executed if expression matches constant2
        break;
    // more cases can be added as needed
    default:
        // code to be executed if expression doesn't match any constant
}
```

Figura 5.2-1 Sintaxa instrucțiunii switch

Exemplu:

```
#include <iostream>

int main (){

int nr = 5;

switch (nr) {

case 1:

                cout<<"Numarul este 1";

 break;

case 2:

                cout<<"Numarul este 2";

        break;
```

```
            case 3:
                        cout<<"Numarul este 3";
                break;
            case 4:
                        cout<<"Numarul este 4";
                break;
            case 5:
                        cout<<"Numarul este 5";
                break;
            default:
                        cout<<"Numarul nu este intre 1 si 5";
            }
        return 0;
    }
```

În exemplul de mai sus, dacă valoarea variabilei *nr* este 5, atunci se va afișa mesajul “Numarul este 5”, dar dacă valoarea variabilei este 6, atunci se va afișa mesajul "Numarul nu este intre 1 si 5".

În instrucțiune switch se poate folosi o singură etichetă *default*. În caz contrar compilatorul va genera erori de sintaxă.

5.3 Instrucțiunea while

Instrucțiunea *while* folosește bucle. O buclă este o structură de control care provoacă executarea unei instrucțiuni sau a unei secvențe de instrucțiuni în mod repetat atâta timp cât sunt îndeplinite una sau mai multe condiții.

Fazele de execuție ale unei bucle sunt:

1. Intrarea în buclă - se ajunge la prima instrucțiune din buclă
2. Iterarea - când se execută o buclă se numește iterație.
3. Testul buclei - se evaluează expresia din *while*. Pe baza rezultatului obținut se ia decizia dacă se va continua cu o nouă iterație sau de a se trece la instrucțiunea imediat următoare buclei.
4. Condiția de terminare - reprezintă condiția care are ca rezultat ieșirea din buclă și trecerea la instrucțiunea următoare buclei.
5. Ieșirea din buclă - apare când expresia din instrucțiunea *while* este falsă, moment în care se întrerupe repetarea corpului buclei.

Sintaxa în C++: *while* (expresie) Instrucțiune

Exemplu:

```
while (a <= b) cin >> a;
```

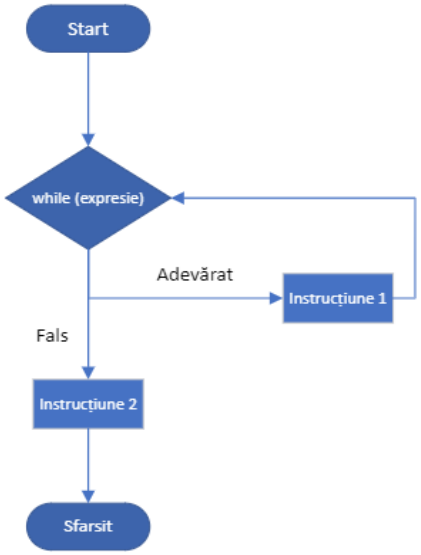

Figura 5.3-1 Schema logică pentru instrucțiunea while

Corpul buclei poate fi format din mai multe instrucțiuni care ne permit să executăm mai multe instrucțiuni în mod repetat.

Exemplu:

```
while (a <= b) {
    cin >> a;
    b++;
    cout <<"b=" <<b;
        }
```

O buclă infinită este o buclă din care programul nu poate ieși (bucla se repetă la infinit).

Exemplu:

```
while (a > 0 ) cout << a ;
```

Buclele pot fi controlate de:

a. un contor - se folosește o variabilă numită variabilă de control a buclei. Aceasta este inițializată înaintea buclei și se incrementează la fiecare iterație.

   Exemplu:

```
int contor = 0;
while (contor < 5) {
        cout<< "contor = "<< contor;
        contor++;
}
```

b. de un eveniment - poate depinde de o valoare semnalizată (santinelă) sau de sfârșitul de fișier (EOF)

   Exemplu:

```
int zi, luna;
cin >> luna>> zi; // se citește primul set de date
while (zi != 12 && luna != 4) //se verifică dacă datele introduse corespund
{ …
cin >> luna >> zi; // se citește următorul set de date
```

```
}
```

Pentru date de tip *char* se poate folosi caracterul *newline* ca valoare de semnalizare.

Exemplu EOF (end of file): Presupunem că avem un fișier care conține numere întregi, unde val reprezintă valoarea citită din fișier, iar inData este un fișier din care se citesc datele.

```
int val;
inData >> val;
while (val) {
   cout <<val << ' ';
   inData >> val;
      }
```

Avem programul de mai jos care adună 5 numere:

```
#include <iostream>
using namespace std;
int main () {
int suma = 0;  // se inițializează suma cu 0
int contor = 1;  // se inițializează contorul cu 1
int nr;
while (contor <= 4) {
   cin >> nr;  //se citește de la tastatură un număr întreg
```

```
            suma = suma + nr;  /* se adună numărul citit de la
tastatură la sumă și suma primește o nouă valoare */
            contor ++;  // se incrementează contorul
        }
        cout << "Suma este" << suma; //afișare
        }
```

După executarea buclei, variabila *suma* va fi formată din suma a 4 valori citite de la tastatură, variabila contor va conține valoarea 5 și variabila *nr* va conține ultima valoare citită de la tastatură.

Se pot folosi instrucțiuni while imbricate. Fiecare buclă are propria inițializare, testare și actualizare.

Exemplu:

```
int a, b,c;
a = 10;
while (a < 20) {
   cin >> b ;
   while (b > a) c++;
   a++;
}
```

Întotdeauna trebuie ca la un moment dat expresia logică să devină *falsă*, dacă nu se intră într-un *ciclu infinit*.

5.4 Instrucțiunea do-while

În C++, instrucțiune do-while este o buclă care execută un bloc de cod în mod repetat până când o condiție definită devine falsă. Diferența majoră dintre o buclă *while* și o buclă *do-while* este că bucla do-while asigură executarea blocului de cod cel puțin odată.

Sintaxa buclei do-while este prezentată mai jos, unde:

- ➔ codul dintre acolade *{}* va fi executat prima dată
- ➔ se va evalua condiția *while*
- ➔ dacă condiția este adevărată, atunci se va executa încă odată bucla
- ➔ dacă condiția este falsă, atunci bucla se va termina și programul va continua execuția codului care se află după bucla *do-while*.

```
do {
    // block of code to be executed
} while (condition);
```

Figura 5.4-1 Sintaxa instrucțiunii do-while

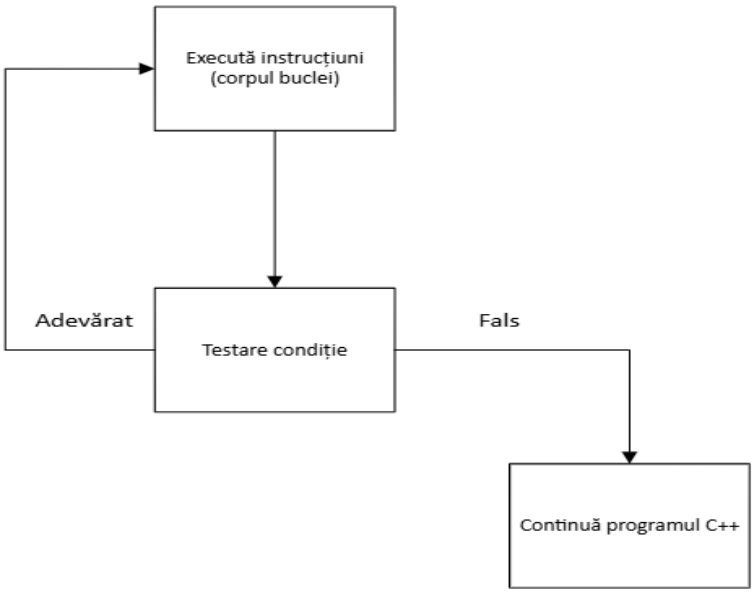

Figura 5.4-2 Schema logică pentru instrucțiunea do-while

Exemplu:

```
#include <iostream>
int main (){
   int nr = 4;
   do {
      cout<<"Valoarea variabilei nr este: "<<nr;
      nr++;
   }while (nr <10)
return 0;
}
```

În exemplul de mai sus, variabila *nr* este inițializată cu 0, apoi se execută ceea ce este în interiorul buclei do și anume se afișează mesajul "Valoarea variabilei nr este: " și se incrementează valoarea variabilei *nr* cu 1. Pe urmă se evaluează condiția și se va executa bucla până când condiția devine falsă (până când valoarea lui *nr* este mai mică ca 10).

5.5 Instrucțiunea for

Instrucțiunea for simplifică scrierea și utilizarea buclelor controlate, aceasta fiind o formă mai compactă a buclei *while*. Instrucțiunea *for* permite executarea codului în mod repetat pentru un anumit număr fix de ori. Această instrucțiune este potrivită când se știe exact numărul de iterații necesare.

Sintaxa instrucțiunii *for* este prezentată mai jos, unde:

- inițializare (en. initialization) - reprezintă inițializarea contorului folosit în buclă. Se execută la începerea executării codului.
- condiție (en. condition) - reprezintă o expresie de tip bool care este evaluată înaintea fiecărei iterații. Dacă aceasta este evaluată ca fiind adevărată, atunci executarea buclei va continua, iar daca aceasta este evaluată ca fiind falsă, se va termina bucla.
- incrementare/decrementare (*en*. increment/decrement): se folosește pentru a se modifica valoarea variabilei de control. Se execută după fiecare iterație a buclei.
- blocul de cod dintre acolade *'{...}'* - se va executa repetat cât timp condiția este adevărată.

```
for (initialization; condition; increment/decrement) {
    // block of code to be executed
}
```

Figura 5.5-1 Sintaxa instrucțiunii for

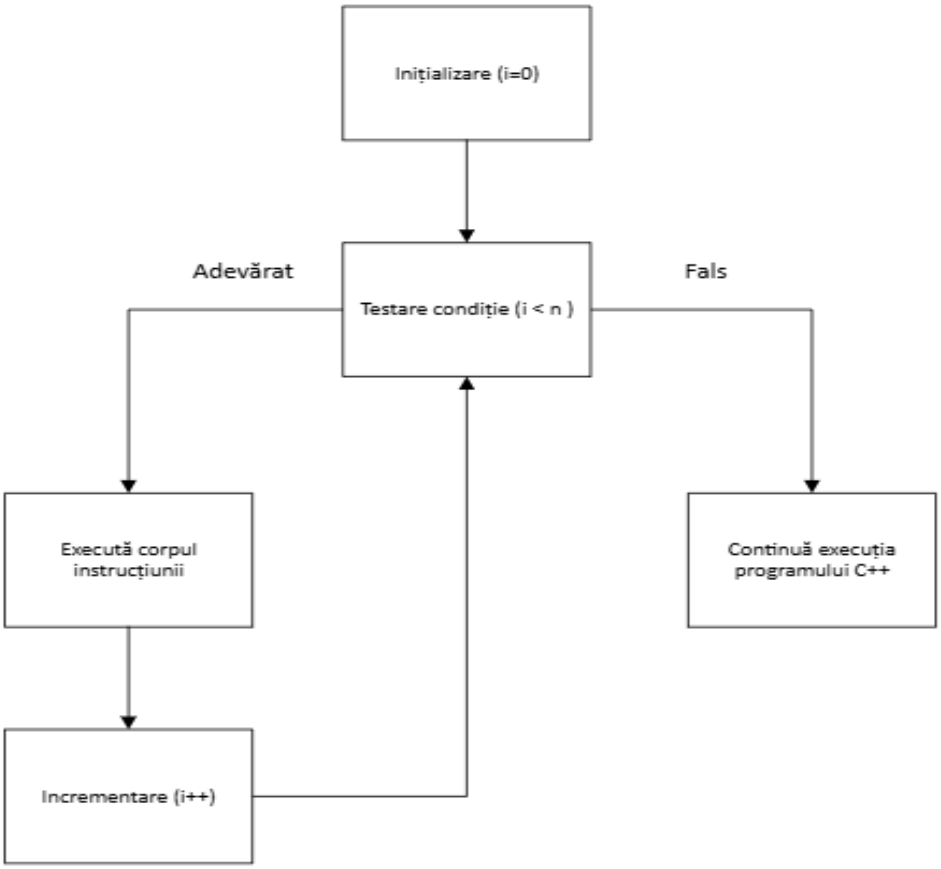

Figura 5.5-2 Schema logică pentru instrucțiunea for

Exemplu:

```
#include <iostream>
```

```
int main() {
   for (int i=1; i<=10; i++)
      cout<<i<<endl;
}
```

În exemplul de mai sus, *i* reprezintă variabila de control și este inițializată cu 1 (int i=1), după care se execută bucla atâta timp cât valoarea lui *i* este mai mică sau egală cu 10 (i <= 10). După fiecare iterație, variabila *i* este incrementată (i++). Bucla afișează pe ecran valoarea variabilei *i*. Atunci când *i* are valoarea 11, condiția i <= 10 este falsă și se iese din bluclă.

Ieșirea programului este:

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

5.6 Instrucțiunile break și continue

În C++, *break* și *continue* sunt instrucțiuni de flux de control utilizate în bucle pentru a modifica fluxul de execuție.

1. **Break**

Când este întâlnită în interiorul unei bucle, instrucțiunea *break* face ca bucla să se termine imediat, iar controlul programului se reia la următoarea instrucțiune care urmează buclei.

Exemplu

```
for (int i=0; i<10; i++){
   if (i==5)
      break;

   cout<<i<<endl;
   }
```

În exemplul de mai sus, cânt variabila *i* are valoarea 5, se execută instrucțiunea "break". Aceasta cauzează terminarea buclei și doar valorile de la 1 la 4 vor fi afișate pe ecran.

2. **Continue**

Când este întâlnită în interiorul unei bucle, instrucțiunea *continue* face ca bucla să sară peste restul iterației curente și să continue cu următoarea iterație.

Exemplu

```
for (int i=0; i<10; i++){
   if (i % 2 == 0)
      continue;

   cout<<i<<endl;
   }
```

În acest exemplu, când variabila *i* este pară, instrucțiunea *continue* se execută determinând bucla să sară peste imprimarea acelui număr și să treacă la următoarea iterație.

Diferența dintre *break* și *continue* este că *break* întrerupe imediat bucla curentă, iar *continue* întrerupe iterația curentă.

Instrucțiunile *break* și *continue* pot fi folosite în buclele *for, while* și *do-while* pentru a controla execuția codului având la bază niște condiții. Acestea sunt folositoare pentru buclele complexe și pentru optimizarea execuției buclelor atunci când sunt îndeplinite anumite condiții.

5.7 Exerciții

1. Scrieți un program C++ care citește de la tastatură un număr de 3 cifre și verifică dacă numărul citit este par sau impar. Dacă numărul este par, să se afișeze pe ecran un mesaj (ex. Numărul este par), iar dacă numărul este impar, să se afișeze un alt mesaj (ex. Numărul este impar).

2. Scrieți un program C++ care citește de la tastatură 2 numere reale și diferite, le ordonează crescător și le afișează.

3. Modificați următorul program C++ astfel încât acesta să afișeze valoarea 40.

```
int a = 4;
int b = 10;
int c = 40;
if (a > b) { a = c ;
cout <<"Valoarea lui a este:" <<a;}
else
{ b = 6;
cout <<"Valoarea lui b este:  "<<b; }
```

4. Identificați și corectați greșelile programului următor:

```
int x, y, z;x = 'c';
y = 'hello';
z = 6.8
if (x < y) cout <<z;
```

5. Ce afișează următorul program?

```
int m = 10;
```

```
int n = 15;
int o, p;

if (m >= n && m == 10)
{
  o = m - n;
cout << " o = " << o;
}
else
{
  p = n * n;
  cout << " p = " << p;
}
```

6. Să se scrie un program C++ care să verifice dacă suma a 2 numere este egală cu 100 și produsul celor 2 numere este egal cu 900. În caz afirmativ să se afișeze pe ecran mesajul “Cele 2 numere respectă condițiile”, iar în caz contrar să se afișeze pe ecran mesajul “Numerele nu sunt potrivite”.

7. Se citește o variabilă a care reprezintă un an calendaristic. Să se afișeze dacă a este an bisect sau nu (divizibil cu 4).

8. Se citesc 3 numere naturale a, b, k. Să se verifice dacă a/b se simplifică cu k și în caz afirmativ se va afișa fracția simplificată.

9. Rescrieți următorul fragment de cod folosind instrucțiunea *switch*:

```
int n, a, b, c;
if (n == 0)
   a = a+2;
   else if (n == 2)
      b++;
   else if (n == 4)
      c = c+3;
```

10. Ce afișează următorul fragment de cod, dacă *n* are valoarea 6? Dar dacă *n* are valoarea 10?

```
switch (n)
{
        case 1: cout <<"Mere";
        case 2: cout<<"Pere";
        case 3: cout<<"Cirese";
```

```
        case 4: cout<<"Visine";
        case 5: cout<<"Pepene";
        case 6: cout<<"Banane";
        default: cout <<"Alte fructe";
    }
```

11. Scrieți o instrucțiune *switch* care să implementeze următoarea secvență:

    Dacă valoarea variabilei lit este:

    'A' - se afișează "Litera este A"

    'B' - se afișează "Litera este B"

    'C' - se afișează "Litera este C"

    'D' - se afișează "Litera este D"

    Dacă valoarea variabilei lit este o altă valoare se va afișa mesajul "S-a executat ramura default".

12. Ce afișează următoarea secvență de program? Câte iterații se fac?

```
int m = 5;
int n, p;
while (m <= 5)
   {cin >> n;
   while (n != 4) p++;
   cout << "p= "< <p <<", n="<< n << endl;
```

```
}
```

13. Folosind instrucțiunea while, să se calculeze suma și produsul a 10 numere consecutive.

14. Ce tipărește bucla următoare?

```
int nr = 2;
while (number < 15){
        nr++;
        cout << nr << endl;
}
```

15. Rearanjați instrucțiunile de la exercițiul anterior astfel încât să se afișeze numerele de la 3 la 20.

16. Folosind instrucțiunea while, scrieți un program care citește dintr-un fișier și care numără de câte ori apare valoarea 70 în fișier.

17. Folosind instrucțiunea while, să se afișeze câte numere pozitive și câte numere negative se găsesc în fișier. Se va ignora valoarea 0.

18. Se citește de la tastatură un număr a cu maxim 9 cifre. Să se afișeze produsul cifrelor sale impare și suma cifrelor pare. se va folosi instrucțiunea while.

19. Scrieți un program care să afișeze următorul rezultat. Se va folosi instrucțiunea while.

    1 2 3 4 5

    1 2 3 4

    1 2 3

    1 2

    1

20. Scrieți un program C++ care să genereze următorul triunghi de numere.

    1

    1 3

    1 3 5

    ------

    1 3 5 ……. 2*n-1

21. Să se citească de la tastatură un număr întreg și să se calculeze suma pătratelor numerelor de la 1 până la acel număr. Exemplu: numărul citit este 6, iar suma va fi:

    sum = 1*1+2*2+3*3+4*4+5*5+6*6 = 1+4+9+16+25.

Se va folosi instrucțiunea while.

22. Folosind instrucțiunea *do-while*, rescrieți programul de la exercițiul precedent.

23. Ce va afișa următorul cod dacă toate variabilele sunt de tip *int*?

```
#include <iostream>
int main() {
   for (int i=1; i<=10; i++){
      for (int j=1; j<=i; j++)
         cout<< '*';
      for (int k=0; k<=i/2; k++)
         cout<<'-';
      for (int m=1; m<i-1; m++)
         cout<<' . '<<endl;
      cout<<endl;
   }
}
```

24. Rescrieți următorul cod folosind instrucțiunea *for*.

```
         sum = 0;
```

```
count = 1;
while (count <=200)
{
   sum = sum+count;
   count++;
}
```

25. Rescrieţi următoarea instrucţiune *for* folosind o buclă *do-while.*

```
for (m = 20; m<= 50; m++)
  cout << m << ' ' << 3 * m << endl;
```

26. Scrieți un program C++ care citește de la tastatură 2 litere care reprezintă aprevierea prescurtată a județelor și care afișează numele complet al județului. Dacă abrevierea nu este validă, programul va afișa un mesaj de eroare și va citi încă o abreviere.

    Sugestie: se va folosi instrucțiunea *switch, while sau for.*

27. Scrieți un program C++ care citește numele și prenumele unei persoane și care afișează inițialele pe ecran și apoi într-un fișier numit *initiale*.

28. Transformați următoarea secvență de cod prin introducerea unor bucle infinite terminate cu instrucțiuni break, astfel încât el să poată fi mai

ușor de înțeles. Scrieți un program complet pentru a rula noua secvență de cod.

```
sum = 5;
count = 1;
do
{
 cin >> int1;
 if( !cin || int1 <= 0)
  cout << “Primul intreg este incorect.”;
 else
 {
  cin >> int2;
  if(!cin || int2 > int1)
   cout << “Al doilea intreg este incorect.”;
  else
  {
   cin >> int3;
   if(! cin || int3 == 0)
    cout << “Al treilea intreg este incorect.”;
   else
```

```
        {
          sum = sum + (int1 +int2) / int3;
          count++;
        }
      }
    }
  } while (cin && int1 > 0 && int2 <= int1 && int3 != 0 && count <=
  100);
```

29. Folosind instrucțiunea for, să se afișeze suma valorilor pare din intervalul [a,b] (unde a și b se vor citi de la tastatură). Se va face o verificare astfel încât a < b.

6. Subprograme (Funcții)

Subprogramele (sau funcțiile) în C++ sunt secțiuni de cod care pot fi apelate din alte părți ale programului pentru a îndeplini o anumită sarcină sau operație. Acestea permit reutilizarea codului, modularizarea și îmbunătățirea claritații și întreținerii programului. În C++, o funcție este un bloc de cod care execută o anumită acțiune, returnând o valoare sau nu. Funcțiile sunt folosite pentru a evita repetarea codului, îmbunătățind astfel eficiența și organizarea programului.

O funcție în C++ este definită de următoarele componente:

1. tipul de retur (Return Type): specifică tipul de date pe care funcția îl returnează (dacă returnează ceva). Dacă funcția nu returnează nimic, se folosește *void*.
2. numele funcției (Function Name): identificatorul prin care va fi apelată funcția.
3. parametrii (Arguments): lista de variabile pe care funcția le primește pentru a executa operațiile sale. Parametrii sunt opționali. Dacă nu există parametri, se folosește o pereche de paranteze rotunde ().
4. corpului funcției (Function Body): secțiunea între acolade care conține instrucțiunile pe care funcția le execută.
5. valoarea returnată (Return Value): dacă funcția este de tipul care returnează o valoare, aceasta va returna un rezultat folosind cuvântul cheie return.

Sintaxa unei funcții este:

```
return_type function_name(parameters) {
```

```
    // corpul funcției
    // instrucțiuni
    return return_value; // Dacă este cazul
}
```

O funcție trebuie definită înainte de a fi apelată. Dacă acest lucru nu se dorește sau nu se poate realiza atunci este suficient să precizăm antetul funcției urmate de punct și virgulă apoi urmând să definim funcția oriunde în program.

Exemplu de funcție în C++

```
#include <iostream>
using namespace std;
// Definirea unei funcții care adună două numere
int aduna(int a, int b) {
    return a + b;  // Returnează suma celor două numere
}
int main() {
    int rezultat = aduna(5, 3);  // Apelăm funcția 'aduna'
    cout << "Rezultatul este: " << rezultat << endl;  // Afișăm rezultatul
    return 0;
}
```

Există 2 categorii de funcții:

1. funcții care returnează un rezultat
2. funcții care nu returnează nicio valoare (execută anumite operații) și spunem că sunt de tip ***void.***

Orice funcție este formată din antetul funcției și corpul funcției. Antetul unei funcții reprezintă tipul de dată returnat *nume ();* sau tip de date returnat *nume(parametri formali);*

În corpul funcției, dacă aceasta nu este de tip void, întâlnim instrucțiunea: **return variabilă**, care reprezintă valoarea rezultată a funcției. La întâlnirea acestei instrucțiuni se iese din corpul funcției și se returnează programului apelant valoarea obținută.

O funcție ca să se execute efectiv, trebuie apelată.

6.1 Funcții void

Funcțiile de tip *void* sunt funcții care nu returnează valori. Se folosesc pentru situațiile când se dorește să se execute anumite acțiuni sau task-uri fără a fi nevoie să se returneze o valoare.

O funcție *void* nu conține nicio instrucțiune de tipul **return** 0**;**

Exemplu de funcție de tip void

```
#include <iostream>

// definirea functiei void

void greet() {

   std::cout << "Hello, world!" << std::endl;

}
```

```
int main() {
   // apelarea functiei void
   greet();  // acest apel de funcție nu returnează nicio valoare
   return 0;
}
```

În exemplul de mai sus, funcția numită “greet()” este o funcție void deoarece nu returnează nicio valoare. Această funcție doar afișează în consolă un mesaj “Hello world!” atunci când este apelată.

Într-un program C++ pot exista mai multe funcții de tip *void*. Definiţiile funcţiilor pot apărea în orice ordine.

Apelul funcțiilor înseamnă executarea corpului funcției apelate. O funcție poate fi apelată astfel:

Nume_Functie (Lista_Parametri_Actuali)

unde *parametrii actuali* sunt parametrii dintr-un apel de funcție. Parametrii care apar în header-ul funcției se numesc *parametrii formali*. În C++, lista parametrilor actuali poate fi vidă, iar dacă lista conține 2 sau mai mulți parametri, aceștia trebuie separați prin virgulă.

Atunci când se apelează o funcție, valoarea parametrilor actuali este transmisă parametrilor formali conform poziției lor, de la stânga la dreapta, apoi se trece la prima instrucțiune din corpul funcției. Când se termină de executat tot corpul funcției, controlul este transmis punctului în care s-a apelat funcția.

Simbolul *"&"* este atașat tipului de dată și este opțional.

6.2 Variabile locale și globale

a. Variabile locale

Aceste variabile sunt vizibile doar în blocul de cod unde au fost declarate sau doar în funcția în care au fost declarate. De obicei, sunt folosite pentru a stoca temporar valorile necesare unei părții din program.

Exemplu de program C++ care folosește variabile locale

```
#include <iostream>

void myFunction() {
   int localVar = 5; // Declarare și inițializare a unei variabile locale
   std::cout << "Variabilă locală în myFunction: " << localVar << std::endl;
}

int main() {
   int localVar = 10; // Declarare și inițializare a altei variabile locale
   std::cout << "Variabilă locală în main: " << localVar << std::endl;
   myFunction(); // Apelarea funcției care are propria sa variabilă locală
   // localVar din myFunction nu este accesibil aici, deoarece este în afara functiei
```

```
    return 0;
}
```

În exemplul de mai sus, sunt definite 2 funcții: myFunction() și main(). Fiecare funcție are propriile variabile numite “localVar”. Deși au același nume, cele 2 variabile sunt separate, independente și fiecare variabilă poate fi accesată doar în funcția în care a fost definită.

b. Variabile globale

Aceste variabile sunt declarate în afara funcțiilor și sunt accesibile pentru orice parte a programului, incluzând funcțiile. Nu se recomandă folosirea variabilelor globale din cauza potențialelor probleme ce pot apărea în modificarea neintenționată și dificultăți în depanarea și menținerea codului.

Exemplu de program C++ care folosește variabile globale

```
#include <iostream>
// Declararea unei variabile globale
int globalVar = 10;
void myFunction() {
    // Accesarea variabilei globale în interiorul funcției
    std::cout << "Variabilă globală în myFunction: " << globalVar << std::endl;
}
int main() {
    // Accesarea variabilei globale în funcția main
```

```
    std::cout << "Variabilă globală în main: " << globalVar << std::endl;
    // Modificarea variabilei globale
    globalVar = 20;
    // Apelarea unei funcții care accesează variabila globală
    myFunction();
    return 0;
}
```

În exemplul de mai sus, variabila "globalVar" este definită ca variabilă globală, în afara funcțiilor. Această variabilă poate fi accesată atât în funcția *main()* cât și în funcția *myFunction()* fără a exista erori sau alte probleme. Folosirea variabilelor globale poate duce la îngreunarea înțelegerii și menținerii codului în special în programele complexe.

6.3 Parametri

Parametrii unei funcții sunt variabile care sunt definite în cadrul unei funcții și care sunt folosite pentru a primi datele (valorile) transmise la apelul funcției. Aceștia permit funcțiilor să lucreze cu valori externe și să efectueze operații pe ele. În C++, funcțiile pot avea diferite tipuri de parametri, în funcție de modul în care sunt transmise valorile către funcție. Parametrii sunt definiți atunci când funcția este declarată sau definită.

Numărul parametrilor actuali din apelul unei funcții trebuie să fie egal cu numărul parametrilor formali din header-ul funcției. De asemenea, tipurile

datelor trebuie să corespundă. Dacă tipurile de dată nu se potrivesc, compilatorul încearcă să aplice operațiile de cast. Pentru a evita aceste conversii implicite de tip, se pot aplica și conversii explicite.

Parametri formali - se declară în mod asemănător cu variabilele și au rolul de a spori gradul de reutilizare a codului scris. Aceștia descriu anumite operații ce se execută în funcțiile unde apar. Orice parametru formal trebuie precedat de tipul său de dată.

Exemplu:

```
void f (int a, int b)
{ ... }
```

Parametri formali sunt de 2 tipuri:

a. **parametri formali transmiși prin valoare**

În acest caz se realizează o copie a parametrilor cu care se apelează funcția, se prelucrează copia, iar originalul rămâne neschimbat.

Se pot folosi constante, variabile sau expresii în apelul funcției deoarece parametrii valoare primesc copii ale parametrilor actuali.

La încheierea funcției, conținutul parametrilor valoare este șters.

Exemplu de funcție cu parametru valoare

```
#include <iostream>
// Funcție care primește un parametru prin valoare
void increment(int x) {
   x++; // Incrementează valoarea lui x
```

```
    std::cout << "În interiorul funcției: " << x << std::endl;
}
int main() {
    int num = 5;
    std::cout << "Înainte de apelul funcției: " << num << std::endl;
    // Apelăm funcția, transmițând num ca argument
    increment(num);
    std::cout << "După apelul funcției: " << num << std::endl;
    return 0;
}
```

În programul de mai sus, funcția increment **(**int x) are un parametru valoare numit **x.** Când funcția **increment(num)**este apelată în funcția **main(),**valoarea variabilei **num** este copiată în **x** din funcția **increment()**. Orice modificare a variabilei **x** în interiorul funcției, nu va afecta variabila **num**.

Programul va afișa:

```
Before function call: 5
Inside function: 6
After function call: 5
```

Se poate observa că, deși **x** a fost incrementat în cadrul funcției **increment()**, variabila **num** inițială din **main()** rămâne neschimbată. Acest lucru se datorează faptului că **x** este o copie locală a lui **num** în cadrul funcției.

Dacă avem 2 variabile locale cu același nume în 2 funcții diferite fiecare este cunoscută în interiorul funcției în care a fost definită și acționează independent față de cealaltă variabilă cu același nume. Două variabile locale cu același nume în aceeași funcție produc eroare.

b. parametri formali transmiși prin referință

În acest caz se lucrează direct cu variabila cu care se apelează funcția și dacă aceasta se modifică în funcție, va rămâne modificată în restul programului. Pentru variabilele simple, parametrii transmiși prin referință conțin un **&** între tipul de dată și identificator.

Exemplu de funcție cu parametru referință

```
#include <iostream>

// Funcție care primește un parametru prin referință

void modifyValue(int& num) {

   num *= 2; // Înmulțim valoarea cu 2

}


int main() {

   int x = 5;
```

```
    std::cout << "Înainte: " << x << std::endl; // Afișează: 5
    // Transmitem x prin referință către funcția modifyValue
    modifyValue(x);
    std::cout << "După: " << x << std::endl; // Afișează: 10
    return 0;
}
```

În exemplul de mai sus, funcția modifyValue are un parametru referință (int**&** num). La apelarea funcției, x este trecut ca referință, deci orice modificarea asupra variabilei num pe parcursul funcției modifyValue este aplicată direct variabilei x.

Parametri prin referință sunt eficienți și permit funcției să modifice argumentele funcției direct.

Vectorii și matricele se transmit prin referință.

Nu putem apela cu o constantă în locul unui parametru formal transmis prin referință.

Variabilele cu care se apelează la un moment dat o funcție se numesc parametrii actuali sau efectivi.

Dacă numele unei variabile globale este identic cu cel al unei variabile locale atunci în funcția unde am declarat variabila locală se lucrează cu aceasta și nu cu variabila globală. În restul programului se lucrează cu variabilă globală.

Numele unui parametru formal nu trebuie să coincidă cu o variabilă locală a aceleași funcții.

Instrucțiunea *return* întrerupe automat execuția unui subprogram în momentul în care se execută. Return este capabilă să returneze o singură valoare de un tip simplu de dată. În cazul unor instrucțiuni If-Else, într-o funcție ce nu returnează void, fiecare ramură trebuie să returneze o expresie.

6.4 Funcții recursive

O funcție recursivă este o funcție care se autoapelează pentru a rezolva o problemă. Recursivitatea este o tehnică de programare în care o problemă mare este împărțită în subprobleme mai mici, care sunt identice în structură cu problema originală. Funcțiile recursive continuă să se autoapeleze până când ajung la un caz de bază, care este simplu de rezolvat și nu implică apeluri suplimentare.

Componentele unei funcții recursive sunt:

1. cazul de bază: este condiția care oprește recursivitatea. Fără un caz de bază, recursivitatea ar continua la nesfârșit, ceea ce ar duce la o eroare de tipul stack overflow.
2. apelul recursiv: este partea din funcție în care funcția se autoapelează pentru a rezolva o subproblemă mai mică.

Exemplu de funcție recursivă - Funcția factorială

```
#include <iostream>

// Funcție recursivă pentru calcularea factorialului unui număr

int factorial(int n) {
```

```
    // Cazul de bază: factorial(0) este 1
    if (n == 0) {
        return 1;
    }
    // Cazul recursiv: n! = n * (n - 1)!
    else {
        return n * factorial(n - 1);
    }
}
int main() {
    int num = 5;
    std::cout << "Factorialul lui " << num << " este " << factorial(num) <<
std::endl;
    return 0;
}
```

În exemplul de mai sus, funcția factorială se autoapelează folosindu-se un argument din ce în ce mai mic până când argumentul devine 0. Atunci când argumentul devine 0, funcția se oprește și se returnează valoarea 1.

Funcțiile recursive oferă un mod de lucru elegant, dar sunt mai puțin eficiente decât soluțiile iterative pentru anumite probleme/cazuri. Pot cauza supraîncărcarea memoriei dacă nu sunt folosite corect.

6.5 Exerciții

1. Scrieți un program C++ care să conțină o funcție de tip void. Funcția va afișa mesajul: “S-a apelat funcția de tip void”.

2. În programul de la exercițiu anterior, adaugați o funcție de tip void care are 2 parametri. Funcția va afișa mesajul: “ *** ”

3. Scrieți un program C++ care să conțină 2 variabile globale și mai multe variabile locale. Programul va calcula suma a 30 de numere consecutive, unde numărul de început va fi citit de la tastatură.

4. Scrieți o funcție care scalculeaza media aritmetică a 5 numere consecutive. Funcția are un parametru transmis prin valoare.

5. Scrieți o funcție care are 2 parametri referință și calculează produsul acestora.

6. Scrieți un program C++ care apelează o funcție recursivă. Funcția calculează factorialul unui număr natural.

7. Să se genereze toate numerele prime de n cifre formate numai cu cifrele: 2, 0, 9.

7. Tablouri

7.1 Tablouri unidimensionale

Tablourile unidimensionale fac parte din Standard Template Library (STL) și oferă funcționalități similare cu array-urile. Elementele vectorului sunt de același tip de dată.

Un tablou unidimensional (vector) reprezintă o structură de date de același tip reunite sub un nume comun. Valorile din tablou vor fi identificate prin poziția lor în cadrul tabloului.

O variabilă de tip tablou pentru a fi declarată trebuie specificat numărul maxim de componente pe care le poate avea tabloul (între paranteze drepte, precedat de tipul de dată al elementelor).

```
tip_nume [dimensiune_maximala]
```

Exemplu:

```
int V[100];
```

Fiecare element din tabloul unidimensional poate fi identificat printr-un **index**, care este o valoare numerică ce indică poziția elementului în cadrul secvenței.

Pentru a accesa o componentă dintr-un tablou unidimensional, se folosește paranteze drepte *[ ]* în care se specifică indicele componentei care se dorește a fi accesată.

Exemplu:

```
V[1] = 46; //prima casuta din tablou
```

V[5] = 20;

Mai jos este prezentat modul de lucru cu vectori. Prima dată se declară vectorul și tipul elementelor din vector: std**::**vector**<**int**>** numbers**;**

Se pot adăuga elemente în vector cu ajutorul funcției push_back, iar dimensiunea vectorului poate fi aflată folosindu-se funcția size.

Elementele vectorului pot fi accesate folosindu-se sintaxa numbers**[i],** unde **numbers** este numele vectorului, iar **i** reprezintă poziția elementului. De asemenea pentru accesarea unui element al vectorului se mai pot folosi iteratori precum: **begin()** și **end()**.

Pentru a se parcurge toate elementele unui vector, se pot folosi instrucțiuni precum *for* și *while*.

Se pot șterge elemente din vector cu funcția **pop_back()**. Această funcție șterge ultimul element din vector.

Exemplu de lucru cu vectorii

```
#include <iostream>
#include <vector>
int main() {
    // Declararea unui vector de numere întregi
    std::vector<int> numbers;
    // Adăugarea elementelor în vector
    numbers.push_back(10);
    numbers.push_back(20);
```

```
numbers.push_back(30);

// Accesarea elementelor folosind sintaxa de tip vector[index]

std::cout << "Primul element: " << numbers[0] << std::endl;  // Output: 10

// Parcurgerea vectorului cu instuctiunea for

std::cout << "Toate elementele:";

for (int i = 0; i < numbers.size(); ++i) {

    std::cout << " " << numbers[i];

}

std::cout << std::endl;  // Output: 10 20 30

// Utilizarea iteratorilor pentru a accesa elementele

std::cout << "Toate elementele (folosind iteratori):";

for (auto it = numbers.begin(); it != numbers.end(); ++it) {

    std::cout << " " << *it;

}

std::cout << std::endl;  // Output: 10 20 30

// Eliminarea ultimului element din vector

numbers.pop_back();

std::cout << "După eliminarea ultimului element:";

for (auto num : numbers) {

    std::cout << " " << num;
```

```
    }
    std::cout << std::endl;  // Output: 10 20
    return 0;
}
```

Citirea unui vector de la tastatură este prezentată mai jos:

```
int n, v[100];
cin>> n;
for (int i=1; i<=n; i++)
   cin>>v[i];
```

Citirea unui vector dintr-un fișier se poate face în 2 moduri diferite:

a. când se cunoaște numărul de elemente

```
f >> n;
for (int i=1; i<=n; i++)
   f>>v[i];
```

b. când nu se cunoaște numărul de elemente

```
n=1;
while (f>>v[n])
   n++;
n--;
```

Într-un vector se poate determina, elementul maxim al vectorului astfel:

```
max = v[1];
for(int i=2; i<=n; i++)
   if(v[i] > max) max=v[i];
```

Determinarea numărului de apariții a unei valori dintr-un vector: variabila *x* este valoarea pe care o căutăm.

```
nr = 0;
for (int i=1; i<= n; i++)
   if (v[i] == x) nr++; // aceasta conditie se schimba in functie de cerinta
```

Elementele vectorului pot fi sortate crescător sau descrescător. Mai jos este prezentată așezarea elementelor în ordine crescătoare.

```
int aux, v[100];
for (int i = 1; i < n; i++)
   for (int j = i+1; j++)
      if (v[i] > v[j] {
         aux = v[i];
         v[i] = v[j];
         v[j] = aux;
      }
```

7.2 Tablouri multidimensionale

Tabloul multidimensional este un array format din mai multe array-uri. Această structură permite organizarea elementelor în matrici.

Matricile sunt caracterizate de numărul de linii și coloane. Fiecare linie conține un număr fix de coloane.

Pentru a se defini o matrice, trebuie declarate 2 variabile :

- rows - reprezentând numărul de linii.
- cols - reprezentând numărul de coloane.

Apoi se definește matricea: array**[**rows**][**cols**].** Matricele pot fi statice și dinamice.

a. matrice statică - se specifică elementele matricei la declararea matricei.

Exemplu de lucru cu matrice statică:

```
#include <iostream>
using namespace std;
int main() {
    // Definirea matricei cu 3 linii si 4 coloane
    const int rows = 3;
    const int cols = 4;
    int array[rows][cols] = {
        {1, 2, 3, 4},
```

```
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    // Asignarea elementelor
    cout << "Element at (1, 2): " << array[1][2] << endl;  // output: 7
    // parcurgere array
    cout << "Array elements:" << endl;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cout << array[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Accesarea unui element din matrice se face prin: array[i][j] sau array[1][2]

b. matrice dinamică - se folosesc pointeri (se folosește *) sau clasa std::vector pentru o mai mare flexibilitate.

Exemplu de matrice dinamică care folosește pointeri:

```
#include <iostream>
using namespace std;
int main() {
    // definirea dimensiunilor matricei
    int rows = 3;
    int cols = 4;
    // alocarea memoriei pentru o matrice dinamica
    int** array = new int*[rows];
    for (int i = 0; i < rows; ++i) {
        array[i] = new int[cols];
    }
    // initializarea elementelor cu valori
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            array[i][j] = (i + 1) * (j + 1);
        }
    }
    // accesarea elementelor
    cout << "Element at (1, 2): " << array[1][2] << endl;  // Output: 6
```

```
    // parcurgere array
    cout << "Array elements:" << endl;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cout << array[i][j] << " ";
        }
        cout << endl;
    }
    // dealocarea memoriei
    for (int i = 0; i < rows; ++i) {
        delete[] array[i];
    }
    delete[] array;
    return 0;
}
```

Mai jos este un exemplu de matrice dinamică care folosește clasa std::vector.

Exemplu de matrice dinamică care folosește clasa std::vector

```
#include <iostream>
```

```
#include <vector>

int main() {

    // definirea dimensiunii matricei

    int rows = 3;

    int cols = 4;

    // creare vector

    std::vector<std::vector<int>> array(rows, std::vector<int>(cols));

    // initializarea elementelor vectorului

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            array[i][j] = (i + 1) * (j + 1);

        }

    }

    // accesarea elementelor vectorului

    std::cout << "Element at (1, 2): " << array[1][2] << std::endl;   //
Output: 6

    // parcurgere array

    std::cout << "Array elements:" << std::endl;

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {
```

```
            std::cout << array[i][j] << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Matrice pătratică

Sunt matrice în care numărul de linii este identic cu numărul de coloane. Sunt singurele matrice care au prima diagonală (diagonală principală) respectiv a doua diagonală (diagonală secundară).

Orice element situat pe prima diagonală are cei 2 indicii egali. Presupunând că indicii sunt i și j, avem relația i == j.

Presupunând că indicii sunt i și j, orice element situat pe a doua diagonală satisface relația i + j == n+1 dacă indicii pornesc de la 1, respectiv i + j == n-1 dacă indicii pornesc de la 0.

O matrice este simetrică după prima diagonală dacă a[i][j] = a[j][i] pentru oricare i, j din matrice.

O matrice este simetrică după a doua diagonală dacă a[i][j] = a[n+1-i][n+1-j] pentru oricare i, j din matrice.

7.3 Exerciții

1. Definiți și afișați elementele unui vector folosind limbajul de programare C++.

2. Să se afișeze dimensiunea vectorului de la exercițiul anterior.

3. Adăugați 2 elemente vectorului de la exercițiul 1 și apoi ștergeți ultimul element introdus. După fiecare operație de introducere/ștergere se va afișa elementele vectorului.

4. Să se definească un vector cu 10 numere reale. Programul afișează suma elementelor impare din vector.

5. Se citește de la tastatură un șir de n valori. Să se afișeze valoarea maximă din șir.

6. Să se construiască vectorul V cu 2*n elemente întregi din intervalul (1; 2*n) astfel încât șirul elementelor impare să fie strict crescător și șirul elementelor pare să fie descrescător.

7. Se citește un vector de la tastatură cu n elemente. Să se adauge valoarea 0 înaintea primului element impar.

8. Se citește de la tastatură un șir de n valori. Să se afișeze ultima valoarea din șir care are suma cifrelor minimă.

9. Să se definească și să se afișeze elementele unei matrici statice care este formată din 3 linii și 4 coloane. Elementele matricei de pe prima linie sunt numere pare, elementele de pe a doua linie sunt numere impare, iar elementele de pe linia a3a sunt suma elementelor de pe primele 2 linii.

   Exemplu:

   Linia 1: 2 4 8 12

   Linia 2: 3 7 11 9

   Linia 3: 5 11 19 21

10. Se citește de la tastatură un șir de n valori. Să se afișeze suma tuturor valorilor din șir care folosesc o singură cifră în scrierea lor.

    Exemplu: n = 3

    x1 = 11

    x2 = 222

    ~~x3 = 135~~

11. Să se definească o matrice dinamică cu pointeri și să se calculeze și afișeze suma elementelor de pe diagonala principală.

12. Să se definească o matrice dinamică folosindu-se std::vector și să se calculeze și afișe pe ecran produsul numerelor pare de pe diagonala secundară.

13. Se citește de la tastatură o matrice pătratică de ordin n. Să se calculeze suma elementelor pare aflate deasupra diagonalei principale și produsul elementelor impare aflate sub diagonala secundară.

14. Să se afișeze elementele aflate deasupra diagonalei secundare inclusiv diagonala secundară care au suma cifrelor un număr par.

8. Pointeri

Un **pointer** este o variabilă specială care **stochează adresa de memorie** a altei variabile.

Exemplu:

```
int x = 10;       // o variabilă normală
int* p = &x;      // un pointer care stochează adresa lui x
```

unde:

- x este o variabilă de tip int cu valoarea 10.
- &x înseamnă „adresa lui x”.
- p este un pointer la un int, adică stochează adresa unde se află x.

Notă:

- * folosit la declarație: definește un pointer.
- **&** (operator „adresă”): obține adresa unei variabile.
- * (operator „dereferențiere”): accesează valoarea de la adresa stocată în pointer.

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

int main() {
    int a = 5;
    int* ptr = &a;          // pointer la a

    cout << "Valoarea lui a: " << a << endl;
    cout << "Adresa lui a: " << &a << endl;
    cout << "Valoarea stocată în ptr (adresa lui a): " << ptr << endl;
    cout << "Valoarea spre care pointează ptr: " << *ptr << endl;

    return 0;
}
```

Figura 7.3-1 Exemplu de program cu pointeri

Programul de mai sus are ieșirea:

```
Valoarea lui a: 5
Adresa lui a: 0x... (variază)
Valoarea stocată în ptr (adresa lui a): 0x...
Valoarea spre care pointează ptr: 5
```

Figura 7.3-2 Ieșirea programului C++ anterior

De ce sunt utili pointerii?

- Acces direct la memorie
- Eficiență (în special în lucrul cu mari volume de date)
- Transmiterea eficientă a variabilelor în funcții (prin referință)
- Lucru cu tablouri, structuri dinamice, alocare dinamică (new / delete)
- Implementarea listelor, arborilor, grafurilor etc.

Alocarea dinamică permite rezervarea memorie în timpul rulării programului folosind *new* și eliberarea memoriei cu *delete*.

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

int main() {
    // Alocare dinamică pentru un singur int
    int* p = new int;   // rezervă memorie pentru un int
    *p = 42;            // atribuie valoare

    cout << "Valoarea stocată în p: " << *p << endl;

    delete p;           // eliberăm memoria

    // Alocare dinamică pentru un tablou de 5 int-uri
    int* arr = new int[5];  // rezervă spațiu pentru 5 int-uri

    for (int i = 0; i < 5; i++) {
        arr[i] = i * 10;
    }

    cout << "Valori în tablou:\n";
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    delete[] arr;       // eliberăm memoria alocată tabloului
```

Figura 7.3-3 Alocare dinamică de memorie

8.1 Pointeri dubli

În C și C++, pointerii dubli (sau pointeri la pointeri) sunt variabile care conțin adresa unui pointer, adică „un pointer la un pointer”.

Declarație: int **pp;

Explicație: pp este un pointer către un pointer la un int

Exemplu:

int x = 10; – o variabilă obișnuită de tip int.

int *p = &x; – p este un pointer către x.

int **pp = &p; – pp este un pointer către p.

Când se folosesc pointerii dubli?

1. Transmisie de pointeri într-o funcție – pentru a modifica adresa unui pointer într-o funcție.

Exemplu:

```
void alocare(int **ptr) {
   *ptr = malloc(sizeof(int));
   **ptr = 42;
}
```

2. Tablouri bidimensionale dinamice – int **matrice este adesea folosit pentru a reprezenta o matrice 2D alocată dinamic.
3. Argumentele argv în main:

Exemplu:

```
int main(int argc, char **argv) {
   // argv este un pointer la un vector de stringuri (pointeri la char)
}
```

8.2 Pointeri la const vs constanta la pointer

În C++, există o diferență importantă între:

a. pointer la constantă (const int *p) - se poate schimba adresa stocată în pointer, dar **nu** se poate modifica valoarea la care pointează.

Exemplu:

```
const int x = 5;

const int y = 10;

const int *p = &x;  // ok

*p = 20;            //  eroare: valoarea e const

 p = &y;            // se schimbă adresa
```

Explicație:

- *p este const int ⇒ valoarea nu poate fi modificată
- p nu este const ⇒ adresa se poate schimba

b. pointer constant (int *const p) – se poate modifica valoarea la care pointează, dar **nu** se poate schimba adresa (adică pointerul e constant).

Exemplu:

```
int x = 5;

int y = 10;

int *const p = &x;  // ok

*p = 20;            // se modifică valoarea
```

```
p = &y;          // eroare: pointerul e const
```

Explicație:

- p este const ⇒ nu poate fi reasignat
- *p este int ⇒ poate fi modificat

c. pointer constant la constantă (const int *const p) – nu se poate modifica nici valoarea și nici adresa.

Exemplu:

```
const int x = 5;

const int *const p = &x;

*p = 10;       // eroare: valoarea e const

p = nullptr;   // eroare: pointerul e const
```

8.3 Exerciții

1. Scrieți un program C++ care:
   a. Declară un vector de n elemente întregi.
   b. Calculează media aritmetică a elementelor folosind pointeri.
   c. Afișează rezultatul pe ecran.

2. Scrieți o funcție care:

   a. Primește ca parametru un vector de n elemente întregi și îl inversează în loc (in-place).

b. Programul principal va citi vectorul, apela funcția, și afișa vectorul rezultat.

Restricții: Folosiți pointeri (int *) pentru parcurgere și schimb.

3. Scrieți o funcție care:
   a. Alocă dinamic o matrice cu m linii și n coloane.
   b. Permite utilizatorului să introducă valorile matricei.
   c. Afișează matricea pe ecran.

   Indicații:
   - Matricea se va reprezenta ca int **mat.
   - Alocarea se face cu new, iar la final trebuie eliberată memoria cu delete.

9. Structuri

Structurile (*en. struct*) ne permit să definim noi tipuri de date ce grupează la un loc date de același tip sau de tipuri diferite. Ulterior sau în momentul definirii se pot declara variabile de tipul înregistrării.

Pentru a defini o înregistrare și a declara variabile de tipul său, folosim una dintre următoarele 3 modalități:

a.
```
struct {
    campuri membre; (proprietati)
}
variabile;
```

Exemplu - pentru a defini o structură în care ținem minte numele și prenumele unui elev de câte 30 caractere precum și nota1, nota2 de tip float și apoi să declarăm o variabilă **x** de acest tip scriem:

```
struct {
   char nume[31], prenume[31];
   float nota1, nota2;
} x;
```

b.
```
struct nume {
    campuri membre(proprietati)
} nume_variabile
```

Pentru exemplul anterior avem:

```
struct elev{
   char nume[31], prenume[31];
   float nota1, nota2;
} elev x;
```

c.
```
struct nume {
   campuri membre(proprietati)
} variabile
```

Opțional putem declara ulterior alte variabile scriind numele variabilelor.

Pentru exemplul anterior scriem:

```
struct elev{
   char nume[31], prenume[31];
   float nota1, nota2;
} x;
```

Indiferent de modalitatea de definire și declarare în restul programului, variabilele de tip *struct* se comportă în același fel.

Informațiile referitoare la o variabilă de tip struct se numește câmp. Accesarea câmpului unui *struct* se face întotdeauna scriind variabila.camp.

Exemplu:

```
cin>>x.nota1;

x.nota2 = 7;
```

Pot exista câmpuri ale unui struct care sunt la rândul lor de tipul unui struct, ca în exemplul:

```
struct punct {float x, y;};

struct triunghi {punct a, b, c;};


triunghi t;
```

Pentru a citi coordonatele celor 3 vârfuri ale triunghiului utilizăm:

```
cin >>t.a.x >>t.a.y;

cin >> t.b.x >> t.b.y;

cin >> t.c.x >> t.c.y;
```

Citiri, afișări, comparații se pot face doar după tipuri simple de date.

Dacă un câmp este de tip șir de caractere atunci pentru prelucrările ce îl implică va trebui să folosim funcții ce prelucrează șiruri de caractere.

Atribuirile se pot face fie la nivel de câmp, fie la nivel de variabilă ce în exemplele:

```
elev d, e;
d = e => {
   d.nota1 = e.nota1;
   d.nota2 = e.nota2;
   strcpy (d.nume, e.nume);
   strcpy (d.prenume, e.prenume);
}
```

9.1 Exerciții

1. Se citesc informații despre n elevi ai unei clase: numele elevu;ui, prenumele elevului și nota luată la bacalaureat. Să se afișeze numele și prenumele elevului/elevilor care au luat cea mai mică notă.

2. Într-un depozit sunt n tipuri de medicamente. Pentru fiecare tip de medicament se cunoaște: codul medicamentului, denumirea (șir de maxim 30 caractere), prețul (float/double). Se citește de la tastatură denumirea unui medicament d. Să se verifice dacă medicamentul se găsește sau nu în depozit. În caz afirmativ să se modifice prețul medicamentelor astfel încât să crească cu p% în afară de medicamentul d.

Unde p = 5% pentru medicamente cu codul mai mic decât 10000 și p = 10% pentru restul codurilor.

3. Se citesc informații despre n numere complexe. Să se afișeze în ordinea descrescătoare a modulelor numerele complexe din vectori.

10. Programarea orientată pe obiecte

OOP sau Object Oriented Programming este o paradigmă des întâlnită în programare. Limbajele de programare care suportă OOP (Java, C++, C#, Python, PHP, Javascript, etc.) au la bază câteva concepte, enumerate în figura de mai jos.

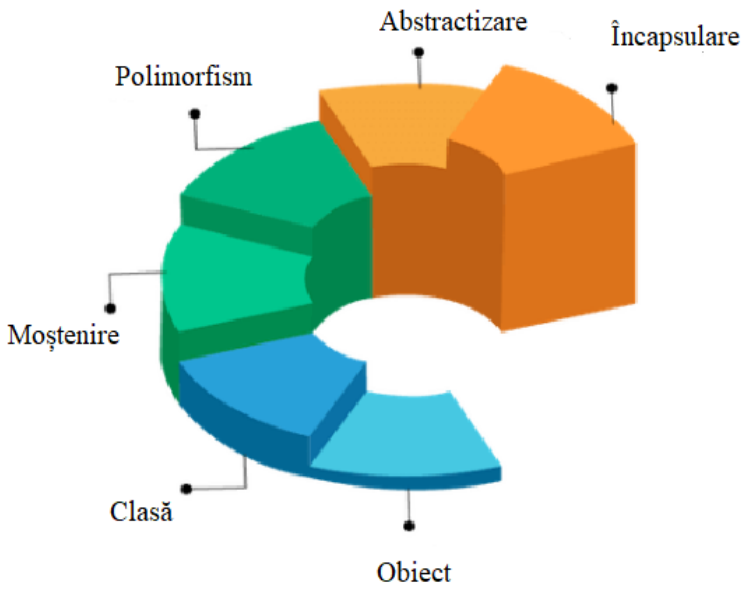

Figura 9.1-1 Conceptele de bază ale programării orientate pe obiecte

Cele 4 mari principii OOP se descriu prin:

- ➔ Abstractizare
- ➔ Încapsulare
- ➔ Moștenire
- ➔ Polimorfism

10.1 Clase și obiecte

Clasa este o colecție de obiecte care au proprietăți, operații și comportamente comune. O combinație de stări (date) și de comportamente (metode).

În OOP, o clasa este un ***tip de date***, iar obiectele sunt ***instanțe*** ale acestui tip de date.

Structura OOP este alcătuită din obiecte, clase, atribute și metode.

- Obiecte - instanțe ale unor clase care reprezintă entități reale sau abstracte.
- Clase - șabloane pentru crearea obiectelor care definesc atributele și metodele acestora.
- Atribute - date despre obiect care stochează starea acestuia.
- Metode - funcții care pot schimba starea obiectului sau pot efectua anumite acțiuni.

Exemplu: pentru a înțelege structura programării orientate pe obiecte:

Clasa: Tester

Obiectul: tester Cristian

Atribute: salariu și responsabilități

Metode: testare software

O clasă definește un tip abstract de date.

Definiție clasă:

```
class nume{
```

lista_elementelor_membre

};

Lista elementelor membre poate conține:

- declarații de date;
- implementări de funcții;
- prototipuri de funcții abstracte.

Datele declarate printr-o definiție de clasă se numesc date membre.

Funcțiile definite sau pentru care este prezent numai prototipul în definiția clasei, se numesc funcții membre sau metode.

Atât datele cât şi metodele pot avea modificatori de acces.

Modificatorii de acces sunt cuvinte rezervate ce controlează accesul celorlalte clase la membrii unei clase. Specificatorii de acces pentru variabilele şi metodele unei clase sunt: public, protected, private şi cel implicit (la nivel de pachet).

Vizibilitatea datelor şi funcțiilor membre:

- public - accesate de funcțiile membre ale clasei şi toate funcțiile nemembre ale programului
- private - accesate doar de funcțiile membre sau prietene clasei
- protected - asemănător cu privat, dar permite accesul claselor derivate la datele şi funcțiile membre
- default - toate datele definite într-o clasă sunt private dacă nu este specificat un specificator de acces

Observații:

- ➔ domeniul de utilizare al unui specificator de acces ține până la întâlnirea altui specificator de acces
- ➔ specificatorul de acces implicit este *private*
- ➔ funcțiile membre clasei ar trebui să fie declarate *publice*
- ➔ datele membre ale unei clase ar trebuie să fie declarate ca fiind private pentru a respecta principiul încapsulării
- ➔ este util să existe funcții *set* şi *get* pentru a accesa datele membre într-un mod controlat

public - full access, no encapsulation

private - least access, best encapsulation

protected - some access, moderate encapsulation

10.2 Abstractizarea datelor

Abstractizarea datelor în C++ este un concept fundamental al programării orientate pe obiect (OOP) și se referă la procesul de ascundere a detaliilor interne de implementare ale unei clase și la expunerea doar a caracteristicilor esențiale relevante pentru utilizator.

În C++, abstractizarea este realizată prin intermediul claselor și accesului controlat la membri (prin specificatori precum: private, protected și public). Ideea este de a separa ce face un obiect (interfața) de cum face (implementarea sa).

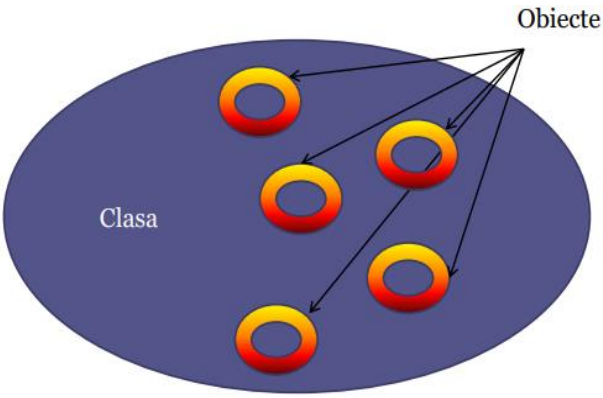

Figura 10.2-1 Abstractizarea datelor

Exemplu:

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

class ContBancar {
private:
    double sold;

public:
    ContBancar() {
        sold = 0;
    }

    void depune(double suma) {
        if (suma > 0) {
            sold += suma;
        }
    }
```

```
    void afiseazaSold() {
        cout << "Soldul curent este: " << sold << " RON" << endl;
    }
};

int main() {
    ContBancar cont;
    cont.depune(500);
    cont.afiseazaSold();

    return 0;
}
```

În exemplul de mai sus utilizatorul nu știe cum e gestionat soldul intern și poate doar să folosească metodele depune() și afiseazaSold().

10.3 Încapsularea

Încapsularea înseamnă, pe lângă regruparea unor elemente (de date – atributele și de comportament - operațiile) și ascunderea și protecția detaliilor.

Avantajele încapsulării:

- datele încapsulate în obiecte sunt protejate de accesul nedorit, fiindu-le garantată integritatea
- utilizatorii unei abstracții nu depind de implementarea acestei abstracții ci de specificația ei, ceea ce reduce cuplajul între modele.

Exemplul numerelor complexe ilustrează și ascunderea detaliilor.

Implicit:

- valorile atributelor unui obiect sunt ascunse în obiecte şi nu pot fi manevrate direct de către alte obiecte (atributele sunt implicit private)
- operațiile sunt publice, toate interacțiunile între obiecte sunt efectuate declanşând diversele operații declarate în specificația clasei şi accesibile altor obiecte.

Exemplu:

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

class Persoana {
private:
    string nume;
    int varsta;

public:
    // Setter pentru nume
    void seteazaNume(string n) {
        nume = n;
    }
    // Getter pentru nume
    string obtineNume() {
        return nume;
    }
    // Setter pentru varsta
    void seteazaVarsta(int v) {
```

```
        if (v >= 0) {
            varsta = v;
        }
    }
    // Getter pentru varsta
    int obtineVarsta() {
        return varsta;
    }
};

int main() {
    Persoana p;
    p.seteazaNume("Maria");
    p.seteazaVarsta(30);
    cout << "Nume: " << p.obtineNume() << endl;
    cout << "Varsta: " << p.obtineVarsta() << endl;
    return 0;
}
```

- Variabilele nume și varsta sunt private → nu pot fi accesate direct din main().

- Accesul se face doar prin metodele publice seteazaNume(), seteazaVarsta(), obtineNume() și obțineVarsta() → control total asupra datelor.

10.4 Constructori

Constructorii sunt funcții speciale în C++ care sunt folosite pentru a inițializa obiectele unei clase atunci când sunt create. Un constructor este invocat automat la crearea unui obiect dintr-o clasă și are rolul de a asigura că obiectul este într-o stare validă.

Constructor: funcție utilizată pentru a inițializa instanțele (obiectele) unei clase.

Caracteristici:

- constructorul are întotdeauna același nume ca și clasa
- nu are tip de return (nici măcar void)
- nu poate fi funcție virtuală
- este invocat automat atunci când un obiect este creat, fără a fi necesar ca programatorul să-l apeleze explicit.
- constructorul este folosit pentru a inițializa variabilele membre ale clasei sau pentru a efectua alte operațiuni necesare înainte ca obiectul să fie utilizat.

Tipuri de constructori:

- constructor implicit (default): este un constructor care nu are parametri. Constructorul implicit este invocat atunci când un obiect este creat fără argumente.

```cpp
class Persoana {
public:
    string nume;
    int varsta;

    // Constructor implicit
    Persoana() {
        nume = "Necunoscut";
        varsta = 0;
    }
};

int main() {
    Persoana p;  // Constructorul implicit este apelat
    cout << p.nume << " " << p.varsta << endl;  // Va afișa: Necunoscut 0
}
```

Figura 10.4-1 Constructor implicit

- constructor cu parametrii: este un constructor care are unu sau mai mulți parametrii și poate fi folosit pentru a inițializa obiectele cu valori specifice. Constructorul cu parametrii permite crearea obiectelor cu stări de început personalizate.

```cpp
class Persoana {
public:
    string nume;
    int varsta;

    // Constructor cu parametri
    Persoana(string n, int v) {
        nume = n;
        varsta = v;
    }
};

int main() {
    Persoana p("Ion", 30);  // Constructorul cu parametri este apelat
    cout << p.nume << " " << p.varsta << endl;  // Va afișa: Ion 30
}
```

Figura 10.4-2 Constructor cu parametri

În C++, o clasă poate avea mai mulți constructori, fiecare cu numărul și tipul parametrilor diferiți. Acest lucru permite crearea de obiecte într-o varietate de moduri, în funcție de necesitățile programului.

Exercițiu:

Realizați o clasa ContBancar care reprezintă un cont bancar pentru un client. Contul bancar va avea un **sold** (suma de bani din cont) și un **titular** (numele titularului contului). Cerințe:

- Clasa ContBancar trebuie să aibă un constructor implicit și un constructor cu parametri.

- Implementarea unei metode afisare() care afișează detaliile contului bancar.
- În funcția main(), se vor crea conturi bancare folosind constructorii definiți și se vor afișa informațiile acestora.

10.5 Compunerea claselor

Compunerea claselor este un concept fundamental în programarea orientată pe obiecte și se bazează pe conceptul **"are-un"** între clase. În contextul C++, compunerea claselor înseamnă că o clasă conține obiecte din alte clase ca date membre. Acest lucru permite crearea de clase complexe și reutilizarea codului, favorizând modularitatea și organizarea mai bună a aplicațiilor.

Compunerea presupune că o clasă (clasa "superioară") include obiecte ale altor clase (clase "inferioare") ca membri. Astfel, o instanță a unei clase poate conține instanțe ale altor clase.

Avantajele compunerii:

- Permite împărțirea unui program în componente mai mici și reutilizabile.
- Ascunde complexitatea implementării și oferă o interfață simplificată pentru utilizator
- Permite folosirea unor clase externe fără a fi necesar să se extindă funcționalitatea lor.

Exemplu de compunere a claselor:

Presupunem că avem 2 clase: Locuinta și Camera. Clasa Locuinta conține un obiect de tip Camera.

```
#include <iostream>
#include <string>
using namespace std;
class Camera {
public:
    string nume_camera;
    int suprafata;  // Suprafața camerei în metri pătrați
    Camera(string nume, int sup) : nume_camera(nume), suprafata(sup) {}
    void descriere() {
        cout << "Camera " << nume_camera << " are " << suprafata << " mp."
<< endl;
    }
};
class Locuinta {
public:
    string adresa;
    Camera camera1;  // Obiect de tip Camera

    // Constructor pentru Locuinta
```

```
    Locuinta(string adr, string nume_camera, int suprafata) : adresa(adr),
camera1(nume_camera, suprafata) {}

    void descriere() {

        cout << "Locuinta de la adresa: " << adresa << endl;

        camera1.descriere();  // Apelăm metoda descriere() din clasa Camera

    }

};

int main() {

    // Creăm un obiect de tip Locuinta, care conține un obiect de tip Camera

    Locuinta loc1("Strada Exemplu, nr. 1", "Dormitor", 20);

    loc1.descriere();

    return 0;

}
```

- **Clasa Camera:** conține informații despre o cameră, cum ar fi numele camerei și suprafața ei.
- **Clasa Locuinta**: conține un obiect de tip Camera și o adresă. Constructorul clasei Locuinta inițializează atât adresa, cât și camera.

Compunere vs. Moștenire

- Compunerea implică includerea unui obiect dintr-o altă clasă în interiorul unei clase, în timp ce moștenirea presupune ca o clasă să extindă o altă clasă, moștenind membrii și comportamentele acesteia.
- Exemplu de diferență:

- compunere: o clasă "are un" obiect din altă clasă.
- moștenire: o clasă "este un" tip de altă clasă.

Exercițiu:

Creați un sistem de gestionare a unei Biblioteci. În cadrul acestui sistem, o bibliotecă poate conține mai multe Cărți. Fiecare carte are un Titlu, un Autor și un An de publicare. De asemenea, fiecare bibliotecă are un Nume și un Număr de cărți. Cerințe:

- Crearea unei clase Carte care să conțină atributele: titlu, autor și an_publicare.
- Clasa Carte trebuie să aibă un constructor pentru inițializarea atributelor și o funcție descriere() care să afișeze detaliile cărții.
- Crearea unei clase Biblioteca care să conțină un atribut nume și un tablou de obiecte Carte.
- Implementarea unor metode pentru a adăuga cărți într-o bibliotecă și pentru a afișa informațiile despre bibliotecă și cărțile sale.
- În funcția main(), să fie creată o bibliotecă cu mai multe cărți, iar apoi să fie afișate informațiile despre bibliotecă și despre cărțile sale.

10.6 Supraîncărcarea operatorilor

Supraîncărcarea operatorilor în C++ este un concept care permite utilizatorilor să redefinească sau să „supraîncarce” comportamentul operatorilor predefiniți (precum +, -, *, =, etc.) pentru tipuri de date definite de utilizator (adică, pentru clasele proprii). Prin supraîncărcarea operatorilor, se

poate face ca operatorii să funcționeze în moduri personalizate pe obiecte ale claselor respective, ceea ce face ca sintaxa și utilizarea să fie mai naturale și mai intuitive.

Supraîncărcarea operatorilor permite utilizarea operatorilor standard într-un mod care este specific clasei sau tipului de obiect, îmbunătățind astfel claritatea și concizia codului. De exemplu, dacă ai o clasă *Complex* pentru numere complexe, se poate supraîncărca operatorul + astfel încât să adune două numere complexe.

Supraîncărcarea operatorilor în C++ se face prin definirea unei funcții membre sau a unei funcții prietene (friend function) care implementează comportamentul dorit pentru operatorul respectiv.

Exemplu 1: Supraîncărcarea operatorului + pentru adunarea a două obiecte ale unei clase.

- Să presupunem că vrem să implementăm o clasă *Complex* pentru numere complexe și să supraîncarcăm operatorul + pentru a aduna două numere complexe.

```
#include <iostream>

using namespace std;


class Complex {

private:

    double real, imag;
```

```
public:
    // Constructor
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    // Supraincarcarea operatorului +
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    // Metoda pentru afisarea numarului complex
    void afiseaza() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex num1(3.0, 4.0);  // Creăm primul număr complex
    Complex num2(1.5, 2.5);  // Creăm al doilea număr complex

    Complex rezultat = num1 + num2;  // Folosim operatorul + supraincarcat
```

```
    rezultat.afiseaza();  // Afisează: 4.5 + 6.5i


    return 0;
}
```

1. Clasa Complex:
   - conține 2 atribute private: real și imag, care reprezintă partea reală și partea imaginară a unui număr complex.
   - constructorul inițializează aceste atribute cu valori implicite sau cu valorile date la crearea obiectului.
   - supraîncărcarea operatorului + adună două obiecte de tip Complex. În funcția **operator+**, se creează un nou obiect de tip Complex care are suma părților reale și imaginare ale celor două obiecte implicate în operație.
2. Funcția main():
   - se crează 2 obiecte de tip Complex: num1 și num2.
   - se aplică operatorul + pe acestea, iar rezultatul este stocat în variabila rezultat.
   - se afișează rezultatul adunării folosindu-se metoda *afiseaza()*.

Exemplu 2: Supraîncărcarea operatorului << pentru afișarea unui obiect.

Presupunem că vrem să supraîncarcăm operatorul << pentru a putea afișa obiectele de tip Complex folosind *cout*:

```
using namespace std;
```

```
class Complex {
private:
    double real, imag;

public:
    // Constructor
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    // Supraincarcarea operatorului <<
    friend ostream& operator<<(ostream& out, const Complex& c) {
        out << c.real << " + " << c.imag << "i";
        return out;
    }
};

int main() {
    Complex num1(3.0, 4.0);
    cout << "Numarul complex este: " << num1 << endl;  // Afiseaza:
Numarul complex este: 3 + 4i
```

```
    return 0;
}
```

1. Supraîncărcarea operatorului <<:
   - se declară operatorul **<<** ca o funcție prietenă a clasei Complex. Aceasta permite accesul la datele private ale clasei Complex și direcționează fluxul de ieșire *out* pentru a afișa numărul complex într-un format adecvat (real + imag i).
2. În funcția main():
   - se crează un obiect Complex și se folosește *cout* pentru a-l afișa direct, utilizând operatorul <<.

Exerciții:

Pe baza exemplelor de mai sus, supraîncărcați operatorii:

1. scădere (numere complexe)
2. înmulțire (numere complexe)
3. împărțire (numere complexe)

10.7 Moștenirea claselor

Moștenirea claselor este un concept fundamental în programarea orientată pe obiecte care permite unei clase (numită subclasă sau clasă derivată) să preia caracteristicile (atributele și metodele) unei alte clase (numită superclasă sau clasă de bază).

Scopul moștenirii:

- Reutilizarea codului – evită duplicarea codului și permite folosirea unor funcționalități existente.

- Extensibilitate – permite adăugarea de noi funcționalități fără a modifica clasa originală.
- Polimorfism – permite utilizarea metodelor suprascrise pentru a crea un comportament specific în subclase.

În moștenirea claselor, **protected** este un modificator de acces care controlează vizibilitatea membrilor clasei în subclasele derivate.

Tabel 4 Private vs Protected vs Public

Modificator	**Accesibil în clasa de bază**	**Accesibil în clase derivate**	**Accesibil în exterior**
private	Da	Nu	Nu
protected	Da	Da	Nu
public	Da	Da	Da

Exemplu:

```
#include <iostream>

#include <iostream>

using namespace std;

// Clasa de bază (superclasa)

class Animal {

protected:
```

```
    string nume;
public:
    Animal(string n) : nume(n) {}
    void afiseazaNume() {
        cout << "Nume: " << nume << endl;
    }
};
// Clasa derivată (subclasa) care moștenește din Animal
class Caine : public Animal {
public:
    Caine(string n) : Animal(n) {}
};
// O altă subclasă
class Pisica : public Animal {
public:
    Pisica(string n) : Animal(n) {}
};
int main() {
    Caine caine("Rex");
    Pisica pisica("Tom");
```

```
    caine.afiseazaNume();

    pisica.afiseazaNume();

    return 0;

}
```

Explicație:

- Clasa Animal – are un atribut nume și o metodă afiseazaNume().
- Clasa Caine și Pisica – moștenesc clasa Animal.
- Testarea în main() – creăm obiecte de tip Caine și Pisica, apelăm metodele lor.

Exercițiu:

Scrieți un program C++ care utilizează moștenirea pentru a modela un sistem de vehicule:

a. să se creeze o clasă de bază numită Vehicul care conține:
   - o dată membră numită vitezaMaxima (de tip int).
   - o metodă numită afiseazaInformatiiVehicul() care afișează viteza maximă.

b. să se creeze două clase derivate:
   - Masina: adaugă o dată membră numită numarUsi și se definește metoda afiseazaInformatiiMasina().
   - Motocicleta:adaugă o dată membră numită tipMotocicleta și se definește metoda afiseazaInformatiiMotocicleta().

c. În main(), să se creeze obiecte de tip Masina și Motocicleta, apoi să se afișeaze informațiile despre ele.

10.8 Funcții virtuale și polimorfism

O funcție virtuală în C++ este o metodă definită într-o clasă de bază care poate fi suprascrisă (override) în clasele derivate. Funcțiile virtuale permit polimorfismul dinamic, adică selecția metodei potrivite în timpul execuției, pe baza tipului real al obiectului, nu al tipului pointerului sau al referinței.

O funcție virtuală se declară folosindu-se cuvântul cheie *virtual* în clasa de bază, ca în exemplul de mai jos:

```
#include <iostream>
using namespace std;
class Baza {
public:
    virtual void afiseaza() {  // Funcție virtuală
        cout << "Afisare din clasa Baza\n";
    }
};
class Derivata : public Baza {
public:
    void afiseaza() override { // Suprascrierea metodei
        cout << "Afisare din clasa Derivata\n";
```

```
    }
};
int main() {
    Baza* ptr;
    Derivata obj;
    ptr = &obj;
    // Apel polimorfic: se va apela funcția din clasa Derivata
    ptr->afiseaza();
    return 0;
}
```

Explicație:

- Funcția afiseaza() este definită ca *virtual* în clasa Baza.
- În clasa Derivata, funcția afiseaza() este suprascrisă.
- Folosind un pointer de tip Baza*, dar care pointează către un obiect Derivata, C++ va apela funcția corespunzătoare clasei derivate.

Beneficiile funcțiilor virtuale:

- Permit scrierea de cod generic folosind pointeri și referințe la clasa de bază
- Evită problemele de selecție a metodei în cazul moștenirii
- Este esențială pentru implementarea interfețelor abstracte în C++

Funcții virtuale pure (clasă abstractă)

O funcție **virtuală pură** în C++ este o funcție virtuală care nu are o implementare în clasa de bază. O clasă care conține cel puțin o funcție virtuală pură devine o clasă abstractă, ceea ce înseamnă că nu poate fi instanțiată direct.

Definirea unei funcții virtuale pure este prezentată mai jos:

```
#include <iostream>

using namespace std;

class Forma {

public:

   virtual void deseneaza() = 0;  // Funcție virtuală pură (clasa devine abstractă)

};

class Cerc : public Forma {

public:

   void deseneaza() override {

      cout << "Desenez un cerc\n";

   }

};

class Patrat : public Forma {

public:

   void deseneaza() override {
```

```
        cout << "Desenez un pătrat\n";
    }
};
int main() {
    // Forma f;  // EROARE! Nu putem crea instanțe ale unei clase abstracte
    Forma* f1 = new Cerc();
    Forma* f2 = new Patrat();
    f1->deseneaza();  // Output: Desenez un cerc
    f2->deseneaza();  // Output: Desenez un pătrat
    delete f1;
    delete f2;
    return 0;
}
```

Explicație:

- Forma este o clasă abstractă deoarece conține o funcție virtuală pură: virtual void deseneaza() = 0;
- Clasele Cerc și Patrat moștenesc clasa Forma și suprascriu metoda deseneaza().
- Putem folosi pointeri de tip Forma* pentru a apela metodele deseneaza() din clasele derivate (polimorfism).
- Dacă o clasă derivată nu suprascrie funcția virtuală pură, aceasta devine și ea abstractă.

O clasă abstractă poate avea mai multe funcții virtuale pure.

```
class Animal {
public:
    virtual void sunet() = 0;
    virtual void mananca() = 0;
};
class Caine : public Animal {
public:
    void sunet() override {
        cout << "Ham ham!\n";
    }
    void mananca() override {
        cout << "Câinele mănâncă oase.\n";
    }
};
```

În exemplul de mai sus, orice clasă care moștenește clasa Animal trebuie să implementeze **toate** funcțiile virtuale pure, altfel rămâne **abstractă**.

Polimorfismul este un concept fundamental al programării orientate pe obiecte care permite aceluiași cod să se comporte diferit în funcție de tipul obiectului. Acesta face codul mai flexibil și reutilizabil.

În C++, există două tipuri principale de polimorfism:

1. Polimorfism la compilare (static)
    - Supraîncărcarea operatorilor (operator overloading)
    - Supraîncărcarea funcțiilor (function overloading)

2. Polimorfism la rulare (dinamic)
    - Funcții virtuale și moștenire
    - Clase abstracte și funcții virtuale pure

Exerciții

1. Folosindu-se funcțiile virtuale și polimorfismul, să se creeze o clasă de bază numită Animal și 2 clase derivate numite Pisica și Caine astfel:

- clasa Animal să conțină o funcție virtuală *sunet()*
- funcția *sunet()* să fie suprascrisă în fiecare clasă derivată
- în funcția *main()*, să se creeze un vector de pointeri la clasa *Animal* și să se afișeze sunetul fiecărui animal

2. Folosindu-se funcțiile virtuale pure și polimorfismul, să se creeze:
    - o clasă abstractă numită *Forma* care conține o funcție virtuală pură numită *aria()*
    - 2 clase derivate: Dreptunghi și Cerc în care să se suprascrie funcția *aria()* pentru fiecare formă
    - în funcția *main()*, să se creeze un vector de pointeri și să se afișeze aria fiecărei forme.

10.9 Standard Template Library

Standard Template Library (STL) este o bibliotecă puternică din C++ care oferă un set de containere, algoritmi și iteratori pentru gestionarea eficientă a datelor. Aceasta permite scrierea de cod modular, reutilizabil și eficient.

Componentele STL sunt:

1. **Containere** - sunt structuri de date predefinite (exemplu: vector, list, map, stack, queue).

Exemplu folosindu-se *stack*:

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << "Elementul din vârf: " << s.top() << endl; // Output: 30
    s.pop();
    cout << "Elementul din vârf: " << s.top() << endl; // Output: 20
    return 0;
}
```

Algoritmii sunt funcții care operează pe containere (exemplu: sort, find, reverse).

Exemplu de sortare folosindu-se *sort()*:

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    vector<int> v = {5, 2, 8, 1, 3};
    sort(v.begin(), v.end());  // Sortează în ordine crescătoare
    for (int num : v) {
        cout << num << " ";  // Output: 1 2 3 5 8
    }
    return 0;
}
```

2. Iteratori - sunt obiecte folosite pentru a parcurge elementele din containere. Tipuri de iteratori: begin(), end().

Exemplu de iteratori pentru un vector:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v = {10, 20, 30, 40};
```

```
    // Iterator clasic
    vector<int>::iterator it;
    for (it = v.begin(); it != v.end(); ++it) {
        cout << *it << " ";  // Output: 10 20 30 40
    }
    cout << endl;
    // Iterator invers
    for (auto rit = v.rbegin(); rit != v.rend(); ++rit) {
        cout << *rit << " ";  // Output: 40 30 20 10
    }
    return 0;
}
```

10.10 Clase template

O clasă template în C++ este un tip de programare generică care permite crearea de clase flexibile și reutilizabile ce pot opera pe diferite tipuri de date (int, double, string, etc.) fără a fi necesară rescrierea codului.

Exemplu simplu de Clasă Template: se crează o clasă Cutie care stochează un singur obiect de orice tip:

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

```
// Definire clasă template
template <typename T>
class Cutie {
private:
    T valoare;  // Poate fi orice tip de dată
public:
    Cutie(T v) : valoare(v) {}  // Constructor
    void afiseaza() {
        cout << "Valoarea din cutie: " << valoare << endl;
    }
};
int main() {
    Cutie<int> c1(10);        // Cutie cu un număr întreg
    Cutie<double> c2(5.5);    // Cutie cu un număr real
    Cutie<string> c3("Salut"); // Cutie cu un string
    c1.afiseaza();  // Output: Valoarea din cutie: 10
    c2.afiseaza();  // Output: Valoarea din cutie: 5.5
    c3.afiseaza();  // Output: Valoarea din cutie: Salut
    return 0;
}
```

Explicație:

- *template <typename T>* - se definește *T* ca tip generic.
- *Cutie<T>* - clasa funcționează pentru orice tip de date (int, double, string, etc.).
- *Cutie<int>*, *Cutie<double>* - se instanțiază clasa pentru un tip specific.

Avantajele claselor Template:

- reutilizare – scriem o singură clasă și o folosim pentru orice tip de date
- flexibilitate – funcționează cu orice tip, de la int la structuri complexe
- eficiență – evită scrierea de cod duplicat și permite optimizări

Clasele template sunt un instrument puternic în C++ pentru programarea generică. Ele ajută la crearea de clase reutilizabile, care pot lucra cu diferite tipuri de date, fără a rescrie codul.