Introducere în programarea CALCULATOARELOR

Angel CAȚARON

ISBN
978–606–19–1135–6

EDITURA
UNIVERSITĂȚII
TRANSILVANIA
DIN BRAȘOV

Angel Cațaron

INTRODUCERE ÎN PROGRAMAREA CALCULATOARELOR

EDITURA
UNIVERSITĂȚII
TRANSILVANIA
DIN BRAȘOV

Introducere

Acest volum oferă o introducere în programarea calculatoarelor pornind de la identificatori, date și tipuri de dată, declarații, variabile și constante, expresii, operații de intrare și ieșire, până la elementele de programare structurată de tip secvență, selecție, buclă, blocuri de instrucțiuni și subrutine. Exemplele sunt oferite sub forma unor programe care pot fi testate în orice mediu de programare cu suport pentru limbajul C++, iar noțiunile prezentate aici facilitează abordarea cu ușurință și a altor limbaje de programare.

Lucrarea se adresează în egală măsură specialiștilor care doresc o inițiere în programarea calculatoarelor, dar și celor care urmăresc să își aprofundeze și să își consolideze cunoștințele.

Cuprins

1. Scrierea programelor

Activitățile umane sunt caracterizate, în general, de secvenţe logice. Învățăm să executăm diverse comenzi, dar învățăm și care sunt tipurile de comportamente pe care putem să le așteptăm de la alţi indivizi. Matematica lucrează, de asemenea, cu secvenţe logice de paşi pentru rezolvarea problemelor sau pentru demonstrarea teoremelor. La fel, procesele de producție constau din succesiuni de operaţii executate într-o anumită ordine.

Atunci când ordonăm un proces, îl *programăm*. În mod particular, ne referim mai departe la programarea unui aparat: *calculatorul*.

Calculatorul este un dispoztiv programabil care poate păstra, regăsi şi procesa date.

Aşa cum un program descrie acţiunile care trebuie executate pentru a atinge un scop, un *program de calculator* descrie paşii pe care trebuie să îi execute calculatorul pentru a rezolva o problemă. În contextul acestui volum, atunci când vom vorbi de *programare* ne vom referi la programarea calculatorului.

Un program de calculator este o listă de instrucţiuni care trebuie urmate de calculator.

Un calculator ne permite să realizăm o serie de acţiuni într-un mod mult mai rapid şi mai precis decât o putem face fără ajutorul său. Pentru a îl folosi, însă, trebuie să specificăm ce dorim să facem, dar şi în ce ordine. Aceste lucruri le facem prin *programare*.

1.1 Cum scriem un program?

Pentru a scrie un program trebuie să parcurgem două faze:

- rezolvarea problemei
- implementarea.

Faza de rezolvare a problemei

1. *Analiza* înseamnă înţelegerea, definirea problemei şi a soluţiei ce trebuie dată;
2. *Algoritmul* presupune dezvoltarea unei secvenţe logice de paşi care trebuie urmaţi pentru rezolvarea problemei;
3. *Verificarea* este parcurgerea paşilor algoritmului pe mai multe exemple pentru a fi siguri că rezolvă problema pentru toate cazurile.

Faza de implementare

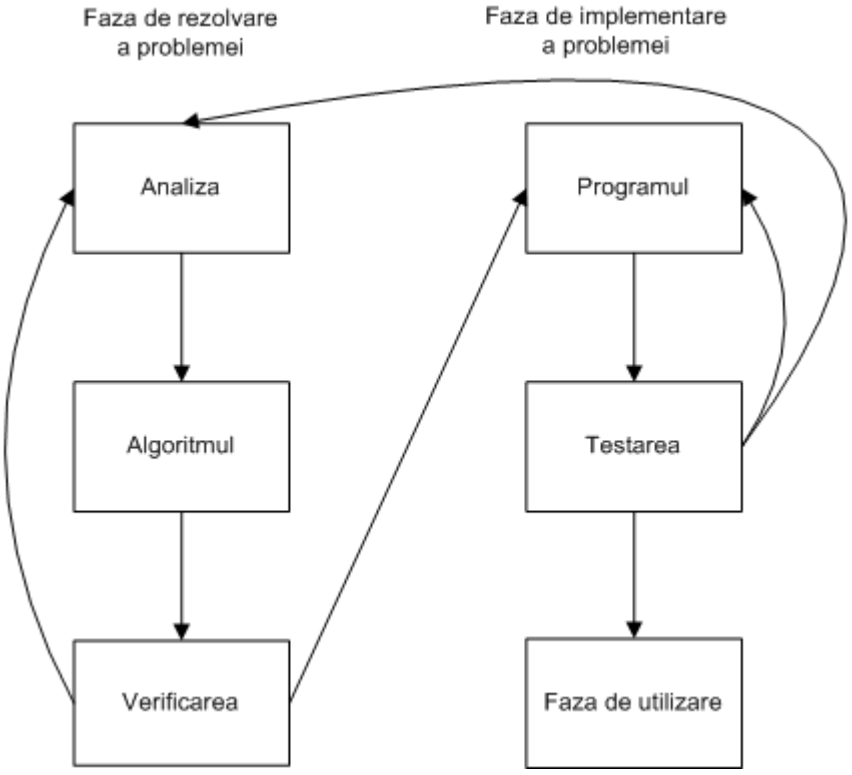
1. *Programul* reprezintă translatarea algoritmului într-un limbaj de programare
2. *Testarea* este etapa în care ne asigurăm că instrucţiunile din program sunt urmate corect de calculator. În situaţia în care constatăm că sunt erori, trebuie să revedem algoritmul şi programul pentru a determina sursa erorilor şi pentru a le corecta.

Aceste două faze de dezvoltare a programului sunt urmate de *faza de utilizare* a programului care înseamnă folosirea acestuia pentru rezolvarea problemelor reale, cele pentru care a fost conceput. Ulterior pot interveni *modificări* ale programului fie pentru a răspunde noilor cerinţe ale utilizatorilor, fie pentru corectarea erorilor care apar în timpul utilizării şi care nu au putut fi găsite în faza de testare.

Calculatorul nu este inteligent. El nu poate analiza problema şi nu poate să dea o soluţie. Programatorul trebuie să analizeze problema, să dea soluţia şi apoi să o comunice calculatorului. Avantajul folosirii calculatorului este că el rezolvă

problemele rapid şi fără erori, eliberându-ne totodată de realizarea unor operaţii repetitive şi consumatoare de timp.

Programatorul începe operaţia de programare prin analiza problemei şi dezvoltarea unei soluţii generale numită *algoritm*. *Algoritmul* este o procedură care descrie paşii ce trebuie parcurşi pentru rezolvarea unei probleme într-un timp finit. Algoritmul este esenţial pentru procesul de programare. Programul este de fapt un algoritm scris pentru calculator.



Un algoritm este o secvenţă logică de acţiuni. Folosim adeseori algoritmi: reţetele culinare, instrucţiunile de folosire sunt exemple de algoritmi care nu sunt, însă, programe. Un exemplu de algoritm poate fi o succesiune de paşi care trebuie urmaţi pentru a porni un autoturism. Un alt exemplu este calculul sumei care trebuie plătită unui salariat după 5 zile lucrătoare.

1. Verificarea sumei plătite pe oră
2. Determinarea numărului total de ore lucrate
3. Se înmulţeşte numărul de ore cu suma plătită pe oră
4. Dacă numărul de ore depăşeşte 40, atunci se scade 40 din numărul de ore lucrate, iar diferenţa de ore se înmulţeşte cu 0,5 ori suma plătită pe oră
5. Adună sumele de la punctele 3 şi 4 şi stabileşte suma finală.

După dezvoltarea soluţiei generale, programatorul poate testa algoritmul mental sau în scris. Dacă algoritmul nu este corect, reluăm paşii descrişi mai devreme.

Când programatorul este satisfăcut de algoritm, poate să îl translateze într-un program scris într-un *limbaj de programare*.

Limbajul de programare este un set de reguli, simboluri şi cuvinte speciale folosite pentru a construi un program.

Limbajul C++ este o variantă simplificată a limbii engleze şi care are un set strict de reguli gramaticale. Datorită numărului mic de cuvinte disponibile, sunteţi obligaţi să scrieţi instrucţiuni simple şi exacte.

Codarea este translatarea algoritmului într-un limbaj de programare.

Execuţia este rularea programului pe calculator (*running*).

Depanarea este faza de determinare şi corectare a erorilor (*debugging*).

Implementarea este combinaţia dintre codarea şi testarea algoritmului.

O parte importantă a programării este scrierea documentaţiei. *Documentaţia* este reprezentată de un text scris şi de comentariile necesare înţelegerii de către alte persoane a programului scris de noi.

După scrierea programului, trebuie să transmitem calculatorului informaţiile sau datele necesare rezolvării problemei.

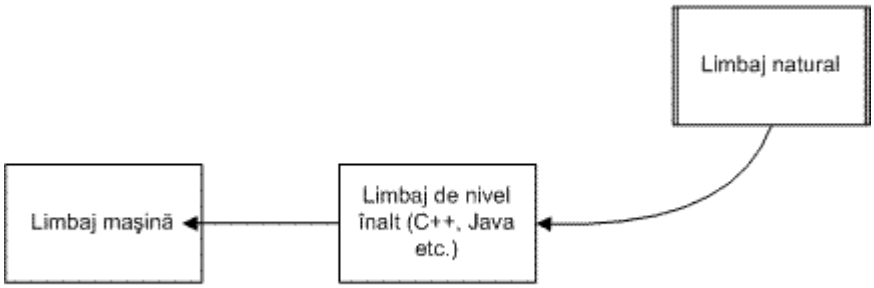
Informaţia este orice cunoştinţă care poate fi comunicată, inclusiv idei abstracte sau concepte.

Datele sunt informaţii transpuse într-o formă care poate fi înţeleasă de calculator.

1.2 Ce este un limbaj de programare?

Programatorii scriu instrucţiuni în diverse limbaje de programare, unele care sunt înţelese în mod direct de calculator, altele care necesită mai mulţi paşi de *translatare*. În prezent există sute de limbaje de programare care pot fi împărţite în trei tipuri generale:

1. Limbaje maşină
2. Limbaje de asamblare
3. Limbaje de nivel înalt



Singurul limbaj de programare pe care calculatorul îl poate executa în mod direct este un set primitiv de instrucţiuni numit *limbaj maşină* sau *cod maşină*. Acesta este “limbajul natural” al unui calculator şi este definit de alcătuirea hardware a fiecărui calculator în parte. Un anumit limbaj maşină poate fi folosit doar pentru un anumit tip de calculator. Limbajul maşină este alcătuit din instrucţiuni codate binar şi poate fi folosit direct de calculator. Limbajele maşină sunt greu de folosit de programatori, aşa cum se poate vedea din următoarea secţiune de program scris în limbaj maşină care adună o sumă suplimentară de bani la suma de bază pe care o primeşte un angajat, rezultând suma finală.

Exemplu

```
+1300042774
+1400593419
+1200274027
```

Pe măsură ce calculatoarele au devenit tot mai populare, a devenit evident că limbajul maşină este greu de folosit, dezvoltarea aplicaţiilor este foarte lentă şi

probabilitatea de apariţie a erorilor de programare este foarte mare. În loc să se folosească numere pentru a programa calculatoarele, s-a trecut la folosirea unor abrevieri ale unor cuvinte din limba engleză care reprezintă operaţii elementare pentru calculator. Aceste abreviaţii formează baza *limbajelor de asamblare*. În acelaşi timp au fost dezvoltate *programe de translatare* sau *asambloare* pentru a converti programele scrise in limbaj de asamblare către programe în limbaj maşină. Secvenţa de instrucţiuni de mai jos realizează aceleaşi operaţii ca cele din exemplul anterior, dar într-o manieră mai clară decât echivalentul în limbaj maşină.

Exemplu

```
LOAD BASEPAY
ADD OVERTIMEPAY
STORE TOTALPAY
```

În prezent se folosesc *limbaje de nivel înalt*, mult mai uşor de folosit decât codul maşină şi care accelerează procesul de dezvoltare software. Un program numit *compilator* translatează un program scris într-un limbaj de nivel înalt în limbaj maşină. Iată o variantă scrisă într-un limbaj de nivel înalt a programului de mai sus.

Exemplu

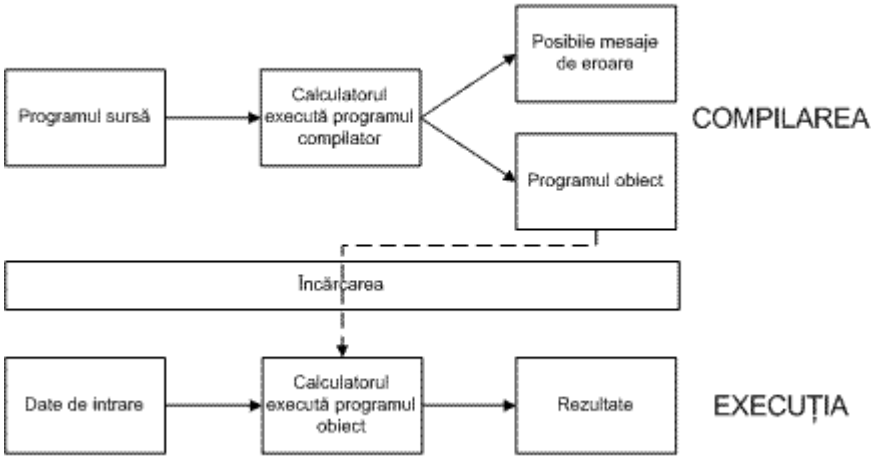
```
totalPay = basePay + overTimePay
```

Pentru a putea rula un program de nivel înalt pe un calculator, pe acesta trebuie să existe un compilator adaptat limbajului dar şi calculatorului.

Programul sursă este un program scris într-un limbaj de nivel înalt.

Programul obiect este versiunea în limbaj maşină a programului sursă şi se obţine în urma compilării.

Compilarea şi *execuţia* sunt două procese distincte.



Instrucţiunile dintr-un limbaj de programare reflectă operaţiile pe care le poate realiza un calculator:

- transferarea datelor dintr-un loc în altul
- citirea datelor de la un dispozitiv de intrare (ex. tastatura) şi transmiterea lor către un dispozitiv de ieşire (ex. ecran)
- stocarea şi aducerea datelor din memorie sau alte dispozitive de memorare
- compararea a două date pentru stabilirea egalităţii sau a inegalităţii
- operaţii aritmetice

Scurt istoric al limbajelor C şi C++

Limbajul C++ a evoluat din limbajul C care, la rândul său, a avut la bază alte două limbaje de programare, BCPL şi B. Limbajul BCPL a fost dezvoltat în 1967 de Martin Richards ca limbaj pentru scrierea sistemelor de operare şi a compilatoarelor. Ken Thompson a modelat multe elemente ale BCPL în limbajul B pe care l-a folosit

pentru scrierea uneia dintre primele versiuni ale limbajului UNIX la Bell Laboratories în 1970. Aceste două limbaje de programare nu foloseau tipuri de date, fiecare dată avea aceeaşi dimensiune în memorie, iar tratarea unei date ca întreg sau real era în responsabilitatea programatorului.

Limbajul C a fost dezvoltat din limbajul B la Bell Laboratories în 1972 de Dennis Ritchie. Iniţial a fost folosit pentru dezvoltarea sistemului de operare UNIX, iar astăzi majoritatea sistemelor de operare sunt scrise în C şi C++.

Limbajul C++ este o extensie a lui C şi a fost creat la începutul anilor 1980 de Bjarne Stroustroup tot la Bell Laboratories. Are toate elementele limbajului C, dar oferă posibilitatea *programării orientate pe obiecte*. *Obiectele* sunt, în principiu, componente software *reutilizabile* care modelează elemente din lumea reală. S-a dovedit că folosirea unei abordări modulare, a design-ului şi a implementării orientate pe obiecte poate face ca grupurile de dezvoltare să fie mult mai productive decât atunci când se folosesc alte tehnici de programare, cum ar fi programarea procedurală.

Biblioteca standard C++

Programele C++ constau din elemente numite *clase* şi *funcţii*. Puteţi programa fiecare piesă de care aveţi nevoie pentru a alcătui un program. Dar cei mai mulţi programatori folosesc avantajul oferit de bogata colecţie de clase şi funcţii oferite de biblioteca standard C++. De aceea, învăţarea limbajului C++ înseamnă, pe de o parte învăţarea limbajului în sine şi, pe de altă parte, deprinderea modului în care se pot folosi clasele şi funcţiile din biblioteca standard C++. Aceste clase şi funcţii sunt, de regulă, oferite odată cu compilatorul, dar există multe biblioteci suplimentare care sunt realizate de companii software independente. Unele dintre acestea pot fi descărcate în mod liber de pe Internet.

Avantajul creării propriilor noastre funcţii şi clase este că vom şti exact cum lucrează, dar dezavantajul este timpul consumat şi efortul depus pentru proiectarea, dezvoltarea şi întreţinerea noilor clase şi funcţii pentru a opera corect şi eficient.

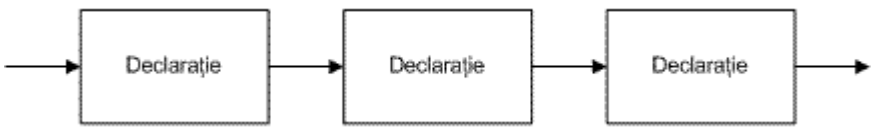
Programarea structurată

După anii 1960, când aplicaţiile au început să devină din ce în ce mai complexe şi când costurile de dezvoltare au început să devină foarte mari, lumea a realizat că acest proces este mai complex decât s-a estimat iniţial. Activităţile de cercetare în domeniu au rezultat în evoluţia către *programarea structurată*, o abordare disciplinată în scrierea programelor care au devenit mai clare, mai uşor de testat, de corectat şi de modificat.

Structuri de program

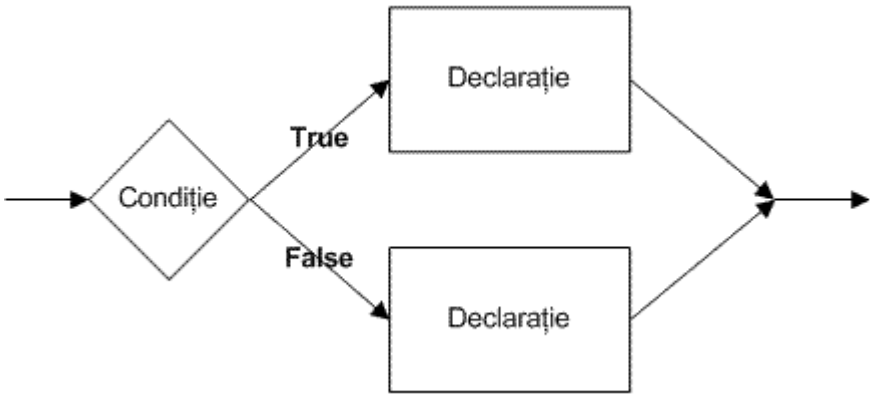
Limbajele de programare folosesc anumite structuri pentru a transpune algoritmii în programe.

Secvenţa



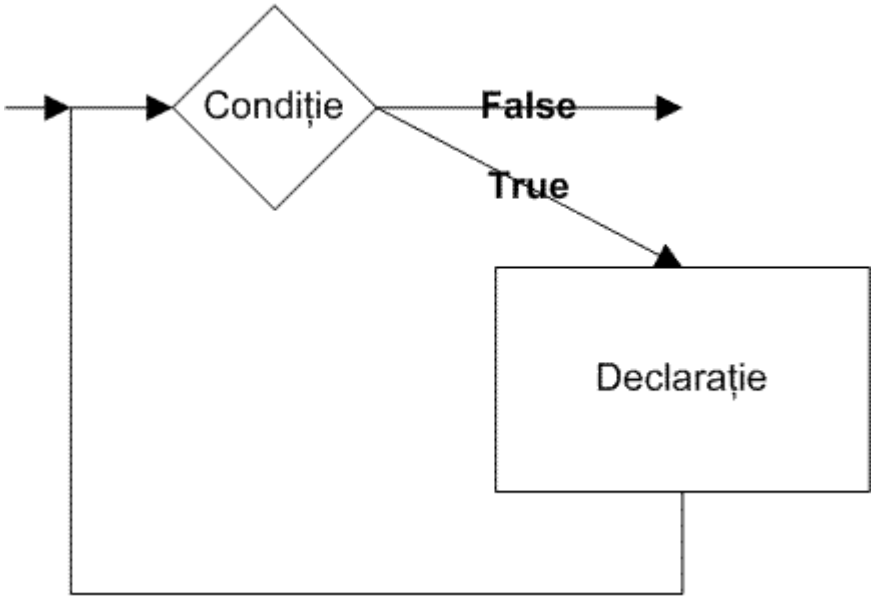
Secvenţa este o serie de declaraţii executate una după alta.

Selecţia (deciza)



`IF` condiţie `THEN` declaraţie1 `ELSE` declaraţie2

Bucla (repetiţia sau iteraţia)



`WHILE` condiţie `DO` declaraţie

Subprogramul (procedura, funcţia, subrutina)

Subprogramul poate fi o combinaţie a structurilor anterioare. Ne permite scrierea separată a unor părţi din program şi apoi asamblarea lor într-o formă finală.

Programarea orientată pe obiecte

Evoluţiile pozitive în dezvoltarea software au început să apară odată cu folosirea programării structurate. În anii 1980, tehnologia programării orientate pe obiecte a început să fie folosită pe scară largă în proiectarea şi dezvoltarea software.

Tehnologia obiectelor este, în principiu, o schemă de „împachetare" care permite crearea unităţilor software cu o semnificaţie proprie. Acestea sunt focalizate pe părţi specifice ale unei aplicaţii. Se pot crea obiecte pentru date, pentru plăţi, pentru facturi, obiecte video, obiecte fişier etc. De fapt, orice substantiv poate fi transpus într-un obiect.

Trăim într-o lume de obiecte. Există în jurul nostru maşini, avioane, oameni, animale, clădiri etc. Înaintea apariţiei limbajelor orientate pe obiecte, limbajele de programare (de ex. FORTRAN, Pascal, Basic, C) erau focalizate pe acţiuni (verbe) şi nu pe obiecte (substantive). O lume a obiectelor trebuie transpunsă, puţin forţat, într-o lume a acţiunilor. Paradigma orientării pe obiecte prin limbaje cum ar fi Java sau C++ a făcut posibil ca programarea să devină o prelungire a realităţii. Acesta este un proces mai natural decât programarea procedurală şi conduce la creşteri semnificative de productivitate.

Una dintre problemele majore ale programării procedurale este ca unităţile de program nu modelează foarte firesc entităţi ale lumii reale şi, de aceea, nu pot fi reutilizate în mod facil. Fiecare nou proiect în programarea procedurală presupunea scrierea codului „de la zero", lucru care înseamnă o risipă de timp, bani şi resurse umane. Tehnologia obiectelor face ca entităţile create într-un proiect (obiectele), dacă sunt corect concepute, să poată fi folosite şi în proiecte viitoare.

Pe de altă parte, este remarcabil faptul că uneori nu reutilizarea codului este marele avantaj al programării orientate pe obiecte, ci faptul că programele sunt mult mai uşor de înţeles, organizat, de întreţinut, de modificat sau de corectat.

Elementele de programare structurată sunt noţiuni cheie în programarea orientată pe obiecte. Fiecare clasă este o unitate care încorporează structuri de program.

1.3 Ce este un calculator?

Un calculator (*computer*) este un dispozitiv capabil să realizeze calcule şi să ia decizii logice cu viteze de milioane sau miliarde de ori mai mari decât oamenii. Aceasta înseamnă că unei persoane îi trebuie câteva milioane de secunde pentru a face calculele pe care le poate face un calculator într-o secundă.

Calculatorul procesează *date* sub controlul unor înşiruiri de instrucţiuni numite *programe de calculator*. Aceste programe dirijează calculatorul să realizeze secvenţe de acţiuni care au fost specificate de persoane numite *programatori*.

Un calculator este alcătuit din diverse dispozitive, cum ar fi tastatura, mouse-ul, discurile, memoria, CD-ROM-ul sau microprocesorul, toate acestea fiind numite generic *hardware*. Programele de calculator care rulează pe calculator sunt numite *software*. Costurile hardware-ului au scazut foarte mult în ultimii ani până la punctul în care un calculator personal a devenit foarte accesibil ca preţ. Din păcate, costurile pe care le implică dezvoltarea software au crescut în tot acest timp pe măsură ce aplicaţiile au devenit din ce în ce mai complexe. În acest volum vom studia metode de dezvoltare software cunoscute prin a căror utilizare se pot reduce costurile: programarea structurată, dezvoltarea top-down, separarea codului în mai multe funcţii ca premise pentru programarea orientată pe obiecte, programarea generică.

Organizarea unui calculator

Se poate programa şi fără a şti prea multe despre alcătuirea internă a calculatorului. Cunoaşterea părţilor componente ajută, însă, la înţelegerea efectului fiecărei instrucţiuni din program.

Calculatorul are 4 componente de bază:

1. *Unitatea de memorare* este o colecţie de celule care stochează datele. Fiecare astfel de celulă are o adresă. Aceste celule se numesc *celule de memorie* sau *locaţii de memorie*.
2. *Unitatea centrală de procesare* (CPU) este cea care urmăreşte instrucţiunile din program. Are două componente:
    a. *Unitatea aritmetico-logică* (ALU) care realizează operaţiile aritmetice şi logice
    b. *Unitatea de control* care controlează acţiunile celorlalte componente astfel încât instrucţiunile să se execute în ordinea corectă
3. *Dispozitivele de intrare/ieşire* (I/O) acceptă date care vor fi procesate şi le prezintă pe cele care au fost procesate
4. *Dispozitivele auxiliare de stocare* păstrează datele şi după oprirea calculatorului

Dispozitivele periferice sunt cele de intrare/ieşire şi cele auxiliare.

Toate aceste componente sunt cunoscute sub numele de *hardware*. Programele care permit hardware-ului să funcţioneze se numesc *software*.

Pe lângă programele scrise de noi, existăun set de programe numite *software de sistem* care simplifică *interfaţa* dintre calculator şi utilizator. În această categorie intră compilatoarele, sistemele de operare sau editoarele de text.

Sistemul de operare coordonează toate resursele calculatorului. El poate rula compilatorul, poate rula programe obiect, poate executa comenzi de sistem

Editorul este un program interactiv folosit pentru crearea şi modificarea programelor sursă sau a datelor.

1.4 Tehnici de rezolvare a problemelor

Adesea în viaţa de zi cu zi suntem puşi în situaţia de a *urma* algoritmi. În faza de rezolvare a unei probleme de programare va trebui să *proiectăm* algoritmi. Este important să ne punem cât mai multe întrebări până când înţelegem exact ce avem de făcut.

Folosirea soluţiilor existente – Întotdeauna trebuie să evităm să reinventăm roata. Dacă există o soluţie, atunci să o folosim. Dacă am rezolvat o problemă similară înainte, trebuie doar să repetăm soluţia pentru că în programare există probleme care apar foarte des (ex. calculul unui minim sau al unui maxim). Dacă avem la dispoziţie o secvenţă de cod care rezolvă o parte a problemei noastre, putem să o folosim. Această metodă se numeşte *software reuse* şi este elementul central în programarea orientată pe obiecte.

Divide et impera – Adeseori este mult mai uşor să rezolvăm o problemă dacă o împărţim în subprobleme mai mici. De altfel, metoda descompunerii unui program în funcţii sau tehnica programării orientate pe obiecte se bazează pe acest principiu.

Dificultatea de a începe – Programatorii se confruntă adesea cu o mare dificultate: se găsesc în faţa unei foi albe şi nu ştiu cum să înceapă. Privesc problema şi li se pare foarte complicată. Pentru a depăşi acest moment, rescrieţi problema cu propriile voastre cuvinte. Încercaţi să o descompuneţi în subprobleme individuale în loc să o analizaţi global. Acest lucru vă va ajuta să extrageţi

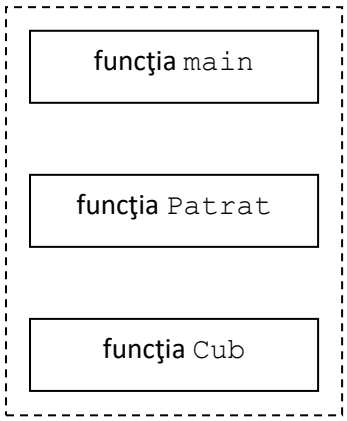
componente mai uşor de rezolvat. De asemenea, acest lucru vă va ajuta să sintetizaţi mai uşor algoritmul de rezolvare a problemei.

2. Sintaxa şi semantica C++

În acest capitol vom vedea care sunt principalele reguli şi simboluri care fac din C++ un limbaj de programare. Vom vedea, de asemenea, ce paşi trebuie parcurşi pentru a scrie un program simplu şi a-l face să ruleze pe un calculator.

2.1 Structura unui program C++

Subprogramele permit scrierea separată a unor părţi din program care, apoi, se asamblează în programul final. În C++, subprogramele se numesc *funcţii*, iar un program C++ este o colecţie de clase şi funcţii.



Fiecare program C++ trebuie să conţină o funcţie care se numeşte `main`. Aceasta poate fi privită ca o funcţie master pentru celelalte funcţii din program. Când `main` este programată să execute subprogramul corespunzător functie `Patrat` spunem că `main` *apelează* sau *invocă* funcţia `Patrat`.

După ce `Patrat` încheie execuţia tuturor instrucţiunilor, ea transmite (sau întoarce) controlul înapoi către funcţia `main` care îşi continuă execuţia.

Următorul exemplu este un program C++ care este alcătuit din 3 funcţii: `main`, `Patrat` şi `Cub`. Detaliile nu sunt importante.

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

int Patrat(int);
int Cub(int);

int main()
{
  cout << "Patratul lui 27 este " << Patrat(27) << endl;
  cout << "si cubul lui 27 este " << Cub(27) << endl;

  return 0;
}

int Patrat(int n)
{
  return n*n;
}

int Cub(int n)
{
  return n*n*n;
}
```

În fiecare dintre funcţii, acoladele { şi } marchează începutul şi sfârşitul instrucţiunilor care trebuie executate. Ceea ce se găseşte între acolade se numeşte *corpul funcţiei* sau *bloc de instrucţiuni*.

Execuţia unui program C++ începe întotdeauna cu prima instrucţiune din funcţia `main`. În programul de mai sus, aceasta este

```
cout << "Patratul lui 27 este " << Patrat(27) << endl;
```

Aceasta este o instrucţiune care produce afişarea unor informaţii pe ecranul calculatorului care este dispozitivul standard de ieșire. Detalii vom afla puţin mai târziu, tot în acest capitol. Pe scurt, instrucţiunea tipăreşte două elemente. Primul este mesajul

```
Patratul lui 27 este
```

Cel de-al doilea este o valoare obţinută prin apelul (invocarea) funcţiei Patrat cu valoarea `27`. Această funcţie realizează ridicarea la pătrat şi trimite rezultatul, `729`, înapoi către *apelant* (*funcţia invocatoare*), adică funcţia `main`. Acum `main` continuă execuţia tipărind valoarea `729` după care trece la instrucţiunea următoare.

Similar, a doua instrucţiune din funcţia `main` tipăreşte mesajul

```
si cubul lui 27 este
```

după care invocă funcţia `Cub`. Aceasta întoarce rezultatul `19683`, care este tipărit. Rezultatul final va fi

```
Patratul lui 27 este 729
si cubul lui 27 este 19683
```

Atat `Patrat` cât şi `Cub` sunt exemple de funcţii care returnează o valoare. O astfel de funcţie transmite o singură valoare către funcţia apelant. Cuvântul `int` aflat la începutul primei linii a funcţiei `Patrat` arată că funcţia întoarce o valoare întreagă (un număr întreg).

Să revenim la funcţia `main`. Prima ei linie este

```
int main()
```

Cuvântul `int` indică faptul că `main` este o funcţie care întoarce o singură valoare, un număr întreg. După tipărirea pătratului şi a cubului lui 27, `main` execută instrucţiunea

```
return 0;
```

pentru a întoarce valoarea 0 către apelant. Dar cine apelează funcţia `main`? Răspunsul este: sistemul de operare. Acesta aşteaptă o valoare (exit status) de la `main` după ce aceasta îşi încheie execuţia. Prin convenţie, valoarea returnată 0 înseamă că totul a decurs OK. O altă valoare (1, 2 etc.) înseamnă că s-a petrecut ceva nedorit.

2.2 Sintaxă şi semantică

Vom începe acum să intrăm în detalii legate de programarea în C++.

Un limbaj de programare este un set de reguli, simboluri şi cuvinte speciale folosite pentru a scrie un program. Regulile sunt valabile atât pentru *sintaxă* (gramatică), cât şi pentru *semantică* (semnificaţie).

Sintaxa este un set de reguli care definesc exact ce combinaţii de litere, numere şi simboluri pot fi folosite într-un limbaj de programare. Nu se acceptă ambiguităţi. Vom vedea că încălcarea oricărei reguli a limbajului, de exemplu scrierea incorectă a unui cuvânt sau uitarea unei virgule pot genera *erori de sintaxă* (syntax errors) şi codul sursă nu poate fi compilat până la corectarea lor.

Semantica este un set de reguli care determină semnificaţia instrucţiunilor scrise într-un limbaj de programare.

Şabloane sintactice

În acest capitol vom folosi şabloane ca şi exemple generice de construcţii sintactice în C++. Cel mai frecvent vom folosi şabloane asemănătoare celui pentru funcţia `main`:

Funcţia `main`

```
int main()
{
      instrucţiune
      …
}
```

Acest şablon arată că funcţia main începe cu cuvântul int urmat de cuvântul main şi o pereche de paranteze rotunde. Prima linie a oricărei funcţii numeşte *heading* sau *antet*. Acest heading este urmat de o acoladă care marchează începutul unei liste de instrucţiuni - *corpul funcţiei*. În final, acolada închisă indică sfârşitul funcţiei. Cele 3 puncte indică faptul că instrucţiunea poate fi urmată de 0 sau mai multe alte instrucţiuni.

Denumirea elementelor programului: identificatorii

În C++ *identificatorii* sunt nume asociate funcţiilor, claselor sau datelor şi sunt folosite pentru a referi funcţii, clase sau date.

Identificatorii sunt alcătuiţi din litere (A-Z, a-z), cifre (0-9) şi caracterul underscore (_), dar trebuie să înceapă cu o literă sau cu underscore.

Exemplu

Identificatori corecţi: `J9`, `GetData`, `sum_of_squares`

Identificatori incorecţi:

`40Hours`	- nu poate începe cu o cifră
`Get Data`	- nu poate conţine spaţiu
`box-22`	- nu poate conţine – pentru că este simbol matematic
`int`	- cuvântul `int` este predefinit în C++

Cuvântul `int` este cuvânt rezervat. Acestea sunt cuvinte care au o utilizare specială în C++ şi nu pot fi utilizate drept identificatori definiţi de programator.

Este foarte util ca identificatorii să fie sugestivi şi uşor de citit.

Exemplu

`PRINTTOPPORTION` faţă de `PrintTopPortion`

2.3 Date şi tipuri de date

De unde ia programul datele de care are nevoie pentru a lucra? Datele sunt păstrate în memoria calculatorului. Fiecare locaţie are o adresă unică pe care o referim atunci când dorim să stocăm sau să aducem date. Adresa fiecărei locaţii de memorie este un număr binar. În C++ folosim identificatori pentru a denumi locaţiile de memorie. Acesta este unul dintre avantajele limbajelor de programare de nivel înalt: ne eliberează de grija de a gestiona locaţiile de memorie la care sunt păstrate datele şi instrucţiunile. În C++ fiecare dată trebuie să fie de un anumit tip. Tipul de dată determină modul în care datele sunt reprezentate în interiorul calculatorului şi operaţiile care se pot realiza asupra lor.

C++ defineşte un set de tipuri standard de date, pe care le vom descrie mai jos. De asemenea, programatorul îşi poate defini propriile tipuri de date. Tipurile standard sunt organizate astfel:

- Tipuri simple
  - Tipuri integrale

        - `char`
        - `short`
        - `int`
        - `long`
        - `enum`
    - Tipuri reale
        - `float`
        - `double`
        - `long double`
- Tipuri adresă
    - pointer
    - referinţă
- Tipuri structurate
    - tablou (array)
    - `struct`
    - `union`
    - `class`

Tipurile integrale

Se numesc aşa pentru că se referă la valori întregi. Despre tipul `enum` nu discutăm în acest capitol. Întregii sunt secvenţe de una sau mai multe cifre. Nu se admite virgula. În multe cazuri, semnul – precedă un întreg.

Exemplu

```
22  16  1  -378  -912
```

Atunci când cuvântul rezervat unsigned precedă un tip de dată, valoarea întreagă poate fi doar pozitivă sau 0.

Exemplu

```
unsigned int
```

Tipurile de dată `char`, `short`, `int` şi `long` reprezintă întregi de diferite dimensiuni cu mai mulţi sau mai puţini biţi.

celula de memorie `char`
celula de memorie `short`
celula de memorie `int`
celula de memorie `long`

În general, cu cât sunt mai mulţi biţi în celula de memorie, cu atât mai mari sunt valorile întregi care pot fi memorate acolo. Dimensiunile acestor celule de memorie sunt dependente de maşină. Pe unele calculatoare, o dată de tip `int` se poate găsi în intervalul -32768 ... +32767, iar pe altele între -2147483648 ... +2147483647. Când programul încearcă să calculeze o valoare mai mare decât valoarea maximă, rezultatul este un *integer overflow*. Un întreg care începe cu cifra 0 este considerat ca fiind scris în baza 8.

Exemplu

`015` este 15_8

Tipul `char`. Acesta este cel mai mic tip de dată care poate fi folosit pentru a reprezenta valori întregi. Se obişnuieşte folosirea acestui tip de dată atunci când se doreşte o economie de memorie şi se folosesc întregi mici. Însă tipul `char`, în mod tipic, se foloseşte pentru a descrie date care constau dintr-un caracter alfanumeric din setul de caractere ASCII (literă, cifră sau simbol special).

Exemplu

```
'A'  'a'  '8'  '2'  '+'  '$'  ' '
```

Fiecare caracter este cuprins între apostroafe, astfel încât C++ face diferenţa dintre data de tip caracter `'8'` şi valoarea întreagă `8`, pentru că cele două sunt păstrate în mod diferit în interiorul calculatorului.

Nu sunt obişnuite operaţii de adunare a caracterului `'A'` cu caracterul `'B'`, de scădere a caracterului `'3'` din caracterul `'8'`, însă aceste caractere se pot compara. Caracterul `'A'` este întotdeauna mai mic decât `'B'`, `'B'` mai mic decât `'C'` etc. Fiecare set de caractere defineşte o astfel de secvenţă.

Tipurile reale (virgulă mobilă)

Aceste tipuri de date se utilizează pentru a reprezenta numere reale. Numerele reprezentate în virgulă mobilă au parte întreagă şi o parte fracţionară, separate de un punct.

Exemplu

```
18.0  127.54  .8  0.57
```

Numărul `0.57` nu este octal. Această regulă este valabilă doar pentru numere întregi.

Aşa cum tipurile integrale din C++ au diferite dimensiuni, la fel se întâmplă şi cu tipurile reale. În ordine crescătoare, acestea sunt `float`, `double` şi `long double`.

Valorile reprezentate în virgulă mobilă pot avea un exponent, ca în notaţia ştiinţifică (un număr este scris ca o valoare înmulţită cu 10 ridicat la putere). În loc de 3.504×10^{12}, în C++ scriem `3.504e12`. „`e`” înseamnă exponent al bazei 10. Numărul dinaintea lui `e` nu trebuie să includă în mod obligatoriu punctul zecimal.

Dintre tipurile de dată reale, cel mai folosit este `float` care, adeseori, este suficient. Valoarea maximă oferită de tipul `float` este, în general, în jur de `3.4e+38`.

Calculatoarele nu pot reprezenta întotdeauna numerele în virgulă mobilă. Datorită faptului că memorarea se face în formă binară, multe valori reale pot fi doar aproximate în acest sistem. De aceea, nu trebuie să ne mire că, pe unele calculatoare, de exemplu `4.8` va fi afişat `4.7999998`, fără a fi vorba de o eroare de programare.

2.4 Declaraţiile

Identificatorii pot fi utilizaţi pentru a denumi constante sau variabile, adică locaţii de memorie al căror conţinut se permite să fie modificat.

Cum spunem calculatorului ce reprezintă un identificator?

Declaraţia este o instrucţiune care asociază un nume (un identificator) cu o dată, o funcţie sau un tip de dată, astfel încât programatorul poate să se refere la acest element prin nume.

Este la fel cum o definiţie dintr-un dicţionar asociază un nume unei descrieri a elementului la care ne referim prin acest nume.

De exemplu, declaraţia

```
int empNum;
```

Anunţă că `empNum` este numele unei variabile al cărei conţinut este de tip `int`. Când declarăm o variabilă, compilatorul alege o locaţie de memorie şi o asociază cu identificatorul păstrând această asociere la dispoziţia programului. Orice identificator dintr-un program trebuie să fie unic în domeniul ei.

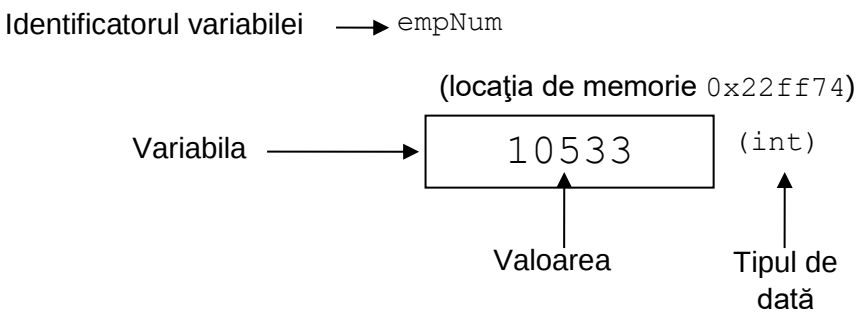
În C++ un identificator trebuie să fie declarat înainte de a fi folosit. Dacă declarăm un identificator ca fiind o constantă şi încercăm mai târziu să îi modificăm valoarea, compilatorul detectează această inconsistenţă şi semnalează eroare.

Constantele şi variabilele poartă numele generic de *date*.

2.5 Variabilele

Variabila este o locaţie de memorie referită printr-un identificator şi care păstrează valoarea unei date ce poate fi modificată.

Numele simbolic asociat variabilei se numeşte *identificatorul variabilei* sau *numele variabilei*.



Declararea variabilei înseamnă specificarea atât a numelui ei cât şi a tipului de dată. O declaraţie se termina întotdeauna cu caracterul ;

Exemplu

```
int empNum;
```

Se pot declara mai multe variabile de acelaşi tip într-o singură declaraţie.

Exemplu

```
int studentCount, maxScore, sumOfScores;
```

Aceasta declaţie este identică cu:

```
int studentCount;
int maxScore;
int sumOfScores;
```

Declararea fiecărei variabile într-o instrucţiune separată ne permite să adăugăm comentarii pentru înţelegerea declaraţiilor la o parcurgere ulterioară a programului.

Exemplu

```
float payRate; //Employee's pay rate
```

Comentariile sunt precedate de `//` şi sunt ignorate de compilator.

2.6 Constantele

Toate numerele, întregi sau reale, sunt constante. La fel, caracterele (cuprinse între `' '`) şi secvenţele de caractere sau string-urile, şirurile (cuprinse între `" "`).

Exemplu

```
16  32.3  'A'  "Boys"
```

În C++ ca şi în matematică, o constantă este un element al cărui valoare nu se schimbă niciodată.

Folosim constante ca părţi ale unor expresii aritmetice. Putem scrie o instrucţiune care adună constantele `5` şi `6` şi plasează rezultatul în variabila numită `sum`. Numim *valoare literală* (sau literal) orice valoare constantă din program.

O alternativă la literale sunt *constantele simbolice* (sau constante cu nume). Acestea sunt locaţii de memorie referite printr-un identificator care păstrează date ce

pot fi modificate. De exemplu, în loc să folosim literalul `3.14159` folosim constanta simbolică `PI`. Acest lucru face programul mai uşor de citit şi de modificat.

Declararea constantelor în C++ începe cu cuvântul rezervat `const`, iar semnul = se aşează între identificator şi valoarea literală.

Exemplu

```
const char BLANK = ' ';
const float PI = 3.14159;
const int MAX = 20;
```

De regulă, constantele simbolice se scriu cu litere mari pentru a le distinge mai uşor de variabile la citirea programului.

2.7 Acţiuni: instrucţiuni executabile

Asignarea

Valoarea unei variabile poate fi schimbată prin asignare.

Exemplu

```
quizScore = 10;
```

Valoarea `10` este asignată variabilei `quizScore`, adică valoarea `10` va fi stocată în locaţia de memorie numită `quizScore`. Semantica operatorului de asignare = este „păstrează”, „stochează”. Orice valoare anterioară stocată la acea locaţie de memorie se pierde, fiind înlocuită de noua valoare.

Într-o instrucţiune de asignare, doar o variabilă poate apărea în stânga operaţiei. Asignarea nu este ca o ecuaţie matematică (x+y=z+4).

Având declaraţiile:

```
int num;
 int alpha;
 float rate;
 char ch;
```

putem face următoarele asignări:

```
 alpha = 2856;
 rate = 0.36;
 ch = 'B';
 num = alpha;
```

Asignarea

```
 ch = ”Hello”;
```

nu este corectă pentru că `ch` este o variabilă de tip `char` iar `”Hello”` este un string.

Pentru o mai mare lizibilitate a programelor pe care le scriem, vom respecta următoarele reguli:

- Folosim iniţiala mică pentru numele de variabile

Exemplu

```
lengthsInYards  hours
```

- Folosim iniţiala mare pentru numele de funcţii sau clase

Exemplu

```
Cub(27)  MyDataType
```

- Folosim litere mari pentru constante simbolice

Exemplu

```
UPPER_LIMIT  PI
```

Expresia din dreapta operatorului de asignare este evaluată şi valoarea ei este stocată în variabila aflată în stânga operaturului.

O *expresie* este alcătuită din constante, variabile şi operatori.

Exemplu
```
alpha+2  rate-6.0  alpha*num
```
Operatorii admişi într-o expresie depind de tipurile de date ale constantelor şi ale variabilelor din expresie.

Operatorii matematici sunt:

Plus unar
Minus unar
Adunare
Scădere
Multiplicare
Împărţire reală (rezultat real) sau împărţire întreagă (rezultat întreg)
Modulo (restul împărţirii)

Operatorii unari folosesc un singur operand. Cei binari folosesc doi operanzi.

Exemplu
```
-54  +259.65  -rate
```
O constantă fără semn este pozitivă.

Împărţirea întreagă este câtul obţinut la împărţirea a două numere întregi. Operaţia modulo reprezintă restul acestei împărţiri şi se aplică doar numerelor întregi.

Exemplu
```
6/2 → 3
7/2 → 3
6%2 → 0
7%2 → 1
```
Împărţirea în virgulă mobilă (reală) generează rezultat real, iar operanzii trebuie să fie reali.

Exemplu
```
7.0/2.0 → 3.5
```

Exemplu

Expresie	Valoare	Expresie	Valoare
3+6	9	7.0/0.0	eroare
3.4+6.9	10.3	7/0	eroare
2*3	6	7%0	eroare
8.0/-2.0	-4.0		
8/9	0		
5%2.3	Eroare		

Pentru că în expresii pot apărea şi variabile, următoarele asignări sunt valide:
```
alpha = num + 6;
num = alpha * 2;
num = num + alpha;
num = 6 % alpha;
```
În cazul instrucţiunii
```
num = num + alpha;
```
valorile lui `num` şi `alpha` sunt adunate, iar rezultatul este păstrat în `num`, înlocuind vechea valoare păstrată aici. Acest exemplu arată diferenţa dintre egaliatea matematică şi asignare. În matematică

num = num + alpha

este adevărată doar când alpha este 0. Instrucţiunea de asignare

```
num = num + alpha;
```

este validă pentru *orice* `alpha`.

Incrementarea şi decrementarea

Pe lîngă operatorii aritmetici, C++ oferă operatorii de incrementare şi decrementare.

Incrementare

`+`

Decrementare

`–`

Aceştia sunt operatori unari. Pentru operanzi întregi şi reali, efectul este de adăugare a valorii 1, respectiv de scădere a valorii 1 din operand.

Exemplu

```
int num = 8;
num++; //num va avea valoarea 9
```

Acelaşi efect poate fi obţinut prin

```
num = num + 1;
```

Operatorii `++` şi `--` pot fi atât *operatori prefix*:

```
++num;
```

cât şi *operatori postfix*:

```
num++;
```

C++ permite folosirea lui `++` şi `--` în interiorul unor expresii mai lungi:

Exemplu

```
alpha = num++ * 3;
```

Afişarea

Tipărirea rezultatelor se face în C++ folosind o variabilă specială numită `cout` împreună cu *operatorul de inserţie* (`<<`).

Exemplu

```
cout << ”Hello”;
```

Această instrucţiune afişează caracterele `Hello` la *dispozitivul standard de ieşire*, de obicei ecranul. Variabila `cout` este predefinită în C++ şi semnifică un *flux de ieşire*. Acesta poate fi imaginat ca o secvenţă nesfârşită de caractere care merg către dispozitivul de ieşire.

Operatorul de inserţie `<<` („put to”) foloseşte 2 operanzi. Cel din stânga este o *expresie flux* (stream) (de exemplu `cout` care se referă la dispozitivul standard de ieșire). În dreapta se află şirul sau expresia al cărei rezultat trebuie afişat.

Exemplu

```
cout << ”The answer is”;
cout << 3 * num;
```

De remarcat faptul că operatorul `<<` arată sensul în care circulă datele: dinspre expresie sau string înspre *stream*-ul de ieşire. Operatorul `<<` poate fi folosit de mai multe ori într-o singură instrucţiune de afişare:

Exemplu

```
cout << ”The answer is” << 3 * num;
```

şi rezultatul este acelaşi.

Exemplu

```
int i = 2;
```

```
int j = 6;
```

Instrucţiune	Ce se tipăreşte
`cout << i;`	`2`
`cout << "i = " << i;`	`i = 2`
`cout << "j: " << j << " i: " << i;`	`j:6 i:2`

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

Exemplu

```
cout << "Al \"Butch\" Jones";
```

iar pe ecranul calculatorului va apărea:

```
Al "Butch" Jones
```

În mod obişnuit, mai multe instrucţiuni de ieşire succesive afişează rezultatele continuu, pe aceeaşi linie:

```
cout << "Hi";
cout << "there";
```

generează

```
Hithere
```

Pentru a tipări pe linii separate scriem:

```
cout << "Hi" << endl;
cout << "there" << endl;
```

şi obţinem:

```
Hi
There
```

Identificatorul `endl` („end line") este un element special din C++: este un manipulator. Este suficient de ştiut acum că `endl` permite terminarea unei linii şi continuarea scrierii pe linia următoare.

2.8 Elaborarea unui program

Vom vedea cum asamblăm diferitele elemente prezentate până acum pentru a construi un program.

Un program C++ este format din clase şi funcţii, una dintre funcţii numindu-se obligatoriu `main`. Un program poate conţine declaraţii în afara oricărei funcţii. Modelul unui program este:

```
Declaraţie
...
Definiţie de clasă
Definiţie de clasă
...
Definiţie de funcţie
Definiţie de funcţie
...
```

O definiţie de funcţie se construieşte după următorul model:

```
Heading
{
 Instrucţiune
 ...
}
```

Vom da exemplu de un program cu o singură funcţie, funcţia `main`.

```
//**************************************
```

```
//Temperaturi.cpp
//Acest program calculeaza temperatura de
//la jumatatea dintre punctul de inghet si
//cel de fierbere a apei
//**************************************

#include <iostream>
using namespace std;

const float INGHET = 0.0; //Punctul de inghet al apei
const float FIERBERE = 100.0;  //Punctul de fierbere

int main()
{
  float temperaturaMedie; //Pastreaza rezultatul medierii
                          //dintre INGHET si FIERBERE

  cout << "Apa ingheata la " << INGHET << " grade";
  cout << " si fierbe la " << FIERBERE << " de grade."
   << endl;

  temperaturaMedie = INGHET + FIERBERE;
  temperaturaMedie = temperaturaMedie / 2.0;
  cout << "Temperatura medie este de ";
  cout << temperaturaMedie << " grade." << endl;

  return 0;
}
```

Programul începe cu un comentariu care explică ce face programul. Urmează

```
#include <iostream>
```

care inserează conţinutul fişierului `iostream` în programul nostru. El conţine informaţiile necesare lucrului cu stream-uri de intrare şi de ieşire, ca de exemplu `cout`.

Urmează declararea constantelor `INGHET` şi `FIERBERE`.

Restul programului este definiţia funcţiei `main`. Mai întâi heading-ul urmat de `{}`. Aceste acolade informază compilatorul că `main` este o funcţie. Corpul funcţiei cuprinde declararea variabilei `tempMedie` urmată de o serie de alte instrucţiuni executabile. Funcţia `main` returnează `0`.

De notat aranjarea liniilor de program, spaţierea şi folosirea comentariilor.

Blocuri (instrucţiuni compuse)

Corpul unei funcţii este un exemplu de bloc:

```
{
 Instrucţiune
 ...
}
```

Un bloc conţine 0 sau mai multe instrucţiuni cuprinse între `{}`. O instrucţiune se termină obligatoriu cu `;` Există şi instrucţiunea vidă:

```
;
```

Obişnuim ca instrucţiunile dintr-un bloc să le deplasăm puţin spre dreapta pentru claritatea programului. De asemenea, putem opta pentru gruparea instrucţiunilor prin separarea grupurilor cu rânduri libere.

2.9 Preprocesorul C++

Imaginaţi-vă că sunteţi în rolul compilatorului de C++ şi vi se prezintă următorul program:

```
int main()
{
  cout << "Happy Birthday" << endl;
  return 0;
}
```

Recunoaşteţi identificatorul `int` ca fiind cuvânt rezervat C++ şi `main` ca fiind numele unei funcţii care trebuie să existe în mod obligatoriu. Dar `cout` şi `endl`? Nu au fost declarate ca şi variabile sau constante şi nu sunt cuvinte rezervate. Veţi afişa următorul mesaj de eroare:

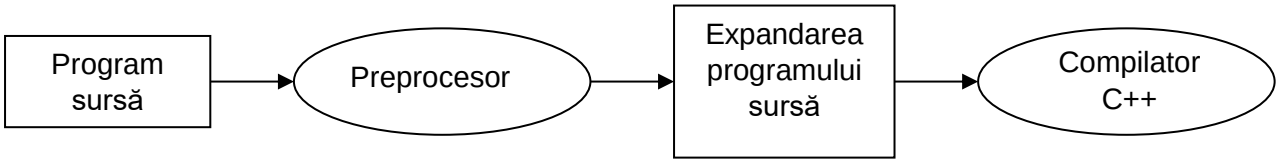
```
In function 'int main()':
Line 3: 'cout' undeclared
Line 3: 'endl' undeclared
```

Pentru a corecta eroarea, programatorul trebuie să insereze la începutul programului linia

```
#include <iostream>
```

Ea spune că întreg conţinutul fişierului `iostream` va fi inserat în program. El conţine declaraţiile lui `cout` şi `endl`.

Instrucţiunea `#include` nu este interpretata de compilatorul C++, ci de un program numit *preprocesor*. El acţionează ca un filtru şi precede faza de compilare. O linie care începe cu # se numeşte *directivă de preprocesare*.



La preprocesare sunt posibile următoarele acţiuni:

- Includerea altor fişiere în fişierul care urmează să fie compilat
- Definirea constantelor simbolice şi a macrourilor (macrourile sunt specifice limbajului C şi nu le vom prezenta)
- Compilarea condiţională

Directiva de preprocesare `#include`

Această directivă de preprocesare produce includerea în codul sursă a unei copii a fişierului specificat. Ea are două forme.

Prima formă este cea în care se folosesc semnele `< >` ca în directiva

```
#include <iostream>
```

Acestea indică faptul că de referim la un fişier din biblioteca standard iar preprocesorul caută acel fişier în *directorul standard* pentru includere.

În varianta

```
#include "consum.dat"
```

Ghilimelele arată că fişierul `consum.dat` se găseşte în acelaşi director cu fişierul sursă.

Directiva de preprocesare `#define`: constante simbolice

Prin directiva de preprocesare `#define` se pot defini *constante simbolice*.

Exemplu

```
#define PI 3.14159
```

Proprocesorul va înlocui toate apariţiile lui `PI` din textul care urmează declaraţiei cu valoarea `3.14159`. Dacă dorim să modificăm valoarea constantei, aceasta poate fi modificată prin înlocuirea valorii din directiva de preprocesare. Diferenţele dintre constantele definite prin cuvântul cheie `const` şi constantele definite prin directive de preprocesare sunt că primele au un tip de dată şi că sunt vizibile în faza de debugging. Odată înlocuită o constantă de către preprocesor, doar valoarea sa va mai fi vizibilă.

Directiva de preprocesare

```
#define DEBUG
```

în care lipseşte valoarea asociată constantei şterge orice apariţie a identificatorului `DEBUG` din fişierul sursă. Identificatorul rămâne definit şi poate fi testat prin directivele `#if defined` sau `#ifdef`.

Există 5 constante simbolice predefinite:

Constanta simbolică	Descrierea
`__LINE__`	Numărul liniei curente din fişierul sursă
`__FILE__`	Numele fişierului sursă
`__DATE__`	Data la care a fost compilat fişierul sursă
`__TIME__`	Ora la care a fost compilat fişierul sursă
`__STDC__`	Constanta întreagă 1

Directiva de preprocesare `#undef` aplicată unei constante simbolice sau unui macro definite prin `#define` realizează ştergerea definiţiei.

Compilarea condiţională

Compilările condiţionate permit programatorilor să controleze execuţia directivelor de preprocesare şi a compilării programului sursă. Compilările condiţionale pot fi realizate prin folosirea directivelor de preprocesare `#ifndef` şi `#ifdef`.

Codul

```
#ifndef NULL
  #define NULL 0
#endif
```

verifică dacă `NULL` a fost deja definită în program, iar daca daca nu, o defineşte.

Compilarea condiţională se foloseşte de regulă pentru debugging. Se folosesc instrucţiuni de afişare care tipăresc valorile unor variabile şi care confirmă fluxul corect al programului. Aceste afişări care nu mai sunt necesare dupa ce programul a fost corectat sunt, de regulă, încadrate de directive condiţionale de preprocesare.

Exemplu

```
#ifdef DEBUG
  cout << "Variabila x = " << x << endl;
#endif
```

3. Expresii aritmetice, apeluri de funcţii şi ieşiri

În acest capitol vom discuta în detaliu despre scrierea expresiilor aritmetice şi despre formatarea ieşirilor. Vom vedea, de asememea, cum putem folosi bibliotecile de funcţii – funcţii scrise anterior şi care fac parte din orice sistem C++.

3.1 Expresii aritmetice

Vom studia modul în care pot fi scrise expresii care conţin mai mult de un operator şi care folosesc operanzi care au diferite tipuri de dată.

Reguli de precedenţă

Expresiile aritmetice sunt formate din constante, variabile, operatori şi paranteze. Ordinea în care sunt realizate operaţiile este stabilită conform *regulilor de precedenţă*. Cele 5 operaţii aritmetice de bază şi parantezele sunt ordonate în felul următor:

Cea mai înaltă precedenţă

```
( )
*  /  %
+  -
```

Cea mai scăzută precedenţă

Exemplu

```
tempMedie = INGHET + FIERBERE / 2.0
```

În acest exemplu, mai întâi se efectuează împărţirea `FIERBERE / 2.0`, iar apoi rezultatul este adunat cu `INGHET`.

Folosind parantezele, se poate schimba ordinea de evaluare a expresiei.

Exemplu

```
tempMedie = (INGHET + FIERBERE) / 2.0
```

Mai întâi sunt evaluate subexpresiile din paranteze, iar apoi urmăm precedenţa operatorilor.

Atunci când apar în aceeaşi expresie mai mulţi operatori cu aceeaşi precedenţă, *ordinea de grupare* sau *asociativitatea* este de la stânga la dreapta. Expresia

```
i1 - i2 + i3
```

este echivalentă cu

```
(i1 - i2) + i3
```

şi nu cu

```
i1 - (i2 + i3).
```

Un alt exemplu:

```
(f1 + f2) / f1 * 3.0
```

Se evaluează mai întâi parantezele, apoi rezultatul se împarte la `f1`, iar în final se realizează multiplicarea cu `3.0`.

Conversii implicite şi explicite

Valorile întregi şi cele reale sunt stocate în mod diferit în memorie. Modelul din memorie al biţilor care reprezintă constanta 2 nu arată ca modelul din memorie al biţilor care reprezintă constanta 2.0. Problema este ce se întâmplă când folosim un întreg şi un real în aceeaşi expresie sau într-o asignare.

Instrucţiuni de asignare

Dacă facem declaraţiile

```
int i;
float m;
```

atunci variabila `i` poate păstra doar valori întregi, iar variabila `m` doar valori în virgulă mobilă. Instrucţiunea de asignare

```
m = 12;
```

pare că încarcă valoarea întreagă `12` în variabila `m`. Însă calculatorul refuză să stocheze altceva decât valori de tip float în variabila `m`. Compilatorul inserează, în acest caz, două noi instrucţiuni care mai întâi convertesc valoarea `12` în `12.0` şi apoi stochează `12.0` în variabila `m`. Această transformare automată a unei valori dintr-un tip de dată în alt tip de dată se numeşte *conversie implicită* (*type coercion*, *forţare de tip*).

Instrucţiunea

```
i = 4.8;
```

provoacă de asemenea o forţare de tip. Când un număr real este asignat unei variabile întregi, partea fracţionară este trunchiată. Ca rezultat, variabilei `i` i se asignează valoarea `4`.

Adeseori, în conversiile implicite sunt implicate expresii întregi. Păstrarea rezultatului unei expresii cu rezultat de tip întreg într-o variabilă reală (float) nu provoacă pierderi de informaţie. Stocarea rezultatului unei expresii reale într-o variabilă întreagă conduce la trunchierea părţii fracţionare.

Pentru a clarifica programul şi pentru a evita erorile putem folosi *conversia explicită* (*type casting*). În C++ o operaţie de cast constă din precizarea tipului de dată pe care dorim să îl aibă rezultatul urmat, între paranteze, de expresia pe care dorim să o convertim.

Exemplu

```
m = float(3 * i + 2);
i = int(5.2 / m - altFloat);
```

Următoarele două instrucţiuni produc rezultate identice:

```
i = m + 8.2;
i = int(m + 8.2);
```

Diferenţa constă în claritatea programului şi eliminarea erorilor sau avertismentelor de la compilare.

Scrierea expresiilor aritmetice

Până acum am vorbit doar despre combinarea diferitelor tipuri de dată în operaţia de asignare. Este, de asemenea, posibilă combinarea datelor de diferite tipuri în expresii.

Exemplu

```
i * m
4.8 + i - 3
```

Întotdeauna, când într-o expresie apar variabile de tip întreg şi variabile de tip real apar conversii implicite după cum urmează:

1. Întregul este forţat temporar la o valoare reală
2. Se efectuează operaţia
3. Rezultatul este real

Să analizăm a doua instrucţiune din exemplul anterior, în care varianta `i` conţine valoarea `2`. Operatorul + are operanzi de tipuri diferite, de aceea valoarea lui

`i` este forţată la `2.0`. Această conversie este temporară şi nu afectează valoarea `2` stocată în `i`. Se efectuează adunarea, iar rezultatul este `6.8`. Scăderea are de asemenea, doi operanzi de tipuri diferite: `6.8` şi `3`. Valoarea `3` este forţată la `3.0`, se execută scăderea şi rezultatul este numărul real `3.8`.

În interiorul expresiilor se pot folosi conversiile explicite de tip pentru a reduce riscul de apariţie al erorilor şi pentru claritate:

Exemplu

```
float(i) * m
4.8 + float(i - 3)
```

Conversiile explicite de tip nu se fac, însă, doar pentru claritate.

Să calculăm media mai multor numere. Suma lor este stocată în `sum` si numărul lor este stocat în `count`. Avem următoarele declaraţii:

```
int sum;
int count;
float average;
```

Valoarea medie se găseşte astfel:

```
average = sum / count; //eroare de logica
```

Dacă `sum` este `60` şi `count` este `80`, rezultatul va fi `0.0`. De ce? Expresia din dreapta operatorului = conţine doi operanzi întregi. În această situaţie, împărţirea este de tip întreg, deci rezultatul este `0`. Apoi, rezultatul este convertit la valoarea reală `0.0` înainte de a fi stocat în average. Pentru a corecta rezultatul, modificăm ultima instrucţiune astfel:

```
average = float(sum) / float(count);
```

Împărţirea va fi reală, iar rezultatul va fi `0.75`.

Până acum ne-am referit doar la tipurile `int` şi `float`. Conversiile se pot aplica şi valorilor `char`, `short` sau `double`.

3.2 Apeluri de funcţii şi biblioteci de funcţii

Apeluri de funcţii

În capitolul anterior am văzut un program care conţinea trei funcţii: `main`, `Patrat` şi `Cub`. Toate trei returnau cate o valoare. `Patrat` si `Cub` returnează valori către funcţiile apelante, iar `main` întoarce o valoare către sistemul de operare.

În instrucţiunea

```
cout << " si cubul lui 27 este " << Cub(27) << endl;
```

secvenţa `Cub(27)` este un *apel de funcţie* sau *invocare de funcţie*. Calculatorul opreşte temporar execuţia funcţiei `main` şi porneşte funcţia `Cub`. Când `Cub` îşi încheie execuţia tuturor instrucţiunilor, calculatorul revine la `main` din punctul în care a fost oprită.

În apelul funcţiei `Cub`, numărul `27` se numeşte *parametru* sau *argument*. Parametrii creează posibilitatea unei funcţii să lucreze cu diferite valori. Astfel, putem scrie

```
cout << Cub(4);
cout << Cub(16);
```

Modelul sintactic al unui apel de funcţie este

```
NumeleFuncţiei(ListăDeParametri)
```

Lista de parametri este mecanismul prin care funcţiile comunică una cu cealaltă.

Unele funcţii, de exemplu `Patrat` sau `Cub`, au un singur parametru în lista de parametri. Alte funcţii, de exemplu `main`, nu au niciun parametru în listă. Există funcţii cu doi, trei sau mai mulţi parametri în listă, separaţi prin virgulă.

Funcţiile care întorc o valoare pot fi utilizate în expresii în acelaşi fel în care se folosesc constantele sau variabilele. Valoarea calculată de funcţie înlocuieşte apelul funcţiei în expresie.

Exemplu

```
i = Cub(2) * 10; //i va pastra valoarea 80
```

Într-o expresie, un apel de funcţie are cea mai mare precedenţă.

Consideraţii referitoare la apelurile de funcţii:

3.2 Apelurile de funcţii sunt folosite în expresii. Nu apar ca instrucţiuni de sine-stătătoare;

3.3 Funcţia calculează o valoare (un rezultat) care poate fi folosit apoi într-o expresie;

3.4 Funcţia întoarce exact un rezultat.

Funcţia `Cub` aşteaptă să i se dea, să i se *transmită* un parametru de tip int. Dacă primeşte un parametru de tip float, compilatorul realizează o forţare implicită a tipului de dată.

Exemplu

`Cub(6.9)` calculează 6^3 şi nu 6.9^3

Până acum am folosit doar constante literale ca şi parametri ai funcţiei `Cub`. Aceştia, însă, pot fi şi variabile sau constante simbolice şi, în general, expresii având un tip potrivit cu cel al parametrului.

În instrucţiunea

```
alfa = Cub(int1 * int1 + int2 * int2);
```

expresia care reprezintă lista de parametri este evaluată prima, şi numai după aceea rezultatul este transmis funcţiei. De exemplu, dacă `int1` conţine `3` şi `int2` conţine `5`, atunci parametrul transmis funcţiei `Cub` este `34`.

O expresie din lista de parametri a funcţiei poate include şi apeluri de funcţii. Putem rescrie apelul precedent folosind funcţia `Patrat`:

```
alfa = Cub(Patrat(int1) + Patrat(int2));
```

Biblioteci de funcţii

Anumite calcule, cum ar fi rădăcina pătrată, sunt foarte des folosite în programe. De aceea, limbajul C++ include o *bibliotecă standard* care este o colecţie de funcţii prescrise care realizează anumite operaţii.

Fişierul header	Funcţia	Tipul parametrilor	Tipul rezultatului	Rezultatul
`<stdlib.h>`	`abs(i)`	`int`	`int`	Valoarea absolută a lui `i`
`<math.h>`	`cos(x)`	`double`	`double`	Cosinusul lui `x` (`x` în radiani)
`<math.h>`	`fabs(x)`	`double`	`double`	Valoarea absolută a lui `x`
`<math.h>`	`pow(x, y)`	`double`	`double`	x^y. Dacă `x=0.0`, `y` trebuie să fie pozitiv. Dacă `x<0.0`, `y` trebuie să fie întreg

Pentru a folosi o bibliotecă de funcţii, trebuie să plasăm directiva `#include` la începutul programului, specificând fişierul header dorit.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
 double alfa = 1.2;
 double beta = -2.5;
 alfa = sqrt(7.3 + fabs(beta));
 cout << alfa << endl;
 return 0;
}
```

Funcţii void

Până acum am discutat despre funcţii care întorc o valoare. Dacă studiem următoarea definiţie de funcţie, observăm că ea începe cu cuvântul `void` în loc de `int` sau `double`:

```
void Calcul(...)
{
     ...
}
```

Acesta este un exemplu de funcţie care nu întoarce nicio valoare către funcţia apelantă. Ea realizează doar o acţiune şi apoi revine. Acestea sunt *funcţii care nu întorc nicio valoare* sau *funcţii void*. În multe limbaje de programare aceste funcţii se mai numesc şi *proceduri*.

Spre deosebire de funcţiile care întorc o valoare, acestea se apelează într-o singură instrucţiune de sine-stătătoare:

Exemplu

```
Calcul(plataPeOra, ore);
```

Din punctul de vedere al apelantului, aceste funcţii arată ca o comandă:

```
ExecutaAsta(x, y, z);
FaAsta();
```

3.3 Formatarea ieşirilor

Formatarea ieşirilor unui program înseamnă modul în care se poate controla apariţia pe ecran sau la imprimantă a rezultatelor programelor. Dacă variabilele `i`, `j` şi `k` au valorile `15`, `2` şi `6`, atunci instrucţiunea

```
cout << ”Rezultate: ” << i << j << k;
```

produce şirul de caractere

```
Rezultate: 1526
```

Fără spaţii între numere, rezultatul este dificil de interpretat.

Spaţierea verticală

Am văzut deja că pentru aceasta se foloseşte manipulatorul `endl`. O secvenţă de instrucţiuni de instrucţiuni de ieşire continuă să scrie pe linia curentă până când `endl` termină linia.

Să vedem ce afişează următoarele instrucţiuni:

```
cout << ”Formatarea ” << endl;
cout << endl;
cout << ”iesirilor. ” << endl;
```

Prima instrucţiune produce afişarea pe ecran a şirului de caractere `Formatarea`, iar `endl` provoacă trecerea pe rândul următor. Următoarea instrucţiune produce o nouă trecere pe rândul următor a cursorului. A treia instrucţiune tipăreşte cuvântul `iesirilor` şi termină linia. Rezultatul este:

```
Formatarea

iesirilor.
```

Instrucţiunile de mai sus sunt echivalente cu:

```
cout << ”Formatarea ” << endl << endl;
cout << ”iesirilor. ” << endl;
```

sau cu

```
cout << ”Formatarea ” << endl << endl << ”iesirilor. ”
     << endl;
```

sau cu

```
cout << ”Formatarea ” << endl << endl
     << ”iesirilor. ” << endl;
```

Ultimul exemplu arată că o instrucţiune C++ poate fi scrisă pe mai multe linii. Compilatorul urmăreşte apariţia semnului `;` şi nu sfârşitul fizic al liniei.

Inserarea spaţiilor într-o linie

Pentru a controla spaţierea orizontală se obişnuieşte introducerea unor spaţii suplimentare. Pentru a preveni afişarea numerelor 15, 2 şi 6 în forma

```
Rezultate: 1526
```

putem tipări câte un singur caracter (constantă tip `char`) între numere:

```
cout << ”Rezultate: ” << i << ‘ ‘ << j << ‘ ‘ << k;
```

Această instrucţiune va afişa:

```
Rezultate: 15 2 6
```

Dacă dorim afişarea unor spaţii mai mari, putem opta pentru folosirea şirurilotr constante care conţin spaţii:

```
cout << ”Rezultate: ” << i << ”    ” << j << ”   ” << k;
```

Această instrucţiune afişează:

```
Rezultate: 15   2  6
```

Pentru a produce ieşirea:

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

putem folosi următoarele instrucţiuni:

```
cout << ” * * * * * ” << endl;
cout << ”* * * * *” << endl;
```

Pentru ca spaţiile să fie tipărite pe ecran, ele *trebuie* incluse între apostrafe sau ghilimele. Remarcăm că instrucţiunea:

```
cout << ‘*’ <<       ‘*’;
```

are următorul efect:

```
**
```

pentru că spaţiile care sunt în afara apostroafelor nu sunt luare în considerare la tipărire.

Manipulatori

Am folosit de multe ori până acum manipulatorul `endl` pentru a termina o linie afişată pe ecran. În C++, un manipulator este o entitate care se comportă ca o funcţie, dar se foloseşte ca o dată. Ca funcţie el produce o acţiune, iar ca dată poate fi plasat într-o serie de operaţii de inserţie:

```
cout << i << endl << m;
```

Manipulatorii se folosesc numai în instrucţiuni de intrare sau de ieşire. Bibliotecile standard C++ oferă o serie întreagă de manipulatori, iar noi vom studia trei dintre ei: `endl`, `setw` şi `setprecision`.

Pentru a îl folosi pe `endl` trebuie să includem fişierul header `iostream`. Pentru ceilalţi manipulatori trebuie să includem, în plus, şi fişierul header `iomanip`.

Exemplu

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

int main()
{
 int i = 2;
 cout << setw(5) << i << endl;
}
```

Manipulatorul `setw` (*set width*) permite stabilirea numărului de coloane folosite pentru următoarea afişare. El se aplică doar numerelor şi string-urilor, nu şi datelor de tip `char`. Parametrul lui `setw` este o expresie întreagă numită *specificaţie a dimensiunii câmpului*. Numărul de coloane stabilite pentru afişare se numeşte *câmp*. Data afişată va fi aliniată la dreapta, iar poziţiile câmpului rămase astfel libere vor fi umplute cu spaţii.

Exemplu

```
int a = 33;
int b = 7132;

cout << setw(4) << a
     << setw(5) << b
     << setw(7) << "Salut";
```

`␣␣33␣7132␣␣Salut`

câmpurile au fost completate cu spaţii; acestea au fost marcate prin `␣`

```
cout << setw(1) << a
     << setw(4) << b
     << setw(5) << "Salut";
```

`337132Salut`

câmpurile au fost mărite automat la dimensiunea datei afişate

Stabilirea dimensiunii câmpului afectează doar următorul element afişat. După aceea, dimensiunea este resetată la 0, ceea ce înseamnă că dimensiunea va fi extinsă la atâtea coloane câte sunt necesare.

Exemplu

```
int a = 33;
int b = 7132;

cout << "Salut"
```

```
        << setw(5)
        << a << b;
```

afişează

```
Salut   337132
```

La specificarea dimensiunii câmpului pentru numerele reale, trebuie să ţinem cont că punctul zecimal ocupă şi el o poziţie. Valoarea `4.85` ocupă 4 coloane, nu 3.

Exemplu

```
float x = 4.85;

cout << setw(4) << x << endl       4.85
     << setw(6) << x << endl         4.85
     << setw(3) << x << endl;      4.85
```

Există câteva observaţii care trebuie făcute în legătură cu afişarea numerelor reale.

1. Numerele foarte mari sunt afişate implicit în formă ştiinţifică.
   Exemplu
   `123456789.5` este afişat `1.23457+008`
2. Dacă numărul afişat este întreg, nu se va tipări ca număr real.
   Exemplu
   `95.0` este afişat `95`

Pentru a evita aceste formate implicite, înaintea afişării oricărui număr real trebuie să includem următoarele două instrucţiuni:

```
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::showpoint);
```

Aceste instrucţiuni folosesc noţiuni mai avansate de C++ care nu pot fi explicate acum în detaliu. `setf` este o funcţie `void` asociată stream-ului `cout` (punctul dintre cout şi `setf` este strict necesar). Prima instrucţiune ne asigură că numerele reale vor fi tipărite în formă zecimală şi nu ştiinţifică. Cea de-a doua instrucţiune specifică faptul că punctul zecimal va fi tipărit întotdeauna, chiar şi pentru numere întregi. Momentan vom utiliza aceste instrucţiuni în această formă. Aceste setări rămân valabile până la o nouă modificare a lor.

Adeseori dorim să controlăm numărul de zecimale afişate, de exemplu pe 12.8 să îl tipărim `12.80` sau pe 16.38753 să îl tipărim `16.39`. Pentru aceasta trebuie să folosim manipulatorul `setprecision`.

Exemplu

```
cout << setprecision(3) << x;
```

Parametrul lui setprecision stabileşte numărul de zecimale cu care va fi tipărit un număr real.

Exemplu

```
float x = 310.0;
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::showpoint);
cout << setw(10)                                  ␣␣␣␣310.00
     << setprecision(2) << x;
cout << setw(7)                                   310.00000
     << setprecision(5) << x;
x=4.827;
cout << setw(6)                                   ␣␣4.83
     << setprecision(2) << x;
```

4. Intrările în program. Scrierea aplicaţiilor

Un program are nevoie de date pentru a opera. Până acum am scris programe în care valorile datelor se găsesc chiar în program, în constante simbolice sau literale. Dacă dorim să modificăm o dată, trebuie să facem o mică modificare în program, să îl recompilăm şi să îl executăm din nou. În acest capitol vom vedea cum putem introduce date în program chiar în timpul rulării lui.

Odată ce am aflat cum să introducem date în program, să procesăm datele şi să afişăm rezultatele, putem să ne gândim la scrierea unor programe mai complicate. Pentru aceasta avem nevoie de o abordare ceva mai organizată. În finalul capitolului vom discuta despre descompunerea în module şi despre programarea orientată pe obiecte.

4.1 Transmiterea datelor către programe

Unul dintre marile avantaje ale calculatorului este că un program poate fi folosit cu diverse seturi de date. Pentru aceasta trebuie să separăm datele de program până în momentul execuţiei. Anumite instrucţiuni din program copiază valori în variabilele din program. După stocarea acestor valori în variabile, programul poate să le folosească în calcule.

Procesul de plasare a unor valori dintr-o mulţime de date în variabile din program se numeşte *intrare* (*input*). Într-o terminologie mai largă, calculatorul se spune că ***citeşte*** date din exterior în variabile. Datele pot proveni dintr-un fişier, de la tastatură etc. Dispozitivul standard de intrare este tastatura.

Stream-uri de intrare şi operatorul de extracţie >>

În C++, conceptul de *stream* este esenţial. Putem gândi un *stream de ieşire* ca pe o secvenţă infinită de date care circulă dinspre program înspre un dispozitiv de ieşire. Un *stream de intrare* este o secvenţă infinită de caractere care porneşte de la un dispozitiv de intrare şi este dirijată către program.

Fişierul header `iostream` conţine, printre altele, definiţiile a două tipuri de date: `istream` şi `ostream`. Aceste tipuri de date reprezintă stream-uri de intrare şi stream-uri de ieşire. Acest fişier header mai conţine două declaraţii care arată aproximativ astfel:

```
istream cin;
ostream cout;
```

Cele două declaraţii arată că `cin` este un obiect de tip `istream` şi `cout` este un obiect de tip `ostream`. În mod explicit, cin este asociat cu tastatura, iar cout cu display-ul.

Am văzut că trimiterea unor valori către `cout` se face folosind *operatorul de inserţie* `<<`:

```
cout << 3 * pret;
```

Similar, putem citi date din `cin` folosind *operatorul de extracţie* `>>`:

```
cin >> cost;
```

Operatorul de extracţie `>>` are doi operanzi. În stânga se găseşte un stream, în cel mai simplu caz `cin`, iar în dreapta se găseşte o variabilă având un tip predefinit (`char`, `int`, `float` etc.).

Putem folosi operatorul de mai multe ori într-o instrucţiune:

```
cin >> lungime >> latime;
```

fiind echivalentă cu:

```
cin >> lungime;
cin >> latime;
```

Trebuie să fim atenţi atunci când folosim cei doi operatori, pentru că `cin` poate fi folosit doar în combinaţie cu `>>`, iar `cout` doar cu `<<`.

Dacă într-o instrucţiune de afişare putem folosi constante, variabile şi chiar expresii foarte complicate, într-o instrucţiune de intrare nu putem folosi decât nume de variabile. Aceasta pentru că o instrucţiune de intrare trebuie să precizeze clar unde se stochează valoarea unei date de intrare. Doar numele de variabile referă locaţii de memorie ale căror valori pot fi modificate în timpul execuţiei unui program.

Atunci când introducem o dată de la tastatură, trebuie să ne asigurăm că tipul introdus şi cel aşteptat de program se potrivesc. Un număr întreg este forţat automat la un număr real. Operaţia inversă, însă, poate conduce la rezultate eronate.

Atunci când extrage valori dintr-un stream, operatorul `>>` ignoră orice spaţiu de la început. De asemenea, ignoră caracterul care marchează sfârşitul liniei. Apoi, operatorul `>>` procedează la extragerea valorilor din stream-ul de intrare. Dacă data aşteptată 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 potriveşte ca tip de dată, cum ar fi un spaţiu.

Exemplu

```
int i, j, k;
char ch;
float x;
```

Instrucţiunea	**Data**	**Conţinutul variabilei după intrare**
`cin >> i;`	`32`	`i = 32`

`cin >> i >> ch >> x;`	`25 A 16.9`	`i = 25`
		`ch = 'A'`
		`x = 16.9`

`cin >> i >> j >> x;`	`12 8`	`i = 12`
		`j = 8`
		Programul aşteaptă al treilea număr

`cin >> i >> x;`	`46 32.4 15`	`i = 46`
		`x = 32.4`
		15 este păstrat pentru o intrare ulterioară

Caracterul *newline*

Fiecare linie are un caracter invizibil care marchează sfârşitul său – caracterul *newline*. Pentru a determina următoarea valoare de intrare, operatorul `>>` trece peste caracterul *newline*, în cazul în care acesta există.

Atunci când utilizăm tastatura, caracterul *newline* este introdus prin apăsarea lui Return sau Enter. Programul poate genera un *newline* folosind manipulatorul `endl` într-o instrucţiune de ieşire. În C++ putem să ne referim la caracterul *newline* folosind simbolurile `\n`. Deşi `\n` constă din două caractere, el se referă la unul singur – caracterul *newline*. Aşa cum putem păstra litera `A` în variabila `ch` de tip `char` prin instrucţiunea

```
char ch = 'A';
```

tot aşa putem păstra caracterul *newline* într-o variabilă:

```
ch = '\n';
```

Exemplu

Când avem o secvenţă de citiri, putem introduce valorile în mai multe feluri:

`cin >> i;`	`25 A 16.9\n`	`25\n`	`25A16.9\n`
`cin >> ch;`		`A\n`	Citirea se opreşte când tipul de dată nu mai corespunde
`cin >> x;`		`16.9\n`	

După citire, variabilele vor avea următoarele valori:

```
i = 25
ch = 'A'
x = 16.9
```

Citirea datelor de tip caracter folosind instrucţiunea `get`

Am văzut că operatorul `>>` ignoră orice spaţiu apărut în stream-ul de intrare. Să presupunem că `ch1` şi `ch2` sunt variabile de tip `char` şi că programul execută instrucţiunea

```
cin >> ch1 >> ch2;
```

Dacă stream-ul de intrare este

```
R 1
```

Atunci operatorul de extracţie va stoca `R` în `ch1`, ignoră spaţiul şi apoi păstrează `1` în `ch2`.

Ce se întâmplă, însă, dacă am fi dorit, de fapt, să introducem trei caractere: `R`, spaţiu şi `1`? Cu operatorul de extracţie acest lucru nu este posibil.

Vom folosi funcţia `get` care încarcă următorul caracter fără a ignora spaţiile. Acest apel de funcţie arată astfel:

```
cin.get(ch);
```

Specificăm numele `istream`-ului, adică `cin`, apoi punem semnul . (punct) urmat de numele funcţiei şi lista ei de parametri. Apelul funcţiei `get` foloseşte sintaxa apelului funcţiilor `void` şi nu a celor care întorc o valoare. Acest apel de funcţie este, deci, o instrucţiune de sine stătătoare. Parametrul funcţiei `get` trebuie să fie o variabilă. Acesta este un exemplu prin care se apelează o funcţie din clasa `istream` – funcţia `get` – pentru un obiect al acestei clase – obiectul `cin`.

Putem folosi următoarele trei apeluri ale lui `get`:

```
cin.get(ch1);
cin.get(ch2);
cin.get(ch3);
```

sau

```
cin >> ch1;
cin.get(ch2);
cin >> ch3;
```

Dacă stream-ul de intrare este tot

```
R 1
```

atunci variabilele `ch1`, `ch2` şi `ch3` vor stoca

```
ch1 = 'R';
ch2 = ' ';
ch3 = '1';
```

Exemplu

```
cin >> ch1;    A B\n          ch1 = 'A';
cin >> ch2;    CD\n           ch2 = 'B';
cin >> ch3;                   ch3 = 'C';

cin.get(ch1); A B\n           ch1 = 'A';
cin.get(ch2); CD\n            ch2 = ' ';
cin.get(ch3);                 ch3 = 'B';

cin >> ch1;    A B\n          ch1 = 'A';
cin >> ch2;    CD\n           ch2 = 'B';
cin.get(ch3);                 ch3 = '\n';
```

Ignorarea caracterelor folosind funcţia `ignore`

Funcţia `ignore` este asociată cu tipul de dată `istream`, fiind o funcţie definită în această clasa `istream`. Este folosită pentru a „sări" peste caractere din stream-ul de intrare. Este o funcţie cu doi parametri, iar un exemplu de apel este următorul:

```
cin.ignore(200, '\n');
```

Primul parametru este o expresie `int`, iar al doilea una `char`.

Acest apel al funcţiei spune calculatorului să ignore următoarele 200 de caractere de intrare sau să sară până întâlneşte caracterul *newline*, în funcţie de care dintre ele apare prima.

Exemplu

```
cin >> i >> j;           957 34 1235\n     i = 957;
cin.ignore(100, '\n');   128 96\n          j = 34;
cin >> k;                                  k = 128;

cin >> ch;               A 22 B 16 C 19\n  ch = 'A';
cin.ignore(100, 'B');                      i = 16;
cin >> i;

cin.ignore(2, '\n');     ABCDEF\n          ch = 'C';
cin >> ch;
```

Intrări şi ieşiri interactive

Un *program interactiv* este unul prin care utilizatorul comunică direct cu calculatorul. Multe dintre programele scrise de noi vor fi interactive, dând şi o anumită etichetă codului.

Atunci când dorim să cerem utilizatorului să introducă o dată în program, este util să îi afişăm înainte un mesaj prin care să îi explicăm ce trebuie să introducă. De asemenea, programul ar trebui să tipărească toate datele introduse pentru ca utilizatorul să se poată verifica. Aceasta se numeşte *tipărire în ecou*.

Programul de mai jos arată cum poate fi scris un cod interactiv..

Exemplu

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

int main()
{
```

```
        int codNumeric;
        int cantitate;
        double pretUnitar;
        double pretTotal;

        cout << "Introduceti codul numeric al produsului
    comandat:" << endl;
        cin >> codNumeric;

        cout << "Introduceti cantitatea:" << endl;
        cin >> cantitate;

        cout << "Introduceti pretul unitar pentru acest
    produs:" << endl;
        cin >> pretUnitar;

        pretTotal = cantitate * pretUnitar;

        cout.setf(ios::fixed, ios::floatfield);

        cout << "Produsul " << codNumeric
             << ", cantitatea " << cantitate
             << ", pretul unitar " << setprecision(2) <<
    pretUnitar
             << " lei " << endl;
        cout << "Total: " << pretTotal << endl;

        return 0;
    }
```

În timpul rulării acestui program se va tipări câte un text care va arăta utilizatorului care este următoarea valoare aşteptată.

Exemplu

```
Introduceti codul numeric al produsului comandat
4671
Introduceti cantitatea:
10
Introduceti pretul unitar pentru acest produs:
272.55
Produsul 4671, cantitatea 10, pretul unitar 272.55 lei
Total: 2725.50
```

Volumul informaţiei afişate depinde de cel care va folosi programul. Dacă el este destinat unor persoane nefamiliarizate cu calculatorul, mesajele vor fi mai detaliate. Dacă programul este folosit frecvent de aceeaşi persoană, putem da mesaje mai scurte sau se pot introduce mai multe valori pe aceeaşi linie..

Intrări şi ieşiri neinteractive

Deşi tindem să dăm exemple de programe interactive, multe dintre programele folosite în viaţa reală sunt neinteractive. Acestea sunt programe care prelucrează cantităţi mari de date, greu de introdus interactiv fără erori. Pentru acest tip de programe, datele se păstrează în fişiere de date pregătite anterior. Aceasta permite verificare şi corectarea datelor înainte de rularea programului.

4.2 Intrări şi ieşiri din fişiere

Fişierul este o zonă pe un suport de stocare, de exemplu hard disc, desemnată printr-un nume, care păstrează o colecţie de date, de exemplu codul programului scris cu un editor.

Un program poate citi datele dintr-un fişier în aceeaşi manieră în care le citeşte de la tastatură. Se poate scrie în fişier la fel cum se scrie pe ecran.

Atunci când programul ajunge să lucreze cu cantităţi mari de date, este de preferat să folosim fişierele. Acestea pot fi scrise cu un editor de texte, corectate şi salvate pe disc. Totodată, nu suntem obligaţi să introducem toate datele dintr-o singură dată.

Modul de utilizare a fişierelor

Pentru a folosi un fişier în operaţiile I/O trebuie să parcurgem patru paşi.

1. Cerem preprocesorului să includă fişierul header `fstream`;
2. Folosim instrucţiuni de declarare pentru a declara stream-urile;
3. Pregătim fişierele pentru citire şi scriere prin instrucţiunea `open`;
4. Specificăm numele stream-ului în fiecare instrucţiune de citire sau de scriere.

Includerea fişierului header `fstream`

Vom modifica programul care măsoară consumul unui autoturism la 100 km pentru ca să citească datele dintr-un fişier.

Exemplu

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

int main()
{
     float cant1;
     float cant2;
     float cant3;
     float cant4;
     float indicatiePlecare;
     float indicatieSosire;
     float litriPerKm;

     ifstream inConsum;
     ofstream outConsum;

     inConsum.open("incons.dat");
     outConsum.open("outcons.dat");

     inConsum >> cant1 >> cant2 >> cant3 >> cant4
              >> indicatiePlecare >> indicatieSosire;

     litriPerKm = (cant1 + cant2 + cant3 + cant4)*100.0/
                  (indicatieSosire - indicatiePlecare);

     outConsum << "Consumul este " << litriPerKm
               << " litri per km." << endl;
     return 0;
```

```
}
```

Mai întâi folosim directiva de preprocesare

```
#include <fstream>
```

Prin acest header se definesc două noi tipuri de date, `ifstream` şi `ofstream`. Cele două tipuri de dată reprezintă, primul, un stream de caractere provenind de la un fişier, iar al doilea un stream de caractere care conduce către un fişier. Toate operaţiile valabile pentru un `istream`: `>>`, `get` sau `ignore` sunt valabile şi pentru tipul `ifstream`. Operaţiile folosite pentru un `ostream`: `<<`, `endl`, `setw`, `setprecision` se pot folosi şi pentru un `ofstream`.

Declararea stream-urilor pentru fişiere

Obiectele de tip stream se declară la fel ca orice variabilă

Exemplu

```
int unInt;
float unFloat;
ifstream unFisier;
ofstream altFisier;
```

Obiectele `cin` şi `cout` nu trebuie declarate în fiecare program pentru că ele sunt declarate în fişierul `iostream`, citirile de la tastatură şi scrierile pe ecran fiind operaţii frecvente. Spre deosebire de ele, stream-urile pentru lucrul cu fişiere trebuie declarate în program pentru că fiecare aplicaţie foloseşte propriile fişiere de date. Pentru programul nostru declarăm două stream-uri:

```
ifstream inConsum;
ofstream outConsum;
```

De notat că `ifstream` se utilizează doar pentru fişiere de intrare, iar `ofstream` doar pentru fişiere de ieşire. Prin intermediul unui obiect `ifstream` se pot face doar citiri, iar prin intermediul unui obiect `ofstream` se pot face doar scrieri.

Deschiderea fişierelor

Trebuie să pregătim acum fişierele pentru citire sau scriere, deci le *deschidem*.

Ne propunem să citim din stream-ul tip fişier `inConsum` şi să scriem în stream-ul `outConsum`. Deschidem fişierele folosind următoarele instrucţiuni:

```
inConsum.open("incons.dat");
outConsum.open("outcons.dat");
```

Funcţia `open` are un singur argument cuprins între ghilimele. Prima instrucţiune este un apel al funcţiei `open` asociate cu tipul de dată `ifstream`, iar a doua apelează funcţia `open` asociată cu `ofstream`.

Funcţia `open` asociază variabila stream din program cu un fişier fizic pe disc. Prima instrucţiune realizează o conexiune între obiectul `inConsum` şi fişierul `incons.dat`. A doua instrucţiune leagă obiectul `outConsum` de fişierul `outcons.dat`. `inConsum` este conectat cu `incons.dat` aşa cum `cin` este legat de tastatură.

Mai departe, acţiunea depinde de tipul de stream.

Pentru un fişier de intrare, funcţia `open` poziţionează *markerul de citire* pe primul element din fişier.

Pentru un fişier de ieşire, funcţia verifică dacă acesta există. Dacă există, şterge vechiul conţinut al său. Dacă nu există, îl creează. Apoi *markerul de scriere* este aşezat pe prima poziţie.

Folosim noţiunile de marker de citire sau de scriere pentru a reprezenta locul din care se va citi dintr-un stream de intrare, respectiv se va scrie într-un stream de ieşire.

Exemplu

```
inConsum                        outConsum
23.2                            _
17.4
19.8
16.7
22451
23544
```

Deschiderea fişierelor trebuie realizată înaintea oricărei utilizări a lor deoarece funcţia `open` le pregăteşte pentru citire sau scriere.

Specificarea stream-urilor în instrucţiuni I/O

Pentru citirea şi scrierea din fişier nu va trebui decât să înlocuim obiectele `cin` şi `cout` cu obiectele de tip stream de fişier declarate mai devreme.

Exemplu

```
inConsum >> cant1 >> cant2 >> cant3 >> cant4
         >> indicatiePlecare >> indicatieSosire;
outConsum << "Consumul este " << litriPerKm
          << " litri per km." << endl;
```

Cel mai interesant este că C++ foloseşte o sintaxă uniformă pentru operaţiile I/O, indiferent dacă este vorba despre fişiere sau dispozitive I/O.

4.3 Erori de citire

La citirea datelor de la tastatură sau dintr-un fişier pot apărea erori.

Să presupunem că programul ne cere să introducem un număr întreg, iar noi introducem caractere. Operaţia de intrare eşuează datorită datelor de intrare invalide. În terminologia C++ stream-ul `cin` intră într-o stare de eroare - *fail state*. Dacă un stream intră într-o astfel de stare, orice altă operaţie ulterioară asupra sa este anulată. Din păcate, calculatorul nu opreşte programul şi nu dă niciun mesaj de eroare în astfel de situaţii.

De cele mai multe ori erorile de intrare apar din cauza nepotrivirii tipurilor de dată.

Exemplu

```
int i = 10;
int j = 20;
int k = 30;
cin >> i >> j >> k;
cout << "i: " << i << "j: " << j << "k: " << k;
```

Dacă tastăm

```
1234.56 7 89
```

programul afişează

```
i: 1234 j: 20 k: 30
```

Un alt motiv pentru care un stream intră în *fail state* este incercarea de deschidere a unui fişier de intrare care nu există. Să presupunem că avem pe disc fişierul `myfile.dat` şi scriem următoarele instrucţiuni care îşi propun să lucreze cu acest fişier:

```
ifstream inFisier;
inFisier.open("myfile.dat");
inFisier >> i >> j >> k;
```

Datorită scrierii incorecte a numelui de fişier, `inFisier` va intra în fail state. Ca urmare, variabilele `i`, `j` şi `k` vor avea nişte valori nedeteminate.

Un stream de citire dintr-un fişier intră în *fail state* atunci când se citeşte din fişier dincolo de caracterul `EOF`, cel care marchează finalul fişierului.

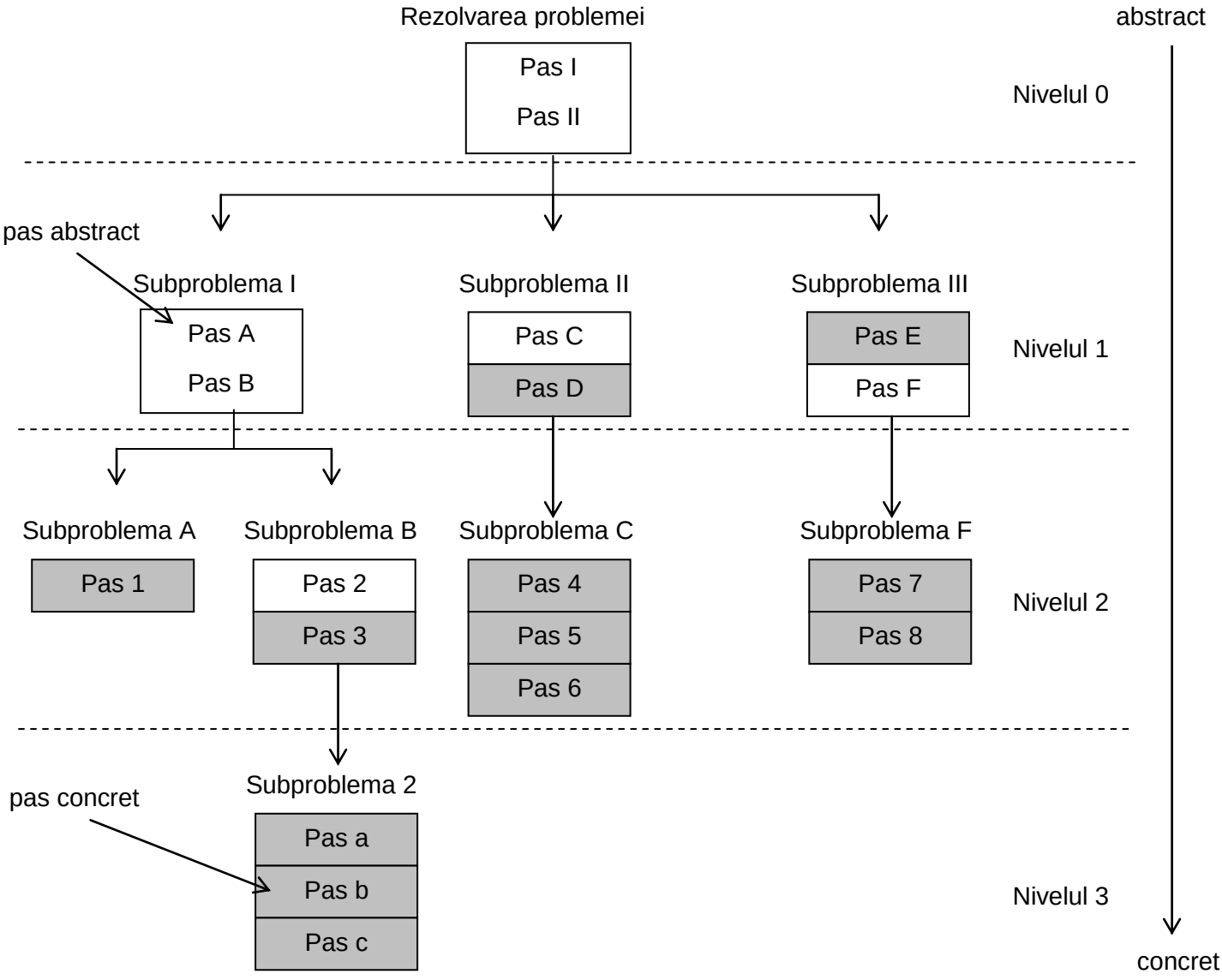
4.4 Scrierea aplicaţiilor

Problemele pe care le-am prezentat până acum au fost simple şi uşor de programat. În acest moment putem scrie şi aplicaţii mai complexe şi de aceea trebuie să vedem cum putem concepe corect o aplicaţie. Vom vorbi despre descompunerea funcţională şi despre proiectarea orientată pe obiecte.

Descompunerea funcţională

Această este o tehnică de dezvoltare a unei părţi a unui program sau chiar a unui program de dimensiuni reduse prin care problema este împărţită în subprobleme mai uşor de rezolvat, soluţii care creează o soluţie globală a întregii probleme.

Printr-o astfel de descompunere creăm o structură ierarhică numită *structură arborescentă*.



Paşii haşuraţi conţin suficiente detalii pentru a putea fi implementaţi în C++. Cei nehaşuraţi trebuie descompuşi în continuare. Fiecare celulă este un modul. Modulele sunt elemente de bază într-o descompunere funcţională.

Pentru conceperea unui modul trebuie să parcurgem următorii paşi:

1. Să schiţăm o soluţie a problemei
2. Să descriem paşii majori
3. Dacă un pas este suficient de simplu pentru a putea fi implementat, nu mai necesită descompuneri ulterioare
4. Dacă pasul poate fi gândit ca o serie de paşi mai mici, este încă un pas abstract

Proiectarea orientată pe obiecte

Descompunerea funcţională poate fi privită ca o metodă de găsire a soluţiei unei probleme cu accent pe algoritmi şi acţiunile care trebuie realizate. Datele, în acest caz, joacă un rol secundar.

Proiectarea orientată pe obiecte se focalizează pe entităţi (obiecte) şi operaţiile posibile asupra acestor entităţi.

Exemplu

O problemă bancară poate avea nevoie de un obiect `contBancar` cu operaţiile asociate `DeschideCont`, `ScrieCec` şi `CreeazaDepozit`. Obiectul `contBancar` constă din date – `numarCont` şi `balantaCurenta`.

Primul pas în proiectarea orientată pe obiecte este identificarea obiectelor din problemă şi a operaţiilor asociate. Soluţia finală va fi exprimată în termeni de obiecte şi operaţii. Datele joacă aici un rol determinant.

În C++ operaţiile asociate cu o clasă sunt scrise ca funcţii şi se numesc *funcţii membre*. O funcţie membră este apelată prin numele unui obiect al clasei urmat de un punct şi de numele funcţiei cu lista de parametri.

Exemplu

```
contBancar.DeschideCont(1000, ”tip1”);
```

Datele care compun obiectul se numesc *date membre*. Instanţele unei clase se numesc *obiecte*, în timp ce instanţele tipurilor de date predefinite cum ar fi `int` se numesc variabile.

Proiectarea orientată pe obiecte conduce la un program care foloseşte o colecţie de obiecte. Fiecare obiect răspunde de o parte din soluţie, iar obiectele comunică între ele prin apelarea funcţiilor membre.

Proiectarea orientată pe obiecte se pretează la scrierea proiectelor mari din următoarele trei motive:

1. Obiectele dintr-un program modelează obiecte din problema de rezolvat;
2. Este posibilă furnizarea şi utilizarea de biblioteci de clase scrise de diverse firme sau persoane independente;
3. Se foloseşte un concept fundamental numit ***moştenire*** care permite adaptarea unei clase la particularităţile problemei fără a modifica codul scris anterior.

Pentru crearea unei soluţii software optime se urmează un proces detaliat pentru obţinerea unei *analize* a *cerinţelor sistemului* (*requirments*) şi *proiectarea* acestuia (*design*) astfel încât să satisfacă cerinţele. Programatorii experimentaţi cunosc faptul că oricât de simplă ar fi problema pe care o au de rezolvat, timpul petrecut cu analiza şi proiectarea poate salva foarte mult timp care se poate pierde cu dezvoltarea unui sistem greşit conceput.

Unified Modeling Language (UML)

Există multe procedee pentru analiza şi proiectarea orientată pe obiecte. Pentru totate acestea se foloseşte un limbaj grafic de comunicare a rezultatelor numit *Unified Modeling Language* (*UML*). Prima versiune a acestui limbaj a fost lansată în anul 1996 de către James Rumbaugh, Grady Booch şi Ivar Jacobson de la Rational Software Corporation. În acelaşi timp, organizaţia non-profit *Object Management Group* (*OMG*) a lansat, la iniţiativa companiilor HP, IBM, Microsoft, Oracle şi Rational Software, o propunere de creare a unui limbaj unic de modelare. UML a fost limbajul adoptat de OMG care, începând cu anul 1997 asigură revizuirea permanentă a sa.

UML este o schemă de reprezentare grafică pentru modelarea sistemelor orientate pe obiecte. Una dintre cele mai atractive caracteristici ale sale este flexibilitatea. UML este extensibil şi independent de multele procese de analiză şi proiectare orientate pe obiecte. Tehnologia obiectelor a devenit indispensabilă în industria software, iar UML este din ce în ce mail folosit. Specificaţiile complete ale UML sunt disponibile la `http://www.uml.org`.

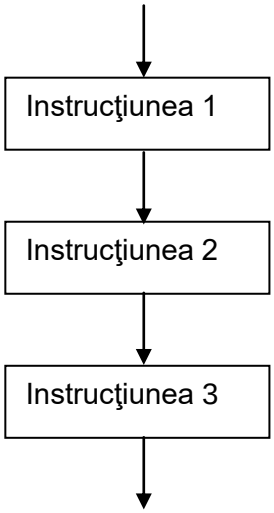
5. Condiţii, expresii logice şi structuri de control pentru selecţie

Până acum, instrucţiunile din programele scrise de noi se executau în ordinea în care erau scrise. Să presupunem acum că dorim să verificăm validitatea unor date de intrare şi apoi să facem un calcul sau să afişăm un mesaj de eroare, dar să nu le facem pe amândouă. Pentru aceasta va trebui să punem o întrebare şi, bazându-ne pe răspuns, să alegem una sau alta dintre variantele de evoluţie a programului.

Instrucţiunea `if` ne permite să executăm instrucţiunile într-o altă ordine decât cea secvențială.

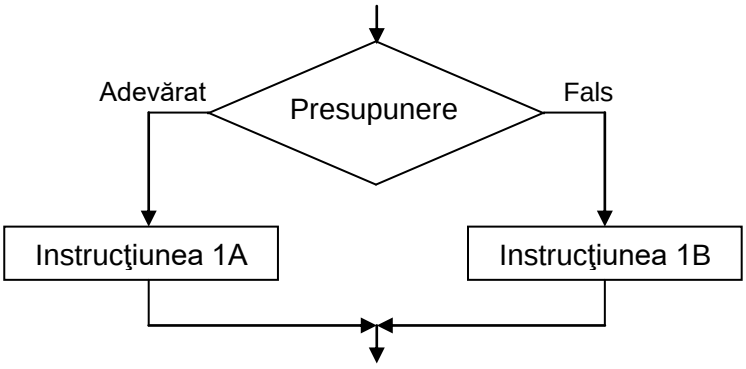
5.1 Ordinea de execuţie a intrucţiunilor

La un moment dat, calculatorul se găseşte sub controlul unei instrucţiuni. După ce această instrucţiune este executată, controlul este trecut următoarei instrucţiuni. În mod obişnuit execuţia este secvenţială.



Acolo unde dorim ca execuţia să nu mai fie secvenţială, introducem *structuri de control*.

Atunci când vrem ca un program să aleagă între două acţiuni alternative, folosim o structură de control al selecţiei. Vom face o presupunere care poate fi adevărată sau falsă. Dacă este adevărată, calculatorul execută o intrucţiune. Dacă este falsă, execută altă instrucţiune.



De exemplu, programul poate testa la un moment dat dacă un angajat va fi plătit pentru ore suplimentare lucrate. Pentru aceasta, el va testa presupunearea că angajatul a lucrat mai mult de 40 de ore într-o săptămână, iar dacă este falsă calculează suma obişnuită.

5.2 Condiţii şi expresii logice

Pentru a pune o întrebare în C++, facem o presupunere care poate fi adevărată sau falsă. Calculatorul *evaluează* presupunerea pentru a constata dacă este adevărată sau falsă.

Expresii logice

În C++ presupunerile iau forma expresiilor logice sau booleene. O astfel de expresie este alcătuită din valori logice şi operaţii.

Datele booleene

Versiunile actuale ale limbajului de programare C++ includ un tip de dată numit `bool` al cărui domeniu de valori este format din constantele literale `true` şi `false`.

Exemplu

```
const bool checkValue = true;
bool dataOK;
dataOK = false;
```

Pentru compatibilitatea cu versiunile mai vechi ale standardului C++ în care acest tip de dată nu exista, valoarea `true` poate fi, de asemenea reprezentată, de orice valoare nenulă, iar `false` poate fi reprezentată de valoarea 0.

Exemplu

```
const bool checkValue = 1;
bool dataOK;
dataOK = 0;
```

Valorile booleene `false` sunt tipărite în mod implicit ca valoare `0`, iar valorile `true` sunt tipărite ca valoare `1`. Operatorul de inserţie în stream `<<` tipăreşte variabilele de tip `bool` ca întregi.

Exemplu

```
bool dataOK;
dataOK = false;
cout << dataOK << endl;
```

Se afișează:

```
0
```

Manipulatorul `boolalpha` setează stream-ul de ieşire ca să afişeze valorile de tip `bool` prin şirurile `true` şi `false`.

Exemplu

```
bool dataOK;
dataOK = false;
cout << boolalpha << dataOK << endl;
```

Se afișează:

```
false
```

Operatori relaţionali

Unei variabile booleene îi poate fi asignat rezultatul comparării a două valori.

Exemplu

```
bool maiMicDecat;
int i, j;
cin >> i >> j;
maiMicDecat = ( i < j );
```

Variabilei `maiMicDecat` i se asignează `true` când `i < j`.

Comparând două valori, prespunem că o relaţie, de exemplu *mai mic decât*, există între ele. Dacă relaţia există într-adevăr, presupunerea este adevărată. Dacă nu, este falsă.

În C++ putem testa următoarele relaţii:

Operator relaţional	**Exemplu**	**Semnificaţie**
==	`x == y`	`x` este egal cu `y`
!=	`x != y`	`x` este diferit de `y`
>	`x > y`	`x` este mai mare decât `y`
<	`x < y`	`x` este mai mic decât `y`
>=	`x >= y`	`x` este mai mare sau egal decât `y`
<=	`x <= y`	`x` este mai mic sau egal decât `y`

În C++, rezultatul unei comparaţii poate fi `true` sau `false`.

Exemplu

Dacă `x` are valoarea `5` şi `y` are valoarea `10`, următoarele relaţii sunt adevărate:

```
x == 5
x != y
y > x
x < y
y >= x
x <= y
```

Dacă `x` este `'M'` şi `y` este `'R'`, expresiile de mai sus sunt de asemenea `true` deoarece sunt comparate codurile ASCII ale caracterelor. În acest cod, literele mari sunt aşezate inaintea celor mici.

Exemplu

`'M' < 'R'` şi `'m' < 'r'` sunt `true`

`'m' < 'R'` este `false`

Trebuie să fim atenţi la tipurile de dată ale valorilor pe care le comparăm. Cel mai sigur este să comparăm int cu int, double cu double etc. Dacă nu, apar conversiile implicite de tip. Ca şi în cazul expresiilor aritmetice, cel mai sigur, în aceste situaţii este să folosim cast explicit pentru a ne face intenţiile cunoscute:

```
nrDouble >= double(nrInt)
```

sau

```
nrDouble >= static_cast<double>(nrInt)
```

De asemenea, valorile char trebuie comparate doar cu valori char.

Exemplu

`'0' < '9'` sau `0 < 9` sunt corecte

`'0' < 9` se realizează cu o forţare de tip şi rezultatul nu este cel aşteptat

Se pot folosi operatori relaţionali nu doar pentru a compara valori ale variabilelor sau constantelor, ci şi pentru a compara valori ale expresiilor aritmetice.

Exemplu

Dacă avem

```
int x = 17, y = 2;
```

atunci următoarea expresie este adevărată:

```
x + 3 == y * 10
```

O eroare des întâlnită în scrierea programelor C++ este confuzia între operatorii `=` şi `==`. Folosirea operatorului `==` pentru asignare sau a operatorului `=` pentru a testa egalitatea sunt erori logice.

Operatori logici

În matematică, operatorii logici AND, OR şi NOT acţionează asupra expresiilor logice care joacă rolul de operanzi. C++ foloseşte simboluri speciale pentru fiecare dintre aceşti operatori.

Operator logic	**Exemplu**	**Semnificaţie**
`&&`	`x && y`	AND logic
`||`	`x || y`	OR logic
`!`	`!x`	NOT logic

Pentru ca rezultatul operaţiei AND (`&&`) să fie `true`, ambii operanzi trebuie să fie `true`. Tabelul de adevăr pentru operatorul `&&` este:

Expresia 1	**Expresia 2**	**Expresia 1 `&&` Expresia 2**
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Operaţia OR (`||`) cere ca cel puţin un operand să fie `true` pentru ca rezultatul să fie `true`. Tabelul de adevăr pentru operatorul `||` este:

Expresia 1	**Expresia 2**	**Expresia 1 `||` Expresia 2**
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Operatorul NOT (!) precede o singură expresie logică şi dă un rezultat opus valorii logice a expresiei. Tabelul de adevăr pentru operatorul `!` este:

Expresia 1	**! Expresia 1**
`false`	`True`
`true`	`False`

NOT ne dă o metodă de a inversa sensul unei presupuneri.

Exemplu

```
!(oreLucrate > 40)
```

este echivalent cu

```
oreLucrate <= 40
```

În unele expresii prima formă este mai clară, în altele cea de-a doua.

Exemple

Conform regulilor lui De Morgan avem:

```
!( a == b )              ⇔  a != b
!( a == b || a == c )    ⇔  a != b && a != c
!( a == b && c > d )     ⇔  a != b || c <= d
```

Evaluări condiţionale

Evaluarea expresiilor logice se face de la stânga la dreapta, aceasta oprindu-se atunci când este cunoscută valoarea întregii expresii. Întrebarea care se pune este cum poate calculatorul să ştie dacă valoarea unei expresii este `true` sau `false` dacă nu a evaluat-o până la capăt.

O operaţie AND este `true` dacă ambii operanzi sunt `true`. Dacă în expresia

```
i == 1 && j > 2
```

`i` are valoarea `10`, subexpresia din stânga va fi `false`, deci expresia nu mai poate avea decât valoarea `false`.

Evaluarea unei expresii OR se opreşte când una dintre subexpresii este `true`, deci rezultatul este `true`.

Precedenţa operatorilor

Am discutat despre precedenţa operatorilor în evaluarea expresiilor aritmetice. Regulile de precedenţă se aplică şi operatorilor logici şi relaţionali. Lista operatorilor şi precedenţa lor este următoarea:

Cea mai înaltă precedenţă

```
( )
!
* / %
+ -
< <= > >=
== !=
&&
||
=
```

Cea mai scăzută precedenţă

Parantezele rotunde au prioritate asupra oricărei operaţii. Acestea trebuie folosite întotdeauna în pereche.

Operatori relaţionali pentru tipuri reale

Operatorii relaţionali pot fi folosiţi pentru orice tipuri de dată.

Vom prezenta cazul valorilor `double`. Ca regulă generală, nu studiaţi egalitatea a două numere reale pentru că, datorită micilor erori care apar la calculele cu numere reale, două astfel de valori nu sunt întotdeauna egale.

Pentru instrucţiunile

```
float oTreime, x;
oTreime = 1.1 / 3.0;
x = oTreime + oTreime + oTreime;
```

Ne aşteptăm ca `x` să aibă valoarea `1.1`, dar probabil că nu va fi aşa. Prima asignare va stoca în variabila `oTreime` o aproximare lui $\frac{1}{3}$, probabil `0.366667`. Cea de-a doua asignare va stoca in variabila `x` o valoare egală cu `3*0.366667`. Dacă vom compara `x` cu `1.1` rezultatul va fi `false`. Diferenţa dintre valorile lui `x` şi `1.1` este un număr foarte mic, de exemplu `2.38419e-008`.

În loc să testăm egalitatea, putem considera că, dacă diferenţa dintre cele două numere este foarte mică, putem considera că numerele sunt egale. Putem înlocui testul de egalitate cu

```
fabs(x-1.1) < 0.00001
```

5.3 Instrucţiunea `if`

Am văzut cum se scriu expresiile logice, vom vedea acum cum putem afecta fluxul secvenţial al programului. Instrucţiunea `if` permite ramificarea fluxului normal. Putem pune o întrebare şi în funcţie de condiţiile existente putem realiza o acţiune sau alta.

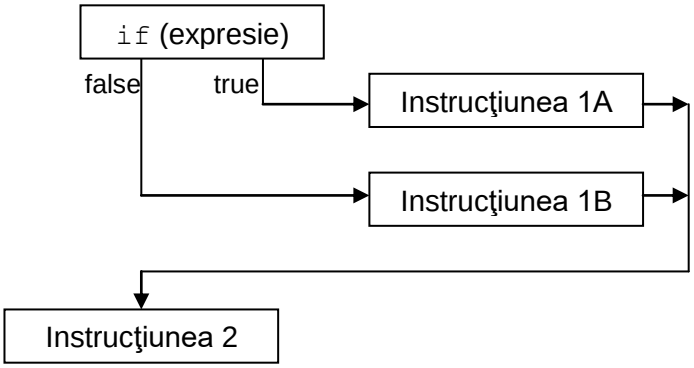
Structura IF-THEN-ELSE

În C++ instrucţiunea if poate fi folosită în două variante: forma IF-THEN-ELSE sau forma IF-THEN. O vom analiza mai întâi pe prima.

Sintaxa formei IF-THEN-ELSE este

```
if(expresie)
        Instrucţiunea 1A
else
        Instrucţiunea 1B
```

Expresia din paranteze poate avea orice tip simplu de dată. Aproape întotdeauna aceasta va fi o expresie logică. Dacă valoarea expresiei este nenulă sau `true`, calculatorul execută *Instrucţiunea 1A*. Dacă valoarea expresiei este `0` sau `false`, se execută *Instrucţiunea 1B*. *Instrucţiunea 1A* se numeşte *clauza then*, iar *Instrucţiunea 1B* se numeşte *clauza else*.



De notat că instrucţiunea `if` foloseşte doar cuvintele rezervate `if` şi `else`, chiar dacă structura se numeşte IF-THEN-ELSE.

Exemple

```
if(oreLucrate <= 40)
    plata = sumaPeOra * oreLucrate;
else
    plata = sumaPeOra * (40.0 + (oreLucrate - 40.0) * 1.5);
cout << plata;
```

În limbaj natural, această instrucţiune spune următorul lucru: *Dacă numărul de ore lucrate este mai mic sau egal decât 40, atunci calculează suma obişnuită de plată şi apoi treci la instrucţiunea de tipărire. Dacă numărul de ore lucrare este mai mare decât 40, atunci calculează suma normală şi suma suplimentară, apoi treci la instrucţiunea de tipărire.*

Blocuri

Pentru a evita împărţirea la zero într-o expresie, să presupunem că atunci când numitorul este egal cu `0` facem două lucruri: tipărim un mesaj de eroare şi încărcăm valoarea `9999` în variabila rezultat. Trebuie, aşadar, să realizăm două instrucţiuni pe aceeaşi ramură, însă sintaxa instrucţiunii if ne limitează la una singură.

Să ne amintim că orice compilator C++ tratează blocul

```
{
...
}
```

ca o singură instrucţiune. Folosind o pereche de acolade pentru a încadra secvenţa de instrucţiuni, instrucţiunile vor deveni un singur bloc.

Exemplu

```
if(numitor != 0)
    rezultat = numarator / numitor;
else
{
    cout << ”Impartirea la 0 nu este permisa.” << endl;
    rezultat = 9999;
}
```

Putem folosi blocuri de instrucţiuni pe ambele ramuri ale unui IF-THEN-ELSE.

Exemplu

```
if(numitor != 0)
{
    rezultat = numarator / numitor;
    cout << ”Impartirea este realizata corect.” << endl;
}
else
{
    cout << ”Impartirea la 0 nu este permisa.” << endl;
    rezultat = 9999;
}
```

După acolada care închide blocul nu se pune niciodată semnul `;`

Forma IF-THEN

Uneori dorim să realizăm o acţiune când se îndeplineşte o condiţie, dar să nu se întâmple nimic atunci când condiţia este falsă. Putem lăsa ramura `else` vidă.

Exemplu

```
if(a <= b)
    c = 20;
else
    ;
```

Mai simplu, putem să nu scriem deloc ramura `else`. Ajungem, astfel, la forma IF-THEN:

`if(`*Expresie*`)`
    Instrucţiune

Putem rescrie instrucţiunea din exemplul anterior astfel:

```
if(a <= b)
    c = 20;
```

Ca şi la forma IF-THEN, ramura din IF-THEN poate fi un bloc de instrucţiuni.

Să presupunem că avem de completat un formular pentru efectuarea unei plăţi. Conform instrucţiunilor, linia 23 din formular trebuie scăzută din linia 17, iar rezultatul trebuie trecut pe linia 24. Dacă rezultatul este negativ, pe linia 24 se trece rezultatul `0` şi se bifează căsuţa 24A. În C++ această cerinţă se implementează astfel:

```
rezultat = linia17 – linia23;
if( rezultat < 0.0 )
{
    cout << ”Bifeaza casuta 24A” << endl;
    rezultat = 0.0;
}
linia24 = rezultat;
```

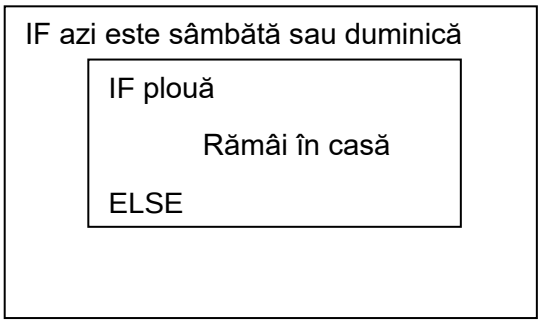
Ce se întâmplă dacă uităm să folosim acoladele?

```
rezultat = linia17 – linia23;
if( rezultat < 0.0 )
    cout << ”Bifeaza casuta 24A” << endl;
    rezultat = 0.0;
linia24 = rezultat;
```

În ciuda intenţiilor noastre, compilatorul tratează clauza `then` ca una cu o singură instrucţiune. Dacă rezultatul este negativ, calculatorul execută instrucţiunea de afişare, setează rezultatul cu `0` şi apoi îl păstrează în `linia24`. Până acum totul este bine. Dacă rezultatul este pozitiv, calculatorul ignoră clauza `then` şi execută următoarea instrucţiune după instrucţiunea `if`. De aceea, rezultatul este setat cu `0`, ceea ce contravine intenţiilor noastre. Concluzia este că alinierea instrucţiunilor nu influenţează în niciun fel compilatorul.

Instrucţiuni `if` imbricate

Nu există restricţii asupra tipului de instrucţiuni pe care le poate conţine o ramură a unui `if`. Prin urmare, o ramură a unui `if` poate conţine un alt `if`. Când folosim o astfel de construcţie, spunem că am creat o *structură de `if` imbricat*.



În general, orice problemă care implică multi-ramificare poate fi codată prin instrucţiuni `if` imbricate.

Pentru a afişa numele unei zile din săptămână putem folosi o serie de instrucţiuni `if` neimbricate.

Exemplu

```
if(zi == 1)
    cout << ”Luni”;
if(zi == 2)
    cout << ”Marti”;
if(zi == 3)
    cout << ”Miercuri”;
if(zi == 4)
    cout << ”Joi”;
if(zi == 5)
    cout << ”Vineri”;
if(zi == 6)
    cout << ”Sambata”;
if(zi == 7)
    cout << ”Duminica”;
```

Acest cod poate fi rescris cu instrucţiuni `if` imbricate.

Exemplu

```
if(zi == 1)
    cout << ”Luni”;
else
```

```
if(zi == 2)
    cout << ”Marti”;
else
    if(zi == 3)
        cout << ”Miercuri”;
    else
        if(zi == 4)
           cout << ”Joi”;
        else
           if(zi == 5)
               cout << ”Vineri”;
           else
               if(zi == 6)
                   cout << ”Sambata”;
               else
                   if(zi == 7)
                       cout << ”Duminica”;
```

Este mult mai eficientă această variantă pentru că implică mai puţine comparaţii. Structura care implementează ramificarea multiplă se numeşte IF-THEN-ELSE-IF. Putem realinia structura de mai sus pentru a evita deplasarea continuă spre dreapta datorită alinierilor convenite.

Exemplu

```
if(zi == 1)
    cout << ”Luni”;
else if(zi == 2)
    cout << ”Marti”;
else if(zi == 3)
    cout << ”Miercuri”;
else if(zi == 4)
    cout << ”Joi”;
else if(zi == 5)
    cout << ”Vineri”;
else if(zi == 6)
    cout << ”Sambata”;
else if(zi == 7)
    cout << ”Duminica”;
```

Folosirea greşită a lui else

La folosirea instrucţiunilor `if` imbricate pot apărea confuzii în legătură cu perechile `if-else`: cărui `if` îi aparţine un `else`?

Să presupunem că dacă nota unui student este mai mică decât 5 dorim să tipărim ”Respins”, dacă nota lui este între 5 şi 6 să tipărim ”Admis”, iar dacă nota este mai mare decât 6 să nu tipărim nimic. Putem coda această problemă astfel:

```
if(media < 6.0)
    if(media < 5.0)
        cout << "Respins" << endl;
    else
        cout << "Admis" << endl;
```

Cum ştim cărui `if` îi aparţine un `else`? O regulă din C++ spune că, în absenţa acoladelor, un `else` este asociat întotdeauna instrucţiunii `if` cea mai apropiată care

nu are încă un `else` drept pereche. Noi obişnuim să aliniem codul astfel încât să reflectăm această împerechere.

Ne propunem să rescriem codul de mai sus în aşa fel încât instrucţiunea `else` să fie asociată primului `if`.

```
if(media < 6.0)
    if(media < 5.0)
        cout << "Respins" << endl;
else
    cout << "Admis" << endl;
```

Această versiune nu rezolvă problema aşa cum dorim noi pentru că simpla aliniere a instrucţiunilor nu este suficientă. Conform regulii, ultimul `else` va fi împerecheat cu al doilea `if`. Pentru o soluţie corectă, plasăm cel de-al doilea `if` într-un bloc de instrucţiuni.

```
if(media < 6.0)
{
    if(media < 5.0)
        cout << "Respins" << endl;
}
else
    cout << "Admis" << endl;
```

Perechea de acolade indică faptul că cea de-a doua instrucţiune `if` este completă, iar `else` trebuie să aparţină primului `if`.

5.4 Testarea stării stream-urilor de intrare / ieşire

În capitolul trecut am prezentat conceptul de stream de intrare şi de stream de ieşire în C++. Am arătat că sunt situaţii în care un stream poate intra in *fail state*:

- date de intrare invalide;
- tentativa de a citi un fişier dincolo de `EOF`;
- intenţia de a deschide pentru citire un fişier inexistent.

C++ oferă un mecanism de a determina când un stream este in *fail state*. În expresii logice se poate folosi efectiv numele stream-ului ca şi cum ar fi o variabilă booleană.

Exemple

```
if(cin)
    ...
if(!inFile)
    ...
```

La testarea stării unui stream rezultatul poate fi nenul, adică ultima operaţie I/O asupra stream-ului a avut succes sau nul, când ultima operaţie I/O asupra stream-ului a eşuat.

6. Bucle

În capitolul trecut am văzut cum putem selecta diferite instrucţiuni pentru execuţie folosind instrucţiunea `if`. O *buclă* este o structură de control care provoacă executarea unei instrucţiuni sau a unei secvenţe de instrucţiuni în mod repetat. Instrucţiunile se execută atâta timp cât sunt îndeplinite una sau mai multe condiţii.

Vom descrie diferite tipuri de bucle şi vom vedea cum se pot implementa acestea folosind instrucţiunea `while`. Vom prezenta, de asemenea, buclele imbricate.

6.1 Instrucţiunea while

Această instrucţiune, ca şi `if`, testează o condiţie. Sintaxa ei este următoarea:

```
while(expresie)
    Instrucţiune
```

Exemplu

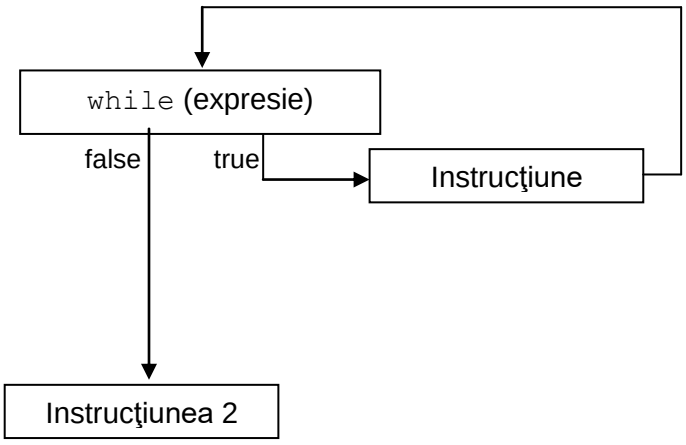
```
while(valIn != 25)
    cin >> valIn;
```

Instrucţiunea care se execută în mod repetat se numeşte *corpul buclei*.

Condiţia din instrucţiunea `while` poate fi o expresie de orice tip de dată. Aproape întotdeauna ea este o expresie logică. Instrucţiunea while din exemplul de mai sus spune următorul lucru: *Dacă valoarea expresiei este `true` (nenulă), execută corpul buclei iar apoi revino şi testează din nou expresia. Dacă expresia este `false` (zero), treci de corpul buclei.*

Dacă expresia este falsă de la început, corpul nu se execută niciodată.

În figura de mai jos arătăm în mod schematic modul de execuţie a instrucţiunii `while`.



Corpul buclei poate fi şi un bloc, fapt care ne permite să executăm mai multe instrucţiuni în mod repetat.

Exemplu

```
while(expresie)
{
...
}
```

Blocul se execută până când expresia devine falsă.

6.2 Fazele de execuţie a unei bucle

1. **Intrarea în buclă**. Este punctul în care programul ajunge la prima instrucţiune din interiorul buclei.
2. **Iterarea**. De fiecare dată când se execută corpul buclei, spunem că facem câte o trecere prin buclă. Această trecere se numeşte *iteraţie*.
3. **Testul buclei**. Reprezintă punctul în care se evaluează expresia din instrucţiunea `while`. În urma acestei evaluări se poate lua decizia de a se începe o nouă iteraţie sau de a trece la instrucţiunea imediat următoare buclei.
4. **Condiţia de terminare**. Este condiţia care provoacă ieşirea din buclă, trecându-se la prima instrucţiune de după buclă. Această condiţie apare în instrucţiunea `while`.
5. **Ieşirea din buclă**. Într-o buclă `while`, ieşirea din buclă apare când expresia din instrucţiunea `while` este `false` sau `0`. În acest moment, se întrerupe repetarea corpului buclei.

Deşi condiţia de terminare poate deveni validă în mijlocul corpului buclei, iteraţia curentă este executată până la capăt şi numai după aceea calculatorul verifică din nou expresia din instrucţiunea `while`.

6.3 Implementarea buclelor folosind instrucţiunea `while`

În rezolvarea problemelor se pot întâlni două tipuri majore de bucle:

- bucla controlată de un contor;
- bucla controlată de un eveniment.

Dacă în timpul unui antrenament sportiv vi se cere să alergaţi *de 3 ori* în jurul stadionului, este vorba de o *buclă controlată de contor*. Dacă, în schimb, vi se cere să alergaţi *până când veţi auzi sunetul fluierului*, avem de a face cu o *buclă controlată de un eveniment*.

Bucla controlată de un contor

O astfel de buclă foloseşte o variabilă numită *variabilă de control al buclei*. Înaintea buclei ea este iniţializată, adică i se atribuie o valoare iniţială. Apoi, în fiecare iteraţie a buclei ea trebuie incrementată.

Exemplu

```
int contorBucla = 1;      //initializare
while(contorBucla <= 10) //test
{
   ...                    //actiune care se repeta
   contorBuclă++;         //incrementare
}
```

În acest exemplu, `contorBucla` este variabila de control al buclei. Ea este iniţializată cu valoarea `1` înainte de intrarea în buclă. Instrucţiunea while testează expresia

```
contorBucla <= 10
```

şi execută bucla atâta timp cât expresia este adevărată. Ultima instrucţiune a buclei incrementează variabila contorBucla. Variabilele folosite în acest fel se numesc *contoare*.

La folosirea acestor bucle, programatorul trebuie să urmărească iniţializarea contorului înaintea instrucţiunii `while`. Trebuie, de asemenea, să urmărească dacă

în interiorul buclei valoarea lui se modifică în aşa fel încât la un moment dat condiţia să devină falsă.

O buclă din care programul nu poate ieşi deloc se numeşte *buclă infinită*. Această situaţie apare atunci când în program se omite incrementarea contorului.

Bucla controlată de un eveniment

Pentru această categorie de bucle condiţia de terminare depinde de un eveniment care poate apărea în timpul execuţiei corpului buclei. Vom studia două tipuri de bucle controlate de evenimente:

- bucla controlată de o valoare de semnalizare (valoare santinelă);
- bucla controlată de sfârşitul unui fişier (`EOF`).

Bucla controlată de o valoare de semnalizare (valoare santinelă)

Aceste bucle se folosesc în special atunci când se prelucrează volume mari de date. La fiecare iteraţie se citeşte şi se prelucrează câte un set de date. Anumite valori dintre cele citite vor semnaliza încheierea buclei `while`. Bucla `while` îşi continuă execuţia atâta timp cât valorile citite nu sunt cele de semnalizare (santinelă).

Exemplu

```
int luna, ziua;
cin >> luna >> ziua;       //citeste primul set de date
while(!(luna == 2 && ziua == 31))
{
   ...                     //procesare
   cin >> luna >> ziua;    //urmatorul set de date
}
```

Este bine ca valorile santinelă să fie dintre cele care nu apar în mod obişnuit între datele valide de intrare.

Înainte de intrarea în buclă este citit primul set de date. Dacă nu este vorba despre valorile santinelă, ele sunt procesate. La sfârşitul buclei se citeşte următorul set de date, revenindu-se apoi la începutul buclei. Bucla se execută până la citirea valorilor santinelă. Acestea nu sunt prelucrate şi conduc la ieşirea din buclă.

Exemplu

Atunci când prelucrăm date de tip `char` putem folosi caracterul *newline* ca valoare santinelă:

```
char inChar;
cin.get(inChar);
while(inChar != '\n')
{
   cout << inChar;
   cin.get(inChar);
}
```

Ce se întâmplă dacă nu introducem valoare santinelă? Un program interactiv ne va cere în continuu noi valori. Dacă intrările în program se fac dintr-un fişier şi datele se epuizează înaintea apariţiei valorii santinelă, stream-ul intră in *fail state*.

O greşeală frecventă în urma căreia programul poate avea o evoluţie nedorită este folosirea neintenţionată a operatorului = în locul lui ==.

Exemplu

```
char inChar, valSemnal;
cin >> inChar >> valSemnal;
while(valSemnal = 1)
   //din greseala am folosit = in loc de ==
{
   ...
   cin >> inChar >> valSemnal;
}
```

Această eroare creează o buclă infinită. Expresia din instrucţiunea `while` este o asignare şi nu o expresie logică. Calculatorul evaluează valoarea variabilei `valSemnal` după asignare. Aceasta va fi `1` şi va fi interpretată ca fiind `true`. Astfel, expresia testată în exemplul de mai sus stochează valoarea `1` în `valSemnal` înlocuind valoarea care tocmai a fost citită. Pentru că expresia este tot timpul `true`, bucla nu se întrerupe niciodată.

Bucla controlată de sfârşitul unui fişier (`EOF`)

După ce programul citeşte şi ultimele date din fişierul de intrare, calculatorul ajunge la sfârşitul fişierului (`EOF`, *end of file*). În acest moment starea stream-ului este normală. Dar dacă încercăm să citim o nouă dată, stream-ul intră în *fail state*. Putem folosi acest comportament în avantajul nostru în buclele `while` în care se citeşte un număr necunoscut de valori. Starea de eroare a stream-ului poate fi interpretată ca valoare santinelă pentru că numele stream-ului poate apărea într-o expresie logică la fel ca o variabilă booleeană. Într-un astfel de test, rezultatul este `true` dacă ultima operaţie de intrare/ieşire a avut succes, sau este `false` dacă aceasta a eşuat.

Să presupunem că avem un fişier de date care conţine valori întregi. Putem scrie:

```
int inVal;
inData >> inVal;
while(inVal)
{
   cout << inVal << endl;
   inData >> inVal;
}
```

Dacă fişierul de date conţine numerele `10`, `20` şi `30`, primele 3 citiri se vor realiza corect. Chiar şi după citirea lui `30` starea stream-ului este normală. Dacă dorim să citim după sfârşitul fişierului, însă, stream-ul va intra în stare de eroare. Aceasta înseamnă că valoarea expresiei logice din `while` va fi `false` provocând ieşirea din buclă. Trebuie spus că orice eroare de citire conduce la intrarea stream-ului *în fail state*.

Similar, se poate folosi şi stream-ul de intrare cin. În sistemele UNIX, combinaţia de taste CTRL-D, iar în sistemele Windows combinaţia de taste CTRL-Z au semnificaţia EOF pentru intrări interactive.

6.4 Operaţii în buclă

Pentru a avea sens, o buclă trebuie să realizeze o operaţie. Vom discuta despre

- contorizare;
- însumare;
- păstrarea unei valori anterioare.

Contorizarea

O operaţie comună este memorarea numărului de iteraţii executate. Programul care urmează citeşte şi numără caracterele dintr-o propoziţie, până la apariţia punctului.

Exemplu

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

int main()
{
  char inChar;
  int count = 0;       //initializarea contorului
  cin.get(inChar);     //citirea primului caracter
  while(inChar != '.')
  {
    count++;           //incrementarea contorului
    cin.get(inChar);   //citirea urmatorului caracter
  }
  cout << "Propozitia are " << count
       << " caractere" << endl;
  return 0;
}
```

După terminarea buclei, `count` va conţine cu 1 mai puţin decât numărul de caractere citite, adică nu numără şi valoarea santinelă (`'.'`). Facem o primă citire înaintea buclei pentru că aceasta este controlată de un caracter de semnalizare.

O variabilă care numără câte iteraţii se execută se numeşte *contor de iteraţii*. În exemplul nostru, variabila `count` este un contor de iteraţii.

Însumarea

O altă operaţie care se poate implementa cu ajutorul buclelor este însumarea unui set de valori.

Exemplu

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

int main()
{
  int numar;
  int suma = 0;
  int contor = 1;
  while(contor <= 5)
  {
    cin >> numar;
    suma = suma + numar;
    contor++;
  }
  cout << "Suma este " << suma << endl;
  return 0;
}
```

Iniţializăm suma cu 0 înainte de începutul buclei. Atunci când se execută prima dată instrucţiunea

```
suma = suma + numar;
```

se adaugă valoarea curentă a variabilei `suma` la valoarea variabilei `numar` pentru a forma noua valoare a variabilei `suma`. După executarea buclei, variabila `suma` va conţine suma celor 5 valori citite, `contor` va conţine valoarea 6 şi `numar` ultima valoare citită.

Păstrarea unei valori anterioare

Avem uneori nevoie în program de o valoare anterioară a unei variabile. Să presupunem că dorim să scriem un program care contorizează numărul de operatori `!=` dintr-un fişier sursă C++. Va trebui să numărăm de câte ori apare semnul `!` urmat de `=`. De fiecare dată vom citi din fişierul de intrare un caracter păstrând ultimele două valori în două variabile diferite. La fiecare iteraţie valoarea curentă devine valoare anterioară şi apoi se citeşte o nouă valoare. Bucla se termină când se ajunge la `EOF`.

Exemplu

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

int main()
{
  int contor = 0;
  char carAnterior;
  char carCurent;
  ifstream inFisier;
  inFisier.open("main.cpp");

  inFisier.get(carAnterior);
  inFisier.get(carCurent);
  while(inFisier)
  {
    if(carCurent == '=' && carAnterior == '!')
      contor++;
    carAnterior = carCurent;
    inFisier.get(carCurent);
  }
  cout << contor
       << " operator(i) != au fost gasiti in fisier"
       << endl;
  return 0;
}
```

Contorul din acest exemplu este un *contor de evenimente*. El este o variabilă care se incrementează atunci când apare un anumit eveniment. Este iniţializat cu valoarea 0 spre deosebire de contorul de iteraţii din exemplul precedent care este iniţializat cu 1.

6.5 Instrucţiuni `while` imbricate

Am studiat în capitolul anterior modul în care se pot scrie instrucţiunile `if` imbricate. Este posibil să folosim şi instrucţiuni `while` imbricate.

Exemplu

Ne propunem să numărăm câte caractere `;` sunt pe fiecare linie dintr-un fişier.

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

int main()
{
  ifstream inFisier;
  inFisier.open("main.cpp");

  char inChar;
  inFisier.get(inChar);
  while(inFisier)
  {
    int contorPunctVirgula = 0;
    while(inChar != '\n')
    {
      if(inChar == ';')
        contorPunctVirgula++;
      inFisier.get(inChar);
    }
    cout << contorPunctVirgula << endl;
    inFisier.get(inChar);
  }
  return 0;
}
```

Să notăm că am omis prima citire pentru bucla interioară. Aceasta a fost deja făcută înaintea buclei exterioare. Dacă o făceam, primul caracter citit s-ar fi pierdut înainte de a-l testa.

Şablonul sintactic al buclelor imbricate este:

```
Iniţializarea_buclei_exterioare
while(condiţia_buclei_exterioare)
{
  ...
  Iniţializarea_buclei_interioare
  while(condiţia_buclei_interioare)
  {
    Procesarea_şi_actualizarea_buclei_interioare
  }
  ...
  Actualizarea_buclei_exterioare
}
```

Fiecare buclă are propria iniţializare, testare şi actualizare. Se poate ca bucla externă să nu facă nicio procesare. Pe de altă parte, bucla interioară poate fi o mică parte a procesării realizate de bucla exterioară.

7. Funcţii

Am folosit deja funcţii C++ atunci când am introdus rutinele din bibliotecile standard, amintind de `sqrt` şi `abs`. Vom analiza în detaliu modul în care programatorul poate să îşi scrie o funcţie, alta decât `main`.

7.1 Funcţii `void`

Limbajul C++ foloseşte două tipuri de subprograme: funcţii void şi funcţii care întorc o valoare. Vom discuta despre prima categorie.

Am văzut că unele funcţii complexe pot fi implementate ca şi colecţii de module, multe dintre acestea putând fi transpuse sub forma unor funcţii `void`.

Scrierea modulelor ca funcţii `void`

În principiu, o astfel de funcţie arată ca funcţia `main`, cu deosebirea că header-ul său foloseşte cuvântul cheie `void` în locul lui `int`. În plus, o funcţie `void` nu conţine nicio instrucţiune de tipul

```
return 0;
```

aşa cum se întâmplă în cazul lui `main`, deci nu întoarce nicio valoare către apelant.

Vom studia un program simplu care foloseşte funcţii `void`. Acest program tipăreşte textul

```
*************
*************
Welcome Home!
*************
*************
*************
*************
```

Iată cum putem schiţa un program care să tipărească acest mesaj:

Nivelul 0

`main`

Tipăreşte două linii de asteriscuri
Tipăreşte textul „Welcome Home!”
Tipăreşte patru linii de asteriscuri

Nivelul 1

`Tipareste2Linii`

Tipăreşte textul „`*************`*”*
Tipăreşte textul „`*************`*”*

`Tipareste4Linii`

Tipăreşte textul „`*************`*”*
Tipăreşte textul „`*************`*”*
Tipăreşte textul „`*************`*”*
Tipăreşte textul „`*************`*”*

Dacă modulele de la nivelul 1 sunt scrise ca funcţii `void`, atunci programul se va scrie astfel:

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

```
void Tipareste2Linii();//prototip de functie
void Tipareste4Linii();//prototip de functie

int main()
{
  Tipareste2Linii();//apel de functie
  cout << "Welcome Home!" << endl;
  Tipareste4Linii();//apel de functie
  return 0;
}

void Tipareste2Linii()
//Aceasta functie tipareste doua linii de asteriscuri
{
  cout << "**********" << endl;
  cout << "**********" << endl;
}

void Tipareste4Linii()
//Aceasta functie tipareste patru linii de asteriscuri
{
  cout << "**********" << endl;
  cout << "**********" << endl;
  cout << "**********" << endl;
  cout << "**********" << endl;
}
```

Se observă similaritatea dintre funcţia `main` şi descrierea de la nivelul 0. Cele două funcţii care fac parte din corpul funcţiei `main` sunt *fara parametri*. Fiecare dintre *definiţiile* acestor funcţii este formată din header-ul funcţiei urmat de un bloc de instrucţiuni care alcătuiesc corpul funcţiei. Header-ul unei funcţii `void` începe cu cuvântul cheie void care semnalează că nu întoarce nicio valoare. Implicit, în corpul unei astfel de funcţii nu va apărea nicio instrucţiune `return` urmată de o valoare. Se poate folosi, totuşi, instrucţiunea

```
return;
```

care încheie execuţia unei funcţii `void`.

Definiţiile funcţiilor pot apărea în orice ordine, deci `main` ar fi putut apărea după cele două funcţii. Cele două declaraţii dinaintea funcţiei `main` se numesc *prototipuri de funcţii*. Ele sunt necesare pentru că regulile din C++ impun declararea unui identificator înaintea folosirii sale. Cei doi identificatori sunt `Tipareste2Linii` şi `Tipareste4Linii` folosiţi în `main`.

7.2 Sintaxa şi semantica funcţiilor void

Apelul funcţiilor (invocarea)

Un apel de funcţie într-un program înseamnă execuţia corpului funcţiei apelate. Şablonul sintactic apelului unei funcţii este următorul:

NumeFuncţie `(` *ListăParametriActuali* `);`

Parametrii dintr-un apel de funcţie se numesc *parametri actuali*. Parametrii care apar în header-ul funcţiei se numesc *parametri formali*. Conform sintaxei C++, lista

de parametri poate să fie vidă. Lista de parametri actuali din şablonul sintactic al apelului de funcţie este subliniată pentru ca poate să lipsească.

Dacă lista conţine doi sau mai mulţi parametri, aceştia trebuie separaţi prin virgulă:

Expresie, Expresie ...

În momentul apelului unei funcţii, parametrii actuali sunt transmişi parametrilor formali conform poziţiei lor, de la stânga la dreapta, iar apoi controlul este transferat primei instrucţiuni din corpul funcţiei. Când se încheie şi execuţia ultimei instrucţiuni din funcţie, controlul este transmis punctului în care s-a făcut apelul.

Declaraţii şi definiţii de funcţii

Pentru că în C++ orice identificator trebuie declarat înainte de a fi folosit, şi funcţiile trebuie să respecte această regulă.

O declaraţie de funcţie anunţă compilatorul despre numele funcţiei, tipul de dată al valorii returnate (poate fi şi void) şi tipurile datelor folosite în lista de parametri. Programul din exemplul de mai sus conţine două declaraţii de funcţii care nu sunt însoţite de corpul funcţiilor. Acestea sunt *prototipuri de funcţii*. Dacă declaraţiile sunt însoţite şi de corpul funcţiei, atunci este vorba despre *definiţii de funcţii*. Toate definiţiile sunt declaraţii, dar nu toate declaraţiile sunt definiţii.

Prototipuri de funcţii

Pentru a satisface cerinţa ca orice identificator să fie declarat înainte de a fi folosit, programatorii C++ plasează prototipurile funcţiilor chiar înaintea definiţiei funcţiei `main`, la începutul programului.

Prototipul de funcţie de mai numeşte în unele limbaje şi *declaraţie forward*. Pentru o funcţie `void`, şablonul sintactic al unei declaraţii este:

`void` *NumeFuncţie* `(`*ListăParametriFormali*`);`

Prototipul nu este însoţit de corpul funcţiei, lista de parametri formali este opţională şi declaraţia se încheie cu `;`

Lista parametrilor formali este opţională şi are următoarea formă:

TipData & NumeVariabilă, TipDată & NumeVariabilă ...

Ampersand-ul `&` este ataşat tipului de dată şi este opţional. Într-un prototip de funcţie, lista parametrilor formali trebuie să specifice tipurile de dată ale parametrilor, dar numele lor poate să lipsească.

Exemplu

```
void Traiectorie(int, double);
```

sau

```
void Traiectorie(int viteza, double unghi);
```

Numele parametrilor sunt utili pentru explicitarea funcţiei, dar compilatorul le ignoră.

Definiţii de funcţii

Aşa cum am văzut deja, definiţia unei funcţii constă din antetul (*header*-ul) funcţiei şi corpul funcţiei care este, de fapt, un bloc. Şablonul sintactic al unei definiţii de funcţie este:

```
void NumeFuncţie (ListăParametriFormali)
{
   Instrucţiune
   ...
}
```

Header-ul funcţiei nu se încheie cu `;` ca la prototipurile de funcţii.

Sintaxa listei de parametri din definiţie diferă de cea folosită în prototip, în sensul că aici trebuie specificate numele tuturor parametrilor formali.

TipData & NumeVariabilă, TipDată & NumeVariabilă ...

7.3 Variabile locale

Deoarece corpul unei funcţii este un bloc, orice funcţie poate include declaraţii de variabile în interiorul ei. Aceste variabile se numesc *variabile locale* pentru că sunt accesibile doar în interiorul blocului în care sunt declarate. În contrast cu variabilele locale sunt variabilele declarate în afara tuturor funcţiilor şi accesibile lor şi se numesc *variabile globale*.

Variabilele locale ocupă spaţiu de memorie doar pe timpul execuţiei funcţiei. Când funcţia îşi încheie execuţia, variabilele locale sunt şterse din memoria calculatorului. Acesta este motivul pentru care la fiecare apel al unei funcţii variabilele locale pornesc cu valori nedefinite, deci trebuie iniţializate în interiorul funcţiei. Fiind şterse din memorie după încheierea apelului, valorile variabilelor locale nu se păstrează între două apeluri de funcţii.

7.4 Parametri

Atunci când se execută o funcţie, ea foloseşte parametrii actuali care i-au fost transmişi prin apelul său, ţinând, însă, cont de natura parametrilor formali. Limbajul C++ acceptă două tipuri de parametri formali: *parametri valoare* şi *parametri referinţă*.

Parametrii referinţă sunt cei pentru care tipul de dată este însoţit, în lista parametrilor formali, de semnul `&`. Funcţia primeşte adresa de memorie a parametrului actual.

Din declaraţiile parametrilor valoare lipseşte semnul `&`, iar funcţia primeşte o copie a valorii parametrului actual.

Tip de parametru	**Folosire**
Parametru actual	Apare în apelul funcţiei. Parametrii corespunzători pot fi atât valoare cât şi referinţă
Parametru valoare formal	Apare în header-ul funcţiei. Primeşte o copie a valorii păstrate în parametrul actual corespunzător
Parametru referinţă formal	Apare în header-ul funcţiei. Primeşte adresa parametrului actual corespunzător

Numărul parametrilor actuali din apelul unei funcţii trebuie să fie egal cu numărul parametrilor formali din header-ul funcţiei. De asemenea, tipurile datelor trebuie să corespundă.

Exemplu

Header: `void ShowMatch(double num1, int num2, char letter);`
Apel: `ShowMatch(varDouble, varInt, varChar);`

Dacă tipurile de dată nu se potrivesc, compilatorul încearcă să aplice operaţiile de cast. Pentru a evita aceste conversii implicite de tip, se pot aplica şi conversii explicite.

Parametrii valoare

Deoarece parametrii valoare primesc copii ale parametrilor actuali, se pot folosi constante, variabile şi expresii în apelul unei funcţii.

Când funcţia se încheie, conţinutul parametrilor valoare este şters, la fel cum se întâmplă în cazul variabilelor locale. Diferenţa dintre parametrii valoare şi variabilelel

locale este că cele din urmă sunt nedefinite la startul funcţiei, în timp ce primele sunt iniţializate automat cu valorile corespunzătoare ale parametrilor actuali.

Parametrii referinţă

Parametrii valoare nu pot fi folosiţi pentru a transmite informaţie către codul apelant pentru că valorile lor se pierd la finalul execuţiei funcţiei. Pentru aceasta se folosesc parametrii referinţă. Prin intermediul parametrilor referinţă funcţiei i se permite să modifice valoarea parametrului actual.

Dacă parametrul unei funcţii este de tip referinţă, se transmite către funcţie locaţia (adresa de memorie) şi nu valoarea parametrului. Există, astfel, o singură copie a informaţiei, folosită atât de apelant cât şi de funcţia apelată. Numele parametrului actual şi cel al parametrului formal devin *sinonime* pentru aceeaşi variabilă. Ceea ce va lăsa funcţia în acea locaţie va fi regăsit de funcţia apelant.

Spre deosebire de cazul parametrilor valoare, către parametrii referinţă pot fi transmise doar nume de variabile, nu şi de constante sau expresii.

Exemplu

Să presupunem că variabila `y` este de tip `double` si variabila `i` este de tip `int`

```
Header:           void Functie2(double val, int& contor);
Apeluri corecte:  Functie2(y, i);
                  Functie2(9.81, i);
                  Functie2(4.9*sqrt(y), i);
Apel incorect:    Functie2(y, 3);
```

O altă diferenţă importantă între parametrii valoare şi cei referinţă apare la verificarea potrivirii tipurilor de dată între parametrii actuali şi cei formali. Dacă în primul caz se realizează conversii implicite, pentru valorile referinţă lucrurile sunt diferite. De această dată compilatorul copiază valoarea parametrului actual într-o *variabilă temporară* de tip corect şi transmite această variabilă temporară parametrului formal. Când funcţia se încheie, variabila temporară este ştearsă din memorie şi modificările din funcţie nu se mai reflectă în codul apelant.

Exemplu

```
#include <iostream>
using namespace std;
int PatratPrinValoare(int);
void PatratPrinReferinta(int&);

int main()
{
 int x = 2;
 int z = 4;
 cout << "x= " << x << " inainte de PatratPrinValoare\n"
      << "Valoarea returnata de PatratPrinValoare: "
      << PatratPrinValoare(x) << endl
      << "x= " << x << " dupa PatratPrinValoare\n"
      << endl;

 cout << "z= " << z << " inainte de PatratPrinReferinta"
      << endl;
 PatratPrinReferinta(z);
 cout << "z= " << z << " dupa de PatratPrinReferinta"
      << endl;
```

```
 return 0;
}

int PatratPrinValoare(int a)
{
 a = a * a;
 return a;
}

void PatratPrinReferinta(int& b)
{
 b = b * b;
}
```

Parametrul referinţă este un nume alternativ pentru argumentul corespondent. Apelul funcţiilor care au parametri valoare este asemănător celui pentru funcţii cu parametri referinţă atunci când parametrii sunt variabile. În ambele situaţii se specifică doar numele variabilei, însă compilatorul stabileşte pe baza tipurilor parametrilor formali care dintre cele două mecanisme de transmitere a parametrilor va fi invocat.

7.5 Funcţii recursive

Programele pe care le-am scris până acum sunt structurate sub forma unor funcţii care apelează alte funcţii într-o manieră ierarhică. În unele aplicaţii, este util ca funcţiile să se poată apela ele însele. O *funcţie recursivă* este o funcţie care se autoapelează.

Rezolvarea problemelor prin recursie presupune scrierea unei funcţii care este apelată recursiv. Această funcţie rezolvă doar cazul cel mai simplu, numit şi *cazul de bază*. Dacă funcţia este apelată pentru cazul de bază, ea returnează un simplu rezultat. Dacă este apelată pentru un caz mai complex, ea divide problema în două module conceptuale: o parte pe care funcţia poate să o rezolve şi o altă parte pe care nu poate să o rezolve. Pentru ca recursia să fie fezabilă, partea care nu poate fi rezolvată imediat trebuie să fie asemănătoare problemei originale, dar să fie un caz mai simplu al acesteia. Această nouă problemă este o variantă a celei originale, iar funcţia va lansa o nouă copie a sa pentru a o trata. Este vorba, deci, de un *apel recursiv* sau de un *pas de recursie*. Pasul de recursie trebuie să includă şi o instrucţiune `return` pentru că rezultatul său va fi combinat cu partea pe care funcţia poate să o rezolve pentru a forma rezultatul final care este transmis codului apelant.

Pasul de recursie se execută în timp ce apelul original este încă activ, acesta nefiind finalizat. Un apel recursiv poate să genereze la rândul său alte apeluri recursive pe măsură ce noua problemă se divide la rândul său în două alte noi probleme. Pentru ca recursia să aibă finalitate, de fiecare dată funcţiile se apelează pe ele însele cu versiuni din ce în ce mai simple ale problemei originale, această secvenţă trebuind să fie concepută în aşa fel încât să conveargă către cazul de bază. La acest punct, funcţia recunoaşte cazul de bază şi se declanşează o serie de return-uri în secvenţă inversă apelurilor, până la apelul original.

Factorialului unui număr întreg nenegativ *n*, notat prin *n*! este dat de produsul

$$n\cdot(n\text{-}1)\cdot(n\text{-}2)\cdot\ldots\cdot 1$$

cu 0!=1.

Această valoare se poate calcula *iterativ* (nerecursiv) folosind o buclă `while`:

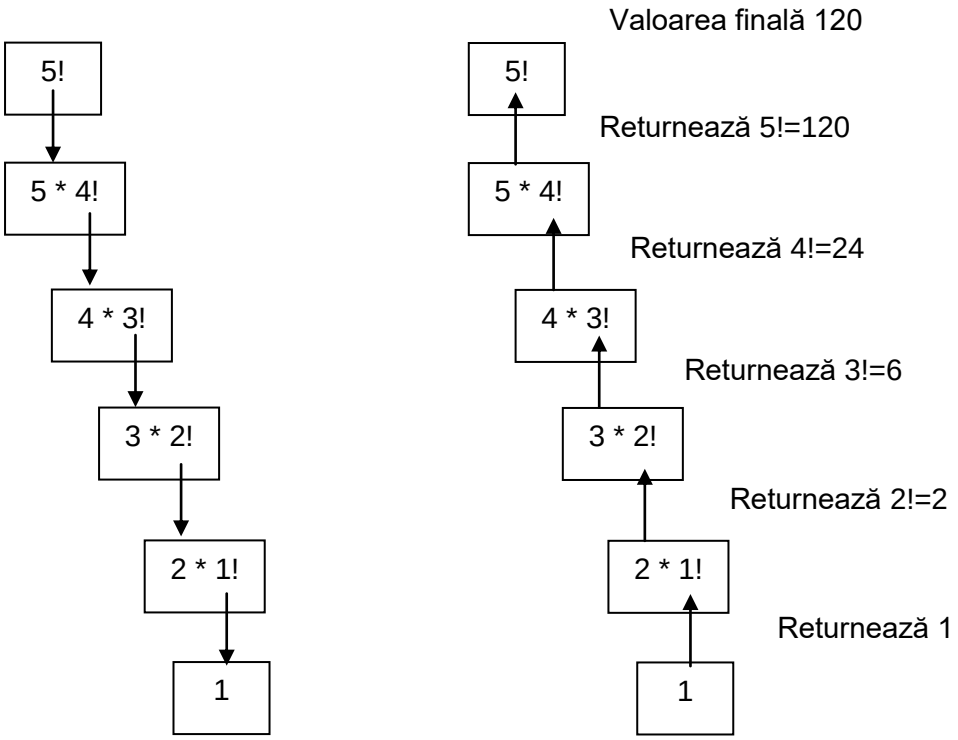
```
int factorial = 1;
```

```
int counter = n;
while(counter >= 1)
{
     factorial = factorial * counter;
     counter--;
 }
```

O definiţie recursivă a acestei operaţii este dată prin relaţia

$$n! = n \cdot (n\text{-}1)!$$

Pentru 5! putem scrie:
5! = 5 · 4 · 3 · 2 · 1
5! = 5 · (4 · 3 · 2 · 1)
5! = 5 · (4!)
Apelurile recursive şi valorile returnate de fiecare dintre ele sunt următoarele:



Programul următor calculează factorialul unui număr n citit de la tastatură:

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

int Factorial(int);

int main()
{
 int n;
 cout << "Introduceti numarul pentru care se calculeaza
factorialul: ";
 cin >> n;
 cout << n << "! = " << Factorial(n) << endl;
 return 0;
}
```

```
int Factorial(int val)
{
 if(val <= 1) //cazul de baza
   return 1;
 else
   return val * Factorial(val - 1);
}
```

Recursie şi iteraţie

Vom compara caracteristicile recursivităţii şi ale iteraţiei.

Ambele tehnici se bazează pe câte o structură de control. Iteraţia foloseşte o structură repetitivă, iar recursivitatea o structură de selecţie. Recursivitatea implementează repetiţia prin apelurile repetate ale aceleiaşi funcţii.

În ambele cazuri este nevoie de un test de terminare. Iteraţia se termină atunci când testul buclei devine fals, în timp ce recursia se termină atunci când se ajunge la cazul de bază. Iteraţia modifică un contor care controlează o buclă. Recursivitatea produce versiuni ale problemei originale până când se ajunge la cazul de bază. Atât iteraţia cât şi recursia pot degenera în bucle infinite dacă testul de final nu este scris corect.

Recursia are, însă, dezavantajul că invocă în mod repetat mecanismul de apel al funcţiilor care presupune ocuparea suplimentară a spaţiului de memorie şi timp de procesor pentru ca la fiecare apel se creează un nou set al parametrilor formali.

Recursivitatea este, totuşi, un mecanism important din punct de vedere al ingineriei software pentru că permite structurarea într-o maniera mult mai judicioasă a anumitor secvenţe de program.

Atunci când suntem puşi în situaţia de a alege între cele două tehnici, va trebui, aşadar, să găsim echilibrul între performanţele aplicaţiei şi dorinţa de a scrie un program corect din punct de vedere conceptual.

8. Tablouri

Tablourile (*arrays*) reprezintă un tip important de structură de date şi sunt colecţii de obiecte de acelaşi tip reunite sub un singur nume.

Uneori este necesar să referim anumite variabile ca un singur grup pentru că este dificil să definim şi să folosim individual fiecare variabilă. De exemplu, dacă dorim să tipărim un set de date în ordine inversă celei în care au fost introduse, trebuie să le citim şi să le salvăm pe toate înainte de a putea face prima tipărire. Dacă avem de a face cu 1000 de valori, trebuie să declarăm 1000 de variabile ca să le stocăm, să scriem 1000 de instrucţiuni de citire şi 1000 de instrucţiuni de tipărire.

Tabloul este un tip de dată care ne permite să programăm mai uşor operaţii asupra grupurilor de valori de acelaşi tip.

8.1 Tipuri de dată simple şi tipuri de dată structurate

O valoare căreia îi este asociat un tip simplu de dată este un element singular care nu poate fi împărţit mai departe în părţi componente. De exemplu, o valoare de tip `int` este un număr întreg şi nu mai poate fi descompus. În contrast cu acesta, o valoare având un tip de dată structurat este o colecţie de elemente. Întreaga colecţie este referită printr-un singur nume şi fiecare componentă poate fi accesată individual.

Un tip de dată structurat este `ifstream` pe care l-am folosit pentru citirea valorilor dintr-un fişier. Atunci când declarăm `inFile` de tip `ifstream`, `inFile` nu reprezintă o singură valoare. El reprezintă întreaga colecţie de date din fişier. Fiecare coomponentă, însă, poate fi accesată individual printr-o operaţie de intrare.

Tipurile simple de date sunt elemente componente pentru tipurile structurate. Un tip de dată structurat pune împreună mulţimea de valori componente şi impune un aranjament specific asupra valorilor.

Aşa cum am văzut deja în clasificarea pe care am făcut-o deja, C++ oferă mai multe tipuri de dată structurate:

- Tipuri structurate
  - tablou (*array*)
  - `struct`
  - `union`
  - `class`

În acest capitol vom discuta despre tablouri.

8.2 Tablouri unidimensionale

Să reluăm exemplul de citire şi afişare în ordine inversă a 1000 de valori. O variantă de implementare a sa este:

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

int main()
{
 int val0;
 int val1;
 ...
 int val999;
 cin >> val0;
 cin >> val1;
```

```
 ...
 cin >> val999;
 cout << val999 << endl;
 ...
 cout << val1 << endl;
 cout << val0 << endl;
 return 0;
}
```

Programul are peste 3000 de linii şi foloseşte 1000 de variabile cu nume asemănătoare. Ar fi mult mai comod dacă numărul din componenţa numelor variabilelor ar fi, de fapt, un contor pe care să îl putem folosi pentru a citi şi scrie bucle `while` în care contorul să ia valori între 0 şi 999.

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

int main()
{
 int val[1000];  //declararea tabloului
 int contor = 0;
 while(contor < 1000)
 {
      cin >> val[contor];
      contor++;
 }
 contor = 999;
 while(contor >= 0)
 {
      cout << val[contor] << endl;
      contor--;
 }
 return 0;
}
```

În acest program am declarat `val` ca fiind un tablou unidimensional, adică o colecţie de variabile, toate de acelaşi tip, în care prima parte a numelui variabilelor este acelaşi, iar ultima parte este un indice cuprins între `[]`. În acest exemplu, valoarea stocată în `contor` se numeşte *indice*.

Declararea tabloului este similară cu declaraţia unei variabile simple, cu o singură excepţie: trebuie declarată şi dimensiunea tabloului. Numărul de componente se declară între `[]`. Declaraţia

```
int val[1000];
```

creează un tablou cu 1000 de componente, toate de tip int. Prima are indicele 0, a doua are indicele 1, iar ultima are indicele 999.

Declararea tablourilor

Un tablou unidimensional este o colecţie structurată de componente (elemente) care pot fi fi accesate individual specificând poziţia componentei printr-un indice o constantă întreagă. Şablonul sintactic al unei declaraţii de tablou unidimensional este următorul:

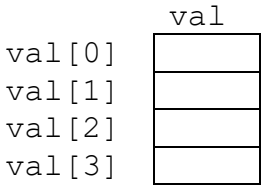
TipDată NumeTablou `[`*ExpresieConstInt*`];`

Componentele unui tablou pot avea aproape orice tip de dată, dar ne vom limita în acest capitol doar la tipurile atomice. Expresia dintre `[]` este o constantă întreagă. Poate fi un literal sau o constantă simbolică. Ea trebuie să fie strict mai mare decât 0 şi defineşte numărul de componente ale tabloului. Dacă valoarea este `n`, domeniul indicilor va fi între `0` şi `n-1`, nu între 1 şi n.

Exemplu

```
int val[4];
```

Prin această declaraţie, compilatorul rezervă într-o zonă compactă de memorie 4 locaţii de tip `int`:



Accesarea componentelor individuale

Pentru a folosi componentele individuale scriem numele tabloului urmat de o expresie indice între `[]`. Expresia specifică numărul componentei accesate şi poate fi atât constantă cât şi variabilă ori o expresie mai complicată. Oricare ar fi, însă, forma indicelui, acesta trebuie să fie o valoare întreagă.

Cea mai simplă formă a expresiei indice este o constantă.

Exemplu

```
int val[4];
val[0] = -2;
val[1] = 4;
val[2] = 18;
val[3] = -199;
```

	val
val[0]	-2
val[1]	4
val[2]	18
val[3]	-199

Spre deosebire de declaraţii, expresiile indice pot fi şi variabile sau expresii întregi.

Exemplu

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

int main()
{
 int i, n[10]; //declararea tabloului
 i = 0;
 while(i < 10)
 {
      n[i] = 0; //initializarea elementelor taboului
      i++;
 }

 cout << "Element" << setw(13) << "Valoare" << endl;
 i = 0;
 while(i < 10)
 {
      cout << setw(7) << i << setw(13)
           << n[i] << endl;
```

```
            i++;
        }

        return 0;
    }
```

Dacă `i` este `0`, atunci se foloseşte `n[0]`, când `i` este `1`, atunci se foloseşte `n[1]` etc.

Fiecare componentă a unui tablou poate fi tratată exact ca o variabilă simplă.

Exemplu

```
int val[4];
val[0] = -2;
cin >> val[2];
cout << val[1];
double x = sqrt(val[2]);
double y = 6.8 * val[2] + 7.5;
```

Indicii din afara limitelor

Să considerăm declaraţia

```
double alfa[100];
```

pentru care domeniul valid al indicilor este 0 – 99. Ce se întâmplă dacă executăm instrucţiunea

```
alfa[i] = 62.4;
```

când `i` < 0 sau `i` > 99 ? Rezultatul este că se accesează locaţii de memorie din afara tabloului. C++ nu verifică încadrarea indicilor între limite şi aceasta este răspunderea programatorului.

Algoritmii de procesare a tablourilor folosesc adeseori bucle pentru a parcurge elementele.

Exemplu

```
int i = 0;
while(i < 10)
{
      alfa[i] = 0.0;
      i++;
}
```

Aceeaşi buclă se poate scrie şi într-o a doua variantă în care variabila de control se compară cu limita superioară a domeniului indicilor.

Exemplu

```
int i = 0;
while(i <= 9)
{
      alfa[i] = 0.0;
      i++;
}
```

Prima variantă este preferată pentru că valoarea cu care se compară variabila de control este aceeaşi cu dimensiunea tabloului, fiind mai sugestivă.

Iniţializarea tablourilor

În limbajul C++ este permisă iniţializarea unei variabile odată cu declararea acesteia.

Exemplu

```
int i = 0;
```

Elementele unui tablou pot fi, de asemenea, iniţializate în instrucţiunea de declarare prin adăugarea unei liste de valori separate prin virgulă, plasate între acolade.

Exemplu

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

int main()
{
 int n[10] = {32, 2, 64, 18, 95, 14, 90, 70, -60, 37};

 cout << "Element" << setw(13) << "Valoare" << endl;
 int i = 0;
 while(i < 10)
 {
      cout << setw(7) << i << setw(13)
           << n[i] << endl;
      i++;
 }

 return 0;
}
```

În acest fel, `n[0]` este iniţializat cu valoarea `32`, `n[1]` cu `2` etc. Între acolade trebuie să se găsească cel puţin o valoare. Dacă se înscriu prea multe valori, compilatorul semnalează o eroare. Dacă sunt mai puţine valori în listă, restul sunt iniţializate cu valoarea `0`.

O facilitate a limbajului C++ este aceea prin care se permite omiterea dimensiunii tabloului atunci când declaraţia şi iniţializarea se fac în aceeaşi instrucţiune.

Exemplu

```
int n[] = {32, 2, 64, 18, 95, 14, 90, 70, -60, 37};
```

Compilatorul stabileşte dimensiunea tabloului la numărul de elemente dintre acolade.

Pentru declararea dimensiunii unui tablou se pot folosi şi constante simbolice. Programul din exemplul următor va citi numerele dintr-un tablou de valori întregi şi va afişa sub formă grafică segmente ale căror lungimi sunt proporţionale cu aceste valori. Acest grafic se numeşte histogramă.

Exemplu

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

int main()
{
 const int dim = 5;
 int val[dim] = {19, 3, 15, 7, 11};

 cout << "Element" << setw(13) << "Valoare"
```

```
            << setw(17) << "Histograma" << endl;

        int i = 0;
        while(i < dim)
        {
            cout << setw(7) << i << setw(13) << val[i]
                 << setw(9);
            int j = 0;
            while(j < val[i])
            {
                cout << '*';
                j++;
            }
            cout << endl;
            i++;
        }

        return 0;
    }
```

Tabourile cu valori tip `char`

Tablourile pot avea orice tip de dată. Vom discuta acum despre stocarea şirurilor de caractere (string-uri) în tablouri de tip `char`. Am văzut că pentru a afişa un string pe ecran putem folosi o instrucţiune de tipărire prin care textul este inserat în stream-ul de ieşire.

Exemplu

```
cout << ”caractere”;
```

Un string precum `”caractere”` este, de fapt, un tablou de valori de tip `char`.

Aceste tablouri au câteva caracteristici speciale care le diferenţiază de celelalte tipuri de tablouri.

Un tablou de caractere poate fi iniţializat printr-un string literal.

Exemplu

```
char clasament[] = ”primul”;
```

Prin această declaraţie, elementele tabloului `clasament` sunt iniţializate cu valorile caracterelor individuale din stringul literal `”primul”`. Dimensiunea tabloului `clasament` este determinata de compilator prin lungimea sirului la care se adaugă automat un caracter special numit *caracterul null* care se numeşte *terminator de şir*. Astfel, tabloul `clasament` va avea şase elemente. Caracterul *null* are codul ASCII 0 iar reprezentarea sa ca şi constantă literală este `’\0’`. Toate string-urile se termină cu acest caracter. Un tablou de caractere care reprezintă un string trebuie declarat întotdeauna suficient de mare ca să cuprindă caracterele şirului şi terminatorul de şir.

Tablourile de caractere pot fi iniţializate şi prin constante individuale printr-o listă de iniţializare.

Exemplu

```
char clasament[] = {’p’, ’r’, ’i’, ’m’, ’u’, ’l’,
’\0’};
```

Putem accesa componentele individuale ale unui tablou de caractere prin folosirea indicilor. În felul acesta, `clasament[0]` este caracterul `’p’`, `clasament[1]` este caracterul `’r’` etc.

Putem introduce valori de la tastatură direct într-un tablou de tip `char` folosind `cin` şi `>>`.

Exemplu

```
char sir[20];
cin >> sir;
```

Prin prima instrucţiune, tabloul `sir` este declarat ca un şir de caractere care poate să stocheze 19 caractere şi terminatorul de şir. A doua instrucţiune citeşte un string de la tastatură adăugându-i automat caracterul null. Este răspunderea programatorului să stabilească dimensiunea tabloului în care se vor păstra aceste caractere. Manipulatorul `setw` poate fi folosit în acest caz pentru a ne asigura că numărul de caractere citite din stream-ul cin nu va depăşi dimensiunea tabloului în care sunt transferate acestea.

Exemplu

Vor fi citite 19 caractere de la tastatură după care se adaugă automat caracterul *null*.

```
cin >> setw(20) >> sir;
```

8.3 Transmiterea tablourilor ca parametri de funcţii

Dacă dorim să transmitem o variabilă ca parametru unei funcţii şi dorim ca funcţia să nu poată să îi modifice valoarea atunci variabila trebuie transmisă prin valoare şi nu prin referinţă. De la această regulă fac excepţie stream-urile. Tot o excepţie o reprezintă şi tablourile pentru că în C++ acestea sunt transmise *întotdeauna* prin referinţă.

Pentru ca o variabilă simplă sau un obiect să fie transmis prin referinţă trebuie ataşat semnul `&` tipului de dată din lista de parametri formali. Fiind transmise întotdeauna prin referinţă, pentru declararea tablourilor nu se foloseşte `&`. Când un tablou este transmis ca parametru, i se transmite de fapt *adresa de bază* care este adresa de memorie a primului său element. În acest fel funcţia îl va putea localiza în memoria calculatorului şi îi va putea accesa elementele.

Exemplu

```
void ModificaTablou(int b[], int dimensiune)
{
  int j = 0;
  while(j < dimensiune)
  {
    b[j] *= 2;
    j++;
 }
}
```

În acest exemplu am folosit operatorul `*=` care este un operator abreviat. În C++ se pot folosi următorii operatori aritmetici de asignare:

Operator	Exemplu	Explicaţie
`+=`	`a += 7`	`a = a + 7`
`-=`	`b -= 6`	`b = b - 6`
`*=`	`c *= 5`	`c = c * 5`
`/=`	`d /= 4`	`d = d / 4`
`%=`	`e %= 3`	`e = e % 3`

Operatorul %= se poate aplica doar variabilelor întregi.

În lista parametrilor formali, declararea unui tablou nu include şi dimensiunea sa între []. Dacă se include dimensiunea, compilatorul o ignoră. Compilatorului îi este necesară doar informaţia referitoare la natura parametrului, adică faptul că este vorba despre un tablou, şi la tipul componentelor sale. Acesta este motivul pentru care trebuie adăugat un al doilea parametru al funcţiei prin care se precizează numărul de componente.

În prototipul unei funcţii care are parametri de tip tablou nu este necesară prezenţa numelor parametrilor formali. De fapt această regulă este valabilă pentru orice funcţie, indiferent de tipul parametrilor săi.

Exemplu

```
void ModificaTablou(int [], int);
```

Programul următor ilustrează diferenţa între trimiterea unui tablou şi a unui element al unui tablou ca parametri de funcţii.

Exemplu

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

void ModificaTablou(int [], int);
void ModificaElement(int);

int main()
{
 const int dimTablou = 5;
 int i, a[dimTablou] = {0, 1, 2, 3, 4};

 cout << "Efectele transmiterii unui tablou "
      << "prin referinta:"
      << "\n\nValorile tabloului original:\n";

 i = 0;
 while(i < dimTablou)
 {
      cout << setw(3) << a[i];
      i++;
 }

 cout << endl;

 //Tablou transmis prin referinta
 ModificaTablou(a, dimTablou);

 cout << "Valorile modificate sunt:\n";

 i = 0;
 while(i < dimTablou)
 {
      cout << setw(3) << a[i];
      i++;
```

```
 }

 cout << "\n\n\n"
      << "Efectele transmiterii unui element "
      << "al tabloului prin valoare:"
      << "\n\nValoarea lui a[3] este "
      << a[3] << '\n';

 ModificaElement(a[3]);
 cout << "Valoarea lui a[3] este " << a[3] << endl;
 return 0;
}

//In functia ModificaTablou, "b" este un alt nume
//pentru tabloul original "a"
void ModificaTablou(int b[], int dimensiune)
{
  int j = 0;
  while(j < dimensiune)
  {
    b[j] *= 2;
   j++;
 }
}

//In functia ModificaElement, "e" este o copie a
//elementului a[3] transmis prin valoare
void ModificaElement(int e)
{
 cout << "Valoarea in functia ModificaElement este "
      << (e *= 2) << endl;
}
```

Programul tipăreşte pe ecran următoarele rezultate:

```
Efectele transmiterii unui tablou prin referinta:

Valorile tabloului original:
  0  1  2  3  4
Valorile modificate sunt:
  0  2  4  6  8


Efectele transmiterii unui element al tabloului prin
valoare:

Valoarea lui a[3] este 6
Valoarea in functia ModificaElement este 12
Valoarea lui a[3] este 6
```

Atunci când se apelează funcţia `ModificaTablou`, i se transmite funcţiei o copie a adresei de memorie a tabloului `a`, iar modificările asupra tabloului `b` vor fi de fapt modificări ale tabloului `a`. Funcţia `ModificaElement` are un parametru de tip valoare, o modificare asupra lui `e` neavând niciun efect asupra parametrului actual `a[3]`.

Pot apărea situaţii în programele noastre când o funcţie nu trebuie să poată modifica elemente ale unui tablou care îi este transmis ca parametru. Deoarece tablourile sunt transmise întotdeauna prin referinţă, modificarea valorilor dintr-un tablou este dificil de controlat. Limbajul C++ oferă mecanismul implementat prin cuvântul cheie `const` care se poate folosi pentru a împiedica modificarea valorilor elementelor unui tablou printr-o funcţie. Dacă parametrul tablou al unei funcţii este de tip `const`, elementele sale devin constante în interiorul funcţiei şi orice intenţie de modificare a valorilor lor este interpretată de compilator ca o eroare de sintaxă.

Exemplu

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

void IncearcaSaModificeTablou(const int []);

int main()
{
     int a[] = {10, 20, 30};
     IncearcaSaModificeTablou(a);
     cout << a[0] << ' ' << a[1] << ' ' << a[2] << endl;
     return 0;
}

void IncearcaSaModificeTablou(const int b[])
{
     b[0] /= 2; //eroare
     b[1] /= 2; //eroare
     b[2] /= 2; //eroare
}
```

Compilatorul semnalează eroare pentru că `b[0]`, `b[1]` şi `b[2]` sunt nemodificabile.

8.4 Tablouri multidimensionale

Tabourile în C++ pot avea mai multe dimensiuni. O modalitate comună de a le folosi este prin *matrice* cu linii şi coloane, fiind vorba în acest caz despre *tablouri cu două dimensiuni*. Pentru a identifica un element al unei astfel de matrice trebuie să specificăm doi indici: primul reprezintă linia iar al doilea reprezintă coloana. Tablourile multidimensionale pot avea şi mai mult de două dimensiuni.

Tablourile multidimensionale pot fi iniţializate odată cu declararea lor în mod asemănător cu iniţializarea tablourilor unidimensionale. De exemplu, un tablou bidimensional `b[2][2]` poate fi declarat şi iniţializat prin instrucţiunea

```
int b[2][2] = { {1,2}, {3,4} };
```

Valorile sunt grupate pe linii, între acolade. Astfel, `1` şi `2` sunt valorile iniţiale pentru `b[0][0]` şi `b[0][1]`, iar `3` şi `4` sunt valorile iniţiale pentru `b[1][0]` şi `b[1][1]`.

Dacă nu sunt suficiente valori pentru o linie, elementele care rămân sunt iniţializate cu valoarea `0`. În felul acesta, declaraţia

```
int b[2][2] = { {1}, {3,4} };
```

iniţializează `b[0][0]` cu `1`, `b[0][1]` cu `0`, `b[1][0]` cu `3` şi `b[1][1]` cu `4`.

Programul următor demonstrează folosirea matricelor multidimensionale.

Exemplu

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

void TiparesteTablou(int [][3]);

int main()
{
     int array1[2][3] = {{1,2,3}, {4,5,6}};
     int array2[2][3] = {1,2,3,4,5};
     int array3[2][3] = {{1,2}, {4}};

     cout << "Valorile din array1 pe randuri sunt:" << endl;
     TiparesteTablou(array1);

     cout << "Valorile din array2 pe randuri sunt:" << endl;
     TiparesteTablou(array2);

     cout << "Valorile din array3 pe randuri sunt:" << endl;
     TiparesteTablou(array3);

     return 0;
}

void TiparesteTablou(int a[][3])
{
     int i = 0;
     while(i < 2)
     {
          int j = 0;
          while(j < 3)
          {
               cout << a[i][j] << ' ';
               j++;
          }
          cout << endl;
          i++;
     }
}
```

Programul apelează funcţia `TiparesteTablou` pentru a tipări conţinutul fiecăruia dintre cele trei tablouri multidimensionale. Atunci când se transmit tablouri unidimensionale ca parametri de funcţii, indicele nu trebuie precizat. În cazul tablourilor multidimensionale, se ignoră doar valoarea primului indice, următorii trebuind să aibă valori. Compilatorul foloseşte aceste dimensiuni pentru a determina poziţia în memorie a elementelor tabloului multidimensional. Toate elementele unui astfel de tablou sunt aşezate fizic în locaţii consecutive, mai întâi cele de pe primul

rând urmate imediat de elementele de pe al doilea rând etc. Această poziţionare în memorie este ilustrată şi de modul în care sunt iniţializate componentele tabloului `array2` din exemplul anterior. Valorile sunt asignate elementelor de pe primul rând, apoi celor de pe al doilea rând, iar `array2[1][2]` primeşte valoarea `0`.

9. Alte structuri de control

În capitolele precedente am văzut care instrucţiuni C++ implementează secvenţele, selecţiile, buclele şi subprogramele. În unele cazuri am introdus mai multe metode de a implementa o structură, ca în cazul instrucţiunii `IF-THEN` sau `IF-THEN-ELSE`.

În acest capitol, vom introduce cinci noi instrucţiuni utile în programare. Prima, `switch`, face mai uşoară scrierea structurilor de selecţie care au mai multe ramuri. Următoarele două instrucţiuni, `for` şi `do-while` facilitează implementarea anumitor tipuri de bucle. Ultimele două instrucţiuni, `break` şi `continue`, sunt folosite de asemenea în bucle şi instrucţiuni de selecţie.

9.1 Instrucţiunea `switch`

Aceasta este o instrucţiune pentru implementarea structurilor de control cu ramificare multiplă. Valoarea *expresiei* `switch` determină ramura care se execută.

Instrucţiunea `switch` constă dintr-o serie de etichete `case` şi o variantă `default`. Programul de mai jos foloseşte instrucţiunea `switch` pentru a contoriza numărul de calificative din categoriile *foarte bine*, *bine*, *satisfăcător* şi *insuficient* pe care le-a primit un elev de-a lungul unui an şcolar. Vom codifica aceste calificative prin literele F – foarte bine, B – bine, S – satisfăcător şi I – insuficient.

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

int main()
{
  char calificativ;
  int nrF = 0,//numarul calificativelor F - foarte bine
      nrB = 0,//numarul calificativelor B - bine
      nrS = 0,//numarul calificativelor S - satisfacator
      nrI = 0;//numarul calificativelor I - insuficient

  cout << "Introduceti litera corespunzatoare"
       << "calificativului." << endl
       << "Tastati caracterul EOF pentru a incheia."
       << endl;

  while((calificativ = cin.get()) != EOF)
  {
    switch(calificativ)
    {
      case 'F':
      case 'f':
        ++nrF;
        break;
      case 'B':
      case 'b':
        ++nrB;
        break;
      case 'S':
```

```
            case 's':
              ++nrS;
              break;
            case 'I':
            case 'i':
              ++nrI;
              break;
            case '\n':
            case '\t':
            case ' ':
              break;
            default://orice alt caracter
              cout << "Ati introdus o litera incorecta."
                   << " Introduceti un nou calificativ." << endl;
          }
        }
        cout << "\n\nTotalurile pentru fiecare calificativ"
             << " sunt:"
             << "\nFoarte bine: " << nrF
             << "\nBine: " << nrB
             << "\nSatisfacator: " << nrS
             << "\nInsuficient: " << nrI << endl;

        return 0;
      }
```

În acest program, utilizatorului i se cere să introducă literele asociate calificativelor. Bucla `while` care implementează operaţia de citire şi prelucrare a datelor este scrisă astfel:

```
while((calificativ = cin.get()) != EOF)
{
  ...
}
```

Testul buclei constă dintr-o asignare urmată de o comparaţie. Asignarea

```
(calificativ = cin.get())
```

dintre parantezele rotunde este executată prima. Funcţia `cin.get()` citeşte un caracter din stream-ul de intrare. Codul ASCII al acestui caracter este, apoi, asignat variabilei `calificativ`. Valoarea acestei variabile este comparată cu `EOF`.

Când caracterul este diferit de `EOF`, calculatorul trece la execuţia instrucţiunii `switch`. Cuvântul rezervat `switch` este urmat, între paranteze, de o expresie integrală (ex. `char`, `int`) a cărei valoare este comparată cu fiecare etichetă `case` care trebuie să fie întotdeauna o expresie constantă integrală. Presupunând că am introdus litera `F` corespunzătoare calificativului *foarte bine*, `F` este comparat cu fiecare `case`. Dacă există o potrivire a valorilor (`case 'F':`), se execută secvenţa de instrucţiuni corespunzătoare acestui `case`. În exemplul nostru, se incrementează variabila `nrF`. Să notăm că, spre deosebire de alte structuri de control, nu este nevoie să folosim blocuri de instrucţiuni pentru a delimita secvenţele asociate unui `case`. Instrucţiunea `break` are ca efect transferul controlului programului la prima instrucţiune de după instrucţiunea `switch`. Dacă omitem instrucţiunea `break`, atunci se execută toate instrucţiunile care urmează ramurii selectate. Dacă nu apare nicio

potrivire a valorilor între expresia din `switch` şi etichetele `case`, se execută varianta `default`.

Sintaxa instrucţiunii switch permite gruparea mai multor etichete case:

```
case 'I':
case 'i':
```

însemnând că pentru ambele cazuri se realizează acelaşi set de acţiuni.

O constantă poate apărea o singură dată în instrucţiunea `switch` pentru că în caz contrar compilatorul generează o eroare de sintaxă. De asemenea, se poate folosi o singură etichetă `default`. Instrucţiunea switch are acelaşi efect ca instrucţiunea IF-THEN-ELSE, fapt ilustrat prin scrierea în două variante a structurii din programul anterior.

Exemplu

```
switch(calificativ)
{
  case 'F':
  case 'f':
    ++nrF;
    break;
  case 'B':
  case 'b':
    ++nrB;
    break;
  case 'S':
  case 's':
    ++nrS;
    break;
  case 'I':
  case 'i':
    ++nrI;
    break;
  case '\n':
  case '\t':
  case ' ':
    break;
  default://alt caracter
    cout << "Ati introdus o litera incorecta."
         << " Introduceti un nou calificativ." << endl;
}
```

sau

```
if(calificativ == 'F' || calificativ == 'f')
  ++nrF;
else if(calificativ == 'B' || calificativ == 'b')
  ++nrB;
else if(calificativ == 'S' || calificativ == 's')
  ++nrS;
else if(calificativ == 'I' || calificativ == 'i')
  ++nrI;
else if(calificativ == '\n' || calificativ == '\t'
        || calificativ == ' ')
{
}
```

```
else
  cout << "Ati introdus o litera incorecta."
       << " Introduceti un nou calificativ." << endl;
```

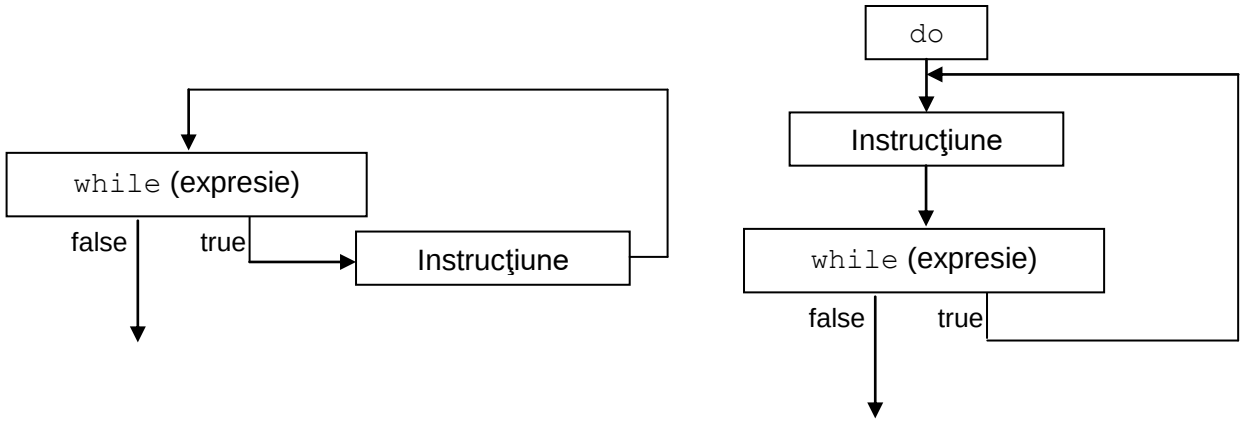
9.2 Instrucţiunea `do-while`

Aceasta este o instrucţiune de control al buclelor în care condiţia de final se testează la sfârşitul buclei, garantând faptul că bucla se parcurge cel puţin o dată.

Şablonul sintactic al instrucţiunii `while` este

```
do
{
   Instrucţiune 1
   Instrucţiune 2
   ...
   Instrucţiune n
} while (expresie);
```

Bucla cuprinsă între `do` şi `while` se execută atâta timp cât expresia din `while` este nenulă sau `true`.



Să comparăm o buclă `while` şi una `do-while` care realizează aceeaşi acţiune: găsesc primul punct din stream-ul de intrare.

```
char inputChar;
cin >> inputChar;
while(inputChar != '.')
  cin >> inputChar;
```

```
char inputChar;
do
  cin >> inputChar;
while(inputChar != '.');
```

Soluţia `while` are nevoie de o primă citire astfel încât `inputChar` să aibă o valoare înainte de intrarea în buclă. Acest lucru nu este necesar pentru soluţia do-while datorită faptului că bucla este executată o dată înainte de a se evalua condiţia.

Putem folosi structura `do-while` pentru a implementa o buclă controlată de un contor dacă ştim în avans că bucla este parcursă cel puţin o dată. Următorul exemplu prezintă două bucle care însumează întregii de la `1` la `n`.

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

int main()
{
  int n = 10;
  int sum = 0;
  int contor = 1;
  while(contor <= n)
```

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

int main()
{
  int n = 10;
  int sum = 0;
  int contor = 1;
  do
```

```
  {                                   {
    sum += contor;                      sum += contor;
    contor++;                           contor++;
  }                                   } while(contor <= n);

  cout << "Suma este "                cout << "Suma este "
       << sum << endl;                     << sum << endl;

  return 0;                           return 0;
}                                   }
```

Dacă `n` este un număr pozitiv, ambele versiuni sunt echivalente. Dar dacă `n` este 0 sau negativ, cele două bucle conduc la rezultate diferite. În versiunea `while`, valoarea finală a lui `sum` este `0` pentru că bucla nu se execută niciodată. În varianta `do-while`, `sum` are valoarea `1` pentru că bucla se execută o dată.

Instrucţiunea `while` testează condiţia înainte de executarea corpului buclei şi se numeşte *buclă pre-test*, în timp ce instrucţiunea `do-while` se numeşte *buclă post-test*.

9.3 Instrucţiunea `for`

Această instrucţiune simplifică scrierea buclelor controlate de contor, în C++ instrucţiunea `for` fiind formă mai compactă a buclei `while`.

Exemplu

```
for(int contor = 1; contor <= n; contor++)
  cout << contor << endl;

int contor = 1;
while(contor <= n)
{
  cout << contor << endl;
  contor++;
}
```

Ambele instrucţiuni tipăresc numerele de la `1` la `n`.

Instrucţiunea for din exemplul de mai sus iniţializează variabila de control al buclei, `contor`, cu valoarea `1`. Atâta timp cât valoarea lui `contor` este mai mică sau egală cu `n` execută instrucţiunea de afişare şi incrementează `contor`.

Corespondenţa dintre componentele instrucţiunii `for` şi cele ale lui `while` este marcată prin caracterele **aldine**. Instrucţiunea `for` plasează iniţializarea dinaintea buclei pe prima poziţie dintre parantezele rotunde. Testul buclei este pe a doua poziţie, iar incrementarea variabilei de contor este pe ultima poziţie. Pe prima poziţie se găsesc acţiunile care se execută înainte de prima iteraţie, iar pe ultima poziţie sunt cele care se execută la sfârşitul fiecărei iteraţii. Dacă este vorba de mai mult de o acţiune, atunci acestea sunt separate prin virgulă.

Ca şi buclele `while`, şi buclele `do-while` sau `for` pot fi imbricate.

Exemplu

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

int main()
{
```

```
  for(int num = 1; num <= 5; num++)
  {
    for(int numToPrint =1; numToPrint <= num; numToPrint++)
      cout << numToPrint;
    cout << endl;
  }

  return 0;
}
```

Rezultatul rulării acestui program este

```
1
12
123
1234
12345
```

9.4 Instrucţiunile `break` şi `continue`

Instrucţiunea `break` introdusă odată cu instrucţiunea `switch` poate fi folosită şi în bucle. Ea provoacă ieşirea imediată din cel mai interior `switch`, `while`, `do-while` sau `for` în care apare.

Una dintre cele mai frecvente situaţii în care se foloseşte este ieşirea din buclele infinite.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
  int num1, num2;
  int contor = 1;
  while(true)
  {
    cin >> num1;
    if(!cin || (num1 >= 100))
      break;
    cin >> num2;
    if(!cin || (num2 <= 50))
      break;
    cout << sqrt(static_cast<double>(num1+num2)) << endl;
    contor++;
    if(contor > 10)
      break;
  }
  return 0;
}
```

Pentru exemplul acesta am fi putut folosi o buclă for de la 1 la 10. În tot cazul, bucla este controlată de contor şi de eveniment, deci preferăm bucla while.

Această buclă conţine 3 puncte distincte de ieşire. Unii programatori se opun acestui stil de programare susţinând că face codul mai greu de urmărit. Rescriind codul, vom vedea ca prima variantă este, totuşi, mai clară.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
  bool num1Valid = true;
  bool num2Valid = true;
  int num1, num2;
  int contor = 1;
  while(num1Valid && num2Valid && contor<=10)
  {
    cin >> num1;
    if(!cin || (num1 >= 100))
      num1Valid = false;
    else
    {
      cin >> num2;
      if(!cin || (num2 <= 50))
        num2Valid = false;
      else
      {
        cout << sqrt(static_cast<double>(num1+num2))
             << endl;
        contor++;
      }
    }
  }
  return 0;
}
```

O altă instrucţiune care afectează fluxul unui program C++ este `continue`. Ea termină iteraţia curentă şi se foloseşte numai în bucle.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
  double valIntrare;
  for(int i = 1; i <= 5; i++)
  {
    cin >> valIntrare;
    if(valIntrare < 0)
      continue;
    cout << sqrt(valIntrare) << endl;
  }
```

```
  return 0;
}
```

Dacă `valIntrare` este negativ, se trece la următoarea iteraţie. Şi această secvenţă de program poate fi rescrisă.

Exemplu

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
  double valIntrare;
  for(int i = 1; i <= 5; i++)
  {
    cin >> valIntrare;
    if(valIntrare >= 0)
      cout << sqrt(valIntrare) << endl;
  }
  return 0;
}
```

Aşadar, diferenţa dintre `break` şi `continue` este că `break` întrerupe imediat bucla curentă, iar `continue` întrerupe iteraţia curentă.

10. Scope. Lifetime. Namespace

Pe măsură ce un program devine tot mai complex, numărul de identificatori creşte. Unii identificatori sunt declaraţi în interiorul blocurilor, alţii în afara lor. Vom studia în acest capitol regulile C++ care permit unei funcţii să acceseze identificatori declaraţi în afara propriului bloc.

Termenul *scope* defineşte domeniul de accesibilitate a identificatorilor.

Lifetime este perioada de existenţă a unui identificator.

10.1 Scope – *domeniul de accesibilitate a unui identificator*

Variabilele locale sunt cele declarate în interiorul unui bloc de instrucţiuni. Corpul unei funcţii este un bloc de instrucţiuni, iar variabilele declarate într-o funcţie sunt variabile locale pentru acea funcţie. Variabilele locale nu pot fi accesate în afara blocului în care au fost declarate. Această regulă se aplică şi constantelor.

Domeniul de accesibilitate, *scope*, este regiunea din program în care se poate folosi un identificator.

C++ defineşte patru categorii de domenii pentru un identificator:

1. *domeniul local* este domeniul unui identificator declarat într-un bloc, din punctul declaraţiei până la sfârşitul blocului;
2. *domeniul global* (domeniul fişier) este domeniul unui identificator declarat în afara oricărei funcţii sau clase, din punctul declaraţiei până la sfârşitul fişierului care conţine codul;
3. *domeniul clasă* se referă la domeniul de vizibilitate introdus prin intermediul claselor;
4. al patrulea domeniu de existenţă este în legătură cu instrucţiunea `go to`.

În C++, numelor de funcţii le corespunde domeniul global sau domeniul clasă. Funcţiile declarate în afara claselor sunt funcţii globale. Odată declarată o astfel de funcţie, ea poate fi folosită de orice altă funcţie din program.

Variabilele globale şi constantele globale sunt declarate în afara tuturor funcţiilor sau claselor. Atunci când o funcţie declară un identificator local cu acelaşi nume ca cel global, identificatorul local este cel folosit în funcţie. Acest principiu se numeşte *precedenţa numelui* sau *ascunderea numelui*.

Exemplu

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

void Functie(double);
const int a = 17;//constanta globala
int b;//variabila globala
int c;

int main()
{
  b = 4;//b: global
  c = 6;
  Functie(42.8);
  return 0;
}

void Functie(double c)//c: local
```

```
{
  double b;//b: local
  b = 2.3; //b: local
  cout << "a = " << a << endl;//a: global
  cout << "b = " << b << endl;//b: local
  cout << "c = " << c << endl;//c: local
}
```

Acest program va afişa :

```
a = 17
b = 2.3
c = 42.8
```

Reguli pentru domeniul de accesibilitate

Când scriem programe C++, declarăm rar variabile globale. Atunci când le folosim, trebuie să ştim exact cum manipulează C++ aceste declaraţii.

Suplimentar accesului global şi local, aceste reguli arată ce se întâmplă dacă într-un bloc includem alt bloc. Identificatorii declaraţi într-un bloc sunt *nelocali* pentru blocurile interioare lui. Spre deosebire de aceştia, identificatorii globali sunt nelocali pentru toate blocurile din program. Dacă un bloc accesează identificatori din afara sa, este vorba de *acces non-local*.

Iată regulile referitoare la domeniul de accesibilitate :

1. Un nume de funcţie care nu este declarat într-o clasă are acces global. Definiţiile de funcţii nu pot fi imbricate în alte funcţii;
2. Domeniul de accesibilitate a unui parametru formal este identic cu cel al unei variabile locale declarate în cel mai exterior bloc al corpului unei funcţii;
3. Domeniul unei constante sau variabile globale se extinde din punctul declaraţiei până la sfârşitul fişierului cu excepţia cazului din regula 5;
4. Domeniul unei constante sau al unei variabile locale este din punctul declaraţiei până la sfârşitul blocului, inclusiv blocurile interioare lui, cu excepţia cazurilor din regula 5;
5. Domeniul de accesibilitate a unui identificator nu include blocurile interioare lui care conţin o declaraţie a unui identificator local cu acelaşi nume.

Exemplu

Acest program ilustrează regulile de accesibilitate din C++.

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

void Bloc1(int, char&);
void Bloc2();
int a1 = 1;
char a2 = 'a';
int main()
{
  //Blocul 1
  Bloc1(a1, a2);
  Bloc2();
  return 0;
}

void Bloc1(int a1, char& b2)
```

```
{
  //Blocul 1
  //Parametrii a1 si b2 ai functiei fac si ei
  //parte din acest bloc
  int c1 = 2;
  int d2 = 3;
  cout << "Bloc1: c1 + d2 = " << c1 + d2 << endl;
}

void Bloc2()
{
  //Blocul 2
  int a1 = 4;
  int b2;
  cout << "Bloc2: b2=";
  cin >> b2;
  if(b2-a1)
  {
    //Blocul 3
    int c1 = 6;
    int b2;
    cout << "Bloc2: b2=";
    cin >> b2;
    cout << "Bloc2: b2-c1 = " << b2-c1 << endl;
  }
}
```

În acest exemplu, într-un bloc poate fi referit un identificator dintr-un alt bloc care îl cuprinde şi pe primul. De exemplu, în blocul 3 putem folosi un identificator declarat în blocul 2. Nu putem folosi, însă, în blocul 3 un identificator declarat în blocul 1 pentru că ar însemna un acces din exterior în interior. Sunt permise doar accese de la un nivel interior spre unul exterior.

Parametrii formali ai funcţiei `Bloc1()` sunt consideraţi ca aparţinând blocului 1. Numele funcţiei, însă, se găseşte la nivelul global.

Să ne închipuim fiecare bloc din programul de mai sus ca fiind înconjurat de câte un contur. Aceste contururi ar reprezenta pereţii unor camere. Pereţii sunt oglinzi cu partea care reflectă orientată în afară, dar transparenţi din interior. Dacă suntem în camera *Blocul 3*, vom putea vedea toate declaraţiile variabilelor din camera *Blocul 2*, dar şi pe cele globale. Nu putem să vedem ce este în *Blocul 1*. Dacă suntem în camera *Blocul 2*, nu putem vedea declaraţiile din *Blocul 3*, dar nici pe cele din *Blocul 1*. Datorită acestei analogii, în mod frecvent se foloseşte termenul de *vizibil* pentru a descrie domeniile de acces. Astfel, variabila `a2` este vizibilă în tot programul, iar variabila `c1` este vizibilă doar în blocul 3.

Trebuie să observăm că variabila `a1` este declarată în două locuri în program. Datorită precedenţei numelui, *Blocul 2* şi *Blocul 3* accesează variabila `a1` declarată în *Blocul 2*, şi nu pe `a1` declarată la nivel global. În mod similar, domeniul de accesibilitate a variabilei `b2` declarată în *Blocul 2* nu include şi *Blocul 3* deoarece acolo este declarată o altă variabilă cu acelaşi nume, `b2`.

Compilatorul lucrează astfel atunci când este vorba de precedenţa numelor. Când o instrucţiune foloseşte un identificator, caută o declaraţie locală. Dacă identificatorul nu este local, trece la un nivel superior până găseşte declaraţia, după

care se opreşte. Dacă ajunge la nivelul global, incluzând şi identificatorii inseraţi cu directiva `#include` şi identificatorul nu este scris greşit, compilatorul generează mesajul de eroare `UNDECLARED IDENTIFIER`.

Declaraţii şi definiţii de variabile

Terminologia C++ face diferenţa între o declaraţie de funcţie şi o definiţie de funcţie. Limbajul C++ aplică aceeaşi terminologie şi variabilelor. Toate declaraţiile de variabile pe care le-am folosit până acum au fost şi definiţii. Să vedem la ce se referă această afirmaţie.

C++ permite scrierea unui program multifişier, adică a unui program pentru care codul este împărţit în mai multe fişiere. Se foloseşte cuvântul rezervat `extern` pentru a referi o variabilă globală localizată în alt fişier.

Dacă `var1` este o variabilă de tip întreg, o declaraţie a sa este următoarea:

```
int var1;
```

care are ca efect rezervarea de către compilator a unei locaţii de memorie de tip `int`. Pe de altă parte, declaraţia

```
extern int var1;
```

arată că variabila este globală şi este declarată în alt fişier, motiv pentru care nu mai trebuie rezervat spaţiu de memorie pentru ea.

Fişierele header conţin declaraţii externe pentru variabilele importante folosite în programe.

Exemplu

În fişierul header `iostream` există declaraţiile

```
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;
```

care permit referirea în program a lui `cin` şi `cout` ca variabile globale.

În terminologia C++, instrucţiunea

```
extern int var1;
```

este o declaraţie a variabilei var1. Pe de altă parte, instrucţiunea

```
int var1;
```

este atât declaraţie cât şi definiţie. O variabilă sau o funcţie pot fi declarate de oricâte ori, dar pot fi definite doar o dată.

Spre deosebire de variabilele globale, folosirea constantelor globale este acceptabilă. Pentru că valorile lor nu pot fi schimbate, folosirea lor nu are comportamente neaşteptate de genul celor care apar atunci când modificarea unei valori a unei variabile globale produce efecte nedorite într-o altă funcţie.

Există două avantaje ale folosirii constantelor globale:

- uşurinţa modificării valorii sale
- consistenţa.

Dacă dorim să modificăm valoarea unei constante globale folosită de mai multe ori în program, trebuie să modificăm doar declaraţia ei. Declarând o constantă şi folosind-o în program, suntem siguri că peste tot folosim aceeaşi valoare.

Nu trebuie să înţelegem că vom declara toate constantele globale. Dacă o constantă este folosită doar într-o funcţie, este lipsit de sens să o declarăm global, ci local.

10.2 Lifetime – *perioada de existenţă a unei variabile*

Perioada de existenţă a unei variabile este perioada de timp în care un identificator are alocată o zonă de memorie.

Exemple

Variabilele locale rămân alocate din momentul intrării în funcţie până la incheierea acesteia, când sunt dealocate.

Perioada de existenţă a variabilelor globale coincide cu perioada de existenţă a programului.

Putem face observaţia că domeniul de accesibilitate este un element legat de faza de compilare, în timp ce perioada de existenţă este legată de faza de execuţie.

În C++ există mai multe categorii de variabile determinate de perioada de existenţă:

- *variabilele automatice* care sunt alocate la intrarea într-un bloc şi dealocate la ieşirea din bloc;
- *variabilele statice* care rămân alocate pe durata întregului program.

Variabilele globale sunt de tip static. Toate variabilele declarate într-un bloc sunt automatice.

Dacă folosim cuvântul cheie `static` atunci când declarăm o variabilă, aceasta devine statică şi perioada ei de existenţă nu se va încheia odată cu terminarea blocului. Ea va fi disponibilă de la un apel al unei funcţii la altul.

Exemplu

```
#include <iostream>
using namespace std;
void a();
void b();
void c();
int x = 1;//variabila globala

int main()
{
  int x = 5;//variabila locala functiei main
  cout << "variabila locala x din functia main "
       << "are valoarea " << x << endl;

  {
    //un nou bloc
    int x = 7;
    cout << "variabila locala x din blocul "
         << "interior functiei main are valoarea "
         << x << endl;
  }

  cout << "variabila locala x din functia main "
       << "are valoarea " << x << endl;

  a();//foloseste o variabila automatica locala x
  b();//foloseste o variabila statica locala x
  c();//foloseste variabila globala x
  a();//reinitializeaza variabila automatica locala x
  b();//variabila statica locala x retine valoarea
```

```
  c();//variabila globala x retine valoarea anterioara

  cout << "variabila locala x din functia main "
       << "are valoarea " << x << endl;

  return 0;
}

void a()
{
  int x = 25;//variabila locala automatica
             //se initializeaza la fiecare apel
             //al functiei a()
  cout << endl << "variabila locala x are valoarea "
       << x << " la intrarea in functia a()" << endl;
  ++x;
  cout << "variabila locala x are valoarea " << x
       << " inainte de iesirea din functia a()" << endl;
}

void b()
{
  static int x = 50;//variabila statica locala
                    //este initializata doar la primul apel
                    //al functiei b()

  cout << endl << "variabila locala statica x are valoarea "
       << x << " la intrarea in functia b()" << endl;
  ++x;
  cout << "variabila locala statica x are valoarea " << x
       << " inainte de iesirea din functia b()" << endl;
}

void c()
{
  cout << endl << "variabila globala x are valoarea " << x
       << " la intrarea in functia c()" << endl;
  x *= 10;
  cout << "variabila globala x are valoarea " << x
       << " inainte de iesirea din functia c()" << endl;
}
```

Rezultatul rulării programului este următorul:

```
variabila locala x din functia main are valoarea 5
variabila locala x din blocul interior functiei main are
valoarea 7
variabila locala x din functia main are valoarea 5

variabila locala x are valoarea 25 la intrarea in functia a()
variabila locala x are valoarea 26 inainte de iesirea din
functia a()
```

```
variabila locala statica x are valoarea 50 la intrarea in
functia b()
variabila locala statica x are valoarea 51 inainte de iesirea
din functia b()

variabila globala x are valoarea 1 la intrarea in functia c()
variabila globala x are valoarea 10 inainte de iesirea din
functia c()

variabila locala x are valoarea 25 la intrarea in functia a()
variabila locala x are valoarea 26 inainte de iesirea din
functia a()

variabila locala statica x are valoarea 51 la intrarea in
functia b()
variabila locala statica x are valoarea 52 inainte de iesirea
din functia b()

variabila globala x are valoarea 10 la intrarea in functia
c()
variabila globala x are valoarea 100 inainte de iesirea din
functia c()
variabila locala x din functia main are valoarea 5
```

Valoarea variabilei statice locale `x` din funcţia `b()` este reţinută de la un apel la altul al funcţiei, spre deosebire de valoarea variabilei locale automatice `x` din funcţia `a()` care se pierde.

În general, se preferă declararea unei variabile locale statice decât folosirea unei variabile globale. Ca şi o variabilă globală, ea rămâne alocată pe durata întregului program, însă este accesibilă doar în interiorul funcţiei în care a fost declarată.

Variabilele statice fie globale, fie locale, sunt iniţializate o singură dată, atunci când programul parcurge prima dată aceste instrucţiuni. Aceste variabile trebuie iniţializate ca si constantele.

10.3 Namespaces (spaţii de nume)

Un program foloseşte identificatori incluşi în diverse domenii de accesibilitate. Uneori, o variabilă dintr-un domeniu se suprapune peste o variabilă cu acelaşi nume din alt domeniu. Suprapunerea numelor poate, uneori, să pună probleme, mai ales atunci când se folosesc în acelaşi program mai multe biblioteci care utilizează la nivel global identificatori cu aceleaşi nume. O astfel de situaţie conduce, de regulă, la erori de compilare.

Limbajul C++ rezolvă această problemă prin *spaţiile de nume* (*namespaces*). Fiecare *namespace* defineşte un domeniu în care sunt plasaţi identificatorii. Pentru a folosi un membru al unui namespace, acesta trebuie să fie precedat de numele *namespace*-ului:

nume_namespace`::`*membru*

Alternativ, se poate folosi declaraţia `using` înaintea identificatorului care este folosit. În mod obişnuit, declaraţiile `using` sunt plasate la începutul fişierului în care sunt folosiţi membrii *namespace*-ului. De exemplu, instrucţiunea

`using namespace` *nume_namespace*`;`

la începutul unui fişier sursă speficică faptul că membrii lui *nume_namespace* pot fi folosiţi în fişier fără a preceda fiecare identificator cu numele *namespace*-ului şi operatorul domeniu `::`.

Exemplu

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

int var = 98;

namespace Exemplu
{
  const double PI = 3.14159;
  const double E = 2.71828;
  int var = 8;
  void TiparesteValori();

  namespace Interior
  {
    enum Ani {FISCAL1 = 2004, FISCAL2, FISCAL3};
  }
}

namespace
{
  double d = 88.22;
}

int main()
{
  cout << "d= " << d;
  cout << "\n(global) var = " << var;
  cout << "\nPI = " << Exemplu::PI << "\nE = "
       << Exemplu::E << "\nvar = "
       << Exemplu::var << "\nFISCAL3 = "
       << Exemplu::Interior::FISCAL3 << endl;

  Exemplu::TiparesteValori();
  return 0;
}

void Exemplu::TiparesteValori()
{
  cout << "\nIn TiparesteValori():\n" << "var = "
       << var << "\nPI = " << PI << "\nE = "
       << E << "\nd = " << d << "\n(global) var = "
       << ::var << "\nFISCAL3 = "
       << Interior::FISCAL3 << endl;
}
```

Programul afişează pe ecran următorul rezultat:

```
d= 88.22
(global) var = 98
```

```
PI = 3.14159
E = 2.71828
var = 8
FISCAL3 = 2006

In TiparesteValori:
var = 8
PI = 3.14159
E = 2.71828
d = 88.22
(global) var = 98
FISCAL3 = 2006
```

Instrucţiunea

```
using namespace std;
```

informează compilatorul că va fi folosit *namespace*-ul `std`. Întreg conţinutul fişierului header `iostream` este definit ca parte a lui `std`. Această declaraţie presupune că unii membri ai lui `std` vor fi folosiţi în mod frecvent în program. Programatorul are, astfel, posibilitatea, să acceseze toţi membrii acestui *namespace* şi să scrie instrucţiuni într-o variantă mai concisă:

```
cout << "d= " << d;
```

decât scriind

```
std::cout << "d= " << d;
```

Fără instrucţiunea de declarare a lui `std`, fiecare `cout` şi `endl` ar trebui să fie precedat de `std::`.

Prin declaraţiile

```
namespace Exemplu
{
  const double PI = 3.14159;
  const double E = 2.71828;
  int var = 8;
  void TiparesteValori();

  namespace Interior
  {
    enum Ani {FISCAL1 = 2004, FISCAL2, FISCAL3};
  }
}
```

se definesc *namespace*-urile `Exemplu` şi `Interior`. Primul dintre ele conţine constantele `PI` şi `E`, variabila `var`, funcţia `TiparesteValori()` şi *namespace*-ul intitulat `Interior` care este declarat în interiorul său. Acesta, la rândul său, conţine o enumerare. Aceasta reprezintă un tip de dată introdus prin cuvântul cheie `enum` urmat de un identificator şi este un set de constante întregi reprezentate de identificatorii dintre acolade. Valorile acestor constante încep cu `0` şi sunt incrementate apoi cu `1`, cu excpţia cazului în care se specifică altfel. În exemplul de mai sus, constantele `FISCAL1`, `FISCAL2` şi `FISCAL3` au valorile `2004`, `2005` şi `2006`.

Un *namespace* poate fi declarat la nivel global sau în interiorul unui alt *namespace*.

Prin secvenţa

```
namespace
```

```
{
  double d = 88.22;
}
```

se declară un *namespace* fără nume care conţine variabila d. Un *namespace* fără nume ocupă întotdeauna domeniul global, cel din care fac parte şi variabilele globale.

Accesul la membrii *namespace*-ului se face prin prefixarea acestora:

```
cout << "\nPI = " << Exemplu::PI << "\nE = "
     << Exemplu::E << "\nvar = "
     << Exemplu::var << "\nFISCAL3 = "
     << Exemplu::Interior::FISCAL3 << endl;
```

`PI`, `E` şi `var` sunt membri ai lui `Exemplu` şi sunt precedaţi de `Exemplu::`. Apartenenţa membrului `var` trebuie precizată pentru că altfel ar fi tipărită variabila globală cu acelaşi nume. `FISCAL3` trebuie precedat de `Exemplu::Interior::` pentru că face parte din *namespace*-ul Interior care este declarat în *namespace*-ul `Exemplu`.

Funcţia `TiparesteValori()` este membru al lui `Exemplu` şi poate accesa în mod direct ceilalţi membri ai *namespace*-ului. Variabila globală var este accesată prin folosirea operatorului domeniu unar `::`, iar `FISCAL3` este accesată prin folosirea înaintea sa a lui `Interior::`.

Cuvântul rezervat using poate fi folosit şi pentru a permite accesul la membri individuali dintr-un *namespace*. Instrucţiunea

```
using Exemplu::PI;
```

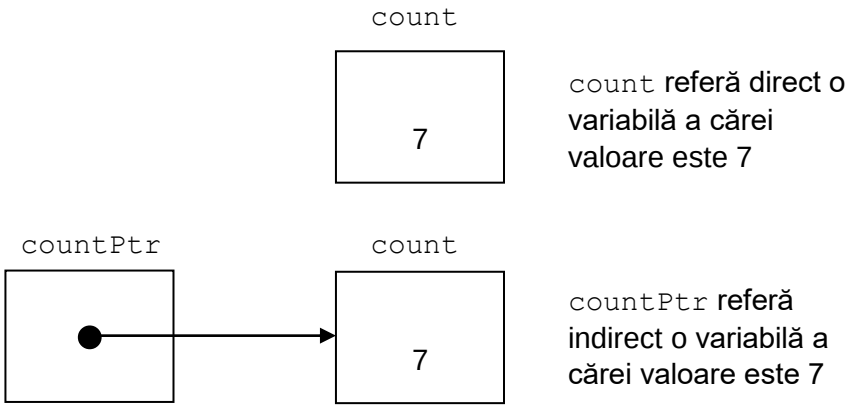
ar permite folosirea în instrucţiunile care urmează a identificatorului `PI` fără ca acesta să mai fie precedat de `Exemplu::`. Aceste declaraţii se fac doar atunci când sunt folosiţi în mod intens într-un program un număr limitat de identificatori dintr-un *namespace*.

11. Pointeri

Pointerii reprezintă caracteristica cea mai puternică a limbajului de programare C++. În capitolele precedente am văzut cum se pot scrie funcţii ale căror parametri sunt transmişi prin referinţă. Mecanismul transmiterii parametrilor prin intermediul pointerilor este o extensie a transmiterii prin referinţă. Am văzut, de asemenea, că tablourile sunt colecţii de elemente de acelaşi tip care sunt stocate în locaţii succesive de memorie. Vom arăta în acest capitol că există o strânsă relaţie între pointeri şi tablouri şi vom studia modul în care se pot manipula tablourile prin intermediul pointerilor.

11.1 Variabilele de tip pointer

Variabilele de tip pointer stochează adrese de memorie. Pot, de exemplu, să păstreze adrese de memorie ale altor variabile care, la rândul lor, conţin alte valori. În acest sens, un nume de variabilă referă *direct* o valoare, iar un pointer referă *indirect* o valoare. Referirea unei valori printr-un pointer se numeşte *indirectare*.



Pointerii, ca orice altă variabilă, trebuie declaraţi înainte de a fi folosiţi.

Exemplu

```
int *countPtr, count;
```

Prin aceste declaraţii, variabila `countPtr` este de tip `int*`, adică este pointer către o valoare întreagă. Variabila `count` este de tip întreg şi nu pointer la întreg. Fiecare variabilă declarată ca pointer este precedată de un asterisc `*`.

Exemplu

```
double *x, *y;
```

Atât `x` cât şi `y` sunt pointeri către valori de tip `double`. Aceste variabile pot păstra adrese de memorie ale unor valori de tip `double`. Pot fi declaraţi pointeri ca să indice către variabile de orice tip de dată.

Este indicat ca pointerii să fie iniţializaţi fie odată cu declaraţia acestora, fie printr-o instrucţiune de asignare. Un pointer poate fi iniţializat cu `0`, `NULL` sau cu o adresă de memorie. Un pointer cu valoarea `0` sau `NULL` nu pointează către nicio zonă de memorie. Constanta `NULL` este declarată în fişierul header `<iostream>` şi în alte câteva fişiere din biblioteca standard. Iniţializarea prin valoarea `NULL` este echivalentă cu iniţializarea prin valoarea `0`, dar în C++ se preferă cea de-a doua variantă. Întregul `0` este convertit automat către o adresă de tipul pointerului. Valoarea `0` este singurul întreg care poate fi asignat direct unei variabile pointer fără o conversie prealabilă.

Pointerii constanţi sunt cei al căror conţinut nu se poate modifica. Un astfel de pointer trebuie iniţializat în instrucţiunea de declarare.

Exemplu

```
int x;
const int *xPtr = &x;
```

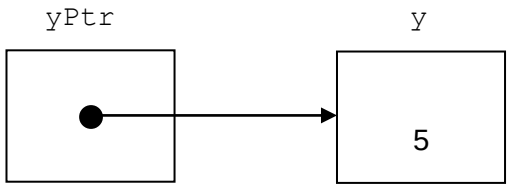
11.2 Operatori pentru pointeri

Operatorul adresă `&` este unar şi returnează adresa operandului său.

Exemplu

```
int y = 5;
int *yPtr;
yPtr = &y;
```

Prin ultima instrucţiune, adresa de memorie a variabilei `y` este încărcată în variabila pointer `yPtr`. În urma acestei asignări, vom spune că `yPtr` pointează către `y`.



Exemplu

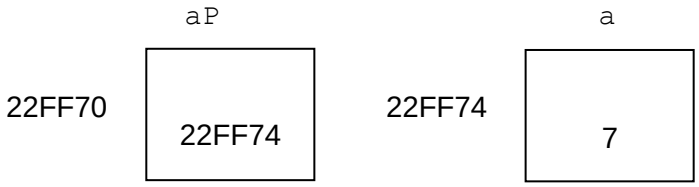
```
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int a;
  int *aP;
  a = 7;
  aP = &a;
  cout << "Adresa lui a este " << &a
       << "\nValoarea lui aP este " << aP;
  cout << "\n\nAdresa lui a este " << a
       << "\nValoarea lui *aP este " << *aP;
  cout << "\n\nOperatorii * si & sunt inversi unul altuia. "
       << "\n&*aP = " << &*aP
       << "\n*&aP = " << *&aP << endl;
  cout << "\n\nAdresa lui aP este " << &aP << endl;

  return 0;
}
```

Acest pProgramul afişează pe ecran următorul rezultat:

```
Adresa lui a este 0x22ff74
Valoarea lui aP este 0x22ff74
Adresa lui a este 7
Valoarea lui *aP este 7
Operatorii * si & sunt inversi unul altuia.
&*aP = 0x22ff74
```

```
*&aP = 0x22ff74
Adresa lui aP este 0x22ff70
```

În figura de mai jos reprezentăm variabila `aP` care presupunem că este localizată în memorie la adresa `22FF70` şi variabila `a` care se găseşte la adresa `22FF74`.



Operandul operatorului adresă trebuie să fie un *lvalue* (*left value*), adică o entitate căruia îi poate fi asignată o valoare, de exemplu valoarea unei variabile. Operatorul adresă nu poate fi aplicat constantelor sau expresiilor al căror rezultat nu poate fi referit.

Operatorul * numit *operator de indirectare* sau de *dereferenţiere* returnează un sinonim sau un alias pentru obiectul asupra pointează operandul său.

Instrucţiunea

```
cout << *aP << endl;
```

tipăreşte valoarea variabilei a care este 7, în acelaşi fel în care o face instrucţiunea

```
cout << a << endl;
```

Folosirea lui `*` în acest mod se numeşte *dereferenţierea unui pointer*.

Un pointer dereferenţiat poate fi folosit în partea stângă a unei instrucţiuni de asignare:

```
*aP = 5;
```

Prin această operaţie, valoarea `5` este asignată variabilei `a`.

- Un pointer dereferenţiat poate fi folosit în diverse operaţii:

```
cin >> *aP;
```

Pointerul dereferenţiat este un *lvalue*.

11.3 Transmiterea prin pointeri a parametrilor funcţiilor

În C++ se pot folosi pointerii şi operatorul de indirectare pentru a simula transmiterea parametrilor prin referinţă.

Exemplu

```
#include <iostream>
using std::cout;
using std::endl;

int cubPrinValoare(int);
void cubPrinReferinta(int*);

int main()
{
  int val = 5;
  cout << "Valoarea numarului este " << val;
  cubPrinValoare(val);
  cout << "\nValoarea dupa apelul cubPrinValoare(val) este "
       << val;
```

```
  val = 5;
  cout << "\n\nValoarea numarului este " << val;
  cubPrinReferinta(&val);
  cout << "\nValoarea dupa apelul cubPrinReferinta(&val) "
       << "este " << val << endl;

  return 0;
}

int cubPrinValoare(int n)
{
     return n * n * n;
}

void cubPrinReferinta(int *nPtr)
{
     *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Acest program afişează:

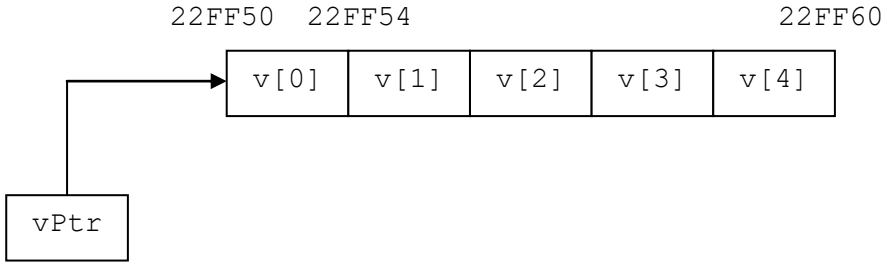
```
Valoarea numarului este 5
Valoarea dupa apelul cubPrinValoare(val) este 5

Valoarea numarului este 5
Valoarea dupa apelul cubPrinReferinta(&val) este 125
```

Mecanismul prin care valoarea parametrului actual `val` este modificată prin funcţia `cubPrinReferinta(&val)` este similar celui prin care parametrul actual este modificat la transmiterea parametrilor prin referinţă. Se transmite adresa de memorie a variabilei care iniţializează parametrul formal `nPtr`. Modificarea valorii la care pointează `nPtr` înseamnă modificarea valorii de la adresa `&val`.

11.4 Aritmetica pointerilor

Pointerii pot fi folosiţi în expresii aritmetice, asignări şi comparaţii, însă nu toţi operatorii pot avea pointeri ca operanzi. Asupra pointerilor pot fi realizate operaţii de incrementare (++), decrementare (−−), adăugare a unui întreg (+ sau +=), scădere a unui întreg (− sau −=) şi scădere a unui pointer din alt pointer.

Să presupunem că declarăm tabloul `int v[5]` al cărui prim element este plasat de compilator la adresa de memorie `22FF50`. Pentru un calculator pe care întregii sunt reprezentaţi pe 4 bytes, cele cinci elemente ale tabloului sunt plasate la adresele de memorie din figura de mai jos.



Pointerul `vPtr` poate fi iniţializat cu adresa primului element al taboului folosind una dintre instrucţiunile următoare:

```
int *vPtr = v;
```

```
vPtr = &v[0];
```

Adresa celui de-al doilea element al tabloului este `&v[1]`.

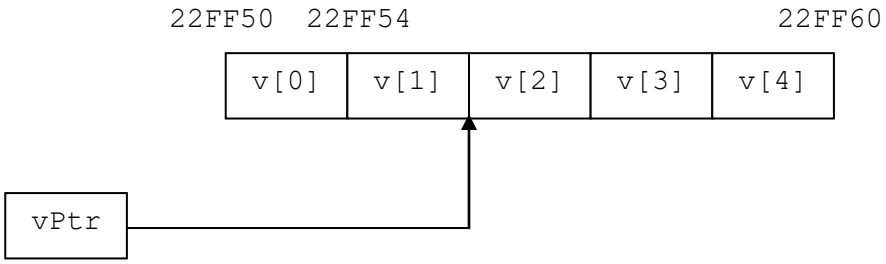
Spre deosebire de operaţiile matematice în care adunarea `22FF50+2` are ca rezultat valoarea `22FF52`, în aritmetica pointerilor adăugarea unui întreg la o adresă de memorie are ca rezultat o nouă adresă de memorie. Aceasta este egală cu adresa iniţială la care se adaugă un număr de locaţii de memorie egal cu valoarea întregului pe care l-am adunat înmulţită cu dimensiunea obiectului la care referă pointerul.

Exemplu

`vPtr += 2;`

are ca rezultat valoarea `22FF58`, adică `22FF50 + 2 *` 4 pentru că am presupus că întregii sunt stocaţi pe 4 bytes.

În urma acestei operaţii, `vPtr` va pointa către `v[2]`.



Pentru un pointer către `double`, aceeaşi operaţie are ca rezultat valoarea `22FF60` pentru că la `22FF50` se adaugă `2 * 8` bytes.

Programul de mai jos prezintă operaţiile care se pot efectua asupra pointerilor.

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

int main()
{
  int v[5];
  int *vPtr = &v[0];
  cout << "Adresa lui v[0] este " << &v[0] << endl;
  cout << "Adresa stocata in vPtr este " << vPtr << endl;

  vPtr += 2;
  cout << "\nAdresa stocata in vPtr dupa operatia vPtr += 2"
       << " este " << vPtr;

  vPtr -= 4;
  cout << "\nAdresa stocata in vPtr dupa operatia vPtr -= 4"
       << " este " << vPtr;

  vPtr++;
  cout << "\nAdresa stocata in vPtr dupa operatia vPtr++"
       << " este " << vPtr;

  ++vPtr;
  cout << "\nAdresa stocata in vPtr dupa operatia ++vPtr"
       << " este " << vPtr;

  vPtr--;
```

```
    cout << "\nAdresa stocata in vPtr dupa operatia vPtr--"
         << " este " << vPtr;

    --vPtr;
    cout << "\nAdresa stocata in vPtr dupa operatia –vPtr"
         << " este " << vPtr;

    int *v2Ptr = &v[4];
    cout << "\nRezultatul operatiei v2Ptr-vPtr este "
         << v2Ptr-vPtr << endl;

    return 0;
}
```

Acest program afişează următorul rezultat:

```
Adresa lui v[0] este 0x22ff50
Adresa stocata in vPtr este 0x22ff50

Adresa stocata in vPtr dupa operatia vPtr += 2 este 0x22ff58
Adresa stocata in vPtr dupa operatia vPtr -= 4 este 0x22ff48
Adresa stocata in vPtr dupa operatia vPtr++ este 0x22ff4c
Adresa stocata in vPtr dupa operatia ++vPtr este 0x22ff50
Adresa stocata in vPtr dupa operatia vPtr-- este 0x22ff4c
Adresa stocata in vPtr dupa operatia -vPtr este 0x22ff48
Rezultatul operatiei v2Ptr-vPtr este 6
```

Un pointer poate fi asignat altui pointer doar dacă cei doi au acelaşi tip. În caz contrar, trebuie aplicată o operaţie de conversie pentru ca pointerul din dreapta asignării să fie adus la tipul pointerului din stânga. Excepţie de la această regulă în face pointerul `void*` care este un tip generic şi poate reprezenta orice tip de pointer fără a mai fi nevoie de castconversie. Pe de altă parte, pointerul `void*` nu poate fi dereferinţiat pentru că numărul de bytes corespunzător lui nu poate fi determinat de compilator.

11.5 Pointeri şi tablouri

Tablourile şi pointerii sunt, în limbajul C++, în strânsă legătură. Un nume de tablou poate fi interpretat ca un pointer constant, iar pointerii pot fi indexaţi ca şi tablourile.

Pentru tabloul `v[5]` am declarat variabila pointer `vPtr` pe care am iniţializat-o cu `v`, adresa primului element al tabloului. Elementul `v[3]` poate fi referit şi prin expresiile pointer

```
*(vPtr + 3)
*(v + 3)
```

Valoarea 3 din aceste expresii se numeşte *offset* la pointer, iar o astfel de expresie care accesează un element al unui tablou se numeşte *notaţie offset* sau *notaţie pointer*. Fără paranteze, expresia

```
*vPtr + 3
```

ar fi adunat valoarea 3 la expresia `*vPtr`, adică la `v[0]`.

Pentru pointeri se pot folosi indici la fel ca şi pentru tablouri.

Exemplu

```
cout << vPtr[1];
```

În schimb, numele unui tablou este un pointer constant şi nu poate fi folosit în operaţii care i-ar schimba conţinutul.

Exemplu

```
v += 3;
```

este o operaţie invalidă pentru că ar modifica valoarea pointerului care reprezintă tabloul printr-o operaţie de aritmetică a pointerilor.

Tablouri de pointeri

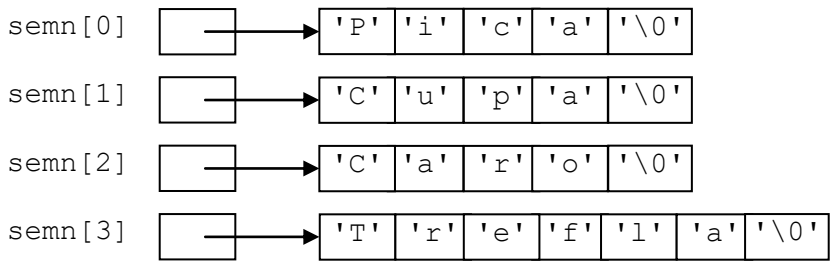
Tablourile pot conţine pointeri. Se pot crea structuri de date care sunt formate din *string*-uri. Fiecare intrare într-un astfel de tablou este un string, dar în C++ un *string* este, de fapt, un pointer la primul său caracter. Astfel, fiecare intrare într-un astfel de tablou este un pointer către primul caracter al unui *string*.

Exemplu

```
const char *semn[4] = {"Pica", "Cupa", "Caro", "Trefla"};
```

Prin această instrucţiune, am declarat un tablou de patru pointeri la `char`.

Vom folosi tabloul `semn` pentru a reprezenta un pachet de cărţi de joc.



Cele patru valori, `"Pica"`, `"Cupa"`, `"Caro"`, `"Trefla"`, sunt păstrate în memoria calculatorului ca şiruri de caractere terminate prin `NULL`. În tabloul `semn` sunt păstrate doar adresele de memorie ale primelor caractere din fiecare şir, iar şirurile au lungimi diferite. Dacă am fi optat pentru varianta unui tablou bidimensional în care fiecare rând reprezintă un tip şi fiecare coloană reprezintă o literă din fiecare tip, ar fi trebuit să fixăm numărul de coloane ale tabloului la dimensiunea maximă pe care o poate avea un şir. Pentru şiruri de dimensiuni mari, memoria neutilizată ar fi fost, astfel, destul de mare.

Vom ilustra folosirea tablourilor de *string*-uri prezentând un program care simulează amestecarea unui pachet de 52 de cărţi de joc şi distribuirea acestora.

Folsim un tablou bidimensional cu 4 linii şi 13 coloane numit `pachet` pentru a reprezenta pachetul de 52 de cărţi de joc.

		As	Doi	Trei	Patru	Cinci	Şase	Şapte	Opt	Nouă	Zece	Valet	Damă	Popă
		0	1	2	3	4	5	6	7	8	9	10	11	12
Pica	0													
Cupa	1													
Caro	2													
Trefla	3													

Linia 0 corespunde semnului "Pica", linia 1 corespunde semnului "Cupa", linia 2 corespunde semnului "Caro", iar linia 3 semnului "Trefla". Coloanele corespund valorilor înscrise pe fiecare carte. Coloanele de la 0 până la 9 sunt asociate cărţilor de la as până la 10, coloana 10 este asociată valeţilor, coloana 11 damelor şi

coloana 12 popilor. Numele semnelor vor fi păstrate în tabloul `semn`, iar valorile cărţilor vor fi stocate în tabloul `valoare`. Celula marcată, `pachet[1][4]`, reprezintă un cinci de cupă.

Acest pachet de cărţi virtual poate fi amestecat astfel: Iniţializăm tabloul pachet cu valori `0`. Alegem apoi la întâmplare o `linie` şi o `coloana`. Inserăm valoarea `1` în celula `pachet[linie][coloana]` pentru a arăta că aceasta este prima carte din pachetul amestecat. Continuăm acest proces inserând aleator valorile `2`, `3` ... `52` în tabloul `pachet` pentru a arăta care carte se va găsi pe poziţia `2`, `3` ... `52`. Pentru că tabloul `pachet` se umple progresiv cu valori, este posibil ca în timpul derulării acestui algoritm o carte să fie selectată din nou. În acest caz, selecţia este ignorată şi se alege un nou `linie` si o noua `coloana` până când este găsită o carte care nu a fost selectată.

Algoritmul de amestecare a pachetului de cărţi are dezavantajul că poate să se deruleze pentru o perioadă nedefinită de timp dacă este aleasă în mod repetat o carte care a fost deja selectată.

Pentru a împărţi prima carte, căutăm în tabloul `pachet` celula `pachet[linie][coloana]` care conţine valoarea `1`. Vom afişa, aşadar, numele cărţii alese care va fi format din `semn[linie]` şi `valoare[coloana]`.

Algoritmul de implementare a soluţiei acestei probleme este următorul:

Iniţializarea tabloului `semn`
Iniţializarea tabloului `valoare`
Iniţializarea tabloului `pachet`
Pentru fiecare dintre cele 52 de cărţi
    Alege aleator o poziţie din tabloul `pachet`

    Atăta timp cât poziţia a fost deja aleasă
        Alege aleator o poziţie din tabloul `pachet`

    Înregistrează numărul de ordine al cărţii în poziţia din tabloul `pachet`

Pentru fiecare dintre cele 52 de cărţi
    Pentru fiecare poziţie din tabloul `pachet`
        Dacă celula curentă din tabloul `pachet` *conţine valoarea corectă*
            Tipăreşte numele cărţii

Programul care implementează această problemă este următorul:

```
#include <iostream>
using std::cout;
using std::ios;
#include <iomanip>
using std::setw;

void Amesteca(int[][13]);
void Imparte(const int[][13], const char *[], const char *[]);

int main()
{
     const char *semn[4] = {"Pica", "Cupa", "Caro", "Trefla"};
```

```
     const char *valoare[13] =
          {"As", "Doi", "Trei", "Patru", "Cinci",
           "Sase", "Sapte", "Opt", "Noua", "Zece",
           "Valet", "Dama", "Popa"};
     int pachet[4][13] = {0};

     srand(time(0));

     Amesteca(pachet);
     Imparte(pachet, semn, valoare);

     return 0;
}

void Amesteca(int lPachet[][13])
{
     int linie, coloana;
     for(int carte=1; carte<=52; carte++)
     {
          do{
               linie = rand()%4;
               coloana = rand()%13;
          } while(lPachet[linie][coloana] != 0);

          lPachet[linie][coloana] = carte;
     }
}

void Imparte(const int lPachet[][13], const char *lSemn[],
             const char *lValoare[])
{
     for(int carte=1; carte<=52; carte++)
          for(int linie=0; linie<=3; linie++)
               for(int coloana=0; coloana<=12; coloana++)
                    if(lPachet[linie][coloana] == carte)
                         cout << setw(6) << lValoare[coloana]
                              << " de "
                              << setw(6) << lSemn[linie]
                              << (carte%2==0?'\n':'\t');
}
```

Programul afişează lista cărţilor după ce acestea au fost amestecate:

```
 Patru de   Pica        As de Trefla
  Noua de   Cupa     Valet de   Pica
   Doi de   Pica      Trei de   Pica
   Opt de   Cupa      Zece de   Caro
 Valet de   Caro     Cinci de   Caro
  Popa de   Pica      Trei de   Caro
  Popa de   Cupa       Opt de Trefla
    As de   Cupa      Sase de Trefla
  Popa de   Caro      Dama de   Cupa
    As de   Caro       Opt de   Pica
```

```
 Noua de Trefla     Valet de   Cupa
  Doi de Trefla      Noua de   Pica
Patru de   Caro     Sapte de   Caro
 Trei de Trefla      Sase de   Caro
 Zece de   Pica        As de   Pica
Valet de Trefla      Zece de   Cupa
 Dama de Trefla       Doi de   Caro
Cinci de   Pica     Cinci de   Cupa
Sapte de   Pica       Opt de   Caro
Patru de Trefla     Sapte de   Cupa
Cinci de Trefla      Sase de   Cupa
 Noua de   Caro     Patru de   Cupa
 Dama de   Caro      Sase de   Pica
  Doi de   Cupa      Trei de   Cupa
Sapte de Trefla      Dama de   Pica
 Zece de Trefla      Popa de Trefla
```

Instrucţiunea

```
carte%2==0?'\n':'\t'
```

foloseşte operatorul ternar `?:` care are şablonul sintactic

condiţie `?` *instrucţiune1* `:` *instrucţiune2*

Atunci când condiţia este adevărată, se execută *instrucţiune1*, iar în caz contrar se execută *instrucţiune2*.

12. Template-uri de funcţii. Tratarea excepţiilor

Template-urile (şabloanele) reprezintă una dintre cele mai puternice caracteristici ale limbajului C++. Acestea permit definirea, printr-un singur segment de cod, a unei întregi game de funcţii supraîncărcate – template de funcţie sau a unei întregi serii de clase – template de clasă. În acest capitol vom discuta cazul funcţiilor. Putem scrie generic, de exemplu, un singur template de funcţie pentru ordonarea unui şir de elemente, iar funcţia poate fi apelată pentru valori de diverse tipuri: `int`, `double` etc.

Tratarea excepţiilor este o metodă care permite programatorilor să scrie programe mai robuste, mai clare şi cu un grad mai mare toleranţă la erori. Programele pot fi concepute astfel încât să detecteze şi să trateze erorile (excepţiile) care pot apărea în timpul rulării, evitând astfel întreruperile neplanificate ale aplicaţiilor.

12.1 Template-uri de funcţii

Funcţiile supraîncărcate realizează, de obicei, operaţii *similare* pentru diferite tipuri de date. Dacă operaţiile sunt *identice* pentru toate tipurile de date, acestea pot fi realizate mai compact folosind *template-uri de funcţii*. Programatorul trebuie să scrie doar o singură definiţie de template de funcţie. Bazându-se pe tipurile argumentelor date explicit sau rezultate din apeluri ale acestor funcţii, compilatorul generează funcţii separate în codul obiect pentru a manipula corespunzător fiecare tip de apel. În limbajul C, acest lucru poate fi realizat folosind macrouri create prin directiva de preprocesare `#define`. Este, însă, o metodă care poate produce efecte nedorite şi care nu permite compilatorului să facă verificări de tip.

Toate definiţiile de template-urile de funcţii încep prin cuvântul cheie `template` urmat de lista parametrilor formali de tip ai template-ului de funcţie plasată între < >. Fiecare parametru formal de tip trebuie să fie precedat de cuvântul cheie `class` sau de `typename`.

Exemple

```
template< class T >
template< typename ElementType >
template< class BorderType, class FillType >
```

Parametrii formali de tip din definiţia unui template de funcţie sunt utilizaţi pentru a specifica tipurile argumentelor funcţiei, tipul valorii returnate de funcţie şi pentru a declara variabile în funcţie. Definiţia funcţiei este, astfel, cea a unei funcţii obişnuite. Cuvintele cheie `class` sau `typename` folosite pentru a specifica parametrii de tip ai template-ului au semnificaţia de „orice tip de dată predefinit sau definit prin program”.

Programul de mai jos defineşte funcţia `PrintArray` care tipăreşte componentele unor tablouri.

Exemplu

```
#include <iostream>
using std::cout;
using std::endl;

template< class T >
void PrintArray(const T *array, const int count)
{
  for(int i = 0; i < count; i++)
```

```
    cout << array[i] << " ";
  cout << endl;
}

int main()
{
  const int aCount = 5, bCount = 7, cCount = 6;
  int a[aCount] = {1, 2, 3, 4, 5};
  double b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
  char c[cCount] = "HELLO";

  cout << "Tabloul a contine: " << endl;
  PrintArray(a, aCount);

  cout << "Tabloul b contine: " << endl;
  PrintArray(b, bCount);

  cout << "Tabloul c contine: " << endl;
  PrintArray(c, cCount);

  return 0;
}
```

Acest program afişează:

```
Tabloul a contine:
1 2 3 4 5
Tabloul b contine:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Tabloul c contine:
H E L L O
```

Funcţia template PrintArray declară un singur parametru formal de tip cu numele `T`. Identificatorul `T` poate fi înlocuit cu orice alt identificator valid. Prin `T` se defineşte generic tipul tabloului ale cărui componente urmează să fie tipărite. Atunci când compilatorul detectează în program un apel al funcţiei `PrintArray`, tipul argumentului generic este înlocuit cu tipul parametrului actual şi se creează o funcţie pentru tipărirea tabloului cu acest tip. Noua funcţie este apoi compilată.

În exemplul de mai sus, sunt instanţiate trei funcţii `PrintArray`, cu argumente de tip `int`, `double` şi `char`.

Exemplu

Instanţierea pentru tipul de dată `int` este:

```
void PrintArray(const int *array, const int count)
{
  for(int i = 0; i < count; i++)
    cout << array[i] << " ";
  cout << endl;
}
```

Fiecare parametru formal de tip din definiţia unui template de funcţie apare, în mod normal, cel puţin o dată în lista de parametri ai funcţiei cel puţin o dată. Numele unui parametru formal de tip poate fi folosit o singură dată în lista de parametri ai header-ului unui template.

În programul de mai sus, funcţia `PrintArray` este apelată de trei ori cu argumentele `a` de tip `int*`, `b` de tip `double*` şi `c` de tip `char*` pentru primul parametru. Apelul

```
PrintArray(a, aCount);
```

face ca apariţiile lui `T` în funcţia `PrintArray` să fie transformate în `int` şi compilatorul instanţiază funcţia `PrintArray` pentru `T` de tip `int`. În mod asemănător se întâmplă şi pentru apelurile

```
PrintArray(b, bCount);
PrintArray(c, cCount);
```

În acest program, mecanismul template-urilor face ca programatorul să nu mai fie nevoit să scrie trei funcţii supraîncărcate cu prototipurile

```
void PrintArray(const int *, const int);
void PrintArray(const double *, const int);
void PrintArray(const char *, const int);
```

12.2 Tratarea excepţiilor

Există mai multe tehnici prin care pot fi tratate erorile care apar în timpul rulării unui program. Cea mai folosită este aceea de a introduce în program secvenţe de cod adaptate prevenirii unor posibile situaţii nedorite. Erorile sunt tratate în locul în care apar în program. Avantajul acestei abordări este că persoana care citeşte codul poate vedea modalitatea de prelucrare a erorii în vecinătatea codului şi poate determina dacă metoda de tratare a excepţiei este implementată corect. Dezavantajul este că prin această schemă codul este oarecum „poluat” cu secvenţe de procesare a erorilor şi devine mult mai greu de citit de un programator care este concentrat pe aplicaţia însăşi. Această metodă face codul mult mai greu de înţeles şi de menţinut.

Tratarea excepţiilor în C++ permite programatorilor să înlăture partea de tratare a excepţiilor din secvenţa principală de program. Stilul C++ de tratare a excepţiilor permite interceptarea tuturor excepţiilor sau a tuturor excepţiilor de un anumit tip. Aceasta face programul mult mai robust, reducând probabilitatea de întrerupere neplanificată a programului.

Tratarea excepţiilor se foloseşte în situaţia în care programul poate să îşi continue rularea după ce depăşeşte eroarea care provoacă excepţia.

Tratarea excepţiilor folosind `try`, `throw` şi `catch`

Tratarea excepţiilor în C++ este o metodă care se aplică atunci când funcţia care detectează o eroare nu o şi tratează. Ea doar *generează* sau *aruncă* excepţia (`throw`). Nu se garantează că excepţia va fi şi tratată în afara funcţiei. Pentru aceasta, trebuie specificată o secvenţă de cod care *detectează* sau *prinde* excepţia şi o *tratează*.

Programatorul trebuie să includă într-un *bloc* `try` codul care ar putea genera o eroare generatoare a unei excepţii. Blocul `try` este urmat de unul sau mai multe *blocuri* `catch`. Fiecare bloc `catch` specifică tipul excepţiei pe care o poate detecta şi trata. Dacă excepţia se potriveşte cu tipul parametrului unuia dintre blocurile `catch`, se execută codul acelui `catch`. Dacă nu este identificat niciun bloc catch care să trateze eroarea, se apelează funcţia predefinită `terminate` care la rândul ei apelează în mod implicit funcţia predefinită `abort` pentru întreruperea programului.

Dacă într-un bloc `try` nu se generează nicio excepţie, programul rulează ignorând blocurile `catch` asociate lui.

Exemplu

```
#include <iostream>
#include <stdexcept>
using std::cin;
using std::cout;
using std::endl;
using std::runtime_error;

double Fractie(int numarator, int numitor)
{
  if(numitor == 0)
    throw runtime_error("Numitorul este 0");
  return static_cast<double>(numarator)/numitor;
}

int main()
{
  int numar1, numar2;
  double rezultat;
  cout << "Introduceti doi intregi (EOF pentru a termina): ";

  while(cin >> numar1 >> numar2)
  {
    try
    {
      rezultat = Fractie(numar1, numar2);
      cout << "Valoarea fractiei este: " << rezultat << endl;
    }
    catch(runtime_error e)
    {
      cout << "A aparut urmatoarea exceptie: "
           << e.what() << endl;
    }

    cout << "Introduceti doi intregi (EOF pentru a termina): ";
  }
  cout << endl;
  return 0;
}
```

Un exemplu de rulare a acestui program este următorul:

```
Introduceti doi intregi (EOF pentru a termina): 3 4
Valoarea fractiei este: 0.75
Introduceti doi intregi (EOF pentru a termina): 5 0
A aparut urmatoarea exceptie: Numitorul este 0
Introduceti doi intregi (EOF pentru a termina): CTRL-D
```

Programul de mai sus calculează rezultatul împărţirii a două numere întregi. Dacă numitorul este 0, este declanşată o excepţie şi se semnalează eroarea.

Programul conţine un bloc `try` care cuprinde codul care ar putea genera o excepţie. Operaţia de împărţire a celor două numere este implementată prin funcţia

`Fractie` care generează o excepţie prin instrucţiunea `throw` atunci când numitorul este 0.

Clasa `runtime_error` face parte din biblioteca standard şi este declarată în fişierul header `stdexcept`. Este derivată din clasa de bază `exception` care face parte tot din biblioteca standard, fiind declarată în fişierul header `exception` şi care este folosită pentru a declanşa excepţii în C++. Declanşarea excepţiilor cu ajutorul lui `runtime_error` permite adăugarea unui mesaj care poate fi citit prin metoda `what()`. Limbajul C++ oferă o întreagă gamă de clase derivate din `exception`, destinate diverselor tipuri de excepţii care se pot declanşa într-un program. La rândul său, programatorul poate să îşi definească propriile sale clase pentru declanşarea excepţiilor, derivându-le, spre exemplu, din clasa `exception` sau din clasele derivate din ea.

În exemplul de mai sus, blocul `try` este urmat imediat de un `catch` care conţine codul de tratare a excepţiei de tip `runtime_error`. Atunci când într-un bloc `try` este generată o excepţie, ea va fi tratată de blocul `catch` care este destinat acelui tip particular de excepţie. Aşadar, blocul `catch` din exemplu va trata doar excepţiile de tip `runtime_error`.

Dacă în timpul execuţiei codul din blocul `try` nu generează nicio excepţie, toate blocurile `catch` asociate acestui `try` vor fi ignorate, iar execuţia continuă cu prima instrucţiune care urmează după acestea.

Generarea Aruncarea unei excepţii – `throw`

Cuvântul cheie `throw` se foloseşte pentru a indica faptul că a apărut o excepţie sau că *s-a aruncat o excepție*. Un `throw` specifică, în mod obişnuit, un operand. Operandul lui `throw` poate fi de orice tip. Dacă este un obiect, îl vom numi *obiect excepţie*. Putem folosi, însă, şi obiecte care nu sunt destinate tratării excepţiilor, şi chiar şi expresii de orice tip.

Imediat ce a fost declanţată o excepţie, aceasta va fi detectată de cel mai apropiată secvenţă de cod destinată tratării excepţiei respective. Secvenţele de tratare a excepţiilor generate într-un bloc `try` sunt listate imediat după blocul `try`.

Ca regulă generală pe care este bine să o urmăm atunci când scriem programe, excepţiile trebuie declanşate doar în interiorul blocurilor `try`. O excepţie care apare în afara unui `try` poate conduce la terminarea programului.

Tratarea unei excepţii – `catch`

Tratarea excepţiilor se face prin blocuri `catch`. Fiecare astfel de bloc constă din cuvântul cheie `catch` urmat, între paranteze rotunde, de tipul excepţiei şi, opţional, de un nume de parametru. Între acolade se găseşte codul care tratează excepţia. Atunci când este detectată o excepţie, se execută codul dintre acolade.

Blocul `catch` defineşte un domeniu de accesibilitate propriu. Dacă parametrul are nume, atunci el poate fi invocat în blocul `catch`. Dacă nu are nume, el este specificat numai în scopul potrivirii dintre blocul `catch` şi tipul excepţiei căreia îi este destinat.

O excepţie care nu este detectată apelează în mod implicit funcţia `terminate()` din biblioteca standard care, la rândul ei, termină programul prin apelul funcţiei `abort()`. Programatorul poate schimba comportamentul funcţiei `terminate()` facând ca aceasta să apeleze o altă funcţie. Numele noii funcţii va fi trimis ca parametru funcţiei `set_terminate`.

Un `catch` care este urmat între parantezele rotunde de trei puncte tratează toate excepţiile.

Exemplu

```
catch(...)
```

Această opţiune este plasată, de obicei, ca o ultimă variantă într-o serie de blocuri `catch`. Un bloc `try` urmat de mai multe blocuri `catch` are un comportament similar instrucţiunii `switch` care foloseşte o ramură `case` în funcţie de valoarea expresiei testate.

Generarea unei noi excepţii în `catch`

Uneori, tratarea unei excepţii nu poate fi făcută în blocul catch care a detectat-o şi atunci se poate decide ca excepţia să fie transmisă mai departe, lasând tratarea ei pe seama altei secvenţe de cod (*rethrowing an exception*). Instrucţiunea

```
throw;
```

lansează din nou excepţia.

Exemplu

```
#include <iostream>
using std::cout;
using std::endl;

#include <exception>
using std::exception;

void ThrowException()
{
  //Lanseaza o exceptie si o prinde imediat
  try
  {
    cout << "Functia ThrowException" << endl;
    throw exception(); //Genereaza exceptia
  }
  catch(exception e)
  {
    cout << "Exceptia tratata in ThrowException" << endl;
    throw;
  }
  cout << "Acest text nu este tiparit";
}

int main()
{
  try
  {
    ThrowException();
    cout << "Acest text nu este tiparit";
  }
  catch(exception e)
  {
    cout << "Exceptia tratata in main" << endl;
  }
```

```
  cout << "Programul continua dupa catch in main" << endl;
  return 0;
}
```

Acest program afişează:

```
Functia ThrowException
Exceptia tratata in ThrowException
Exceptia tratata in main
Programul continua dupa catch in main
```

Funcţia `ThrowException` este apelată în blocul `try` din `main`. În blocul `try` din funcţia `ThrowException` este generată o excepţie de tip `exception` care este detectată imediat de blocul `catch` asociat acestui `try`. Aici, excepţia este relansată, fiind tratată în blocul `catch` din `main`.

Specificarea excepţiilor

Lista excepţiilor care pot fi generate de o funcţie se poate specifica astfel:

```
int g(double h) throw(a, b, c)
{
  //corpul functiei
}
```

Prin această listă se restrânge gama excepţiilor care pot fi declanşate de funcţia `g` la tipurile `a`, `b`, `c`. Dacă funcţia generează o altă excepţie decât cele listate, se apelează automat funcţia `unexpected` din biblioteca standard care, la rândul ei apelează funcţia `terminate`. Comportamenul funcţiei `unexpected` poate fi modificat asemănător modului în care se intervine asupra funcţiei `terminate`, dar apelând, de data aceasta, funcţia `set_unexpected`.

O funcţie pentru care nu există o astfel de listă poate genera orice excepţie.

Bibliografie

[1] Nell Dale, Chip Weems – „Programming And Problem Solving With C++: Comprehensive”, Jones & Bartlett Learning; 6th edition, 2013.

[2] Paul Deitel, Harvey Deitel – „C++ How to Program”, 6th Edition, Pearson Education, 2007.

[3] Paul Deitel, Harvey Deitel – „C++ How to Program”, 10th Edition, Pearson, 2016.

[4] Stanley B. Lippman, Josée Lajoie, Barbara E. Moo – „C++ Primer”, 5th Edition, Addison-Wesley Professional, 2012.

[5] Bjarne Stroustrup – „Programming: Principles and Practice Using C++”, 2nd Edition, Addison-Wesley Professional, 2014.