

Mariana Miloșescu

Informatică

C++

**Filiera teoretică, profilul real,
specializarea matematică - informatică
și
filiera vocațională, profilul militar M.Ap.N.,
specializarea matematică - informatică**

Manual pentru clasa a XI-a



EDITURA DIDACTICĂ ȘI PEDAGOGICĂ S.A.

1. Tehnici de programare	3
1.1. Subprogramele	3
1.1.1. Definiția subprogramului	3
1.1.1.1. Necesitatea folosirii subprogramelelor	4
1.1.1.2. Terminologie folosită pentru subprograme	5
1.1.1.3. Avantajele folosirii subprogramelelor	6
1.1.2. Parametrii de comunicare	6
1.1.3. Elementele subprogramului	7
1.1.4. Clasificarea subprogramelelor	8
1.1.4.1. Clasificarea în funcție de modalitatea de apel	8
1.1.4.2. Clasificarea în funcție de autor	10
1.1.5. Reguli pentru construirea subprogramelelor C++	11
1.1.5.1. Definiția subprogramului	11
1.1.5.2. Prototipul subprogramului	13
1.1.5.3. Activarea (apelul) subprogramului	13
1.1.5.4. Parametrii de comunicare	14
1.1.5.5. Utilizarea stivei de către subprograme	16
1.1.6. Transferul de parametri între subprograme	18
1.1.7. Clasificarea variabilelor de memorie	22
1.1.7.1. Durata de viață a variabilelor de memorie	23
1.1.7.2. Domeniul de vizibilitate al identificatorilor	23
1.1.8. Alegerea modului de implementare a subprogramului	27
1.1.9. Tablourile de memorie și subprogramele	31
1.1.10. Dezvoltarea programelor	32
1.1.11. Subprogramele de sistem	38
Evaluare	39
1.2. Recursivitatea	46
1.2.1. Definiția procesului recursiv	46
1.2.2. Reguli pentru construirea unui subprogram recursiv	50
1.2.3. Variabilele locale și subprogramele recursive	51
1.2.4. Implementarea recursivă a algoritmilor elementari	53
1.2.4.1. Algoritmul pentru determinarea valorii minime (maxime)	53
1.2.4.2. Algoritmul pentru calculul c.m.m.d.c. a două numere	53
1.2.4.3. Algoritmi pentru prelucrarea cifrelor unui număr	54
1.2.4.4. Algoritmul pentru verificarea unui număr dacă este prim	56
1.2.4.5. Algoritmi pentru determinarea divizorilor unui număr	58
1.2.4.6. Algoritmi pentru conversia între baze de numerație	59
1.2.5. Implementarea recursivă a algoritmilor pentru prelucrarea tablourilor de memorie	59
1.2.6. Recursivitatea în cascadă	64
1.2.7. Recursivitatea directă și indirectă	66
1.2.8. Avantajele și dezavantajele recursivității	68
Evaluare	71
1.3. Analiza algoritmilor	77
1.4. Metode de construire a algoritmilor	79
1.5. Metoda Backtracking	80
1.5.1. Descrierea metodei Backtracking	80
1.5.2. Implementarea metodei Backtracking	84
1.5.3. Probleme rezolvabile prin metoda Backtracking	88

1.5.3.1. Generarea permutărilor	88
1.5.3.2. Generarea produsului tensorial	90
1.5.3.3. Generarea aranjamentelor	93
1.5.3.4. Generarea combinațiilor	95
1.5.3.5. Generarea tuturor partițiilor unui număr natural	97
1.5.3.6. Generarea tuturor partițiilor unei mulțimi	100
1.5.3.7. Generarea tuturor funcțiilor surjective	101
1.5.3.8. Problema celor n dame	103

1.6. Metoda „Divide et Impera”

1.6.1. Descrierea metodei „Divide et Impera”	104
1.6.2. Implementarea metodei „Divide et Impera”	105
1.6.3. Căutarea binară	113
1.6.4. Sortarea prin interclasare (MergeSort)	115
1.6.5. Sortarea rapidă (QuickSort)	116
1.6.6. Problema turnurilor din Hanoi	119

Evaluare

2. Implementarea structurilor de date

2.1. Datele prelucrate de algoritmi

2.2. Tabloul de memorie bidimensional (matricea)

2.2.1. Implementarea tabloului bidimensional în C++	132
2.2.2. Algoritmi pentru prelucrarea tablourilor bidimensionale	133
2.2.2.1. Algoritmi pentru parcurgerea elementelor unei matrice	133
2.2.2.2. Algoritmi pentru prelucrarea matricelor pătrate	139

Evaluare

2.3. Șirul de caractere

2.3.1. Implementarea șirului de caractere în limbajul C++	146
2.3.2. Citirea și scrierea șirurilor de caractere	148
2.3.3. Algoritmi pentru prelucrarea șirurilor de caractere	151
2.3.3.1. Prelucrarea a două șiruri de caractere	154
2.3.3.2. Prelucrarea unui șir de caractere	158
2.3.3.3. Prelucrarea subșirurilor de caractere	163
2.3.3.4. Conversii între tipul șir de caractere și tipuri numerice	173

Evaluare

2.4. Înregistrarea

2.4.1. Implementarea înregistrării în limbajul C++	183
2.4.1.1. Declararea variabilei de tip înregistrare	183
2.4.1.2. Accesul la câmpurile înregistrării	185
2.4.2. Înregistrări imbricate	187
2.4.3. Tablouri de înregistrări	192

Evaluare

2.5. Lista

2.5.1. Implementarea listelor în limbajul C++	200
2.5.1.1. Implementarea prin alocare secvențială	201
2.5.1.2. Implementarea prin alocare înlănțuită	201
2.5.2. Clasificarea listelor	204
2.5.3. Algoritmi pentru prelucrarea listelor generale	205
2.5.3.1. Inițializarea listei	206
2.5.3.2. Alocarea memoriei	206
2.5.3.3. Crearea listei	207
2.5.3.4. Adăugarea primului nod la listă	207

2.6.3.5. Adăugarea unui nod la listă	207
2.5.3.6. Parcurgerea listei	210
2.5.3.7. Căutarea unui nod în listă	210
2.5.3.8. Căutarea succesivului și a predecesorului unui nod	211
2.5.3.9. Eliminarea unui nod din listă	211
2.5.3.10. Prelucrarea listelor	213
2.5.4. Algoritmi pentru prelucrarea stivelor	222
2.5.4.1. Inițializarea stivei	224
2.5.4.2. Adăugarea unui nod la stivă	224
2.5.4.3. Extragerea unui nod din stivă	224
2.5.4.3. Prelucrarea stivei	224
2.5.5. Algoritmi pentru prelucrarea cozilor	227
2.5.5.1. Inițializarea cozii	230
2.5.5.2. Adăugarea unui nod la coadă	230
2.5.5.3. Extragerea unui nod din coadă	230
2.5.5.4. Prelucrarea cozii	231

Evaluare	232
-----------------------	------------

2.6. Graful **236**

2.6.1. Definiția matematică a grafului	236
2.6.2. Graful neorientat	238
2.6.2.1. Terminologie	238
2.6.2.2. Gradul unui nod al grafului neorientat	240
2.6.3. Graful orientat	242
2.6.3.1. Terminologie	242
2.6.3.2. Gradele unui nod al grafului orientat	244
2.6.4. Reprezentarea și implementarea grafului	246
2.6.4.1. Reprezentarea prin matricea de adiacență	246
2.6.4.2. Reprezentarea prin lista muchiilor (arcelor)	252
2.6.4.3. Reprezentarea prin lista de adiacență (listele vecinilor)	256
2.6.5. Grafuri speciale	261
2.6.5.1. Graful nul	261
2.6.5.2. Graful complet	261
2.6.6. Grafuri derivate dintr-un graf	263
2.6.6.1. Graful parțial	264
2.6.6.2. Subgraful	266
2.6.7. Conexitatea grafurilor	269
2.6.7.1. Lanțul	269
2.6.7.2. Ciclul	273
2.6.7.3. Drumul	276
2.6.7.4. Circuitul	277
2.6.7.5. Graful conex	278
2.6.7.6. Graful tare conex	282
2.6.8. Parcurgerea unui graf	286
2.6.8.1. Parcurgerea în lățime – Breadth First	286
2.6.8.2. Parcurgerea în adâncime – Depth First	289

2.7. Arborele **293**

2.7.1. Definiția arborelui	293
2.7.2. Arborele parțial	295

Evaluare	296
-----------------------	------------

Anexă	306
--------------------	------------

1. Tehnici de programare

1.1. Subprogramele

1.1.1. Definiția subprogramului

Știți deja că **blocul** este unitatea de bază a oricărui program C++ și că este încapsulat într-o instrucțiune compusă, delimitată de caracterele `{ ... }`. El este format din două părți:

→ **Partea declarativă** conține definiții de elemente necesare algoritmului pentru rezolvarea problemei: constante (**const**), variabile de memorie și tipuri de date (**typedef**). Definirea lor se face cu ajutorul **instrucțiunilor declarative**.

→ **Partea executabilă** sau **partea procedurală** conține instrucțiunile care descriu pașii algoritmului care trebuie implementat pentru rezolvarea problemei. Aceste instrucțiuni se numesc **instrucțiuni imperative**. Ele sunt: **instrucțiunea expresie** (prin care se evaluează o expresie) și **instrucțiunile de control** (prin care se modifică ordinea de execuție a programului). Instrucțiunea expresie prin care se atribuie unei variabile de memorie o valoare se mai numește și **instrucțiune de atribuire**, iar instrucțiunea expresie prin care se cere execuția unui subprogram se mai numește și **instrucțiune procedurală**.

Mai știți că în limbajul C++ blocurile sunt încapsulate în **funcții**, **orice program C++ fiind o colecție de definiții de variabile și funcții**. Funcția este un bloc precedat de un **antet** prin care se precizează numele ei și, dacă este cazul, tipul rezultatului pe care-l întoarce prin chiar numele său și, eventual, parametri de execuție (valori care se transmit blocului și care sunt necesare atunci când se execută blocul):

```
<tip_rezultat> <nume_funcție>(<listă_parametri>)
```

Una dintre funcțiile programului C++ este **funcția rădăcină**. Ea este obligatorie și este primul bloc cu care începe execuția programului. Numele său este **main**. Antetul acestei funcții este:

```
void main()
```

ceea ce semnifică faptul că funcția nu întoarce nici un rezultat (**void**) și nu necesită parametri pentru apelare – parantezele `()` nu conțin listă de parametri.

Studiu de caz

Scop: exemplificarea structurii unui program C++.

Enunțul problemei: Se consideră trei numere naturale, **a**, **b** și **c**. Să se verifice dacă pot forma o progresie aritmetică.

Se vor executa următorii pași:

Pas1. Se ordonează crescător cele trei numere (se schimbă valorile între ele, astfel încât să se respecte relația de ordine $a \leq b \leq c$).

Pas2. Dacă între valorile celor trei variabile există relația $b = (a+c) / 2$, atunci cele trei numere formează o progresie aritmetică.

partea executabilă

```
#include <iostream.h>

void main() → antetul funcției
{
  int a,b,c,x; } → partea declarativă

  cout<<"a= "; cin>>a;
  cout<<"b= "; cin>>b;
  cout<<"c= "; cin>>c;

  if (a>b) {x=a;
           a=b;
           b=x;}
  if (b>c) {x=b;
           b=c;
           c=x;};
  if (a>b) {x=a;
           a=b;
           b=x;};

  if (b==(a+c)/2.)
    cout<<a<<" "<<b<<" "<<c<<"sunt in progresie aritmetica";
  else
    cout<<a<<" "<<b<<" "<<c<<"nu sunt in progresie aritmetica";
}
```

Secvență de instrucțiuni care se execută repetat de trei ori, de fiecare dată cu alte date de intrare:

- 1) a și b
- 2) b și c
- 3) a și b



Prin definiție:

Subprogramul este o secvență de instrucțiuni care rezolvă o anumită sarcină și care poate fi descrisă separat de blocul rădăcină și lansată în execuție din cadrul unui bloc, ori de câte ori este nevoie.

În limbajul C++, subprogramele se mai numesc și funcții.

1.1.1.1. Necesitatea folosirii subprogramelor

În practica programării pot să apară următoarele cazuri:

- O secvență de instrucțiuni se repetă** de mai multe ori în cadrul unui program (ca în programul C++ din exemplul anterior). Secvența de instrucțiuni care se repetă poate fi implementată sub forma unui subprogram.
- Rezolvarea unei anumite sarcini este necesară în mai multe programe**, ca, de exemplu, diferite operații matematice (extragerea radicalului, extragerea părții întregi sau a părții fracționare dintr-un număr real, ridicarea unui număr la o putere etc.), diferite operații cu tablouri de memorie (crearea, parcurgerea și sortarea tabloului de memorie, ștergerea sau inserarea unui element etc.), diferite operații cu fișiere (deschiderea unui fișier, închiderea unui fișier, testarea sfârșitului de fișier etc.). Secvența de instrucțiuni care rezolvă o anumită sarcină ce poate să apară în mai multe programe poate fi implementată cu ajutorul unui subprogram.
- Orice problemă poate fi descompusă în subprobleme**. Subproblemele în care este descompusă se numesc **module**. Descompunerea poate continua până când se obține

un modul cu rezolvare imediată. Această metodă de rezolvare a unei probleme se numește tehnica **top-down** de proiectare a algoritmilor. Ea este foarte utilă în cazul programelor care trebuie să rezolve probleme complexe (de exemplu, prelucrarea vectorilor). În aceste cazuri se obțin programe foarte mari și complexe. Pentru a obține programe mai simple și mai clare, se poate descompune problema inițială în subprobleme, fiecare subproblemă fiind descrisă printr-un subprogram.

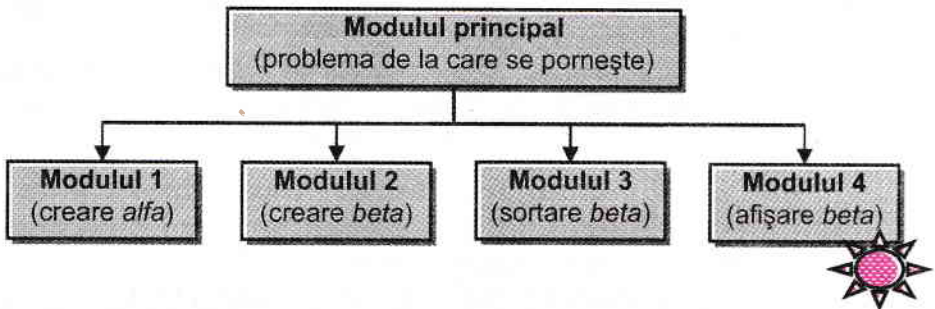
Studiu de caz

Scop: exemplificarea modului în care o problemă poate fi descompusă în subprobleme folosind **tehnica top-down**.

Enunțul problemei: Se introduc de la tastatură mai multe numere întregi, într-un vector **alfa**. Să se transfere în vectorul **beta** elementele pozitive din **alfa** și apoi să se afișeze elementele vectorului **beta**, ordonate crescător.

Problema poate fi împărțită în patru subprobleme (**module**):

- crearea vectorului *alfa*, prin introducerea valorilor de la tastatură;
- crearea vectorului *beta*, prin copierea valorilor pozitive din vectorul *alfa*;
- sortarea vectorului *beta*;
- afișarea elementelor vectorului *beta*.



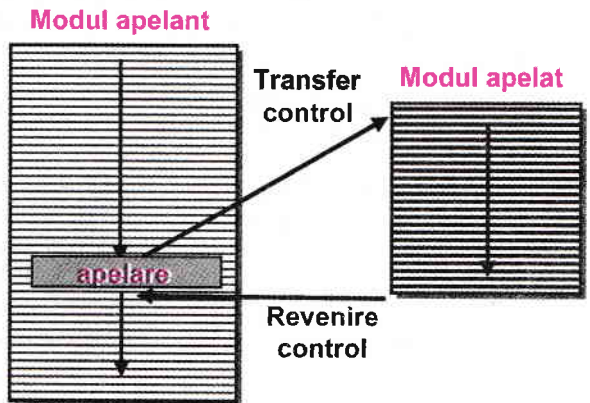
Așadar, în toate cele trei cazuri prezentate, **soluția o reprezintă subprogramele**.

1.1.1.2. Terminologie folosită pentru subprograme

Într-o structură modulară în care fiecare modul este descris printr-un subprogram, modulele se clasifică astfel:

→ **Modul apelant.** Este modulul care, pentru rezolvarea propriei probleme, apelează la alte module, fiecare dintre ele rezolvând o anumită subproblemă. La apelare, el transferă controlul modulului apelat. În exemplul anterior, *Modulul principal* este modulul apelant.

→ **Modul apelat.** Este un modul apelat de un alt modul, pentru a-i rezolva o subproblemă. După ce își termină execuția, el redă controlul modulului apelant. În exemplul anterior, *Modulul 1*, *Modulul 2* și *Modulul 3* sunt module apelate.



Vom considera funcția rădăcină `main()` ca fiind **modulul principal** sau **programul principal**, iar celelalte funcții (module) pe care le vom defini le vom numi **subprograme**.

1.1.1.3. Avantajele folosirii subprogramelor

În practică, pentru rezolvarea unor probleme complexe care ajută la îndeplinirea unor activități, cum sunt de exemplu prelucrările de texte, contabilitatea unei întreprinderi, inventarierea unor depozite de materiale, gestionarea unei biblioteci etc., trebuie să se conceapă programe sofisticate numite **aplicații**. În construirea unei aplicații, folosirea subprogramelor oferă următoarele avantaje:

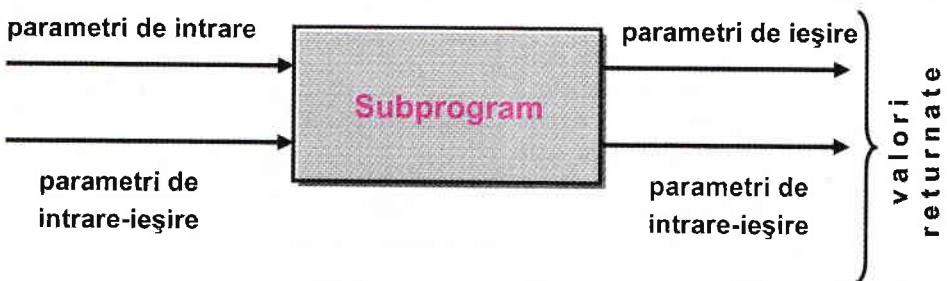
- **Se face economie de memorie internă.** Un grup de instrucțiuni care trebuie să se execute de mai multe ori într-o aplicație (chiar cu date de intrare și de ieșire diferite) se va scrie o singură dată într-un subprogram și se va executa prin apelarea subprogramului ori de câte ori este nevoie.
- **Se favorizează lucrul în echipă** pentru aplicațiile mari. Fiecare programator va putea să scrie mai multe subprograme, independent de ceilalți programatori din echipă. Pentru a realiza subprogramul, este suficient să i se precizeze programatorului specificațiile subprogramului: datele de intrare, datele de ieșire și problema pe care trebuie să o rezolve.
- **Depanarea și actualizarea aplicației se fac mai ușor.** După implementare și intrarea în exploatare curentă, o aplicație poate necesita modificări, ca urmare a schimbării unor cerințe. Este mult mai simplu să se gândească modificarea la nivelul unui subprogram, decât la nivelul întregii aplicații.
- **Crește portabilitatea programelor.** Subprogramele sunt concepute independent de restul aplicației și unele dintre ele pot fi preluate, fără un efort prea mare, și în alte aplicații, în care trebuie să fie rezolvate sarcini similare.

1.1.2. Parametrii de comunicare

Pe de o parte, subprogramul nu este o entitate independentă. El trebuie asamblat în interiorul programului, adică trebuie **stabilite legături între modulul apelant și modulul apelat**.

Pe de altă parte, în procesul de prelucrare dintr-un modul sunt necesare date care trebuie prelucrate (date de intrare) și care uneori trebuie să fie preluate din modulul apelant. La rândul său, în urma prelucrărilor, modulul apelat furnizează rezultate (date de ieșire) către modulul care l-a apelat. Datele care circulă astfel între module se numesc **parametri de comunicare**. Așadar:

Parametrii de comunicare se folosesc pentru a realiza legătura dintre module.



După modul în care intervin în comunicarea cu modulul apelant, parametrii de comunicare se clasifică în:

- **Parametrii de intrare.** Sunt date care urmează să fie prelucrate de subprogram (**SP**) și care îi sunt comunicate de către modulul apelant (**P**). Subprogramul le primește în momentul activării: **P** ⇒ **SP**.
- **Parametrii de ieșire.** Sunt rezultate obținute de subprogram în urma prelucrării și pe care le comunică modulului apelant. Modulul apelant le primește după ce subprogramul își termină execuția: **SP** ⇒ **P**.
- **Parametrii de intrare-ieșire.** Sunt date care participă la calculul datelor de ieșire și sunt accesibile atât modulului apelant, cât și modulului apelat. Valoarea lor poate fi modificată atât de subprogram, cât și de modulul apelant. Subprogramul le primește la activare, iar modulul apelant le primește după ce subprogramul își termină execuția: **P** ⇔ **SP**.

Parametrii de ieșire și parametrii de intrare-ieșire prin care subprogramul transmite rezultatele modulului apelant se mai numesc și **valori returnate** de către subprogram. La apelarea subprogramelor, parametrii de intrare pot fi și constante sau expresii, iar parametrii de ieșire și parametrii de intrare-ieșire pot fi numai variabile de memorie.

1.1.3. Elementele subprogramului

În limbajul C++ există trei elemente implicate în utilizarea unui subprogram:

- **definiția subprogramului** – conține numele subprogramului, tipul argumentelor și al valorilor returnate și specifică ceea ce trebuie să realizeze subprogramul;
- **prototipul subprogramului** – comunică compilatorului informații despre subprogram (modul în care se poate apela subprogramul);
- **apelul subprogramului** – execută subprogramul.

```

#include<iostream.h>
void scie (); ← Prototipul subprogramului
void main()
{scie (); ← Apelul subprogramului
}

void scie () ← Antetul subprogramului
{cout<<"Subprogram"; } ← Corpul subprogramului
    
```

Modul apelant
main ()

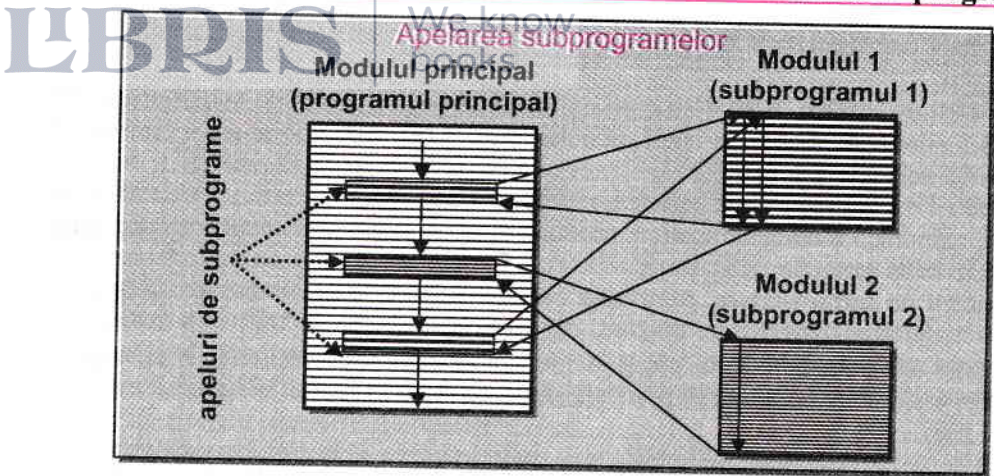
↓

Modul apelat
scie ()

Subprogramul se poate identifica printr-un **nume** care este folosit atât pentru **definiția subprogramului**, cât și pentru **prototip** și **activarea** lui (apelarea lui).

Apelarea subprogramului în cadrul unui bloc înseamnă **activarea subprogramului**, adică lansarea lui în execuție. Subprogramul poate fi apelat ori de câte ori este nevoie (nu există restricții pentru numărul de apeluri).

Modulul apelant se execută secvențial (instrucțiune cu instrucțiune). La apelarea subprogramului, este părăsit blocul modulului apelant și se trece la executarea instrucțiunilor din subprogram. După ce se termină executarea acestor instrucțiuni, se revine la blocul apelant și se continuă execuția cu instrucțiunea care urmează apelului.



1.1.4. Clasificarea subprogramelor

Pentru clasificarea subprogramelor se pot folosi două criterii:

- modalitatea de apel, determinat de modul de returnare a valorilor rezultate;
- autorul.

1.1.4.1. Clasificarea în funcție de modalitatea de apel

Subprogramele se împart în:

- subprograme apelate ca instrucțiuni procedurale;
- subprograme apelate ca operanzi.

Deoarece, în limbajul C++, toate subprogramele, indiferent de modul în care sunt apelate, se numesc **funcții**, pentru a identifica cele două tipuri de subprograme, le vom denumi ca funcții procedurale și funcții operand.

Funcții procedurale

Funcția procedurală este subprogramul care returnează una, mai multe sau nici o valoare. Valorile se returnează prin **intermediul parametrilor**.

Modalitatea de apel. Subprogramul se apelează printr-o **instrucțiune procedurală** care are următoarea sintaxă (o instrucțiune expresie care are un singur operand – apelul subprogramului):

`nume_subprogram (listă_parametri);`

} optional

Observații:

1. În listă, parametrii sunt separați prin virgulă.
2. Parametrii pot fi nume de variabile de memorie, expresii sau valori constante. Ei se mai numesc și **argumentele** funcției.

Exemple de apeluri de funcții procedurale implementate în limbajul C++:

`clrscr();`

Apelul unei funcții procedurale fără parametri (**CLear SCREEN**) care șterge informațiile afișate pe ecranul calculatorului.

randomize () ;

Apelul unei funcții procedurale fără parametri care inițializează generatorul de numere aleatoare.

swab (s1, s2, n) ;

Apelul unei funcții procedurale cu trei parametri: copiază **n** caractere (**n** fiind un număr par), din șirul de caractere **s1**, la începutul șirului de caractere **s2**, inversând caracterele adiacente. Parametrul **s2** este un parametru de intrare-ieșire, iar parametrii **s1** și **n** sunt parametri de intrare.

gotoxy (x, y) ;

Apelul unei funcții procedurale cu doi parametri: în modul de lucru text, mută cursorul în fereastra de text curentă, în poziția precizată prin coordonatele **x** și **y**. Parametrii **x** și **y** sunt parametri de intrare.

Funcții operanzi

Funcția operand este un subprogram care returnează un rezultat prin chiar **numele său**, și eventual și alte rezultate, prin intermediul parametrilor.

Modalitatea de apel. Subprogramul se activează în interiorul unei expresii unde este folosit ca operand. Expresia poate să apară fie în membrul drept al unei instrucțiuni de atribuire, fie în cadrul unei instrucțiuni de control, fie în lista de parametri ai unei alte funcții (funcție operand sau instrucțiune procedurală). De exemplu:

```
nume_expresie = nume_variabilă +|-|*|/ nume_funcție (listă_parametri);
```

opțional

Observații:

1. La fel ca și la subprogramele apelate ca instrucțiuni procedurale, parametrii din listă sunt separați prin virgulă și pot fi nume de variabile de memorie, expresii sau valori constante.
2. În cazul funcției operand care returnează un singur rezultat prin chiar numele ei, parametrii din lista de parametri sunt de obicei numai parametri de intrare.
3. Funcția operand poate fi apelată și ca o funcție procedurală. În acest caz se pierde valoarea returnată prin numele ei.

Exemple de apeluri de funcții operand implementate în limbajul C++:

```
x=3.5; e=5+floor(x);
```

La calculul expresiei care se atribuie variabilei **e** se activează funcția **floor ()** prin care se determină cel mai mare întreg mai mic decât valoarea parametrului. Funcția are un singur parametru – **x**, care este parametru de intrare și are valoarea 3.5. Rezultatul (data de ieșire) este furnizat prin numele funcției și are valoarea 3. Așadar, variabilei de memorie **e** i se va atribui valoarea: 8 (5+3).

```
for (i=0; i<=sqrt(n); i++);
```

La calculul expresiei ce se atribuie valorii finale a contorului structurii repetitive **for**, se activează funcția **sqrt (n)** care furnizează radicalul de ordinul 2 din valoarea parametrului. Funcția are un singur parametru – **n**, care este parametru de intrare.

```
x=sqrt(pow(3,2)+pow(4,2));
```

La calculul expresiei care se atribuie variabilei **x** se activează de două ori funcția **pow()**: o dată pentru a calcula 3 la puterea 2, returnând valoarea 9, și o dată pentru a calcula 4 la puterea 2, returnând valoarea 16. Funcția **pow()** are doi parametri de intrare: primul este baza, iar al doilea este exponentul. Rezultatul este furnizat prin numele funcției. Rezultatul obținut prin evaluarea expresiei $9+16 = 25$ va fi parametru de intrare pentru funcția **sqrt()** care extrage radicalul de ordinul 2 din valoarea lui. Rezultatul funcției **sqrt()** – data de ieșire – este furnizat prin numele funcției și are valoarea 5. El este atribuit variabilei de memorie **x**. Așadar variabila de memorie **x** va avea valoarea 5.

1.1.4.2. Clasificarea în funcție de autor

Subprogramele se împart în:

- **subprograme standard** sau **subprograme de sistem**;
- **subprograme nestandard** sau **subprograme utilizator**.

Subprograme de sistem

Sunt subprograme predefinite de autorii limbajului de programare, care sunt furnizate împreună cu limbajul de programare. Ele se găsesc grupate, după funcțiile pe care le realizează, în **bibliotecile limbajului de programare**. Aceste subprograme rezolvă probleme generale ale utilizatorului, ca de exemplu:

- probleme matematice: calculul funcțiilor trigonometrice (*sin()*, *cos()* etc.), calculul unor funcții matematice (radicalul – *sqrt()*, exponențialul – *exp()*, logaritmul – *log()*, puterea – *pow()*), calculul părții întregi și fracționare dintr-un număr real (*modf()*, *fmod()*) etc.
- operații cu fișiere: deschiderea unui fișier – *fopen()*, închiderea unui fișier – *fclose()* etc.

Înainte de apelarea unui astfel de subprogram, trebuie făcut cunoscut compilatorului prototipul subprogramului, prin instrucțiunea pentru preprocesor:

```
#include <nume_fisier_antet.h>;
```

Așadar, lucrul cu un subprogram de sistem presupune două operații:

- **includerea fișierului care conține prototipul subprogramului în fișierul sursă al programului**,
- **apelarea subprogramului**.

Subprograme utilizator

Sunt subprograme create de programator pentru a rezolva unele sarcini (cerințe) specifice aplicației sale. Astfel, în exemplul de program prin care se verifică dacă trei numere pot forma o progresie aritmetică, pentru ordonarea celor trei numere **a**, **b** și **c**, programatorul poate construi un subprogram care să execute secvența de instrucțiuni prin care se realizează interschimbarea a două valori, secvență care se repetă de trei ori în cadrul programului.

Aceste subprograme trebuie declarate sau definite de programator înainte de apelul lor din funcția rădăcină sau dintr-un alt subprogram.

Lucrul cu un subprogram utilizator presupune două operații:

- **definirea subprogramului** și, dacă este cazul, **precizarea prototipului**.
- **apelarea subprogramului**.

1.1.5. Reguli pentru construirea subprogramelor C++

1.1.5.1. Definiția subprogramului

Definiția unui subprogram este formată din antetul și corpul subprogramului:

```

<antetul subprogramului>
{
    <declarații proprii subprogramului>
    <instrucțiuni>
    [return <expresie>;]
}
    
```

- a. **Antetul subprogramului.** Este o linie de recunoaștere a subprogramului, în care i se atribuie un nume. El specifică începutul subprogramului.
- b. **Corpul subprogramului.** La fel ca orice bloc C++, este încapsulat într-o instrucțiune compusă, delimitată de caracterele `{...}` și este format din două părți:
 - **Partea declarativă.** Conține definiții de elemente folosite numai în interiorul subprogramului: tipuri de date, constante și variabile de memorie. Nu se pot defini și alte subprograme (nu este valabilă tehnica de imbricare a subprogramelor existentă în alte limbaje de programare).
 - **Partea executabilă.** Conține instrucțiunile prin care sunt descrise acțiunile realizate de subprogram.

Antetul subprogramului

Subprogramul trebuie să aibă un antet prin care se precizează interfața dintre programul apelant și subprogram. El conține trei categorii de informații:

- **Tipul rezultatului.** Pentru funcțiile operand se precizează tipul rezultatului furnizat de subprogram prin chiar numele său. Pentru funcțiile procedurale, tipul rezultatului este **void** (nu întoarce nici un rezultat prin numele său; rezultatele vor fi întoarse prin parametri subprogramului). Dacă nu se precizează tipul rezultatului, compilatorul va considera că acesta este implicit de tip **int**.
- **Numele subprogramului.** Este un identificator unic, care se atribuie subprogramului. Numele trebuie să respecte aceleași reguli ca orice identificator C++.
- **Parametrii folosiți pentru comunicare.** Pentru fiecare parametru se precizează numele și tipul.

Antetul unui subprogram este de forma:

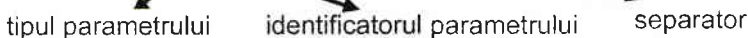
```

tip_rezultat  nume_subprogram (listă_parametri)
    
```

Lista de parametri este de forma:

```

tip1 p1, tip2 p2, tip3 p3, ... tip_n p_n
    
```



Exemplul 1:

```

float alfa(int a, int b, float c)
    
```

Acesta este antetul unei funcții operand care furnizează un rezultat de tip **float**. Numele funcției este *alfa*, iar parametrii folosiți pentru comunicare sunt *a* și *b* de tip **int** și *c* de tip **float**.

Exemplul 2:

```
void beta(int a, float b, float c, char d)
```

Acesta este antetul unei funcții procedurale. Numele funcției este *beta*, iar parametrii folosiți pentru comunicare sunt: *a* de tip **int**, *b* și *c* de tip **float** și *d* de tip **char**.

Exemplul 3:

```
void gama()
```

Acesta este antetul unei funcții procedurale. Numele funcției este *gama* și nu folosește parametri pentru comunicare.

Observații:

1. Separarea parametrilor în listă se face prin caracterul virgulă (,). Dacă există mai mulți parametri de același tip, ei **nu pot pot fi grupați ca la declararea tipului variabilelor de memorie**. Pentru fiecare parametru trebuie precizat tipul.
2. Tipul parametrilor poate fi:
 - orice tip standard al sistemului folosit pentru date elementare – întreg (**int**, **unsigned**, **long**), real (**double**, **float**, **long double**) sau caracter (**char** sau **unsigned char**) –, tipul pointer sau orice tip de structură de date (vector, matrice, șir de caractere sau înregistrare);
 - orice tip definit de utilizator înainte de a defini subprogramul.
3. Pentru rezultatul funcției nu se poate folosi tipul tablou de memorie.

Exemplu:

```
float a[10] tablou(int v, unsigned n)
```

Acest antet de subprogram va produce **eroare** deoarece tipul funcției este tablou de memorie.

4. Numele subprogramului poate fi folosit în trei locuri distincte:
 - în **prototipul subprogramului**, unde are un rol declarativ;
 - în **antetul subprogramului**, unde are un rol de definiție, dar și declarativ;
 - în **apelul subprogramului**, unde are rol de activare.

Corpul subprogramului

Corpul subprogramului este un **bloc** care conține atât instrucțiuni declarative, cât și instrucțiuni imperative. Variabilele de memorie declarate în corpul subprogramului se numesc **variabile locale**. În cazul unei funcții operand, ultima instrucțiune din corpul subprogramului trebuie să fie instrucțiunea **return**, care are sintaxa:

```
return <expresie>;
```

Valoarea obținută prin evaluarea expresiei *<expresie>* va fi atribuită funcției operand (va fi valoarea returnată prin numele funcției). Rezultatul expresiei trebuie să fie de același tip cu tipul funcției sau de un tip care poate fi convertit implicit în tipul funcției.

Atenție

Când compilatorul C++ întâlnește într-un subprogram instrucțiunea **return**, termină execuția subprogramului și redă controlul modulului apelant. Prin urmare, dacă veți scrie în subprogram, după instrucțiunea **return**, alte instrucțiuni, ele nu vor fi executate.

