MINISTERUL EDUCAŢIEI NAŢIONALE

Mariana Miloşescu

Informatică

C++

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

Manual pentru clasa a XI-a

EDITURA DIDACTICĂ ŞI PEDAGOGICĂ S.A.

Cuprins

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, parametrii 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 $a \le b \le c$).

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

```
#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";
}
```

partea executabilă

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:

a. **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.

b. **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.

c. **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.

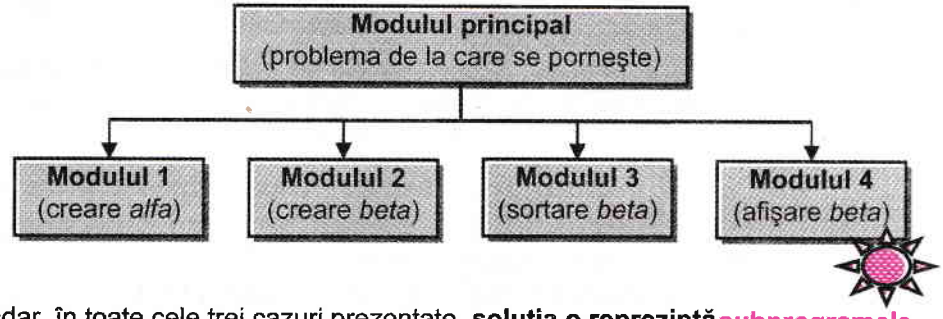
Studiu de caz

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*.



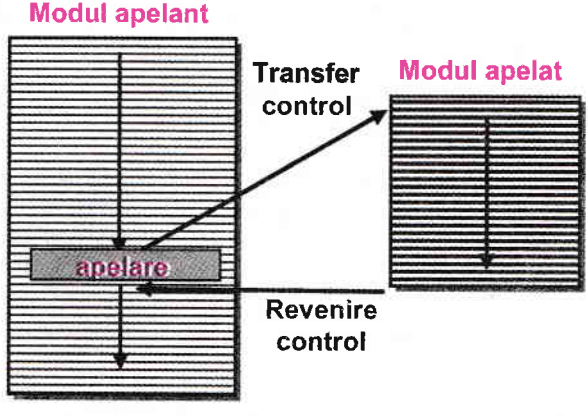
Așadar, în toate cele trei cazuri prezentate, **soluția o reprezintă subprogramele**.

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.

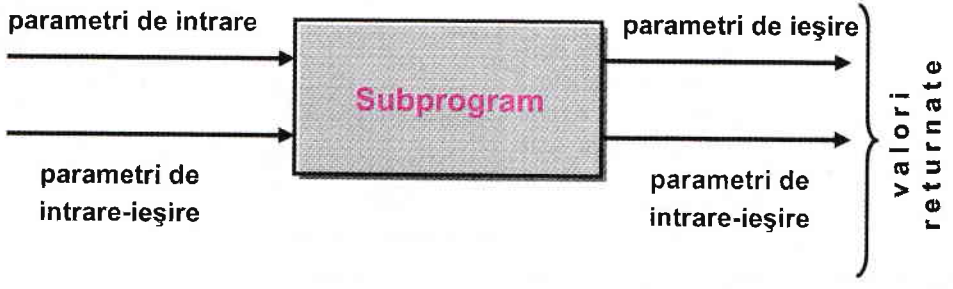
→ **Crește portabilitatea programelor**. Subprogramele sunt concepute independent de restul aplicației și unele dintre ele pot fi preluate, fără un efort prea mare, și în alte aplicații, în care trebuie să fie rezolvate sarcini similare.

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:

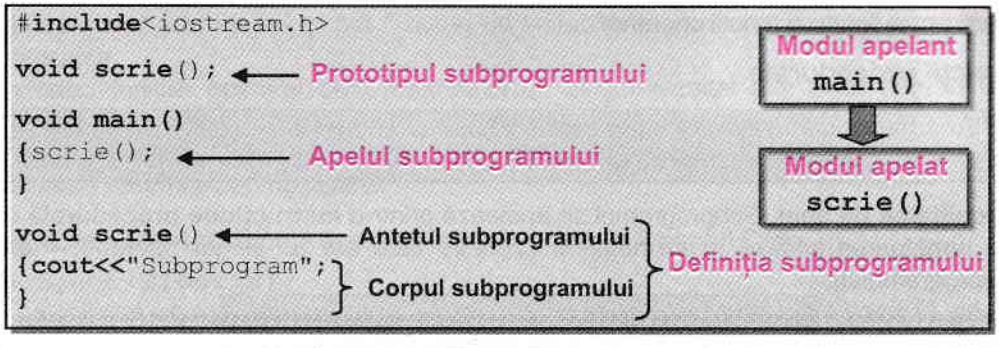
- → **Parametrii de intrare.** Sunt date care urmează să fie prelucrate de subprogram (**SP**) şi care îi sunt comunicate de către modulul apelant (**P**). Subprogramul le primeşte în momentul activării: **P** ⇒ **SP**.
- → **Parametrii de ieşire.** Sunt rezultate obţinute de subprogram în urma prelucrării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

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

1.1.3. Elementele subprogramului

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

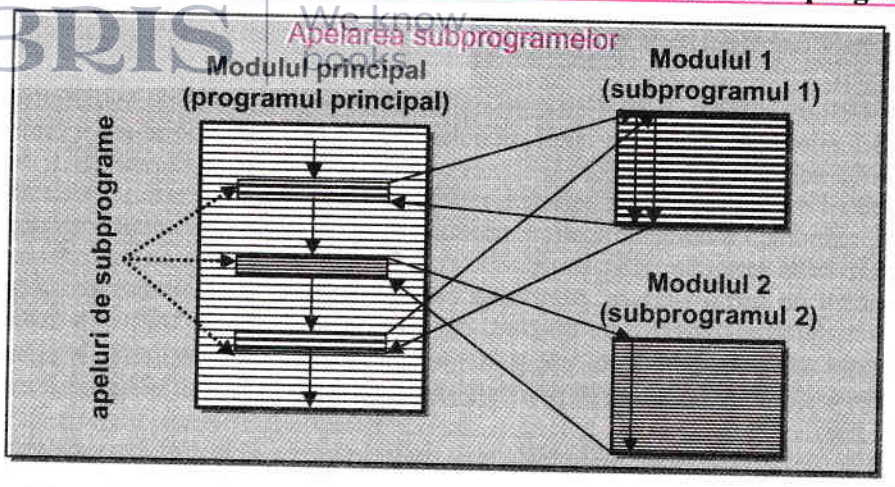
- → **definiţia subprogramului** – conţine numele subprogramului, tipul argumentelor şi al valorilor returnate şi specifică ceea ce trebuie să realizeze subprogramul;
- → **prototipul subprogramului** – comunică compilatorului informaţii despre subprogram (modul în care se poate apela subprogramul);
- → **apelul subprogramului** – execută subprogramul.



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

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

Modulul apelant se execută secvenţial (instrucţiune cu instrucţiune). La apelarea subprogramului, este părăsit blocul modulului apelant şi se trece la executarea instrucţiunilor din subprogram. După ce se termină executarea acestor instrucţiuni, se revine la blocul apelant şi se continuă execuţia cu instrucţiunea 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 şi 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);

opțional

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 şi **argumentele** funcției.

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

`clrscr();`

Apelul unei funcții procedurale fără parametri (**CLeaR SCReen**) care şterge informațiile afişate 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, şi 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 ieşire – este furnizat prin numele funcției şi are valoarea 5. El este atribuit variabilei de memorie **x**. Aşadar 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 şi fracționare dintr-un număr real (*modf()*, *fmod()*) etc.
→ operații cu fişiere: deschiderea unui fişier – *fopen()*, închiderea unui fişier – *fclose()* etc.

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

```
#include <nume fisier antet.h>;
```

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

→ **includerea fişierului care conține prototipul subprogramului în fişierul 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** şi **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** şi, 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 şi corpul subprogramului:

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

a. **Antetul subprogramului.** Este o linie de recunoaştere 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 **{...}** şi este format din două părți:
   → **Partea declarativă.** Conține definiții de elemente folosite numai în interiorul subprogramului: tipuri de date, constante şi variabile de memorie. Nu se pot defini şi 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 şi 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 parametrii 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 aceleaşi reguli ca orice identificator C++.
→ **Parametrii folosiți pentru comunicare.** Pentru fiecare parametru se precizează numele şi 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
```

tipul parametrului — identificatorul parametrului — separator

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* şi *b* de tip **int** şi *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* şi *c* de tip **float** şi *d* de tip **char**.

Exemplul 3:

```
void gama()
```

Acesta este antetul unei funcții procedurale. Numele funcției este *gama* şi nu foloseşte parametri pentru comunicare.

Observații:

1. Separarea parametrilor în listă se face prin caracterul virgulă (,). Dacă există mai mulți parametri de acelaşi 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, şir 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 şi declarativ;
   → în **apelul subprogramului**, unde are rol de activare.

Corpul subprogramului

Corpul subprogramului este un **bloc** care conține atât instrucțiuni declarative, cât şi 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 acelaşi tip cu tipul funcției sau de un tip care poate fi convertit implicit în tipul funcției.

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

1.1.5.2. Prototipul subprogramului

Este o linie de program, aflată înaintea modulului care apelează subprogramul, prin care se comunică compilatorului informații despre subprogram (se **declară** subprogramul). Prin declararea programului, compilatorul primeşte informații despre modul în care se poate apela subprogramul şi poate face verificări la apelurile de subprogram în ceea ce priveşte tipul parametrilor folosiți pentru comunicare şi a modului în care poate face conversia acestor parametri.

Un subprogram, pentru a putea fi folosit, trebuie declarat. Pentru declararea lui se foloseşte prototipul. El conține trei categorii de informații, la fel ca şi antetul subprogramului: **tipul rezultatului**, **numele subprogramului** şi **tipul parametrilor folosiți pentru comunicare.** Pentru fiecare parametru din antetul subprogramului, se poate preciza numai tipul, nu şi numele lui.

Prototipul unui subprogram este de forma:

```
tip_rezultat nume_subprogram (listă_tipuri_parametri);
```

Lista tipului parametrilor este de forma:

```
tip_1, tip_2, tip_3, ... tip_n
```

tipul parametrului — separator

Observații:

1. Separarea tipurilor de parametri în listă se face prin caracterul virgulă (,). Lista trebuie să conțină atâtea tipuri de parametri câți parametri au fost definiți în antetul subprogramului. În listă se precizează tipul de dată la care se referă, în aceeaşi ordine în care au fost scrişi parametrii la definirea lor în antet.
2. Spre deosebire de antetul subprogramului, prototipul se termină cu caracterul ;.

Pentru funcțiile al căror antet a fost precizat anterior, prototipurile vor fi:

Exemplul 1:

```
float alfa(int, int, float);
```

Exemplul 2:

```
void beta(int, float, float, char);
```

Exemplul 3:

```
void gama();
```

1.1.5.3. Activarea (apelul) subprogramului

Subprogramul trebuie să fie cunoscut, atunci când se cere prin apel activarea lui:

→ Dacă subprogramul este **standard**, trebuie **inclus** fişierul care conține prototipul subprogramului în fişierul sursă.

→ Dacă subprogramul este **utilizator**, trebuie **declarat** fie prin folosirea **prototipului**, fie prin **definirea** lui înaintea apelului.

În funcție de modul în care a fost definit, subprogramul se activează fie printr-o instrucțiune procedurală, fie ca operand într-o expresie.

Pentru funcțiile al căror antet a fost precizat anterior, activarea se poate face astfel: