7. Funcții

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

7.1 Funcții `void`

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

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

Scrierea modulelor ca funcții `void`

În principiu, o astfel de funcție arată ca funcția `main`, cu deosebirea că header-ul său foloseşte cuvântul cheie `void` în locul lui `int`. În plus, o funcție `void` nu conține nicio instrucțiune 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 funcții `void`. Acest program tipăreşte textul

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

Iată cum putem schița 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 funcții `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 funcția `main` şi descrierea de la nivelul 0. Cele două funcții care fac parte din corpul funcției `main` sunt *fara parametri*. Fiecare dintre *definițiile* acestor funcții este formată din header-ul funcției urmat de un bloc de instrucțiuni care alcătuiesc corpul funcției. Header-ul unei funcții `void` începe cu cuvântul cheie void care semnalează că nu întoarce nicio valoare. Implicit, în corpul unei astfel de funcții nu va apărea nicio instrucțiune `return` urmată de o valoare. Se poate folosi, totuşi, instrucțiunea

```
return;
```

care încheie execuția unei funcții `void`.

Definițiile funcțiilor pot apărea în orice ordine, deci `main` ar fi putut apărea după cele două funcții. Cele două declarații dinaintea funcției `main` se numesc *prototipuri de funcții*. Ele sunt necesare pentru că regulile din C++ impun declararea unui identificator înaintea folosirii sale. Cei doi identificatori sunt `Tipareste2Linii` şi `Tipareste4Linii` folosiți în `main`.

7.2 Sintaxa şi semantica funcțiilor void

Apelul funcțiilor (invocarea)

Un apel de funcție într-un program înseamnă execuția corpului funcției apelate. Şablonul sintactic apelului unei funcții este următorul:

NumeFuncție(*ListăParametriActuali*);

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

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

Dacă lista conține doi sau mai mulți parametri, aceştia trebuie separați prin virgulă:

Expresie, Expresie ...

În momentul apelului unei funcții, parametrii actuali sunt transmişi parametrilor formali conform poziției lor, de la stânga la dreapta, iar apoi controlul este transferat primei instrucțiuni din corpul funcției. Când se încheie şi execuția ultimei instrucțiuni din funcție, controlul este transmis punctului în care s-a făcut apelul.

Declarații şi definiții de funcții

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

O declarație de funcție anunță compilatorul despre numele funcției, tipul de dată al valorii returnate (poate fi şi void) şi tipurile datelor folosite în lista de parametri. Programul din exemplul de mai sus conține două declarații de funcții care nu sunt însoțite de corpul funcțiilor. Acestea sunt *prototipuri de funcții*. Dacă declarațiile sunt însoțite şi de corpul funcției, atunci este vorba despre *definiții de funcții*. Toate definițiile sunt declarații, dar nu toate declarațiile sunt definiții.

Prototipuri de funcții

Pentru a satisface cerința ca orice identificator să fie declarat înainte de a fi folosit, programatorii C++ plasează prototipurile funcțiilor chiar înaintea definiției funcției `main`, la începutul programului.

Prototipul de funcție de mai numeşte în unele limbaje şi *declarație forward*. Pentru o funcție `void`, şablonul sintactic al unei declarații este:

`void` *NumeFuncție*`(`*ListăParametriFormali*`);`

Prototipul nu este însoțit de corpul funcției, lista de parametri formali este opțională şi declarația se încheie cu `;`

Lista parametrilor formali este opțională şi are următoarea formă:

TipData & NumeVariabilă, TipDată & NumeVariabilă ...

Ampersantul `&` este ataşat tipului de dată şi este opțional. Într-un prototip de funcție, 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 funcției, dar compilatorul le ignoră.

Definiții de funcții

Aşa cum am văzut deja, definiția unei funcții constă din header-ul funcției şi corpul funcției care este, de fapt, un bloc. Şablonul sintactic al unei definiții de funcție este:

```
void NumeFuncție(ListăParametriFormali)
{
   Instrucțiune
   ...
}
```

Header-ul funcției nu se încheie cu `;` ca la prototipurile de funcții.

Sintaxa listei de parametri din definiție 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 funcții este un bloc, orice funcție poate include declarații 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 funcțiilor şi accesibile lor şi se numesc *variabile globale*.

Variabilele locale ocupă spațiu de memorie doar pe timpul execuției funcției. Când funcția îşi încheie execuția, variabilele locale sunt şterse din memoria calculatorului. Acesta este motivul pentru care la fiecare apel al unei funcții variabiilele locale pornesc cu valori nedefinite, deci trebuie inițializate în interiorul funcției. Fiind şterse din memorie după încheierea apelului, valorile variabilelor locale nu se păstrează între două apeluri de funcții.

7.4 Parametri

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

Parametrii referință sunt cei pentru care tipul de dată este însoțit, în lista parametrilor formali, de semnul &. Funcția primeşte adresa de memorie a parametrului actual.

Din declarațiile parametrilor valoare lipseşte semnul &, iar funcția primeşte o copie a valorii parametrului actual.

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

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

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 operațiile 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 funcții.

Când funcția se încheie, conținutul parametrilor valoare este şters, la fel cum se întâmplă în cazul variabilelor locale. Diferența dintre parametrii valoare şi variabilelel

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

Parametrii referință

Parametrii valoare nu pot fi folosiți pentru a transmite informație către codul apelant pentru că valorile lor se pierd la finalul execuției funcției. Pentru aceasta se folosesc parametrii referință. Prin intermediul parametrilor referință funcției i se permite să modifice valoarea parametrului actual.

Dacă parametrul unei funcții este de tip referință, se transmite către funcție locația (adresa de memorie) şi nu valoarea parametrului. Există, astfel, o singură copie a informației, folosită atât de apelant cât şi de funcția apelată. Numele parametrului actual şi cel al parametrului formal devin *sinonime* pentru aceeaşi variabilă. Ceea ce va lăsa funcția în acea locație va fi regăsit de funcția apelant.

Spre deosebire de cazul parametrilor valoare, către parametrii referință 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 diferență importantă între parametrii valoare şi cei referință apare la verificarea potrivirii tipurilor de dată între parametrii actuali şi cei formali. Dacă în primul caz se realizează conversii implicite, pentru valorile referință 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 funcția se încheie, variabila temporară este ştearsă din memorie şi modificările din funcție 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 referință este un nume alternativ pentru argumentul corespondent. Apelul funcțiilor care au parametri valoare este asemănător celui pentru funcții cu parametri referință atunci când parametrii sunt variabile. În ambele situații 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 Funcții recursive

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

Rezolvarea problemelor prin recursie presupune scrierea unei funcții care este apelată recursiv. Această funcție rezolvă doar cazul cel mai simplu, numit şi *cazul de bază*. Dacă funcția 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 funcția 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 funcția 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 instrucțiune `return` pentru că rezultatul său va fi combinat cu partea pe care funcția 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 funcțiile se apelează pe ele însele cu versiuni din ce în ce mai simple ale problemei originale, această secvență trebuind să fie concepută în aşa fel încât să conveargă către cazul de bază. La acest punct, funcția recunoaşte cazul de bază şi se declanşează o serie de return-uri în secvență 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`:

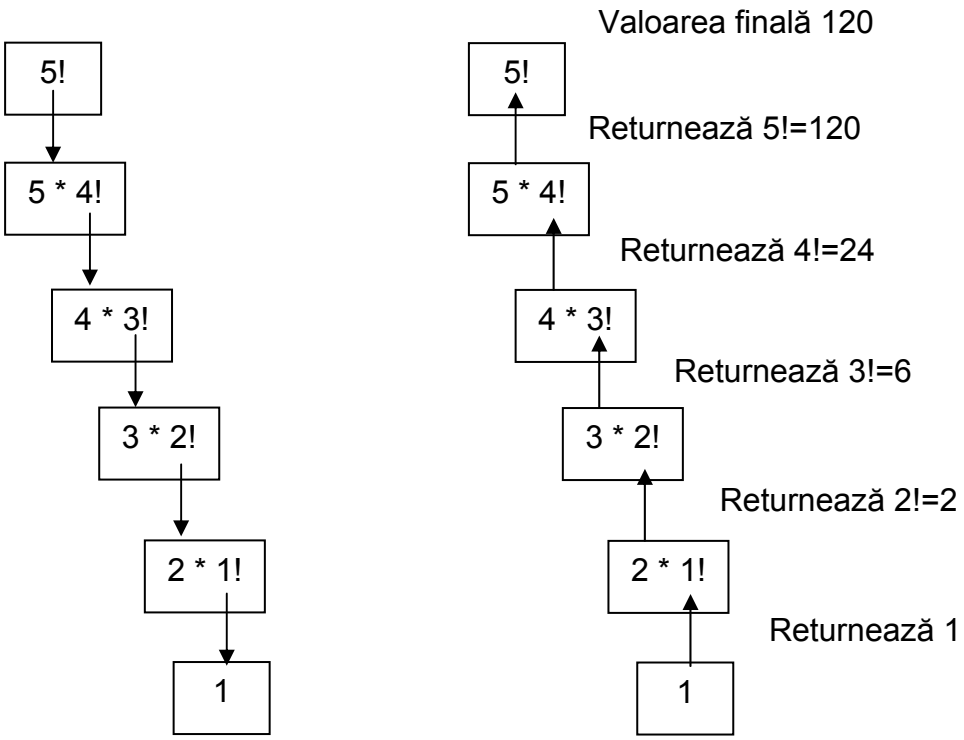
```
int factorial = 1;
```

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

O definiție recursivă a acestei operații este dată prin relația

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

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 iterație

Vom compara caracteristicile recursivității şi ale iterației.

Ambele tehnici se bazează pe câte o structură de control. Iterația foloseşte o structură repetitivă, iar recursivitatea o structură de selecție. Recursivitatea implementează repetiția prin apelurile repetate ale aceleiaşi funcții.

În ambele cazuri este nevoie de un test de terminare. Iterația se termină atunci când testul buclei devine fals, în timp ce recursia se termină atunci când se ajunge la cazul de bază. Iterația modifică un contor care controlează o buclă. Recursivitatea produce versiuni ale problemei originale până când se ajunge la cazul de bază. Atât iterația 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 funcțiilor care presupune ocuparea suplimentară a spațiului 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 secvențe de program.

Atunci când suntem puşi în situația de a alege între cele două tehnici, va trebui, aşadar, să găsim echilibrul între performanțele aplicației şi dorința de a scrie un program corect din punct de vedere conceptual.