Lucrarea 6

Declaraţii de funcţii – partea 1

Un program C este un ansamblu de funcţii care efectuează fiecare o anumită activitate bine definită. Funcţiile primesc parametri (argumente) pentru a efectua prelucrări cu diverse valori iniţiale şi pot să întoarcă o valoare rezultat. Există întotdeauna o funcţie numită main() şi aceasta este apelată la lansarea în execuţie a programului. Programatorul dispune de o bibliotecă amplă de funcţii uzuale, la care se adaugă funcţii specifice aplicaţiei.

Fişierul sursă al unui program C poate fi partiţionat în mai multe fişiere alcătuind un proiect. Fiecare fişier constă dintr-un set de funcţii şi declaraţii globale , dar numai unul conţine funcţia main().

În cele ce urmează ne vom ocupa de definiţiile şi declaraţiile funcţiilor şi modul de transfer al parametrilor şi rezultatului, aducând o serie de completări în privinţa declaraţiilor identificatorilor în general.

În C se utilizează declaraţii şi definiţii pentru funcţii:

- *definiţia* conţine antetul cu numele, lista parametrilor şi tipul funcţiei, urmat de blocul care descrie cu instrucţiuni C prelucrările efectuate de funcţie.
- *declaraţia* informează compilatorul despre numele, tipul rezultatului şi, eventual, tipurile parametrilor funcţiei.

1. Definirea funcţiilor

Sintaxa definiţiei unei funcţii este:

```
<tip_rezult> nume_functie (lista_parametri>)
      {<lista parametri locali>
      lista_instructiuni
      <return valoare;>
      }
```

tip_rezult este un tip de date oarecare, mai puţin tipul tablou şi reprezintă tipul rezultatului întors de funcţie. Pentru o funcţie care nu întoarce un rezultat se foloseşte tipul **void**. Dacă tip_rezult nu este specificat se consideră (implicit) că rezultatul este de tip **int**.

nume_functie reprezintă numele funcţiei.

lista_parametri este încadrată între paranteze şi constă din enumerarea declaraţiilor parametrilor, separate cu virgulă cu sintaxa:

```
tip_p identif_parametru1, tip_p identif_parametru2,…,tip_p identif_parametruN
ex1: (int a, float b, char c)
ex2: (int w = 3, float f = 3.265, char c =  ’s’)
```

Tipul parametrului, tip_p, poate fi orice tip valid de date, inclusiv adresa unei funcţii (pointer de funcţie).

Nu este admisă definirea unei funcţii în blocul altei funcţii şi nu sunt permise salturi cu instrucţiunea goto în afara funcţiei (dintr-o funcţie în alta).

Exemplu 1: Definirea funcției MAXF ce afișează maximul si media a două numere reale:

```
#include <iostream>
#include<stdio.h>
using namespace std;
void MAXF(float n1, float n2)
    {
    float max;   /* declaratie locala*/
    max=(n1>n2)?n1:n2;
    printf("Max=%f; Medie=%f\n",max,(n1+n2)/2);
}
 main()
{
    float x = 56.55;
    MAXF(x,12.55);
return 0;
}
```

Declararea funcției **MAXF** în afara funcției **main()**
n1 și n2 sunt parametru formali

apelul funcției **MAXF**
x și 12.55 sunt parametrii efectivi

Rezultat:

```
Max=56.549999; Medie=34.550000
```

Apelul unei funcţii constă din numele funcţiei urmat de lista de variabile sau constante asociate parametrilor, încadrată de paranteze ( ) care reprezintă operatorul de apel de funcţie. Vom conveni să numim *parametri formali* identificatorii declaraţi în lista de argumente din definiţia funcţiei şi *parametri efectivi* variabilele, constantele sau expresiile din lista unu apel al funcţiei.

Dacă funcţia nu are parametri, lista de parametri efectivi este vidă (doar parantezele), deci apelul este de forma fct().

2. Declaraţii de funcţii. Prototipuri

Definiţia unei funcţii apare în cadrul fişierului sursă înaintea oricărui apel numai în cazuri particulare. Acest lucru nu este posibil în general, fie datorită modului în care funcţiile se apelează unele pe altele, fie pentru că definiţia nu se află în fişierul sursă. Definiţia lipseşte în cazul funcţiilor din biblioteci (standard sau definite de utilizator, disponibile sub formă de fişiere obiect) sau atunci când se află în alt fişier sursă din proiect (nu se admit două definiţii ale unei funcţii într-un proiect).

Pentru a oferi compilatorului posibilitatea să efectueze verificarea validităţii apelurilor (numărul şi tipurile parametrilor) şi eventuale conversii de tip pentru parametri şi rezultat, sunt prevăzute declaraţii fără definire ale funcţiilor, numite prototipuri. Sintaxa generală este:

```
<tip> nume_functie (<lista_parametri>);
```

În cazul în care funcţia nu are parametri de intrare este obligatorie inserarea tipului **void** între paranteze, iar în C++ este suficientă lipsa oricărei informaţii între paranteze.

Funcţia main() are un rol distinct în program şi poate primi doar parametri predefiniţi ca pointeri de funcţie, aşa cum se va vedea ulterior. Din acest motiv nu este necesar un prototip şi în mod uzual nu se mai indică lipsa parametrilor prin cuvântul void.

Prototipul trebuie inserat în program înaintea oricărui apel ale funcţiei (de regulă este plasat chiar la începutul programului). Similar cu declaraţia de variabilă, domeniul de valabilitate al unei funcţii este limitat la partea care urmează declaraţiei:

- din fişierul sursă dacă apare în afara oricărei funcţii (la nivel global) sau
- din funcţia (blocul) în care apare, în caz contrar.

Prototipurile funcţiilor din bibliotecile C sunt oferite împreună cu declaraţiile de date şi macrodefiniţiile necesare în fişierul antet (header) identificate prin extensia ”.h” şi plasate în directorul INCLUDE.

Utilizarea unei funcţii din bibliotecă impune includerea în program a fişierului asociat cu ajutorul directivei #include. Programatorul îşi poate crea propriile fişiere antet conţinând declaraţiile funcţiilor, tipurilor globale, macrodefiniţiilor, etc. utilizate în program.

Este posibilă crearea unor funcţii care să primească un număr variabil de parametri. În acest caz, lista de parametri formali specifică numai sublista fixă urmată de ”…” . Compilatorul verifică la apelurile funcţiei numai parametri ficşi. De exemplu, funcţiile scanf() şi printf() au nevoie de şirul de control al formatului şi o sublistă variabilă de parametri, reprezentând valorile transferate între program şi consolă. Prototipurile sunt:

```
int scanf(const char *format,…);
int printf(const char *format,…);
```

Exemplu 2 : apel funcţii ce intorc un rezultat:

```
void showChoices();
float add(float, float);
float subtract(float, float);

int main()
{
    float x, y;     int choice;
    do
    {
        showChoices();      cin >> choice;
        switch (choice)
        {
        case 1:
            system("cls");
            cout << "Enter two numbers: ";
            cin >> x >> y;
            cout << "Sum " << add(x,y) <<endl;
            break;
        case 2:
            cout << "Enter two numbers: ";
            cin >> x >> y;
            cout << "Difference " << subtract(x,y) <<endl;
            break;
        case 5:
            break;
        default:
            cout << "Invalid input" << endl;
        }
    }while (choice != 5);
    return 0;   }
```

```
    void showChoices()
{
    cout << "MENU" << endl;
    cout << "1: Add " << endl;
    cout << "2: Subtract" << endl;
    cout << "5: Exit " << endl;
    cout << "Enter your choice :";
}
float add(float a, float b)
{   return a + b;}
float subtract(float a, float b)
{   return a - b;}
```

```
MENU
1: Add
2: Subtract
5: Exit
Enter your choice :
```

Temă: Adăugați la programul anterior și operațiile de înmulțire și împărțire, Rezultatul final trebuie să arate ca în figura de mai jos:

```
MENU
1: Add
2: Subtract
3: Multiply
4: Divide
5: Exit
Enter your choice :
```

Exemplu3: calculul cmmdc apelând o funcție din alt fișier.

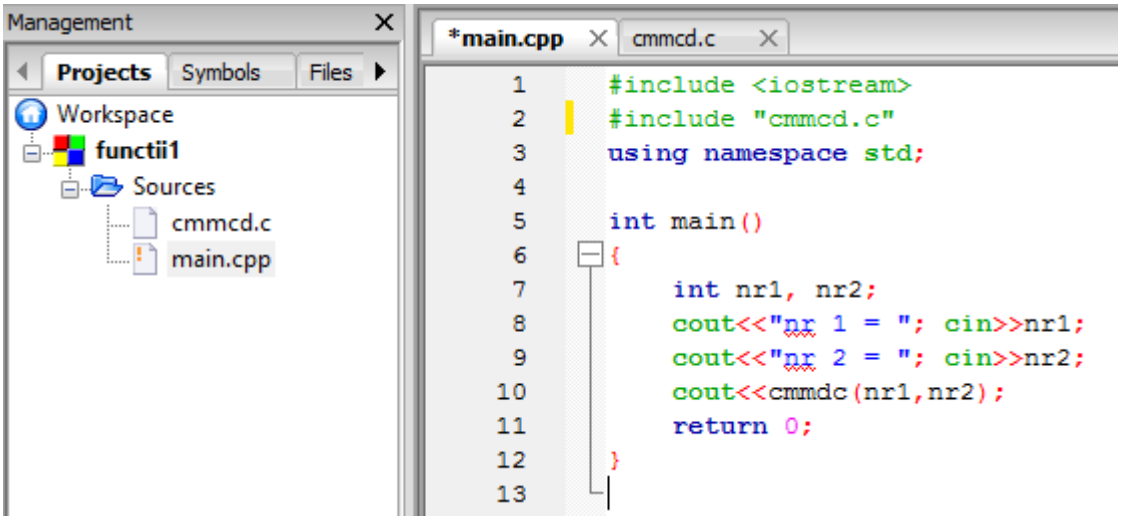

Fig. Corpul principal al programului, cu funcția main().

*cmmcd.c

```
int cmmdc(int x, int y)
{
if (x==0 || y==1 ||  x==1 || y==0)  return 1;
if (x<0)     x=-x;
if (y<0)     y=-y;
while (x != 0)
    {
    if ( y > x )
        {   int z=x;
            x=y;
            y=z;
        }
        x-=y;
    }
return y;
}
```

Fișierul cmmcd.c ce conține funcția cmmcd apelată de programul principal.

3. Transferul parametrilor

3.1. Transferul prin valoare.

Conversii de tip

Parametri formali reprezintă variabile locale care au ca domeniu funcţia, iar timpul de viaţă corespunde duratei de execuţie a funcţiei. Valorile parametrilor şi a altor variabile locale ale funcţiei sunt memorate în stivă sau în registrele microprocesorului.

În C procedeul de transfer al parametrilor constă în încărcarea valorii parametrului efectiv în zona de memorie a parametrului formal. Procedeul se numeşte transfer prin valoare.

Dacă parametru efectiv este o variabilă, orice operaţie efectuată în funcţie asupra parametrului formal nu o afectează, ceea ce poate constitui o protecţie utilă (variabila din programul apelant şi parametrul formal sunt obiecte distincte).

Transferul valorii este însoţit de conversii de tip realizate corespunzător informaţiilor de care dispune compilatorul despre funcţie. Prin utilizarea operatorului ”cast” sunt posibile conversii explicite.

Dacă prototipul precede apelul funcţiei şi nu există o sublistă variabilă de parametri, conversiile se fac exact ca în cazul unei atribuiri (conversia valorii parametrului efectiv la tipul parametrului formal). Dacă o asemenea conversie nu este posibilă se semnalează o eroare.

Dacă în fişierul compilat nu apare un prototip înainta apelării funcţiei, transferul este însoţit de o extindere automată a reprezentării anumitor tipuri (tipurile întregi scurte către int, float către double).

Dacă prototipul specifică o sublistă variabilă de parametri, valorile lor sunt supuse extinderii automate a reprezentării, ca în cazul în care lipseşte declaraţia.

Lipsa prototipului este o situaţie anormală în C şi este sancţionată întotdeauna cu eroare în C++.

3.2. Transferul prin referinţă

Pentru ca o funcţie C să poată modifica valoarea unei variabile indicate ca parametru efectiv, trebuie declarat un parametru formal de tip pointer, iar la apelare trebuie să i se ofere explicit adresa variabilei.

Funcţia **schimba** din exemplul următor trebuie să inverseze valori a două variabile întregi. Evident, funcţia are nevoie de adresele celor două variabile, nu de valoarea lor:

Exemplu4: transfer prin referinta.

```
using namespace std;
void schimba(int *a, int *b)
    {
int temp;
temp=*a; *a=*b; *b=temp;
}
int main()
{
    int x=10 ,y = 15;
    schimba(&x,&y) ;
    cout<<" x nou = " <<x;
    cout<<"\n y nou = " << y;
    return 0;
}
```

Acest mod de lucru este inevitabil, deoarece sintaxa limbajului C nu prevede transferul prin referinţă.

Cazul parametrilor de tablou constituie o excepţie de la regula transferului prin valoare, deoarece funcţia primeşte adresa tabloului ca parametru. Procedura este justificată de faptul că un tablou conţine în general o cantitate prea mare de date, ceea ce ar face neeficientă operaţia de copiere a valorilor tabloului într-o variabilă locală a tabloului.

4. Rezultatul unei funcţii

Instrucţiunea `return`

Instrucţiunea `return` determină încheierea execuţiei unei funcţii şi revenirea în funcţia apelantă. Sintaxa instrucţiunii este:

```
return <expresie>
```

Valoarea expresiei reprezintă rezultatul întors de funcţie, deci trebuie să fie compatibilă cu tipul din prototip şi definiţie. Dacă tipul rezultatului este void (funcţia nu întoarce un rezultat), instrucţiunea return apare numai dacă este necesară revenirea înainte de executarea secvenţei de instrucţiuni şi expresia lipseşte. În caz contrar, lipsa instrucţiunii return cu o expresie compatibilă cu tipul rezultatului este semnalată de compilator printr-un mesaj de avertisment sau de eroare.

Funcţiile C întorc o valoare în majoritatea cazurilor. Aceasta este rezultatul operaţiei efectuate sau doar o valoare convenţională care dă informaţii asupra modului în care s-a efectuat operaţia (valoarea –1). Dacă funcţia întoarce un rezultat apelul ei poate să apară în orice expresie şi la evaluarea expresiei se utilizează rezultatul întors.

Pot apărea conversii implicite ale valorii expresiei din instrucţiunea return la tipul specificat.

Exerciţiu 5 : Să se calculeze valoarea lui y, u şi m fiind citite de la tastatură:
$z=2\omega\ (\ 2\ \varphi(u)\ +\ 1,\ m)\ +\ \omega\ (2\ u^2\ -\ 3,\ m\ +\ 1)$, unde:

$\omega(x,n) = \sum_{i=1}^{n} \sin(ix)\cos(2ix)$, $\varphi(x) = \sqrt{1+e^{-x^2}}$, $\omega: R \times N \to R$, $\varphi: R \to R$

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

double omega(double x, int n)
{ double s=0; int i;
    for (i=1; i<=n; i++)
        s+=sin(i*x)*cos(i*x);
return s;
}
double psi( double x)
{   return sqrt( 1 + exp (- pow (x, 2)));
}
int main()
{
    double u, z;
    int m;
    cout<<"u="; cin>>u;
    cout<<"m="; cin>>m;
    z=2*omega(2* psi(u) + 1, m) + omega(2*pow(u,2) - 3, m+1);
    cout<<"z="<<z<<'\n';

    return 0;
}
```

5. Funcţii recursive

O funcţie poate să apeleze la rândul ei alte funcţii.

Dacă o funcţie se apelează pe sine însăşi, atunci funcţia este recursivă.

Pentru a evita un număr infinit de apeluri recursive, trebuie ca funcţia să includă în corpul ei o condiţie de oprire, astfel ca, la un moment dat, recurenţa să se oprească şi să se revină succesiv din apeluri. Condiţia trebuie să fie una generică, şi să oprească recurenţa în orice situaţie. Această condiţie se referă în general la parametrii de intrare, pentru care la un anumit moment, răspunsul poate fi returnat direct, fără a mai fi necesar un apel recursiv suplimentar.

Exemplu 6: Calculul recursiv al factorialului

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

double fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}

double fact1(int n) {
  return (n >= 1) ? n * fact(n - 1) : 1;
}

int main()
{
int n;
cout<<" n = "; cin>>n;
cout<<" n! = "<< fact(n)<<endl;
cout<<"n1!="<<fact1(n);
return 0;
}
```

Întotdeauna trebuie avut grijă în lucrul cu funcţii recursive deoarece, la fiecare apel recursiv, contextul este salvat pe stivă pentru a putea fi refăcut la revenirea din recursivitate. În acest fel, în funcţie de numărul apelurilor recursive şi de dimensiunea contextului (variabile, descriptori de fişier, etc.) stiva se poate umple foarte rapid, generând o eroare de tip stack overflow (vezi şi Infinite recursion pe Wikipedia).

Exerciţii de Laborator

1. Analizaţi programul de mai jos. Modificaţi sursa astfel încât programul să funcţioneze corect, fara a utiliza transmitere prin adresă de memorie.

```
#include<stdio.h>

void sum(int a, int b, int s) {
  s = a + b;
}

int main() {
  int s;
  sum(2, 3, s);
  printf("Suma este %d\n", s);

  return 0;
      }
```

2. Scrieți o funcție recursivă care să ridice un număr x la o putere dată y pozitivă.

```
power(2, 3); // rezultat 8
power(7, 3); // rezultat 343
```

3. Scrieți o funcție recursivă care să returneze numărul de cifre al unui numar întreg.

```
number_of_digits(34); // rezultat 2
number_of_digits(2533); // rezultat 4
```

4. Folosindu-vă de funcțiile scrise anterior scrieți o funcție recursivă ce inversează ordinea cifrelor unui numar întreg pozitiv.

```
reverse_number(23); // rezultat 32
reverse_number(3523); // rezultat 3253
```

5. Pentru un număr dat, determinați cel mai mic număr palindrom mai mare sau egal decât acel numar. Un palindrom este un număr care citit de la stânga la dreapta sau de la dreapta la stânga rezultă același numâr.

   ```
   next_palindrome(120); // rezultat 121
   ```

6. De la tastatură se citește o listă de numere pozitive. Pentru fiecare element citit se va afișa numărul prim cel mai apropiat de acesta. Dacă există două numere prime la fel de apropiate de elementul listei, se vor afișa amândouă. Dacă numărul este prim, nu se mai afișează nimic. Programul se încheie în momentul în care este citit un număr negativ. De exemplu:

```
27
* 29
13
*
68
* 67
69
* 67 71
      -1
```