UNIVERSITATEA TEHNICĂ A MOLDOVEI

Programarea calculatoarelor

Funcții definite de utilizator în limbajului C

Profesor: Maria Guțu

UNIVERSITATEA TEHNICĂ A MOLDOVEI

Cuprins

- Noțiune de subprogram & Structura unei funcții în C.
- Declararea și definirea funcțiilor în C.
- Apelarea funcțiilor.
- Tipuri de funcții.
- Exemplu de cod.

Noțiune de subprogram

În programarea calculatoarelor, pe lângă noțiunea de program se utilizează și cea de **subprogram**. ***Un subprogram*** este o secțiune independentă și reutilizabilă a programului, destinată rezolvării unei subprobleme specifice. Împărțirea unei probleme complexe în subprobleme și elaborarea de subprograme constituie principiul central al paradigmei programării procedurale, pe care se bazează limbajul C.

Exemple uzuale includ subprograme pentru introducerea sau afișarea elementelor unui tablou, ce pot fi integrate și reutilizate în diverse programe de prelucrare a datelor.

Noțiune de subprogram

Un subprogram este o parte din program care:

- are un nume,
- execută o sarcină bine definită,
- poate fi apelat ori de câte ori este nevoie.

În limbajul de programare C, subprogramele se implementează sub formă de funcţii.

Noțiune de funcție

În programarea C, funcțiile definite de utilizator sunt funcții scrise de programator pentru a efectua sarcini specifice care nu sunt disponibile prin funcțiile standard ale limbajului. Acestea permit modularizarea și reutilizarea codului, ceea ce îmbunătățește claritatea, structura și întreținerea unui program.

Exemple de funcții în C

În limbajul C, atât programul principal *main()*, cât și toate subprogramele sunt denumite funcții. Orice program scris în C poate include și utiliza una sau mai multe funcții, însă în mod obligatoriu trebuie să conțină funcția *main()*.

Din perspectiva utilizării, funcțiile se clasifică în:

- **funcții standard**, predefinite în bibliotecile limbajului, precum *scanf()* sau *printf()*, incluse prin directive de preprocesor (ex. *#include <stdio.h>*);
- **funcții definite de utilizator**, create de dezvoltator, precum *main(), sum(), max(), swap(), readArray(), showArray(), etc*.

Din perspectiva interacțiunii dintre ele, funcțiile pot fi:

- **funcții apelante**, care lansează în execuție alte funcții (de exemplu: *main(), sortArray(), etc.*);
- **funcții apelate**, care sunt lansate în execuție de alte funcții (de exemplu: *scanf(), printf(), sum(), max(), swap()*).

*Este important de menționat că funcția **main()** este unică și este lansată în execuție direct de către sistemul de operare.*

Structura unei funcții în C

O funcție în C este compusă din următoarele părți:

- **Tipul de întoarcere** (Return type): Acesta specifică tipul de valoare pe care funcția o va returna (*e.g., int, void, float etc*.).
- **Numele funcției**: Reprezintă identificatorul unic al funcției, care este folosit pentru a o apela.
- **Lista de parametri**: Parametrii sunt valorile pe care funcția le primește pentru a efectua operații. Aceștia sunt specificați între paranteze rotunde.
- **Corpul funcției**: Acesta conține instrucțiunile care sunt executate atunci când funcția este apelată.
- **Valoarea de retur**: Aceasta este valoarea pe care funcția o returnează (dacă returnează ceva).

Sintaxa de bază a unei funcții

```
tip_returnat nume_functie(tip_parametru1 parametru1, tip_parametru2 parametru2, ...)   // antetul funcției
{
    // Corpul funcției
    return valoare;  // Dacă funcția nu este de tip void
}
```

Antetul funcției conține trei elemente:

- tip_returnat - reprezintă tipul valorii returnabile de funcţie sau tipul funcției - coincide cu tipul expresiei a instrucțiunii return;
- **nume_functie** - denumirea funcției;
- **lista_parametri** - reprezintă lista tipurilor de parametri ai funcției (variabilelor locale folosite în cadrul funcţiei) cu denumiri generice, formale. Se mai numesc parametri formali. De menționat, că denumirile parametrilor nu trebuie neapărat să coincidă cu denumirile variabilelor respective folosite în funcția apelantă. Lista parametrilorformali ai funcției poate fi vidă (***void***).

Declararea funcției / Prototipul funcției

Prototipul unei funcții specifică tipul de întoarcere, numele funcției și tipurile parametrilor. Declarația unei funcții este plasată de obicei înainte de funcția **main()** sau într-un fișier header **.h** pentru a permite utilizarea acesteia în alte fișiere.

Exemplu de prototip:
int suma(int x, int y);

Aici, *int suma(int x, int y);* este un prototip al funcției care primește doi parametri de tip *int* și returnează un *int*.

Definirea funcției

Definirea funcției este unde scriem efectiv codul care va fi executat atunci când funcția este apelată.

Exemplu:

```
int suma(int a, int b) {
    return a + b;
}
```

Apelarea funcțiilor

Funcțiile sunt apelate folosind numele lor și transmițând parametrii corespunzători. În exemplul anterior, funcția ***suma*** ar putea fi apelată astfel:

Exemplu:

```
int suma(int a, int b) {
    return a + b;
}
int rezultat = suma(5, 10);
printf("Suma este: %d\n", rezultat);
```

Parametrii folosiți în apelul funcțiilor se numesc parametri actuali (sau efectivi). Ei pot fi constante (suma(5, 10);), expresii (suma(a + b, 10);) sau variabile (suma(a, b);).

Tipuri de funcții:
Funcții cu parametri și valoare de retur

Aceste funcții primesc argumente și returnează o valoare.

```
float inmultire(float x, float y) {
    return x * y;
}
```

Tipuri de funcții:
Funcții fără parametri, dar cu valoare de retur

Aceste funcții nu primesc parametri, dar returnează o valoare.

```
int obtineNumarAleator() {
    return rand() % 100;
}
```

Tipuri de funcții:
Funcții cu parametri, dar fără valoare de retur

Aceste funcții primesc argumente, dar nu returnează nimic (de obicei de tip `void`).

```
void afiseazaMesaj(char mesaj[]) {
    printf("%s\n", mesaj);
}
```

Tipuri de funcții:
Funcții fără parametri și fără valoare de retur

Aceste funcții nu primesc argumente și nici nu returnează o valoare.

```
void afiseazaSalut() {
    printf("Salut!\n");
}
```

Parametri transmiși prin valoare

Când o funcție primește parametri prin valoare, aceasta creează o copie a valorilor transmise. Orice modificare făcută asupra acestor parametri nu afectează valorile originale. În mod implicit, funcțiile în C folosesc transmiterea prin valoare.

```
void incrementare(int x) {
    x = x + 1;  // x++;    x +=1;
}  ...
int z = 5;  incrementare (z);   printf ("%d \n", z); // z => 5
```

Apelarea acestei funcții cu *incrementare(z);*, unde *z* = 5, nu va modifica valoarea originală a variabilei *z*.

Transmitere prin pointeri

Când o funcție primește parametri prin pointeri, ea poate modifica direct variabilele originale.

```
void incrementare(int *x) {
    (*x)++;
}...
int z = 5;  incrementare (&z);
printf ("%d \n", z);  // z = 6
```

```
void decrementare(int *x) {
    (*x)--;
}...
int z = 5;  decrementare (&z);
printf ("%d \n", z);  // z = 4
```

La apelul funcției ***incrementare*(&z)**;/***decrementare*(&z)**; se trimite adresa lui ***z***, iar funcția lucrează direct cu valoarea din acea adresă prin dereferențiere (*x).

Funcții recursive

Funcțiile recursive sunt acele funcții care se apelează pe ele însele pentru a rezolva o problemă. Un exemplu clasic de funcție recursivă este funcția care calculează factorialul unui număr.

```
int  factorial (int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial (n - 1);
}
```

Scopul și durata de viață a variabilelor în funcții

Variabile locale: Variabilele declarate într-o funcție sunt locale și pot fi utilizate doar în acea funcție. Durata lor de viață este limitată la execuția funcției.

Variabile globale: Aceste variabile sunt declarate în afara tuturor funcțiilor și pot fi accesate din orice funcție din program. Ele au o durată de viață pe toată durata execuției programului.

Avantajele utilizării funcțiilor create de utilizator

Modularizarea codului: Funcțiile ajută la împărțirea unui program în unități mai mici și mai ușor de gestionat.

Reutilizarea codului: O funcție poate fi apelată de mai multe ori în diverse părți ale unui program, eliminând redundanța.

Facilitează depanarea: Dacă funcțiile sunt bine definite, localizarea și rezolvarea erorilor devine mai ușoară.

Claritatea și întreținerea: Codul devine mai ușor de citit și de înțeles, deoarece fiecare funcție are o responsabilitate clară.

Problema 1

Se dau două numere naturale n și m. Să se determine suma dintre oglinditul lui n și oglinditul lui m.

```
#include <stdio.h>
int oglindit (int x) {
    int r = 0;
    do {
        r = 10 * r + x % 10;
        x /= 10;
    } while(x != 0);

return r;
}
```

```
int main(){
    int n , m;
    scanf("%d%d", &n, &m);
    printf("%d", oglindit(n) + oglindit(m));
    return 0;
}
```

Antetul funcției

Funcția are un **antet**: int oglindit(int x), din care deducem că:

- funcția se numește oglindit;
- funcția are un parametru, x, de tip int. Parametrul este important, deoarece prin el se precizează care este numărul pentru care se determină oglinditul:
  - o funcție poate avea mai mulți parametri;
  - parametrii care apar în antetul funcției se numesc **parametri formali**;
  - parametrii unei funcții se mai numesc și **argumente**;
- funcția are un anumit tip, aici int, care precizează care este tipul rezultatului;

Corpul funcției

Funcția are un bloc de instrucțiuni, **corpul funcției**, delimitat de acolade {}, care precizează operațiile prin care se obține rezultatul. Mai mult:

corpul funcției poate avea propriile variabile (aici r);

- aceste variabile se numesc **variabile locale**;
- în corpul funcției, parametrul se comportă ca o variabilă locală;
- nu putem avea o variabilă locală cu același identificator ca parametrul;
- în corpul funcției nu se cunosc valorile parametrilor formali. Funcția trebuie să determine rezultatul corect indiferent de valoarea lor;

rezultatul determinat în cadrul funcției este întors în programul apelant prin instrucțiunea **return**. Rezultatul este calculat în mod obișnuit în variabila r. Prin instrucțiunea return r;, valoarea curentă a variabilei r este returnată în programul appellant.

Apelul funcției

În funcția main găsim **apelul** funcției oglindit:

- apelul funcției se face într-o expresie:

printf("%d", oglindit(n) + oglindit(m));. Pentru operația de adunare, operanzii vor fi rezultatele apelurilor celor două funcții;

- parametrii ***n***, respectiv ***m***, întâlniți în apel, se numesc **parametri efectivi** sau **parametri actuali**. Valorile parametrilor actuali sunt cunoscute – valorile citite pentru cele două variabile.

Exemplu: Codul (sau definiția) funcției suma():

```
int suma(int n) // antetul funcției
{ // corpul funcției
     int s = 0;
     for(int i=1; i<=n; ++i){
          s += i;
     }
return s;
}
```

Exemplu: Codul (sau definiția) funcției max():

```
int max(int a, int b) // antetul funcției
{ // corpul funcției
     if (a>b) return a;
          //  else
     return b;
}
```

Exemplu: Codul (sau definiția) funcției readArray() :

```
void readArray(int *arr, int n) {
      int i;
      printf ("Dati elementele tabloului: ");
      for (i = 0; i < n ; i ++ ) {
            scanf(“%d”, &arr[i]);
      }
// return;
}
```

Exemplu: Codul (sau definiția) funcției showArray() :

```
void showArray(int *arr, int n){
     int i;
     printf ("Elementele tabloului: \n");
     for (i = 0; i < n ; i ++ ) {
          printf ("%d\t", arr[i]);
     }
     printf("\n");
// return;
}
```

Exemplu: Codul (sau definiția) funcției swap():

```
void swap(int* x, int* y) {
     int temp;
     temp = *x;
     *x = *y;
     *y = temp;
// return;
}
```

Apelul unei funcţii

Forma generală a apelului funcției este următoarea: ***nume_functie (arg1, arg2, …)***

Argumentele apelului care se mai numesc parametrii actuali (reali sau efectivi) pot fi expresii, variabile sau valori ce substitue parametrii formali respectivi ai codului funcției la momentul apelului. Cu alte cuvinte, parametrii formali ai funcţiei fiind variabile locale ale funcției sunt iniţializaţi cu valorile argumentelor la momentul apelului. Numărul, tipul și ordinea argumentelor (parametrilor actuali) și parametrilor formali trebuie să coincidă.
De menționat, că există două feluri de apel la funcție: apel-valoare și apel-instrucțiune (prin pointeri).

Exemplu de program cu apel la funcția utilizatorului (apel-valoare)

```
#include <stdio.h>

int suma(int num){
      int s = 0;
      for(int i=1; i<=num; ++i)
            s += i;
return s;
}
```

```
int main(){
      int sum, n= 100;
      sum=suma(n);
      printf("sum= %d\n",sum);
return 0;
}
```

Exemplu de program cu apel la funcția utilizatorului (apel-instrucțiune)

```
#include <stdio.h>

void showArray(int *arr, int n){
	int i;
	printf ("Elem. tabloului: \n");
	for (i = 0; i < n ; i ++ ) {
		printf ("%d\t", arr[i]);
	}
	printf("\n");
// return;
}
```

```
int main(){
	int n=10;
	int tab[]={12, -3, 0, 25, 15};
	showArray(tab, n);
return 0;
}
```

Prototipul (sau declarația) unei funcţii

Prototipul unei funcţii are următoarea formă generală (analogică cu prima linie a codului – antetul funcției): ***tip_returnat nume_funcţie (lista_parametri);***

(De menționat, că prototipul funcției se termină cu punct și virgulă și nu conține corpul funcției)

- **tip_returnat** – reprezintă tipul valorii returnate de funcţie;
- **nume_functie** – denumirea funcției;
- **lista_parametri** – reprezintă lista tipurilor de parametri ai funcției (variabile locale folosite în cadrul funcţiei), dar care pot avea opțional denumiri generice, formale (parametri formali).

De menționat, că anologic cu antetul funcției denumirile parametrilor nu trebuie neapărat să coincidă cu denumirile variabilelor respective folosite în funcția apelantă. Astfel, atunci când se apelează o funcţie folosind ca variabile parametrii actuali, numele variabilelor utilizate nu au nici o legătură cu numele parametrilor formali.

Prototipul (sau declarația) unei funcţii

Pentru funcţionarea corectă a programului funcţia utilizatorului trebuie definită (prin codul funcției) sau declarată (prin prototipul funcției) anterior apelului funcției.

Declararea funcției prin prototip este necesară dacă funcţia este definită (prin codul funcției) în altă parte decât în fişierul în care este apelată, sau dacă este definită în acelaşi fişier, dar după apelare la funcție.

Exemplu de program (utilizarea prototipului funcției)

```
#include <stdio.h>
int suma(int num);

int main(){
      int sum, n= 100;
      sum=suma(n);
      printf("sum= %d\n",sum);
return 0;
}
```

// codul functiei suma() se va scrie după f. *main()*

```
int suma(int num){
      int s = 0;
      for(int i=1; i<=num; ++i)
            s += i;
return s;
}
```

Interschimbarea datelor între două funcții

Pentru trimiterea datelor de la funcția apelantă la funcția apelată există 2 modalități:

1. Utilizarea variabilelor globale (nu se recomandă)
2. Utilizarea parametrilor funcției de tip regulat sau/și de tip pointer

(! Pentru tablouri trimitem în funcția apelată adresa tabloului în memorie, ci nu întreg tablou)

Pentru obținerea datelor și rezultatelor executării de la funcția apelată la funcția apelantă există 3 modalități:

1. Utilizarea variabilelor globale **(nu se recomandă)**
2. Utilizarea valorii returnabile ale funcției **(! o singură valoare de tip regulat sau de tip pointer)**
3. Utilizarea parametrilor funcției de tip pointeri **(! mai multe valori și rezultate corecte)**

Exemplu: Utilizarea parametrilor de tip pointeri

```
#include <stdio.h>
void swap(int* x, int* y);

int main(){
      int a = 10, b= 30;
      swap(&a, &b);
printf("a=%d\nb=%d ", a, b);
return 0;
}
```

```
// codul functiei swap() se va scrie după f. main()
void swap(int* x, int* y) {
      int temp;
      temp = *x;
      *x = *y;
      *y = temp;
}
```

Concluzie

Funcțiile reprezintă elemente fundamentale în limbajul C, asigurând ***modularitatea, claritatea și reutilizarea codului***. Prin împărțirea programului în subprograme mai mici, se facilitează rezolvarea problemelor complexe și întreținerea codului.

În C, funcțiile pot fi ***standard*** (din biblioteci) sau ***definite de utilizator***, iar **transmiterea parametrilor** se poate realiza **prin valoare** (copierea datelor) sau **prin adresă** (modificarea directă a variabilelor). În mod obligatoriu, orice program conține funcția principală **main(),** punctul de pornire al execuției. Astfel, funcțiile constituie pilonul central al programării procedurale în C.

Aplicații Practice!!!

(maria.gutu@iis.utm.md)