Cursul 9
FUNCTII (III)

	Obiective	
	Prezentare generala	

OBIECTIVE

- prototipul functiilor.
- functii fara argumente sau cu numar variabil de argumente.

PREZENTARE GENERALA

Prototipul functiilor

Inainte de a folosi o functie trebuie sa o declaram. Exista doua metode de a face acest lucru:

- declaram pur si simplu functia inainte de a fi folosita (definitia functiei); Definitia unei functii apare in cadrul fisierului sursa inaintea oricarui apel numai in cazuri particulare. Acest lucru nu este posibil in general, fie datorita modului in care functiile se apeleaza unele pe altele, fie pentru ca definitia nu se afla in fisierul sursa. Definitia lipseste in cazul functiilor din biblioteci (standard sau definite de utilizator, disponibile sub forma de fisiere obiect) sau atunci cand se afla in alt fisier sursa din proiect. Sintaxa definitiei unei functii este urmatoarea :

```
<tip_r> identif_functie(<lista_declar_parametri>)
{
    <lista_declaratii_locale>
    lista_instructiuni
}
```

- utilizam prototipul functiei. Prin prototip al unei functii intelegem o declaratie fara definire a functiei, in care se fac cunoscute numele, tipul returnat si lista parametrilor sai (ca numar, tip si, eventual, ca identificatori). Sintaxa prototipului unei functii este urmatoarea :

```
 <tip> identif_functie (<lista_declar_parametri>);
```

Specificarea numelor parametrilor este optionala, dar dupa inchiderea parantezelor rotunde trebuie sa se

puna punct si virgula.

```
void f(void);
```

Prototipul de fata indica faptul ca **f** este o functie fara parametri si care nu returneaza nici o valoare.

```
double a(void);
```

Functia **a** nu are parametri. Ea returneaza o valoare flotanta in dubla precizie.

```
void c(int x, long y[], double z);
```

Functia **c** nu returneaza nici o valoare. Are trei parametri:

- primul este de tip **int**;
- al doilea este un tablou unidimensional de tip **long**;
- al treilea este de tip **double**.

```
void c(int, long [], double);
```

Acest prototip exprima acelasi lucru cu cel precedent.

Aceeasi sintaxa utilizata in cadrul prototipului va trebui sa fie utilizata si in momentul definirii functiei.

```
// prototipul functiei patrat (declararea functiei)
double patrat(double);

// definitia functiei patrat double patrat(double x)
{ return x * x ;}
```

Se observa ca declaratia functiei **patrat()** nu include numele parametrului sau. De asemenea observam ca nu este necesar sa declaram prototipul unei functii inaintea definirii functiei, definitia insasi putand servi drept prototip. In mod obisnuit, prototipul functiei va aparea intr-un fisier de tip header. De obicei, declaratia unei functii este globala. Prototipul unei functii poate fi insa specificat in interiorul functiei care o apeleaza. In acest fel se "ascunde" acest prototip de alte functii. In consecinta, alte functii nu pot apela functia decat daca sunt declarate dupa declaratia acesteia din urma.

Prototipurile functiilor din biblioteci sunt oferite impreuna cu declaratiile de date si macrodefinitiile necesare in fisiere antet (header) identificate prin extensia "**.h**" si plasate in directorul **INCLUDE**. Serviciul help din mediul integrat precizeaza pentru fiecare functie fisierul antet in care este declarata.

Utilizarea unei functii din biblioteca impune includerea in program a fisierului asociat cu ajutorul directivei **#include.**Programatorul isi poate crea propriile fisiere antet continand declaratiile functiilor, tipurilor globale, macrodefinitiilor utilizate in program. Drept exemplu avem prezentarea unei situatii tipice de declarare, definire, si utilizare a functiior intr-un fisier sursa.

```
/* declaratii ptr. functii din biblioteci */
#include <stdio.h>

/* prototipuri ale functiilor definite in program */
void af_max(float, float);

/* alte prototipuri si declaratii globale pentru date */
void main()
{ float r1,r2;

    ...
    af_max(r1,r2); /*  apelul functiei af_max care primeste ca
                       parametrii doua valori float, afiseaza
                       valoarea maxima si media si nu intoarce
                       nici un rezultat */
}

/* definitia functiei af_max */
void af_max(float n1, float n2);
{ float max;

    max=(n1>n2)? n1:n2;
    printf("Max=%f; Media=%f\n",max,(n1+n2)/2 );
}
```

Limbajul C impune ca o functie sa fie intai declarata, in cazul in care ea este apelata inaintea definirii ei. In cazul absentei prototipului unei functii in C obtinem, cel mult, un avertisment din partea compilatorului. Mai mult, compilatorul considera, in mod implicit, ca tipul returnat de catre functii este **int**. Acest lucru poate conduce la aparitia situatiilor conflictuale, in special in cazul programelor multi-fisier.

```
// myfile1.c
double modul(double x)
{
    if (x<0) return -x;
```

```
    else return x;
}

// myprog1.c
#include <stdio.h>
void main()
{   int am, m=-3;

    am=modul(m);
    printf(" am= %d\n",am);
}
```

Deoarece **modul( )** nu are prototip in **myprog1.cpp**, compilatorul C trateaza argumentul si tipul returnat de aceasta functie ca fiind **int.** Totusi programul poate fi rulat,dar rezultatul este total eronat. Chiar daca vom utiliza prototipul functiei, nu suntem siguri ca am inlaturat sursa de erori. De exemplu, ce se intampla daca specificam prototipuri incorecte:

```
// myprog2.c
# include <stdio.h>
int modul(int x);//argument si tip returnat incorect

void main()
{   int am,m=-3;

    am=modul(m);
    printf(" am= %d\n",am);
}
```

Un compilator C va accepta aceasta fara sa furnizeze vreun avertisment macar, el presupunand ca parametrii sunt furnizati corect. Putem ajuta compilatorul sa depisteze erorile, procedand dupa cum urmeaza: mai intai prototipurile de functii la vom scrie in fisiere header, dupa care includem aceste fisiere atat in fisierele in care sunt implementate functiile, cat si in cele in care acestea sunt apelate. In acest mod, vom avea aceleasi prototipuri in ambele categorii de fisiere. Mai jos se afla scris programul conform celor spuse:

```
// myfile3.h
double modul(double x);//Prototipul din header

// myfile3.c
#include "myfile3.h"
```

```
double modul(double x)
{
    if (x<0) return -x;
    else return x;
}

// myprog3.c
#include <stdio.h>
#include "myfile3.h"

void main()
{   int am, m=-3;

    am=modul(m);
    printf(" am= %d\n",am);
}
```

In acest fel eroarea este depistata. Probabil veti crede ca, procedand astfel, erorile nu mai pot patrunde in programele noastre. Din nefericire nu este asa. De exemplu, putem gresi chiar prototipul functiei in cadrul headerului, si deci, vom folosi un prototip eronat in ambele fisiere sursa.

```
// myfile.h
double modul(int x);//Argument incorect
```

Din nefericire, nu se garanteaza functionarea corecta a unor functii pentru care prototipul nu este corect specificat. Acest lucru se datoreaza faptului ca tipul returnat nu este verificat, nici de compilator si nici de catre link-editor. Ca atare, urmatoarea eroare nu va fi depistata:

```
// myfile4.h
int modul(double x);//tipul valorii returnate eronat

// myfile4.c
#include "myfile4.h"
double modul(double x)
{
    if (x<0) return -x;
    else return x;
}

// myprog4.c
```

```
#include <stdio.h>
#include "myfile4.h"

void main()
{   int am, m=-3;

    am=modul(m);
    printf(" am= %d\n",am);
}
```

Mentionam ca, in cazul absentei tipului valorii returnate in cadrul prototipului unei functii, acesta va fi considerat cel implicit, deci **int.**

Functii fara argumente sau cu numar variabil de argumente

In C, cel mai bun mod de a obtine prototipurile unor astfel de functii este urmatorul:

```
int f(void); //f nu are nici un argument
int f(...); //f are numar variabil de argumente
```

Pentru siguranta, este bine ca intotdeauna sa se specifice numarul argumentelor, fie prin mentionarea parametrilor formali, fie utilizand ellipsis "**(...)**", fie prin **(void).** Ellipsis este un grup de trei puncte fara spatii intre ele si se utilizeaza in lista de parametri formali ai prototipului unei functii pentru a indica prezenta unui numar variabil de argumente in linia de apel a functiei sau a argumentelor cu tipuri variabile. De exemplu,

```
void funct(int n,char c,...);
```

declara **funct( )** ca fiind o functie in al carui apel va aparea cel putin doi parametri efectivi, un **int** si un **char**, dar fiind permisa si prezenta altor parametri.

In C, prototipul trebuie sa specifice explicit faptul ca functia nu are parametri prin utilizarea cuvantului **void**. Pentru compatibilitate cu declaratia clasica, daca lipseste lista de parametri din declaratie si nu apare cuvantul **void**, compilatorul C nu decide ca functia nu are parametri si nu verifica parametrii efectivi la apelare. Este suficient ca lista de parametri din prototip sa specifice tipurile. Identificatorii nu sunt semnificativi, dar pot fi utili pentru documentarea programului si sunt utilizati de compilator in cadrul mesajelor de eroare pentru identificarea mai usoara a parametrului eronat.

Definirea unei functii cu un numar variabil de argumente de catre utilizator este o problema delicata, deoarece necesita cunoasterea modului in care sunt memorate valorile parametrilor efectivi. O alta problema este referirea acestor valori, in lipsa unor identificatori asociati in lista de parametri formali. O

solutie simpla si portabila o ofera un set de functii (de fapt macrodefinitii) declarate in **stdarg.h**. Acestea permit accesul la lista de parametri in cazul in care functia nu cunoaste numarul si tipurile parametrilor. Fisierul antet **stdarg.h** declara tipul **va_list** si functiile **va_start( )**, **va_arg( )** si **va_end( )**,unde:

- **va_list** este un pointer catre lista de parametri. In functia definita de programator trebuie declarata o variabila de acest tip care va permite adresarea parametrilor (fie **ap** numele variabilei).
- **va_start( )** initializeaza variabila **ap** de tipul **va_list** cu adresa primului parametru din sublista variabila. Prototipul este:

  ```
  void va_start(va_list ap, ult_fix);
  ```

  unde **ult_fix** este numele ultimului parametru din sublista fixa.

- **va_arg( )** intoarce valoarea parametrului urmator din sublista variabila. Prototipul este:

  ```
  tip_p va_arg(va_list ap, tip_p);
  ```

  unde **tip_p** este tipul parametrului urmator. La fiecare apelare **va_arg( )** intoarce valoarea parametrului indicat de **ap** si modifica variabila **ap** astfel incat sa indice parametrul urmator. In acest scop folosim tipul specificat al parametrului (**tip_p**)pentru a stabili dimensiunea zonei de memorie alocata. Datorita conversiilor implicite la transferul valori din sublista variabila, tipurile **char**, **unsigned char**si **float**nu se folosesc cu **var_arg( ).**Valorile transferate sunt intotdeauna extinse la tipul **int**, respectiv **double**.

- **va_end( )** incheie operatia de extragere a valorii parametrilor si trebuie apelata neaparat inainte de revenirea in functie

  ```
  void va_end(va_list ap);
  ```

Dupa ce se executa **va_end( )** pentru a relua extragerea parametrilor este necesar sa se apeleze din nou **va_start( )**.