Cursul 7
FUNCTII (I)

OBIECTIVE

- Fundamente,
- Functii care returneaza non-intregi.
- Argumentele functiilor.
- Argumentele liniei de comanda.

PREZENTARE GENERALA

In acest laborator se vor studia avantajele utilizarii functiilor in limbajul programare C++ si principalele caracteristici ale acestora. Scopul laboratorului este de a utiliza functiile C si de a oferi un suport programatorilor in programarea orientata pe obiecte. Limbajul C a fost proiectat pentru a face functiile eficiente siusor de folosit, programele C constau,in general,.mai degraba,din numeroase functii mici decit din functii mari. Un program poate fi compus din unul sau mai multe fisiere sursa,acestea putind fi compilate separat si incarcate impreuna,impreuna cu alte functii compilate anterior,care se gasesc in biblioteci. Aceste functii trunchiaza programele mari in mai multe programe mici si permit programatorului sa construiasca incepind de la ceea ce au facut altii deja,in loc de a porni totul de la inceput.

FUNDAMENTE

Fiecare functie C este de forma:

```
tip_returnat nume (declaratii_argumente, daca exista)
{
    declaratii si instructiuni, daca exista
}
```

Asa cum am sugerat, anumite parti pot sa lipseasca, functia minima fiind

```
nimic() { }
```

care nu face nimic. (O functie care nu face nimic este utila uneori ca loc pastrat pentru dezvoltari ulterioare in program). Numele functiei poate fi, de asemenea, precedat de un tip, daca functia returneaza altceva decat o valoare intreaga. (Acesta este subiectul urmatoarei sectiuni).

Obs.:Un program este tocmai un set de definitii de functii individuale. Comunicarea intre functii este (in acest caz) facuta prin argumente si valori returnate de functii. Ea poate fi facuta, de asemenea, prin variabile externe. Functiile pot aparea in orice ordine in fisierul sursa, iar programul sursa poate fi spart in mai multe fisiere. In schimb o functie nu poate fi sparta in mai multe fisiere.

Instrunctiunea **return** este mecanismul de returnare in apelant a unei valori din functia apelata. Orice expresie poate urma dupa instructiunea return:

```
return (expresie)
```

sau

```
return expresie
```

Functia apelanta este libera sa ignore valoarea returnata,daca doreste.Mai mult, nu e necesar sa existe o expresie dupa **return**,caz in care nu va fi returnata nici o valoare apelantului.Totusi, compilatorul va avertiza asupra acestui fapt. Controlul este, de asemenea, returnat apelantului, fara nici o valoare,atunci cand executia functiei apelate atinge cea mai din dreapta acolada. Nu este ilegal ca o functie sa returneze o valoare intr-un loc si nici o valoare din altul. In orice caz , valoarea unei functii care nu returneaza nici una este sigur un non-sens.Mecanismul prin care se compileaza si se incarca un program al carui cod este format prin compunerea mai multor fisiere sursa variaza de la un sistem la altul.

Functii care returneaza non-intregi

Daca un nume care nu a fost declarat apare intr-o expresie si este urmat de o paranteza stanga, el este declarat prin context ca fiind nume de functie. Mai mult, implicit se presupune ca o functie returneaza un **int**. Deoarece **char** se transforma in **int** in expresii,nu e nevoie sa declaram functiile care returneaza **char**. Aceste prezumptii acopera majoritatea cazurilor, inclusiv o mare parte din elementele de acum.

Dar ce se intampla daca o functie trebuie sa returneze o valoare de alt tip?

Multe functii numerice, ca **sqrt()**, **sin()**, **cos()**, returneaza **double**; alte functii specializate returneaza alte tipuri. Pentru a ilustra modul lor de folosire vom scie si vom folosi o functie **atof(s)** care converteste sirul s in echivalentul lui in dubla precizie; **atof()** este o prezenta sau o absenta atat a partii intregi, cat si a partii fractionare.

In primul rand, **atof()** insasi trebuie sa declare tipul valorii pe care o va returna, deoarece acesta nu este

int. Deoarece **float** este convertit in double in expresii, nu are nici un rost sa spunem ca **atof()** returneaza un **float**; putem, la fel de bine, sa facem uz de precizie suplimentara si sa declaram ca ea returneaza double. Numele tipului precede numele functiei, ca in :

```
double atof(char *s)
{   double val, putere;
    int i, semn ;

    for(i=0;s[i]==' ' ||s[i]=='\n' ||s[i]=='\t ' ;i++);

    semn=1;
    if(s[i]=='+' || s[i]== '-') semn=(s[i++]=='+')?1: -1;
    for(val=0 ;s[i]>='0' &&s[i]<='9' ; i++)
        val=10*val+s[i]-'0' ;
    if (s[i]=='.') I++;
    for (putere =1;s[i]>='0' && s[i]<='9' ; i++)
    {    val=10*val+s[i]-'0' ;
         putere*=10;
    }
    return semn*val/putere;
}
```

Declaratia

```
double atof (char *s);
```

spune ca **atof()** este o functie care returneaza o valoare double.

Daca **atof()** insasi si apelul ei din **main()** au tipuri inconsistente in acelasi fisier sursa, acest lucru va fi depistat de catre compilator.Dar, daca **atof()** se compileaza separat, nepotrivirea nu va fi declarata si **atof ()** va returna un **double**, pe care **main()** il va trata ca un intreg, rezultand raspunsuri imprevizibile.Fiind dat **atof()**, putem scrie, in principiu, functia atoi() (conversie de sir in intreg) astfel:

```
atoi( char s[])
{   double atof() ;

    return atof (s);
}
```

Sa remarcam structura declaratiilor si a instructiunii **return**.Valoarea expresiei din:

```
return expresie
```

este intodeauna convertita in tipul functiei inainte ca returnarea rezultatului sa aiba loc.Deci valoarea lui **atof()**, un **double**, este convertita automat in **int**, cand apare in instructiunea **return**, deoarece functia **atoi()** returneaza un **int**. (Conversia unei valori flotante intr-un intreg trunchiaza orice parte fractionara.).

Mai multe despre argumentele functiilor

Argumentele functiilor Ctrimise prin valoare ,adica functia apelata primeste o copie temporara si privata a fiecarui argument. Aceasta inseamna ca functia nu poate afecta argumentul original din functia apelanta. Intr-o functie, argumentul este , de fapt , o variabila locala,initializata cu valoarea cu care functia este apelata. Cand un nume de tablou apare ca argument al unei functii, locatia de inceput a tabloului este cea trimisa efectiv; elementele nu sunt copiate. Functia poate altera elementele tabloului, indexand aceasta valoare. Efectul este ca tablourile sunt trimise prin referinta.

Deocamdata sa ilustram aceasta proprietate printr-o versiune a functiei **strlen()**,care calculeaza lungimea unui sir.

```
int strlen(char *s)
{

    for (int n=0;*s!='\0' ;n++);
    return n;
}
```

Incrementarea lui **s** este perfect legala deoarece el este o variabila pointer; **s++** nu are efect pe sirul de caractere in functia carea apelat-o pe **strlen()**, ci incrementeaza doar copia adresei.

Ca parametrii formali in definirea unei functii,

`char s[]` si **char *s;**

sunt echivalenti; alegerea formei efective este determinata in mare parte de expresiile ce vor fi scrise in cadrul functiei.Atunci cand un nume de tablou este transmis unei functii, aceasta poate, dupa necesitati, s-o interpreteze ca tablou sau ca pointer si sa-l trateze in consecinta.

Functia poate efectua chiar ambele tipuri de operatii, daca i se pare potrivit si corect. Este posibila si transmiterea de catre o functie doar a unei parti dintr-un tablou,prin transmiterea unui pointer la inceputul subtabloului.

De exemplu, daca **a** este un tablou,

f(&a[2]) si f(a+2)

transmit functiei **f** adresa elementului a[2], deoarece &a[2] si a+2 sunt expresii pointer care refera al treilea element al lui **a**. In cadrul functiei f, declaratia parametrului poate fi:

```
f(int arr[ ])
{ . . . }
```

sau

```
f(int *arr)
{ . . . }
```

Astfel, dupa cum a fost conceputa functia f, faptul ca argumentul refera,de fapt, o parte a unui tablou mai mare nu are importanta.

In cazul unui tablou bidimensional trebuie transmis unei functii, declararea argumentelor in functie trebuie sa includa dimensiunea liniei, dimensiunea coloanei fiind irelevanta si aceasta deoarece unei functii i se transmite, ca si in cazurile anterioare un pointer. De exemplu, daca trebuie transmisa o matrice cu 2 linii si 7 coloane ale carei elemente sunt intregi, vom utiliza un pointer care parcurge tablouri de cate 7 **int**. Astfel , declaratia functiei **f** va fi:

```
f(int matrice[2][7])
{ . . . . . }
```

Declararea argumentului **f** poate fi , de asemenea:

```
int matrice[][7]
```

din moment ce numarul liniilor este irelevant, sau ar putea fi

```
int (*matrice)[7]
```

care spune ca argumentul este un pointer pe un tablou de 7 intregi.

Dupa cum am mai semnalat intr-o discutie anterioara, parantezele curbe sunt necesare datorita faptului ca parantezele drepte au prioritate mai mare decat * fara paranteze,iar:

```
int *matrice[7];
```

este un tablou de 7 pointeri la intregi,ceea ce ar echivala cu transmiterea matricei pe coloane ,si nu pe linii.

Argumentele liniei de comanda

Si, cum **main()** este o functie ca oricare alta functie, ea poate avea parametrii. Acestia, insa nu pot fi transmisi prin metode clasice, ci numai in cadrul liniei de comanda , deci in momentul lansarii programului in executie. Pentru aceasta, la inceperea executiei, **main()** primeste doua argumente. Primul (numit conventional **argc**) contine numarul parametrilor din linia de comanda prin care a fost apelat programul, iar al doilea (**argv**) este un pointer la un tablou de siruri de caractere care contine argumentele, cate unul in fiecare sir. Manipularea acestor siruri de caractere este o utilizare comuna a nivelelor multiple de pointeri.

Cea mai simpla ilustrare a declaratiilor necesare si a celor de mai sus amintite este programul **echo**, care pune, pur si simplu, pe o singura linie argumentele liniei de comanda, separate prin blancuri. Astfel,daca este data comanda:

```
echo Bine ati venit in lumea C
```

iesirea este:

Bine ati venit in lumea C

Prin conventie, **argv[0]** este numarul prin care se recunoaste programul, asa ca **argc** este 1.

In exemplul de mai sus, **argc** este 7 si **argv[0]**, **argv[1]**,**argv[3]**,**argv[4]**,**argv[5]** si **argv[6]** sunt , respectiv, **echo**, **Bine**, **ati**,**venit**,**in**,**lumea** si **C**. Aceasta este ilustrata in **echo:**

//**echo**- prima versiune

```
#include <stdio.h>
void main( int argc, char *argv[ ])
{   int i;

    for (i=1; i<argc; i++)
        printf("%s%c", argv[i],(i<argc-1)?' ':'\n');
}
```

argv fiind un pointer la un tablou de pointeri, exista cateva modalitati de a scrie acest program care implica manipularea pointerului mai curand decat indexarea tabloului. Nu este lipsit de interes sa prezentam inca doua variante:

// **echo**- a-II-a versiune

```
void main ( int argc, char *argv[ ])
{
    while (--argc>0)
        printf("%s%c",*++argv,(argc>1)?' ' : '\n');
}
```

Daca argv este un pointer la inceputul tabloului care contine siruri de argumente cu 1 (prin **++argv** face ca el sa indice pe **argv[1]** in loc de **argv[0]**. Fiecare incrementare succesiva muta pe **argv** pe urmatorul argument; **argv** este, deci, pointerul la acel argument.

Simultan, **argc** este decrementat; atunci cand el devine zero, nu mai exista argumente de imprimat.

// **echo- a III a versiune**

```
void main( int argc,char *argv[ ])
{
    while (--argc>0)
        printf((argc>1)? "%s" : "%s\n" ,*++argv);
}
```

Aceasta versiune arata ca formatul argumentului lui printf poate fi o expresie ca oricare alta .Aceasta utilizare nu este foarte frecventa, dar este bine sa fie retinuta.