Cursul 10
FUNCTII (IV)

OBIECTIVE

- Transmiterea parametrilor prin referinte
- Functii inline
- Rescrierea functiilor

PREZENTARE GENERALA

Transmiterea parametrilor prin referinta

In limbajul C, toti parametri sunt transmisi prin valoare (fiecarei functii ii sunt transmise valorile efective ale parametrilor). De aceea, este de preferat sa utilizam pointeri ca parametri.

```
void actual(int *t,int s)
{ *t=s; }
/*  t este transmis prin referinta, prin intermediul
    unui pointer iar s prin valoare */
```

Exista diferente intre apelul cu pointeri ("->") si apelul cu referinte("."). Bineinteles ca putem utiliza oricare dintre cele doua metode, dar se pare ca functiile utilizand referintele sunt mai clare la apelare.

Functii inline

```
/* cazurile in care ati utilizat expresii macro in
vederea simulari apelurilor functiilor. */
#define INC(i) i++
...
k=INC(j);
```

Problemele vin tocmai din faptul ca aceste apeluri seamana cu apelul functiilor, dar nu sunt functii. Parametrii expresiilor macro sunt inlocuiti inline atunci cand o astfel de expresie este intalnita de catre compilator. In schimb, parametrii functiilor sunt verificati din punct de vedere al tipului si pasati utilizand scopul fiecaruia.

```
k=INC(3+5); // va produce
k=3+5++; // o sintaxa incompilabila.
```

In vederea rezolvarii unor astfel de situatii, C a fost inzestrat cu functii inline = combinatie intre expresii macro (apelul unei functii inline este expandat prin inlocuirea apelului sau cu corpul corespunzator implementarii functiei) si functii(se executa verificarea tipurilor parametrilor, acestia din urma fiind transmisi ca unei functii normale).

Declararea functiilor inline se face prin utilizarea cuvantului cheie **inline**, exact inaintea definitiei functiei (inaintea tipului returnatde functie).

```
inline int inc(int n)
{ return n++; }
...
int i;
i=inc(3+5);
```

In momentul apelarii functiei **inc()**, compilatorul expandeaza functia inline, dar nu inainte de a aduna 3 cu 5 si de a depune rezultatul intr-o variabila temporara. Rezultatul acestei adunari este apoi incrementat si atribuit lui **i**. Un cod echivalent cu apelul anterior ar putea fi:

```
int temp=3+5;
i=temp++;
```

Utilizand cuvantul **inline**, nu putem fi siguri ca functia careia i-am atasat acest cuvant va fi considerata de catre compilator astfel. Asemeni cuvantului **register**, **inline** constitue mai degraba o recomandare facuta compilatorului, noi considerand ca functia este suficent de mica pentru a fi expandata inline. Compilatorul poate sa ia in considerare sau nu aceasta recomandare. In cazul in care utilizam optiunea de compilare **-vi**, ii semnalam compilatorului sa ignore cuvantul **inline**. O situatie in care am putea dori aceasta este atunci cand depanam un program, functiile expandate inline fiind greu de urmarit de catre depanator.

Functii inline si fisierele header

Functiile inline sunt omoloage constantelor (identificatorii declarati cu ajutorul constantelor (in mod uzual in cadrul fisierelor header) sunt initializati in momentul declararii lor):

- corpul unei functii inline va fi definit in momentul declararii acesteia ca fiind inline;
- functiilor inline li se vor lega atribute la link-editare;
- declararea functiilor inline se va face in cadrul fisierelor header.

```
// header1.h
void arata_valoarea(int v);

// file1.c
#include <stdio.h>
#include "header1.h"

inline void arata_valoarea(int v)
{
    printf("Valoarea este %d\n",v);
}

// progr1.c
#include "header1.h"

void actiune(int &v)
{
    v^=0*1010;
    arata_valoarea(v);
}

void main()
```

```
{   int biti=42;

    actiune(biti);
}
```

Functia **arata_valoarea()** a fost declarata ca fiind inline in cadrul fisierului ***file1.c***, acolo unde a fost declarata aceasta. In schimb, in cadrul headerului ***header1.h*** a fost omis cuvantul inline. Problema care apare este aceea ca odata declarand **arata_valoarea()** ca fiind inline, s-a realizat si legarea interna a acesteia. Deci ea va apartine modulului ***file1.c*** fiind astfel inaccesibila din exteriorul acestuia. In acest context, compilatorul va semnala eroarea legala de imposibilitatea de a localiza functia **arata_valoarea()** in **progr1.c**.

Daca am fi renuntat la cuvantul **inline**, atunci **arata_valoarea()** ar fi avut legare externa la link-editare, asa cum o au toate functiile normale, programul s-ar fi compilat si link-editat cu succes, dar **arata_valoarea()** nu ar fi fost inline.

Solutia este de a defini aceasta functie ca fiind inline in cadrul fisierului header, caruia, de fapt, ii si apartine. Procedand astfel, prezenta fisierului ***file1.c*** este nejustificata.

```
// header2.h

inline void arata_valoarea(int v)
{ printf("Valoarea este %d\n",v); }

// progr2.c
#include "header2.h"

void actiune(int &v)
{ v^=0*1010; arata_valoarea(v); }

void main()
{   int biti=42;

    actiune(biti);
}
```

Ascunderea functiilor sub diferite nume

```
// functie de deschidere a fisierelor
FILE *fopen(const char *fisier, const char *mod);

/* indicatorul mod va indica situatia in care fisierul
este deschis pentru citire si/sau scriere. */
```

Problema care apare in astfel de situatii este ca se poate gresi foarte usor. In plus, un astfel de flag (fanion) are rolul de a indica deschiderea unui fisier existent sau a unuia nou, simpla inversiune a rolurilor acestui indicator putand fi fatala unui fisier deja existent, in cazul utilizarii indicatorului de creare, in locul celui de deschidere normala. Pentru a evita astfel de situatii, putem sa ne scriem functiile noastre inline, de deschidere si de creare de fisiere:

```
inline FILE *initializare(char *fnume)
{ return fopen(fnume,"w"); }
```

```
inline FILE *deschidere(char *fnume)
{ return fopen(fnume,"r"); }
```

Aceste doua functii sunt mult mai usor de utilizat, si mai sigure in acelasi timp, decat **fopen().** Pe de alta parte, fiind functii inline, nu afecteaza, in nici un fel, timpul de executie sau dimensiunea codului.

Rescrierea functiilor

Este permisa utilizarea mai multor functii avand acelasi nume, asemenea functii purtand numele de functii redefinite (rescrise, overloaded sau suprapuse).

```
// progr3.c
/* trei functii aduna(), fiecare pentru cate unul
dintre tipurile int, double si char*(siruri de caractere). */

#include <stdio.h>
#include <string.h>
int aduna(int a, int b)
{ return a+b; }

double aduna(double a, double b)
{ return a+b; }

char *aduna(char *a, char *b)
{ strcat(a,b); return a; }

void main()
{   int i=aduna(42,17);
    double d=aduna(42.0,17.0);
    char s1[80]="C++";
    char s2[80]="is the best!";

    printf("\n i=%d \n d=%f \n %s \n",i,d,aduna(s1,s2));
}
```

Nu este neaparat nevoie sa specificam acelasi numar de parametri intr-un set de functii redefinite.

```
// putem scrie inca o functie aduna(), care sa accepte un
// singur parametru, fara a fi considerata ca fiind o functie eronata:

int aduna(int i)
{ return i+42; }
```

Rezolvarea ambiguitatilor dintre functiile redefinite

(sau cum decide compilatorul asupra variantei functiei **aduna()** ce urmeaza a fi utilizata)

Atunci cand compilatorul depisteaza apelul unei functii redefinite, cauta in lista redefinirilor acelei functii, una a carei lista de parametri formali sa coincida, ca numar, tip si pozitie, cu lista parametrilor efectivi, utilizati in acel apel. Aceasta identificare este facuta cu o oarecare usurinta in cazul in care numarul parametrilor difera de la o implementare la alta, acest numar intervenind in

mod direct in procesul de identificare.

In general, regulile care intervin in localizarea functiei dorite sunt destul de complexe. Pentru fiecare argument efectiv, compilatorul va utiliza un set de reguli prestabilite, in vederea gasirii celei mai bune potriviri de tip. Regulile sunt apelate in ordinea in care apar enumerate in continuare, cele mentionate mai intai fiind considerate mai bune decat cele lasate spre sfarsitul listei. Strategia de baza este de a cauta potrivirea exacta, iar, in cazul in care aceasta este imposibil de gasit, sa se incerce simple conversii de tip, in vederea obtinerii unei potriviri cat mai bune. Iata deci, regulile, urmate de conversiile triviale disponibile:

- Cauta potrivirea exacta si o utilizeaza in cazul gasirii ei. De asemenea, cauta potrivirea ce utilizeaza conversii triviale.
- Cauta conversii de la **float** la **double** sau de la **char**, **shor**t, **enum** si **campuri de biti** la **int**.
- Cauta potriviri in care intervin conversii aritmetice standard, cum ar fi **int** la **double**, **unsigned** la **signed**, etc. De asemenea, se vor cauta conversiile de la orice tip de pointer la **void***, si se va converti constanta **0** la pointerul **NULL**. In plus, se va incerca conversia pointerilor la clase derivate, in pointeri la clase de baza si referinte la clase derivate, in referinte la clase de baza.
- Se vor cauta conversii care necesita crearea unui obiect temporar, asemeni transmiterii unui obiect constant unei functii ce are drept parametru un obiect neconstant.
- Se vor cauta potrivirile de tip utilizand conversiile de tip definite de utilizator.
- Se va incerca potrivirea de tip prin **ellipses**,**f(...)**.

Tip actual	**Tip formal**	**Descrierea conversiei**
T	T&	De la un obiect de tip **T**, la o referinta catre un obiect de tip **T**
T&	T	De la o referinta catre un obiect de tip **T**, la un obiect de tip **T**
T[ ]	*T	De la un tablou de obiecte de tip **T**, la un pointer catre un obiect de tip **T**
T	const T	De la un obiect de tip **T**, la un obiect constant de tip **T**
T	volatile T	De la un obiect de tip **T**, la un obiect volatil de tip T
F(args)	(*F)(args)	De la o functie cu tipurile argumentelor specificate la un pointer catre o functie cu aceeasi lista de argumente

Prin variabila volatila, indicata cu ajutorul cuvantului cheie **volatile**, intelegem o variabila ce poate fi modificata prin intermediul unei rutine. Indicatorul este utilizat asemeni lui **register** sau **auto**, variabila fiind insa incarcata in memorie si nu intr-un registru al calculatorului. Putem avea chiar si clase volatile.

Compilatorul cauta numai tipul argumentelor si nu verifica tipul valorii returnate de catre functie. Astfel, doua functii nu pot sa difere numai prin tipul acestei valori.

De asemenea, este un bun stil de scriere a programelor acela de a utiliza, ori de cate ori este posibil, conversia explicita de tip, pe langa claritatea parcurgerii programului castigand si claritatea preluarii de catre compilator a acestuia.

Multe din regulile de prelucrare implica efectuarea diferitelor tipuri de conversii. In unele cazuri, compilatorul ar putea executa multiple astfel de conversii in vederea stabilirii listei exacte de parametri.

Este bine ca, atunci cand este posibil, lista parametrilor efectivi sa contina conversiile explicite in vederea obtinerii unei claritati a programului.