Cursul 8
FUNCTII (II)

OBIECTIVE

- exemplificam folosirea pointerilor la functii, a vectorilor de pointeri la functii si a pointerilor la functii ce returneaza pointeri.
- exemplificam utilizarea functiilor recursive.

PREZENTARE GENERALA

In procesul compilarii programului, numele variabilelor sunt transformate in adrese de memorie unde sunt stocate si unde pot fi recuperate datele. Si pointerii la adrese pot accesa aceste adrese. Aceasta etapa de prelucrare se aplica atat variabilelor, cat si functiilor. Compilatorul transforma numele unei functii intr-o adresa de cod executabil.

Limbajul C extinde strategia manipularii variabilelor prin pointeri la functii. La fel ca oricare alt pointer si pointerul la o functie trebuie initializat inainte de a fi folosit, aceasta realizandu-se prin atribuirea numelui functiei, pointerului.

Variabilele pointer care contin adresa unei functii permit:

-transferul functiei asociate, ca parametru;
-apelul functiei prin intermediul pointerului.

Declararea unui pointer la o functie se face astfel:

tipRezultat (*pointerFunctie) (listaParametri);

Aceasta forma precizeaza compilatorului ca **pointerFunctie** este un pointer la o functie care intoarce un rezultat de tipul **tipRezultat** si care are o lista de parametri.

Exemplu:

```
double (*fx)(double x );
void (*sortare )(int* tablInt,unsigned n);
```

```
unsigned (*cautare ) (int cheieCautare , int *tablInt , unsigned *n);
```

Initializarea unui pointer la o functie se face astfel:

```
pointerFunctie = numeFunctie ;
```

Functia atribuita trebuie sa intoarca un rezultat de acelasi tip cu cel intors de pointerul la functie si sa aiba aceeasi lista de parametrii cu acesta. In caz contrar compilatorul semnalizeaza eroare.

Exemplu :

```
void (* sortare )(int* tablInt,unsigned n);
sortare=qsort;
```

Apelul pointerilor la o functie se face astfel:

```
(*pointerFunctie) (<de argumente>);
(*pointerFunctie[indice])(<de argumente>);
```

Exemplu:

```
(*sortare)(&tablInt,n);
(*sortare[0])(tablInt,n);
```

Prin recursivitate se intelege, in programare, proprietatea unui functii de a se putea apela pe ea insasi. Apelul unei functii poate sa apara si in definitia sa. In acest caz, functia se numeste recursiva. Nu toate limbajele de programare suporta functii recursive. Un algoritm in a carui descriere este necesara referirea la el insusi se numeste algoritm recursiv.Aceasta clasa de algoritmi este frecvent intalnita si ofera descrieri simple si elegante ale operatiilor de efectuat. Pentru programarea lor este necesara folosirea functiilor recursive.

In C nu exista restrictii speciale pentru functii recursive. Trebuie mentionata insa problema generala a iesirii din recursivitate. Practic este necesar ca apelul recursiv sa apara in blocul unei instructiuni de decizie sau ciclare a carei conditie este modificata de functie astfel incat sa opreasca secventa de apeluri.

Exemplul clasic il constituie calculul factorialului, pe baza relatiei :

```
n!=n*(n-1)!
```

Exemplu:

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

void main()
{
    printf("%d",fact(10));
}
```

Se observa ca incheierea apelurilor recursive are loc cand se ajunge la apelul **fact(0)**. Cand este apelata o functie, parametrii si datele locale sunt salvate in stiva.

Astfel, cand o functie este apelata recursiv functia incepe executia cu un nou set de parametri si variabile locale, dar codurile care constituie functia raman aceleasi. La fiecare apel, se consuma timp pentru transferul parametrilor si rezultatului si se incarca stiva suplimentar cu obiectele asociate parametrilor si rezultatului. Stiva este eliberata treptat abia in secventa de reveniri. Din aceste motive, functiile recursive pot deveni neconvenabile. Desi in principiu este posibila trecerea de la orice algoritm recursiv la algoritmi nerecursivi, rezultatul este greoi de urmarit si de programat. De multe ori, datorita simplitatii, este preferata folosirea functiilor recursive, daca viteza de executie este acceptabila.