CAPITOLUL 10

10. Noțiunea de funcție

Definiție: O funcție C este o colecție de operații specifice limbajului

Până acum s-a discutat despre programe extrem de simple. De asemenea s-au folosit apeluri de funcții predefinite, al cărui cod este gata scris. Se poate spune că o funcție este o bucată de program separată (subprogram) care este folosită în mai multe scopuri:

1. Realizarea unui cod sursă cât mai clar.
2. Simplificarea proiectării unei aplicații (prin structurare).
3. Realizarea de biblioteci de funcții, deci creşterea vitezei de dezvoltarea a aplicațiilor ulterioare

Pentru introducerea acestui concept în mod corect trebuie discutate întâi un minim de tehnici de proiectarea a unei aplicații şi aceasta deoarece o funcție nu se introduce în cadrul unui program din motive "artistice" ci are o justificare la nivel logic foarte clară.

Ca tehnici de proiectare există două metode: *top-down* (de sus în jos) şi *bottom-up* (de jos în sus). Proiectarea unei aplicații constă în descompunerea logică pe submodule.

Funcție de complexitatea acestei aplicații descompunerea poate continua recursiv la nivelul fiecărui modul rezultat până când:

- acesta este extrem de simplu de realizat
- este deja implementată respectiva parte într-o bibliotecă a compilatorului
- este deja dezvoltată anterior
- datorită simplității programului nu mai are sens.

În cazul metodei de proiectare top-down proiectarea decurge exact cum am descris mai sus. Uzual în programarea C se foloseşte această metodă. În cazul bottom-up se porneşte de la un set de funcții şi operațiuni existente şi se construiesc module care la rândul lor vor construi recursiv alte module, până când rezultă aplicația. Această metodă este folosită uzual în cazul proiectării aplicațiilor pentru bazele de date.

Totuşi este greu de impus un anumit tip de gândire când se realizează proiectarea unei aplicației. Există o teorie foarte bine fundamentată şi documentată privind aceste probleme. Într-o primă fază esențiale sunt experiența avută în proiectarea unui anumit tip de aplicație, şi apoi experiența de programare.

Deja pentru aplicații foarte mari se lucrează în echipe cu membri extrem de specializați în câte o fază a procesului de proiectare, implementare, testare. Însă indiferent cât de complicată ar fi distribuția unei echipe programatorului propriu zis i se dau fie specificațiile relativ la ce va trebui să facă partea de cod care o dezvoltă fie în anumite cazuri algoritmul. În ambele situații modul în care îşi descompune aplicația pentru a o putea realiza nu priveşte pe nimeni. La sfârşit aceasta trebuie să funcționeze corect şi atât.

Pasul imediat următor în aplicarea metodei top-down este rafinarea paşilor

Exemplu: Cum s-ar proiecta un program care să "Numere cuvintele dintr-un fişier" ?

Descriere în pseudocod

```
Deschide fişierul
atâta timp cât mai sunt cuvinte in fişier fă
{
        citeşte un cuvânt
        creşte numărul de cuvinte cu 1;
}
Închide fişierul.
Afişează numărul de cuvinte
```

Rafinarea paşilor

```
Deschide fişierul
Atâta timp cât mai sunt cuvinte in fişier
{
        citeşte caractere până se întâlneşte un caracter diferit de ' '
        citeşte caracterele până găsesc un ' '
        incrementează( nr_cuv )
}
Închide fişierul
Afişează ( nr_cuv )
```

10.1. Declararea şi definirea funcțiilor

Funcția *main()* a fost folosită în toate exemplele de până acum. De câte ori un program C este executat el începe cu prima instrucțiune din funcția *main()*. Bibliotecile C ne pun la dispoziție o sumă variabilă de rutine precompilate. Totuşi pentru a putea rezolva probleme mai complexe este necesar să se proiecteze (declarare şi definire) propriile funcții.

10.1.1. Declararea funcțiilor

O funcție este declarată exact ca o variabilă, iar tipul de dată pe care-l returnează trebuie specificat. În C trebuie avut grijă ca numele unei funcții proprii să nu coincidă cu numele unei funcții de bibliotecă. În caz contrar link-editorul nu va şti la generarea de legături între fişierele obiect, dacă apelul respectivei funcții se referă la rutina precompilată existentă deja sau la corpul funcției proprii. Se poate chiar mai rău să nu se genereze un mesaj de eroare, acest lucru se întâmplă când prototipurile celor două funcții se potrivesc, iar acest tip de eroare este extrem de greu de detectat.

De asemenea trebuie specificate şi argumentele ce vor fi transmise funcției la apel (tipul, ordinea lor). O formă generală pentru declararea unei funcții este:

<tip_dată> nume_funcție(<listă_argumente>);

unde :

<listă_argumente>={tip_dată nume_arg1, tip_dată nume_arg2, … tip_dată nume_argn}.

Funcțiile sunt declarate înainte de funcția *main()*. Acest lucru permite ca oricare funcție să fie apelată de oricare alta inclusiv de ea însăşi (cazul funcțiilor recursive).

Se pot declara funcții şi în interiorul altor funcții dar în acest caz ele nu pot fi apelate decât în interiorul funcției unde au fost declarate. De aceea metoda de declarare globală a lor este cel mai des folosită.

Este necesară să se realizeze specificarea tipului de dată pentru argumente şi valoarea returnată. Acest lucru permite compilatorului să verifice dacă funcțiile sunt apelate corect şi să realizeze orice conversie implicită ar fi necesar.

De exemplu un apel de funcție ca mai jos:

```
#include <math.h>
...
double x;
...
x=sqrt(4);
...
```

poate fi considerat ca incorect deoarece funcția radical are ca parametru de intrare o variabilă de tip double. Datorită faptului că funcția este declarată în cadrul bibliotecii math.h care este inclusă în program, compilatorul va face conversia de la întreg la double a valorii 4 astfel încât la apel funcția radical va primi o valoare corectă (4.0).

10.1.2. Declarații implicite

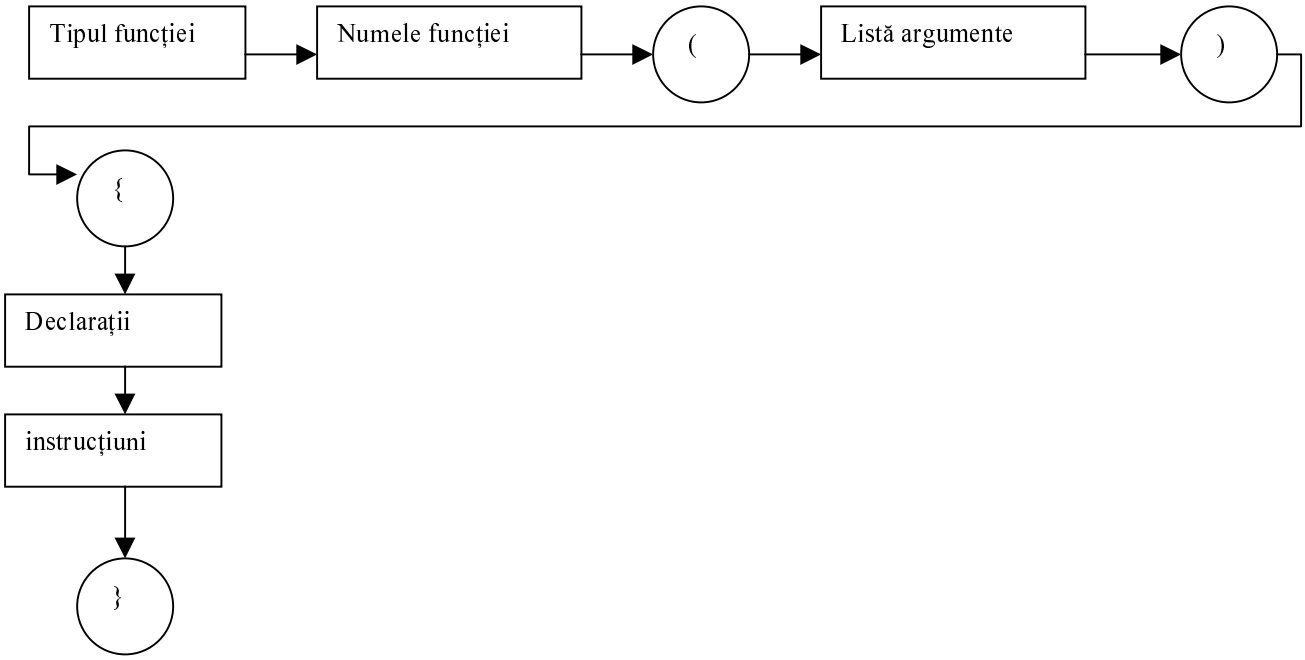
Dacă nu este specificat nici un tip de dată la valoarea pe care trebuie să o returneze o funcție atunci compilatorul va considera că funcția returnează un întreg. Similar dacă funcția nu este declarată, compilatorul va considera că returnează un întreg şi va da eroare dacă în program este folosită altfel.

Totuşi dacă se ține cont numai pe tipul implicit compilatorul nu va şti să facă conversiile de tip la apel.

10.2. Definirea funcției

Definirea unei funcții se referă la o bucată de cod C care rezolvă o anumită problemă (aceea pentru care s-a introdus funcția). Această definiție se face în afara corpului funcției main() sau al altei funcții şi are forma generală dată mai jos:

Structura generică a unei funcții C



Exemplu:

```
   tip    nume      declarație argument
int          pw2          ( int num )
{                                  nume argument
 int rez;  ──────────►  declarație variabilă locală
rez = num * num;
return rez;                instrucțiuni C
}
```

corpul funcției

sau:

<tip_dată> nume_funcție(<listă argumente>)
{
<declarații variabile locale>
<instrucțiuni>
return <variabilă/expresie>
}

Prima linie trebuie să fie identică cu cea din declarație. Grupul de variabile locale (automate) care pot fi folosite în interiorul unei funcții constă în cele declarate în lista de argumente sau la începutul funcției.

Ca şi în cazul funcției *main()* variabilele trebuiesc declarate imediat după paranteza deschisă şi înainte de alte instrucțiuni.

Ca un exemplu de funcție să luăm cazul calculului rădăcinii pătrate dar cu specificarea preciziei dorite. Se va folosi metoda Newton Raphson pentru aproximarea rădăcinii.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>

double radical(double x,float precizie);

void main(void)
    {
    float nr, prec;
    double rezultat;
    clrscr();
    printf("Dati numarul");
    scanf("%f",&nr);
    printf("\nDati precizia ");
    scanf("%f",&prec);

    switch((rezultat=radical(nr,prec)))
        {
        case –1.0: printf("Eroare, Numar Negativ"); break;
        case –2.0:printf("Eroare, Precizia este intre 0 si 1 ");break;
        default: printf("\nRadicalul este %lf",rezultat);
        }
    printf("\n Press any key to continue!");
    getch();
    }

double radical(double x,float precizie)
    {
    double x_vechi, x_initial=x;
    if(x<0.0)return –1;
    if(!x)return x;
    if((precizie>1)||(precizie<0))return –2;
    do
    {
        x_vechi=x;
        x=0.5*(x+x_initial/x);
    }while(fabs((x-x_vechi)/x)>precizie);
    return x;
    }
```

Se observă că aproape jumătate din corpul funcției tratează erorile ce pot apărea. Deşi la prima vedere acest lucru apare ca un efort suplimentar, nejustificat în programare, în momentul în care este cazul unei aplicații cu peste 20 funcții care se apelează unele, pe altele codul este relativ greu de urmărit şi depanat în caz contrar.

În condițiile în care s-a prevăzut fiecare funcție de la început cu aceste verificări atunci este extrem de uşor ca să se poată depista erorile. Dacă această politică este aplicată la orice nivel al programului va conduce la scrierea unui cod stabil. Stabilitatea se referă la faptul că dacă apare undeva în cursul execuție o eroare datorată fie condițiilor externe (intrări greşite) fie condițiilor interne (de exemplu eroare de programare sau virus) aceasta va fi semnalată utilizatorului. Ea se va propaga pe cale ierarhică din apel în apel de funcție, fără ca programul să iasă din parametri normali de funcționare, nu se pierde controlul asupra aplicației.

Pentru realizarea unor programe care să primească parametri de intrare din linia de comandă funcția *main()* poate primi un set de parametri care să conțină exact sirurile de caractere din linia de comandă. Mai jos este dat un exemplu simplu de folosirea a acestei facilități şi anume un program care va afişa parametrii primiți prin linia de comandă.

```
#include <stdio.h>
void main(int argc, char **argv)
{
    int i;
    printf("argc = %d\n\n",argc);
    for (i=0;i<argc;++i)
     printf("argv[%d]: %s\n",i, argv[i]);
}
```

Se observă că primul *arc* specifică numărul de lemente din care este compusă linia de comandă, iar *argv* reține adresa de început a zonei de memorie în care sunt stocate acestea.

10.3. Tipul de dată void

Geneza pointerului void

A fost introdus de standardul ANSI pentru a rezolva următoarea ambiguitate de sintaxă:

Ştim că avem posibilitatea de a aloca memorie.

Funcția întoarce un pointer către începutul zonei de memorie alocată (indiferent că suntem sub DOS, WINDOWS sau UNIX). Întrebarea este de ce tip este acest pointer sau, tradus, cum va fi interpretată zona contigua de memorie rezervată .

1) Ca fiind grupată în elemente de lungime 1 octet(signed sau unsigned sau char)
2) Ca fiind grupată în elemente de lungime 2 octeți (signed sau unsigned)

ş.a.m.d.

Pentru a rezolva această semiambiguitate d.p.d.v. al compilatorului şi nu numai, s-a decis ca pointerul returnat să fie de tip void. Un tip inexistent, dar convertibil fie explicit prin cast (K&R standard), fie implicit prin asignare (ANSI) la tipul variabilei careia i se asignează.

În limbajul C o funcție trebuie să returneze un tip de date la terminarea execuției. Totuşi există cazuri când nu se fac calcule directe ci se realizează anumite acțiuni cum ar

fi de exemplu afişarea unui mesaj. În aceste cazuri, totuşi trebuie specificat compilatorului faptul că funcția nu returnează nimic.

Există şi cazul invers adică o funcție care nu primeşte nici un parametru dar returnează rezultate (cazul citirii unui port de date). Pentru a putea specifica aceste situații se foloseşte tipul de dată void care în traducere înseamnă: lipsit de conținut, nimic.

Transferul Argumentelor

Argumentele unei funcții, reprezintă căile de transfer a datelor către funcție. Multe limbaje de programare transferă argumentele prin referințe, ceea ce înseamnă că trimit un pointer către argument. În C argumentele sunt trimise prin valoare ceea ce înseamnă că o copie a argumentului este trimisă funcției. Funcția poate modifica valoarea copiei, dar nu poate modifica valoarea care o are argumentul în rutina apelantă.

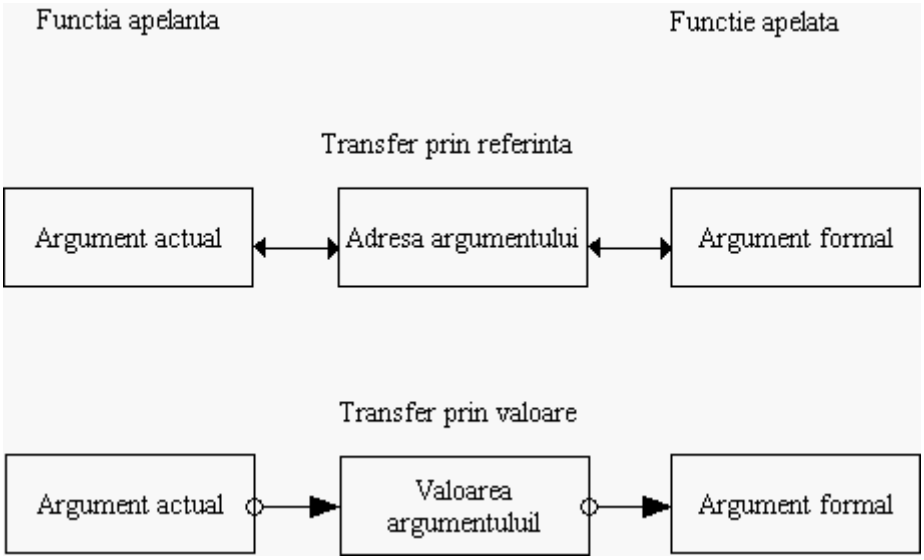


Figura arată diferența; se observă că săgeţile de la transferul prin referință, sunt bidirecționale, ceea ce specifică faptul că modificările locale se repercutează direct.

Argumentul trimis este de obicei numit argument actual, în timp ce copia recepționată este numită argument formal sau parametru formal.

Deoarece C transferă argumentele prin valoare, o funcție poate asigura noi valori la argumentele formale fără a le afecta pe cele actuale. De exemplu:

```
#include<stdio.h>
void main(void)
        {extern void f( );
        int a=2;
        f(a);
        printf("y=%d \n ", a);
        exit(0);
        }
```

```
void f (int arg)
        {
        arg=3;
        }
```

va avea ca rezultat afişarea valorii 2, modificarea copiei nemaiafectând valoarea din funcția apelantă. Dacă se doreşte ca funcția să poată schimba valoarea unui obiect, atunci trebuie ca aceasta să primească adresa obiectului, deci să i se trimită un pointer la obiect, ca în exemplul următor:

```
void swap(int * x,int* y)
        {
        register int temp;
        temp=*x;
        *x=*y;
        *y=temp;
        }
void main(void)
        {
        int a=2,b=3;
        swap (&a, &b);
        printf("a= %d , b=%d\n",a,b);
        }
```

unde se va afişa a=3 şi b=2;

Cu transferul prin referință ne-am întâlnit de la primele lecții: scanf("%d",&nr).

Declarații şi apeluri

Funcțiile pot apărea într-un program

→ Definiție: o declarație care defineşte ce face funcția precum şi numărul şi tipul argumentelor ei.

→ Function allusive: se declară o funcție definită în altă parte. Acest lucru specifică ce tipuri de valori întoarce funcția, precum şi ce tipuri de valori primeşte

→ Apelul funcției : apel propriu-zis la întâlnirea lui în contextul execuției programului Sare la zona corpului funcției execută ce este acolo şi apoi se reîntoarce la locul apelului.

Declararea argumentelor

Declararea argumentelor se supune la aceleaşi reguli ca oricare declarare de variabile cu următoarele excepții:

1. singura clasă de stocare permisă este register
2. char şi short sunt convertite la întregi; float sunt convertite la double. Noile facilități ANSI C permit deyactivarea acestor conversii aotomat
3. un argument formal declarat ca funcție este convertit la pointer către funcție
4. se poate să nu includ un inițializator într-o declarație de argumente

Corpul funcției

Corpul unei funcții este delimitat de un set de paranteze {}. El poate fi gol, fapt ce este folositor în diverse perioade a dezvoltării unui program.

Unul din primele lucruri în cazul proiectării unei aplicații mari, este definirea unui set de operații de nivel înalt ce corespund ca funcționare. În aceste stagii este folositor să am un program, funcția, declarații, parametrii specificați, dar corpul gol. La apel ea nu va executa nimic. Ulterior aceste corpuri vor fi completate.

Valorile returnate (întoarse)

O funcție poate returna o singură valoare prin intermediul cuvântului cheie return. Această valoare poate fi un pointer pentru tablou(valori multiple) sau o funcție. Indirect însă şi acestea pot fi întoarse, dacă se foloseşte un pointer la acestea ca valoare atinsă. O structură sau uniune pot fi returnate direct, deşi acest lucru nu este recomandat datorită dimensiunilor mari ale lor.

Corpul unei funcții poate conține oricâte apeluri de return. La primul întâlnit, execuția funcției este încheiată; în lipsa totală a unui return, întorcerea în funcția apelantă, se face atunci când se întâlneşte marcatorul de sfârşit de bloc " } ".

Valoarea întoarsă trebuie să fie compatibilă cu tipul funcției. Ex. de caz în care valoarea întorsă este diferită, este dat mai jos, dar în acest caz conversia la valoarea la valoarea returnată se face automat.

ATENȚIE!!! Acest lucru poate crea o groază de neplăceri fiind o eroare extrem de greu de găsit.

```
Float f(void)
{
        int a;
        char c;
        float f2;
if ( a > (c+0x30))
        {
        f2=a;   // a convertit la float
        return a;
        }
else
        {
        f2=c;  //c convertit la float
        return c;
        }
}
```

Limbajul C este foarte riguros relativ la potrivirea tipurilor de pointeri. În exemplul următor f( ) este declarată ca o funcție ce întoarce un pointer către char, sunt prezente unele posibilități legale sau ilegale de întoarcere de rezultat.

```
Char* f( )
{
        char **cm, *cv, *cv1, ca[10];
        int *ip, *ip1;
        cv=cv1;    //admis, au acelaşi tip
        return cv1;  //admis, cv1 este char*

sau
        cv1= *cm;   // admis, au acelaşi tip
        return *cm;  //admis, *cm este char*
sau
        cv=ca;   //admis, au acelaşi tip
        return ca; //admis, ca este char*
sau
        cv=*cv1; //eroare, cv este char*, iar *cv1 este char**
        return *cv1; //eroare din acest motiv
sau
        cv1=ip1; //eroare char* ≠ int*
        return ;   // Atenție!!! Produce un comportament impredictibil, se aşteaptă  o
                   // valoare să fie întoarsă
```

Aluzii la funcții

O aluzie la funcție reprezintă o declarație a unei funcții care este definită în altă parte, de obicei într-un fişier sursă diferit. Implicit toate funcțiile sunt presupuse a returna int.

Scopul principal al redeclarării este de a indica prototipul (ca compilatorul să ştie cum să transmită variabilele la apelul ei), precum şi clasa de stocare. În cazul în care clasa de stocare nu este precizată, ea este considerată extern implicit. Acest lucru înseamnă că definiția funcției precum şi corpul ei pot apărea în oricare din fişierele sursă a aplicației, inclusiv în cel curent. Dacă se omite şi specificarea tipului, atunci implicit este int.

Scopul în cazul unei aluzii la funcție este tipic ca la oricare altă variabilă, apelurile în bloc au scop bloc, restul au scopul fişier.

Conversiile automate de argument

În absența prototipului (alt motiv pentru aluziile la funcții) toate argumentele mai mici ca int sunt convertite la int, iar toate argumentele float sunt convertite la double. Deci poate fi o sursă bine ascunsă de erori.

Pointeri la funcții

Sunt instrumente puternice pentru ca ne permit o modalitate elegantă de apel pentru diferite funcții, bazându-se pe datele de intrare.

Înainte de a discuta pointerii la funcție, trebuie vorbit puțin despre modul în care compilatorul interpretează declarațiile funcțiilor precum şi apelurile acestora.

Sintaxa pentru declararea şi invocarea funcțiilor este foarte apropiată de sintaxa folosită în cazul tablourilor. De ex. în declarația int v[5], v este pointer către primul

element al tabloului. Atunci când simbolul este urmat de un nume/număr între paranteze, pointerul este indexat cu acel număr apoi referit. Un proces similar apare şi în cazul funcțiilor.

Fie declarația int f( ); în acest caz f este un pointer la funcție. Când el este urmat de o listă de argumente inclusă între paranteze, el este referit (alt mod de a spune că o funcție este apelată). De asemenea, dacă int v[5] se referă la un pointer constant la fel şi în cazul int f( ), f este tot un pointer constant, însă este ilegal să se asigneze o valoare lui f..

Pentru a declara o variabilă pointer către o funcție, numele pointerului trebuie precedat de *; de exemplu int (*pf)( ) este un pointer către o funcție ce întoarce un întreg. De fapt, se declară o variabilă de tip pointer, care este capabilă să rețină un pointer către o funcție ce întoarce un întreg.

Parantezele sunt necesare, deoarece în lipsa lor pf ar fi considerată o funcție care întoarce un pointer la întreg.

Asignarea unei valori pentru pointerii la funcții se realizează astfel: fie funcția int fat_boy( ) şi pointerul la o funcție ce întoarce un întreg int (*pf) ( ). Asignarea se relizează direct pf=fat_boy. Devin astfel clare următoarele erori:

1. pf=f1( ) unde pf este pointer la funcție iar f1( ) este un întreg
2. pf=&f1( ) nu se poate obține adresa rezultatului unei funcții deoarece transferul se face prin stivă
3. pf = &f1 unde pf este pointer la int iar &f1 este pointer la pointer

Convenții de tip în cazul de întoarcere parametrii

Nu trebuie uitat că tipul de date întors de funcție trebuie să fie în coincidență cu tipul de pointer la funcție declarat.

De ex. fie declarațiile:

```
int if1( ) , if2( ) , (*pif)( );
float ff1( ) , (*pff)( );
char cf1( ) , (*pcf)( );
```

şi secvența de asignări :

```
pif = if1;  //legal, tipurile sunt identice
pcf=cf1;   //ilegal, nepotrivire de tip
pff=if2;    //ilegal, nepotrivire de tip
pcf=cf1;   //legal
if1=if2;     //ilegal, se încearcă asignarea unei noi valori la o constantă
```

Apelul unei funcții folosind pointeri

Ex.:

```
int f1( );
int (*pf)( );
int rez;
…
pf=f1;  //am asignat pf la funcție
...
rez= (*pf)(a);   //am apelat funcția f1 cu a ca parametru
```

Obs: (*pf)(a) este ECHIVALENT cu (****pf). Sintaxa ANSI permite ca în loc de (*pf)(a) să folosesc pf(a). Pentru portabilitatea unui program este indicat să se folosească prima formă.

Întoarcerea ca rezultat de pointer la funcții

O funcție poate întoarce un pointer la altă funcție, dar trebuie avut grijă despre modul în care se va face declararea acestei funcții. De exemplu:

```
Int (*f(float x, float y))( )
sau
int (*f(x,y))( )
float x,y;
{
…
}
```

declară o funcție ce are variabile de intrare x,y şi care întoarce un pointer către o funcție ce întoarce un întreg.

Să luăm un exemplu mai concret de folosire a acestor facilități şi anume o funcție de sortare, optimală din punct de vedere al vitezei.

Să analizăm puțin problema. Există mai mulți algoritmi de sortare, fiecare cu avantaje şi dezavantaje:

Quicksort

→ foarte rapid pentru date aleator aranjate

→ ineficient dacă datele sunt parțial sortate pe grupuri

Mergesort

→ unul dintre cei mai eficienți, dar cere memorie

Heapsort

→ cel mai indicat pentru arii masive de date, deoarece necesită cea mai puțină memorie

Pp. că în urma unor analize putem decide care dintre cei trei algoritmi vor fi utilizați. Poate fi scrisă următoarea funcție:

```
Void (*my_sort (float in_data)( )
        {
        extern void quick_sort( ) , merge_sort ( ) , heap_sort ( );
        // heap_sort poate lipsi
        // se realizează analiza datelor . Să presupunem că decizia va fi
        //depusă într-un flag
        switch ( flag )
                case 1: return quick_sort;
                case 2: return merge_sort;
                case 3: return heap_sort;
                default: puts(" Error unknown ");
                        exit(1) break;
        }
la apel ( my_sort (list)) ( list );
```

Recursivitatea

O funcție recursivă este una care se autoapelează. De ex.:

```
void recurs (void)
{
static cont=1;
printf("%d \n ", cont);
cont++;
recur( );
}
void main (void)
{
recur();
}
```

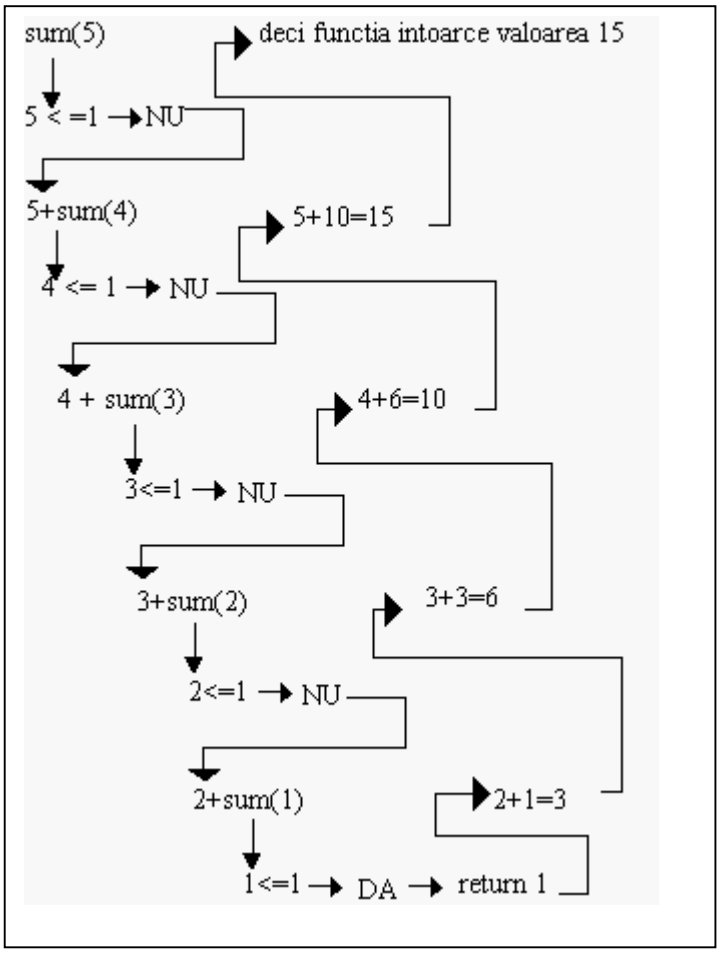
```
void recur (void)
{
static cont=1;
if (cont > 3 ) return;
else
{
printf(" %d \n ", cont);
cont++;
recur( );
}
}
```

La fiecare apel de funcție în stivă sunt salvați o serie de parametrii. Programul de mai sus va autoapela funcția la $\infty(1,2,3...)$, dar stiva PC este finită, având ca rezultat o blocare generală. De aceea orice funcție recursivă trebuie să aibă o condiție, numită caz de bază, care să permită stoparea acestui proces. Un exemplu simplu ar fi în cazul nostru limitarea contorului. Deci:

Întrebări:

1.Ce se întâmplă dacă ştergem else şi cele două acolade de după el ?

2.Ce se întâmplă dacă cont nu e declarat static ?



Folosirea variabilelor fixe este o cale de a controla recursivitatea , după cum am văzut anterior. În exemplul de mai jos se calculează $\sum_{i=1}^{n} i$, i∈ N, recursiv.

```
Int sum(int n)
{
if (n<=1) return n;
else return (n+sum (n-1));
}
```

Pt. n=5 să vedem cum se execută pas cu pas

(Clase de stocare) Storage classes

Majoritatea aplicațiilor mari sunt scrise de echipe de programatori. După proiectarea (schițarea) aplicației în comun fiecare programator pleacă şi realizează o bucată izolată de program. Izolarea se referă la faptul că la realizarea acelui program nu se ține cont de ceilalți, ci la funcțiile care trebuie să le îndeplinească programul. La sfârşit aceste programe sunt unite (linked) împreună pentru a forma aplicația dorită.

Pentru ca în această fază să nu apară probleme majore trebuie să existe un mecanism ce asigură ca variabilele declarate de un programator să nu intre în conflict cu variabilele cu acelaşi nume folosite de un alt programator (de ex. variabilele de contor ale unei bucle: i,j,k). Pe de altă parte este necesar ca anumite date să fie comune ,vizibile şi/sau modificabile de către mai multe surse ce provin de la autori diferiți. Deci trebuie să existe un mecanism care să asigure că numele variabilei declarate în diverse fişiere se referă la aceleaşi locații memorie.

În C se defineşte

→ Cand trebuie să fie o variabilă comună sau împarțită (shared)

→ În ce parte de cod este valabil acest lucru prin atribuirea unui scop (scope) unei declarații

"Scope" este un termen tehnic care denotă regiunea textului sursei C în care numele (declarat) al unei variabile este activ(există).

Altă proprietate a variabilelor este durata care se referă la (descrie) durata de viață în memorie a unei variabile.

Variabilele cu durată fixă sunt garantate că-şi rețin datele chiar după ce şi-au îndeplinit scopul.

În schimb în cazul variabilelor cu durată automată acest lucru nu mai poate fi garantat.

Definiție:**Prin clasă de stocare (storage class) se înțelege scopul şi durata unei variabile.**

Fie următorul segment de program:

```
void f1(void)
{int j;
static int vect[]={1,2,3,4};
…
}
```

Se observă că există două variabile: j şi vect. Ele au un scop de bloc deoarece sunt declarate în interiorul acestuia. Variabilele cu scop de bloc sunt de obicei numite variabile locale. Implicit, variabilele locale au durata automată. În cazul lui j, el va avea atribuită o nouă adresă de memorie la fiecare execuție a respectivului bloc. În cazul lui vect datorită folosirii cuvântului cheie static atribuirea unei adrese de memorie se face o singură dată şi se păstrează pe toată durata execuției programului.

Durata fixă față de cea automată

După cum o precizează şi numele, o variabilă fixă este una staționară, în timp ce pentru o variabilă automatică, zona de stocare în memorie este aleasă automat în timpul execuției programului. Deci o variabilă fixă are memorie alocată, memoria la începerea execuției programului şi-şi păstrează această alocare pe toată durata execuției acestuia.

Variabila automatică se referă la zona de memorie ataşată ei doar atâta timp cât codul în care-şi are scopul este executat. O dată încheiat scopul(zona unde era fixată) compilatorul asigură eliberarea zonei de memorie ocupată de aceasta.

Variabilele locale (cu scopul limitat la nivel de bloc) sunt implicit automatice. Ele pot fi forțate să lucreze static prin folosirea cuvântului cheie static.

Inițializarea variabilelor

Diferența dintre variabilele fixe şi cele automate este importantă în special pentru variabilele neinițializate. În cazul celor automate, ele vor fi reinițializate la fiecare trecere prin zona lor de existență.

```
void mc(void)
{
        int j=1;
        static int k=1;
        j++;k++;
        printf("j=%d \t k=%d \n",j,k);
}

void main(void)
{
        mc();                   j=2     k=2
        mc();                   j=2     k=3
        mc();                   j=2     k=4
}
```

Altă diferență apare la inițializarea implicită a variabilelor:

→ variabilele automate nu sunt inițializate

→ variabilele statice sunt implicit inițializate cu 0

Ex:

```
void mc1(void)                          void main(void)
    {                                   {
        int j;                              mc();
        static int k;                       mc();
        j++;k++;                            mc();
        printf("j=%d \t k=%d \n",j,k);  }
    }
```

Cu rezultatul: j=3604481 k=1
j=3265349 k=2
j=3456645 k=3

Altă diferență între variabilele cu durată fixă şi cele automate se referă la tipurile de expresii ce pot fi folosite în utilizare.

De ex. următoarele declarații sunt legale:

```
int j=0;k=1;
int m=j+k;
float x=3.141*2.3;
```

în timp ce următoarele nu, deoarece j şi k apar în expresii înainte de a fi declarate:

```
int m=j+k;
int j=0;k=1;
```

Regulile de inițializare a variabilelor cu durată fixă sunt foarte stricte. Inițializarea se poate face numai cu o expresie constantă (deci NU poate conține nume de variabilă).

Folosirea variabilelor cu durată fixă

Un exemplu clasic de folosire a acestora este contorizarea numărului de execuții ale unei funcții pentru a-i putea modifica execuția la intervale regulate.

De ex. să presupunem că avem un program care formatează un text citit dintr-un fişier şi scrie rezultatul în alt fişier. Una din funcții este este print_header () apelată la începutul fiecărui program nou. Eventual aceasta poate fi diferit funcție de paritatea/imparitatea numărului de pagini.

```
#define ODD  0
#define EVEN 1
print_header (char* chop_title)
        {
        static char page_type =ODD;
        if (page_type==ODD)
                {
                printf("\t\t\t\t %s\n\n",chop_title);
                page_type=EVEN;
                }
        else
                {printf("%s\n\n",chop_title);
                page_type=ODD;
                }
        }
```

"Scopul" unei variabile (Scope)

În cadrul proiectării unui program trebuie ținut cont de faptul că există 4 tipuri de regiuni fiecare cu un scop bine definit, în care se poate accesa o variabilă prin nume şi anume: program, funcție, bloc, fişier.

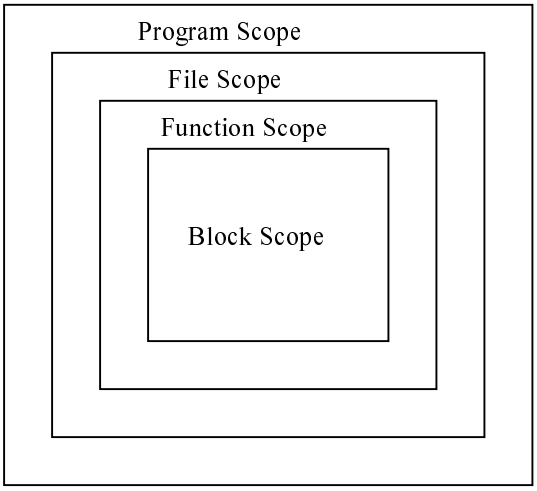
La nivel de program (program scope) înseamnă că variabila este activă între diferitele fişiere sursă ce corespund programului executabil. Variabilele de acest tip sunt cunoscute de obicei sub numele de variabile globale.

La nivel de fişier (file scope) se referă la faptul că respectiva variabilă este activă de la punctul de declarații până la sfârşitul fişierului sursă ce o conține.

La nivel de funcție (function scope) variabila este activă doar în cadrul funcției.

La nivel de bloc de instrucțiuni (block scope) variabila e activă de la punctul de declarație până la delimitatorul de sfârşit de bloc**.**

În general scopul unei variabile este determinat de locul în care se realizează declararea acesteia.



Ex.:
```
int I;          /* Program Scope */
static int j;   /* File Scope */
func(int k)   /* Program  Scope */
{int m;         /* Block Scope */
start:          /* Function Scope */
…
```

Această ierarhizare permite ca în două funcții diferite să avem variabile cu acelaşi nume sau de ex.:

```
int j=10;
main()
        {
        int j;
        for (j=0;j<5;++j)
                printf("j=%d\n",j);          lucrează normal dar dpdv al altui
        }                                    fişier sursă j=10;
```

10.5. Variabile globale (program sau file scope)

Până acum s-a discutat numai despre variabile locale. Există totuşi cazuri când se doreşte ca o valoare depozitată într-o variabilă să fie vizibilă sau modificabilă din interiorul oricărei funcții. Pentru aceasta este suficient ca respectivele variabile să fie declarate în afara corpului oricărei funcții şi ele vor căpăta astfel un caracter global. Acest tip de variabile sunt automat inițializate cu valoarea zero.

În general în programarea structurată se recomandă evitarea excesului de variabile globale, întrucât nu sunt neapărat necesare decât eventual ca să scriem mai repede şi uşor un cod, dar acest cod nu este neapărat cel mai bun sau cel mai stabil. Mai jos vom prezenta un exemplu simplu relativ la folosirea variabilelor globale.

```
#include <stdio.h>
#include <conio.h>

int global_v1;

int ori_doi(int x);

void main(void)
{
int local_v1;
clrscr();
printf("Variabila globala are valoarea %d\n", global_v1);
printf("Variabila locala are valoarea %d\n", global_v1);
local_v1=ori_doi(local_v1);
printf("Variabila globala are valoarea %d\n", global_v1);
printf("Variabila locala are valoarea %d\n", global_v1);
}

int ori_doi(int x)
  {
   int retval;
   retval = x * 2;
 global_v++;
 return retval;
  }
```

Tematica claselor de stocare ANSI

locul unde se declară / Specificator de clasă	În afara funcției	în interiorul funcției	argumente funcție
auto sau register	NEPERMIS	Scop:block Durata :automatic	Scop: bloc Durata:automatic
Static	Scop:fişier Durata:fixă	Scop: bloc Durata: fixă	NEPERMIS
Extern	Scop: program Durata: fixă	Scop: bloc Durata: fixă	NEPERMIS
Nici un specificator prezent	Scop: program Durata: fixă	Scop: bloc Durata: dinamică	Scop: bloc Durata: automatică

10.6. Macrouri

În mod normal când o funcție este apelată, se creează setul de variabile locale, apoi acestea se inițializează. Acest lucru durează, în unele cazuri, mai mult decât execuția corpului funcției. Apare întrebarea cum poate evita pierderea de viteză şi totuşi să rezulte un program structurat.

Limbajul C este prevăzut cu macrouri (macrodefiniții) care permit implementarea rapidă de operații simple fără dezavantajele de timp din cazul funcțiilor.

Dacă se doreşte declararea unei valori constante se poate folosi directiva preprocesor #define, ca mai jos:

#define pi 3.1416

La întâlnirea respectivei directive preprocesorul va înlocui fiecare menționare în cod a lui pi cu însăşi valoarea 3.1416. Similar se va întâmpla şi cu corpul unui macro.

Observație: o variabilă poate fi declarată fi declarată ca fiind statică, adică nu-şi va modifica valoarea pe parcursul execuție cu ajutorul cuvântului cheie static pus înaintea specificatorului de tip.

Fie exemplele de mai jos unde sunt prezentate două macrouri, unul calculează x^2 iar altul realizează maximum dintre două numere.

```
#define my_max(x,y)  (((x)>(y))?(x):(y) )
#define my_sqr(x)  ((x)*(x))
```

Dacă se vor folosi aceste macrouri, în locul apelului unei funcții, se va insera direct codul corespunzător implementării definițiilor din dreapta.

Totuşi folosirea macrourilor ascunde un mare pericol pentru cineva care nu are suficientă experiență. Dacă în cazul unei funcții se realizează întâi evaluarea parametrilor, la apelul unui macro acest lucru nu se mai face. Apar probleme de ordine a execuției operațiilor ca în exemplele de mai jos:

```
rez=my_sqr(x+1)  // evaluarea este rez=x+1*x+1 în loc de
                 //(x+1)*(x+1)
```

sau

```
rez=z/my_sqr(x)  //evaluarea este rez = (z/x)*x în loc de z/(x*x)
```

Din acest motiv la compilatoarele de OOP s-a introdus noțiunea de funcție “in-line” care elimină dezavantajele macrourilor dar se comportă ca acestea.

Trebuie menționat că la depanarea pas cu pas se poate intra în interiorul unui apel de macro ceea ce face ca erorile de tipul prezentat mai sus să fie greu depistabile.

10.7 Macrouri predefinite

10.7.1. ASSERT

Folosirea acestei macrodefiniții este indicată din următoarele motive:

- creşte foarte mult viteza de depanare a unui program
- programarea în C la ora actuală este realizată sub compilatoare de C++ care acceptă şi cod C clasic, compilatoare care o au predefinită.

Este un macro care funcționează exact ca o instrucțiune if dar, spre deosebirea de aceasta, în momentul în care condiția de evaluat ia valoarea zero blochează execuția programului apelând funcția abort. La stoparea execuției se trimite către dispozitivul standard de ieşire un mesaj specific care precizează numele fişierului sursă şi linia din cadrul acestuia, care a generat eroarea.

Dacă valoarea evaluată este diferită de zero assert nu are nici o influență asupra execuției programului. Prototipul este definit în biblioteca ASSERT.H şi arată ca mai jos:

void assert (int test).

De obicei este folosit pentru a testa dacă o încercare de alocare dinamică a unei noi zone de memorie a reuşit sau nu. După cum am spus el este folosit în cadrul depanării unei aplicații. O dată terminată această fază se poate face ca preprocesorul C să ignore liniile din codul sursă ce conțin apeluri de acest tip prin includerea în codul sursă a directivei

#define NDEBUG

înaintea directivei ce include biblioteca ASSERT.H. Un exemplu scurt de folosire este prezentat mai jos:

```
// aici se va include #define NDEBUG
#include <assert.h>
#include <alloc.h>

void main(void)
    {
    char *s;

    s=( char *)malloc(20*sizeof(char));
    assert(s);
    s=NULL;
    assert(s);
    }
```

La execuția acestui program se va obține activarea assert de-abia în ce-a de-a doua situație.

10.7.2. Macrouri specializate în clasificarea caracterelor.

Au fost introduse pentru a uşura gândirea şi scrierea codului în cazul în care se fac prelucrări pe şiruri de caractere. Aceste prelucrări sunt deseori întâlnite. S-a preferat implementarea ca macrouri pentru a obține o viteză maximă de execuție. Vom prezenta mai jos cele mai folosite macrouri puse la dispoziție de biblioteca CTYPE.H. Toate aceste macrouri returnează o valoare diferită de zero în cazul în care sirul de caractere evaluat are respectiva caracteristică testată de macrou sau zero în caz contrar.

int islower(int c);

Testează dacă caracterul c este din categoria literelor mici , c∈[a…z].

int isalpha(int c);

Testează dacă caracterul c este literă, c∈[a…z] sau c∈[A…Z].

int isupper(int c);

Testează dacă caracterul c este literă mare c∈[A…Z].

int isprint(int c);

Testează dacă caracterul c este tipăribil c∈[0x20…0x7e]

int isascii(int c);

Testează dacă caracterul octetul low al lui c exte în intervalul [1…127].

int ispunct(int c);

Testează dacă caracterul c este un caracter de punctuație.

int isspace(int c);

Testează dacă caracterul c este spațiu, tab, CR, salt la linie nouă , tab vertical sau formfeed adică c∈[0x09…0x0d, 0x20]

int isdigit(int c);

Testează dacă caracterul c este cifră, c∈[0…9].

int isgraph(int c);

Similară cu isprint numai că se elimină spațiile.

int isxdigit(int c);

Testează dacă caracterul c aparține bazei 16, c∈[0…9] sau c∈[a…f] sau c∈[A…F].

10.7. 3. Macrodefiniții predefinite

Unul din următoarele simboluri este predefinit, în funcție de modelul de memorie ales pentru compilare:

__COMPACT__ __MEDIUM__ __HUGE__

__LARGE__ __SMALL__ __TINY__

Alte macrodefiniții implicite:

__cplusplus definit dacă se compilează un program C++
__DATE__ data compilării, ca "mmm dd yyyy"
__FILE__ numele fişierului sursă curent (afectat de #line), ca un şir de caractere
__LINE__ numărul liniei curente (afectat de #line), ca un număr întreg
__MSDOS__ întotdeauna definit (1)
__OVERLAY__ #definit dacă este activată opțiunea de lucru cu reacoperiri (opțiunea -Y activă)
__PASCAL__ #definit dacă opțiunea -p este activă
__STDC__ #definit dacă se cere compatibilitate cu standardul ANSI (opțiunea -A, sau din mediul integrat)
__TIME__ ora curentă "hh:mm:ss"
__TURBOC__ versiunea compilatorului, ca o constantă hexazecimală.