

CAPITOLUL 11

11. Pointeri și alocarea dinamică

11.1. Introducere.

Primele minisisteme cu microprocesoare pe 8 biți aveau 64 ko memorie RAM. Această dimensiune era aleasă din cauza unei limitări datorate numărului n de biți necesari pentru construirea adresei $2^{16} = 2^{10} * 2^6 = 2^6$ ko = 64 ko deci 16 biți de adresă. Întrucât procesorul lucra pe 8 biți pentru a avea acces la o dată oarecare din memorie el trebuia să facă una sau două operații (depinzând de construcția internă). Sistemele cu procesoare pe 16 biți (8086 /V24 /8088) aveau un nou set de instrucțiuni, erau mai rapide și accesau mai multă memorie RAM .

Problema era că deja se cheltuise foarte mult timp și bani pentru a dezvolta programe pentru sistemele anterioare. Multe aplicații mai importante trebuiau rescrise cu un preț minim. În toate programele se ținuse cont în lucrul cu datele din memorie de limitarea de 64 ko.

Cel mai mic efort de programare se făcea dacă se proiecta un sistem de operare care să “vadă “ memoria RAM ca fiind alcătuită din bucăți (segmente) de câte 64 ko. Aceasta a fost ideea de bază a sistemului de operare MS-DOS unde adresa de memorie a oricărei date este compusă din două părți :

- adresa de segment: arată a câta locație din cele 2^{16} locații ale unui segment este data utilă;
- adresa de offset: arată în al câtelea segment , din cele n segmente din memorie, ne aflăm, unde:

$$n = \frac{\dim mem.RAM[ko]}{64[ko]} .$$

În programarea de nivel înalt (la acea dată) vezi C, Pascal, Basic, Cobol ș.a.m.d., actual numite limbaje de nivel mediu, s-a introdus noțiunea de pointer. Aceasta este o variabilă care în loc să rețină o dată utilă direct în calcule, reține adresa în memorie a unei date utile.

11.1.1. Noțiunea de pointer

Una din cele mai “puternice” modalități de acces la date se bazează pe posibilitatea de a avea acces direct la adresa de memorie a unei date stocate acolo.

O variabilă care stochează adresa altei variabile se numește pointer.

Avantajul folosirii de către programator a unui asemenea concept este acela că în cazul manipulării unor masive de date de mari, care la rândul lor conțin elemente de mari dimensiuni (vezi cazul unei structuri) nu se va mai transfera toată zona de memorie ocupată de o dată ci numai cu adresa ei de început. Acestea se vor observa și înțelege mai clar din exemple.

Pentru a vedea mai clar folosirea conceptului de pointer să considerăm programul de mai jos. Acest program declară un vector de elemente întregi (deci reprezentate pe 2 octeți fiecare) precum și un pointer la un întreg. Aceasta înseamnă că compilatorul va ști în cazul manipularilor matematice cu adresa care o conține să facă calculele corect, după cum vom vedea mai jos.

Geneza operatorului &op

Se traduce prin “adresa lui” aplicată unui operator op; este o moștenire din vremea DECPDP11 care folosea acest sistem.

Este neplăcut că terminologia s-a păstrat , deoarece ambiguitatea apare în faptul că nu știm concret despre ce adresă este vorba. La ce ne referim: cea ce poate reprezenta o adresare perfect normală pentru procesorul X poate să fie imposibilă pentru controllerul produs de Y pentru o mașină de spălat care folosește adresarea de 17 biți . Deci problema portabilității absolute s-a păstrat pentru că a inventa un nou termen ar fi bulversat mai mult situația.

Un pointer este folositor dacă este o cale de acces la valoarea a cărei adresa o reține. Limbajul C folosește pentru aceasta operatorul *.

```
/******începe zona includeri de biblioteci *****/
#include <stdio.h>
#include <conio.h>
/******sfârșit zona includeri de biblioteci *****/
// nu am funcții proprii
// nu am variabile globale
/****** incepe zona programului principal *****/
void main(void)
{
    int *p; //pointer la o variabilă de tip întreg
    int v[20]; // tablou de 20 elemente de tip întreg
    int n,i; //variabile necesare prelucrării masivului de date
    clrscr();// șterg ecran
    printf("dati dimensiunea vectorului N=");
    scanf("%d",&n);//citesc dimensiunea vectorului
    for(i=0;i<n;i++)
    {
        printf("\nV[%d]=",i);
        scanf("%d",&v[i]);
    }// citesc vectorul
    p=v;// depun adresa de început a masivului v în variabila p
    for(i=0;i<n;i++)
    {
        printf("\nContinutul lui *v+%d=%d",i,*(v+i));
        printf("\nAdresa lui V[%d] este %p",i,v+i);
    }
    for(i=0;i<n;i++)
    {
```

```

        printf("\nContinutul lui *p+%d=%d",i,*(v+i));
        printf("\nAdresa lui p+%d este %p",i,p+i);
    }
    getch();
}
/*****sfârșit zona program principal*****/
// neavând funcții proprii nu am aici nimic

```

Îeșirea acestui program în cazul execuției pentru un vector de trei elemente arată ca mai jos:

```

dati dimensiunea N=3
V[0]=1
V[1]=2
V[2]=3
Continutul lui (v+0)=1
Adresa lui V[0] este 0C5E:0FD0
Continutul lui *v+1=2
Adresa lui V[1] este 0C5E:0FD2
Continutul lui *v+2=3
Adresa lui V[2] este 0C5E:0FD4
Continutul lui *p+0=1
Adresa lui p+0 este 0C5E:0FD0
Continutul lui *p+1=2
Adresa lui p+1 este 0C5E:0FD2
Continutul lui *p+2=3
Adresa lui p+2 este 0C5E:0FD4.

```

11.1.2. Operații aritmetice cu pointeri

Operațiile aritmetice permise sunt:

- adunarea unui pointer cu un întreg
- scăderea unui întreg dintr-un pointer
- scăderea a doi pointeri
- compararea unui pointer cu constantă NULL (valoarea întreaga 0)
- compararea a doi pointeri.

Singura operație aritmetică permisă asupra pointerilor către funcții este compararea cu constanta NULL.

Adunarea unui pointer cu un întreg are ca rezultat un pointer de același tip. În cazul în care pointerul are ca valoare adresa unui element de tablou rezultatul adunării este adresa elementului al cărui indice este egal cu suma dintre indicele elementului către care indică pointerul, plus valoarea întregului.

Din motive legate de modul de generare a adreselor fizice de către procesor, rezultatul nu este valid decât dacă indicele rezultat din calcul este mai mare sau egal cu zero și mai mic sau egal cu numărul de elemente din tablou.

Scăderea unui întreg dintr-un pointer. Se face ca și cum s-ar aduna la pointer valoarea întreagă respectivă cu semn schimbat.

Scăderea a doi pointeri. Cei doi pointeri trebuie să indice către două elemente din același tablou; rezultatul este un întreg egal cu diferența indicilor celor doua elemente.

Compararea a doi pointeri. Cei doi pointeri trebuie să indice către două elemente din același tablou; rezultatul comparației este egal cu rezultatul comparației indicilor celor doua elemente de tablou.

Valorile aparținând unui tip pointer pot fi convertite către orice alt tip pointer prin folosirea unei conversii explicite.

11.1.3 Modele de memorie

Structura procesoarelor 80x86 a determinat maniere diferite de generare a codului de către compilatoare, în funcție de modurile de adresare utilizate pentru accesul la date, sau de modul de realizare a apelurilor de funcții.

Dacă programatorul cunoaște complexitatea aplicației și volumul de date manipulat, el poate impune o anumită modalitate de generare a codului, astfel încât el să fie optim. Așa a apărut noțiunea de "model de memorie" specific majorității compilatoarelor de pe o mașină compatibilă IBM-PC.

Pentru manipularea datelor se folosesc, adesea, pointeri. În funcție de modurile de adresare utilizate, ei pot avea atributele:

- **near pointer** pe 16 biți, reprezentând, de fapt, un offset în interiorul unui segment;
- **far pointer** pe 32 biți, reprezentând o combinație segment:offset (adică o adresă reală completă); un pointer "far" poate avea orice valoare (în limita a 1 MB de memorie), dar în calcule, segmentul nu poate fi modificat;
- **huge pointer** este similar pointerului "far", dar în calcule segmentul poate fi modificat, astfel încât să poată fi adresată toată memoria, fără schimbarea explicită a segmentului. Această caracteristică este dată de operația de normalizare la care sunt supuși pointerii "huge" în orice operații de calcul.

Observație: Încălcarea regulilor de mai sus nu este semnalată întotdeauna prompt de compilator. Rezultatele pot fi incerte. Modelele de memorie, utilizate de Turbo C, sunt: TINY, SMALL, MEDIUM, COMPACT, LARGE și HUGE. Diferențierea lor este dată de modul de utilizare a zonelor de cod, date și stivă, precum și de atributele implicite ale pointerilor utilizați în modelul respectiv. Vom face distincție între zonele de date statice și zonele de date alocate dinamic (heap). Ca reguli generale pentru alocarea datelor se țin cont de următoarele:

- datele statice se alocă în segmentul de date (DS) și sunt limitate la cel mult 64 kB;
- datele alocate dinamic se plasează fie în segmentul de date (la modelele TINY, SMALL, MEDIUM), fie în "heap" (la modelele COMPACT, LARGE, HUGE);
- datele auto se alocă în segmentul de stivă (SS) și sunt limitate la cel mult 64 kB.

11.2. Alocare dinamică.

11.2.1. Introducere

Până acum s-a discutat despre ceea ce se numește alocare statică . Acest termen se referă la faptul că o dată alocată o variabilă în program, memoria corespunzătoare ei numai poate fi refolosită dacă la un moment dat nu ar fi nevoie de ea pe parcursul execuției programului. Deoarece foarte multe ori memoria RAM era extrem de scumpă, iar necesitățile de memorie foarte mari s-a încercat rezolvarea acestui conflict.

Prima metodă era ca atunci când nu ajungea memoria RAM se folosea o unitate externă de stocare cu posibilități de scriere sau citire. Această metodă este și acum folosită. Singurul dezavantaj al acestei metode este faptul că operațiile cu discul sunt mult mai lente decât cu memoria RAM.

De aceea s-a pus problema folosirii cât mai optime a acestei memorii. În acest scop proiectanții unui sistem de operare au prevăzut în cadrul acestei funcții specifice de gestionare a memoriei.

Printre facilitățile de bază ale acestor funcții sunt :

- alocare de memorie;
- dealocare de memorie;
- verificarea spațiului de memorie rămas nealocat .

Alocarea memoriei. Prin termenul de alocare se înțelege rezervarea unei zone de memorie de o anumită dimensiune, această zonă de memorie “dispare” practic din zona de memorie în care sistemul citește sau scrie fără restricții. La alocare se returnează adresa de început a acestei zone.

Dezalocarea memoriei. Prin termenul de dealocare se înțelege fenomenul invers alocării. Se trimite către sistem adresa de început a unui bloc alocat (rezervat) și i se cere eliberarea lui adică reintegrarea în spațiul de memorie ce poate fi accesat de orice alt program.

Verificarea spațiului de memorie nealocat. După un număr oarecare de alocări, fără a mai face și dealocări spațiul rămas liber tinde către zero. Funcțiile de alocare în momentul în care li se cere să aloce o anumită zonă (n -octeți) și nu mai au suficient spațiu liber returnează NULL pentru a semnaliza această imposibilitate. Totuși aceasta reprezintă o situație critică și nu se mai poate face mare lucru. De aceea, ca tehnică de programare, se poate face din când în când în punctele cheie verificarea spațiului de memorie rămas liber pentru a calcula automat dacă programul poate continua sau nu.

Eroare de gândire a începătorului relativ la operatorul sizeof

Acest operator știe să întoarcă numărul de octeți pe care este stocat un tip de dată DEFINIT fie implicit (tipurile de dată implicate : char , int , long, double) fie cele definite de utilizator (structurile).

Dar, NU POATE calcula dimensiunea unei zone despre care nu are informații clare în momentul compilării.

Exemplu clasic :

```
char S[10];
```

```
.....
```

```
gets(S);
```

```
n=sizeof(S) // eroare
```

```
// S poate avea fie 255 caractere (caz în care se păstrează sigur numai
```

```
// primele 10, fie nici un caracter
```

```
// deci nu poate fi aplicat.
```

De aceea s-a introdus funcția strlen(S) în acest caz.

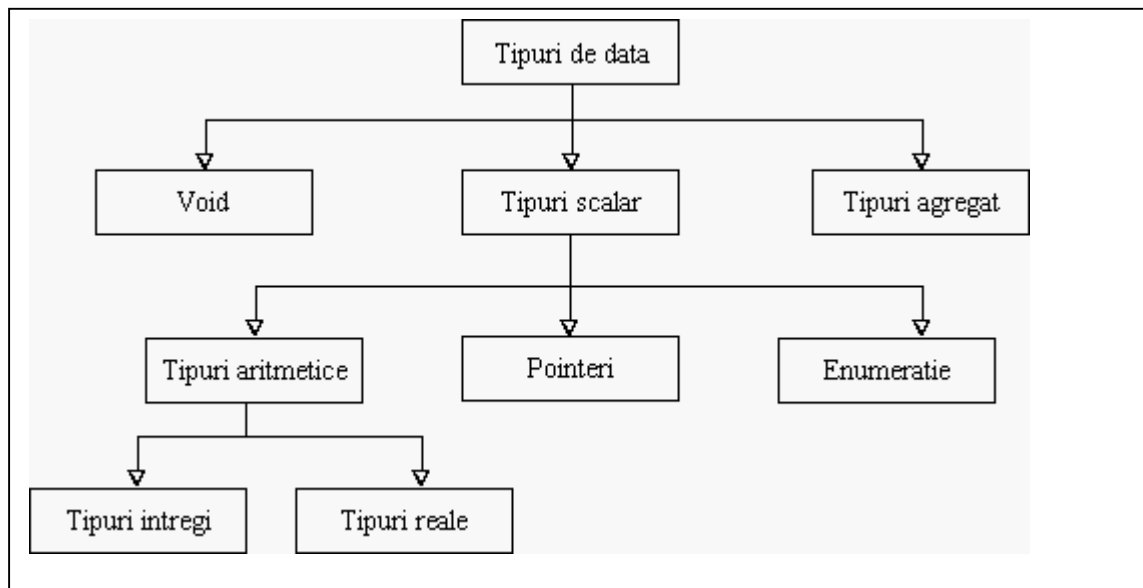
O versiune de strlen(S) ar fi

```
nr_cars=0;
```

```
while ( *S[i++]!='\0') nr_cars++;
```

```
return nr_cars;
```

Mai jos prezentăm o altă clasificare a tipurilor de date\$



11.2.2. Alocarea dinamică

Orice tip de dată predefinit sau definit de utilizator poate fi alocat static sau dinamic. Funcția de bază în limbajul C, folosită pentru alocarea dinamică are următorul prototip:

```
void *malloc(size_t size);  
void far *farmalloc(unsigned long nbytes);
```

Alocarea dinamică a unui bloc de memorie de lungime dată (size, sau nbytes). "malloc" permite alocarea din segmentul de date a cel mult 64 kB, iar "farmalloc" permite alocarea din "heap" a cel mult 1 MB (de fapt în limitele memoriei rămase disponibile). Funcțiile returnează pointer NULL, dacă nu mai există suficientă memorie pentru alocare.

```
void *calloc(size_t nelem, size_t elsize);  
void far *farcalloc(unsigned long nunits, unsigned long unitsz);
```

Alocarea dinamică a unui bloc de memorie pentru un număr dat de elemente (nelem, nunits) de dimensiune cunoscută (elsize, unitsz). Dimensiunea totală alocată este produsul celor doi parametri. "malloc" permite alocarea din segmentul de date a cel mult 64 kB, iar "farmalloc" permite alocarea din "heap" a cel mult 1 MB (de fapt în limitele memoriei rămase disponibile). Funcțiile returnează pointer NULL, dacă nu mai există suficientă memorie pentru alocare.

```
void *realloc(void *block, size_t size);  
void far *farrealloc(void far *oldblock, unsigned long nbytes);
```

Modificarea dimensiunii unui bloc de memorie deja alocat. Pointerul block (oldblock) trebuie să fi fost obținut cu o funcție de alocare, apelată anterior: "malloc" ("farmalloc"), "calloc" ("farcalloc"). Dacă redimensionarea nu se poate face, funcțiile returnează NULL. Este posibil ca noul pointer rezultat să fie diferit de pointerul inițial.

```
void free(void *block);  
void farfree(void far *block);
```

Eliberează un bloc de memorie deja alocat. Pointerul block trebuie să fi fost obținut cu o funcție de alocare, apelată anterior: "malloc" ("farmalloc"), "calloc" ("farcalloc"). Încercarea de a apela una din funcțiile de mai sus cu un pointer obținut pe alte căi, poate duce la rezultate imprevizibile.

În programele ce utilizează alocare dinamică, se recomandă ca orice zonă, astfel obținută, să fie eliberată după utilizare.

Obs. Compilatorul nu sesizează erori de tipul celor menționate mai sus; depanarea este, de asemenea, dificilă.

```
unsigned coreleft(void);  
unsigned long farcoreleft(void);
```

Evaluează disponibilul de memorie posibil de a fi folosit în alocarea dinamică. Funcția "coreleft" returnează memoria liberă din segmentul de date, iar "farcoreleft" memoria liberă din "heap".

Funcțiile de alocare primesc numărul exact de octeți ce se doresc a fi alocați și returnează adresa de început a respectivei zone. Din motive de portabilitate a programului nu este indicat să se calculeze dimensiunea zonei de alocat cu formula:

$$\text{Nr_elemente} * \text{nr_octeți_ocupat_de_respectivul_tip_dată}$$

Și acest lucru deoarece funcție de arhitectura sistemului de calcul numărul de octeți pe care este reprezentat același tip de date poate diferi. Deci programul ar trebui modificat în cazul recompilării sub altă platformă.

Portabilitatea unui program

Se referă la modul de scriere a unui cod sursă astfel încât la recompilarea sau la execuția lui sub diferite platforme hardware sau software, modificările ce trebuie făcute pentru a-l face funcțional să fie minime sau inexistente.

Pentru a asigura portabilitatea din punct de vedere al alocării dinamice se va folosi un operator special al limbajului C și anume `sizeof(tip_dată)`. Numele se traduce prin “mărimea lui”. El returnează numărul de octeți pe care este reprezentat respectivul tip de dată. În aceste condiții rezultă clar că zona de memorie ce se dorește a fi alocată trebuie calculată cu următoarea formulă:

$$\text{Nr_octeți_de_alocat} = \text{nr_elemente} * \text{sizeof}(\text{tip_dată})$$

Acest operator știe să facă calculul numărului de octeți ocupați și în cazul unui tip de dată definit de utilizator.

11.2.3. Alocarea dinamică a tablourilor de date

După cum știm în cazul tipurilor de date alocate static se pot defini tablouri care să aibă până la trei dimensiuni. Există în respectivul caz o limitare a zonei de memorie care poate fi astfel alocată și anume între 64 ko și 1 Mo funcție de modelul de memorie ales pentru compilare. Dacă în cazul declarării statice compilatorul se ocupă de tot procesul în cazul alocării dinamice acest proces devine ceva mai complicat. Există două metode de alocare:

Metoda I

Alocarea se face o singură dată pentru toate elementele și este o singură instrucțiune. Metoda are, pentru tablourile a căror dimensiuni sunt ≥ 2 , dezavantajul că accesul la un element se poate face numai prin intermediul unui pointer și trebuie să folosesc o formulă de calcul a adresei elementului. Această formulă trebuie să țină cont de adresa de început a masivului, de indicii elementului și de numărul de octeți pe care este reprezentat. Așa se lucra la nivelul limbajului de asamblare unde nu aveam alte posibilități.

Formula de echivalență **$\text{ar}[\text{i}] \leftrightarrow *(\text{av} + \text{i})$** .

Vom prezenta un scurt exemplu de folosire a primei metode în cazul unui tablou monodimensional de caractere.

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

void main(void)
{
    char strg[40], *there, one, two;
    int *pt, list[100], index;

    strcpy(strg, "Acesta e un sir de caractere.");

    one = strg[0];          /* one și two sunt identici ca valoare */
    two = *strg;
    printf("one, two= %c %c\n", one, two);

    one = strg[8];          /* one și two sunt identici */
    two = *(strg+8);
    printf(" one, two=%c %c\n", one, two);

    there = strg+10;        /* strg+10 este identic cu strg[10] */
    printf("strg[10]= %c\n", strg[10]);
    printf("there= %c\n", *there);

    for (index = 0; index < 100; index++)
        list[index] = index + 100;
    pt = list + 27;
    printf("list[27]= %d\n", list[27]);
    printf("*pt= %d\n", *pt);
    getch();
}
```

Metoda II

Pornește de la observația că, compilatorul face automat calculul adresei elementului în șirul de elemente în cazul declarării statice. Se poate folosi și în cazul alocării dinamice această facilitate a compilatorului cu condiția ca modul în care se realizează alocarea să respecte modalitatea internă folosită de compilator în cazul alocării statice. Acest lucru permite ca, deși se va alocă dinamic un tablou, ulterior în cod accesul la oricare element să poată fi realizat folosind metoda indiceală tipică cazului alocării statice.

11.2.3.1. Alocarea tablourilor monodimensionale

În acest caz alocarea se face conform formulei de mai jos:

Nume_tablou=(tip_elem_tablou)malloc(nr_elem_tab *sizeof(typ_el_tablou))

Un exemplu de aplicare în program a acestei formule

```
void main(void)
{
    int v*;
    ...
    v=(int *)malloc(10*sizeof(int));
    ...
}
```

Se observă că valorii returnate de “malloc” îi aplicăm un operator de cast. În mod normal conversia de tip se face automat, totuși în cazul alocării este indicat să fie făcută explicit.

Vectorul va putea fi folosit conform celor două metode ca mai jos:

1. $*(v+4*\text{sizeof}(\text{int}))=0x32;$
2. $v[4]=0x32;$

Un caz special este cazul alocării dinamice al unui tablou monodimensional de caractere. Orice șir de caractere este prevăzut cu un terminator special ‘\0’. Aceasta provine din limbaj de asamblare, având în vedere faptul că funcțiile DOS de lucru cu șirurile de caractere sunt gândite să primească adresa de început din zona de memorie și să prelucreze element cu element șirul până când este întâlnit terminatorul ‘\$’. Dacă inițializarea se va face element cu element în program ‘\0’ trebuie depus explicit. Folosirea funcțiilor de bibliotecă dă avantajul că terminatorul este pus automat de respectivele funcții.

11.2.3.2. Alocarea tablourilor bidimensionale

Formula de echivalență $\mathbf{ar[m][n]} \leftrightarrow *(\mathbf{ar[m]+n}) \leftrightarrow *((\mathbf{ar+m})+n)$

Dacă ar este de întregi int $\mathbf{ar[m][n]}$ atunci $\mathbf{ar[i][j]}$ este echivalent cu

$*(\mathbf{int*})((\mathbf{char*})\mathbf{ar} + (\mathbf{i*a*sizeof(int)} + \mathbf{j*sizeof(int)}))$

Cast-urile trebuiesc puse pentru că se observă că calculele de deplasament, ținând cont de de spațiul ocupat de un element din tablou, au fost făcute explicit.

În acest caz alocarea este mai complicată, însă ulterior există avantajul de a se putea lucra și prin specificarea poziției elementului.

```

int far **t_alloc(int lin, int col)
{
    int i;
    register int far **plin, *pdata;

    pdata=(int far *)farcalloc(lin*col,sizeof(int far*));
    if(pdata==(int far*)NULL)err(20,30,"Nu exista spatiu");

    plin=(int far**)farcalloc(lin,sizeof(int far*));
    if(plin==(int far**)NULL)err(20,30,"Nu exista spatiu");

    for(i=0;i<lin;i++)
    {
        plin[i]=pdata;
        pdata+=col;
    }
    return plin;
}
/*****/
void t_free(int far **pa)
{
    farfree(*pa);
    farfree(pa);
}

```

Exemplul de mai sus ne permite să pot accesa în program un element din tablou în maniera `m[i][j]` ceea face codul mai clar.

11.2.3.3. Tablouri tridimensionale

Se tratează după aceeași gândire ca și cele bidimensionale, mai jos vom prezenta un exemplu de tranfer de parametru în cazul tablou tridimensional.

fie

```

void main(void)
{
    int m [5][6][7];
    ...
    f2(m, x, y, z);
}
void f2 ( int *** argm, dim1, dim2, dim3)
{int x;
    ...

```

atunci $x=2 + ((\text{int } x) \text{ argm} + i*\text{dim3}*\text{dim2} + j*\text{dim2} + k)$ (echivalent cu `argm[i][j]`)

11.3. Pointeri la structuri

Având în vedere definiția pointerului nu există nimic care să împiedice declararea unor pointeri la o structură. În aceste condiții tipul de date grupă și student prezentate anterior se vor rescrie ca mai jos.

```
typedef struct Student  
{  
    char *name;  
    unsigned short int age;  
    unsigned char n_lab;  
    unsigned char n_ex;  
} stud;
```

```
typedef struct Grupa  
{  
    int nume;  
    int an;  
    stud *t;  
}gr;  
gr *p1,*p2;
```

11.3.1. Accesarea membrilor

De această dată se va folosi \rightarrow în loc de punct, restul regulilor de accesare rămânând identice. Acest nou operator de acces a fost introdus doar pentru a scrie cod cât mai clar deoarece dacă avem *stud *p*; atunci *p* \rightarrow *nume* este perfect echivalent cu *(*p).nume* din punct de vedere sintactic.

11.3.2. Alocarea structurilor de date

Trebuie avut grijă ca tot spațiu de care este nevoie să fie alocat. Există două nivele de alocare: primul se referă la alocarea de spațiu pentru noul tip de dată, al doilea nivel se referă la alocarea de spațiu, unde este cazul, pentru membrii noului tip de dată.

Alocarea la primul nivel:

```
Gr *p1;  
p1=(gr *)malloc(25*sizeof(gr));
```

Nivelul doi de alocare este clar de-abia în faza de inițializare a unei structuri.

11.3.3. Inițializare și distrugere structuri

Dacă alocarea dinamică pentru tipurile de date predefinite e relativ simplă în majoritatea cazurilor în cazul structurilor de date trebuie ca programatorul să-și construiască propriile funcții de inițializare / eliberare a datelor.

Se vor prezenta funcțiile de inițializare pentru un obiect de tip grupa. Se observă că un obiect de tip grupa conține un tablou de obiecte de tip student și deci trebuie întâi scrisă funcția care inițializează pe acesta.

```
Stud *init_stud(char *name,unsigned short int age,
                unsigned char n_lab,n_ex)
{
    stud *p;
    if(!(p=(stud *)malloc(sizeof(stud))))error("Not enough memory");
    if(!(p->name=(char *)malloc(strlen(name)*sizeof(char))))
        error("Not enough memory");
    strcpy(p->name,name);
    p->age=age;
    p->n_lab=n_lab;
    p->n_ex=n_ex;
    return *p;
}

void free_stud(stud *p)
{
    free(p->name);
    free(p);
}

gr *init_gr(int nume, stud *st, char name, int age, n1,n2)
{
    gr *p;
    if(!(p=(gr *)malloc(sizeof(gr)))) error("Not enough memory");
    p->nume=nume;
    p->an=age;
    p->stud=st;
    return p;
}

void err(char *s)
{
    clrscr();
    puts(s);
    exit(0);
}
```

După cum se observă cele două funcții alocă câte un element. Alocarea pentru tabloul de studenți al cărui pointer este primit în `init_gr` nu a fost prezentată, pentru că se folosește funcția de alocare a unui student pentru fiecare din elemente și ele pot fi depuse într-un tablou alocat static sau dinamic sau într-o listă înlănțuită.

12. Declarații complexe în C

Peste un anumit nivel de experiență, există tendința folosirii unor declarații compacte, dar complexe, și aceasta compilatorul le acceptă. acest lucru conduce la scăderea clarității codului. de aceea recomandăm evitarea pe cât posibil. De exemplu următoarea declarație declară pe x ca fiind un pointer la o funcție care întoarce un pointer către al cincilea element dintr-un tablou de pointeri la întregi. Complicat , NU?

```
int *(*(*x))()[5];
```

Mai fose prezentăm o metodă simplă pentru evitarea acestui tip de complicații:

```
typedef int *AP[5] //al 5-lea element dintr-un tablou de pointeri la întreg
typedef AP *FP() // funcția ce va returna un pointer la al 5-lea element al unui
                  tablou de pointeri către întregi
FP *x             // pointer la.....
```

Motivul principal pentru care declarațiile de tipul celor prezentate mai sus sunt așa complicate în C, este datorat faptului că operatorul pointer { * } este de tip prefix, iar operatorii de funcție și tablou sunt de tip postfix.

Pentru a descifra declarații complexe, trebuie privit dinăuntru înspre afară, adăugând * în stânga și () sau [] în dreapta.

Descifrarea declarațiilor complexe în C

Cea mai bună strategie în descifrarea declarațiilor complexe, este să pornim de la numele variabilei și apoi să adăugăm(să ținem cont de efectul) fiecare parte(operator) ai expresie pornind cu operatorii cei mai apropiați de numele variabilei.

În lipsa parantezelor: se va ține cont întâi de operatorii de tip funcție sau tablou (din dreapta variabilei, deoarece au prioritate mai mare, și de-abia la sfârșit se va ține cont de efectul operatorilor de tip prioritar)

De exemplu:

char **[]; va fi descifrată în următorii pași:

1. x[] este un tablou
2. **[] este un tablou de pointeri
3. char *x[] este un tablou de pointeri către caractere.

Parantezele pot fi folosite fie pentru claritate, fie pentru schimbare a precedenței. De exemplu:

int (**[]) (); va fi descompus după cum urmează:

1. x[] este un tablou
2. (**[]) este un tablou de pointeri
3. (**[]) () este un tablou de pointeri către funcții
4. int ((*x []) () este un tablou de pointeri către funcții ce returnează un întreg

Compunerea declarații complexe

Pentru a compune declarații complexe procesul este identic ca la descompunere. De exemplu dacă se dorește declararea unui pointer către un tablou de pointeri către funcții care întorc pointeri la tablouri de structuri cu numele S, se vor folosi următorii pași:

1. (*x) este un pointer
2. (*x)[] este un pointer la un tablou

3. ((*x[])) este un pointer la un tablou de pointeri
4. ((*x[])) () este un pointer la un tablou de pointeri la funcții
5. ((* (*x[])) ()) este un pointer la un tablou de pointeri la funcții ce întorc pointeri
6. ((* (*x[])) ()) [] este un pointer la un tablou de pointeri la funcții ce întorc pointeri către tablouri
7. struct S ((* (*x[])) ()) [] este un pointer la un tablou de pointeri la funcții ce întorc pointeri către tablouri de structuri cu numele S

În scopul ușurării înțelegerii, se va prezenta un tablou cu diverse expresii complexe uzuale în codurile C, și anumite greșeli comune.

- int i; un întreg
- int* p; pointer la întreg
- int a[]; tablou de întregi
- int f(); funcție ce întoarce un întreg
- int **p; pointer la pointer către întreg
- int (*pa) []; pointer la tabloul de întregi
- int (*pf) (); pointer la o funcție ce întoarce un întreg
- int *ap []; un tablou de pointeri către întregi
- int aa [] [] un tablou de tablou de întregi
- int af [] (); un tablou de funcții ce întorc întreg NEPERMIS
- int *fp(); o funcție ce întoarce un pointer la întreg
- int fa () []; o funcție ce întoarce un tablou de întregi NEPERMIS
- int f () (); o funcție ce întoarce o funcție ce întoarce un înt NEPERMIS
- int ***p; pointer la pointer la pointer la int
- int (**ppa) []; pointer la pointer către un tablou de int
- int (**ppf) []; pointer la pointer către o funcție ce întoarce int
- int * (*pap) []; pointer către un tablou de pointeri la întregi
- int (*paa) [] []; pointer la un tablou de pointeri ce rețin întregi
- int (*paf) [] (); pointer la un tablou de funcții ce întorc întregi NEPERMIS
- int (* *pfp) (); un pointer la o funcție ce întoarce întreg
- int (*pfa) () []; pointer la o funcție ce întoarce un tablou de întregi
NEPERMIS
- int faf() [] (); funcție ce întoarce un tablou de funcții ce întorc întregi
NEPERMIS
- int *ffp() (); funcție ce întoarce o funcție ce întoarce un pointer la int
NEPERMIS
- int (* pff) () (); un pointer la o funcție ce întoarce o funcție ce întoarce un întreg
NEPERMIS
- int **app[]; un tablou de pointeri la pointeri către întregi
- int (*apa[]) []; un tablou de pointeri către un tablou de întregi
- int (*apf[]) (); un tablou de pointeri către funcții ce întorc întregi
- int *aap[] []; un tablou de tablouri de pointeri la întregi

- `int aaa[] [] []`; un tablou de tablouri de tablouri de întregi
- `int aaf[] [] ()`; un tablou de tablouri de funcții ce întorc întregi NEPERMIS
- `int *afp[] ()`; un tablou de funcții ce întorc pointeri la întregi
- `int afa[] () []`; un tablou de funcții ce întorc tablouri de întregi NEPERMIS
- `int aff[] () ()`; un tablou de funcții ce întorc funcții care întorc întregi
NEPERMIS
- `int **fpp ()`; o funcție ce întoarce un pointer la pointer de întregi
- `int (*fpa ()) []`; o funcție ce întoarce un pointer către tablouri de întregi
- `int (* fpf ()) ()`; funcție ce întoarce un pointer către o funcție ce întoarce un întreg
- `int faa () [] []`; funcție ce întoarce un tablou de tablouri de întregi
NEPERMIS
- `int *fap () []`; funcție ce întoarce un tablou de pointeri către întregi
NEPERMIS

Câteva tehnici de creștere a vitezei de execuție a unui cod

Propagarea copiei (se pune la asignare)

Fie următoarea secvență de cod:

`x=y`

`z=1.+x`

corect este:

`x=y`

`z=1.+y` (elimină interdependențele dintre cele două relații)

Reducerea puterii

Ex: `y = pow (x,2)` -> ia foarte mult timp

Înlocuiți cu `y=x*x` sau `j=k*2` -> ia mult timp în comparație cu adunarea `j=k+k` sau cu deplasarea la stânga `j=k<<2`

Redenumirea variabilelor

Câteodată (când se vrea viteză maximă) este mai bine să evităm refolosirea variabilelor

`x=y*z`

`q=r+x+z`

`x=a+b`

A	b
<code>x=y*z</code>	<code>x0=y*z</code>
<code>q=r+x+z</code>	<code>q=r+x0+x0</code>
<code>x=a+b</code>	<code>x=a+b</code>

Compilerul realizează automat aceste lucruri, dar pentru calcule de finețe este bine să facem personal optimizarea codului. Deoarece a) mănâncă timp de calcul iar b) mănâncă memorie, în opțiunile compilerului există:

1. optimizare pentru viteză (stil b)
2. optimizare pentru dimensiune (stil a)

Eliminarea subexpresiilor comune

Ex.	$D=C*(A+B)$		$t=A+B$
	$E=(A+B)/2$	----->	$D=C*t$
	$F=sqr(A+B)$		$E=t/2$
			$F=sqr(t)$

Atenție la ordinea în care se scriu subexpresiile deoarece pt. compilator $a+b+c$ poate fi diferit de $B+C+a$ chiar când se face optimizare automată.

Eliminarea codurilor constante din bucle

```
for (i=0;i<u;i++)
{ A[i]=B[i]+C*D;
  E=G[k];
}
```

Se observă că $C*D$ este calculată de N ori și nu este o funcție de (i) . Absurd!
La fel și $E = G[k]$ este un transfer de memorie realizat de n ori și nu este o funcție de i ,
deci corect bucla este :

```
temp=c*d;
for(i=0;i<n;i++) A[i]=B[i]+t;
E=G[k];
```

Simplificarea prin inducție

Ex. for (i=0;i<n;i++)
 $k=i*4+M$

După o scurtă analiză matematică (temă acasă) rezultă o secvență echivalentă dar se scutesc n înmulțiri.

```
k=m
for(i=0;i<n;i++) k=k+4;
```

Se folosește când referim direct tablouri

```
adresa = adresa_de_baza(A)+(i-1)*sizeof(data)
```

```
adresa = adresa_de_baza(A)-sizeof(data)
adresa = adresa+sizeof(data)
```

dacă aplic și eliminarea constantelor, codul devine :

```
temp = sizeof(data)
adresa = adresa_de_baza(a)-temp
```

adresa = adresa+temp

Lucrul cu fişierele de date (I/O FILE) Streamuri (streams)

C nu face distincţie între diverse dispozitive cum ar fi terminal, ZIP, HDD.

În toate cazurile I/O este realizat prin aceste stream-uri care sunt asociate cu fişiere sau cu dispozitive.

Def: Un **stream** constă dintr-o înşiruire ordonată de octeţi. Poate fi gândit cumva ca un vector, numai că în acest caz nu am acces paralel ci serial la fiecare element.

Pentru a realiza accesarea unui stream trebuie declarat un pointer la o structură standard numită FILE definită în stdio.h.

Ea conţine:

- Informaţii despre numele fişierului
- Modul de acces
- Un pointer către următorul caracter din stream

Aceste informaţii sunt necesare sistemului de operare, din punctul de vedere al programatorului accesul se face prin intermediul unui pointer la această structură numit pointer la fişier (file pointer).

Un program poate avea unul sau mai multe streamuri deschise simultan.

Unul din câmpurile din FILE este indicatorul de poziţie în fişier care se referă la următorul octet ce va fi scris / citit în / din fişier.

După fiecare operaţie R/W S.O modifică automat acest pointer. Accesarea se poate face direct dar acest caz particular, de care trebuie ținut seama, va face ca un program să nu mai fie portabil.

Atenţie: A nu se confunda pointerul la fişier cu indicatorul de poziţie !!! Unul este un pointer altul este un deplasament.

Streamuri standard

Există 3 streamuri care sunt deschise automat pentru fiecare program. Ele se numesc stdin, stdout, stderr. În mod normal ele se referă la terminal dar multe sisteme de operare permit redirectarea lor.

Ex.: un mesaj de eroare poate fi scris într-un fişier în loc de terminal; printf() şi scanf() folosesc streamurile stdin şi stdout. Aceste funcţii pot fi folosite pentru lucrul cu fişierele prin redirectarea stdin şi stdout către respectivele fişiere folosind _____. Dar mai comod este folosirea funcţiilor fscanf() şi fprintf().

Format text şi format binar

Datele pot fi accesate în unul din cele două formate standard ANSI –text sau binar. Un stream de tip text constă dintr-o serie de linii, unde fiecare linie este terminată cu un caracter de salt la linie nouă ‘\n’.

Orice S.O poate avea altă politică de stocare a liniilor pe disc deci terminatorul ‘\n’ nu este obligatoriu. Dar din punct de vedere al programatorului este ca mai sus, restul fiind ascuns de compilatorul C. Totuşi lucrul în mod text nu garantează o portabilitate totală a

unui program. În formatul binar compilatorul nu realizează nici o interpretare a biților ; îi citește sau scrie exact cum apar.

Bufferarea (folosirea unei zone de memorie ca tampon)

Deoarece comparativ cu memoria, dispozitivele de stocare în masă sunt extrem de încete se încearcă reducerea numărului de operații de I/O cu discul. Bufferarea este cea mai simplă metodă pentru a realiza aceasta.

Un buffer este o zonă unde sunt stocate temporar datele înainte de a fi trimise către destinația lor finală. Toate S.O folosesc buffere pentru a citi/scrie din dispozitivele I/O numai că, în acest caz, bufferul are dimensiune fixă. De obicei, este de 512 sau 1024 octeți. Acest lucru înseamnă că, chiar dacă se dorește citirea unui singur caracter dintr-un fișier, S.O citește tot blocul care conține respectivul caracter.

Biblioteca C conține un strat adițional de bufferare care apare în 2 forme:

- buffere de linie
- buffere de bloc

La bufferarea în linie, sistemul stochează caracterele până la primul ‘\n’ sau până se umple bufferul și apoi trimite întreaga linie la S.O pentru procesare. Acest lucru se întâmplă și când citesc date de la terminal.

La bufferarea în bloc, sistemul stochează caracterele până când acel bloc este umplut.

Implicit streamurile I/O relativ la fișiere sunt bufferate în bloc, iar cele care se referă la terminal(stdin, stdout) sunt bufferate în linie.

De asemenea biblioteca C include și un manager de buffer care ține acel buffer cât mai mult posibil în memorie, deci dacă se accesează aceeași porțiune dintr-un stream de mai multe ori, există mari șanse ca să se evite accesul la dispozitivul de stocare.

Atenție! Acest lucru poate crea probleme de consistență, dacă fișierul este folosit în comun de mai multe procese. Pentru sincronizarea între procese este nevoie de:

- scrierea în ASM a propriilor funcții
- folosirea funcțiilor specifice S.O pe care se lucrează

În ambele moduri de bufferare, bufferul poate fi golit la cerere cu ajutorul lui fflush() trimițând, astfel , datele din el la destinația lor.

Deși acest mod de bufferare este mai eficient decât procesarea individuală a fiecărui caracter, există cazuri în care ne deranjează , și anume când se dorește explicit procesarea instantanee a unui caracter fie la intrare fie la ieșire. Cel mai simplu exemplu este atunci când se dorește procesarea caracterelor introduse de la KBD fără a mai aștepta ‘\n’.

C permite modificarea memoriei bufferului chiar la acest nivel,tocmai pentru a putea rezolva și problemele de acest tip.

Citirea și scrierea datelor

O dată ce s-a deschis un fișier se va folosi pointerul la fișier pentru a putea realiza operații de scriere/citire. Există 3 nivele de granularitate (finețe):


1. la nivel de caracter
 - câte un caracter odată
2. la nivel de linie
 - câte o linie odată

3. la nivel de bloc
 - câte un bloc odată

O regulă care se aplică la toate nivelele de I/O este aceea că nu pot citi dintr-un stream și apoi scriu în el fără a apela `fseek()`, `rewind()` sau `fflush()`. La fel și în cazul invers. Acest lucru este necesar pentru că trebuie golit bufferul înaintea oricărei operații pentru a evita problemele de inconsistență.


1. Lucrul la nivel de caracter

`getc()` - macro , citește un caracter
`putc()` - macro , scrie un caracter



de la stream-ul implicit

`fgetc()`
`fputc()`



identice cu anterioarele lor dar sunt funcții

Deși primele sunt macrouri deci mult mai rapide pot exista probleme de ex:

```
putc('x', fp[j++]);
```

s-ar putea să nu meargă corect.

De aceea este indicat că atunci când se lucrează ca în exemplu să folosim `fputc()` sau `fgetc()`;

2. La nivel de linie

Funcții: `fgets()` și `fputs()` și se specifică parametri

- s -> pointer la tabloul de caractere
- n -> nr. Maxim de caractere ce vor fi procesate
- stream -> streamul cu care se lucrează

3. La nivel de bloc

Un bloc poate fi gândit ca un tablou . Când citim/scriem în bloc trebuie să specificăm atât numărul de elemente al blocului cât și dimensiunea fiecărui element.

Funcții: `fread()` și `fwrite()` cu următorii parametri:

- ptr -> pointer către tabloul ce stochează caracterele
- size -> mărime(nr. de octeți al tipului de dată folosit, recomand `sizeof()` pentru portabilitatea programului
- număr elemente ce trebuie procesat
- pointer la stream

I/O Nebufferat

Anterior am menționat faptul că există situații în care bufferarea datelor nu deranjează. Exemplu clasic este editorul de text, când se scrie un text și se apasă tastele de navigare prin el.

Pentru a elimina buferarea pentru `stdin` se poate folosi fie `setbuf()`, fie `setvbuf()`.
`Setbuf(stdin, NULL)` sau `stat= setvbuf (stdin, NULL, IONBUF, 0).`

Accesul aleator într-un fișier

Este asigurat de funcția `fseek` care poziționează pointerul la fișier la un deplasament dorit față de începutul acestuia.

`Ftell` returnează valoarea pointerului de fișier.