Curs 6. POINTERI

Un pointer este o variabilă care conține o adresă de memorie.
Pointerii sunt foarte mult utilizați în C pe de o parte pentru că uneori sunt singura cale de rezolvare a unei anumite probleme, iar pe de altă parte pentru că folosirea lor duce la alcătuirea unui cod mai compact și mai eficient.
Ca metodă, pointerii se utilizează pentru un plus de simplitate.

6.1. Pointeri și adrese

Un pointer fiind o variabilă ce conține o adresă de memorie, in particular, ea poate referi adresa unei variabile din program. Deci este posibilă adresarea acestei variabile "indirect" prin intermediul pointerului.
Variabilele de tip pointer se declară prin construcţii de forma:

```
tip *nume;
```

Fie x o variabilă, de tip`int` și px este un pointer la acea variabilă.
Operatorul & dă adresa unei variabile, astfel încât instrucţiunea :

```
px=&x
```

dă variabilei px adresa lui x, px înseamnă "pointează pe x". Operatorul & poate fi aplicat numai variabilelor şi elementelor unui tablou.

Invers, dacă avem un pointer px, prin *px se face referire la valoarea care se găseste memorată la adresa pointată de px.

Exemple.

```
int *p, n=5, m;        //declarăm p ca pointer la întreg şi n şi m întregi
p=&n;                  //p va conţine adresa variabilei n deci p va pointa (indica) spre n
m=*p;                  //lui m i se dă valoarea care se găseşte la adresa indicată de p
                       //deci m va conţine valoarea lui n; în consecinţă m=5
m=*p+1;                //m va conţine valoarea 6
```

Construcții ca &(x+1) și &3 sunt interzise. Este de asemenea interzisă păstrarea adresei unei variabile registru.

Operatorul unar * tratează operandul său ca o adresă, accesează această adresă și îi obține conținutul.

Astfel, dacă y este tot un `int`

```
y = *px
```

asignează lui y, ori de câte ori este cazul, conținutul locației unde pointează px.

Astfel secvența

```
px = &x;
y = *px;
```

asignează lui y aceiaşi valoare ca şi

```
y = x
```

Totodată este necesară declararea variabilelor care apar în secvența:

```
int x, y;
int *px;
```

Declararea lui x și y este deja cunoscută. `*px` este un int, adică în momentul în care px apare în context sub forma `*px`, este echivalentă cu a întâlni o variabilă de tip int. De fapt, sintaxa declararii unei variabile imită sintaxa expresiilor în care ar putea să apară respectiva variabilă. Acest raționament este util în toate cazurile care implică declarații complicate.
Exemple:

```
double atof(), *dp; //atof() și *dp au valoare de tip double.
```

De notat declarația implicită, ceea ce vrea să însemne că un pointer este constrâns să pointeze o anumită categorie de obiecte (funcție de tipul obiectului pointat).

Pointerii pot apare în expresii.

De exemplu, dacă px pointează pe întregul x atunci *px poate apare în orice context în care ar putea apare x.

```
y = *px + 1
```

dă lui y o valoare egală cu x plus 1.

```
printf("%d\n", *px)
```

imprimă o valoare curentă a lui x și `d = sqrt((double) *px)` face ca d = radical din x, care este forțat de tipul double înainte de a fi transmis lui `sqrt`.

În expresii ca

```
y = *px + 1
```

operatorii unari * și & au prioritate mai mare decât cei aritmetici, astfel această expresie, ori de câte ori pointerul px avansează, adună 1 și asignează valoarea lui y.

Referiri prin pointer pot apare și în partea stângă a asignărilor. Daca px pointează pe x atunci

```
*px = 0
```

îl pune pe x pe zero și

```
*px += 1
```

îl incrementează pe x, ca și

```
(*px)++
```

In acest ultim exemplu parantezele sunt necesare; fără ele, expresia va incrementa pe px în loc să incrementeze ceea ce pointează px deoarece operatorii unari * și + sunt evaluați de la dreapta la stânga.

Dacă pointerii sunt variabile, ei pot fi manipulați ca orice altă variabilă. Daca py este un alt pointer pe int, atunci

```
py = px
```

copiază conținutul lui px în py făcând astfel ca py să se modifice odată cu px.

O mare atenţie trebuie acordată tipului variabilei spre care pointează un pointer. Următorul exemplu este sugestiv în acest sens:

```
int *p;
double x=1.23, y;
p=&x;
y=*p;              //valoarea lui y va fi total eronată datorită tipului pointerului p
      //p este pointer spre întreg; *p va lua din x doar primii 4 octeti si apoi ii va converti la int
                   // valoarea *p va fi convertita la double si salvata in y
                   //atentie : nu rezulta eroare de compilare, se da doar un warning
```

6.2. Aritmetica adreselor

Pentru salvarea datelor, un program scris în C poate folosi 3 tipuri de memorie – care partajază o zonă comună și anume zona de date a programului. Astfel, zona de date se imparte in următoarele: memoria statică, stiva și memoria dinamică (heap).
Codul aferent funcțiilor care sunt necesare pentru execuția programului (funcția main precum si toate celelalte funcții care sunt apelate în program) este salvat într-o altă zonă de memorie denumită zona de cod.

Zona de date împreună cu zona de cod reprezintă spațiul de memorie alocat de sistemul de operare pentru execuția unui program.

In cele ce urmează, ne vom referi la zona de date. Pointerii obisnuiți din program referă adrese din această zonă.

1. **Memoria globală sau statica** găzduiește variabile definite globale sau variabilele definite cu static (în fișiere, funcții...). Caracteristica esențială a acestei zone este că ea este alocată și inițializată de compilator înainte ca execuția programului să intre în funcția main, și există până după ce această funcție principală se încheie. Mărimea zonei globale este determinate la compilare si este dată de mărimea variabilelor globale și statice din program. Mărimea acestei zone este fixă, pe toată durata execuției programului. Dacă nu se specific altfel, toate variabilele din zona globală sunt initializate cu valoarea 0.

2. **Stiva** este o zonă de memorie unde compilatorul alocă spații conform principiului stivei (LIFO). Aceste spații sunt legate strict de funcții, de variabilele definite în cadrul funcțiilor. În această zonă se definesc de către compilator variabilele locale (funcțiilor sau domeniilor de vizibilitate) Aici intră parametrii cu care funcția este apelată și variabilele locale obișnuite - **NU** și cele precedate de static. Deci, funcția main are zona de stiva A, la intrare în funcție. Dacă în main există un apel la funcția f(), la apelul acesteia, în stivă se adaugă zona B care conține variabilele definite în f. O zonă pe stivă există cât timp execuția se află în respectiva funcție sau în funcții apelate mai departe din aceasta. Odată ce funcția returnează, zona aferentă de stivă este dezalocată, și zona de stivă curentă va fi cea aferentă a funcției apelante.

3. În **memoria dinamică** sau **heap** alocăm variabilele dinamic, adică în timpul rulării programului (cu funcția malloc). Această zonă de memorie stă la dispoziția programului, care face alocări în funcție de cele petrecute în timpul execuției.

Pe parcursul execuției programului, zona de stivă respectiv heap crește si descrește, după cum programul intră sau iese din funcții, sau după cum programatorul alocă / dezalocă variabile dinamice.

Pentru primele 2 tipuri de memorie (zona globală și zona de stivă), programatorul știe la momentul scrierii programului cantitatea de memorie necesară la un anume punct in execuția programului, si anume variabilele disponibile intr-un punct de execuție, însă pe heap se alocă variabile în funcție de necesarul la rulare. Spre exemplu, un editor text va aloca șirul de caractere pe heap, câtă vreme nu știe dacă utilizatorul va introduce 10, 100 sau mai multe caractere.

Dacă p este un pointer, atunci p++ incrementează pe p în așa fel încât acesta să pointeze pe elementul următor indiferent de tipul variabilei pointate, iar p+=i incrementează pe p pentru a pointa peste i elemente din locul unde p pointează curent.

Dacă p și q sunt pointeri, relatii ca <, >, ==, !=, funcționează.
p < q
este adevarata, de ex, în cazul în care adresa lui p este mai mica (logic) decât adresa lui q. dacă p și q pointează către 2 elemente ale aceluiași tablou, atunci p < p înseamnă faptul că p arată către un element cu indice mai mic decât q.
Dacă se testează cu < > pointeri care sunt adrese efective, rezultatul (din punct de vedere al variabilelor din program) nu are sens. Insă acest lucru este permis de către compilator.

Relatiile == și != sunt și ele permise, intre 2 pointeri. De asemenea, orice pointer poate fi testat cu NULL. Dacă un pointer este == NULL inseamnă că adresa de memorie spre care pointează este adresa 0, ceea ce e un non-sens.

Instructiunea

```
p + n
```

desemneaza al n-lea obiect din memorie dupa cel pointat curent de p. Acest lucru este adevarat indiferent de tipul obiectelor pe care p a fost declarat ca pointer. Compilatorul atunci cind il intilneste pe n, il decaleaza în functie de lungimea obiectelor pe care pointeaza p, lungime determinata prin declaratia lui p (sizeof(*p)).

Este validă și scăderea pointerilor: dacă p și q pointează pe elementele aceluiaşi tablou, p-q este numărul de elemente dintre p și q. Acest fapt poate fi utilizat pentru a scrie o nouă versiune a lui strlen.

```
int strlen(char *s)//returneazã lungimea sirului
{
     char *p = s;
     while (*p != '\0')
          p++;
     return(p-s);
}
```

Prin declarare, p este initializat pe s, adică să pointeze pe primul caracter din s. In cadrul buclei while este examinat fiecare caracter până se întâlneşte \0 care semnifică sfirşitul sirului de caractere iar apoi se scad cele 2 adrese.
Este posibila omiterea testului expilcit iar astfel de bucle sint scrise adesea

```
while (*p)
p++;
```

Deoarece p pointeaza pe caractere, p++ face ca p sa avanseze de fiecare data pe caracterul urmator, iar p-v da numarul de caractere parcurse, adica lungimea sirului. Aritmetica pointerilor este consistenta: daca am fi lucrat cu float care ocupa mai multa memorie decit char, și daca p ar fi un pointer pe float, p++ ar avansa pe urmatorul float.

Toate manipularile de pointeri iau automat în considerare lungimea obiectului pointat în asa fel încât trebuie să nu fie alterat.

Operații permise : adunarea sau scaderea unui pointer cu un intreg, scaderea sau compararea a doi pointeri.
Nu este permisa adunarea, impartirea, deplasarea logica, sau adunarea unui float sau double la pointer.

6.3. Transmiterea argumentelor la apelurile de funcții

La apelul unei funcţii argumentele se transmit prin stivă, in ordinea în care apar în lista de argumente a funcțiilor. Aceasta înseamnă că pentru fiecare argument al funcției, pe stivă se crează o variabilă locală (funcției) cu numele argumentului, iar valoarea transmisă este utilizată la inițializarea acestei variabile locale. Evident, la sfârșitul execuției funcției, aceste variabile locale fiind salvate pe stivă se pierd, si valorile din ele nu mai sunt disponibile după terminarea apelului funcției. Spunem că parametrii s-au transmis **funcției prin valoare.**

Practic, la transmiterea prin valoare, se creează copii temporare ale parametrilor care se transmit. Funcţia apelată lucrează cu aceste copii iar la revenire valorile variabilelor parametru vor fi nemodificate. Deci este imposibil ca funcţia apelată să modifice unul din argumentele reale din apelant.

Exemplu:

```
#include<stdio.h>
void schimbare(int x, int y)     //se vor crea copii ale variabilelor x şi y
{ int tmp;
  tmp=x;
  x=y;
  y=tmp;              //în cadrul copiilor variabilelor se face inversarea
}                     //la revenire copiile variabilelor x şi y se distrug

int main()
{ int x=5, y=7;
  schimbare(x,y);           //transmitere prin valoare
  printf("%d %d\n",x,y);//valorile rămân nemodificate adică se va afişa 5 7
}
```

In acest fel, se transmit doar argumentele de tipul input pentru funcții.

Dacă dorim ca funcția să modifice valorile variabilelor primite ca și argument iar valorile să se regăsească modificate în funcția apelantă după finalizarea apelului funcției apelate, trebuie să facem disponibile funcției apelate adresele variabilelor transmise ca si argumente. In acest caz, spunem că facem transmitere prin **adresă sau referință**.

La transmiterea prin adresă, se transmite adresa variabilelor parametru. Toate operaţiile în cadrul funcţiei apelate se fac asupra zonei originale. Dacă se doreşte alterarea efectivă a unui argument al funcţiei apelante, trebuie furnizată adresa variabilei ce se doreşte a se modifica (un pointer la această variabilă). Funcţia apelată trebuie să declare argumentul corespunzător ca fiind un pointer.

Exemplu:

```
#include<stdio.h>
void schimbare(int *x, int *y)
{
      int tmp;
      tmp=*x;
      *x=*y;
      *y=tmp;               //se face inversarea asupra zonei originale
}

int main()
{ int x=5, y=7;
  schimbare(&x,&y);         //transmitere prin adresă
  printf("%d %d\n",x,y);//valorile sunt inversate adică se va afişa 7 5
}
```

Pentru o funcție, argumentele care sunt de tipul Input-output sau doar output trebuiesc transmise (obligatoriu) prin adresă.

În cazul în care apar masive, având în vedere că numele tabloului este un pointer constant spre primul element al lui, se foloseşte numele masivului ca adresă de început. Elementele masivului nu vor fi copiate iar operaţiile se vor face astfel asupra zonei originale.

O utilizare comună a argumentelor de tip pointer (transmitere prin adresă) se întâlnește în cadrul funcțiilor care trebuie să returneze mai mult decât o singură valoare. De exemplu, constatăm faptul că funcția scanf utilizează transmiterea prin adresă, datorită faptului că, in argumentele sale, trebuie să regăsim valorile care se citesc.

6.4. Pointeri și tablouri

In C, există o relatie strânsă între pointeri și tablouri, încât pointerii și tablourile pot fi tratate simultan. Orice operație care poate fi rezolvată prin indici în tablouri, poate fi rezolvată și cu ajutorul pointerilor. Versiunea cu pointeri va fi în general, mai rapidă.

Declarația

```
int a[10]
```

definește un tablou de dimensiunea 10, care este un bloc de 10 variabile de tip int salvate la adrese consecutive numite a[0], a[1], ..., a[9] notația a[i] desemneaza elementul deci poziţiile in tablou, numărate de la începutul acestuia.

Daca pa este un pointer pe un întreg, declarat ca

```
int *pa
```

atunci asignarea

```
pa = &a[0]
```

face ca `pa` să pointeze pe al "zero-ulea" (primul) element al tabloului a; aceasta inseamna ca `pa` conţine adresa lui `a[0]`.

Așadar asignarea `x = *pa` va copia conținutul lui `a[0]` în x.

Daca `pa` pointează pe un element oarecare al lui a atunci prin definiţie `pa+1` pointeaza pe elementul urmator și în general `pa-i` pointeaza cu i elemente inaintea elementului pointat de pa iar `pa+i` pointeaza cu i elemente dupa elementul pointat de pa.

Astfel, dacă pa pointează pe a[0]

```
*(pa + 1)
```

refera conţinutul lui `a[1]`, `pa + i` este adresa lui `a[i]` și `*(pa+i)` este conţinutul lui `a[i]`.

Aceste observații sunt adevărate indiferent de tipul variabilelor din tabloul a. Definiția "adunării unității la un pointer " și prin extensie, toată aritmetica pointerilor este de fapt calcularea prin lungimea în memorie a variabilei pointat. Astfel, în pa+i, i este înmulţit cu lungimea variabilei pe care pointează pa (sizeof(*pa)) înainte de a fi adunate la pa.

Corespondența între indexare și aritmetica pointerilor este evident foarte strânsă. De fapt, numele unui tablou este convertit de către compilator într-un pointer constant pe începutul zonei de memorie unde se află tabloul.
Efectul este că numele unui tablou este o expresie pointer.

Aceasta are câteva implicații utile. Din moment ce numele unui tablou este sinonim cu locația elementului său zero, asignarea

```
pa = &a[0]
```

poate fi scrisă și `pa = a`

O referință la `a[i]` poate fi scrisă și ca `*(a+i)`. Evaluând pe `a[i]`, C îl convertește în `*(a+i)`; cele două forme sunt echivalente. Aplicând operatorul & ambilor termeni ai acestei echivalențe, rezultă ca &a[i] este identic cu a+i: a+i adresa elementului al i-lea în tabloul a.
Reciproc, dacă pa este un pointer el poate fi utilizat în expresii cu un indice; `pa[i]` este identic cu `*(pa+i)`.

Pe scurt orice tablou și exprimare de indice pot fi scrise ca un pointer și deplasament și orice adresă chiar în aceeași instrucțiune. Trebuie ținut seama de o diferență ce există între numele tablou și un pointer. Un pointer este o variabilă, astfel ca `pa=a` și `pa++` sunt operaţii.

Dar, un nume de tablou este o constantă, de aceea construcţii ca a=pa sau a++ sunt interzise.

Atunci când se transmite un nume de tablou unei funcţii, ceea ce se transmite este locația de inceput a tabloului. In cadrul funcţiei apelate acest fapt argument este o variabilă ca oricare alta astfel încat un argument nume de tablou este un veritabil pointer, adica o variabila continind o adresa.

Ne vom putea folosi de aceasta pentru a scrie o nouă versiune a lui strlen, care calculeaza lungimea unui sir.

```
int strlen(char *s) // returnează lungimea sirului s
{
     int n;
     char *ps;
     for (n = 0, ps = s; *ps != '0'; ps++)
          n++;
     return n;
}
```

Ca parametri formali în definirea unei functii

```
char s[]
```

și

```
char *s;
```

sunt echivalenti; alegerea celui care trebuie scris este determinata în mare parte de expresiile ce vor fi scrise în cadrul functiei. Atunci cind un nume de tablou este transmis unei functii, aceasta poate, dupa necesitati s-o interpreteze ca tablou sau ca pointer și sa-l manipuleze în consecinta. Functia poate efectua chiar ambele tipuri de operatii daca i se pare potrivit și corect.

Este posibilă și transmiterea către o funcție doar a unei părți dintr-un tablou prin transmiterea unui pointer pe începutul subtabloului. De exemplu, dacă a este un tablou;
`f(&a[2])`
si
`f(a + 2)`
ambele transmit functiei f adresa elementului a[2] deoarece &a[2] și a+2 sint expresii pointer care refera al treilea element al lui a. În cadrul lui f, declarea argumentului poate citi

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

```
sau
f(int *arr)
     {
        ...
     }
```
Astfel, dupa cum a fost concepută funcția f aptul că argumentul referă de fapt o parte a unui tablou mai mare, nu are consecințe.

6.5 Pointeri pe caractere și functii

Un sir constant scris astfel " … " este un tablou de caractere (de tipul char[]). În reprezentare internă, compilatorul termină un tablou cu caracterul \0 în aşa fel încât programele să poată detecta sfârşitul.
Lungimea în memorie pentru un sir de caractere ("…") este astfel mai mare cu 1 decât numărul de caractere cuprinse între ghilimele.

```
char message[40] = "now is the time";
```

defineste un sir de 40 de caractere şi îl asignează cu sirul de caractere `"now is the time".`

Exemplu : `strcpy(char s[], char t[])` copiaza sirul t în sirul s, versiunea cu tablouri :

```
int mystrcpy(char s[], char t[]) //copiaza t în s
{
   int i;
   i = 0;
   while ((s[i] = t[i]) != '\0')
   i++;
   return i;
}
```
Versiunea lui mystrcpy cu pointeri

```
strcpy(char *s, char *t)   //copiaza t în s, versiunea pointeri
    {
while ((*s = *t) != '\0') {
s++;
t++;
        }
    }
```

Deoarece argumentele sunt transmise prin valoare (atentie: avem pointeri transmisi prin valoare), mystrcpy poate utiliza s şi t în orice fel se doreşte. Aici ei sunt convenţional utilizati ca pointeri, care parcurg tablourile pina în momentul în care s-a copiat \0 sfirsitul lui t, în s.

In practică, mystrcpy nu va fi scris asa cum s-a aratat mai sus. O a doua posilitate ar fi

```
int mystrcpy(char s[], char t[]) //copiaza t în s
{
      int i = 0;
      while ((*s++ = *t++) != '\0') i++;
      return i;
}
```

In aceasta ultimă versiune se imită incrementarea lui s şi t în partea de test. Valoarea lui *t++ este caracterul pe care a pointat inainte ca t sa fi fost incrementat; prefixul ++ nu-l schimba pe t inainte ca acest caracter sa fi fost adus.înacelasi fel, caracterul este stocat în vede a pozitie s inainte ca s sa fie incrementat. Acest caracter este deasemenea valoarea care se grupeaza cu \0 pentru simboul buclei. Efectul net este ca, caracterele sint copiate din t în s, inclusiv sfirsitul lui \0.

Ca o ultima abreviere vom observa ca şi gruparea cu \0 este redundantă, astfel că bucla while poate fi scrisă:

```
while (*s++ = *t++)
```

Desi aceasta versiune poate părea complicata la prima vedere, aranjamentul ca notatie este considerat suveran daca nu exista alte raţiuni de a schimba.

Rutina este `mystrcmp(s, t)`, compara sirurile de caractere s şi t şi returneaza negativ, zero sau pozitiv în functie de relatia dintre s şi t; care poate fi: s<t, s=t sau s>t.
Valoarea returnată este obţinută prin scăderea caracterului de pe prima poziţie unde s diferă de t.

```
int mystrcmp(char s[], char t[])  // returneaza <0 daca s<t, 0 daca s==t, >0 daca s>t
{
      int i;
      i = 0;
      while (s[i] == t[i])
            if (s[i++] == '\0')
                  return(0);
      return(s[i] - t[i]);
}
```

Versiunea cu pointeri a lui strcmp.

```
int mystrcmp(char s[], char t[]) // returneaza <0 daca s<t, 0 daca s==t, >0 daca s>t
{
      for ( ; *s == *t; s++, t++)
            if (!*s)
                  return(0);
      return(*s - *t);
}
```

Dacă ++ şi -- sunt folosiţi altfel decât operatori prefix sau postfix pot apare alte combinaţii de * şi ++ şi --, deşi mai puţin frecvente.

Exemplu: `*++p` incrementeaza pe p inainte de a aduce caracterul pe care pointeaza p.
`*--p` decrementează pe p în acelasi condiţii.

La lucrul cu siruri de caractere, se preferă iterarea folosind pointeri de tipul char*, si nu indecşi pe sir.

6.6 Pointeri pe caractere şi functii

Se solicită atenţie sporită la initializarea sirurilor de caractere.

Astfel, dacă în program scriem un sir de caractere utilizand semnul “”, programul va aloca sirul ca şi o constantă şi il va păstra la o zonă de memorie care este dincolo de controlul nostru in program.

Astfel, şirul “how are you?” este tratat de către program ca şi o constantă de tipul char[], iar pentru această constantă se alocă 13 octeti (12 octeţi pentru caracterele sirului şi un octet pentru \0).

Următoarea initializare este validă și corectă:

char sir[30] = “how are you?”;

astfel, compilatorul alocă o zonă de memorie pentru variabila sir , de mărime 30 octeți, iar pe această zonă copiază sirul “how are you?”.

Daca scriem:

char *psir = “how are you?”;

atunci compilatorul alocă o constantă de tipul char* cu valoarea “how are you?” și face ca pointerul psir să arate către zona de memorie unde este alocată constanta de tipul char*. Aceasta inseamnă că daca vom dori să modificăm continutul zonei de memorie pointate de psir (care este constant) va genera o eroare de compilare.

Concluzia este că dacă dorim să utilizăm variabile (neconstante) de tip sir de caractere in program, avem 2 alternative:

1. Definim variabila ca si sir de caractere folosind sintaxa de sir : char* sir[dimensiune]. In acest caz, se alocă (in zona globală sau zona de stivă) variabila sir de dimeniunea specificiată, iar mai apoi vom putea manipula această zonă de memorie după cum dorim
2. Definim variabila ca și pointer de tipul char*. : char *psir;. Apoi va trebui să alocăm dinamic acest sir folosind funcția de alocare malloc. După aceea vom putea manipula continutul acestei zone de memorie. La finalul utilizării, programatorul trebuie să să dezaloce zona de memorie folosind funcția free. Alocarea de memorie este realizată dinamic, in zona de heap.

In ambele cazuri, programatorul va manipula continutul zonei de memorie folosind funcțiile de lucru pe siruri de caractere din string.h: strcpy, strcmp, strcat etc.

In cazul 2, programatorul trebuie sa aloce o zonă de memorie cel putin egală cu lungimea sirului care se doreste a fi salvat + 1, si anume strlen(sir) + 1

Pe parcursul programelor pe care le scrie, vom evita să realizăm atribuiri de tipul:

pchar = “text”;

pentru ca acestea nu au sens: ele fac variabila pointer de tipul char* să pointeze către o zonă de memorie dincolo de controlul programului nostru.