Programarea Calculatoarelor. Laborator

Versiune 0.31

Cristian Vicas

nov. 20, 2023

Contents

L1. Setarea mediului de lucru

1.1 Bine Ați Venit!

Bine ați venit la "Programarea Calculatoarelor", laborator!

Scopul primului laborator este de a vă seta și pune la punct mediul de lucru. Odată ce exemplul "Hello World" rulează, setarea mediului de lucru se consideră bine făcută.

Obiective:

- Mediul de dezvoltare trebuie să meargă pe calculatorul de laborator și pe dispozitivul studentului.
- Exemplul "Hello World" rulează pe toate dispozitivele.
- Puteți spune, empiric, ce rol are fiecare element din program.

1.2 Setare mediu de lucru

Sunt mai multe alternative, pentru început vă recomand pe cele simple, OnlineGDB și CodeBlocks. Acestea au "împachetate" toate cele necesare rulării unui program simplu. (IDE, toolchain)

După ce știți să scrieți câteva linii de cod, puteți încerca Eclipse. Eclipse este "doar" un IDE. Trebuie să aveți pe calculator un *toolchain*. Există tutoriale pentru toate sistemele de operare mari, urmăriți-le cu atenție.

Programele mari (gen Eclipse sau NetBeans) nu vin cu toolchain-ul lor preinstalat pentru că, în practică, există zeci sau sute de *toolchain*-uri. Sunt foarte multe tipuri de arhitecturi, foarte multe variante, iar IDE-urile mai serioase permit să lucrați relativ ușor cu aceste toolchain-uri variate.

1.2.1 CodeBlocks

Pe calculatoarele din sala de laborator aveți instalat mediul *CodeBlocks*. Urmați pașii din îndrumătorul de laborator al colegului meu, Ionel Giosan: Introducere în programare. Scheme logice și limbaj pseudocod. Familiarizarea cu mediul de dezvoltare CodeBlocks

Urmați instrucțiunile de la secțiunea: **Primul proiect în CodeBlocks**

Mai aveți câteva opțiuni la dispoziție, mai ales pe dispozitivele de acasă.

1.2.2 OnlineGDB

Există un portal unde puteți scrie cod direct în browser: OnlineGdb. Vă recomand să vă faceți un cont pentru a vă putea salva munca. Portalul este relativ limitat (nu se poate face *debug* ușor) dar este suficient pentru primele 2-3 laboratoare!

Din păcate, unele setări de securitate și dependința de un serviciu extern il fac bun doar pentru a încerca idei, rapid, atunci când nu avem acces la un mediu complet.

1.2.3 Eclipse CDT

Mediul de dezvoltare Eclipse, este un mediu profesionist, *open-source* și *heavyweight* ("greu"). Deși va fi greoi ca primul contact cu C-ul, (trebuie să aveți răbdare să citiți tutoriale) îl recomand cu căldură. CodeBlocks își va arăta limitarea rapid, când avem proiecte un pic mai mari.

Eclipse CDT funcționează pe majoritatea platformelor (Windows, Linux, Mac).

Mergeti la unul din tutorialele care explică cum să vă instalați toolchain-ul.

Căutați `Eclipse IDE for C/C++ Developers` aici: Eclipse IDE

Pe scurt, trebue să:

- Instalați MinGW (Minimalist GNU for Windows)
  - Căutați mingw-get-setup.exe
  - Instalați
  - Rulați Instalation Manager și selectați mingw32-base, mingw32-gcc-g++, msys-base. Apoi `Installation -> Apply Changes`
- Editați căile sistemului (la fel ca în tutorial)
- Instalați Eclipse CDT (dezarhivați zip-ul)
- Creați un nou proiect "Hello World" C, click dreapta pe el `Build Project` apoi `Run As -> Local C/C++ Application`
- Erori? Întrebați!

1.2.4 NetBeans

NetBeans este un alt IDE popular, mai ales în mediul Java. Trebuie să vă instalați Java Runtime Environment de la Oracle pentru a putea rula NetBeans.

Apoi, intrați pe tutorial pentru a vă activa *plugin*-ul de C și instala *toolchain*-ul.

L2. Instrucțiuni de control al execuției programului

2.1 Recapitulare

În acest moment putem să apelăm un toolchain care să transforme codul sursă în cod executabil. Putem executa acest cod și observa rezultatele.

Am văzut structura unui program minimal, unde trebuie să scriem cod, și câteva blocuri constructive care declară variabile și execută afișări pe ecran. Deocamdată le vom folosi așa cum sunt, mai încolo, intrând în detalii.

- "Linia de text" `int i;` declară o variabilă în care putem stoca și opera cu întregi cu semn.
- "Linia de text" `printf("%d ", i);` cu care pot afișa conținutul lui `i` împreună cu un mesaj, pe ecran
- Construcția `\n` care scrisă între ghilimelele de la `printf` trece la o linie nouă pe ecran
- Instrucțiunile pot fi grupate în blocuri de instrucțiuni
- Codul se scrie în funcția `main()` și se va executa secvențial.
- Expresii matematice simple.

Câteva expresii:

- `a = 5` va păstra în `a` valoarea 5
- `i = i + 1` va păstra în `i` valoarea incrementată cu 1 a lui `i`
- `a == 5` va fi adevărată dacă `a` are valoarea 5.
- Operații matematice, `+ - * /`

Am introdus oficial instrucțiunile: *for* și *if*.

Convenția din C este că dacă o expresie are valoarea zero (0) ea este considerată falsă din punct de vedere al logicii boleene.

2.1.1 *for*

Definiția este:

`for(` **A** `;` **B** `;` **C** `)` **instructiune**

A este expresia de inițializare, **B** condiția de continuare (sau de terminare) iar **C** este expresia de reinițializare

A se execută o singură dată la începutul ciclului *for*, apoi vine instrucțiunea care va fi repetată (care poate fi și un bloc de instrucțiuni). Apoi, pe rând, se execută **C** urmat de evaluarea lui **B**. Dacă **B** rămâne adevărată, iterația continuă.

Atenție, la final, expresia **C** va avea cu o execuție în plus față de numărul de rulări ai blocului **instrucțiune**.

2.1.2 *if*

`if(` **condiție** `){`

    cod A

`}else{`

    cod F

`}`

Dacă **condiție** se evaluează la o valoare diferită de zero, se va executa **cod A**. Dacă se evaluează ca fals, se va executa **cod F**. Asta înseamnă că bucata **cod A** se va executa doar când condiția este adevărată. Partea cu `else` poate să lipsească.

Atenție la înlocuirea blocului cu o singură instrucțiune și la confuziile între = și ==.

2.2 Noțiuni noi introduse

2.2.1 *while*

`while(` **B** `)` **instructiune**

Cât timp expresia **B** este adevărată se va executa instrucțiunea.

Atenție, utilizatorul este responsabil de a termina bucla while. Dacă condiția **B** nu va fi falsă niciodată, avem un ciclu infinit. **instructiune** poate fi și un **bloc de instrucțiuni**.

2.2.2 *do ... while*

`do` **instructiune** `while(` **B** `);`

Execută instrucțiunea cât timp B este adevărată

Se va executa **instructiune** cel puțin o dată. Apoi se va evalua **B**. În funcție de valoarea de adevăr, se va relua ciclul.

Reiterăm, de obicei, în limbajul C, acolu unde este legal să apară o instrucțiune este legal să apară și un bloc de instrucțiuni.

2.2.3 *break*

Dacă vrem să ieșim mai repede din ciclul *for* putem apela această instrucțiune. Se va ieși imediat. Util când, căutăm o valoare într-un șir și odată găsită, nu are rost să iterăm până la capăt.

2.2.4 *continue*

Se întâmplă să putem decide că iterația curentă nu mai trebuie executată până la capăt și că putem trece la următoarea.

În momentul întâlnirii ei, se va executa expresia de reinițializare și condiția de continuitate (la *for*, *while*) respectiv se va reveni la începutul blocului de instrucțiuni pentru *do...while*

2.2.5 *goto și switch*

Instrucțiunea *goto* nu o folosiți

La *switch* vom reveni după ce stăpâniți mai bine expresiile și constantele.

2.3 Probleme rezolvate

2.3.1 Afișarea numerelor divizibile cu 3

Problemă: Afișati numerele divizibile cu 3, mai mari decât zero și mai mici decât 100. Reluați și adaptați exemplul de la curs!

Sugestie de rezolvare:

Facem iterativ. Primul pas al fiecărei iterații, *proiectarea* este dat. Voi trebuie să executați *implementarea*, *rularea* și *testarea*.

Preferabil, la fiecare iterație să adăugați cod. Nu ștergeți codul vechi din iterația anterioară dacă nu vă mai trebuie, Comentați-l!

Puteți să vă verificați și singuri de succesul fiecărei iterații!

Iterația 1:

Creați un proiect nou cu "Hello World" și rulați să fiți siguri că mediul de dezvoltare este configurat ok

Iterația 2:

Declarați o variabilă, inițializați cu o valoare și afișați-o

Iterația 3:

Afișați restul împărțirii cu 3 a variabilei. Încercați mai multe numere pentru variabilă.

Iterația 3:

Adăugați cod care să afișeze dacă variabila este divizibilă cu 3 sau nu. Încercați mai multe numere.

Iterația 4:

Iterați pentru toate valorile între 0 și 100, cele două instrucțiuni de afișare de mai sus.

Iterația 5:

Adăugați condiție pentru a afișa doar numerele divizibile cu 3. Dacă facem această iterație am îndeplinit cerințele problemei!

Iterația 6:

Curățați codul, aranjați textul, aliniați instrucțiunile și blocurile, înfrumusețați ieșirea în consolă și mai faceți o ultimă verificare.

Felicitări!

Mai jos, găsiți o altă soluție posibilă. tot cu o abordare iterativă. De la simplu la complex.

Listing 2.1: Iterația 1. Hello World!

```
#include <stdio.h>
int main()
{
    printf("Hello World");
    return 0;
}
```

Listing 2.2: Iterația 2 Variabile și afișare

```
#include <stdio.h>
int main()
{
    int i=32;
    printf("i este %d", i);
    return 0;
}
```

Listing 2.3: Iterația 3. Calculul restului pentru o valoare.

```
#include <stdio.h>
int main()
{
    int i=32;
    printf("i este %d\n", i);
    printf("Restul este %d", i % 3);
    return 0;
}
```

Listing 2.4: Iterația 4. Calculul restului pentru toate valorile

```
#include <stdio.h>
int main()
{
    int i=32;
    for(i = 0; i < 100; i = i + 1){
       printf("i este %d\n", i);
       printf("Restul este %d\n", i % 3);
    }
    return 0;
}
```

Listing 2.5: Iterația 5. Afișarea condiționată

```
#include <stdio.h>
int main()
{
    int i=32;
    for(i = 0; i < 100; i = i + 1){
       if (i % 3 == 0)
          printf("i este %d\n", i);
//        printf("Restul este %d\n", i % 3);
    }
    return 0;
}
```

Listing 2.6: Iterația 6. Curățare cod și afișare mai elegantă.

```
#include <stdio.h>
int main()
{
    int i;
    printf("Numerele intre 0 si 100, divizibile cu 3, sunt:\n");
    for(i = 0; i < 100; i = i + 1)
       if (i % 3 == 0)
          printf("%d ", i);
    return 0;
}
```

2.4 Probleme propuse

Acum e rândul vostru să abordați problemele iterativ. Puteți să faceți iterații mai mari, dar atenție, dacă vă împotmoliți sau merge mai greu, reveniți la iterații mici!

2.4.1 Minimul unei funcții pe un interval dat I

Luați funcția matematică $f(x) = x^4 - 5x^2 + 3x + 20$ și găsiți punctul x_0 pentru care funcția e minimă. Se dă un interval de căutare [-4, 4] și un pas de căutare, *eps=0.001*.

Sugestii:

Cu instrucțiunea `double x;` vă declarați o variabilă ce poate stoca numere în virgulă mobilă. Afișarea numerelor cu zecimale se face cu instrucțiunea `printf("%f", x);` unde `x` este variabila de tip `double` declarată anterior.

NU rezolvați problema matematic (algebric) din prima. Parcurgeți intervalul dat, cu pasul cerut. Țineți minte cea mai mică valoare, într-o variabilă.

Evaluați funcția la fiecare pas. Dacă valoarea curentă e mai mică decât valoarea stocată anterior, rețineți noua valoare.

Afișați atât punctul de minim cât și valoarea funcției în punctul respectiv.

Pentru verificare rezolvați analitic sau puteți introduce expresia în zona de căutare a motorului Google. Vă va plota frumos graficul. Verificați dacă răspunsul dvs este în zona minimului.

2.4.2 Primele numere divizibile cu un număr dat

Scrieți primele *n* numere mai mari sau egale cu 0, divizibile cu un număr *a*. Valorile *n* și *a* se scriu în variabile.

De exemplu, puteți declara așa, în interiorul funcției `main`: `double a = 7;` și `double n = 20;`.

ATENȚIE!! Se cere primele *n* numere, și NU numerele mai mici decât *n*, ca la problema rezolvată.

Puteți începe de la problema anterioară. Vă sugerez să folosiți instrucțiuni *while*. Cu atenție să nu intrați în ciclu infinit.

Ca o primă iterație, rezolvați problema analitic și folosiți un `for`. Dar vă rog să încercați să folosiți `while` și instrucțiunea *break* pentru a ieși din bucla *while*.

L3. Operatori și Expresii I

3.1 Recapitulare

În acest moment:

- Putem lucra cu variabile de tip întreg și virgulă mobilă, putem face operații matematice, putem afișa rezultatele, putem controla execuția programului, itera și lua decizii pe baza valorii expresiilor.
- Am văzut cum arată diverse tipuri de date, cum să determinăm câtă memorie ocupă un tip de date in memorie.
- Am văzut pe scurt, cum se definește o funcție și știm cum să apelăm funcții din diversele biblioteci.

3.1.1 Expresii

Egalitatea e testată cu ==. Avem clasicul *strict mai mic* < și *strict mai mare* > pe care le putem combina cu egalitatea: *mai mic sau egal* <= respectiv >=.

Avem operații logice *și* &&, *sau* `||`, *negare* `!` și prescurtarea pentru *diferit*, `!=`.

Dacă unele concepte enumerate nu vă sunt clare, reluați cursurile și laboratoarele anterioare.

Pe lângă cunoștințele acumulate la laboratorul anterior, am văzut cum se poate lucra cu intrarea și ieșirea la tastatură.

3.1.2 Afișări pe ecran

`printf("%d ", i)` afișează variabila de tip `int` pe ecran. Construcția `%d` specifică cum anume se va interpreta pattern-ul de biți a lui `i`. (Specificatorul de format). Avem specificatori pentru tipul `float`, `char`, și șiruri de caractere.

3.2 Noțiuni noi introduse

3.2.1 Operatorii relaționali:

3.2.2 Incrementarea

Operația `i = i + 1` o putem prescurta cu operatorul de incrementare, ca `i++` sau `++i`. Căutați diferența între valorile acestor operații, după execuția lor. Primul se numește *postfix increment* al doilea *prefix increment*.

Similar se poate folosi și `i--` respectiv `--i`. Folosirea lor în expresii mai complexe nu este recomandată deoarece introduce o dificultate la citire.

Altă prescurtare este `i += 1`. Operatorul de egalitate se poate combina în acest fel și cu alți operatori matematici și pe biți.

3.2.3 Operatori pe biți

Între două variabile se pot executa operații pe biți. Ele sunt:

și & pe biți, *sau* |, *negare* ~, *sau-exclusiv* ^, și două operații mai speciale, deplasarea pe biți:

<< deplasarea la stânga a pattern-ului de biți. Valorile din stânga se pierd și se inserează zero-uri în dreapta.

>> deplasare la dreapta: Biții din dreapta se pierd, la stânga se inseră o valoare dependentă de tip și de compilator. De obicei, operațiile cu biți le executați pe tipuri fără semn (cu modificatorul *unsigned*). Pe tipuri fără semn, se va insera 0. Pe tipurile cu semn, uneori, se inseră bitul de semn.

3.2.4 Asocierea

Există reguli de asociere. Deocamdată știm că înmulțirea este mai prioritară decât adunarea. Pentru restul, folosim deocamdată parantezele.

3.3 Probleme propuse

3.3.1 Minimul unei funcții pe un interval dat II

Continuați problema din laboratorul anterior (problema *Minimul unei funcții pe un interval dat I*) și calculați valoarea polinomului folosind o funcție. Suplimentar (și iterativ):

- Inițializați valoarea minimă candidată cu prima evaluare a funcției fără să folosiți o santinelă
- Intervalul de căutare, pasul de căutare, îl definiți la începutul funcției `main` inițializând variabilele.
- Formatați frumos afișarea rezultatelor.

3.3.2 Verificați deplasarea la dreapta sau la stânga

Matematic, deplasările pe biți sunt echivalente înmulțirii sau împărțirii cu 2.

Verificați acest lucru! Luați un număr oarecare, mai mare. (gen 3423433). Faceți operații de deplasare cu diverse valori (mai mici decât 10). Verificați și înmulțirea respectiv împărțirea cu 2.

Exemplu:

3423433 << 5 este 109549856

La fel, 3423433 >> 5 este 106982

Luați un ciclu *for* care să împartă sau să înmulțească numărul cu 2, de atâtea ori de cât este precizat în operandul din dreapta.

Atenție, folosiți o variabilă pentru numărul inițial, pentru a putea controla toate operațiile dintr-un singur punct!

Adică, luați variabila `n` pentru numărul inițial și `a` pentru numărul de deplasări. Tot restul codului să fie scris în funcție de `n` și `a`. Atenție, `n` trebuie să fie de tip `unsigned int`.

L4. Operatori și Expresii II

4.1 Recapitulare

În acest moment:

- Putem lucra cu variabile de tip întreg și virgulă mobilă, putem face operații matematice, putem afișa rezultatele, putem controla execuția programului, itera și lua decizii pe baza valorii expresiilor.
- Am văzut cum arată diverse tipuri de date, cum să determinăm câtă memorie ocupă un tip de date in memorie.
- Am văzut pe scurt, cum se definește o funcție și știm cum să apelăm funcții din diversele biblioteci.

4.1.1 Expresii

Egalitatea e testată cu ==. Avem clasicul *strict mai mic* < și *strict mai mare* > pe care le putem combina cu egalitatea: *mai mic sau egal* <= respectiv >=.

Avem operații logice *și* &&, *sau* `||`, *negare* `!` și prescurtarea pentru *diferit*, `!=`.

Dacă unele concepte enumerate nu vă sunt clare, reluați cursurile și laboratoarele anterioare.

Pe lângă cunoștințele acumulate la laboratorul anterior, am văzut cum se poate lucra cu intrarea și ieșirea la tastatură.

4.1.2 Intrări și ieșiri

`printf("%d ", i)` afișează variabila de tip `int` pe ecran. Construcția `%d` specifică cum anume se va interpreta pattern-ul de biți a lui `i`. (Specificatorul de format). Avem specificatori pentru tipul `float`, `char`, și șiruri de caractere.

Funcția `scanf`: `scanf("%d", &i);` preia un șir de caractere din buffer-ul de tastatură și îl interpretează ca un zecimal întreg (datorită lui `%d`). ATENȚIE la caracterul `&` în față la variabilă!

Căutați pentru `printf` și `scanf` documentația. Familiarizați-vă cu afișarea pe un număr fix de caractere, afișarea unui număr limitat de zecimale, spații, alinierea la dreapta sau la stânga, etc.

4.1.3 Abordarea iterativă

Ați văzut exemple de cum să abordați problemele pas cu pas. Trebuie ca această abordare să vă intre în reflexe. Câteva elemente de strategie:

- Alegeți o subproblemă pe care o știți rezolva (exemplu, declararea variabilelor)
- Simplificați problema (ex: Se cere afișarea unui număr cu fix 6 zecimale. Puteți simplifica, afișând simplu)
- Rulați codul des. Verificați ceea ce apare pe ecran și comparați cu ceea ce vă așteptați să apară.
- Dacă totul este ok, reluați de la început, alegând o nouă subproblemă sau introducând restricțiile eliminate anterior.
- Repetați până problema e rezolvată. Nu uitați să testați des!
- Nu ștergeți codul care nu vă mai trebuie. Comentați-l.

4.2 Probleme propuse

4.2.1 Minimul unei funcții pe un interval dat III

Continuați problema din laboratorul anterior (problema *Minimul unei funcții pe un interval dat II*) dar citind de la tastatură valorile intervalului și a pasului.

4.2.2 Afișați valoarea binară a unui întreg fără semn I

Declarați o variabilă întreagă fără semn `unsigned int` și afișați-i pattern-ul de biți. Afișați numărul în zecimal după care afișați pattern-ul de biți.

Observații:

Operatorul `sizeof()` este folosit pentru a afla numărul de octeți ocupat de variabilă.

Folosiți operatorii de deplasare și AND pe biți pentru a afla valoarea unui bit.

Complicare:

1) Afișați și valoarea hexazecimală a numărului.
2) Puneți codul într-o funcție.
3) Scrieți o funcție (sau funcții) care să afișeze CORECT pattern-ul de biți la valori de tip long double sau SIGNED int
4) Citiți valoarea de la tastatură

4.2.3 Scrieri sofisticate

Testați afișarea formatată. Veți afișa niște numere în diverse formaturi

Citiți un număr *n*, întreg. Veți citi un alt număr `a`, în virgulă mobilă, de exemplu `342.34435` și veți afișa pe rând numărul `a` la puterea 1, apoi 2, și tot așa până la `n`. Veți calcula și logaritm (în baza *e*) a rezultatului.

La fiecare pas, afișați rezultatul (pe un rând, fiecare rând va conține câteva câmpuri):

- Puterea
- Rezultatul ca număr în virgulă mobilă

- Rezultatul doar cu 3 zecimale
- Rezultatul în format științific/exponențial (cu E)
- Logaritmul, cu 2 zecimale.

Fiecare câmp va avea un număr mai mare de caractere decât necesare. Numerele vor fi aliniate la dreapta. Între câmpuri se va scrie virgulă.

Separați locul unde se fac calculele de locul unde se afișează (eg. nu efectuați înmulțiri în apelul la `printf`).

L5.Tabele I

5.1 Recapitulare

În cursul anterior v-ați familiarizat cu un tip de date structurate, tabelele de valori.

Se declară tipul, urmat de numele variabilei urmat de numărul de elemente dorit:

```
int tab[100];
```

Tabelul se poate inițializa la declarare, folosind acoladele: `int tab[4] = {5, 4, 3, 2};`

Accesul se face tot cu operatorul paranteză dreaptă: `tab[2] = 9`. Atenție, în C, indexarea este *zero based*. Primul element are indexul zero!

Atenție și la alocarea de memorie multă, pe stivă. Limitați dimensiunile tablourilor locale. Dacă aveți nevoie de memorie mai multă, rezervați pe heap, folosind `malloc` și pointeri.

Un tablou se poate declara la compilare sau se poate aloca dinamic, folosind pointerii. Dacă vrem să rezervăm o zonă de memorie de 5000 valori de tip `float` trebuie să:

- Declarăm o variabilă care să poată gestiona zona de memorie (un pointer de tip `float*`)
- Cerem de la sistemul de operare numărul de **octeți** dorit
- Folosim memoria (putem scrie operații de indexare ca și la tabele)
- Dealocăm.

Listing 5.1: Rezervarea memoriei pentru tabel, pe heap

```
int *pi;
pi = (int*) malloc(sizeof(int) * 1000000);
if (pi == NULL){
   // Tratez eroarea legata de lipsa memoriei.
}
// Folosesc zona de memorie alocată, folosindu-ma de variabila pi.
pi[10] = 32;
printf("%d\n", pi[16]);

free(pi); // Dealoc memoria cand am terminat de lucrat.
```

5.2 Noțiuni noi introduse

5.2.1 Constante și identificatori

Numele variabilelor și a identificatorilor în C, pot fi compuse din litere mari, litere mici, cifre și caracterul underscore _. Obligatoriu identificatorul trebuie să înceapă cu o literă sau cu underscore. Dacă nu v-ați dat seama până acum, limbajul C-ul este *case sensitive*. Variabila `suma` este diferită de variabila `Suma`. La fel și pentru instrucțiuni.

Constantele numerice se specifică implicit în baza 10, tip `int` Dacă adăugăm un `l` sau `L` la sfârșit, obținem tipul `long int`. Litera `u` sau `U` va determina un tip fără semn.

Constantele fără semn se pot specifica în hexazecimal punând prefixul `0x`, în binar `0b` iar pentru octal, prefixați cu un 0 urmat de cifre între 0 și 7.

Pentru constante în virgulă mobilă, avem notația cu punct zecimal, 32.4 sau notația exponențială: 0.324e2.

Constantele de tip șir de caractere sunt păstrate între ghilimele ". Vom reveni asupra lor. Deocamdată le folosim la afișarea și citirea formatată.

Constantele de tip caracter (UN SINGUR CARACTER) sunt păstrate între apostrof 'A'. Ele sunt interpretate ca întregi fără semn. Se pot folosi fără probleme în operații aritmetice.

Caracterele speciale tab, newline, se specifică cu un backslash: `\n` pentru newline și `\t` pentru tab. În constante, pentru backslash trebuie puse două backslash-uri: `\\`. La fel, caracterul apostrof sau ghilimele, trebuie precedat de backslash: `'\''`. Avem apostrof (început de constantă caracter), backslash, apostrof, apoi apostroful de închidere a constantei caracter. La fel și ghilimelele, dar DOAR în constantele de tip șir de caractere. Vice-versa, se poate și fără backslash: `'"'` respectiv: `"un apostrof: '"`.

5.3 Probleme propuse

5.3.1 Numere pare

Citiți `n` numere întregi de la tastatură. Memorați-le.

Afișați pe ecran doar numerele pare din șirul citit. Atenție, afișarea începe după ce s-au terminat de citit toate numerele.

Puteți lua un tabel de aprox 100 elemente.

5.3.2 Transformare carteziană - polară

Citiți o coordonată carteziană (formată din perechea x, y) și afișați coordonate polare (unghi în grade și lungimea vectorului).

Căutați și folosiți funcția **atan2** din biblioteca `<math.h>`

5.3.3 Afișați valoarea binară a unui întreg fără semn II

Reluați problema din laboratorul anterior (*Afișați valoarea binară a unui întreg fără semn I*) cu câteva modificări.

Scopul exercițiului este de a învăța să separați responsabilitățile codului. Există cod care generează și calculează reprezentarea binară și există cod responsabil de afișare.

Calculați rezultatul într-un tabel de 32 de elemente.

Scrieți funcții care să afișeze tabelul:

- cu toate cifrele binare continue
- cu cifrele binare grupate câte 4

Puteți "uni" cele două funcții folosind doar un nou parametru?

5.3.4 Evaluarea unui polinom I

Problemă: Se citește un polinom de numere reale de la tastatură. Se citește și un punct x și se evaluează polinomul.

Detalii: Se citește `n`, numărul de coeficienți. Apoi, fiecare coeficient pe rând. Ultima dată se citește valoarea `x`. Se va afișa frumos polinomul și rezultatul.

Observație: Coeficienții se vor stoca într-un tabel. Pot fi cel mult 100 de coeficienți. Dacă `n` are valoare mai mare, se va da un mesaj de eroare.

L6. Tabele II

6.1 Recapitulare

Acum ar trebui să fiți familiari cu a scrie instrucțiuni, cu a afișa sau a citi date și cu a efectua operații matematice mai complexe, cu variabile. Puteți opera cu funcții simple.

Din Iterația II, deocamdată știți să creați structuri de date, manipula zone de memorie (tablouri, pointeri), aloca șiruri de caractere.

6.2 Noțiuni noi introduse

6.2.1 Citirea unui tablou de elemente

Un pattern de programare des întâlnit este acela în care citim un număr (de ex `n`) apoi citim *n* valori.

Deocamdată alocăm memorie static sau pe stivă, pentru tablou. Luăm o valoare maximă și validăm `n` să nu depășească această valoare.

Exemplu de citire a `n` valori:

Listing 6.1: Citire de *n* valori într-un tabel

```
#include <stdio.h>
#define MAX_ELEM 100
int main()
{
    int n, i;
    int tab[MAX_ELEM];
    printf("Introduceti valoarea lui n ");
    scanf("%d", &n);
    if ((n <= 0) || (n >= MAX_ELEM)){
        printf("Eroare, n trebuie sa fie intre 0 si 100.");
        printf(" Ati introdus %d.\nProgramul se termina", n);
        return -1;
    }

    for(i = 0; i < n; i++){
        printf("tab[%3d]:", i);
```

(continues on next page)

(continuare din pagina precedentă)

```
17          scanf("%d", &tab[i]);
18      }
19
20
21      // Optional, afisam ce am citit
22      printf("Tabelul este:\n");
23      for(i = 0; i < n; i++){
24          printf("tab[%2d] = %5d\n", i, tab[i]);
25      }
26
27      return 0;
28  }
```

Deosebit este la linia 2, am declarat o constantă simbolică. Deocamdată considerați că peste tot în cod unde apare construcția `MAX_ELEM` va fi înlocuită cu construcția `100`. Evităm astfel folosirea repetată a numărului 100.

La prima scriere, acest lucru pare mai laborios, tastăm mai mult. Dar atenție, dacă modificăm valoarea maximă stocată? Și uităm să modificăm peste tot? Apar bug-uri! Evitați la maxim să vă repetați! Mai ales când vine vorba de constante. Codul și constantele "magice" sunt o sursă mare de bug-uri.

Atenție la citirea din tabel, am folosit sintagma `&tab[i]`!

6.2.2 Tablouri în funcții.

Funcțiile pot primi un tablou de elemente ca variabilă de intrare. Parametrul formal se scrie ca `TIP NUME[]`. De obicei trebuie să trimitem și numărul de elemente:

Listing 6.2: Exemplu de funcție ce acceptă un tabel.

```
int CalculSuma(int tabel[], int nr_elemente){

}

// undeva in main

     rezultat = CalculSuma(tab, n);
```

Sau, alternativ, cu pointeri:

Listing 6.3: Exemplu de funcție ce acceptă un tabel.

```
int CalculSuma(int *tabel, int nr_elemente){

}

// undeva in main

     rezultat = CalculSuma(tab, n);
```

Folosirea lui `tab`/`tabel` în interiorul `CalculSuma` sau `main` este cvasiidentică între cele două exemple.

6.2.3 Efecte secundare

Din cauză că funcția primește direct zona de memorie a tabelului, aceasta POATE MODIFICA valorile! Mare atenție, să nu introduceți efecte secundare, dacă nu există astfel de cerințe.

6.3 Probleme propuse

6.3.1 Afișați un tabel de numere

Scrieți **funcții** care acceptă (1) un tabel și (2) afișează conținutul, element cu element. Va trebui sa concepeți două funcții, una pentru citire și alta pentru afișare.

Funcțiile ar trebui să accepte tabele alocate static sau dinamic. Demonstrați ambele posibilități.

Aflarea numărului de elemente și alocarea se fac în `main()`.

6.3.2 Calculul sumei unui șir de numere

Folosindu-vă de exemplul dat anterior, scrieți **o funcție** care să calculeze suma elementelor unui tabel. Afișați.

6.3.3 Efecte secundare

Scrieți o funcție care adaugă 10 la valoarea fiecărui element. Din `main`, apelați și funcțiile de citirea și afișarea. Afișați înainte și după modificare!

6.3.4 Evaluarea unui polinom II

Evaluați un polinom știind coeficienții și un punct x.

Citiți un număr `n` și `x`. Citiți `n` coeficienți. Calculați și afișați elegant.

Ex:

`2x^2 + 5x + 1 = 1, pentru x = 0`

Coeficienții și punctul `x` sunt numere reale. Alocați doar memoria necesară pentru coeficienți. Folosiți alocarea dinamică.

Atenție! Faceți funcții pentru fiecare etapă: Citire de polinom (și stocare într-o zonă prealocată), evaluare polinom (se vor da la funcție, zona de memorie unde sunt stocați coeficienții, numărul de coeficienți și valoarea lui `x`). Faceți funcție și pentru afișare.

Ca și la alte probleme care se continuă din laboratoarele anterioare, vă puteți folosi de codul de la laboratoarele/problemele trecute!

6.3.5 Produs cartezian

Se citesc doi vectori de întregi de la tastatură. Se afișează produsul cartezian a celor doi vectori. Exemplu: Vectorii

`1 2` și `7 8 9` au produsul cartezian: `(1 7) (1 8) (1 9) (2 7) (2 8) (2 9)`.

Ambii vectori se citesc cu o funcție. Funcția va primi numărul de elemente și zona prealocată. Va fi o singură funcție pentru ambele citiri! Aflarea numărului de elemente și alocarea, se poate face deocamdată cu cod dedicat pentru fiecare vector.

Produsul se calculează de o altă funcție. Acea funcție va putea afișa rezultatul direct pe ecran.

Ordinea operațiilor I/O:

- Citire n1, n2, numărul de elemente din cei doi vectori
- Citirea, pe rând, a elementelor celor doi vectori. Afișați un mesaj când se trece la următorul vector
- Afișați produsul cartezian.

Opțional, dacă modularizați și funcția de citire, puteți citi pe rând vectorul 1 (n1, elementele), și 2 (n2 și elementele)

Puncte bonus:

Faceți ca produsul cartezian să fie stocat într-un vector (va trebui să folosiți și structuri, pentru a stoca, pentru fiecare element, cele două valori ale produsului cartezian). Funcția de calcul va scrie rezultatul aici.

Faceți o funcție de afișare care e capabilă să primească un pointer la o zonă cu elemente de produs cartezian și să afișeze ce e acolo.

L7. Structuri. Șiruri de caractere

7.1 Recapitulare

7.1.1 Structuri

O structură se definește cu `struct` urmat opțional de numele structurii, apoi un bloc de declarații.

Pentru ușurința folosirii tipului structură, îl denumim:

Listing 7.1: Rezumat al operațiilor cu structuri

```
typedef struct {
   int a, b;
} Structura;

//Declarare de variabila, tabel, pointeri
Structura s;
Structura tab_struct[10];
Structura *tab_ptr_struct = (Structura*)malloc(sizeof(Structura) * 50000);

tab_struct[2].b = 1;
tab_ptr_struct->a = 44;
```

Acum, în interiorul unei funcții, putem declara variabile de acest tip: `Structura s`. Pentru acces la membri scriem `s.a`. Această expresie poate fi folosită peste tot unde putem pune o variabilă. Putem să stocăm ceva acolo, putem să îi luăm adresa (eg ca să o trimitem la `scanf`).

La declararea structurii se alocă memorie pentru toți membrii.

Putem declara tabele de structuri sau structuri care să conțină tabele.

Putem avea pointeri spre structuri. Și ei trebuie alocați. Pentru acces la elemente se folosește operatorul `->`.

7.1.2 Șiruri de caractere

Sunt o zonă de memorie cu o convenție specială, ultimul caracter din șir are pattern-ul zero: `'\0'`. Funcțiile care "procesează" șiruri de caractere se așteaptă la această convenție. De aceea, ele nu necesită să trimitem la input și câtă memorie este alocată acolo.

Putem stoca un șir de caractere într-un tabel sau o zonă alocată dinamic:

```
char sir[100];
```

```
char * str = (char*) malloc(sizeof(char) * 10000);
```

Putem folosi `sir` sau `str` direct în operații ce necesită tipul `char *` (eg funcții din `<string.h>`)

Listing 7.2: Exemplu de alocare statică (tabel) și citire de șir de caractere

```
#define MAX_STR_LEN 100

char sir[MAX_STR_LEN];
fgets(sir, MAX_STR_LEN, stdin);
```

Listing 7.3: Exemplu de alocare dinamică și citire.

```
#define MAX_STR_LEN 100000

char *sir = (char*)malloc(sizeof(char) * MAX_STR_LEN);
fgets(sir, MAX_STR_LEN, stdin);
```

Citiți din materialul de curs câteva funcții utile din `<string.h>`.

Atenție, lungimea unui șir de caractere se calculează până când se întâlnește valoarea 0. (*Null terminated string*)

Funcțiile de procesare a șirurilor de caractere se găsesc în `<string.h>`

Observație utiă în "parsarea" șirurilor. Funcția `sscanf` poate fi folosită pentru a "extrage" valori numerice sau cuvinte dintr-un șir de caractere deja aflat în memorie. Un comportament util al lui `sscanf` este că atunci când primește modificatorul `%s` va începe să acumuleze caractere până întâlnește un caracter alb. Comportament util când vrem să "despărțim" în cuvinte un șir (cuvintele fiind separate de *whitespaces*). Mai încolo veți învăța și despre `strtok`.

7.1.3 Citire din fișier

Funcția `fgets` citește până întâlește caracterul `newline` sau EOF. Acest lucru este util pentru a:

- Citi o linie
- Citi până când nu mai există nimic în fișier.

După citirea unei linii este bine a o procesa și/sau muta ceea ce s-a citit într-o altă zonă de memorie. Zona de memorie trimisă la `fgets` se poate refolosi. Valoarea maximă care o trimiteți la `fgets` include și octetul zero de la sfârșit.

Odată citită o linie, cu `fgets`, o putem "sparge" (*parsa*) în siguranță folosind `sscanf`.

Dacă știți că trebuie memorate `n` caractere alocați suficientă memorie pentru `n+1` caractere! (și pentru terminarea șirului de caractere).

Important: DIN ACEST MOMENT ESTE INTERZIS SĂ MAI FOLOSIȚI `scanf` SAU `fscanf` PENTRU A CITI ȘIRURI DE CARACTERE!!

Funcția `sscanf` se poate folosi însă, pentru că, avem control asupra lungimii a ceea ce se află în șirul de intrare.

7.2 Probleme propuse

7.2.1 Structuri simple

Creați, citiți și afișați structuri pentru a ține minte date despre un student: Nume (un cuvânt) și nota.

Numele îl păstrați într-un tabel de tip `char`, în interiorul structurii.

Citirea o faceți de la tastatură, **folosind fgets**!

7.2.2 Numere complexe

Definiți o structură care poate memora numere complexe.

Definiți câteva funcții care pot citi numere complexe (și returnează o structură), funcții cre pot afișa, care pot aduna, scădea, înmulți numere complexe

7.2.3 Operații cu numere complexe

Citiți dintr-un fișier text, un șir de numere complexe și efectuați operații cu ele. Numerele complexe se stochează în structuri și în tabele (dinamice) de structuri. Faceți funcții pentru diverse operații.

Într-un fișier avem un întreg `n` urmat pe următoarea linie de `n` **perechi** de `float`-uri. Fiecare pereche e un număr complex. Nu există separatori speciali, în afară de spatiu, între perechi sau elementele din pereche.

Scrieți o funcție care citește fișierul și returnează o structură care conține

- un tablou dinamic alocat (pointer) de tipul numărului complex:
- numărul de elemente (perechi) citite din fișier, adică câte elemente sunt alocate în pointer.
- Nu e necesar să returnați un pointer la structură.

Să se facă o funcție care acceptă structura de mai sus și o afișează frumos.

Să se calculeze suma numerelor complexe (folosind bineînțeles o funcție care acceptă structura de mai sus)

Să se calculeze și să se stocheze într-o altă structură, conjugata numerelor complexe citite. Să se afișeze.

Se vor folosi funcții, variabile locale, cod elegant, memorie alocată dinamic, etc.

Notă Problema este relativ complexă, atingând multe concepte. Vă recomand să o abordați iterativ. Nu vă opriți până când toate cerințele din laborator sunt îndeplinite. Puteți, ca primă iterație, să scrieți tot codul în `main()`, fără alocare dinamică, doar tabele, structuri si doar o singură funcționalitate (ex citire și afișare). Apoi, introduceți, în fiecare nouă iterație: Alocarea dinamică, funcțiile (pe rând), funcționalități noi, etc. până îndepliniți cerințele. Atenție, când credeți că totul e gata, mai faceți o iterație pentru curățat codul, verificat cazuri particulare sau cazuri mai complexe, verificarea respectării fiecărei cerințe, formatare cod, etc.

Succes!

L8. Operații cu date compuse I

8.1 Recapitulare

Revizuiți noțiunile introduse până acumm. Structuri, tabele, șiruri de caractere, alocarea dinamică a lor folosind pointerii, citirea din fișiere, instrucțiuni de control, expresii, variabile.

8.1.1 Șiruri de caractere

Putem stoca un șir de caractere, dinamic, într-un pointer: `char * str = (char*) malloc(sizeof(char) * 10000)`.

Dacă însă avem nevoie să stocăm mai multe șiruri (eg lista de cuvinte) putem folosi tabel de pointeri (Tabele care să memoreze adrese). Putem face asta static sau dinamic:

Listing 8.1: Liste de șiruri de caractere

```
// alocare statica
char* lista_sir[30];
// alocare dinamica
char** lista_sir_ptr = (char**)malloc(sizeof(char*) * 30);

// Folosirea este identică. Trebuie să alocăm fiecare element, înainte de folosire
lista_sir[3] = (char*)malloc(sizeof(char) * 1000);
lista_sir_ptr[3] = (char*)malloc(sizeof(char) * 1000);
strcpy(lista_sir_ptr[3], "Popescu are mere!");
fgets(lista_sir[3], 1000, stdin);
```

8.2 Probleme propuse

8.2.1 Litere mari - mici

Transformați din litere mici în litere mari un cuvânt. Atenție, literele mari și cifirele nu trebuie modificate

8.2.2 Număr palindorm

Folosind `sprintf` verificați dacă un număr este palindrom.

8.2.3 Citire de șiruri de caractere

Deschideți un fișier în modul text și determinați câte linii are.

8.2.4 Citire mod binar

Deschideți un fișier în modul binar și determinați frecvența de apariție a fiecărui octet.

Observații:

Știți din start câte valori posibile sunt pentru octeți (256).

Funcția `fread` returnează numărul de elemente citit. Util pentru "capătul" fișierului, când buffer-ul va fi umplut doar parțial.

Puteți procesa datele "on the fly" fără să le memorați.

8.2.5 Frecvență litere

Deschideți un fișier în mod text și numărați frecvența de apariție a fiecărei litere. Faceți asta fără să țineți seama de litere mici - litere mari.

Afișați ordonat alfabetic, caracterul și frecvența de apariție.

8.2.6 Propoziții și cuvinte

Citiți un text din fișier și despărțiți-l în cuvinte.

Se citește un text din fișier. Numărul de linii nu este precizat. (dar pot fi maxim 10000). Se stochează textul.

Se va "sparge" fiecare linie de text în cuvinte. Un cuvânt este un șir de caractere "non albe" despărțite de caractere albe: `spațiu`, `newline` sau `line-feed` (Restul separatorilor îi ignorați, adică îi considerați ca făcând parte din cuvinte.)

Se va afișa numărul total de linii și cuvinte.

Se va afișa, pe câte o linie pe ecran:

- Numărul liniei
- Numărul de cuvinte
- Linia de text

Se vor folosi funcții, variabile locale, cod elegant, memorie alocată dinamic, etc.

L9. Operații cu date compuse II

Vom sedimenta instrumentele dobândite până acum, rezolvând niște exemple mai complexe, care acoperă o arie mai largă de mecanisme din limbajul C.

Încercați să faceți codul cât mai "frumos" și "elegant", după ce reusiți să rezolvați cerințele.

9.1 Probleme propuse

9.1.1 Structuri cu pointeri

Creați, citiți și afișați date despre studenți Nume, Prenume (câte un cuvânt) și nota. Citirea se face din fișier.

Pe prima linie găsim numărul de studenți. Apoi, în fișier găsim:

- mai multe perechi de linii care definesc studentul:
- prima linie din pereche conține numele și prenumele
- a doua linie din pereche conține nota, ca întreg.

Pentru nume și prenume, folosiți un pointer la `char`. În fișier aveți numele și prenumele scris pe un rând. Aici trebuie păstrat separat. Presupuneți că nu sunt mai mult de două cuvinte pe linie. Atenție, după ce citiți, stocați strict câtă memorie e nevoie!

Păstrați datele într-un tabel de structuri.

Trebuie rezervată memoria dinamic, cu pointeri la structuri.

După ce ați închis fișierul și ați încărcat datele, faceți o statistică:

- Frecvența prenumelor (fiecare prenume, de câte ori apare)
- Frecvența pe note, afișată descrescător (eg. câți de 10, câți de 9, etc).
- Pentru fiecare notă, studenții care au acea notă.

9.1.2 Operații cu numere

Se dă un fișier cu mai multe linii. Pe prima linie din fisier este un număr N. Fiecare din cele N linii de după, conține numere întregi. Numărul de numere de pe fiecare linie este dat ca primul întreg de pe linia respectivă si variaza de la o linie la alta.

După ce se termină cele N linii, urmează un întreg M, pe o linie separată. După, vin M perechi, fiecare pereche pe linia ei. Perechile definesc operatii si au forma:

`operatie nr_linie numar_virgula_mobila`

Exemplu:

```
* 3 34
+ 4 99.5
+ 1 -1e-3
```

Operatiile sunt: adunare (+), inmultire (*), Operatiile se aplică pe fiecare număr din lista de linii definită de primul număr citit `nr_linie`. Operațiile sunt clare, fiecare element din rând se va aduna/înmulti cu valoarea citită dupa semnul operatiei. (ex pentru `* 3 34` se va înmulți fiecare element din linia a 4-a cu 34).

Să se citească fișierul și să se memoreze liniile de numere. Atenție fiecare linie poate avea număr diferit de elemente. Recomand un tabel de tabele, totul ținut într-o structură:

- pointer dublu la double[1]. Se stochează rândurile de date
- pointer la șir de întregi, se stochează câte elemente sunt pe fiecare linie
- numărul de linii

Să se aplice operațiile pe linii. Atenție, fiecare operație este independentă și nu afectează rezultatul altor operații

Să se afișeze rezultatul operațiilor.

Operațiile se pot procesa "on the fly" adică pe măsură ce se citesc din fișier.

Exemplu. In fisierul de intrare avem un sir de trei de 1 si o operatie de adunare:

```
1
3 1 1 1
1
+ 0 1
```

Exemplu de afișare:

```
Randul 0, operatie + 1 rezultat: 2 2 2
```

[1] *Pun intended*

L10. Structuri de date complexe I

10.1 Recapitulare

Pentru a aloca o matrice de elemente 2D putem apela la următoarea bucată de cod:

Listing 10.1: Alocare și dealocare matrice 2D de tip double

```
double** alocare_2d_double(int m, int n){
   double **matrice = (double**)malloc(sizeof(double*) * m);
   for(int i = 0; i < m; i++)
      matrice[i] = (double*)malloc(sizeof(double) * n);
   return matrice;
}

void dealocare_2d_double(double ** ptr, int m){
   for(int i = 0; i < m; i++)
      free(ptr[i]);
   free(ptr);
}
```

Putem grupa datele necesare unei matrici, într-o structură:

```
typedef struct{
   double ** data;
   int m, n;
} Matrice_double;
```

Deasemenea, putem avea matrici continue a căror alocare este mai simplă: `double *tab`. Se vor aloca `m * n` elemente: `tab = (double*) malloc(sizeof(double) * m * n);`

La accesare va trebui să calculăm adresa elementului dorit, la fiecare accesare. Exemplu: `tab[i * n + j] = 0`.

10.2 Probleme propuse

10.2.1 Alocare matrici în cele două formate

Într-un fișier găsiți o matrice specificată astfel: Pe prima linie, doi întregi, numărul de rânduri și numărul de coloane. Pe următoarele linii de text, rândurile matricii.

Implementați cele două moduri de a stoca matricile. Scrieți funcții care să aloce, citească, afișeze în cele două formate.

Atenție, grupați modul de stocare a matricii (eg în structură) cu modul de alocare (funcția de la curs, returnează un pointer). Faceți astfel încât funcția de alocare să returneze o structură!

Și pentru formatul continuu, puteți crea o structură de date!

10.2.2 Citire date binare

Deschideți în mod *read-only* și binar, fișiere de pe disc. Citiți primii 10-20 octeți și afișați primele caractere.

Vânați și identificați "constantele magice". Cum începe un fișier .exe? Dar .jpg?

10.2.3 Parsare de fișiere

Citiți dintr-un fișier o matrice. Matricea NU are specificat numărul de linii și coloane. Puteți presupune un maxim de 100 linii și 100 coloane.

Folosiți `fgets`, `sscanf` și valorile returnate de ele, pentru a vă da seama câte linii sunt și câte valori sunt pe fiecare linie.

10.2.4 Bubble sort

Citiți `n` numere întregi dintr-un fișier și sortați-le folosind *bubble sort* (Căutați)

10.2.5 Sortare cuvinte

Citiți cuvinte din fișier. Fiecare cuvânt este scris pe câte o linie.

Păstrați cuvintele într-un tablou de pointeri la char.

Sortați și afișați cuvintele alfabetic.

10.2.6 Instrumente de lucru cu matrici

Implementați câteva instrumente: alocare, citire, afișare, adunare, transpusa, dealocare pentru matrici 2D de tip `int` și `double`, stocate discontinuu (liste de vectori), și a căror informație este stocată în structuri.

Trebuie să aveți funcții pentru fiecare operație și tip de dată stocată. Funcțiile și structura se păstrează în alt **modul**. Din modulul principal demonstrați folosirea instrumentelor.

Operațiile de transformare (eg adunarea) vor returna alte matrici, cele inițiale rămânând intacte.

Atenție la denumirile funcțiilor. Vă recomand sa specificați și tipul elementelor în nume. Exemplu, `aduna_double()` ca nume de funcție care adună două matrici cu elemente de tip `double`.

Faceți-vă structura fișierului de intrare astfel încât să vă facă citirea ușoară (ex precizați de la început numărul de linii și coloane)

L11. Structuri complexe II

11.1 Recapitulare

Recitiți din curs, declararea de matrici multidimensionale, alocarea matricilor, stocarea informațiilor despre matrici în structuri. Stocarea matricilor în format continuu.

11.2 Probleme propuse

11.2.1 Array-uri 2D

Citiți dintr-un fișier o matrice. Matricea NU are specificat numărul de linii și coloane. Puteți însă presupune un maxim de 100 linii și 10000 de caractere pe rând. Sunt maxim 100 de coloane.

Știți că `fgets` va returna NULL când nu se mai pot citi linii.

Folosiți `fscanf`, `sscanf` și valorile returnate de ele, pentru a vă da seama câte linii și câte valori sunt pe fiecare linie.

Funcția `sscanf` acceptă ca modificator `%n` care returnează în variabila corespunzătoare, numărul de caractere citit. Funcția returnează ca valoare, câte variabile din input a putut consuma. Împreună cu operațiile pe pointeri, veți putea parsa numărul de elemente din fiecare linie.

Opțional, puteți studia și `strtok`.

După terminarea citirii, să rămână alocată strict câtă memorie e necesară stocării matricii.

Stocați matricea într-o structură care să conțină dimensiunea matricii și pointer-ul la tabloul de pointeri.

Implementați operații pe o astfel de structură: Alocarea, Afișarea, Duplicarea, Transpusa unei matrici.

11.2.2 Array-uri 2D. Operatii

Într-un fișier aveți două matrici de numere în virgulă mobilă. Fieare matrice e declarată astfel: Pe prima linie sunt 2 întregi reprezentând numărul de rânduri și coloane a matricii. Pe următoarele linii sunt rândurile matricii. A doua matrice începe imediat după prima matrice.

Scrieți funcții care: alocă, citesc o matrice, afișează o matrice, adună două matrici stocând rezultatul în a treia. Implementați funcțiile pentru (1) metoda de stocare a matricilor ca liste discontinue (2) metoda continuă de stocare a matricilor.

Nu este voie să aveți variabile globale. Cele două metode trebuie să coexiste în același program. Opțional puteți să faceți o conversie între cele două tipuri, sărind astfel funcția de citire.

Vă puteți folosi de cod deja implementat **de voi** în activitățile anterioare (recomandat, laboratorul 9).

L12. Recursivitate și pointeri la funcții

12.1 Recapitulare

12.1.1 Apel recursiv

O funcție se poate apela pe ea însăși. Variabilele locale se alocă la fiecare apel. Pentru a nu intra în ciclu infinit, este important ca primul lucru la intrarea în funcție, să se verifice condiția de oprire.

12.1.2 Variabile statice

Cuvântul cheie `static` face ca variabila să fie alocată în zona de memorie statică. Această variabilă își va păstra valoarea între două apeluri de funcție.

Mai jos se arată o astfel de funcție, care păstrează intern, date între apelări.

În biblioteca <strings.h> avem funcția `strtok`. Aceasta permite despărțirea, "spargerea" (*tokenizarea*) unui șir de caractere la nivelul unor separatori. De exemplu, un text spart în cuvinte, despărțite de spații.

```
char * strtok ( char * str, const char * delimiters );
```

La primul apel, se inițializează și se returnează adresa primului token. Următoarele apeluri așteaptă un pointer NULL pentru `str` și va returna pe rând următoarele tokenuri. După ce s-au returnat toate tokenurile, se returnează NULL.

Funcția stochează intern, între apeluri, într-un câmp static, ultima locație returnată. Așa va ști să continue căutarea din ultimul loc returnat.

Listing 12.1: Exemplu complet de tokenizare

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

#define MAX_CHARS 1000

int main()
{
    char *sir = (char*)malloc(sizeof(char) * MAX_CHARS);
    const char* punctuatie = " ,.";
    char *subsir;

```

(continues on next page)

(continuare din pagina precedentă)

```
    fgets(sir, 100, stdin);

    subsir = strtok(sir, punctuatie);
    while (subsir != NULL) {
    printf("Cuvant: %s\n", subsir);
        subsir=strtok(NULL, punctuatie);
    }
}
```

12.1.3 Pointeri la funcții

Codul unei funcții trăiește în memorie la o anumită adresă. Această adresă poate fi manipulată cu variabile.

Câteva exemple când folosim astfel de construcții:

1) Interacțiunea cu sistemul de operare, când vrem să fim anunțați de anumite evenimente
2) Apelarea de algoritmi interni bibliotecii standard.

Aici dăm un exemplu pentru (2). În biblioteca standard, <stdlib.h> avem funcția `qsort`:

```
void qsort (void* base, size_t num, size_t size, int (*compar)(const void*,const void*));
```

Observați `compar` este un pointer la o funcție ce acceptă doi pointeri `void` și returnează un întreg.

Citiți cu atenție exemplul și documentația de pe internet .

12.2 Probleme propuse

12.2.1 Progresie aritmetică

Scrieți cod care să dea valorile funției $f(x) = f(x-1)+5$, $f(0) = 0$. Rezolvați analitic problema $f(x) = 5 * (x-1)$ și verificați că, pentru mai multe valori ale lui x, calculul algoritmic se face corect.

Rezolvați problema de laborator, recursiv!

12.2.2 Seria Fibonacci

Găsiți valorile lui $f(x_n) = f(x_{n-1}) + f(x_{n-2})$ unde $f(0) = 0$ și $f(1) = 1$.

Rezolvați problema:

- Folosind recursivitatea
- Calculând totul de la 0 la *n*
- Folosind formula analitică de mai jos.

Analitic, termenul din progresia aritmetică se poate calcula:

$f(x_n) = \lceil \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \rceil$

unde

$\phi = \frac{1+\sqrt{5}}{2}$ este rația de aur.

Pentru ultima variantă, folosiți biblioteca matematică pentru rotunjirea rezultatului final. Verificați comparând rezultatele ultimelor două metode de rezolvare.

12.2.3 Fibonacci reinstanțiat

Implementați recursiv funcția lui Fibonacci. Modificați funcția astfel încât să "numere" intern câte apelări au fost. Terminați calculul dacă se depășesc un anumit număr de apeluri (și eventual semnalizați rezultatul incorect)

12.2.4 Produs cartezian II

Citiți (din fișier) câteva rânduri de cifre întregi. Se dă și numărul de linii și numărul de elemente de pe fiecare linie.

Calculați produsul cartezian între toți vectorii. Implementați recursiv. Se va afișa fiecare tuplă generată.

Exemplu:

```
3
2 1 2
2 3 4
1 5
```

Adică, la intrare sunt 3 vectori, primii doi vectori au câte 2 elemente iar ultimul un element. Ar trebui afișat:

```
(1 3 5)
(1 4 5)
(2 3 5)
(2 4 5)
```

Formatați elegant textul astfel încât valorile să fie aliniate.

12.2.5 Căutări rapide în șiruri sortate

Fie șirul de 16 de numere sortate:

1, 3, 6, 7, 8, 9, 11, 13, 17, 19, 21, 22, 23, 24, 29, 31

Citiți un număr de la tastatură și căutați dacă numărul există sau nu în acest șir.

Scrieți o funcție care acceptă așa:

- șirul
- două numere reprezentând indecsi ai: capătului din stânga (inclusiv) și ai capătului din dreapta (exclusiv)
- numărul căutat.

Ca să căutați, luați o secțiune din șir și găsiți care ar fi poziția de mijloc. Vedeți dacă valoarea căutată este egală cu ceea ce se află în mijloc. Dacă da, ați terminat. Dacă valoarea din șir e mai mare sau mai mică decât cea căutată, apelați recursiv la stânga sau la dreapta valorii centrale. Dacă intervalul curent este vid (indexul capătului din stânga este mai mare sau egal cu indexul capătului din dreapta), atunci numărul nu se află în șir. Dacă numărul căutat este pe capetele intervalului se declară numărul găsit.

Exemplu: Caut valoarea 7.

Indecșii capătului sunt 0 și 16. Primul apel al funcției compară valoarea de mijloc, sir[8] == 17 cu 7. Este mai mare. Apelul se va duce înspre stânga. Adică, se va apela recursiv cu indecșii capătului 0 și 8.

Iarăși se va lua elementul de mijloc, sir[4] == 8. Este mai mare decât 7. Se va merge tot în stânga: Apel recursiv pe 0, 4. Acum se va compara elementul sir[2] == 6. Este mai mic. Se va merge în dreapta. Apel recursiv de 2, 4. Se ia mijlocul, sir[3] = 7. Este egal cu numărul căutat.

Implementați și versiunea nerecursivă, clasică. Comparați metodele, căutând pentru toate numerele de la 0 la 32.

12.2.6 Tokenizare

Citiți un fișier de text și despărțiți textul în cuvinte. Memorați fiecare cuvânt din fișier într-un tabel. Afișați cuvintele, în ordinea citirii, după ce ați închis fișierul. Puteți presupune că nu sunt mai mult de 1000 cuvinte în fișier.

Alocați dinamic memoria pentru cuvinte! Doar strictul necesar!

12.2.7 Sortați un șir de numere

Citiți un șir de numere dintr-un fișier text. (Se poate specifica numărul de numere, pe prima linie din fișier).

Sortați folosind `qsort`

12.2.8 Sortați un tablou de structuri

Funcția de `callback` poate fi complicată. Trebuie să înțelegeți cum "circulă" pointerii la datele dvs din `qsort` în `callback`.

Reluați exemplul cu citirea unei liste de studenți din fișier. În fișier găsiți numărul de studenți (pe prima linie) urmat de *n* perechi de linii, nume și nota.

Păstrați datele într-un tabel dinamic de structuri (pointer spre structuri). Numele studenților pot să fie stocate inițial în tabele statice (`char nume[100]`)

Sortați, folosind qsort:

- Alfabetic, după nume
- După notă.

Atenție, trebuie să implementați două funcții de `callback`, câte una pentru fiecare tip de comparație.

Folosiți debug-ul pentru a înțelege și verifica dacă convertiți bine pointerii.

12.2.9 Sortați o listă de cuvinte

Citiți un fișier de text și despărțiți textul în cuvinte. Memorați fiecare cuvânt unic din fișier. Numărați de câte ori apare fiecare cuvânt. Închideți fișierul. Afișați cuvintele, în ordinea primei apariții. Sortați apoi cuvintele, în ordinea descrescătoare a numărului de apariții. Afișați cuvintele și numărul de apariții.

Puteți presupune că nu sunt mai mult de 1000 cuvinte în fișier.

Sortați folosind `qsort`. Creați-vă fișierul sau luați un text mai scurt de pe Wikipedia.

Folosiți structuri pentru a ține minte cuvintele. Căutarea după cuvinte existente NU trebuie să fie optimă. Un algoritm ce verifică cuvânt cu cuvânt în lista de cuvinte deja existentă, este suficient.

Bibliografie

1. Ignat I., C.L. Ignat, "Programarea calculatoarelor – Descrierea algoritmilor și fundamentele limbajului C/C++", Ed. Albastră, Cluj-Napoca, 2005
2. Liviu Negrescu, "Limbajele C şi C++ pentru începători. Limbajul C", Editura Albastră, Cluj-Napoca, 1994