Lucrarea 1

Obiective

1. Recapitularea noțiunilor de bază privind folosirea limbajului C
2. Recapitulare limbajul C: pointeri

Noțiuni teoretice

Noțiuni elementare despre algoritmi și reprezentarea lor

Etapele necesare rezolvării unei probleme cu ajutorul calculatorului sunt următoarele:

1. Analiza problemei
2. Proiectarea programului
3. Implementarea programului
4. Întocmirea documentației programului
5. Întreținerea programului

În **etapa de analiză a problemei** se determină conținutul clar și precis al problemei ce trebuie rezolvată cu ajutorul calculatorului, altfel spus se enunță ce trebuie să facă programul. De asemenea în această etapă se determină datele de intrare și datele de ieșire, respectiv organizarea datelor pe suportul extern (structura datelor în fișiere și aspectul interfeței cu utilizatorul).

Pentru **proiectarea programului** este elaborat un algoritm care să realizeze funcția programului. Dacă problema rezolvată cu ajutorul calculatorului este complexă, atunci se împarte problema în subprobleme și pentru fiecare subproblemă se determină specificațiile subproblemei (date de intrare/ieșire, funcția subproblemei) și algoritmul de rezolvare. Logica de rezolvare este specificată de regulă folosind pseudocod sau diagrama logică. Ca și tehnici de proiectare a structurii ierarhice a modulelor componente pentru un program complex există tehnica "top down" și tehnica "bottom up". De obicei o problemă în care rezolvarea pleacă de la zero se folosește tehnica top-down. Probleme a căror rezolvare implică reutilizarea unor module folosite la rezolvarea unor alte probleme (programe existente) pot folosi tehnica bottom-up sau o combinație între cele două.

Definiție Algoritm:

O secvență finită de operații, ordonată și complet definită, care pornind de la date de intrare produce într un timp finit date de ieșire.

În **etapa de implementare a programului** sunt realizate următoarele acțiuni:

- Codificarea fiecărui modul într-un limbaj de programare;
- Obținerea cu ajutorul unui editor de texte a programului sursă;
- Translatarea programului sursă în program obiect și eliminarea erorilor sintactice (în C compilare);
- Trecerea programului obiect în program imagine memorie translatabilă și eliminarea erorilor (în C linkeditare);
- Încărcarea în memoria internă a programului imagine memorie translatabilă, lansarea și execuția, testarea și eliminarea erorilor de logică.

Este foarte important să se facă diferența între un translator și compilator. În cazul unui Translator, o instrucțiune este executată (interpretată) imediat ce a fost scrisă cu ajutorul editorului. Ca exemplu poate fi menționat aici interpretorul pentru limbaj SQL existent la Sistemele de Gestiune de Baze de Date (SGBD) Relaționale. În particular în cazul SGBD Oracle există *sqlplus* unde se poate executa un script SQL ce conține o secvență de instrucțiuni SQL ce sunt tratate una câte una.

În cazul limbajului C vom folosi la lucrările de laborator IDE Codeblocks cu ajutorul căruia vom edita codul sursă al unui program, iar pentru a executa (Run) programul va fi nevoie de compilare (Compile) și linkeditare (Build). Se poate verifica ușor acest lucru urmărind pas cu pas ce fișiere sunt create în folderul proiectului. Astfel, în urma compilării se crează *main.o*, iar în urma linkeditării se obține fișierul *nume_proiect.exe* (în foldere specifice).

Revenind la importanța faptului că un program C necesită compilare, dacă facem modificări în codul sursă SE RECOMANDĂ SĂ EFECTUĂM DIN NOU TOȚI PAȘII (ori facem Compile și Build ori facem Rebuild). În caz contrar riscăm să introducem confuzie în executarea programului (la execuție programul face altceva decât ne-am aștepta).

Întocmirea documentației programului este o activitate importantă care are importanță corelat cu activitatea de întreținere a programului. De multe ori activitatea de întreținere este realizată de altă persoană decât programatorul care a scris inițial programul și existența unei documentații oferă productivitate.

Întreținerea programului este necesară deoarece lumea înconjurătoare este în permanentă schimbare, ceea ce poate conduce la modificarea ipotezelor inițiale când s-a făcut analiza problemei sau pur și simplu apar funcționalități noi.

Tipuri de date simple și structurate

Tipul unei date stabilește mulțimea de valori pe care le poate lua și operațiile permise asupra valorilor sale. Datele pot fi elementare (primitive, simple) sau structurate.

În limbajul C există următoarele tipuri elementare de date: char, short, int, long, long long, float si double.

Observație, în C nu există tipul boolean (true/false).

Cel mai comun exemplu de tip de dată structurat este *tabloul*. În limbajul C un caz particular de tablou este șirul de caractere. De xemplu:

```
char sir[21];
```

definește un șir de caractere ce poate avea lungimea maximă 20 de caractere. Explicația pentru caracterul în plus de la definiție este caracterul terminanator de șir '\0'.

Accesul la un element al tabloului se face cu ajutorul unei variabile indexate. De exemplu primul caracter din șir este sir[0] (în limbajul C indexul este între 0 și n-1, unde n este dimensiunea șirului). Costul din punct de vedere al complexității unui algoritm, de acces la elementul unui tablou când se cunoaște indexul elementului, este 1 (acces direct). Spre deosebire de accesul direct la elementul unui tablou prin index, se va vedea la liste înlănțuite, costul de acces depinde de poziția elementului în cadrul listei (acces secvențial).

Tabloul bidimensional trebuie văzut în limbajul C ca un vector de vectori. De exemplu:

```
int mat[5][5];
```

reprezintă definiția unei matrici pătratice de 5x5. De fapt există vectorul de linii cu 5 elemente, fiecare element fiind la rândul lui un vector de 5 numere întregi. Primul element al matricei este mat[0][0], iar ultimul element este mat[4][4].

Un alt tip de dată compus este *înregistrarea*. În limbajul C , se folosește cuvântul cheie *struct*.

Expresii şi instrucţiuni

Definiție expresie: un operand sau mai mulți operanzi legați prin operatori.
Un **operand** poate fi:

- o constantă;
- o constantă simbolică;
- numele unei variabile simple;
- numele unui tablou;
- numele unei structuri;
- numele unui tip;
- o variabilă indexată;
- numele unei funcții;
- referință la elementul unei structuri;
- apelul unei funcții;
- o expresie inclusă în paranteze rotunde.

Operatorii sunt de următoarele tipuri:

- operatori aritmetici:

```
operatori unari: +, -
operatori binari multiplicativi: *, /, %
operatori binari aditivi: +, -
```

- operatori relaționali: <, <=, >, >=
- operatori de egalitate: = =, !=
- operatori logici: !, &&, ||
- operatori logici pe biți: ~, <<, >>, &, ^, |
- operatori de atribuire: =, /=, *=, %=, +=, --=, <<=, >>=, &=, ^=,
- operatori de incrementare: ++,
- operatorul de forțare tip: (tip)
- operatorul dimensiune: sizeof
- operatorul adresă : &
- operatori paranteză : ( ), [ ]
- operatori condiționali: ?, :
- operatorul virgulă : ,
- operatorul de dereferențiere: *
- operatorul de acces la componenta unei structuri: . , ->

Prioritățile operatorilor pot fi observate în tabelul urmator.

Prioritățile	Operatori
1	() [] • ->
2	+(unar) -(unar) &(unar) *(unar) ++ -- (tip) sizeof ! ~
3	*(binar) / %
4	+(binar) -(binar)
5	<< >>
6	< <= > >=
7	== !=
8	&(binar)
9	^
10	|
11	&&
12	||
13	? :
14	= <<= >>= += -= *= /= %= &= ^= |=
15	,

Asociativitatea operatorilor este de la stânga la dreapta, excepție fac operatorii unari, condiționali și de atribuire, care se asociază de la dreapta la stânga.

Regula conversiilor implicite:

- dacă un operator binar se aplică la operanzi de același tip, atunci rezultatul va avea tipul comun al operanzilor;
- dacă un operator binar se aplică la doi operanzi de tipuri diferite, atunci operandul de tip inferior se convertește implicit spre tipul superior al celuilalt operand, iar rezultatul va avea tipul superior

Ordinea descrescătoare a priorității tipurilor:

- long double (prioritate maximă);
- double;
- float;
- unsigned long;
- long;
- int (prioritate minimă).

Cu alte cuvinte, dacă într-o expresie operanzii sunt de mai multe tipuri, se convertește implicit la tipul cel mai prioritar. În exemplul cu împărțire reală s-a convertit unul din operanzi explicit la tipul *float* și în continuare celălalt operand având tipul *int*, de prioritate inferioară, s-a convertit mai întâi la *float* și apoi s-a efectuat efectiv împărțirea.

Instrucțiunea expresie are o importanță specială deoarece permite exprimarea concisă a unei condiții fără a necesita o structură de control decizie. O expresie devine instrucțiune la apariția caracterului ; .

Exemplu: c = a > b ? a : b; // c primește max(a,b)

Echivalentul ar fi: if (a > b) c = a; else c = b;

Instrucțiunea compusă (sau bloc de instrucțiuni) este scrisă în limbajul C folosind parantezele acoladă ({…}) de obicei fie la decizie fie la ciclu, unde implicit ar exista o instrucțiune simplă, dar

poate fi scrisă oriunde într-o secvență. Recomandarea este să se folosească acoladele chiar și atunci când există o singură instrucțiune pentru claritate și pentru a evita erori la dezvoltarea ulterioară a codului. Într-un bloc de instrucțiuni este permisă orice declarație de variabile.

Instrucțiunea if (decizia)

- forma 1:

```
if ( expresie )
  instrucțiune
```

- forma 2:

```
if ( expresie )
  instrucțiune_1
else
  instrucțiune_2
```

Expresia instrucțiunii if este evaluată și dacă este true (diferită de zero) atunci se execută instrucțiunea de pe ramura if. Dacă este false (egală cu zero) atunci se execută instrucțiunea de pe ramura else (forma 2). Se recomandă să se acorde atenție la folosirea operatorului de comparație == deoarece este frecventă greșeala folosirii unui singur simbol = (atribuire!!!) și atunci se va modifica accidental conținutul variabilei comparate și se va executa totdeauna instrucțiunea de pe ramura if. De asemenea la existența mai multor instrucțiuni if imbricate, se recomandă neapărat folosirea parantezelor acoladă.

Instrucțiunea switch (decizia multiplă)

```
switch ( expresie )
{
  case C1: șir_instrucțiuni_1;
  break;
  case C2: șir_instrucțiuni_2;
  break;
  ….....................................
  case Cn: șir_instrucțiuni_n;
  break;
  default: șir_instrucțiuni
}
```

Instrucțiunea *switch* este folosită pentru decizie multiplă (mai mult de două cazuri true/false) și va exista câte o ramură pentru fiecare valoare posibilă a expresiei, plus ramura *default* (care este opțională). Se recomandă folosirea instrucțiunii *break* la fiecare ramură, pentru a evita execuția instrucțiunilor de pe mai multe ramuri. Se mai poate termina execuția instrucțiunii *switch* cu *return* sau apelul funcției *exit*(cod).

Instrucțiunile repetitive (ciclu) sunt de două tipuri principale: cu număr cunoscut de pași (*for*) respectiv când numărul de pași nu se cunoaște există două instrucțiuni repetitive, cu test inițial (*while*) sau cu test final (*do .. while*).

Instrucțiunea for (ciclu cu număr cunoscut de pași):

```
for ( expr1; expr2; expr3 )
  instrucțiune;
```

Oricare dintre cele trei expresii pot să lipsească. Dacă lipsesc toate trei atunci bucla este infinită. Prima expresie este inițializarea unei variabile contor. A doua expresie este o expresie logică ce

verifică numărul de pași cât se repetă instrucțiunea. A treia expresie este incrementarea/decrementarea contorului. Cea mai folosită instrucțiune for este pentru parcurgerea a n elemente ale unui tablou:

```
for(i = 0; i < n; i++)
        instrucțiune;
```

Instrucțiunea while (ciclu cu test inițial):

```
while ( expresie )
        instrucțiune;
```

Expresia este evaluată și dacă este diferită de zero se execută instrucțiunea, după care se re-evaluează expresia. Acest proces continuă până când expresia devine zero. Rezultă că atunci când se dorește o buclă infinită expresia poate fi înlocuită cu valoarea 1 (while(1) ...). Dacă instrucțiunea este compusă, se poate folosi *break* pentru a forța ieșirea din buclă sau se poate folosi *continue* pentru a reitera. Următoarea instrucțiune while este echivalentă cu o instrucțiune for:

```
expr1;
while ( expr2 ) {
        instrucțiune;
        expr3;
}
```

Instrucțiunea do while (ciclu cu test final):

```
do
        instrucțiune
while ( expresie );
```

Spre deosebire de ciclu cu test inițial, unde există șansa ca instrucțiunea să nu se execute niciodată, la ciclul cu test final instrucțiunea se execută cel puțin o dată. Bucla se termină când expresia devine zero (false). În practică este mai puțin folosită decât celelalte două instrucțiuni repetitive, dar există cazuri când este mai avantajoasă instrucțiunea *do .. while* decât *for* sau *while*.

Pointeri

Un **pointer** este o variabilă care are ca valori adrese de memorie, deci este un număr întreg și la rândul lui ocupă în memorie 4 B sau 32 biți. Dacă pointerul p are ca valoare adresa de memorie a variabilei x, se spune că p pointează spre x.

```
        tip *nume;
```

Exemplu:

```
        int *p;
```

Operatorul referențiere (&)

Exemplu:

```
        int x = 2000;
        int *p;
        p = &x;
        /* presupunem că x este stocat în memorie la adresa α și presupunem că p este stocat
        în memorie la adresa β și are valoarea α */
```

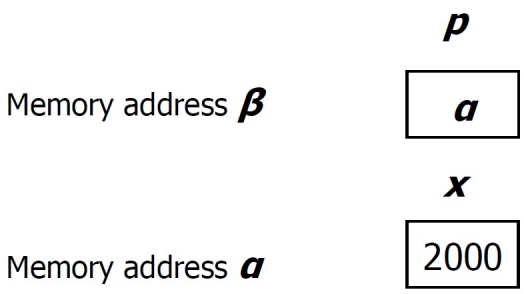

Figura 2-1 Exemplu pointer, referențiere/dereferențiere

Operatorul dereferențiere (*)

Conform exemplului de mai sus și figurii 2-1, dacă p conține α atunci *p este valoarea 2000.

Alte exemple:

```
int x, y;
int *p;
```

1. Instrucțiunea x = y; este echivalentă cu:

```
p = &x; *p = y; sau
p = &y; x = *p;
```

2. Instrucțiunea x++; este echivalentă cu:

```
p = &x; (*p)++;
```

Pointer generic

```
void *nume;
```

Este folositor în practică pentru a schimba tipul de dată stocat într-o variabilă, la execuție. Prin natura limbajului C o variabilă dacă primește un tip prin declarare, variabila va fi de acel tip până la terminarea programului. Cu ajutorul pointerului generic se poate „jongla" cu tipul de dată asociat unei variabile. Un exemplu care folosește pointer generic este funcția *malloc*() despre care se va vorbi mai jos în această lucrare.

Exemplu:

```
int x;
float y;
void *p;
p = &x;
*(int *)p = 10;
p = &y;
*(float *)p = 2.57;
```

În exemplul precedent variabila p este un pointer generic. Mai întâi pointează către o valoare întreagă și mai apoi pointează către o valoare reală.

Legătura între numele unui tablou și un pointer are următoarele consecințe:

1. numele unui tablou are drept valoare adresa primului său element,
2. numele unui tablou este un pointer constant, neputând fi modificat în timpul execuției.

Exemplu:

```
int tab[100];
int *p;
int x;
...
```

```
p = tab;
/* p primește ca valoare adresa elementului tab[0] */
```

În exemplul de mai sus, x = tab[0]; este echivalent cu x = *p;

Operații cu pointeri

1. Incrementare/decrementare cu 1: p++

Valoarea pointerului este incrementată/decrementată cu numărul de octeți (B) necesari pentru a păstra o dată de tipul de care este legat pointerul.

Exemplu:

```
int tab[100];
int *p;
…
p = &tab[10];
p++;
/* Valoarea lui p este incrementată cu 4 B, având adresa elementului tab[11] */
```

2. Adunarea și scăderea unui întreg dintr-un pointer: p ± n

Are drept efect creșterea, respectiv scăderea valorii pointerului p cu n*numărul de octeți necesari pentru a păstra o dată de tipul de care este legat pointerul.

Exemplu:

```
int tab[100];
int *p;
…
p = tab;
x = *(p+i); /* echivalent cu x = tab[i]; sau x = *(tab+i); */
```

3. Diferența a doi pointeri

Dacă doi pointeri p și q pointează spre elementele i și j ale aceluiași tablou tab (j > i), adică p = &tab[i] și q = &tab[j], atunci (q – p) este egal cu (j - i) înmulțit cu numărul de octeți necesari pentru a pastra o dată de tipul de bază al tabloului.

De exemplu dacă tabloul este un șir de caractere, diferența+1 ar reprezenta lungimea subșirului (i .. j).

4. Compararea a doi pointeri

Se folosesc operatorii relaționali: >, <, >=, <=, ==, !=.

Alocarea/eliberarea dinamică a memoriei heap

Tipurile de variabile cu care se lucrează în limbajul C sunt următoarele:

1. *globale* și *statice* ce folosesc alocare statică (atenție la suprapunerea numelui variabilelor, ce introduce efecte laterale în program) adică adresa variabilei rămâne aceeași pe tot parcursul programului;
2. *automatice* (*locale*) alocare dinamică pe *memoria stivă* (valoarea se pierde la terminarea funcției).

Memoria heap este diferită de *memoria stivă* („stack” în limba engleză) și este gestionată special prin următoarele funcții ce necesită biblioteca stdlib.h:

```
void *malloc (unsigned n);
void *calloc(unsigned nr_elem, unsigned dim);
```

void free (void *p);

Fucția *malloc*() alocă în heap o zonă contiguă de n octeți de memorie și returnează un pointer generic către zona de memorie alocată. Deoarece pointerul returnat este de tip generic, se recomandă conversia explicită de tip către tipul de dată dorit. Se evită astfel erori neplăcute ce vor fi observate doar la execuție, erori ce se manifestă prin mesaj de genul „acces la memorie inexistentă" și terminarea bruscă a programului.

Funcția *calloc*() alocă în heap o zonă contiguă de nr_elem * dim în octeți. Și această funcție returnează în caz de succes adresa de început a zonei de memorie alocate (pointer generic, vezi observațiile de mai sus).

Ambele funcții returnează în caz de insucces 0 (zero, pointerul NULL), atunci când nu a fost posibilă alocarea de memorie (lucru ce se poate întâmpla prin alocări/dealocări repetate și fragmentarea memoriei heap).

Probleme

1. Să se scrie un program pentru generarea tuturor numerelor prime mai mici sau egale cu un număr natural n.

2. De pe mediul de intrare se citeşte un număr natural n. Să se verifice dacă numărul respectiv este palindrom.

3. Să se scrie un program pentru efectuarea operaţiilor de adunare, scădere, înmulţire şi împărţire între două polinoame:

   $A(x)=a_0 +a_1x^1+.....+ a_nx^n$

   $B(x)=b_0 +b_1x^1+.....+ b_mx^m$

   Gradele şi coeficienţii reali ai polinoamelor se citesc de pe mediul de intrare.

4. Folosind numai pointeri şi expresii cu pointeri se vor scrie funcţii de sortare a unui vector cu elemente reale.

5. Folosind numai pointeri şi expresii cu pointeri se vor scrie funcţii de citire, afişare şi înmulţire a două matrici.