Laborator 2
Pointeri

1. Tablouri de pointeri şi pointeri la tablouri

Pointerii fiind variabile, pot fi folosiţi pentru a forma alte tipuri de date compuse. Spre exemplu se pot forma tablouri de pointeri. Sintaxa generală de utilizare este :

*Tip *tablou[dim] ;*

Exemple:

1. Declaraţia *char *s[25];* reprezintă un tablou de 25 pointeri la caracter.
2. Sortarea unor şiruri de caractere:

```
#include<iostream.h>
#include<string.h>
void sort_lines(char *tp[], int n)
{
int i, sort=0;
char *temp;
while (!sort)
{
sort=1;
for(i=0;i<n-1;i++)
        if(strcmp(tp[i],tp[i+1])>0)
        {
        strcpy(temp,tp[i]); strcpy(tp[i],tp[i+1]);strcpy(tp[i+1],temp);
    sort=0;
        }
}
}
main()
{
int i;
char *sir[]=({“manual”},{“carte”},{“mare”},{“12345”},{“4542”});
sort_lines(sir,5);
for(i=0;i<5;;i++)
        cout<<sir[i]<<’ ‘;
cout<<endl;
}
```

Pentru a arăta că folosim pointeri la tablouri (şi nu tablouri la pointeri) avem pentru declaraţie sintaxa următoare:

*Tip (*p) [dim] ;*

Diferenţa dintre pointerii la tablouri de un tip şi tablourile de pointeri la acelaşi tip este modul de stabilire a unităţii de deplasare.

În acest sens, sunt concludente următoarele exemple:

3. În definiţia

a) *int (*x) [50];*

x este un pointer la un tablou de 50 întregi.

b) *int* x[50];*

x este un tablou de 50 pointeri la întregi.

Se observă diferenţa de semnificaţie care are loc o dată cu folosirea parantezelor.

4. Fie declaraţia

*short (*n) [10];*

În acest exemplu ***n*** reprezintă adresa unui tablou de 10 întregi. Chiar dacă acest lucru este iniţial sinonim cu adresa de început a tabloului, diferenţa este esenţială: ***aritmetica*** acestora cu numerele întregi. Astfel, incrementarea lui ***n*** va avea ca efect deplasarea cu 10 întregi de tip short (deci saltul este de 20 de octeţi), pe când în cazul unui pointer la întreg deplasarea ar fi fost de 2 octeţi.

Astfel, în programul:

/* deplasarea pentru un tablou la pointer */

```
#include<iostream.h>
void main()
{
        short *a, (*b) [10], c[10];
        a=c;
        b=c;
        cout<<"\nInainte de incrementare:\n Adresa pointer la intreg:<<a<<"\t Adresa pointer la tablou de intregi: "<<b;
        a++;
        b++;
        cout<<"\nDupa incrementare:\n Adresa pointer la intreg:"<<a<<"\t Adresa pointer la tablou de intregi: "<<b;
}
```

S-au obţinut rezultatele:

Înainte de incrementare:

Adresa pointer la întreg: FFE2 Adresa pointer la tablou de întregi: FFE2

După incrementare:

Adresa pointer la întreg: FFE4 Adresa pointer la tablou de întregi: FFF6

5. Declaraţiile următoare sunt corecte:

void matr(int *a[], int n); /* a este un tablou de pointeri */

void matr(int **a, int n);

În ambele antete, primul parametru poate fi folosit pentru matrici de dimensiune variabilă.

2. Tratarea tablourilor bidimensionale folosind pointeri

Numele unui tablou bidimensional, la fel ca numele unui tablou unidimensional, are ca valoare *adresa primului element al tabloului*.

Fie, de exemplu, declaraţia:

tip tab[...][...] ;

Atunci ***tab*** are *ca valoare adresa lui tab[0][0]*.

Dacă în cazul tablourilor unidimensionale, *tab+n* este adresa elementului tab[n], adică &tab[n], în cazul tablourilor bidimensionale, expresia *tab+n* este *adresa elementului tab[n][0]*.

Această interpretare rezultă din faptul că un tablou bidimensional trebuie considerat ca fiind un tablou de tablouri unidimensionale. În general, un tablou k-dimensional este un tablou de tablouri k-1 dimensionale.

Fie, de exemplu, declaraţia: *tip tab[m][n];*

Tabloul *tab* poate fi privit ca un tablou de tablouri unidimensionale. Astfel, *tab[0], tab[1], ..., tab[m-1]* sunt cele *m* elemente ale lui *tab* şi fiecare este un pointer spre câte un tablou unidimensional de *n* elemente. De aici rezultă că *tab[0]* are *ca valoare adresa lui tab[0][0]*, *tab[1]* are *ca valoare adresa lui tab[1][0]*, iar ***în general tab[i] are ca valoare adresa lui tab[i][0]***. Cum ****(tab+i) are ca valoare chiar tab[i]***, rezultă că ***tab[i][0] are aceeaşi valoare cu *(*(tab+i))***.

De asemenea, *tab[i]+j* are *ca valoare adresa lui tab[i][j]*. Deci tab[i][j] are aceeaşi valoare ca şi expresia **(tab[i]+j),* iar aceasta din urmă are aceeaşi valoare cu expresia **(*(tab+i)+j).*

Înseamnă că atribuirile **(*(tab+i)+j)=x* si *tab[i][j]=x* sunt echivalente.

Dacă *tab* este un tablou unidimensional declarat prin: *tip tab[...];* atunci *tab are tipul tip* adică este pointer spre tip.*

Dacă *tab* este un tablou bidimensional declarat prin *tip tab[m][n];* atunci *tab are tipul tip (*)[n];* adică *pointer spre un tablou de n elemente de tipul tip.*

Menţionez că în declaraţia de mai sus *m* şi *n* sunt expresii constante.

Să presupunem că tab este parametru la apelul funcţiei f: …f(tab). În acest caz, parametrul formal al funcţiei f poate fi declarat în următoarele moduri:*…f(tip t[][n]) sau …f(tip(*p)[n])*.

Fie declaraţia: *tip *tab1[n]*;

În acest caz, *tab1 este un tablou unidimensional de pointeri spre tipul tip*, adică fiecare element din cele n ale tabloului tab1 este un pointer spre *tip*. *Tablourile tab şi tab1 nu trebuie confundate.* Tabloului *tab* i se alocă o memorie de *m*n*sizeof(tip)* octeţi. Tabloului *tab1* i se alocă *n*sizeof(tip*)* octeţi.

Dacă o funcţie *q* are la apel ca parametru pe tabloul *tab1:...q(tab1)* atunci parametrul formal al funcţiei *q* poate fi declarat în următoarele moduri:*…q(tip *p[])* sau *…q(tip **p).*

Exemple:

1. Fie declaraţiile:

```
double tab[2][3]={{0,1,2},{3,4,5}}
double *p;
```

Atunci avem:

INSTRUCŢIUNI	ELEMENTUL AFIŞAT	VALOAREA ELEMENTULUI
p=tab[0]; printf(”%g”,*p) ;	tab[0][0]	Zero
p=tab[1]; printf(”%g”,*p) ;	tab[1][0]	3
printf(”%g”,*(*(tab))) ;	tab[0][0]	Zero
printf(”%g”,*(*(tab+1)+2)) ;	tab[1][2]	5

1. Considerăm funcţia *f* definită astfel:

```
void f (double (*p)[3])
{
int i,j;
for (i=0;i<2;i++)
        for(j=0;j<3;j++)
                printf(“%g ”,p[i][j]);
}
```

La apelul *f(tab)*; unde tab este tabloul definit la exemplul anterior, se afişează valorile elementelor lui tab separate de câte un spaţiu:
0 1 2 3 4 5.
Expresia *p[i][j]* poate fi înlocuită cu **(*(p+i)+j)*.

2. Fie declaraţiile:

```
    double *t[2];
    double t0[3]={10,11,12};
    double t1[3]={13,14,15};
```

şi atribuirile **t[0]=t0; t[1]=t1;**
Definim funcţia *f1* astfel:

```
void f1(double *p[])
{
…
}
```

unde corpul funcţiei *f1* coincide cu al funcţiei *f*.

La apelul funcţiei *f1(t)*; se listează valorile elementelor tablourilor t0 şi t1, separate prin câte un spaţiu, adică: 10 11 12 13 14 15

Acelaşi rezultat se obţine dacă antetul lui f1 se schimbă cu: *void f1(double **p)*.

În ambele situaţii, expresia p[i][j] poate fi schimbată cu **(*(p+i)+j)*.

Probleme rezolvare cu siruri de caractere si pointeri

Problema 1

Scrieti un program C++ care citeste de la tastatura doua siruri de caractere de maximum 100 de litere mici si verifica utilizand apeluri ale functiei **aparitii** daca cele doua siruri sunt anagrame (contin aceleasi litere, dar in ordine diferita). Se cere afisarea mesajului **anagrame** in caz afirmativ si a mesajului **nu sunt anagrame** in caz contrar.
Exemplu: Pentru sirurile adrian si nairda se afiseaza anagrame.

```
#include<iostream.h>
#include<string.h>
#include<fstream.h>
int main()
{
	char s[101], t[101],c,*p; int k,x,ok=1;
	cout<<"Introduceti primul sir: "; cin.get(s,101);
	cin.get();
	cout<<"Introduceti al doilea sir: "; cin.get(t,101);
	if(strlen(s)!=strlen(t)) ok=0; //au lungimi diferite
	else
	{
		for(c='a';c<='z' && ok==1;c++) //fiecare litera din alfabet
		{
			x=k=0;
			p=strchr(s,c);
			while(p!=0) //se numara de cate ori apare litera c in sirul s
			{
				x++; p=strchr(p+1,c);
			}
			p=strchr(t,c);
			while(p!=0) //se numara de cate ori apare litera c in sirul t
			{
				k++; p=strchr(p+1,c);
			}
			if(x!=k) ok=0; //daca numarul de aparitii difera
		}
	}
	if(ok==1) cout<<"anagrame";
	else cout<<"nu sunt anagrame";
	return 0;
}
```

Problema 2

Se citeste de la tastatura un text format din cuvinte separate intre ele prin cate un singur spatiu. Fiecare cuvant are cel mult 40 de caractere, doar litere mici ale alfabetului englez. Textul are cel mult 200 de caractere. Sa se scrie un program C++ care sa afiseze pe ecran, pe linii separate, doar cuvintele din textul citit care contin cel mult trei vocale. Se considera vocale: **a, e, i, o, u**.

Exemplu:

Pentru textul:

pentru examenul de programarea calculatoarelor se folosesc fisiere

se afiseaza:

pentru
de
se
folosesc

```
#include<iostream.h>
#include<string.h>
#include<fstream.h>
int main()
{
        char s[201], *p, *q, v[]="aeiou"; int i,k;
        cout<<"Introduceti textul:";
        cin.get(s,201);
        p=strtok(s," "); //primul cuvant
        while(p!=0)
        {
                k=0; //se numara vocalele din p
                for(i=0; p[i]!=0; i++)
                        if(strchr(v,p[i])!=0) k++;
                if(k<=3) cout<<p<<endl;
                p=strtok(NULL," "); //urmatorul cuvant din text
        }
        return 0;
}
```

Problema 3

Sa se scrie un program C++ care citeste de la tastatura un cuvant de cel mult 20 de litere mici ale alfabetului englez si care sa afiseze pe ecran, pe linii diferite, cuvintele obtinute prin stergerea succesiva a vocalelor in ordinea alfabetica a lor (**a, e, i, o, u**). La fiecare pas se vor sterge toate aparitiile din cuvant ale unei vocale.
Exemplu:
daca se citeste cuvantul **programare** se va afisa:
progrmre (s-au sters cele doua aparitii ale vocalei **a**)
progrmr (s sters unica aparitie ale vocalei **e**)
prgrmr (s sters unica aparitie ale vocalei **o**)

```
#include<iostream.h>
#include<string.h>
#include<fstream.h>
int main()
{
        char s[21], v[]="aeiou",*p; int i=0;
        cout<<"Introduceti cuvantul: "; cin.get(s,21);
        for(i=0;v[i]!=0;i++) //se parcurge multimea vocalelor
        {
                p=strchr(s,v[i]); //vocale v[i] apare in text
                if(p!=0)
                {
                        while(p!=0) //se sterg toate aparitiile
                        {
                                strcpy(p,p+1); p=strchr(s,v[i]);
                        }
                        cout<<s<<endl; //se afiseaza sirul obtinut
                }
        }
        return 0;
}
```

Probleme propuse spre rezolvare

1. Sa se ruleze toate exemplele prezentate mai sus.

Algoritmi de căutare

1. Căutare secvențială
2. Căutare binară
3. Căutare prin interpolare

1. Căutare secvențială

Să presupunem că dorim să determinăm dacă un element aparţine sau nu unui vector de elemente.
Dacă nu ştim nimic despre elementele vectorului avem soluţia căutării secvenţiale, până găsim elementul căutat sau până testăm toate elementele.
Algoritm descris în pseudocod

```
găsit ←0
pentru i← 0,n-1 execută
        dacă x=v[i] atunci
                găsit ←1
        sfârşit dacă
sfârşit pentru
```

Implementare C++:

```
int Caută_Secvential(int x, int v[], int n)
{
        for(int i=0; i<n; i++)
                if(x==v[i])      return 1;
        return 0;
}
```

Implementare C++:

```
int Caută_Secvential_Rapid(int x,int v[],int n)
{
        int i=0;
        while ( ( v[i]!=x ) && ( i<n ) )
                i++;
        if(i<n) return 1;
        else return 0;
}
```

2. Căutare binară

Dacă elementele vectorului sunt ordonate crescător, putem să ne dăm seama dacă elementul nu există în vector fără a fi nevoie să parcurgem toate elementele vectorului.

Unul dintre algoritmii folosiţi în acest caz este algoritmul de căutare binară.

Acest algoritm are la bază principiul înjumătăţirii repetate a domeniului în care se caută elementul, prin împărţirea vectorului în doi subvectori.
Notăm cu st primul indice al vectorului şi cu dr ultimul indice al vectorului, iar m este indicele elementului din mijloc al vectorului m=(st+dr)/2. Se compară valoarea căutată cu valoarea elementului din mijloc. Dacă cele două valori sunt egale înseamnă că s-a găsit elementul.
Dacă nu sunt egale vectorul v-a fi împărţit în doi subvectori.

Laborator 2

Operaţia de căutare constă în identificarea subvectorului în care se poate găsi elementul, prin compararea valorii căutate cu cea din mijloc, după care se divizează acest subvector în doi subvectori ş.a.m.d. până când se găseşte elementul, sau până când nu se mai poate face împarţirea în subvectori, ceea ce înseamnă că nu s-a găsit elementul.

Algoritm descris în pseudocod

```
Funcţia CautareBinara(n, A, x)
st←1  dr←n
cât timp st≤dr execută
        m ← [(st+dr)/2]
        dacă x=A[m] atunci
                CautareBinara ← m
                st ← dr+1
        altfel
                dacă st<A[m] atunci dr ← m-1
                altfel st ← m+1
        sfârşitdacă
 sfârşitdacă
sfârşitcât timp
sfârşitCautareBinara
```

Implementare C++:

```
int CautareBinara(int n, int a[], int x)
{
        int st, dr, m;
        st = 0; dr = n - 1;
        while(st <= dr)
        {
                m = (st + dr) / 2;
                if (x == a[m])
                {
                        return m;
                        st = dr + 1;
                }
        }
}
```

3. Căutare prin interpolare

Este similară cu căutarea binară, dar foloseşte o altă formulă pentru calculul lui m şi anume:
m=st+(x-v[st])*(dr-st)/(v[dr]-v[st]) ceea ce conduce la o delimitare mai rapidă a zonei din tablou în care s-ar putea găsi x. Ca principiu, metoda este inspirată după procedeul căutarii într-o carte de telefon. Aplicarea căutării prin interpolare necesită ca elementul de căutat, x, să se afle în interiorul intervalului v[1],...,v[n], astfel apare riscul ca valoarea calculată a lui m sa depaşească n.

Algoritm descris în pseudocod

```
Functia CăutareInterpolare(V,n,x)
st ← 0 dr ← n-1  gasit ← fals
dacă ((x<=v[dr]) şi (x>=v[st])) execută
        repetă
                m ← st +(x-v[st])*[(dr-st)/(v[dr]-v[st])]
                dacă x ≥ v[m] atunci
                        st ← m+ 1
                        altfel dr ← m-1
                sfârşit dacă
        până când ( (v[m]≠x) şi (st<dr)şi (v[st]=v[dr]) şi (x≥v[st]) şi (x≤v[dr]) )
sfârşit dacă
dacă v[m]=x atunci  gasit ← adevarat
sfârşit dacă
sfârşit functie
```

Implementare în C++

```
int CăutareInterpolare(int  v[], int n, int x)
{
        int st,dr,m;
        st=0;
        dr=n-1;
        if ((x<=v[dr]) && (x>=v[st]))
        do
        {
                m=st+(x-v[st])*(dr-st)/(v[dr]-v[st]);
                if(x>v[m])  st=m+1;
                else dr=m-1;
        } while((v[m]!=x) && (st<dr) && (v[st]==v[dr]) && (x>=v[st]) && (x<=v[dr]));
if(v[m]==x)
        return 1;
else
        return 0;
}
```

Această metodă este eficientă în cazul în care n este foarte mare şi valorile elementelor tabloului au o distribuţie uniformă în intervalul v[1],...,v[n].