Tehnici de programare

declaraţii complexe; pointeri la funcţii

Uneori declaraţiile C care includ pointeri, vectori şi funcţii pot deveni destul de complexe. Pentru citirea şi alcătuirea lor există o regulă simplă, numită **regula dreapta-stânga**, care ţine cont de faptul că, în cadrul unei declaraţii, modificatorii de tip din dreapta simbolului declarat au precedenţa mai mare decât cei din stânga.

Regula dreapta-stânga se poate enunţa astfel:

1. Se începe citirea de la simbolul declarat (ex: numele variabilei)
2. Atâta timp cât există modificatori de tip în dreapta simbolului (vectori, funcţii), se citesc aceştia în ordine de la stânga la dreapta
3. După ce s-au terminat modificatorii de tip din dreapta, se vor citi cei din stânga (pointeri, tip de bază) în ordine, de la stânga către începutul declaraţiei
4. Dacă există paranteze ce modifică ordinea de citire a modificatorilor de tip, se citeşte prima oară doar ceea ce este între paranteze, iar apoi se iese din paranteze, repetând paşii (2) şi (3), până când s-au citit toate componentele tipului

Exemplu: Ce tip de date are variabila *v* din declaraţia următoare: "*struct Punct *(*v)[10][3]*"?

Aplicând *regula dreapta-stânga*, vom parcurge următoarele etape:

- Pornim de la simbolul declarat: *v*
- Citim toţi modificatorii de tip din dreapta: în dreapta lui *v* se află doar paranteza închisă, deci deocamdată nu avem ce citi
- Citim toţi modificatorii de tip din stânga către început: avem "*", deci citim *"pointer la"*
- Deocamdată nu mai avem modificatori în stânga, din cauza parantezei deschise, dar încă nu am terminat de citit toată declaraţia, astfel încât continuăm citirea
- Deoarece am citit tot ceea ce se afla în paranteze, ieşim din ele şi repetăm citirea dreapta-stânga
- În dreapta, de la stânga la dreapta avem 2 vectori, deci citim *"o matrice de 10 linii şi 3 coloane, fiecare element având tipul"*
- În stânga, de la stânga către început, avem: "*", deci citim *"pointer la"* şi tipul de bază *"struct Punct"*
- Citirea se încheie, deoarece am citit toate componentele tipului. Tipul lui v complet se poate citi astfel: *"v este un pointer la o matrice de 10 linii şi 3 coloane, fiecare element din matrice având tipul pointer la struct Punct"*.
  Dacă structura *Punct* are membrul *x* de tip *int*, o posibilă folosire a lui *v* este : *"int k=(*v)[i][j]->x;"*

Pointeri la funcţii

În limbajul C putem declara pointeri la funcţii, dacă dorim să stocăm adresa funcţiei într-o variabilă sau să o transmitem ca argument unei alte funcţii. Pointerii la funcţii se declară astfel:

*tip_bază (*ptr_fn)(argumente_funcţie);*

Parantezele din jurul lui *"*ptr_fn"* sunt necesare deoarece, aşa cum s-a discutat mai sus, modificatorii de tip din dreapta au precedenţa mai mare decât cei din stânga. Dacă ar fi lipsit parantezele, deci declaraţia ar fi fost *"tip_bază *ptr_fn(argumente_funcţie)"*, *ptr_fn* ar fi fost o funcţie care ar fi returnat un pointer la *tip_bază*.

La argumentele funcției se poate da doar tipul lor, fără a se mai scrie și numele, acesta nefiind relevant. Numele se poate da, dacă dorim ca declarația să fie mai descriptivă.

Când se preia adresa unei funcții, se folosește operatorul **&** (adresă), iar când se folosește pointerul la funcție, acesta se dereferențiază pentru a obține funcția pointată, ca în exemplul de mai jos:

```
#include <stdio.h>

int f1(int a,int b)
{
   return a+b;
}

int f2(int a,int b)
{
   return a-b;
}

int main(void)
{
   int a=7,b=5;
   int (*pf)(int,int);       // pf - pointer la o functie cu doi parametri de tip int, care returneaza o valoare de tip int
   pf=&f1;                   // se seteaza pf cu adresa functiei f1
   printf("op(%d,%d)=>%d\n",a,b,(*pf)(a,b));          // se apelează functia pointata de pf cu argumentele a si b
   pf=&f2;
   printf("op(%d,%d)=>%d\n",a,b,(*pf)(a,b));
   return 0;
}
```

La apelul funcției pointată de pointerul *pf* s-au folosit paranteze în jurul pointerului, *"(*pf)"*, deoarece apelul de funcției are precedență mai mare decât dereferențierea. Dacă nu s-ar fi pus parantezele, ar fi însemnat că se apelează funcția, iar apoi se dereferențiază rezultatul returnat de aceasta (un *int*).

Pointerii la funcții sunt compatibili doar cu funcții care au exact același tip de parametri și valoare returnată ca cele declarate în pointer. De exemplu, nu putem folosi un pointer la o funcție cu 2 parametri pentru a apela o funcție cu 3 parametri sau un pointer la o funcție care returnează *int* ca să returnăm o valoare de tip *double*.

Limbajul C permite preluarea directă a adresei unei funcții, fără folosirea operatorului & și, analogic, apelul funcției pointate fără a mai fi necesară dereferențierea pointerului. Funcția *main* din exemplul de mai sus devine:

```
int main(void)
{
   int a=7,b=5;
   int (*pf)(int,int);  // pf este un pointer la o functie cu doi parametrii de tip int, care returneaza o valoare de tip int
   pf=f1;               // se seteaza pf cu adresa functiei f1
   printf("op(%d,%d)=>%d\n",a,b,pf(a,b));   // se apelează functia pointata de pf cu argumentele a si b
   pf=f2;
   printf("op(%d,%d)=>%d\n",a,b,pf(a,b));
   return 0;
}
```

Pointerii la funcții se folosesc în multe situații, cum ar fi: algoritmi generici, colecții eterogene, implementarea acțiunilor din interfețele grafice utilizator (GUI - Graphics User Interface), etc. Vom discuta în continuare doar primele două dintre aceste situații.

Algoritmi generici

Una dintre aplicaţiile pointerilor la funcţii o constituie implementarea *algoritmilor generici*. De exemplu, testarea dacă toate elementele unui vector îndeplinesc o anumită condiţie, presupune iterarea vectorului şi testarea fiecărui element conform condiţiei date. Dacă dorim să testăm diverse condiţii, partea de iterare rămâne identică şi se modifică doar condiţia de testat. Am putea defini un algoritm generic de testare, care să accepte diferite condiţii, ca în exemplul următor:

```
#include <stdio.h>

int pozitiv(int e)
{
   return e>=0;
}

int par(int e)
{
   return e%2==0;
}

// testeaza daca toate elementele din vectorul v, de dimensiune n, indeplinesc conditia cond
int testare(int *v,int n,int(*cond)(int))
{
   int i;
   for(i=0;i<n;i++){
      if(!cond(v[i]))return 0;
      }
   return 1;
}

int main(void)
{
   int v[5]={4,8,1,2,0};
   printf("toate elementele sunt pozitive: %d\n",testare(v,5,pozitiv));
   printf("toate elementele sunt pare: %d\n",testare(v,5,par));
   return 0;
}
```

Funcţia *testare* iterează vectorul dat şi returnează 1 dacă toate elementele îndeplinesc condiţia dată, altfel returnează 0. Condiţia este dată prin intermediul unui pointer la o funcţie, care primeşte un element şi îl testează, returnând 1 sau 0, dacă acel element îndeplineşte sau nu condiţia dată. Se constată în *main* că, apelând *testare* cu diverse condiţii, putem refolosi această funcţie pentru diverse situaţii, trebuind să rescriem doar condiţia de testat. Funcţiile care returnează valori logice, se mai numesc şi *predicate*, astfel încât în exemplul de mai sus avem predicatele *pozitiv* şi *par*.

Pentru simplificarea declaraţiilor care conţin pointeri la funcţii, putem folosi *typedef* ca să dăm nume specifice acestor pointeri. Pentru aceasta, înlocuim numele variabilei pointer la funcţie cu numele de tip dorit. Folosind *typedef*, funcţia *testare* de mai sus devine:

```
typedef int(*Conditie)(int);

// testeaza daca toate elementele din vectorul v, de dimensiune n, indeplinesc conditia cond
int testare(int *v,int n,Conditie cond)
{
```

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

Funcţia qsort

Pentru algoritmi simpli nu este neapărat o îmbunătăţire folosirea pointerilor la funcţii dar, dacă algoritmul este foarte complex, conţinând sute sau chiar mii de linii de cod, atunci implementarea unei variante generice a sa, care permite refolosirea unei mari părţi din algoritm, reprezintă o certă îmbunătăţire. În biblioteca standard C există doi algoritmi generici, *qsort (*quick sort) şi *bsearch* (binary search), ambii declaraţi în antetul *stdlib.h*.

Funcţia *qsort* este o implementare performantă a algoritmului quick sort şi poate fi folosită pentru sortarea oricăror tipuri de vectori. *qsort* este declarat astfel:

```
void qsort(void *vector, size_t nElemente, size_t dimensiuneElement,
           int (*compar)(const void *ptrElement1,const void *ptrElement2));
```

qsort are următorii parametri:

- *vector* - un vector de elemente
- *nElemente* - numărul elementelor din vector
- *dimensiuneElement* - dimensiunea unui element din vector, exprimată în octeţi
- *compar* - o funcţie care primeşte pointeri la două elemente din vector (transfer prin adresă). Parametrii sunt de tipul "*const void **", adică pointeri generici la valori constante. Funcţia *compar* va fi apelată de *qsort* cu perechi de elemente, primul element fiind în vector în stânga celui de al doilea. *compar* trebuie să compare elementele şi să returneze un întreg cu următoarele semnificaţii:
    - <0 - ordinea elementelor este corectă, deci vor fi lăsate de *qsort* la poziţiile lor prezente
    - 0 - elementele sunt egale, deci ordinea nu contează
    - >0 - ordinea este incorectă, deci elementele trebuie inversate

Pointerii la valori de tip **const** specifică faptul că nu se poate modifica valoarea pointată. Pointerul în sine poate fi modificat, de exemplu să fie folosit ca iterator. Declaraţiile devin mai descriptive, iar compilatorului i se permite să facă anumite optimizări.
Câteva proprietăţi ale pointerilor la valori constante sunt redate în următoarea secvenţă de program:

```
char v[2]={'a', 'b'};
const char *p;          // pointer la valori de tip char constante
p=v;                    // corect, se modifica pointerul
p++;                    // corect, se modifica pointerul
printf("%c",*p);        // corect, valoarea doar se citeste, fara a fi modificata
*p='7';                 // gresit, se incearca modificarea unei valori constante
```

Exemplu qsort: Se dă un vector de puncte în plan, având coordonatele (x,y) de tipul *double*. Se cere să se sorteze acest vector în ordinea distanţelor punctelor faţă de origine.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

typedef struct{
    double x,y;
    }Pt;
```

```
double dist(const Pt *pt)                    // distanta fata de origine
{
   return sqrt(pt->x*pt->x+pt->y*pt->y);
}

// deoarece qsort transmite functiei de comparare adresele elementelor, functia va primi pointeri la elemente
// in acest caz, deoarece vectorul are elemente de tipul Pt, functia va primi parametri de tipul Pt*
int cmpDist(const void *elem1,const void *elem2)
{
   const Pt *p1=(const Pt*)elem1;
   const Pt *p2=(const Pt*)elem2;
   double d1=dist(p1);
   double d2=dist(p2);
   if(d1<d2)return -1;
   if(d1>d2)return 1;
   return 0;
}

int main(void)
{
   Pt puncte[5]={{1,3},{7,5},{0,0},{-2,4},{1,1}};
   int i,n=5;
   qsort(puncte,n,sizeof(Pt),cmpDist);
   for(i=0;i<n;i++){
      printf("%g, %g\n",puncte[i].x,puncte[i].y);
      }
   return 0;
}
```

Notă: în Linux, la compilarea programelor care conţin funcţii din *math.h* (*sqrt*, *cos*, …), trebuie specifică în linia de comandă opţiunea ”-lm”, ceea ce înseamnă adăugarea (link) a funcţiilor matematice. Comanda va fi de forma:

gcc -lm -Wall -o prg prg.c

Aplicaţia 3.1: Se cere un număr *n* şi apoi un vector de *n* rezultate ale studenţilor la un examen. Fiecare rezultat este definit prin *(nume,nota)*. Se cere să se sorteze folosind *qsort* vectorul în ordinea notelor, notele cele mai mari fiind primele. Dacă două note sunt identice, rezultatele respective se vor sorta în ordinea alfabetică a numelor.

Colecţii eterogene

În anumite situaţii este necesar ca elementele unei mulţimi să aibă tipuri diferite, ca de exemplu: figurile geometrice dintr-un program CAD, personajele unui joc, produsele unui magazin virtual, etc. Aşa cum s-a discutat în laboratoarele anterioare, folosirea structurilor cu selectoare de tip şi uniuni pentru comasarea mai multor tipuri de conţinut oferă o modalitate de implementare a colecţiilor eterogene. Pointerii la funcţii oferă o a doua modalitate.

Exemplu: Avem de memorat mai multe figuri geometrice. Pentru o figură implementăm două operaţii: *aria* şi *perimetrul*. Folosind o structură care comasează toate tipurile de figuri, un fragment de program arată astfel:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
```

```c
#define PI      3.14159

void *alocare(size_t nOcteti)
{
    void *p=malloc(nOcteti);
    if(!p){
        printf("memorie insuficienta\n");
        exit(EXIT_FAILURE);
        }
    return p;
}

typedef enum{TFCerc,TFTriunghi,TFDreptunghi}TipFigura;
typedef struct{
    TipFigura tip;
    union{
        struct{
            double r;       // raza
            }cerc;
        struct{
            double a,b,c;   // lungimile laturilor
            }tr;
        struct{
            double latime,inaltime;
            }dr;
        };
    }Figura;

// analogic pentru arie
double perimetru(const Figura *f)
{
    switch(f->tip){
        case TFCerc:return 2*PI*f->cerc.r;
        case TFTriunghi:return f->tr.a+f->tr.b+f->tr.c;
        case TFDreptunghi:return 2*(f->dr.latime+f->dr.inaltime);
        }
}

// analogic pentru Triunghi_new, Dreptunghi_new
Figura *Cerc_new(double r)
{
    Figura *f=(Figura*)alocare(sizeof(Figura));

    f->tip=TFCerc;
    f->cerc.r=r;
    return f;
}

int main(void)
{
    Figura *figuri[100];        // vector de pointeri la figuri alocate dinamic
    int i,nFiguri=0;
    // introducere figuri
    figuri[nFiguri++]=Cerc_new(7.3);
```

```
    figuri[nFiguri++]=Triunghi_new(3,4,5);
    // afisare perimetre
    for(i=0;i<nFiguri;i++){
        printf("%g\n",perimetru(figuri[i]));
        }
    return 0;
}
```

Pentru fiecare tip de figură avem o funcție specifică pentru crearea ei, iar apoi, după ce figurile au fost stocate în colecția eterogenă *figuri*, vor fi folosite funcții care pot discerne între diverse tipuri de figuri.

Acest tip de implementare este simplu și în marea majoritate a cazurilor este suficient. Printre dezavantajele sale sunt următoarele:

- Orice element are dimensiunea elementului maximal ca memorie ocupată, ceea ce duce la risipă de memorie. În exemplul de mai sus, elementul maximal ca memorie ocupată este un triunghi, cu 3 atribute de tip *double*, pe când un cerc are doar 1 atribut. În acest caz, fiecare cerc are alocată în plus memorie pentru încă două variabile de tip *double*, memorie care este irosită. În special când sunt multe elemente și între ele sunt mari diferențe de dimensiuni, risipa de memorie poate deveni semnificativă.
- Dacă nu avem acces la codul sursă, astfel încât să putem modifica tipurile de date implicate și funcțiile care operează în mod eterogen pe ele (ex: *arie* și *perimetru*), atunci este imposibil să adăugăm noi tipuri de elemente. Această situație apare în cazul programelor care acceptă module (plugins, add-ons), pe care le încarcă la execuție. Multe programe cunoscute (ex: Photoshop, Chrome, programe de CAD, etc) acceptă module. De exemplu, Photoshop poate încărca module care definesc noi tipuri de filtre grafice. În această situație, implementarea colecțiilor eterogene folosind structuri cu tipuri comasate, nu mai este posibilă.

Dezavantajele de mai sus se pot elimina implementând colecțiile eterogene cu pointeri la funcții. Acest gen de implementare este folosit și de limbajele orientate pe obiecte (LOO) pentru implementarea **metodelor polimorfice.** Aceste metode sunt declarate într-o clasă de bază iar apoi primesc implementări specifice în clasele derivate. La execuție, codul va determina automat ce clasă are obiectul care apelează metoda respectivă și va apela implementarea ei specifică pentru acea clasă. De exemplu, metoda polimorfică *arie* din clasa *Figura* va fi implementată diferit în clasele derivate *Cerc* și *Triunghi.* La execuție, codul va determina automat dacă un element *Figura* este un *Cerc* sau un *Triunghi* și va apela metoda *arie* corespunzătoare.

Metodele polimorfice poartă diverse nume (ex: metode virtuale în C++ și C#, metode non-finale în Java, etc), dar mecanismul lor de implementare este asemănător și aseamănă destul de mult cu ce vom prezenta în continuare.

Mecanismul de implementare al colecțiilor eterogene este următorul:

- Vom stabili care sunt funcțiile polimorfice, adică funcțiile care primesc ca parametri elemente ce pot avea tipuri diferite (ex: *arie*, *perimetru*). Aceste funcții, pe lângă alți parametri, trebuie primească și elementul asupra căruia acționează (acesta este obiectul *this* din LOO).
- Funcțiile pe care le-am definit mai sus vor fi implementate separat pentru fiecare tip de element. De exemplu, pentru *arie* vom avea 3 funcții, câte una pentru aria fiecărui tip de figură geometrică (*Cerc*, *Triunghi*, *Dreptunghi*).
- Vom defini o structură care conține doar pointeri la toate funcțiile polimofice. Această structură se numește de obicei *VTable* (virtual table), *Dispatch* (dynamic dispatch table), etc. Pentru fiecare tip de element (ex: *Cerc*), pointerii din *VTable*-ul acelui tip vor pointa la funcțiile care implementează funcționalitățile specifice tipului.
  Pentru exemplu de mai sus, *VTable* va conține 2 pointeri la funcții, pentru calculul ariei și perimetrului.
- Vom considera o structură inițială (de bază), care conține un pointer la *VTable, numit vTable*. Această structură are rolul clasei de bază din ierarhia de obiecte a LOO.
- Pentru fiecare tip de element vom declara o structură care conține ca prim membru structura inițială și după ea atributele specifice acelui tip (ex: raza pentru *Cerc*). Vom implementa toate funcțiile din *VTable*

pentru tipul nou. Totodată vom implementa o funcţie de creare a unui obiect de acel tip (constructorii din LOO).

- Cu aceste structuri de date, apelul unei funcţii polimorfice pentru un element eterogen este simplă: pentru un anumit element, vom folosi chiar *vTable*-ul stocat în acel element. Din *vTable* folosim pointerul la funcţia dorită (ex: *aria*), care de fapt va pointa la funcţia ce implementează *aria* pentru tipul respectiv. Apelăm funcţia, transmiţându-i totodată şi elementul curent, pentru ca ea să aibă acces la câmpurile elementului.
- Deoarece structurile care implementează tipurile de elemente pot avea dimensiuni diferite, colecţia eterogenă va trebui neapărat să fie implementată ca pointeri la structuri, iar structurile în sine vor fi alocate dinamic de funcţiile de creere.

Pentru exemplul de mai sus cu figurile geometrice, implementarea cu pointeri la funcţii arată în felul următor:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define PI      3.14159

void *alocare(size_t nOcteti)
{
   void *p=malloc(nOcteti);
   if(!p){
      printf("memorie insuficienta\n");
      exit(EXIT_FAILURE);
      }
   return p;
}

// predeclararea structurii Figura, deoarece ea este folosita in VTable inainte de a fi definita
struct _Figura;
typedef struct _Figura Figura;

typedef struct{
   double (*arie)(Figura *f);              // pointer la functia care va calcula aria elementului f
   double (*perimetru)(Figura *f);       // pointer la functia care va calcula perimetrul elementului f
   }VTable;

struct _Figura{          // structura initiala din care vor fi derivate toate tipurile de elemente
   VTable *vTable;      // pointer la vTable-urile specifice fiecarui tip de figura
   };

typedef struct{
    Figura figura;       // figura trebuie sa fie pe prima pozitie in tipurile derivate din Figura, pentru a se putea folosi
campul vTable indiferent de tipul elementului (in orice tip, vTable va fi la offsetul 0 fata de inceputul structurii)
   double r;          // elemente specifice unui cerc
   }Cerc;

// functiile care implementeaza functiile polimorfice pentru tipul Cerc
double Cerc_arie(Figura *f)
{
   Cerc *c=(Cerc*)f;
   return PI*c->r*c->r;
}
```

```
double Cerc_perimetru(Figura *f)
{
   Cerc *c=(Cerc*)f;
   return 2*PI*c->r;
}

// variabila care contine pointeri la implementarile functiilor polimorfice pentru Cerc
VTable vTable_cerc={Cerc_arie, Cerc_perimetru};

Cerc *Cerc_new(double r)
{
   Cerc *c=(Cerc*)alocare(sizeof(Cerc));
   c->figura.vTable=&vTable_cerc;      // seteaza pointerul vTable la structura cu functiile specifice pentru Cerc
   c->r=r;
   return c;
}

// analogic se implementeaza triunghiul si dreptunghiul

int main(void)
{
   Figura *figuri[100];       // vector de pointeri la figuri alocate dinamic
   int i,nFiguri=0;
   // introducere figuri
   figuri[nFiguri++]=(Figura*)Cerc_new(1);
   figuri[nFiguri++]=(Figura*)Triunghi_new(3,4,5);
   // afisare perimetre
   for(i=0;i<nFiguri;i++){
      printf("%g\n",figuri[i]->vTable->perimetru(figuri[i]));
      }
   return 0;
}
```

Analizând codul de mai sus, constatăm următoarele:

- Mulţimea elementelor eterogene (*figuri*) are elemente de tipul pointer la structura iniţială (*Figura*). Ţinând cont că *figura* este primul câmp din toate celelalte structuri derivate din ea (ex: *Cerc*), *vTable* are aceeaşi poziţie (offsetul 0 faţă de început) atât în *Figura* cât şi în *Cerc*. Din acest motiv, putem converti un pointer de la *Cerc* la *Figura*, iar apoi să folosim *vTable* din el.
- Mecanismul de apel al unei funcţii polimorfice (ex: *"figuri[i]->vTable->perimetru(figuri[i])"*) este următorul: *vTable*-ul unui element conţine pointerii la funcţiile specifice tipului elementului. De exemplu, dacă tipul elementului de la *figuri[i]* este *Cerc*, *"figuri[i]->vTable->perimetru"* va pointa la funcţia *Cerc_perimetru*. Această funcţie se apelează având ca argument chiar elementul curent, *figuri[i]*, adică elementul de tip *Cerc* la care vrem să-i calculăm perimetrul.
- În această implementare, fiecare tip de element consumă doar atâta memorie cât îi este necesară. Un *Cerc* va necesita memorie doar pentru pointerul *vTable* şi pentru raza sa.
- În această implementare se pot adăuga oricând noi tipuri de obiecte, fără a se modifica cu nimic codul existent. De exemplu, dacă mai adăugăm tipul *Triunghi*, nimic din implementarea lui *Cerc*, *Figura* sau *VTable* nu se va schimba.

Aplicaţii propuse

Aplicaţia 3.2: Să se citească fiecare dintre următoarele declaraţii:

int *(*a[10])(double); double *b(int(*)(float),char);

Aplicaţia 3.3: Să se implementeze o funcţie care primeşte ca parametri un vector de întregi, numărul elementelor din vector transmis prin adresă şi un predicat care testează dacă un întreg îndeplineşte o anumită condiţie. Funcţia va sterge din vector toate elementele care nu îndeplinesc condiţia dată şi va actualiza numărul elementelor cu numărul de elemente rămas în vector. Să se testeze funcţia cu un predicat care testează dacă un număr este negativ şi să se afişeze vectorul rezultat.

Aplicaţia 3.4: Se introduce un număr întreg *n*<10 şi apoi *n* numere de tip *float*. Să se folosească funcţia *qsort* pentru a sorta aceste numere în mod crescător. Se va introduce apoi un număr *x* de tip *float*. Folosind funcţia *bsearch,* să se determine dacă *x* există în vectorul sortat. Se pot consulta paginile de manual pentru funcţiile *qsort* şi *bsearch*.

Aplicaţia 3.5: Să se implementeze o funcţie care tabelează o funcţie matematică de un parametru, primită ca argument. Funcţia tabelată primeşte un parametru de tip *double* şi returnează o valoare tot de tip *double*. Funcţia de tabelare va primi ca parametri: *a* şi *b* - capetele de tip *double* ale intervalului închis de tabelat, *n* - un număr întreg care arată în câte segmente egale se împarte intervalul *[a,b]*, incluzând capetele acestuia şi *f* - funcţia de tabelat. Să se testeze funcţia de tabelare cu valori *a,b* şi *n* citite de la tastatură, tabelând funcţiile *cos*, *sqrt* şi *fabs* din *math.h*
Exemplu: *tabelare(-10,10,20,fabs)* va afişa pe câte o linie valori de forma *f(-10)=10 f(-9)=9 … f(10)=10*

Aplicaţia 3.6: Să se completeze exemplul de implementare a colecţiilor eterogene folosind pointeri la funcţii cu tipul de elemente *Triunghi*. Un triunghi este definit prin lungimile laturilor sale.

Aplicaţia 3.7: În exemplul de implementare a colecţiilor eterogene folosind pointeri la funcţii să se adauge o nouă funcţie polimorfică, *nume*, care returnează numele tipului elementului, ca pointer la un şir de caractere. Să se testeze funcţia adăugată pe colecţia eterogenă.

Aplicaţia 3.8: Folosind funcţii polimorfice, să se implementeze o colecţie eterogenă de produse. Vor fi 2 tipuri de produse: *calculatoare*, definite prin {nume CPU, capacitate memorie în GB} şi *monitoare*, definite prin {diagonală în inch, rezoluţie orizontală, rezoluţie verticală}. Singura funcţie polimorfică va fi *afisare*, care afişează tipul şi toate proprietăţile unui produs pe o linie. De exemplu, dacă avem un calculator şi un monitor, afişarea va fi de forma:
calculator: cpu:Ryzen, memorie:8GB
monitor: diagonala:24, rezolutie:1920x1080