Tehnici de programare

funcții cu număr variabil de argumente; parametri în linia de comandă

Funcții precum *printf* sau *scanf* acceptă oricâte argumente. Ele se numesc funcții cu număr variabil de argumente (varargs). Pentru a defini asemenea funcții, pe ultima poziție a parametrilor se pun trei puncte, **...** (ellipsis). Aceste trei puncte (care au un rol analogic lui ”etc”) se vor putea înlocui la apelul funcției prin oricâte argumente.

Limbajul C are un fel minimalist de a trata argumentele variabile. Ele sunt pur și simplu depuse într-o zonă de memorie, fără a se memora nimic despre ele: nici numărul și nici măcar tipul lor. Această tratare minimalistă asigură o viteză mare de execuție, dar în schimb necesită ca funcția să mai primească informații auxiliare despre valorile cu care a fost apelată. Situația este analogică celei a vectorilor, care, din cauză că nu își memorează numărul de elemente, pe lângă vectorul de elemente, mai trebuie furnizat și acest număr. Din acest motiv, în funcții precum *printf* sau *scanf* trebuie specificate placeholdere (ex: %d) care să specifice ce argumente au fost transmise funcției.

Pentru implementarea funcțiilor varargs, limbajul C pune la dispoziție mai multe funcții și macrouri definite în antetul **<stdarg.h>**. Există două metode de a folosi funcții varargs:

1. În interiorul funcției nu se face nimic cu argumentele variabile, ci acestea sunt doar pasate mai departe altor funcții. Această metodă este foarte utilă pentru a încapsula funcții precum *printf* sau *scanf* în funcții proprii și a le oferi astfel o funcționalitate specifică.
2. Argumentele variabile sunt tratate în interiorul funcției.

Vom începe cu exemplificarea primei metode, în care lista de argumente variabile se pasează altei funcții. Pentru aceasta vom implementa un program cu următoarele cerințe: Se citește un $0<n<=100$ întreg și apoi două valori x și y reale, $x<y$. Se vor citi apoi n valori, care fiecare trebuie să fie în intervalul $[x,y]$. În final se va afișa minimul și maximul valorilor citite. Dacă vreuna dintre valorile introduse nu este conformă cerințelor, programul va afișa valoarea greșită și se va termina imediat.

Exemplul 1: Vom rezolva problema fără funcții varargs:

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

int main()
{
   int i,n;
   float x,y,e,min,max;

   printf("n=");scanf("%d",&n);
   if(n<=0||n>100){
      printf("n invalid: %d\n",n);
      exit(EXIT_FAILURE);
      }
   printf("x=");scanf("%g",&x);
   printf("y=");scanf("%g",&y);
   if(x>=y){
      printf("x=%g nu este mai mic decat y=%g\n",x,y);
      exit(EXIT_FAILURE);
      }
   max=x;
```

```
    min=y;
    for(i=0;i<n;i++){
        printf("e=");scanf("%g",&e);
        if(e<x||e>y){
            printf("element invalid: %g\n",e);
            exit(EXIT_FAILURE);
            }
        if(e<min)min=e;
        if(e>max)max=e;
        }
    printf("min: %g, max: %g\n",min,max);
    return 0;
}
```

Se constată că în program se repetă de 3 ori secvența *printf()/exit()*. Dacă am vrea să scriem o funcție separată pentru afișarea erorii și ieșire din program, problema este că fiecare funcție ar necesita alți parametri: primul *printf* are nevoie de un *int*, al doilea de două valori de tip *float*, iar al treilea de un *float*. În această situație ar trebui să scriem trei funcții separate, câte una pentru fiecare tip de eroare. Pentru a rezolva această situație, ar fi nevoie de o funcție gen *printf*, dar care să iese din program folosind *exit*.

Exemplul 2: Rezolvarea problemei folosind o funcție varargs pentru erori:

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

void err(const char *fmt,...)
{
    va_list va;                 // pointer la lista de argumente variabile (varargs)
    va_start(va,fmt);           // va trebuie initializat cu ultimul argument fix al functiei
    fprintf(stderr,"eroare: ");
    vfprintf(stderr,fmt,va);    // varianta de fprintf care primeste o lista de argumente variabile
    va_end(va);                 // dupa folosirea argumentelor variabile, trebuie apelat va_end
    fputc('\n',stderr);
    exit(EXIT_FAILURE);
}

int main()
{
    int i,n;
    float x,y,e,min,max;

    printf("n=");scanf("%d",&n);
    if(n<=0||n>100)err("n invalid: %d\n",n);
    printf("x=");scanf("%g",&x);
    printf("y=");scanf("%g",&y);
    if(x>=y)err("x=%g nu este mai mic decat y=%g\n",x,y);
    max=x;
    min=y;
    for(i=0;i<n;i++){
        printf("e=");scanf("%g",&e);
        if(e<x||e>y)err("element invalid: %g\n",e);
        if(e<min)min=e;
        if(e>max)max=e;
        }
```

```
    printf("min: %g, max: %g\n",min,max);
    return 0;
}
```

Datorită celor trei puncte din declarație, funcția *err* acceptă un număr variabil de argumente. Această funcție va încapsula funcționalitatea lui *fprintf* și va apela *exit*. Pentru a avea acces la argumentele variabile, a trebuit să includem antetul <*stdarg.h*>. În acesta se găsesc mai multe declarații care încep cu **va**_ (variable arguments):

- **va_list** - este un tip de date, analogic unui pointer, care va pointa la argumentele variabile
- **va_start***(va, ultimul_arg_fix)* - atribuie lui *va*, care trebuie să fie de tipul *va_list*, adresa de început a argumentelor variabile. Pentru aceasta se folosește poziția ultimului argument fix al funcției, de dinainte de **...**
- **va_end**(*va*) - după folosirea listei de argumente variabile *va*, trebuie apelat *va_end* pentru a elibera eventualele resurse alocate lui *va*.

Observație: din cauză că pentru inițializarea listei de argumente variabile este nevoie de poziția ultimului argument fix, nu este posibil să folosim funcții doar cu argumente variabile (ex: f(...))

În exemplul de mai sus, am declarat variabila *va* ca fiind de tipul *va_list*. După *va_start(va,fmt)*, unde *fmt* este ultimul argument fix al funcției *err*, *va* pointează la începutul listei de argumente variabile.

În antetul <*stdio.h*> există mai multe variante pentru funcțiile *printf* și *scanf*. Pentru o listă completă a acestora puteți studia o referință a limbajului C. Se poate căuta și pe internet numele unui antet, pentru se afișa informații despre funcțiile din el. Ca denumire, aceste variante diferă între ele prin folosirea unor litere prefix: **f**-file, **s**-string, **v**-varargs, **n**-max number. Iată pentru *printf* variantele standard:

- *int* **printf***(const char *format, ...)* - scrie argumentele date la *stdout*
- *int* **vprintf***(const char *format,va_list arg)* - primește argumentele sub forma unei liste varargs și le scrie la *stdout*
- *int* **fprintf***(FILE *stream,const char *format, ...)* - scrie argumentele date în fișierul specificat
- *int* **vfprintf***(FILE *stream,const char *format,va_list va)* - primește argumentele sub forma unei liste varargs și le scrie în fișierul specificat
- *int* **snprintf***(char *buf, size_t n,const char *format, ...)* - scrie maxim n caractere din argumentele date în vectorul specificat
- *int* **vsnprintf***(char *buf, size_t n,const char *format,va_list va)* - primește argumentele sub forma unei liste varargs și scrie maxim n caractere în vectorul specificat
- *int* **sprintf***(char *buf,const char *format, ...)* - scrie argumentele date în vectorul specificat. Această funcție se va folosi doar atunci când se știe sigur că argumentele se încadrează în vectorul dat.
- *int* **vsprintf***(char *buf,const char *format,va_list arg)* - primește argumentele sub forma unei liste varargs și le scrie în vectorul specificat. Această funcție se va folosi doar atunci când se știe sigur că argumentele se încadrează în vectorul dat.

Revenind la funcția *err* din exemplul anterior, din cauză că se preferă scrierea mesajelor de eroare la *stderr*, am folosit prima oară *fprintf* pentru a afișa șirul *"eroare: "*, iar apoi *vfprintf* pentru a afișa eroarea propriu-zisă. *vfprintf* (**v**arargs **f**ile *printf*) primește lista de argumente variabile *va* de la funcția *err* și o va formata conform șirului *fmt*. După aceasta, deoarece nu mai este nevoie de lista de argumente variabile *va*, se eliberează resursele alocate pentru ea folosind *va_end(va)*. În final se va mai afișa un \n și se va ieși din program cu *exit*.

A doua metodă de folosire a funcțiilor varargs este cu tratarea argumentelor variabile în interiorul funcției. Pentru aceasta, se iterează lista de argumente variabile și se folosește pe rând fiecare argument. Reamintim faptul că nu există stocate nicăieri nici numărul de argumente variabile și nici tipurile lor, deci aceste informații trebuie să fie accesibile prin alte mijloace, ca de exemplu prin folosirea placeholderelor în *printf*.

Iterarea listei de argumente variabile se face folosind macroul **va_arg***(va,tip)*. Acest macro are doi parametri: lista de argumente variabile care se iterează (inițializată anterior cu *va_start*) și tipul de date corespunzător argumentului curent. La fiecare apel, *va_arg* va returna valoarea argumentului curent din listă, ca fiind de tipul dat și va trece la următorul argument. În acest fel, prin apeluri succesive ale lui *va_arg*, se vor itera toate argumentele din listă.

La iterarea listei de argumente trebuie ținut cont că pentru argumentele varargs, compilatorul face unele conversii implicite. Cele mai importante reguli pentru conversii implicite sunt:

- *float* → *double*
- orice tip întreg cu o dimensiune mai mică decât *int* (*char*, *short*) → *int*
- conversiile tipurilor întregi țin cont că tipul este sau nu cu semn (ex: *short* → *int*, *unsigned short* → *unsigned int*)

Știind aceasta, putem să ne dăm seama de ce la *printf* putem folosi %d pentru afișarea valorilor de tip *char*, *short* sau *int*: din cauză că toate acestea vor fi convertite implicit la *int*. La *scanf* în schimb, trebuie să folosim placeholdere diferite la citirea valorilor numerice (%hhd - *char*, %hd - *short*, %d - *int*), deoarece *scanf* primește ca argumente adresele unde se vor memora valorile citite (transmitere prin adresă) și el trebuie să știe exact tipul variabilei, pentru a stoca la adresa respectivă o valoare de exact acel tip.

Atenție: o constantă numerică, dacă nu are punct sau exponent, este considerată ca fiind de tip *int*. Din acest motiv, dacă dorim să pasăm constante numerice într-o listă variabilă de argumente, iar acolo este nevoie de valori reale, va trebui să scriem constantele numerice ca fiind reale (ex: să adăugăm **.0** după ele, chiar dacă nu au parte zecimală).

Exemplul 3: Să se scrie o funcție *maxN* care primește ca prim parametru un număr *n* de elemente, iar apoi *n* elemente de tip real, care vor fi date sub forma unei liste variabile de argumente. Funcția va returna maximul tuturor argumentelor:

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

double maxN(int n,...)
{
   va_list va;                          // lista de argumente variabile
   va_start(va,n);                      // initializare cu ultimul argument fix
   double max=va_arg(va,double);        // preia prima valoare din va, ca fiind de tip double
   while(--n){
      double e=va_arg(va,double);       // returneaza pe rand fiecare valoare ramasa in va ca fiind de tip double
      if(max<e)max=e;
      }
   va_end(va);
   return max;
}

int main()
{
   printf("%g\n",maxN(3,8,0,5));                // eroare la executie: 8, 0 si 5 sunt considerate ca fiind de tip int, deci
maxN nu le va procesa corect
   printf("%g\n",maxN(3,(double)8,0.0,(float)5));       // apel corect => 8
   return 0;
}
```

Deoarece în lista de argumente varargs *float* se convertește automat la *double*, funcția *maxN* va putea procesa atât valori de tip *float*, cât și valori de tip *double*. Toate valorile de tip *float* vor fi convertite la *double* înainte de apel, deci în interiorul lui *maxN* toate valorile vor fi de tip *double*. Se poate constata cum la fiecare apel al lui *va_arg* se preia câte o valoare din lista de argumente varargs, astfel încât prin apeluri succesive se vor procesa toate valorile.

În *main*, prima oară s-a apelat *maxN* cu valorile 8, 0 și 5. Din cauză că aceste valori nu au nici punct și nici exponent, ele sunt considerate ca fiind de tip *int*, iar *maxN* nu le va procesa corect. La al doilea apel al lui *maxN* valorile au fost convertite la tipuri reale în diverse feluri: ori prin folosirea unei conversii explicite (cast), ori prin adăugarea unei

părți zecimale nule la număr. În cazul conversiei explicite la *float*, aceasta a fost făcută înainte de apel, astfel că la apel valoarea *float* rezultată va fi convertită automat la *double*.

În acest exemplu, folosirea argumentelor varargs a fost simplificată de faptul că toate argumentele au același tip. Dacă argumentele pot avea tipuri diferite, atunci trebuie adaptată folosirea lui *va_arg* la tipurile de argumente cu care a fost apelată funcția.

Exemplul 4: Să se scrie o funcție *citire* care primește ca prim argument un șir de caractere *fmt*, iar apoi o listă de argumente varargs, care reprezintă fiecare adresa unei variabile. Funcția va citi de la tastatură valori și le va depozita în adresele de memorie care sunt date în lista varargs. Pentru a se ști ce tip de date se citește, în *fmt* se dă câte o literă specifică pentru fiecare adresă din listă: d-*int*, f-*float*, s-șir de caractere:

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

void citire(const char *fmt,...)
{
    va_list va;                         // lista de argumente variabile
    va_start(va,fmt);                   // initializare cu ultimul argument fix
    int *addrInt;
    float *addrFloat;
    char *addrStr;
    for( ; ; fmt++ ){                   // bucla infinita in care se proceseaza fiecare caracter din fmt
        switch(*fmt){
            case '\0':                  // daca sa ajuns la sfarsitul lui fmt, iesire din functie
                va_end(va);
                return;
            case 'd':
                addrInt=va_arg(va,int*);    // se poate scrie mai concis: scanf("%d",va_arg(va,int*));
                scanf("%d",addrInt);
                break;
            case 'f':
                addrFloat=va_arg(va,float*);
                scanf("%g",addrFloat);
                break;
            case 's':
                addrStr=va_arg(va,char*);
                scanf("%s",addrStr);
                break;
            }
        }
}

int main()
{
    int n,m;
    float x;
    char buf[100];
    printf("introduceti n, m, y si buf:\n");
    citire("ddfs",&n,&m,&x,buf);
    printf("n=%d, m=%d, x=%g, buf=%s\n",n,m,x,buf);
    return 0;
}
```

Funcția *citire* iterează fiecare caracter din *fmt* și, în funcție de acesta, tratează în mod corespunzător argumentul variabil care urmează să fie returnat de *va_arg*. De exemplu, dacă litera curentă din *fmt* este *"d"*, înseamnă că avem adresa unei valori de tip *int*, deci vom folosi *va_arg(va,int*)*. La această adresă se citește o valoare corespunzătoare folosind *scanf*. Se poate observa că *scanf*-urile primesc ca argumente variabile *addrInt*, *addrFloat* și *addStr* fără operatorul & (adresă) în fața lor, deoarece aceste argumente deja reprezintă adresele unde se vor citi valori.

Pentru situații în care se cere parcurgerea argumentelor variable de mai multe ori, aceasta se poate realiza în două feluri:

- după ce se încheie o parcurgere cu *va_end*, se poate folosi din nou *va_start* pentru reinițializarea listei de argumente variabile
- dacă dorim să ținem minte o anumită poziție din lista de argumente variabile, putem salva lista respectivă într-o copie, folosind macroul **va_copy***(va_destinație, va_sursă)*. *va_copy* copiază *va_sursă* în *va_destinație*, deci ambele liste vor pointa la același argument. După *va_copy, va_destinatie* va trebui eliberată cu *va_end*.

Parametri în linia de comandă

După cum s-a putut constata mai sus, unui program îi pot fi furnizați parametri folosind linia de comandă. Parametrii sunt scriși după numele programului pe care îl rulăm. De exemplu, dacă avem un program *prog*, putem scrie în linia de comandă:

```
./prog ana 21 ana@mail.com
```

Această linie de comandă va transmite lui *prog* la execuție 3 parametri. Fiecare parametru este separat prin spațiu. Toți parametrii se transmit ca șiruri de caractere, chiar dacă ele de fapt sunt numere (ex: 21 este transmis ca un șir de 2 caractere, nu ca un număr de tip *int*). Dacă vrem ca un parametru să includă spațiu, putem folosi ghilimele sau apostroafe. Tot ceea ce se află între ghilimele va fi considerat ca fiind un singur parametru (ex: *./prog "ana ionescu" 21 ana.ionescu@mail.com*).

În interiorul unui program C, parametrii din linia de comandă sunt preluați în felul următor:

- funcția *main* se declară ca având 2 parametri, care trebuie să aibă exact tipurile specificate: **int main(int argc, char *argv[])**
- **argc** (arguments count) este un întreg care specifică numărul parametrilor din linia de comandă, incluzând numele programului executat. Dacă în linia de comandă vor fi dați 3 parametrii, *argc* va fi 4.
- **argv** (arguments vector) este un vector care conține câte un parametru pe fiecare poziție. Este inclus și numele programului executat pe poziția 0, deci primul parametru propriu-zis începe de la indexul 1. După cum se vede din tipul lui *argv*, acesta este un vector de pointeri la *char*, ceea ce în C se folosește pentru a memora șiruri de caractere. Reamintim că toți parametrii, chiar dacă de fapt sunt numere, sunt transmiși ca șiruri de caractere. La terminarea programului, vectorul *argv* și conținutul său nu trebuie eliberate din memorie, aceasta fiind sarcina SO.

Observație: aplicațiile IDE permit specificarea parametrilor din linia de comandă, de exemplu pentru *Code::Blocks* din meniul *Project->Set programs' arguments...*, iar pentru *Visual Studio* din meniul *Project->Properties->Configuration Properties->Debugging->Command Arguments*. Dacă se execută programul având acești parametri setați, este ca și când parametrii respectivi ar fi fost scriși în linia de comandă a programului.

Exemplu: un program C care afişează toţi parametri primiţi în linia de comandă este:

```
```

Salvăm programul într-un fişier *lcom.c* şi îl compilăm cu "*gcc -Wall -o lcom lcom.c*". Va rezulta fişierul binar *lcom*. Dacă scriem în linia de comandă *./lcom* (apel fără niciun parametru), rezultatul va fi:

Avem 1 parametri din linia de comanda.
Parametrul 0: './lcom'

Dacă scriem câţiva parametri din linia de comandă, programul îi va afişa pe toţi. Din nou SO va plasa pe prima poziţie numele programului care a fost rulat: *./lcom "ana ionescu" 21 ana.ionescu@mail.com*

Avem 4 parametri din linia de comanda.
Parametrul 0: './lcom'
Parametrul 1: 'ana ionescu'
Parametrul 2: '21'
Parametrul 3: 'ana.ionescu@mail.com'

După cum se vede, folosirea ghilimelelor a anulat rolul de separator al spațiului, și astfel *”ana ionescu”* a fost considerat ca fiind un singur parametru (atenție când se face copy/paste la cod cu ghilimele, deoarece editoarele de text pot înlocui ghilimelele cu caractere asemănătoare, dar care în C sunt tratate diferit de ghilimele). Numărul 21 a fost transferat tot ca șir de caractere. Pentru a-i obține valoarea numerică, putem folosi funcții de conversie, gen *atoi*, din biblioteca <*stdlib.h*>.

Un exemplu de program care foloseşte parametrii din linia de comandă sub formă de numere întregi este următorul:

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

void stelute(int n)
{
   while(n--)printf("*");
}

void spatii(int n)
{
   while(n--)printf(" ");
}

int main(int argc, char *argv[])
{
   int i, n;
   // se incepe de la 1, pentru a nu se considera si numele programului
   for (i = 1; i < argc; i++) {
      n = atoi(argv[i]);
      // numerele de la pozitii impare sunt considerate stelute, iar cele de la pozitii pare spatii
      if (i % 2 == 1)
         stelute(n);
         else
         spatii(n);
   }
   printf("\n");

   return 0;
}
```

Programul citeşte numere care i se transmit din linia de comandă. În funcţie de aceste numere el afişează pe o singură linie, succesiv, steluţe şi spaţii. Spre exemplu, dacă se transmit numerele 3, 4, 5 şi 6, programul afişează pe aceeași linie 3 steluţe, 4 spaţii, 5 steluţe şi 6 spaţii. Observaţi că s-a folosit funcţia *atoi* pentru a converti parametrii din şir de caractere în numere întregi.

Fișiere de comenzi (opțional)

De regulă parametrii din linia de comandă se folosesc pentru a automatiza execuţia programelor. Automatizarea execuţiei programelor se face prin fișiere de comenzi, sau scripturi care sunt rulate de SO. Pentru Linux, asemenea scripturi se scriu sub formă de fişiere cu extensia **.sh** (shell). Un script conține o secvenţă de comenzi care se doreşte să fie rulate împreună. Fișierele de comenzi *.sh* pot fi destul de complexe, ele putând include pe lângă comenzi, structuri specifice limbajelor de programare, cum ar fi *if, for, while,* etc.

Dacă compilăm programul anterior obținând un fișier executabil având numele *stelute*, putem scrie următorul fișier de comenzi, cu numele *tp.sh*:

```
./stelute 7 3 5
./stelute 0 3 1 6 1 4 1
./stelute 0 3 1 6 1 5 1
./stelute 0 3 1 6 1 5 1
./stelute 0 3 1 6 1 4 1
./stelute 0 3 1 6 5
./stelute 0 3 1 6 1
./stelute 0 3 1 6 1
./stelute 0 3 1 6 1
./stelute 0 3 1 6 1
./stelute 0 3 1 6 1
```

Pentru a putea rula scriptul *tp.sh*, trebuie să marcăm fișierul ca fiind executabil. Acest lucru se face cu comanda ”**chmod +x tp.sh**” (change mode), pentru a seta dreptul de execuție (**+x** eXecute). Pentru fișierele compilate anterior cu *gcc* nu a trebuit să facem aceasta, deoarece *gcc* setează automat fișierele create ca fiind executabile.

Dacă rulăm fişierul *tp.sh*, efectul va fi că se va executa în mod automat programul *stelute* de 11 ori, cu parametrii specificaţi. Pe ecran vor apărea literele TP (de la Tehnici de Programare):

```
./tp.sh
*******    *****
   *       *     *
   *       *       *
   *       *       *
   *       *     *
   *       *****
   *       *
   *       *
   *       *
   *       *
   *       *
```

Aplicații propuse – funcții cu număr variabil de parametrii

Aplicația 6.1: Să se modifice exemplul 2 astfel încât funcția *err* să încapsuleze și condiția de eroare. Pentru aceasta, ea primește în plus pe prima poziție o variabilă de tip *int* care reprezintă o valoare logică. Dacă valoarea este *true*, se va executa *err* ca în exemplu, altfel *err* nu va avea niciun efect. Exemplu de folosire: err(n<=0||n>100,"n invalid: %d\n",n); // fără if în față, deoarece err încapsulează condiția

Aplicația 6.2: Să se scrie o funcție *float *allocVec(int n,...)* care primește pe prima poziție un număr de elemente iar apoi *n* elemente reale. Funcția va aloca dinamic un vector de tip *float* în care va depune toate elementele. Exemplu: allocVec(3,7.2,-1,0) => {7.2, -1, 0}

Aplicația 6.3: Să se scrie o funcție *absN(int n,...)* care primește un număr *n* de adrese de tip *float* și setează la fiecare dintre aceste adrese valoarea absolută de la acea locație. Exemplu: absN(2,&x,&y); // echivalent cu x=fabs(x); y=fabs(y);

Aplicația 6.4: Să se scrie o funcție *crescator(int n,char tip,...)* care primește un număr *n* de valori și returnează 1 dacă ele sunt în ordine strict crescătoare, altfel 0. Caracterul *tip* indică tipul valorilor și poate fi 'd' - *int*, 'f' - *double*. Exemplu: printf("%d",crescator(3,'d',-1,7,9));

Aplicația 6.5: Să se implementeze o funcție *input(const char *fmt,...)*. În șirul *fmt* pot fi caractere obișnuite (orice în afară de %) și placeholdere (% urmat de o literă). Pentru fiecare placeholder posibil (%d - *int*, %f -*float*, %c - *char*), în lista de argumente variabile se va afla adresa unei variabile de tipul corespunzător. Funcția afișează pe ecran caracterele obișnuite și citește de la tastatură pentru placeholdere. Exemplu: input("n=%dch=%c",&n,&ch); // citește o valoare de tip *int* în *n* și de tip *char* în *ch*

Aplicația 6.6: Să se scrie o funcție *char *concat(int n,...)* care primește un număr de șiruri de caractere și apoi șirurile propriu-zise. Funcția va concatena într-un nou șir, alocat dinamic, conținuturile tuturor șirurilor date, cu câte un spațiu între ele și va returna acest șir. Exemplu: concat(3,"Ion","si","Ana") => "Ion si Ana"

Aplicația 6.7: Să se scrie o funcție *int comune(int nVec,...)* care primește un număr de vectori cu valori de tip *int* și vectorii propriu-ziși. Fiecare vector se dă prin 2 argumente: un pointer la elementele sale și dimensiunea. Funcția va returna numărul de elemente comune care se regăsesc în toți vectorii. Exemplu: comune(3,v1,2,v2,3,v3,3) => returnează 2 pentru v1={5,8}, v2={8,3,5}, v3={5,0,8}

Aplicația 6.8: Să se scrie o funcție *sortare(int n,...)* care pe prima poziție are numărul de elemente de sortat, iar pe următoarele poziții *n* adrese de valori de tip *int*. Funcția va sorta crescător valorile de la adresele date. Exemplu: sortare(3,&i,&j,&k); // sortează crescător valorile lui i, j și k

Aplicații propuse – parametrii în linia de comandă

Aplicația 6.9: Scrieți un program care calculează suma parametrilor primiți din linia de comandă (numere reale). Exemplu: execuția fără parametri va fișa 0, iar execuția cu parametrii 1.1 6.57 99.122 va afișa 106.792

Aplicația 6.10: Să se scrie un program denumit *criptare*, care criptează/decriptează un text primit în linia de comandă. Programul va fi apelat în felul următor: ./criptare operatie cheie cuvant1 cuvant2 ... Operația poate fi **enc** pentru criptare sau **dec** decriptare. Cheia este un număr întreg. Algoritmul de criptare este următorul: pentru fiecare literă din cuvinte se consideră codul său ASCII, la care se adună cheia specificată, rezultând un nou cod ASCII, cel al literei criptate. Adunarea este circulară, adică 'z' incrementat cu 1 devine 'a'. Decriptarea este inversă: din cuvintele criptate se scade circular cheia specificată. Literele mari se transformă la început în litere mici.
Exemple:
./criptare enc 1 zaraza -> absbab
./criptare dec 1 bcfdfebs -> abecedar
./criptare enc 10 vine VINE primavara PRImaVAra -> fsxo fsxo zbswkfkbk zbswkfkbk

Aplicația 6.11: Un program primește în linia de comandă o expresie formată din numere reale și operatorii {add-adunare, sub-scădere, mul-înmulțire, div-împărțire}. Programul va calcula valoarea expresiei și va afișa rezultatul. Expresia se calculează de la stânga la dreapta, fără a ține cont de precedența matematică a operatorilor. Exemplu: ./calcul 10.5 add 1.5 div 3 va afișa: 4

Aplicația 6.12: Un program primește în linia de comandă o serie de opțiuni și de cuvinte, mixate între ele. Opțiunile specifică operații care se execută asupra cuvintelor. Opțiunile încep cu - (minus) și pot fi: **u**-transformă toate literele cuvintelor în majuscule; **f**-transformă prima literă în majusculă și următoarele în minuscule; **r**-inversează literele din cuvinte. După fiecare cuvânt se resetează toate opțiunile anterioare. Să se afișeze cuvintele primite în linia de comandă cu transformările aplicate. Exemplu: ./procesare -f mAria -r -u abac va afișa: Maria CABA

Aplicația 6.13: Un program primește în linia de comandă ca prim argument un cod de operație și pe urmă o serie de cuvinte. Codul poate fi: **0**-afișează numărul total de litere din toate cuvintele: **1**-afișează fiecare literă de câte ori

apare în toate cuvintele.
Exemplu: ./statistica 0 ion merge la film va afișa: 14