

Programarea calculatoarelor

FMI

Secția Calculatoare și tehnologia informației, anul I

Cursul 12 / 18.12.2023

Chestiuni organizatorice

- Data examenului scris (numit în continuare Curs):

30 ianuarie 2024, ora 9.00-11.00

- Regulament de evaluare și notare

$$Nota = \min (10, \underset{6p}{Curs} + \underset{4p}{Laborator} + \underset{1p}{Seminar})$$

Chestiuni organizatorice

- Laborator= maximum 4puncte
- Seminar = maximum 1punct (prezență*0.1 +activitate)
- Curs = notă de la 0 la 10. Nu există punct din oficiu! Nota 10=6puncte
- NU intrați în examen (/restanță) dacă:
 - Nu aveți cel puțin jumătate din punctajul de la laborator (2p din 4p)
 - Nu ați participat la cel puțin 3 seminarii (0.3p din 1p)
- Aveți restanță dacă:
 - Nu intrați în examen
 - Nu vă prezentați la examen
 - Nu luați cel puțin nota 5 la testul/ examenul scris
- Restanța: se pastrează punctele obținute pe parcursul semestrului la seminar și laborator.

Programa cursului

□ Introducere

- Algoritmi
- Limbaje de programare.

□ Fundamentele limbajului C

- Introducere în limbajul C. Structura unui program C.
- Tipuri de date fundamentale. Variabile. Constante. Operatori. Expresii. Conversii.
- Tipuri derivate de date: pointeri, tablouri, șiruri de caractere, structuri, uniuni, câmpuri de biți, enumerări
- Instrucțiuni de control
- Directive de preprocesare. Macrodefiniții.
- Funcții de citire/scriere.
- Etapele realizării unui program C.

□ Fișiere text

- Funcții specifice de manipulare.

□ Funcții (1)

- Declarare și definire. Apel. Metode de transmitere a paramerilor. Pointeri la funcții.

□ Tablouri și pointeri

- Legătura dintre tablouri și pointeri
- Aritmetica pointerilor
- Alocarea dinamică a memoriei
- Clase de memorare

□ Șiruri de caractere

- Funcții specifice de manipulare

□ Fișiere binare

- Funcții specifice de manipulare

□ Structuri de date complexe și autoreferite

- Definire și utilizare

□ Funcții (2)

- Funcții cu număr variabil de argumente
- **Preluarea argumentelor funcției main din linia de comandă**
- **Programare generică**
- Recursivitate

Cuprinsul cursului de azi

0. Declarații complexe

**1. Preluarea argumentelor funcției
main din linia de comandă**

2. Programare generică

Declarații complexe

- declarație complexă = combinație de pointeri, tablouri și funcții
- combinăm *atributele*:
 - () – funcție: `int f();`
 - [] – tablou: `int t[20];`
 - * - pointer: `int* p;`
- importanța parantezelor:
 - `int* f()` – f este o funcție ce returnează un pointer
 - `int *f()` – f este o funcție ce returnează un pointer (la fel ca sus)
 - * este un operator prefixat și are o precedență mai mică decât cea a lui ()
 - `int (*f)()` - pointer la o funcție
 - parantezele sunt necesare pentru a impune asocierea corespunzătoare;
 - declarațiile nu pot fi citite de la stânga la dreapta

Declarații complexe

❑ *combinatii valide de attribute:*

- ❑ **int * f()** – funcție ce returnează un pointer la int
- ❑ **int (*f)()** – pointer la o funcție fără argumente, ce returnează un int
- ❑ **int *t[10]** - tablou de 10 pointeri la int
- ❑ **int (*p) [13]** – pointer la un tablou de 13 int
- ❑ **int t[5][6]** – tablou bidimensional
- ❑ **int* (*f)()** - pointer la o funcție ce returnează un pointer
- ❑ **int **p** - pointer la un pointer (pointer dublu) la int
- ❑ **int (* p[7]) ()** - tablou de 7 pointeri la funcții
- ❑ **int ((*f())[3][4]** - pointer la o funcție ce returnează un pointer la un tablou cu 3 linii si 4 coloane de int
- ❑ **etc**

Declarații complexe

- *combinatii invalide de attribute:*

- `[] ()` – funcție ce returnează un tablou
- `() []` – tablou de funcții
- `() ()` – funcție ce returnează o funcție

- în limbajul C nu sunt permise declararea:

- unui tablou de funcții,
- unei funcții care returnează un tablou/o funcție

- dacă vrem ca o funcție să întoarca rezultatul sub forma de tablou

- transmitem tabloul ca argument prin adresa primului element
 - `void numeFuncie(int tablou[]), void numeFuncie(int* tablou)`
- sau creăm tabloul (în Heap) și întoarcem pointerul corespunzător
 - `int* numeFuncie()`

Declarații complexe

- interpretarea unei declarații complexe se face prin înlocuirea *atributelor* (pointeri, funcții, tablouri) prin următoarele *șabloane text*:

| Atribut | Șablon text |
|---------|--------------------|
| () | funcția returnează |
| [n] | tablou de n |
| * | pointer la |

- descifrarea unei declarații complexe se face aplicând *regula dreapta – stânga*, care presupune următorii pași:
 - se începe cu identificatorul
 - se caută în dreapta identificatorului un atribut
 - dacă nu există, se caută în partea stângă
 - se substituie atributul cu șablonul text corespunzător
 - se continuă substituția dreapta-stânga
 - se oprește procesul la întâlnirea tipului datei.

Declarații complexe

□ *regula dreapta – stânga:*

- se începe cu identificatorul
- se caută în dreapta identificatorului un atribut
- dacă nu există, se caută în partea stângă
- se substituie atributul cu șablonul text corespunzător
- se continuă substituția dreapta-stânga
- se oprește procesul la întâlnirea tipului datei.

int (* a[10]) ();

- tablou de 10 pointeri la funcții ce returnează int

double ((*pf())[5])[5];

- pointer la o funcție ce returnează un pointer la un tablou cu 5 linii și 5 coloane de double

Declarații complexe

- <http://cdecl.org/> - instrument pentru conversia declarațiilor din C în limbaj natural

cdecl

C gibberish ↔ English

```
int (* a[10]) ( );|
```

declare a as array 10 of pointer to function returning int

cdecl

C gibberish ↔ English

```
double (*( *pf)()) [5] [5];|
```

declare pf as pointer to function returning pointer to array 5 of array 5 of double

Cuprinsul cursului de azi

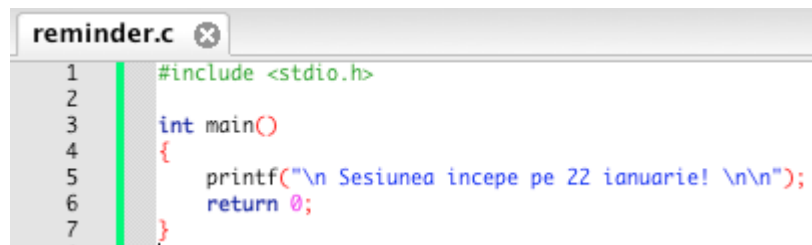
0. Declarații complexe

**1. Preluarea argumentelor funcției
main din linia de comandă**

2. Programare generică

Preluarea argumentelor funcției main din linia de comandă

- ❑ un program executabil (comandă) poate fi lansat în execuție de către interpretorul de comenzi al sistemului de operare.
- ❑ de exemplu, programul **reminder** care afișează la terminal un mesaj:



```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("\n Sesiunea incepe pe 22 ianuarie! \n\n");
6     return 0;
7 }
```

- ❑ program fără parametri, lansat în execuție prin: `>./reminder`

Preluarea argumentelor funcției main din linia de comandă

- ❑ un program poate avea și parametri, care apar după numele comenzii și sunt separați prin spații libere.
- ❑ de exemplu, programul **reminder** ar putea avea un parametru șir de caractere, fiind lansat în execuție, în acest caz prin:
 > **./reminder parametru**
- ❑ programul poate avea acces la parametrii liniei de comandă, dacă funcția **main()** prezintă argumente:
 int main(int argc, char *argv[]) { ... }
- ❑ primul argument, **argc** (*argumentul contor*) este o variabilă de tip întreg care reprezintă *numărul de parametri ai programului/comenzii*
- ❑ al doilea argument, **argv** (*argumentul vector*) este un pointer la un tablou de pointeri la șiruri de caractere, care conțin argumentele (fiecare element al tabloului = un pointer la un șir de caractere = un argument).

Preluarea argumentelor funcției main din linia de comandă

- primul argument, **argc** (*argumentul contor*)
- al doilea argument, **argv** (*argumentul vector*)
- **argv[0]** conține întotdeauna *numele programului*;
- **argv[1]** conține *primul parametru* ca șir de caractere;
- **argv[2]** conține *al doilea parametru* ca șir de caractere;
- **argv[argc-1]** conține *ultimul parametru* ca șir de caractere;
- **argv[argc]** va fi un pointer la un șir vid (**NULL**);
- pentru o comandă fără parametri **argc=1**, iar o comandă cu 2 parametri va avea **argc=3**.

Cuprinsul cursului de azi

0. Declarații complexe

1. Preluarea argumentelor funcției
main din linia de comandă

2. Programare generică

Pointeri generici

- ❑ **pointeri generici** = pointeri la tipul void
- ❑ pot stoca adresa de memorie a unui obiect oarecare
- ❑ dimensiunea zonei de memorie indicate și interpretarea informației conținute nu sunt definite
- ❑ nu poate fi dereferențiat

voidPointer.C

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int i = 6;
6      void *p;
7
8      p = &i;
9      printf("p pointeaza catre o valoare intreaga %d\n", * p);
10
11     return 0;
12 }
```

In function 'int main()':
error: 'void*' is not a pointer-to-object type

Pointeri generici

- ❑ pointeri la tipul void (pointeri generici)
- ❑ pot stoca adresa de memorie a unui obiect oarecare
- ❑ dimensiunea zonei de memorie indicate și interpretarea informației conținute nu sunt definite;
- ❑ nu poate fi dereferențiat
- ❑ pentru a-l folosi (la dereferențiere) trebuie convertit la un alt tip de pointer prin **operatorul cast** (tip *)
- ❑ un singur pointer poate pointa la diferite tipuri de date la momente diferite pe parcursul execuției unui program

Pointeri generici

- ❑ pentru a folosi (la dereferențiere) un pointer generic trebuie convertit la un alt tip de pointer prin operatorul cast (tip *)
- ❑ un singur pointer poate pointa la diferite tipuri de date la momente diferite pe parcursul execuției unui program

voidPointer1.C

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int i = 6;
6      char c = 'a';
7      void *p;
8
9      p = &i;
10     printf("p pointeaza catre o valoare intreaga %d\n", *(int*) p);
11
12     p = &c;
13     printf("p pointeaza catre un char %c\n", *(char*) p);
14
15     return 0;
16 }
```

p pointeaza catre o valoare intreaga 6
p pointeaza catre un char a

Pointeri generici

- pointer la tipul void (pointer generic)

```
void_pointer.c ✕
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     int i = 300;
7     void *p;
8
9     p = &i;
10    printf("\n Valoare pointer void = %p \n", p);
11    printf("\n Valoare p dereferentiat ca int = %d \n", *(int*)p);
12    printf("\n Valoare p dereferentiat ca char = %d \n", *(char*)p);
13
14    // 300 in binar = 0000 0001 0010 1100
15    // 44 in binar =          0010 1100
16
17
18    float v[5] = {1.2, 0.4, 0.6, 6.2, 5.1};
19    p = v;
20    printf("\n Valoare pointer = %f \n\n", ((float*)p)[2]);
21    return 0;
22
23 }
```

Pointeri generici

- pointer la tipul void (pointer generic)

```
void_pointer.c ✕
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     int i = 300;
7     void *p;
8
9     p = &i;
10    printf("\n Valoare pointer void = %p \n", p);
11    printf("\n Valoare p dereferentiat ca int = %d \n", *(int*)p);
12    printf("\n Valoare p dereferentiat ca char = %d \n", *(char*)p);
13
14    // 300 in binar = 0000 0001 0010 1100
15    // 44 in binar =          0010 1100
16
17
18    float v[5] = {1.2, 0.4, 0.6, 6.2, 5.1};
19    p = v;
20    printf("\n Valoare pointer = %f \n\n", ((float*)p)[2]);
21    return 0;
22
23 }
```

Valoare pointer void = 0xbfd2e278

Valoare p dereferentiat ca int = 300

Valoare p dereferentiat ca char = 44

Valoare pointer = 0.600000

Programare generică

- ❑ **programare generică = paradigmă de programare** în care codăm diferiți algoritmi pentru tipuri de date generice (neinstantiate)
- ❑ **exemplu**: un algoritm de sortare al unui tablou de numere arată la fel și pentru date de tipul char, int, float, double. => putem particulariza funcția de comparare să lucreze cu date de tipul char, int, float, double.
- ❑ în programare generică folosim pointeri generici

Programare generică

- ❑ **exemplul 1:** Scrieți o funcție generică de căutare care să returneze un pointer generic către prima apariție a valorii **c** în tabloul unidimensional **v** format din **n** elemente, fiecare având dimensiunea **dim** octeți, sau pointerul **NULL** dacă valoarea **c** nu se găsește în tablou.

```
5
6 void* cauta(const void* v, const void* c, int n, int dim)
7 {
8     char *p = (char *)v;
9     int i;
10
11     for (i=0; i<n; i++)
12         if (!memcmp(c, p+i*dim, dim)) return p+i*dim;
13
14     return NULL;
15 }
16
```

Funcția **memcmp**: *compare bytes in memory*

```
#include <string.h>
```

```
int memcmp(const void *s1, const void *s2, size_t n);
```

- The *memcmp()* function shall compare the first *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*.
- The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the objects being compared.

Sa presupunem ca vrem sa construim functia de cautare secventiala a unui element într- un vector de întregi; functia va returna pozitia elementului sau -1 daca el nu exista.

```
int cauta(int *v, int n, int k)  
{  
    for(int i=0; i<n; i++) if( v[i]==k ) return i;  
    return -1;  
}
```

Functia primeste ca parametri vectorul (int *v), numarul de elemente (int n) si elementul de cautat (int k).

Pentru a nu fi dependenta de tipul de data al vectorului, functia va fi reconstruita astfel încât vectorul sa fie primit prin intermediul unui pointer la void.

Lucru cu o adresa spre un tip nedefinit are implicatii în sensul ca e necesar sa se transmita un parametru suplimentar care sa exprime lungimea tipului de data a elementelor vectorului.

```
int cauta(void *v, int n, int dim_el, void *el)
{
    char *ec=(char*)v;
    for(int i=0; i<n; i++)
        if(!memcmp(ec+i*dim_el,el,dim_el))
            return i;
    return -1;
}
```

unde **int dim_el** = dimensiunea în baidi a tipului elementelor vectorului, elementul de cautat este dat prin adresa lui (parametrul void *el), lungimea se considera a fi aceeasi cu a elementelor vectorului.

Se observa ca pointerul primit în functie (v) este convertit din void* în char* pentru ca dimensiunea unui caracter este de un octet, iar dimensiunea unui element (dim_el) este exprimata în baidi.

Compararea dintre elemente se face folosindu-se functia de biblioteca memcmp (al cărei prototip se află în fisierul header memory.h) care primeste doua adrese si lungimea în baidi a zonei de memorie de comparat.

main.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <memory.h>
4
5  int cauta(void *v,int n,int dim_el,void *el)
6  {
7      char *ec=(char*)v;
8      for(int i=0; i<n; i++)
9          if(!memcmp(ec+i*dim_el,el,dim_el))
10             return i;
11     return -1;
12 }
13
14 int main()
15 {
16     int k;
17     int a[]={3,6,4,1,8},t=4;
18     (k=cauta(a,sizeof(a)/sizeof(int),sizeof(int),&t)) == -1 ?
19         printf("\nElement inexistent!") : printf("\nElementul pe pozitia %d",k);
20
21     double b[]={3.5,6.2,4.9,28.15},el_r=6.2;
22     (k=cauta(b,sizeof(b)/sizeof(double),sizeof(double),&el_r)) == -1 ?
23         printf("\nElement inexistent!") : printf("\nElementul pe pozitia %d",k);
24
25     char *sir="Un Sir de Caractere!", el_c='i';
26     (k=cauta(sir,strlen(sir),sizeof(char),&el_c)) == -1 ?
27         printf("\nElement inexistent!") : printf("\nElementul pe pozitia %d",k);
28
29     return 0;
30 }
```

Elementul pe pozitia 2
Elementul pe pozitia 1
Elementul pe pozitia 4

Programare generică

void_pointer2_int.c ✕

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <memory.h>
4
5 void* cauta(const void* v, const void* c, int n, int dim)
6 {
7     char *p = (char *)v;
8     int i;
9
10    for (i=0; i<n; i++)
11        if (!memcmp(c, p+i*dim, dim)) return p+i*dim;
12
13    return NULL;
14 }
15
16 int main()
17 {
18     int v[5] = {3, 0, 6, 2, 5};
19     int s=2, *p;
20
21     p = (int *) cauta(v, &s, sizeof(v)/sizeof(int), sizeof(int));
22
23     if (p) printf("\n Primul %d se afla pe pozitia %d \n\n ", s, p-v);
24
25     return 0;
26 }
```

Programare generică

void_pointer2_int.c ✕

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <memory.h>
4
5 void* cauta(const void* v, const void* c, int n, int dim)
6 {
7     char *p = (char *)v;
8     int i;
9
10    for (i=0; i<n; i++)
11        if (!memcmp(c, p+i*dim, dim)) return p+i*dim;
12
13    return NULL;
14 }
15
16 int main()
17 {
18     int v[5] = {3, 0, 6, 2, 5};
19     int s=2, *p;
20
21     p = (int *) cauta(v, &s, sizeof(v)/sizeof(int), sizeof(int));
22
23     if (p) printf("\n Primul %d se afla pe pozitia %d \n\n ", s, p-v);
24
25     return 0;
26 }
```

Primul 2 se afla pe pozitia 3

Programare generică

```
void_pointer2_char.c ✕
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <memory.h>
4
5
6 void* cauta(const void* v, const void* c, int n, int dim)
7 {
8     char *p = (char *)v;
9     int i;
10
11     for (i=0;i<n;i++)
12         if (!memcmp(c,p+i*dim,dim)) return p+i*dim;
13
14     return NULL;
15 }
16
17
18
19 int main()
20 {
21     char string[] = "Un sir de caractere oarecare";
22     char chr='c',*r;
23
24     r = (char *) cauta(string,&chr,sizeof(string),1);
25
26     if (r) printf("Primul %c se afla pe pozitia %d \n ",chr, r-string);
27
28     return 0;
29 }
```

Programare generică

```
void_pointer2_char.c ✕
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <memory.h>
4
5
6 void* cauta(const void* v, const void* c, int n, int dim)
7 {
8     char *p = (char *)v;
9     int i;
10
11     for (i=0; i<n; i++)
12         if (!memcmp(c, p+i*dim, dim)) return p+i*dim;
13
14     return NULL;
15 }
16
17
18
19 int main()
20 {
21     char string[] = "Un sir de caractere oarecare";
22     char chr='c', *r;
23
24     r = (char *) cauta(string, &chr, sizeof(string), 1);
25
26     if (r) printf("Primul %c se afla pe pozitia %d \n ", chr, r-string);
27
28     return 0;
29 }
```

Primul c se afla pe pozitia 10

Cursul 7: Utilitatea pointerilor la funcții

- se folosesc în programarea generică, realizăm apeluri de tip callback
- o funcție C transmisă, printr-un pointer, ca argument unei alte funcții F se numește și funcție “callback”, pentru că ea va fi apelată “înapoi” de funcția F
- exemple:
int suma(int n, int (*expresie)(int)); (sumă generică de n numere)

void qsort(void *adresa, int nr_elemente, int dimensiune_element, int (*cmp)(const void *, const void *)) ;
(funcția qsort din stdlib.h)

Cursul 7: Utilitatea pointerilor la funcții

□ **exemplul 2:** Să se calculeze suma $S_k(n) = \sum_{i=1}^n i^k$

$$S_1(n) = 1 + 2 + \dots + n$$

$$S_2(n) = 1^2 + 2^2 + \dots + n^2$$

$$S_k(n) = \sum_{i=1}^n \textit{expresie}(i)$$

Folosind pointeri la funcții pot
să văd funcția ca o variabilă

Cursul 7: Utilitatea pointerilor la funcții

■ **exemplul 2:** Să se calculeze suma $S_k(n) = \sum_{i=1}^n i^k$ pentru $k=\{1,2\}$.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int suma(int n, int (*expresie)(int))
5  {
6      int i,s=0;
7      for(i=1;i<=n;i++)
8          s = s + expresie(i);
9      return s;
10 }
11
12 int expresie1(int x)
13 {
14     return x;
15 }
16
17 int expresie2(int x)
18 {
19     return x*x;
20 }
21
22 int main()
23 {
24
25     int S1 = suma(5,expresie1);
26     printf("S1 = %d\n",S1);
27     int S2 = suma(5,expresie2);
28     printf("S2 = %d\n",S2);
29     return 0;
30 }
```

Cursul 7: Utilitatea pointerilor la funcții

- **exemplul 2:** Să se calculeze suma $S_k(n) = \sum_{i=1}^n i^k$ pentru $k=\{1,2\}$.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int suma(int n, int (*expresie)(int))
5  {
6      int i,s=0;
7      for(i=1;i<=n;i++)
8          s = s + expresie(i);
9      return s;
10 }
11
12 int expresie1(int x)
13 {
14     return x;
15 }
16
17 int expresie2(int x)
18 {
19     return x*x;
20 }
21
22 int main()
23 {
24
25     int S1 = suma(5,expresie1);
26     printf("S1 = %d\n",S1);
27     int S2 = suma(5,expresie2);
28     printf("S2 = %d\n",S2);
29     return 0;
30 }
```

S1 = 15
S2 = 55

Process returned 0 (0x0)
Press ENTER to continue.

execution time : 0.005 s

Cursul 7: Utilitatea pointerilor la funcții

❑ funcția `qsort` din `stdlib.h` folosită pentru sortarea unui vector/tablou.

❑ antetul lui `qsort` este:

```
void qsort (void *adresa, int nr_elemente, int dimensiune_element, int  
(*cmp) (const void *, const void *))
```

- **adresa** = pointer la adresa primului element al tabloului ce urmeaza a fi sortat (pointer generic – nu are o aritmetică inclusă)
- **nr_elemente** = numarul de elemente al vectorului
- **dimensiune_element** = dimensiunea in octeți a fiecărui element al tabloului (char = 1 octet, int = 4 octeți, etc)
- **cmp** = funcția de comparare a două elemente

Programare generică - qsort

- ❑ funcția qsort din stdlib.h - pentru sortarea unui vector/tablou.

```
void qsort (void *adresa, int nr_elemente, int dimensiune_element,  
            int (*cmp) (const void *, const void *))
```

```
int cmp(const void *a, const void *b)
```

adresele a două elemente din tablou



- ❑ **cmp** este o funcție generică comparator, compară 2 elemente de orice tip. Întoarce:
 - un număr < 0 dacă vrem a la stânga lui b
 - un număr >0 dacă vrem a la dreapta lui b
 - 0, dacă nu contează

Programare generică - qsort

Exemplu de funcție cmp pentru sortarea unui vector de numere întregi:

```
int cmp(const void *a, const void *b)
{
    int va, vb;
    va = *(int*)a;
    vb = *(int*)b;
    if(va < vb) return -1;
    if(va > vb) return 1;
    return 0;
}
```



```
int cmp(const void *a, const void *b)
{
    return *(int*)a - *(int*)b;
}
```

Programare generică - qsort

Exemplu: sortarea unui vector de numere întregi:

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int cmp(const void *a, const void *b)
5  {
6      return *(int*)a - *(int*)b;
7  }
8
9  int main()
10 {
11     int v[] = {0,5,-6,9, 7, 12, 8, 7, 4};
12     qsort(v,9,sizeof(int),cmp);
13     int i;
14     for(i=0;i<9;i++)
15         printf("%d ",v[i]);
16     printf("\n");
17
18     return 0;
19 }
```

-6 0 4 5 7 7 8 9 12

Programare generică

□ **exemplul 3:** scrieți o funcție generică de căutare care să returneze un pointer generic către prima apariție a valorii **x** în tabloul unidimensional **t** format din **n** elemente, fiecare având dimensiunea **d** octeți, sau pointerul NULL dacă valoarea **x** nu se găsește în tablou.

Folosiți o funcție comparator **cmp** cu antetul

int (cmpValori*)(const void*, const void *)

care returnează valoarea 1 dacă valorile aflate la adresele primite ca parametri sunt egale sau 0 în caz contrar.

Explicitarea funcției concrete de comparație se va face în apel.

Vom considera două cazuri:

- compararea a două valori întregi,
- compararea a două șiruri de caractere.

Programare generică

❑ exemplul 3:

cautareGenerica1.C

```
1  #include <stdio.h>
2  #include <memory.h>
3
4  void* cauta(const void *x, const void *t, int n, int d, int (*f)(const void*, const void*))
5  {
6      char *p = (char*)t;
7      for(int i=0; i<n; i++)
8          if(f(x, p+i*d))
9          {
10             return p+i*d;
11         }
12     return NULL;
13 }
14
15 int cmpInt(const void* a, const void* b)
16 {
17     return *((int*)a) == *((int*)b) ;
18 }
19
20 int cmpChar(const void* a, const void* b)
21 {
22     return (*((char*)b) == *((char*)a));
23 }
24
```

Programare generică

❑ exemplul 3:

```
26 int main()
27 {
28     int v[] = {10,1,12,3,14,5};
29     int x;
30     printf("x = ");scanf("%d",&x);
31     int *p;
32     p = (int*)cauta(&x,v,sizeof(v)/sizeof(int),sizeof(int),cmpInt);
33     if(p)
34         printf("%d se afla in tabloul v pe pozitia %d \n",x,p-v);
35
36     char w[] = "Orice student intelege programarea generica";
37     char ch = 's';
38     char *q;
39     q = (char*)cauta(&ch,w,sizeof(w)/sizeof(char),sizeof(char),cmpChar);
40     if(q)
41         printf("%c se afla in tabloul w pe pozitia %d \n",ch,q-w);
42
43     return 0;
44 }
```

x = 12
12 se afla in tabloul v pe pozitia 2
s se afla in tabloul w pe pozitia 6

Cursul 12

1. Preluarea argumentelor funcției main din linia de comandă
2. Programare generică

Cursul 13

1. Funcții recursive
2. *Exemple de subiecte*



Cursul 13/ 08.01.2024

Cursul 14/ 15.01.2024

Recapitulare

Examen/ 30.01.2024

Restanță ? (19-25.02.2024)