Mircea-F. VAIDA
Ligia D. CHIOREAN
Adriana STAN
Cosmin STRILEȚCHI
Petre G. POP
Ștefan-S. DRAGOȘ

Aplicații de bază folosind C/C++. Elemente practice. Varianta bilingvă

Basic applications using C/C++. Practical elements. Bilingual variant

Editura UTPRESS
Cluj-Napoca, 2023
ISBN 973-606-737-679-1

Mircea-F. VAIDA
Ligia D. CHIOREAN
Adriana STAN
Cosmin STRILEȚCHI
Petre G. POP
Ștefan-S. DRAGOȘ

Aplicații de bază folosind C/C++. Elemente practice. Varianta bilingvă

Basic applications using C/C++. Practical elements. Bilingual variant

UTPRESS
Cluj-Napoca, 2023
ISBN 978-606-737-679-1

Editura UTPRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director: ing. Dan COLȚEA

Recenzia: Prof.dr.ing. Eugen Lupu
Conf.dr.ing. Simina Emerich

Pregătire format electronic on-line: Gabriela Groza

ISBN 978-606-737-679-1

Cuvânt înainte

În cadrul cărții au fost tratate cele mai importante concepte necesare pentru învățarea elementelor de bază ale limbajului de programare C/C++: tipurile de date, operatorii, instrucțiunile, operațiile de intrare ieșire cu funcții C, respectiv mecanismul din C++, funcțiile, tipurile agregate de tip tablou, pointeri și referințe, alocarea dinamica a memoriei, structuri și alte tipuri utilizator, respectiv lucrul cu fișiere. Toate aceste noțiuni sunt introduse treptat și ilustrate prin exemple clare.

Fiecare capitol al cărții prezintă obiectivele urmărite prin parcurgerea lui, concepte teoretice esențiale, exemple, întrebări care să verifice înțelegerea noțiunilor studiate și propune probleme care pot fi rezolvate folosind elementele prezentate în capitolul respectiv.

Breviarul teoretic este prezentat în română și în engleză, fiind un suport atât pentru studenții care urmează studii în limba română, cât și celor de la specializări în limba engleză. Problemele rezolvate sunt documentate prin comentarii în limba engleză, pentru a fi utile tuturor studenților, dar fară a încărca exagerat codul, permițând astfel urmărirea și înțelegerea acestuia fără dificultate.

Conținutul este structurat cu formate speciale, dedicate secvențelor de cod, exemplelor, observațiilor, etc. Aceste lucruri contribuie din plin la parcurgerea cu ușurință a conținutului, facilitând înțelegerea.

Lucrarea de față este foarte utilă ca suport pentru studenți, mai ales cei de la profilul electronică și telecomunicații, facilitând înțelegerea unui limbaj util în implementarea de aplicații cu preponderență inginerești, atât pentru proiectele pe care le au de realizat pe durata studiilor, cât și în profesia pe care o vor practica în viitor.

Autorii

Cluj-Napoca, 2023

Cuprins/Contents

1. Aplicații minimale C/C++

Basic C/C++ applications

1.1. Obiective

- Înțelegerea structurii unui program C/C++
- Înțelegerea noțiunilor: comentarii, directive preprocesor, declarații globale, funcții, definiția și prototipul unei funcții, apelul unei funcții, parametri formali și parametri actuali
- Scrierea și testarea unor programe simple C/C++

1.2. Objectives

- Understanding the structure of a C/C++ program
- Understanding the meaning of comments, preprocessor directives, global declarations, functions, a function's definition and prototype, calling functions, formal and actual parameters
- Writing and testing some simple C/C++ programs

1.3. Breviar teoretic

Forma generală a unei aplicații C/C++ urmărește de obicei următoarele etape:

- Comentarii inițiale, ce prezintă scopul aplicației și realizatorul ei
- Directive preprocesor de tip *include*
- Directive preprocesor de tip *define*
- Declarații globale de variabile sau alte tipuri de date
- Prototipuri de funcții
- Funcția *main()*
- Definirea celorlalte funcții din cadrul aplicației

Limbajul C este un *limbaj procedural*. La baza limbajelor procedurale stă conceptul de *apel de procedură*, iar în C avem atât *proceduri* cât și *funcții*, procedura fiind considerată de unii un caz particular de funcție care nu returnează nimic (`void`).

Limbajul C++ păstrează caracteristicile limbajului C adăugând elemente specifice limbajelor orientate pe obiecte, generice, funcționale.

Declarațiile variabilelor în limbajul C/C++ au forma:

```
tip_de_date_1 nume_variabilă_1, nume_variabilă_2, ...;
tip_de_date_2 nume_variabilă_3;
```

Tipurile de date de bază în limbajul C pot fi recunoscute după *cuvintele cheie*:

`char`	pentru caractere;
`int`	pentru întregi cu semn;
`float`	real simplă precizie;
`double`	real dublă precizie.

Tipurile de date de bază pot fi modificate cu ajutorul cuvintelor cheie `signed`, `unsigned`, `long` și `short` pentru a obține variante ale lor stocate pe un număr mai mare de biți sau cu comportament diferit.

Pentru a consulta listele de cuvinte cheie specifice limbajului C/C++, puteți consulta următoarele referințe:

- Cuvinte cheie C: https://en.cppreference.com/w/c/keyword
- Cuvinte cheie C++: https://en.cppreference.com/w/cpp/keyword

În C++, avem în plus, printre altele, următoarele tipuri de date:

`bool`	tipul Boolean, pentru stocarea valorilor logice `true` (1) sau `false` (0);
`wchar_t`	un tip de date pentru reprezentarea caracterelor pe 2 octeți;
`long long int`	un tip de date pentru întregi pe 64 de biți (implementarea poate diferi în funcție de compilator).

Preprocesarea permite prelucrarea unui program sursă C sau C++ înainte de a fi supus compilării și asigură:

- includeri de fișiere cu text sursă;
- definiții și apeluri de macro-uri;
- compilare condiționată.

Constantele sunt valori fixe (numerice, caractere sau șiruri de caractere), care nu pot fi modificate de program.

Pentru a specifica caractere care nu pot fi afișate se utilizează *secvențe escape*. Acestea încep întotdeauna cu caracterul „\” („backslash”).

Noile compilatoare C++ permit utilizarea caracterelor *wide* și a *șirurilor de caractere wide* atunci când caracterele sunt reprezentate pe mai mult de 1 octet.

Din motive de securitate, este indicat ca șirurile de caractere definite în programele C/C++ să fie marcate constante cu modificatorul `const`, ca în exemplul următor:

```
const char tab[] = "Program de test";
```

Parametrii și funcțiile `const` sunt utile în evitarea erorilor legate de *efecte secundare*, dar funcțiile sunt încă susceptibile la schimbări în mediul de execuție global.

1.4. Theoretical brief

The general form of a C/C++ application usually follows the following stages:

- Initial comments presenting the purpose of the application and its developer
- Preprocessor directives of *include* type
- Preprocessor directives of *define* type
- Global variable declarations or other data types
- Function prototypes
- The *main()* function
- Definition of other functions within the application

The C language is a *procedural language*, which has at their base the concept of a *procedure call*. In C we have both *procedures* and *functions*, the procedure being considered a particular type of function that returns nothing (`void`).

The C++ language preserves the characteristics of C by adding elements specific to object-oriented, generic, functional languages.

Variables in C/C++ are declared in the following way:

```
data_type_1 variable_1_name, variable_2_name, ...;
data_type_2 variable_3_name;
```

The basic data types in C language can be recognized through the following *keywords*:

`char`	for characters;
`int`	for signed integers;
`float`	for single-precision floating point numbers;
`double`	for double-precision floating point numbers.

The basic data types can be modified using the keywords `signed`, `unsigned`, `long` and `short` in order to obtain variants that are stored on a larger number of bits or that have a different behavior.

The C/C++ language has many other keywords that can be consulted at the following links:

- C keywords: https://en.cppreference.com/w/c/keyword
- C++ keywords: https://en.cppreference.com/w/cpp/keyword

In C++, among other things, we have some new basic data types:

`bool`	the Boolean type, for storing the logic values *true* (1) or *false* (0);
`wchar_t`	a data type for representing characters on 2 bytes;
`long long int`	a data type for 64 bit integers (the implementation may differ between compilers).

Preprocessing allows the processing of a C or C++ source program before compilation and ensures:

- inclusion of files containing source code;
- definitions and calls of macros;
- conditional compilation.

Constants are fixed values (numeric, characters or character strings), that cannot be modified by the program.

To specify characters that cannot be displayed *escape sequences* are used. Escape sequences always start with the „\" („backslash") character.

New C++ compilers allow using *wide* characters and *wide character strings* when characters are represented on more than 1 byte.

Out of security reasons, predefined character strings should be marked as constant with the `const` keyword, like in the following example:

```
const char tab[] = "Test program";
```

The parameters and functions marked with the `const` keyword are useful in avoiding errors related to *side effects*, but functions are still susceptible to changes in the global execution environment.

1.5. Despre exemple

Exemplele următoare, atât din acest capitol cât și din celelalte, au fost testate în medii de programare Microsoft Visual Studio Community pe 32 / 64 biți.

Specific acestui mediu de programare este utilizarea pentru intrări/ieșiri C++ a combinației:

```
#include <iostream>
using namespace std; //specifica utilizarea spatiului de nume standard
```

Această combinație include implicit și bibliotecile de bază ale limbajului C.

În alte medii de programare C/C++ se folosește pentru operații de intrare/ieșire C++ doar directiva *include* următoare:

```
#include <iostream.h>
```

1.6. About examples

The following examples, both from this chapter and others, have been tested in Microsoft Visual Studio Community on 32 / 64 bits programming environments.

Specific to this programming environment is the usage of the C++ input/output combination:

```
#include <iostream>
using namespace std; //specifies the use of the standard namespace
```

This combination implicitly includes the basic C language libraries as well.

In other C/C++ programming environments, only:

```
#include <iostream.h>
```

is used for C++ input/output operations.

1.7. Exemple cu cod sursă / Examples of source code

1.7.1. Ex. 1

```
/* Program for reading and displaying an integer using functions.   */

// Preprocessing directives
#define _CRT_SECURE_NO_WARNINGS //allows using the standard scanf( ) in VC++

// Header files containing prototypes of functions from the standard library
// used in the program.
#include <stdio.h>
//#include <conio.h>

// Global declarations
/* Function prototypes: for each function specify
- the name
- the list of formal parameters
- the returned type
*/
```

```
int read_int(void);
void display_int(int);

// Function main(): has the same structure as any function
int main( ){        // Function body
// Local declarations

// Local variable for storing a number
int n;

// Instructions

n = read_int(); // Call the reading function
display_int(n); // Call the display function

//  To see the result, wait for a key press
// no longer necessary in latest VC++
//_getch();

return 0;
}//main

// Function definitions

// Function for reading an integer
int read_int(void){ // Function body
// Local declarations
int nr;

// Instructions

// Call a function from the standard library for displaying a message
printf("\n Type an integer number: ");

// Call a function from the standard library for reading an integer
scanf("%d", &nr);

// Return result
return nr;
} //read_int

// Function for displaying an integer
void display_int(int nr){
//  Call a function from the standard library for displaying a message
// and the integer number received as parameter
printf("\nYou've entered the number:  %d\n", nr);
} // display_int
```

1.7.2. Ex. 2

```
/* Program for calculating the average of two integers.     */

// Preprocessor directives
#define _CRT_SECURE_NO_WARNINGS
```

```
// header files used in the program
#include <stdio.h>
//#include <conio.h>

// Global declarations

// Function prototypes
int read_int();
float average(int, int);

int main(){
 // Local declarations
 float avg;  // floating point variable for storing the average

 int n1, n2; // integer variables for the values to be read
 n1 = read_int();
 n2 = read_int();

 avg = medie_a(n1, n2); // Call the function that determines the average

 // Print result
 printf("\n\t Average value : %f\n", avg);
 //_getch();
return 0;
}//main

// Function definitions

// Function that calculates the average of two integers
float average(int n1, int n2){
// Local declarations
float avg;    // floating point variable for storing the result

avg = (n1 + n2)/2.0f;
return avg;
} // average

// Function for reading an integer
int read_int(){
int nr;
printf("\nType an integer number : ");
scanf("%d", &nr);
return nr;
}// read_int
```

1.7.3. Ex. 3

```
/* Program for exemplifying characters, character strings, simple and wide     */
#include<iostream>
using namespace std;

int main( ) {
    char character = 'a';//0x61=97
    wchar_t wide_character = L'a';
    cout << "The character is: " << character << endl;
```

```
    cout << "The character size: " << sizeof(character) << endl;//1
    wcout << "The wide character is: " << wide_character << endl;
    cout << "Wide character size: " << sizeof(wide_character);//2
    const char str1[ ] = "\nThis is character array";
    cout << str1 << "\nThe character array size: " << sizeof(str1) << endl;
    const wchar_t str2[ ] = L"This is wide character array";
    wcout << str2 << "\nThe wide character array size: " << sizeof(str2)<< endl;
    return 0;
}//main
```

1.8. Întrebări

1. Care este structura unui program C/C++ ?
2. Ce sunt comentariile ? La ce se folosesc ?
3. Care este deosebirea între prototipul și definiția unei funcții ?
4. Cum se face apelul unei funcții ?
5. Cum se face revenirea din funcții ?

1.9. Questions

1. What is the structure of a C/C++ program?
2. What are comments? What are they used for?
3. What is the difference between a function's prototype and definition?
4. How can a function be called?
5. How is a function's returning done?

1.10. Lucru individual

1. Să se scrie un program pentru determinarea mediei aritmetice a trei numere neîntregi. Considerați și varianta că numerele sunt întregi.
2. Să se scrie un program pentru determinarea mediei geometrice a două numere întregi (folosiți funcția `sqrt()` din `<math.h>`, sau implementați un algoritm de aproximări succesive, vezi metoda lui Newton).
3. Să se scrie un program C/C++ care definește o variabilă întreagă care va fi inițializată cu valori constante. Afișați rezultatul cu ajutorul supraîncărcării operatorului `<<` și a lui `cout`.
4. Definiți un șir de caractere care va fi afișat cu `cout`. Definiți alte șiruri de caractere folosind secvențe escape. Verificați utilizarea spațiilor albe.
5. Să se scrie un program în care se dau 3 numere întregi și se cere să se calculeze suma lor ponderată, ponderile fiind numere subunitare a căror sumă este 1 (date prin program sau citite de la consolă).
6. Definiți într-un program constante simbolice de tipuri diferite (întregi, reale, șiruri de caractere). Afișați valorile acestor constante utilizând operatorul `<<` și fluxul `cout`.
7. Definiți 3 numere reale a, b, și c. Afișați rezultatul operației `1/a+1/b+1/c`. Efectuați aceeași operație considerând ca și intrare 3 numere întregi.

1.11. Individual work

1. Write a program that determines the average value of 3 non-integer numbers. Also consider the option that the numbers are integers.

2. Write a program that determines the geometric average of 2 integer numbers (use the `sqrt()` function from `<math.h>`, or implement a successive approximation algorithm, see Newton's method).
3. Write a C/C++ application that defines an integer variable, initialized with several constant values. Display its value by overloading the << operator and by using the **cout** object.
4. Define an array of characters that will be displayed using `cout`. Display other character arrays and use escape sequences. Verify the usage of the whitespaces.
5. Write a program that defines 3 integer values. Calculate and display their weighted sum, the weights being represented as positive values smaller than 1 that add up to 1 (given by the program or read from the console).
6. Define several symbolic constants of different types (integer numbers, real numbers, arrays of characters). Display their values using `cout` and the << operator.
7. Define 3 real numbers named a, b and c. Display the value of `1/a + 1/b + 1/c`. Display the same result considering as input 3 integer numbers.

2. Aplicaţii cu funcţii şi intrări/ieşiri în C/C++

C/C++ applications using functions and input/output operations

2.1. Obiective

- Înţelegerea noţiunii de flux (stream)
- Înţelegerea sintaxei funcţiilor `printf()` şi `scanf()`
- Utilizarea specificatorilor de format pentru `printf()` şi `scanf()`
- Înţelegerea sintaxei operatorilor >> (extracţie) şi << (inserţie)
- Scrierea şi testarea unor programe simple ce folosesc intrări/ieşiri C/C++

2.2. Objectives

- Understanding the stream concept
- Understanding the `printf()` and `scanf()` syntax
- Understanding the usage of `printf()` and `scanf()` format specifiers
- Understanding the extraction >> and insertion << operators' syntax
- Writing and testing some simple programs that use C/C++ I/O

2.3. Breviar teorectic

2.3.1. Flux (stream)

Sistemul de intrări-ieşiri din C/C++ operează prin stream-uri (fluxuri). Un stream (flux) este un dispozitiv logic, care fie produce, fie consumă informaţie. Altfel spus, reprezintă totalitatea modalităţilor de realizare a unor operaţii de citire sau scriere.

Transferurile cu dispozitivele periferice (consolă, disc, imprimantă) se fac prin intermediul fluxurilor şi prin intermediul sistemului de operare. În acest mod, detaliile funcţionale ale dispozitivelor periferice pot fi ignorate la nivelul programului.

Un flux poate fi de intrare, de ieşire sau bidirecţional.

Există fluxuri predefinite (standard) care se crează automat la lansarea în execuţie a unui program şi sunt închise automat la încheierea execuţiei programului.

2.3.2. Intrări/ieşiri C

Cele mai importante fluxuri predefinite (standard) din limbajul C sunt următoarele:

stdin: intrare standard, asociată consolei în intrare, adică tastaturii;

stdout: ieşire standard, asociată consolei în ieşire, adică ecranului;

În limbajul C, sistemul de intrări/ieşiri nu este o parte a limbajului, ci este adăugat printr-un set de funcţii din biblioteca standard. Funcţiile de intrare/ieşire pentru consolă utilizează implicit dispozitivele predefinite stdin (tastatura) şi stdout (ecranul), în mod transparent pentru programator.

2.3.3. Funcţia `printf()`

- permite formatarea şi afişarea de caractere şi valori către ieşirea standard, stdout.
- se apelează astfel: `printf(format[,argi]);`

unde

`format` este un şir de caractere care defineşte textele, secvenţele escape şi formatele de scriere a datelor precizate prin specificatori de format care se scriu în caz că există argi;

`argi` sunt argumente care trebuie să corespundă specificatorilor de format corespunzând specificatorului i, de forma:

```
% [indicator] [dimensiune_minimă_câmp] [.precizie_afişare] [size_format_
tip_dată] [h|l|L] tip
```

Datele gestionate de către `printf()` sunt supuse unor transformări din cauza existenţei unui format intern (ce depinde de tipul datelor) şi a altuia extern a datelor (uzual ASCII). Specificatorii de format definesc aceste conversii. Câmpul de tip size specifică dimensiunea argumentului consumat și convertit.

Pentru afișarea datelor de tip `wchar_t` se utilizează o funcția `wprintf()` - versiune a `printf()` definită in header-ul `<cwchar>`

Specificatorii de format încep totdeauna cu caracterul `%`. Formatele pentru datele specifice utilizate în `printf()` sunt:

`%c`	afişare caracter unic; valoarea lui este interpretată ca fiind codul ASCII al caracterului;
`%lc`	afişare caracter unic; valoarea lui este interpretată ca fiind codul ASCII sau Unicode al caracterului wide;
`%s`	afişare şir de caractere până la caracterul NULL;
`%ls`	afişare şir de caractere wide până la caracterul NULL;
`%d`	afişare număr întreg în baza zece cu semn;
`%i`	afişare număr întreg în baza zece cu semn;
`%u`	afişare număr întreg în baza zece fără semn realizând conversia unei date binare de tip unsigned în zecimal;
`%f(F)`	afişare număr real, notaţia zecimală realizând conversia datei de tip float sau double la forma, întreg.fractionar;
`%e(E)`	afişare număr real, notaţia exponenţială realizând conversia datei de tip float sau double la forma, întreg.fractionarE[+,-]exp;
`%g(G)`	afişare număr real, cea mai scurtă reprezentare dintre %f şi %e;
`%a(A)`	afişare număr real în hexazecimal
`%x(X)`	afişare număr hexazecimal întreg fără semn convertind datele int sau unsigned;
`%o`	afişare număr octal întreg fără semn convertind datele int sau unsigned;

%p	afişarea unui pointer la void (orice tip de dată) sub forma unei adrese compuse din segment şi offset în afişare cu cifre hexa;
%n	înscrie în variabila de tip întreg a cărei adresă e dată ca argument, numărul de caractere scrise deja in flux cu printf() pana la intalnirea lui %n (implicit dezactivat in Visual Studio)

În plus, mai pot fi utilizate următoarele prefixe:

l	cu d,i, o, u, x, X pentru date de tip long;
l, L	cu e, f, g, E, G, a, A pentru date de tip double;
h	cu d, i, o, u, x, X pentru date de tip short,

precum si alte prefixe introduse in versiunile ulterioare (z, ...).

Pentru a specifica dimensiunea câmpului de afişare şi precizia de afişare a datelor, în cadrul datelor, specificatorul de format va arăta astfel:

%[-,+]a.b f unde:

%	este semnul începutului specificatorului de format;
-	indică că alinierea are loc la stânga;
+	Valorile pozitive sunt precedate de +
a	precizează dimensiunea minimă a câmpului (inclusiv punctul care se scrie dacă este cazul);
b	dă precizia de afişare;
f	arată formatul tipului de dată specific, f/lf pentru real flotant/double, d pentru întreg, s pentru şir de caractere, etc.

Semnificația preciziei este:

d, i, o, x, X	se afişează minimum b digiţi, eventual completaţi la stânga cu 0 (implicit: b=1)
e, E, f, F, a, A	se afişează maximum b zecimale (implicit b=6)
g, G	se afişează maximum b cifre semnificative (implicit: toate cifrele semnificative)
s	se afişează maximum b caractere din şir (implicit: toate caracterele)

Semnificatia indicatorului este:

nimic	aliniere la dreapta și umplere la stânga cu spațiu sau 0
-	aliniere la stânga și umplere la dreapta cu spațiu
+	date cu semn prefixate de - sau +
spatiu	pentru datele negative se afișează - și spațiu pentru pozitiv
#	se folosește următorul format special:

`c,s,d,i,u`	fără efect
`o`	adaugă prefixul 0
`x` sau `X`	adaugă prefixul 0x, respectiv 0X
`e, E, f, a, A`	ia în considerare întotdeauna punctul zecimal
`g, G`	ca și `e, E`; nu elimină 0 final

Observații

1. precizia poate fi stabilită și folosind caracterul '*' după caracterul '.', valoarea preciziei fiind dată printr-un argument suplimentar al funcției printf(), care precede argumentul ce trebuie formatat.

2. Dimensiunea minimă poate fi specificată în mod similar, folosind caracterul '*' între % și specificatorul de format, valoarea dimensiunii minime a câmpului fiind specificată printr-un argument suplimentar plasat înaintea argumentului care va fi formatat.

2.3.4. Funcţia `scanf()`

permite introducerea de date tastate la terminalul standard de intrare stdin, date specificate de argumentele `argi` sub controlul unor formate.

se apelează astfel: `scanf(format [,argi]);`

unde,

`format` este un şir de caractere care dă formatele datelor şi eventual textele aflate la stdin; caracterele albe sunt neglijate; în rest sunt specificatori de format şi alte caractere care trebuie să existe la intrare în poziţiile corespunzătoare, caractere în general folosite la efectuarea de controale asupra datelor citite.

`argi`, sunt argumente care corespund adreselor zonelor în care se păstrează datele citite după ce au fost convertite din formatul lor extern în cel intern corespunzător, de forma:

`% format_ tip_ dată`

Specificatorii de format pentru `scanf()`:

`d`	întreg zecimal
`D`	întreg zecimal
`o`	întreg octal
`O`	întreg octal
`x`	întreg hexazecimal
`X`	întreg hexazecimal
`i`	întreg (d, o, x)
`I`	întreg (D, O, X)
`u`	întreg zecimal fără semn
`U`	întreg zecimal fără semn

`e, E, f, F, g, G` număr real

`c` caracter

`lc` caracter wide

`s` şir de caractere

`ls` şir de caractere wide

Între caracterul `%` şi literele specifice specificatorului de format se mai pot utiliza:

- un caracter `*` opţional, precizând faptul că ceea ce urmează a fi preluat şi interpretat este funcţie de tipul specificat, dar nu este memorat;
- un şir de cifre opţional care defineşte lungimea maximă a câmpului din care se citeşte data sub controlul formatului respectiv.

Funcţia `scanf()` citeşte toate câmpurile care corespund specificatorului de format, inclusiv eventuale texte prezente în câmpul format. În caz de eroare, citirea se opreşte, funcţia `scanf()` returnând numărul de câmpuri citite corect.

Înainte de citirea unui caracter sau wide caracter unele medii de dezvoltare, cum e Visual Studio solicita introducerea unui spatiu inainte de specificatorul de format (dacă nu e prima citire din program):

Exemplu:

```
char ca;
printf("\nIntroduceti un caracter : ");
scanf(" %c", &ca); /*un spatiu inainte de %, in special daca se fac alte citiri in prealabil*/
```

Noi elemente legate de operațiile de intrare/ieșire cu `printf()/scanf()` au fost introduse în `cstdio` și `cinttypes` (`inttypes.h`) în C++0x/1y/2z (`long long int` (long long int -`%lld` – pentru signed, unsigned long long int –`%llu` – pentru unsigned, etc.)), vedeți la adresele :

http:://www.cplusplus.com/reference/cstdio/printf/

http://www.cplusplus.com/reference/cstdio/scanf/

In C++0x/1y/2z pot fi utilizate expresii regulare pentru a permite citirea caracterelor, chiar dacă se întâlnesc caractere de spaţiere.

Exemplu:

```
char name[20]="";
scanf ("%[^\n]%*c", name);
```

unde:

`[^\n]` specifică citirea caracterelor până la sfârşit de linie.

`%*c` caracterul `*` indică faptul că se extrage din flux până la caracterul newline (deci nu va afecta citiri ulterioare), dar acesta nu va fi memorat.

Același efect îl obținem și cu construcția:

```
scanf("%[^\n]s",name);
```

`\n` - setează delimitatorul pentru şirul citit

2.3.5. Microsoft VC++ `scanf_s()` și alte funcții asociate

`scanf_s()` citește date formatate de la intrarea standard. Există mai multe versiuni ale lui `scanf_s()`, `_scanf_l()`, `wscanf()`, `_wscanf_l()` (ultimele două pentru citire date de tip `wchar_t`) care au îmbunătățiri de securitate, cum e descris în "Security Features în the CRT".

Sintaxa:

```
int scanf_s( const char *format [,argument]... );
int _scanf_s_l(const char *format, locale_t locale [,  argument]... );
```

unde:

`format`	reprezintă șirul de control al formatului
`argument`	identifică argumentele
`locale`	reprezintă eventualele variabile `locale` de folosit

Exemplu:

```
const int DIM=10;
char s[DIM];
scanf_s("%9s", s, (unsigned)_countof(s)); /* buferul are dimensiunea 10,
numarul maxim de caractere citite este 9,  unde _countof(s) este in
<stdlib.h> */
```

```
#define _CRT_SECURE_NO_WARNINGS
```

Directiva e folosită pentru a considera funcția standard C/C++ `scanf()`, fără a avea mesaje de incompatibilitate. În acest mod se asigură compatibilitatea între mai multe platforme ce folosesc C/C++.

2.3.6. Intrări/ieşiri C++

Limbajul C++ conţine toate rutinele din biblioteca de intrări/ieşiri a limbajului C, dar ne pune la dispoziţie şi un sistem propriu de intrări/ieşiri orientat pe obiecte, implementat prin aşa numita bibliotecă <`iostream`>.

În limbajul C++ sunt predefinite dispozitivele logice de intrare/ieşire standard similare celor din C (prin intermediul unor obiecte statice):

- `cin` (console input): dispozitiv de intrare consolă, asociat tastaturii (echivalentul lui stdin);
- `cout` (console output): dispozitiv de ieşire consolă, asociat monitorului (echivalentul lui stdout).

Transferul informaţiei cu formatare se poate face cu operatori (fac parte din limbaj) specializaţi:

>> (extracţie) pentru intrare (de la cin): `cin >> var;`

<< (inserţie) pentru ieşire (către cout): `cout << var;`

Sunt posibile operaţii multiple:

```
cout << var1 << var2...<< varn ;
cin >> var1 >> var2...>> varn ;
```

chiar dacă variabilele implicate au tipuri diferite.

Utilizarea dispozitivelor şi operatorilor de intrare/ieşire C++ impune includerea fişierului antet iostream și a spatiului de nume std, (în alte medii `iostream.h`).

```
#include <iostream>
using namespace std;
```

În acest mod nu mai e necesară includerea bibliotecilor specifice din limbajul C, ele fiind incluse în spațiul de nume standard, cu unele excepții.

Aceşti operatori nu necesită specificatori de format pentru fiecare tip de data, deoarece se foloseşte un format implicit. Particularizarea formatului este posibilă, dar este diferită faţă de lucrul cu funcţiile `printf()`/`scanf()`.

2.4. Theoretical brief

2.4.1. Streams

The C/C++ input-output system operates via streams. A stream is a logical device that either produces or consumes information. In other words, it represents the totality of ways of performing reading or writing operations.

Data transfers with peripheral devices (console, disk, printer) are made through streams in correlation with the operating system. This way, the functional details of peripheral devices can be ignored at the program level.

A data stream can have input, output, or bidirectional type.

There are predefined (standard) streams that are created automatically when a program is launched and are automatically closed at the end of program execution.

2.4.2. C input/output

The most important predefined (standard) streams in C are the following ones:

stdin: standard input, associated with the console, a.k.a. the keyboard;

stdout: Standard output, associated with the display, a.k.a. the screen;

In the C language, the I/O system is not a part of the language and was added through a set of functions that reside in the standard library. The console input/output functions use stdin (keyboard) and stdout (screen) devices by default.

2.4.3. The `printf()` function

- Allows formatting and displaying characters and values to the standard output, stdout.
- It is called as follows: `printf(format[,argi]);`

where

`format` indicates a string defining texts, escape sequences and data writing format specifiers to be written in case `argi` exists;

`argi`, are arguments that must match the format specifiers corresponding to specifier i, like:

```
% [indicator] [min_size] [.displaying_precision] [size_format_data_type]
[h|l|L] type
```

The data managed by `printf()` is subject to transformation due to the existence of an internal format (which depends on the data type) and another external format (usually ASCII). The format specifiers define these conversions. The size field specifies the size of the argument consumed and converted.

For displaying the `wchar_t` data the `wprintf()` function is used - a `printf()` variant defined in the header file `<cwchar>`

The format specifiers always begin with the `%` character The specific formats used by `printf()` are:

`%c`	displays a unique character; its value is interpreted as the ASCII code of the character;
`%lc`	displays a unique character; its value is interpreted as the ASCII or Unicode code of the wide character;
`%s`	displays a string until the NULL character is encountered;
`%ls`	displays a wide string up to NULL character;
`%d`	displays a signed base 10 integer;
`%i`	displays a signed base 10 integer;
`%u`	displays an unsigned base 10 integer starting from an unsigned binary value;
`%f(F)`	displays a real value;
`%e(E)`	displays a real value by representing a float or a double number as E[+,-]exp;
`%g(G)`	displays a real value, using the shortest representation between %f and %e;
`%a(A)`	displays a real value in hexadecimal;
`%x(X)`	displays a signed/unsigned integer in hexadecimal;

%o	displays a signed/unsigned integer in octal;
%p	displays a generic pointer (indicating any data type) as an address composed of a segment and an offset, using hexadecimal digits;
%n	writes in the integer variable whose address is given as argument the number of characters written until %n is met (disabled by default in Visual Studio)

In addition, the following prefixes can be used:

l	with d,i, o, u, x, X for long variables;
l, L	with e, f, g, E, G, a, A for double data;
h	with d, i, o, u, x, X for short data,

as well as other prefixes introduced in later versions (z, ...).

To specify the display field size and accuracy of the displayed value, within the data, the format specifier will look like this:

%[-,+]a.b f where,

%	is the beginning indicator of the format specifier;
-	indicates a left align;
+	positive values are preceded by +
a	specifies the minimum size of the displaying field (including the point to be written, if applicable);
b	indicates the displaying precision;
f	shows the specific data type format, f/lf for real float/double, d for integer, s for character array, etc.

The significance of precision:

d, i, o, x, X	at least b digits are displayed, possibly left filled with 0 (implicit: b=1)
e, E, f, F, a, A	maximum b decimal places are displayed (implicit b=6)
g, G	maximum b significant digits are displayed (implicit: all the significant digits)
s	maximum b characters from the string are displayed (implicit: all the characters)

The significance of the indicator is:

nothing	right align and left fill with Space
-	left align and right fill with Space
+	Signed data prefixed with – or +
space	for negative values display – and space for positive

#	The following special format shall be used:
	`c,s,d,i,u` no effect
	`o` add prefix "0"
	`x,X` add 0x and 0X respectively
	`e, E, f, a, A` always consider the decimal point
	`g, G` like e, E; does not remove 0 at the end

Observations

1. Precision can be determined using the character '*' after the character '.', the precision value being given by an additional argument of the printf() function, which precedes the argument to be formatted.

2. The minimum size can be specified similarly, using the character '*' between % and the format specifier, the minimum field size value being specified by an additional argument placed before the argument to be formatted.

2.4.4. `scanf()` function

Allows data input typed at the standard input terminal stdin, data specified by argi arguments, being controlled by some format specifiers.

It is called like this: `scanf(format [,argi]);`

where

`format` is a string of characters giving data formats and, possibly, texts at stdin; white spaces are neglected; otherwise, there are format specifiers and other characters that must exist at the input in the appropriate positions, characters generally used to verify the read data.

`argi` are arguments that correspond to the addresses of the areas where the read data is kept, after being converted from their external format to the corresponding internal one, like:

```
% data_type_format
```

Format specifiers for `scanf()`:

`d`	decimal integer
`D`	decimal integer
`o`	octal integer
`O`	octal integer
`x`	hexadecimal integer
`X`	hexadecimal integer
`i`	integer (d, o, x)
`I`	integer (D, O, X)
`u`	unsinged decimal integer
`U`	unsinged decimal integer

`e, E, f, F, g, G` real number

`c` character

`lc` wide character

`s` array of characters

`ls` array of wide characters

Between `%` and the letters of the format specifier, the following can be used:

- an `*` optional character, specifying that what is to be retrieved and interpreted depends on the specified type, but is not memorized;

- an optional array of digits defining the maximum length of the field from which the data is read.

The `scanf()` function reads all fields corresponding to the format specifier, including any other text found in the format field. In case of an error, the reading stops, with the scanf() function returning the number of correctly read fields.

Before reading a character or wide character, some development environments, such as Visual Studio, require inserting a space before the format specifier (if is not the first reading in the program):

Example:

```
char ca;
printf("\nEnter a character : ");
scanf(" %c", &ca); /*a space before %, especially if other readings have been already performed */
```

New elements related to input/output operations with `printf()`/`scanf()` have been introduced in `cstdio` and `cinttypes` (`inttypes.h`) in C++0x/1y/2z (`long long int` (long long int -`%lld` – for signed, unsigned long long int -`%llu` – for unsigned, etc.)). Please go to:

http:://www.cplusplus.com/reference/cstdio/printf/

http://www.cplusplus.com/reference/cstdio/scanf/

In C++0x/1y/2z regular expressions can be used for reading characters, even if spacing characters are encountered.

Example:

```
char name[20]="";
scanf ("%[^\n]%*c", name);
```

Where:

`[^\n]` – indicates reading all the characters until the end of the line.

`%*c` – character * indicates that the newline character is extracted from the stream (so it won't affect further readings), but it will not be memorized.

The same effect is obtained with the syntax:

```
scanf("%[^\n]s",name);
```

`\n` - sets the delimiter for the read string

2.4.5. Microsoft VC++ `scanf_s()` and other related functions

`scanf_s()` reads formatted data from standard input. There are several versions of `scanf_s()`, `_scanf_l()`, `wscanf()`, `_wscanf_l()` (the last two for reading wchar_t data) that have security enhancements, as described in "Security Features in the CRT".

Syntax:

```
int scanf_s( const char *format [,argument]... );
int _scanf_s_l(const char *format, locale_t locale [,  argument]... );
```

Parameters:

- `format`, format specifier.
- `argument`, argument.
- `locale`, local variables to be used

Example:

```
const int DIM=10;
char s[DIM];
scanf_s("%9s", s, (unsigned)_countof(s)); /* the buffer has the size 10,
maximum 9 characters to be read, where _countof(s) is from <stdlib.h> */
```

```
#define _CRT_SECURE_NO_WARNINGS
```

The directive is used to consider the standard C/C++ function scanf(), without incompatibility messages. This ensures compatibility between multiple platforms using C/C++.

2.4.6. C++ Input/Output

The C++ language contains all the routines in the I/O library of the C language, but it also provides its own object-oriented I/O system, implemented in the so-called library iostream.

In C++ the logical standard Input/Output devices are defined (analogous to C, by static objects):

- `cin` (console input): console input device, associated with the keyboard (equivalent to stdin);

• `cout` (console output): console output device, associated with the monitor (equivalent to stdout).

The transfer of formatting information can be done with specialized operators:

>> (extraction) for input: `cin >> var;`

<< (insertion) for output: `cout << var;`

Multiple operations are possible:

```
cout << var1 << var2...<< varn ;
cin >> var1 >> var2...>> varn ;
```

even if the variables have different types.

Using C++ input/output devices and operators requires the inclusion of the `<iostream>` header file and `std` namespace, (in other environments `<iostream.h>`).

```
#include <iostream>
using namespace std;
```

In this way, it is no longer necessary to include specific C libraries, they are included in the standard namespace, with some exceptions.

The reading/writing operators don't require format specifiers for each data type, because a default format is used. Customizing the format is possible, but it's different from working with `printf()`/`scanf()`.

2.5. Exemple/ Examples

2.5.1. Ex. 1

```
/* Program that displays an integer in decimal, octal, and hexadecimal format */

// Preprocessor inclusion directives
#include <stdio.h>

// symbolic constant
#define V 12345
//constexpr auto V1 = 12345;
constexpr auto length = 7;

int main(){
	printf("\n Decimal: %d", V);
	printf("\n Octal: %#o", V);
	printf("\n Hexadecimal: %#x", V);
	printf("\n Displaying an integer in a field with the width length %d:%*d",
length, length, V);
	return 0;
} //end_main
```

2.5.2. Ex. 2

```
/* program that reads a character and a wide character using the function
scanf()/wscanf(); displays the ASCII code of the character in decimal, hexadecimal
and the size occupied in the memory */

//ASCII  code for characters, wide_characters
#define _CRT_SECURE_NO_WARNINGS
//#include <cstdio>
#include <cwchar> // Header file containing wide functions and types

int main( ){
       char car;
       wchar_t lcar;
       printf("\nEnter a character: ");
       scanf(" %c", &car);
       //scanf_s(" %c", &car,1);
       printf("\n The ASCII code of the character:\n \tdecimal: %d\n\thexa: %x, \t
size= %zd\n", car, car, sizeof(char));
       wprintf(L"\nEnter a wide character: ");
       wscanf(L" %lc", &lcar);
       //wscanf_s(L" %lc", &lcar,2);
       wprintf(L"\n The ASCII code of the wide character:\n \tdecimal: %d\n\thexa:
%x, \t size= %zd\n", lcar, lcar, sizeof(lcar));
       return 0;
}//end_main
```

2.5.3. Ex. 3

```
/* program that displays the Pi value in different formats */
// preprocessor inclusion directives

#ifndef _USE_MATH_DEFINES
#define _USE_MATH_DEFINES
#endif

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

int main( ){
 printf("\nPi=%f",M_PI);   // Pi=3.141593
 printf("\nPi=%-10f",M_PI);// Pi=3.141593, 10 positions, left alignment
 printf("Pi=%10f",M_PI);   // Pi=  3.141593, on 10 positions, right alignment
 printf("\nPi=%e",M_PI);   // Pi=3.141593e+00 (scientific notation)
 printf("\nPi=%g",M_PI);   // Pi=3.14159, default display on 6-digit
 printf("\nPi=%06.2f",M_PI);      // Pi=003.14, display on 6 positions, 2 //decimal
                                  places and left fill with zeros
 printf("\nPi=%10.*f",2 ,M_PI);   // Pi=      3.14, on 10 positions and //precision
                                  2, right alignment
 printf("\nPi=%a",M_PI);  // Pi=0x1.921fb5p+1, display in hexadecimal with
                                  //lowercase letters
printf("\nPi=%.4A\n",M_PI);  // Pi=0X1.9220P+1, display in hexadecimal //with
uppercase letters, precision 4
}//end_main
```

2.5.4. Ex. 4

```
/* Using the %n specifier and displaying the address of a variable using %p */

#include<stdio.h>

int main( ) {
   int i;
   _set_printf_count_output( 1 ); // enable %n by calling the function with //a
parameter different than 0
   printf( "12345%n6789\n", &i );// i receives the number of characters //already
written on the screen with printf() up to %n,
//meaning  i=5. Obs: i has its address specified
   printf( "i = %d\n", i );     // display the value of i
   printf("Adresa lui i=%#p\n\n", &i); //%p for address display and # for 0X
//before the hexadecimal value
   return 0;
}//end_main
```

2.5.5. Ex. 5

```
/* program that reads a float and a string using scanf_s() in Microsoft VC++ */

//#define _CRT_SECURE_NO_WARNINGS
#include <cstdio>
#include <cstdlib> //_countof

const int MAX=20;
//const int MAX=5;

int main( ) {
       char buff[MAX]; //fixed size
       float a;
       printf("\nEnter a float and a character string:");
       scanf_s("%f", &a);
       scanf_s("%s", buff, _countof(buff));     //sizeof(buff) ?
       // scanf("%s", buff);
       printf("\nFloat is: %6.2f \n", a);
       printf(buff);
       return 0;
} //end_main
```

2.5.6. Ex. 6

```
/* Program that reads the name, surname and year of birth of a person separated by
white spaces, then displays the information obtained on a single line
C variant
*/

#define _CRT_SECURE_NO_WARNINGS
```

```
#include <cstdio>
const int MAX =255;

int main( ){
	char name[MAX], surname[MAX];
	int year;
	printf( "\nEnter name: ");
	scanf("%s", name);
	printf("\nEnter surname: ");
	scanf("%s", surname);
	printf("\nEnter birth year: ");
	scanf("%d", &year);
	printf("\nPerson data are: %s, %s, %d\n", name, surname, year);
	return 0;
}//end_main
```

2.5.7. Ex. 7

```
/* Program that reads the name, surname and year of birth of a person separated by
white spaces, then displays the information obtained on a single line
C++ variant
*/
```

a)

```
// Preprocessor inclusion directives – simple version
#include <iostream>
using namespace std;
const int MAX =255;

int main( ){
	char name[MAX], surname[MAX];
	int year;
	cout << "\nEnter name: ";
	cin >> name;
	cout << "\nEnter surname: ";
	cin >> surname;
	cout << "\nEnter birth year: ";
	cin >> year;
	cout <<'\n' <<"Person data are: "<<name<<"  "<<surname << ", " << year;
	cout <<endl ;//will flush the buffer and new line
}//end_main
```

b)

```
//Wide character strings
#include <iostream>
using namespace std;
const int MAX = 255;

int main( ) {
wchar_t wname[MAX], wsurname[MAX];
int year;
wcout << L"\nEnter name - wide string: ";
wcin >> wname;
wcout << L"\nEnter surname - wide string: ";
```

```
wcin >> wsurname;
wcout << L"\nEnter birth year: ";
	cin >> year;
	wcout << endl << L"Person data are: " << wname << "   " << wsurname << ", "
<< year <<endl;
}//end_main
```

2.5.8. Ex. 8

```
/* program that uses regular expressions to read strings with spaces and other
types of data */

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#define DIM 20

int main( ) {
	int mark1, mark2;
	char name[DIM] = "";
	printf("\nEnter a string with space: ");
	scanf("%[^\n]%*c", name);
	printf(name);
	printf("\nEnter a new string with space: ");
	scanf("%[^\n]s", name);
	printf(name);

	printf("\nEnter an int: ");
	scanf("%d", &mark1);
	printf("\nThe mark is = %d ", mark1);
	printf("\nEnter another int: ");
	scanf("%d", &mark2);
	printf("\nThe mark is = %d ", mark2);

	printf("\nEnter a new string with space: ");
	cin.ignore( );//to empty the buffer
	scanf("%[^\n]s", name);
	printf(name);
	printf("\nEnter a new string without space: ");
	cin.ignore( );//to empty the buffer
	scanf("%s", name);
	printf(name);
}//end_main
```

2.5.9. Ex. 9

```
/* reading/displaying data of type wchar_t  using Microsoft VC++*/

#include <cwchar> // for wide functions and types
#include<stdlib.h>

int main( ) {
	wchar_t ch;
	wchar_t name[ ] = L"Students";
	wchar_t message[30];
```

```
	printf("\nEnter a wide character: ");
	wscanf_s(L" %lc", &ch, sizeof(ch));    //stdlib
	wprintf(L"character = %lc \n", ch);

	wprintf(L"HELLO %ls \n", name);
	printf("\nEnter a message: ");
	wscanf_s(L"%ls", message, (unsigned)_countof(message));
	wprintf(L"Message: %ls \n", message);
}//end_main
```

2.5.10. Ex. 10

```
/* program that reads a calendar date in the form of: dd mm yy and displays it as:
yy/mm/dd. */

// preprocessor inclusion directives
#include <iostream>
using namespace std;

#include <iostream>
using namespace std;

int main( ){
	int day, month, year;
	cout << "\nEnter a date (dd mm yy): ";
	cin >> day >> month >> year;
	cout << '\n' << year << '/'<< month <<'/'<< day << '\n';
}//end_main
```

2.5.11. Ex. 11

```
/* Program that uses auto type variables */

#include <iostream>
using namespace std;

int main( ) {
	auto x = 4; // x -> int
	auto y = 3.37; // y -> double
	auto z = 'x'; // z -> char
	cout << "\n The initial values of the variables: \t x= " << x << " y= " << y
<< " z= " << z << '\n';
	cout << "\n Enter 3 variables (int, double, char): \n";
	cin >> x >> y >> z;
	cout << "\n The new values are: \t x= " << x << " y= " << y << " z= " << z
<< endl;
}//end_main
```

2.6. Întrebări

1. Ce este un flux (stream) ?
2. Care sunt cele mai importante fluxuri în limbajul C ?

3. Care sunt cele mai importante fluxuri în limbajul C++ ?
4. Care sunt principalele funcţii ce permit efectuarea operaţiilor de intrare/ieşire în limbajul C ?
5. Care sunt operatorii ce permit efectuarea operaţiilor de intrare/ieşire în limbajul C++?

2.7. Questions

1. What is a data stream?
2. Which are the most important streams in the C language?
3. Which are the most important streams in the C++ language?
4. Which are the main functions that allow I/O operations in the C language?
5. Which are the operators that allow I/O operations in the C++ language?

2.8. Lucru individual

1. Realizaţi o aplicaţie care citeşte de la intrarea standard două valori pentru variabilele r1 şi r2 (întregi) - reprezentând valori de rezistenţe, apoi apelează funcţii ce calculează rezistenţa echivalentă grupării serie, respectiv grupării paralel, după care afişează valorile returnate de funcții, cu 3 zecimale şi aliniere la dreapta într-un câmp de dimensiune 10.
2. Scrieţi o aplicaţie care citeşte de la intrarea standard două valori pentru variabilele c1 şi c2 (întregi) - reprezentând valori de capacitati, apoi apelează funcţii ce calculează capacitatea echivalentă grupării serie, respectiv grupării paralel, după care afişează valorile returnate de funcții, cu 4 zecimale şi aliniere la stânga, într-un câmp de dimensiune 10, una după alta pe acelaş rând.
3. Citiţi de la tastatură două valori întregi care reprezintă catetele unui triunghi dreptunghic. Determinaţi ipotenuza şi perimetrul triunghiului. Afişaţi rezultatul pe linii diferite.
4. Citiţi de la tastatură două valori întregi a şi b (a diferit de 0), unde a şi b sunt coeficienţii ecuaţiei ax+b=0. Rezolvaţi ecuaţia şi afişaţi rezultatul.
5. Considerând că în problemele 1 și 2 că valorile întregi sunt rezistente și capacităţi, calculaţi valorile corespunzătoare grupării serie şi paralel funcție de opțiunea specificată în program.
6. Se citesc de la tastatură numele a 2 studenți și nota fiecăruia la programare. Să se afișeze pe linii separate numele fiecărui student, într-un câmp de afișare cu dimensiunea 20, aliniat la dreapta, respectiv la stânga, și media notelor lor, cu o precizie de 2 zecimale la partea fracționară.
7. Se citește de la tastatură un număr întreg, ce reprezintă raza unui cerc. Să se afișeze lungimea și aria cercului de rază dată, cu o precizie de 3 zecimale, într-un câmp de afișare cu dimensiunea 10. Pentru valoarea lui PI definiți o constantă simbolică, sau folosiți M_PI din <math.h>
8. Se citesc de la tastatură ora plecării unui tren din Cluj-Napoca și ora la care ajunge la Brașov (oră și minute). Să se calculeze și afișeze durata călătoriei (în ore și minute).
9. Două maşini se deplasează una spre cealaltă cu vitezele v1 şi v2, plecând din două oraşe situate la distanţa d=100 km. Vitezele celor două mașini se citesc de la tastatură. Afișați timpul după care se întâlnesc, exprimat în minute.

2.9. Individual work

1. Implement an application that reads from the standard input 2 values for 2 resistors (integers) identified with the r1 and r2 variables. The program calls 2 functions that calculate the series and parallel equivalent resistance. After that, it displays the results right aligned and with 3 digits precision in the fractional part in a field with the length 10.

2. Implement an application that reads from the standard input 2 values for 2 capacitors identified with the c1 and c2 variables (integers). The program calls 2 functions that calculate the series and parallel equivalent capacity. After that, it displays the results left aligned and with 4 digits precision in the fractional part, in a field with the length 10, one by one on the same row.
3. Read from the standard input 2 integer values that represent the catheters of a rectangular triangle. Determine the hypotenuse and the perimeter of the triangle. Display the results on different rows.
4. Read from the keyboard 2 integer values, a and b (a different from 0), representing the coefficients of the equation ax+b=0. Solve the equation and display the result.
5. Implement the 1-st and 2-nd problems in a single program. Make use of all the possible similarities.
6. Read from the keyboard the names and the marks of 2 students (each student has a single mark). Display, on separate lines, the name of each student (right aligned the first and left aligned the second) in two 20 characters fields and their marks with 2 digits precision at the fractional part.
7. Read from the keyboard an integer number that represents the radius of a circle. Display the length and area of the circle with 3 digits precision, in a field that can store 10 characters. Use a symbolic constant for the value of PI, or use M_PI from <math.h>.
8. Read from the keyboard a train's exact departure time from Cluj-Napoca (hour and minutes) and arrival time in Brașov. Calculate and display the travelling time, represented in hours and minutes.
9. Two cars move towards each other, having the speeds v1 and v2 and starting from two cities 100 km apart. The speeds' values are read from the keyboard. Display how many minutes will pass until the two cars meet in the same point.

3. Aplicaţii folosind operatori şi expresii C/C++

Applications using C/C++ operators and expressions

3.1. Obiective

- Înțelegerea noțiunilor de expresie, operanzi și operatori
- Trecerea în revistă a principalilor operatori şi reținerea celor mai importanți
- Înțelegerea modului în care se face evaluarea expresiilor
- Scrierea și rularea de programe simple care folosesc operatori

3.2. Objectives

- Understanding the concept of expression, operand and operator
- Reviewing the main operators and remembering the most important ones
- Understanding the expression evaluation
- Writing and running some simple applications that use operators

3.3. Breviar teoretic

Operatorii sunt simboluri ce specifică operaţiile de efectuat asupra operanzilor. În urma aplicării unui operator se obţine un rezultat.

Operatorii limbajului C se pot grupa pe clase astfel:

Operatori aritmetici

unari : + (fără efect), - (negativare)

binari :

multiplicativi

*	înmulţire ;
/	împărţire, dacă operanzii sunt întregi rezultatul va fi partea întreagă a câtului;
%	modulo, doar pentru operanzi întregi şi are ca rezultat restul împărţirii întregi;

aditivi

+	adunare
−	scădere

Operatori de relaţie şi de egalitate

operatori de relaţie, care sunt: <, <=, >, >=,

operatori de egalitate, care sunt:

== egal

!= diferit.

Operatori logici

! (**not**) negaţia logică, care se utilizează astfel:

!operand = 0 (False), dacă operand != 0

!operand = 1 (True), dacă operand = 0.

&& (**and**) şi logic,

E1 && E2, şi are valoarea 1 (True) dacă ambele expresii E1 şi E2 sunt True (!=0), în rest are valoarea False (0).

|| (**or**) sau logic,

E1 || E2 care are valoarea 0 (False) dacă ambele expresii E1 şi E2 au valoarea 0 (False). În rest expresia are valoarea 1 (True).

Operatori pe biţi

~ (**compl**) complement faţă de unu, schimbă fiecare bit 1 al operandului în 0, respectiv fiecare bit 0 în 1;

<< deplasare la stânga a valorii primului operand cu un număr de poziţii egal cu valoarea celui de-al doilea operand, forma de utilizare fiind: `op1 << op2`; este echivalent cu înmulţirea cu puteri ale lui 2;

>> deplasare la dreapta a valorii primului operand cu un număr de poziţii egal cu valoarea celui de-al doilea operand, forma de utilizare fiind: `op1 >> op2;` este echivalent cu împărţirea cu puteri ale lui 2;

& (**bitand**) şi logic pe biţi, se execută bit cu bit conform lui ŞI-LOGIC; este folosit la lucrul cu măşti;

^ (**xor**) sau exclusiv pe biţi, se execută bit cu bit conform lui SAU-EXCLUSIV; este folosit pentru a anula sau seta diferiţi biţi.

| (**bitor**) sau logic pe biţi, se execută bit cu bit conform lui SAU-LOGIC; este folosit pentru a seta diferiţi biţi.

Operatori de atribuire (asignare)

asignare simplă, care se notează prin caracterul = şi are forma: v=(expresie) unde v este o variabilă simplă, referinţă la un element de tablou sau de structură. Se asociază de la dreapta la stânga astfel: v_n=...=v_1=v=expresie; asignarea făcându-se mai întâi v=expresie, apoi se asociază rezultatul lui v_1, apoi v_2 şi tot aşa până la v_n.

asignare compusă, caz în care pentru atribuire se foloseşte forma: operator=, unde operator poate fi unul binar aritmetic sau unul logic pe biţi, deci poate fi unul dintre operatorii următori: /, %, *, -, +, <<, >>, &, ^, | .

Sintaxa: var operator= expresie;

este echivalentă cu: var = var operator (expresie);

Remarcă: În versiunile inițiale, asignarea compusă era permisă și postfix. Când compilatorul acceptă atât =- cât și -=, unele compilatoare au adăugat un mesaj de avertizare pentru vechiul operator, ceva

de genul: „Operatorul =- este depreciat. Utilizați -=”, alte compilatoare ignoră operatorul postfix compus.

Operatori de incrementare şi decrementare (operatori unari). Aceştia sunt ++ şi --, putând fi postfixaţi (după operand), respectiv prefixaţi (înaintea operandului).

Operator de forţare a tipului sau de conversie explicită (cast)

`(tip)` operand prin care valoarea operandului se converteşte la tipul indicat în C sau tip (operand) în C++.

Operator de determinare a dimensiunii

`sizeof(data)` sau `sizeof(tip)`, determinând dimensiunea în octeţi a unei date sau a unui tip;

Operatori de adresare (&, referenţiere) şi de indirectare (*, dereferenţiere)

Operatori paranteză

parantezele rotunde, ( )

paranteze pătrate, [ ]

Operator condiţional

`E1 ? E2 : E3`, unde E1, E2, E3 sunt expresii.

Operator virgulă

leagă două expresii în una singură astfel: `E1, E2`

Alţi operatori (pt. structuri)

punct .

săgeată ->

C++ defineşte cuvintele cheie `and, or, not, xor, and_eq, or_eq, not_eq, xor_eq`, etc. ca alternativa pentru `&&, ||, !, ^, &=, |=, != and ^=`, numite, “digraphs”, care sunt rareori utilizate!

(https://en.cppreference.com/w/cpp/language/operator_alternative)

O expresie poate folosi operatori din clase diferite, la evaluarea ei ţinându-se cont de:

- precedenţă (prioritate): dă ordinea de efectuare a operaţiilor;

- asociativitate: indică ordinea de efectuare a operaţiilor care au aceeaşi precedenţă; poate fi stânga->dreapta (S->D) sau dreapta->stânga (D->S);
- regula conversiilor implicite: asigură stabilirea unui tip comun pentru ambii operanzi, la fiecare operaţie care solicită acest lucru şi în care tipurile diferă; regula de bază: se promovează tipul cu domeniu de valori mai mic către tipul cu domeniul de valori mai mare;

Ordinea de evaluare dată de precedență şi asociativitate poate fi modificată grupând operaţiile cu ajutorul parantezelor.

3.4. Theoretical brief

Operators are symbols that specify the operations to be performed on operands. Following the application of an operator, a result is obtained.

C language operators can be grouped into classes as follows:

Arithmetic operators

unary: + (no effect), - (negation)

binary:

multiplication

* multiplication ;

/ division, if the operands are integer, the integer quotient will be returned;

% modulo, only for integer operands, obtains the remainder of the integer division;

addition

+ addition

- subtraction

Relational and equality operators

relational operators: <, <=, >, >=,

equality operators:

== equal

!= different

Logical operators

! (**not**) logical negation, which is used as it follows:

!operand = 0 (False), if operand != 0

!operand = 1 (True), if operand = 0.

&& (**and**) logical and,

E1 && E2 is evaluated to 1 (True) if both expressions E1 are E2 are True (!=0), otherwise False (0).

|| (**or**) logical or,

E1 || E2 is evaluated to False (0) if both expressions E1 and E2 are 0 (False). otherwise 1 (True).

Bitwise operators

~ (**compl**) complement, changes every bit of the operand from 1 to 0 and from 0 to 1;

<< (left shift), shifts to the left the value of the first operand by a number of positions equal to the value of the second operand, being used as: `op1 << op2;` is equivalent to multiplying by powers of 2;

>> (right shift), shifts to the right shift the value of the first operand by a number of positions equal to the value of the second operand, being used as: `op1 >> op2;` is equivalent to dividing by powers of 2;

& (**bitand**), bitwise AND applies a logical AND upon the corresponding bits of the operands; it is frequently used when working with masks;

^ (**xor**), exclusive OR, applies a logical XOR upon the corresponding bits of the operands; is used to cancel or set various bits.

| (**bitor**), bitwise OR, applies a logical OR upon the corresponding bits of the operands; is used to set various bits.

Assignment operators

simple assignment, which is denoted by the character = and has the form: v=(expression) where v is a simple variable, reference to an array or structure element. It is associated from right to left: v_n=…=v_1=v=expression; assigning first v=expression, the results are associated to v_1, then v_2, etc. until v_n.

compound assignment, in which a distinct operator is associated with the assignment operator (operator=), where the operator can be binary arithmetic or binary bitwise, so it can be one of the following operators: /, %, *, -, +, <<, >>, &, ^, | .

The syntax: var operator= expression;

is equivalent to: var = var operator (expression);

Remark: Early C versions allowed postfix compound assignment. When the compiler would accept both =- and -=, some compilers added a warning message for the old operator, something like: “Operator =- is deprecated. Use -=”, others ignore the compound postfix operator.

Increment and decrement (unary) **operators**. These are ++ and --, can be postfixed (after the operand), respectively prefixed (before the operand).

Type forcing operator (explicit conversion, cast)

`(type)operand`, by which the value of the operand is converted in C to the specified type, or type(operand) in C++.

Size determining operator

`sizeof(data)` or `sizeof (type)`, determining the size in bytes of a variable or type;

Addressing (&, referencing) and indirection (*, dereferencing) operators

Parenthesis operators

round parenthesis, ( )

square parenthesis, [ ]

Conditional operator

`E1 ? E2 : E3`, where E1, E2, E3 are expressions

Comma operator

links two expressions into a single one, like this: `E1, E2`

Other operators (used for structures)

point .

arrow ->

C++ defines the keywords `and, or, not, xor, and_eq, or_eq, not_eq, xor_eq`, etc. as an alternative for `&&, ||, !, ^, &=, |=, != and ^=`, named, “digraphs” (which are rarely used)

(https://en.cppreference.com/w/cpp/language/operator_alternative)

An expression may use operators from different classes, being evaluated considering:

- the precedence (priority): reflects the order of performing operations;
- associativity: indicates the order in which operations having the same precedence are performed; can be left->right or right->left;
- Implicit conversions rule: ensures that a common type is established for both operands, each time the situation imposes it; Basic rule: the type with the smaller value range is promoted to the type with the larger value range;

The order of evaluation given by precedence and associativity can be modified by grouping the operations with round parentheses.

3.5. Exemple / Examples

3.5.1. Ex. 1

```
/* conditional operator */

#include <iostream>
using namespace std;

int main ( ){
	double n1, n2, n3;
	cout << "Enter 3 real numbers: ";
	cin  >> n1 >> n2 >> n3;
	cout << (n1 <= n2 && n2 <= n3 ? "Are ordered" : "Are not ordered") << '\n';
	return 0;
}//end_main
```

3.5.2. Ex. 2

```
/* logical and relational operators
The program checks if the character entered is an uppercase or lowercase letter */

#include <iostream>
using namespace std;

int isAlpha (char ch);//prototype

int main( ){
	char ch;
	cout << "Enter a character: ";
	cin >> ch;
	cout << (isAlpha(ch) != 0 ? "Is a letter" : "Is not a letter") << '\n';
	//cout << (isAlpha(ch) ? "Is a letter" : "Is not a letter") << '\n';
	return 0;
}//end_main

int isAlpha (char ch){
	return ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z';
}// end_isAlpha
```

3.5.3. Ex. 3

```
/* bitwise operators
The program applies the bitwise operators and displays the results in octal */

#include <iostream>
using namespace std;

int main( ){
	unsigned char x = '\011';
	unsigned char y = '\027';
	cout << "x = " << oct << short(x) << '\n';
	cout << "y = " << oct << short(y) << '\n';
	cout << "~x = " << oct << (short)(~x) << '\n';
	cout << "x & y = " << oct << (short)(x & y) << '\n';
	cout << "x | y = " << oct << (short)(x | y) << '\n';
	cout << "x ^ y = " << oct << (short)(x ^ y) << '\n';
```

```
	cout << "x << 2 = " << oct << (short)(x << 2) << '\n';
	cout << "x >> 2 = " << oct << (short)(x >> 2) << '\n';
	return 0;
}//end_main
```

3.5.4. Ex. 4

```
/* logical operators in literal representation */

#include <iostream>
#include <bitset>// display bitwise representation from STL
using namespace std;

int main( ) {
int a = 5, b = 9;  // a = 5(00000101), b = 9(00001001)
cout << "a = " << bitset<8> (a) << "," << " b = " << bitset<8>(b) << ", ! a = " <<
bitset<8>(!a) << "," << " not b = " << bitset<8>(not b) << endl;
		// The logical and bit result is 00000001
cout << "a & b = " << bitset<8>(a & b) << hex << " Hex value bitand: " << (a bitand
b) << ", a && b = " << bitset<8>(a && b) << ", a and b = " << bitset<8>(a and b) <<
endl;
		// The logical or bit result is 00001101
cout << "a | b = " << dec << bitset<8>(a | b) << hex << " Hex value bitor: " << (a
bitor b) << ", a || b = " << bitset<8>(a || b) << ", a or b = " << bitset<8>(a or
b) << endl;
		// The logical xor bit result is 00001100
cout << "a ^ b = " << dec << bitset<8>(a ^ b) << ", a xor b = " << bitset<8>(a xor
b) << hex << " Hex value: " << (a xor b) << endl;
		// The logical complement bit result is 11111010
cout << "~(a=" << a << ") = " << dec << bitset<8>(~a) << hex << " Hex value compl:
" << (compl a) << endl;
		// The result is 00010010 for left shift
cout << "b << 1" << " = " << dec << bitset<8>(b << 1) << hex << " Hex value: " <<
(b << 1) << endl;
		// The result is 00000100 for right shift
cout << "b >> 1 " << "= " << dec << bitset<8>(b >> 1) << hex << " Hex value: " <<
(b >> 1) << endl;
}//end_main
```

3.5.5. Ex. 5

```
/* compound assignment operators
program that reads an integer from the console and performs with it:
multiplication, division modulo, bitwise XOR, bitwise AND */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main( ){
	int x;
	printf("Enter the value of x: ");
	scanf("%d", &x);
	x ^= 0x0f;
	printf("\n%x", x);
	x %= 2;
	printf("\n%d", x);
	x &= 0x01;
```

```
	printf("\n%x\n", x);
	x *= 4;
	printf("\n%d", x);
	return 0;
}//end_main
```

3.5.6. Ex. 6

```
/* Program that reads an integer and applies the postfixed increment operator and
the prefixed increment operator, displaying the result each time */

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

int main( ){
	int x;
	printf("Enter the value of x: ");
	scanf("%d", &x);

	printf("\n%d", x++);
	printf("\n%d", x);

	printf("\n%d", --x);
	printf("\n%d\n", x);

	//increment/decrement
	int a = 3;
	cout << "\n a initialized = " << a << '\n';//3
	int rez = a++ + ++a * a++;
	cout << "\n a processed = " << a << " rez= " << rez <<'\n';
	//6, 20, or 28, or 33 - depending on the compiler

	return 0;
}//end_main
```

3.5.7. Ex. 7

```
/* sizeof and casting operators
Program that displays the size (in bytes) for some data types and constants
*/

#include <iostream>
using namespace std;


int main( ){
	cout << " Size of char = " << sizeof(char) << " bytes\n";
	cout << " Size of char* = " << sizeof(char*) << " bytes\n";
	cout << " Size of short = " << sizeof(short) << " bytes\n";
	cout << " Size of int = " << sizeof(int) << " bytes\n";
	cout << " Size of long = " << sizeof(long) << " bytes\n";
	cout << " Size of float = " << sizeof(float) << " bytes\n";
	cout << " Size of double = " << sizeof(double) << " bytes\n";
	cout << " Size of 1.55 (double) = " << sizeof(1.55) << " bytes\n";
```

```
        cout << " Size of 1.55 (float) = " << sizeof((float)1.55) << " bytes\n";
        cout << " Size of 1.55L = " << sizeof(1.55L) << " bytes\n"; //8 or 16
        cout << " Size of SALUT = " << sizeof("SALUT") << " bytes\n";
        return 0;
}//end_main
```

3.6. Întrebări

1. Cum se clasifică operatorii funcție de numărul de parametrii?
2. Care este semnificația operatorilor logici pe operand?
3. Care este semnificația operatorilor logici pe biți?
4. Cum se evaluează o expresie în care sunt operanzi de incrementare prefixați și postixați?
5. Asignarea compusă poate fi pre și post fixată în variantele moderne C/C++?

3.7. Questions

1. How are operators classified according to the number of parameters?
2. What is the significance of logical operators on the operand?
3. What is the significance of bitwise logical operators?
4. How is evaluated an expression where increment operands are prefixed and postixed?
5. Can compound assignement be pre and post fixed in modern C/C++ variants?

3.8. Lucru individual

1. Să se scrie un program care afişează valoarea polinomului de gradul 3 pentru o anumită valoare a variabilei reale x. Citiți coeficienții polinomului și valoarea x.
2. Să se scrie un program care citeşte lungimile laturilor unui triunghi (folosind variabile întregi) şi afişează aria triunghiului ca valoare reală.
3. Să se scrie un program care afişează valorile biţilor unei variabile de tip unsigned char aplicând succesiv operatorul de deplasare dreapta şi operatorul %.
4. Să se scrie un program care monitorizează un canal de 16/32/64 biţi. Pentru aceasta citiţi de la tastatură o valoare întreagă fără semn x, care va fi afişată în zecimal, binar, octal şi hexazecimal. Folosiţi o funcţie pentru conversia numerelor din baza 10 în baza 2 sau facilitatea bitset din C++. Implementaţi o funcţie numită getsets() care primeşte trei valori ca parametri:
   x: valoarea citită de la tastatură
   p: poziţia unui bit din cei 16/32 sau 64 de biţi (numărând de la dreapta)
   n: numărul de biţi care vor fi extraşi din valoarea citită.
   Funcţia returnează, aliniaţi spre dreapta, acei n biţi ai valorii x, pornind de la poziţia p, unde p<8*sizeof(x) şi p>n. Afişaţi rezultatul în binar, octal şi hexazecimal.
5. Să se scrie un program care citeşte de la intrarea standard un număr întreg şi afişează numărul de zerouri semnificative din reprezentarea sa binară.
6. Se citește de la intrarea standard o valoare întreagă. Să se afișeze în format zecimal valoarea fiecărui octet al întregului citit.
7. Se citesc de la tastatură 2 numere reale. Să se realizeze operațiile de adunare, scădere, înmulțire și împărțire cu valorile date. Să se afișeze rezultatele obținute, apoi să se rotunjească valorile obținute la valori întregi, folosind operatorul cast și fără a folosi funcții specifice. Să se afișeze apoi valoarea minimă dintre numerele citite folosind operatorul condițional.

8. Citiți de la tastatură mai multe caractere reprezentând litere mici. Să se transforme caracterele citite în litere mari în 2 moduri: printr-o operație aritmetică, respectiv folosind o operație logică și o mască adecvată.
9. Citiți de la tastatură 2 numere întregi. Dacă sunt egale, determinați aria cercului cu raza egală cu valoarea citită. Folosiți M_PI pentru calculul ariei din <cmath>. Dacă sunt diferite calculați aria dreptunghiului cu laturile egale cu valorile date. Afișați aria calculată specificând forma geometrică pentru care s-a făcut calculul.

3.9. Individual work

1. Write a program that displays the value of a 3-rd degree polynom, calculated in a certain point x. Read the coefficients of the polynom and the value of x from the KB.
2. Write a program that reads from the keyboard 3 values representing the lengths of a triangle's sides (using integer values) and than displays the triangle's area.
3. Write a program that reads an unsigned char value and displays the binary values resulting by successively applying the right shift operator and the % operator.
4. Write a program that monitors a communications channel on 16/32/64 bits. In order to do that, read from the keyboard an unsigned int value x, that will be displayed in decimal, binary, octal and hexadecimal counting bases. Use a function for converting the number from base 10 in base 2, or bitset facility provided in C++.
   Implement a function called getsets() that receives 3 parameters:
   x: the value read from the keyboard
   p: the position of a bit of those 16/32 or 64 bits (counting from the right).
   n: the number of bits that will be extracted from the number.
   The function returns, adjusted to the right, those n bits from the value x, starting with the position p, where p<8*sizeof(x) and p>n. Display the result in binary, octal and hexadecimal
5. Write a program that reads from the standard input an integer number and displays the number of zeroes (significant) from it's binary representation.
6. Read an integer value from the standard input. Write, in decimal format, the value of each byte of the value.
7. Read 2 float numbers from the keyboard. Calculate the results obtained by applying the main arithmetic operations (+, -, *, /) upon them. Display the obtained results, then round the obtained values to integer values, using the cast operator and without using specific functions. Then display the minimum value of the numbers read using the conditional operator.
8. Read from the keyboard several lowercase characters. Convert the characters read to uppercase in 2 ways: by an arithmetic operation, respectively by using a logical operation and a suitable mask.
9. Read 2 integer values from the keyboard. If they are equal, determine the area of a circle that has the radius equal with the read value. Use M_PI from <cmath>. If the values are not equal, determine the area of the rectangle that has as sides the read values. In both cases, display the name of the geometrical shape that corresponds to the calculated area.

4. Aplicaţii folosind instrucţiuni în C/C++

Applications using instructions in C/C++

4.1. Obiective

- Înţelegerea categoriilor de instrucţiuni din C/C++

- Înţelegerea modului în care lucrează instrucţiunile în C/C++

- Scrierea şi rularea de programe simple în care sunt folosite aceste instrucţiuni

4.2. Objectives

- Understanding the categories of C/C++ instructions

- Understanding the C/C++ instructions' operating mode

- Writing and running some simple programs that use instructions

4.3. Breviar teoretic

Se pot defini următoarele categorii de instrucţiuni.

Instrucţiunea compusă (secvenţială) este o secvenţă de [declaraţii şi] instrucţiuni, scrisă între acolade {....}

```
{
      //declaraţii
      //instrucţiuni
}
```

Instrucţiunea condiţională (de ramificaţie, decizională, alternativă)

```
if (condiţie) {
      //instrucţiuni
}
else {
      //alte instrucţiuni
}
```

Instrucţiuni ciclice (repetitive)

Instrucţiunea `while` (instrucțiune ciclică codiţionată anterior)

```
while (condiţie){ // antet
      //instrucţiuni (corpul buclei)
}
```

unde:

- dacă condiție != 0 (adevărat) atunci se execută secvența de instrucţiuni, după care se revine la punctul în care se evaluează din nou expresia, corpul executându-se atâta timp cât condiția este adevarată (condiție != 0); când condiție = 0 se trece la următoarea instrucţiune după ciclul while;
- dacă condiție = 0 de la început, atunci corpul nu se execută nicio dată.

Instrucţiunea `for` (instrucțiune ciclică codiţionată anterior):

```
for(exp1;exp2;exp3){
      //instrucţiuni (corpul buclei)
}
```

unde `exp1, exp2, exp3` sunt expresii cu următoarea semnificaţie:

- `exp1` reprezintă partea de iniţializare a ciclului for
- `exp2` reprezintă condiţia de continuare a ciclului for (are acelaşi rol cu condiția din ciclul while)
- `exp3` reprezintă partea de reiniţializare a ciclului for

Biblioteca STL conţine o funcţie template `for_each( )`.

În C++0x/1y/2z a fost introdusă o instrucţiune for pentru parcurgerea colecţiilor cu iteratori (for range based). Se utilizează pentru prelucrarea tuturor elementelor dintr-un domeniu.

```
for (declaratie : expresie_domeniu) {
      //instrucţiuni (corpul buclei)
}
```

unde:

- `declaratie` e declaraţia unei variabile de tipul unui element din domeniu sau o referinţă de acel tip. Deseori utilizează specificatorul `auto` pentru deducerea automată a tipului.
- `expresie_domeniu` sunt secvenţe de elemente cum sunt tablourile, containerele şi alte tipuri care definesc domeniul de valori.

Instrucţiunea do-while (instrucțiune ciclică condiţionată posterior):

```
do{
      //instrucţiuni, se executa cel putin o data
}
while(condiţie);
```

Instrucţiunea selectivă

```
switch (expresie){
      case c1:    //instrucţiuni_1
            [break;]
      case c2:    //instrucţiuni_2
            [break;]
      ...
```

```
      case cn:    //instrucţiuni_n
                  [break;]
      [default:   //alte_instrucţiuni]
}
```

unde:

- `c1, …, cn` sunt constante ce pot fi: întregi, caractere, elemente de enumerare.
- `instrucţiuni_1, …, instrucţiuni_n, alte_instrucţiuni`, sunt secvenţe de instrucţiuni.

Noile compilatoare C++ (nu de la Microsoft) permit instrucţiuni `switch-range`, cu valori într-un domeniu pentru `ci`.

Funcţii şi instrucţiuni de salt

Funcţia `exit()`, declarată în `<stdlib.h>`, are următorul prototip:

```
void exit(int cod); // termină programul în curs de execuţie
```

Instrucţiunea `continue`, are următorul prototip:

```
continue;   //termină iteraţia curentă
```

Instrucţiunea `break`, are următorul format:

```
break;  //termină bucla curentă
//sare la prima instructiune după linia finală a buclei
```

Instrucţiunea `return`, are următoarele formate:

```
return;    //revine din funcţia curentă, fără o valoare asignată
return expresie; /*revine din funcţia curentă, returnând
valoarea expresiei */
```

Instrucţiunea `goto`, are următorul format:

```
goto eticheta;      /* eticheta este un identificator al unei alte
secvenţe de cod */
```

4.4. Theoretical brief

The following categories of instructions can be defined.

The compound (sequential) instruction is a sequence of [statements and] instructions, written between curly braces `{....}`

```
{
      //declarations
      //instructions
```

```
}
```

The conditional instruction (logical branches, decision-making, alternative)

```
if (condition) {
      //instructions
}
else {
      //other instructions
}
```

Loop (repetitive) instructions

`while` instruction (pre-test loop)

```
while (condition){      //header
      //instructions (loop body)
}
```

where:

- if condition != 0 (true) the sequence of instructions is executed, after which the program returns to the point where the expression is re-evaluated, the loop body being executed as long as condition is true (condition != 0); when condition = 0 (false) the program proceeds to the next instruction after the while cycle;
- if condition = 0 from the beginning, the loop is never executed.

`for` instruction (pre-test loop)

```
for(exp1;exp2;exp3){
      //instructions (loop body)
}
```

where `exp1, exp2, exp3` are expressions with the following meaning:

- `exp1` represents the initialization part of the loop
- `exp2` represents the loop continuing condition (has the same role as the condition in the while cycle)
- `exp3` represents the re-initialization part of the for loop

The STL library contains a template function named `for_each( )`.

In C++0x/1y/2z a specific for instruction has been introduced for iterating collections (for range based). It is used to process all elements in a domain.

```
for (declaration: domain_expression) {
      //instructions (loop body)
}
```

where:

- `declaration` is declaration of a variable having the type corresponding to the elements in the domain. Frequently the `auto` specifier is used for automatically determining the variable's type.

- `domain_expression`: sequences of items such as arrays, containers, and other types that define the collection.

do-while instruction (posterior loop instruction)

```
do{
      //instructions, executed at least once
}
while(condition);
```

The selective instruction

```
switch (expression){
      case c1:    //instructions_1
                  [break;]
      case c2:    // instructions_2
                  [break;]
      ...
      case cn:    // instructions_n
                  [break;]
      [default:   //other_instructions]
}
```

where:

- `c1, …, cn` are constants having as type: integer, character, enumeration elements.
- `instructions_1, …, instructions_n, other_instructions`, are sequences of instructions.

New compilers (not from Microsoft) allow switch range, as a range value for `ci`.

Jump functions and instructions

Function `exit()`, declared in `<stdlib.h>`, has the following prototype:

```
void exit(int cod); // terminates the program
```

Instruction `continue`, has the following prototype:

```
continue;   //terminates the current iteration
```

Instruction `break`, has the following format:

```
break;
//terminates the current loop
//jumps to the first instruction after the final line of the loop
```

Instruction `return`, has the following formats:

```
return;    /*returns from current function without having a value
assigned */
return expression; /*returns from the current function, returning the
expression value */
```

Instruction `goto`, has the following format:

```
goto label;      // label is an identifier of another code sequence
```

4.5. Exemple / Examples

4.5.1. Ex. 1

```
/* Program that reads multiple integers in a one-dimensional array, validating the
number of elements, and then display them */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define MAX  10

int main() {
    int i, n, tab[MAX];
    //reading the number of values
    do
    {
        printf("\nEnter the number of values n: ");
        scanf("%d", &n);
    if (n <= 0 || n > MAX)printf("\n\t Incorrect size (must be >0 and <=%d !", MAX);
    } while (n <= 0 || n > MAX);

    printf("\nEnter %d integer numbers: ", n);
    for (i = 0; i < n; i++) {
        printf("\n\t Waiting for number %d : ", i + 1);
        scanf("%d", &tab[i]);
    } // end for
    //... data processing
    printf("\nYou entered the following values :\n");
    for (i = 0; i < n; i++)
        printf("\t%d\n", tab[i]);
    return 0;
}//end_main
```

4.5.2. Ex. 2

```
/* program that reads a number in the range [1...7] and displays the name of the
corresponding day of the week */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int  main( ){
       int i;
       puts("Enter a number in the interval [1,7]: ");
       scanf("%d", &i);

       switch(i){
```

```
        case 1:
                puts("Monday");
                break;
        case 2:
                puts("Tuesday");
                break;
        case 3:
                puts("Wednesday");
                break;
        case 4:
                puts("Thursday");
                break;
        case 5:
                puts("Friday");
                break;
        case 6:
                puts("Saturday");
                break;
        case 7:
                puts("Sunday");
                break;
        default:
                puts("Wrong number");
        }
        return 0;
}//end_main
```

4.5.3. Ex. 3

```
/* Adds the values entered from the keyboard. Confirmation required */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
//#include<conio.h>

int main( ){
        int m, sum = 0;
        char key;
        do{
                printf("Enter a number: ");
                scanf("%d", &m);
                sum += m;
        printf("Press any key to continue. N or n stops the reading: \n");
                //key = _getch();
                scanf(" %c", &key);
        } while (!((key == 'n') || (key == 'N')));
        printf("The sum of the entered numbers is: %d\n", suma);
        return 0;
}//end_main
```

4.5.4. Ex. 4

```
/* The program evaluates whether an entered number is prime or not. Optimize the
algorithm */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
```

```
int nrPrim(int);//prototype

int main( ){
	int nr;
	printf("\nWrite a number : ");
	scanf("%d", &nr);
	if(nrPrim(nr))
		printf("\nNumber %d is prime\n", nr);
	else
		printf("\nNumber %d is not prime\n", nr);
	return 0;
}// end_main

// The function returns 1 if n is prime or 0 if it is not
int nrPrim(int n){
	int i;
	if(n <= 1)
		return 0;
	if(n == 2)
		return 1;
	if(n % 2 ==0)
		return 0;
	for(i=3; i<n; i+=2) {
		if(n % i == 0)
			return 0;
	} // for
	return 1;
}// end_nrPrim
```

4.5.5. Ex. 5

```
/* Program that displays the values in a one-dimensional array, using 2 variants of
the for loop */

//#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

int main( ) {
	int arr[ ] = { -1, 20, 3, 4 }; // pre-initialized uni-dimensional array
	cout<< "The array has: " << _countof(arr) <<" elements.\nThey are: \n";

	for (int i : arr)  // iterator with specified type
		cout << i << " ";

	cout << "\nThe values will be modified!\n";
	for (auto& i : arr) { // iterator with auto reference
	cout << "\nThe value of the current item is = " << i << "\t enter a new
value: ";
	cin >> i;// scanf("%d", &i);
// both work the same, scanf() requires #define _CRT_SECURE_NO_WARNINGS
	}

	cout << "\nThe new array (using sizeof):\n";
	for (int i = 0; i < sizeof(arr) / sizeof(int); i++)// standard for
```

```
            cout << arr[i] << ' ';

      cout << "\n\nThe new array, (using _countof)" << '\n';
      for (int i = 0; i < _countof(arr); i++)// standard for
            cout << arr[i] << ' ';
      cout << '\n';

      cout << "\nfor-range access to an init list\n";
      for (int i : {-1, 1, 3, 5, 7}) // the initializer may be a braced-init-list
            cout << i << ' ';
      cout << '\n';

      cout << " \nAnother display...\n";
      for (int i : arr)
            cout << 7 << ' ';

      cout << "\nArray of pointers to constant character strings\n";
      const char* words[ ] = { "alfa", "beta", "gama" };
      for (const char* word : words)
            cout << word << '\t';

      return 0;
} // main
```

4.5.6. Ex. 6

```
/* Switch range gcc compilers, not VC++ Microsoft */

#include <iostream>
using namespace std;

int main( ){
      int score;
      cout << "Score values 0-100:";
      cin >> score;
      switch(score){
       case 0:
            cout << "a"; break;
       case 1 ... 10:
            cout << "b"; break;
       case 11 ... 24:
            cout << "c"; break;
       case 25 ... 49:
            cout << "d"; break;
       case 50 ... 100:
            cout << "e"; break;
        default:      cout << "BAD VALUE";
      }
      cout << endl;
}//end_main
```

4.6. Întrebări

1. Evidenţiaţi diferenţa dintre structura while şi do...while printr-un exemplu.
2. Care este rolul instrucţiunii break în cadrul instrucţiunii switch?

3. Care este rolul instrucţiunii break în cadrul instrucţiunilor ciclice?
4. Care este rolul instrucţunii continue în cadrul instrucţiunilor ciclice ?

4.7. Questions

1. Give an example that highlights the difference between the while structure and the do... while.
2. What is the role of break instruction inside a switch?
3. What is the role of break instruction in cyclic instructions?
4. What is the role of continue instruction in cyclic instructions?

4.8. Lucru individual

1. Se citesc trei numere de la tastatură a, b şi c. Să se determine aria dreptunghiului ale cărui laturi sunt a şi b. Verificaţi dacă diagonala dreptunghiului este egală cu c, considerând o precizie eps=0.01 definită iniţial.
2. Să se scrie un program care verifică dacă un număr citit de la tastatură este pătrat perfect.
3. Să se scrie un program care calculează an, prin înmulţiri repetate ale lui a, unde n este citit de la tastatură (a se defineşte în program sau se citeşte de la tastatură).
4. Să se scrie un program care citeşte de la tastatură o valoare întreagă n şi calculează n! (n-factorial) în mod nerecursiv.
5. Să se scrie un program care citeşte de la intrarea standard un număr dat şi:
   - determină cel mai mare număr prim mai mic decât numărul dat
   - determină toate numerele prime mai mici decât numărul dat.
6. Să se scrie un program care determină cel mai mare divizor comun a doi întregi introduşi de la tastatură.
7. Să se scrie un program care determină toţi divizorii unui număr introdus de la tastatură.
8. Calculaţi produsul a două numere întregi folosind numărul corespunzător de adunări ale primului număr, dat de al doilea.
9. Să se scrie un program care determină câtul împărţirii a doi întregi introduşi de la tastatură folosind scăderi succesive.
10. Să se scrie un program care determină numărul de cifre care compun un număr întreg citit de la tastatură.
11. Să se scrie un program care citeşte de la tastatură n numere întregi. Afişează toate numerele impare introduse şi le stochează într-un tablou.
12. Să se citească un număr întreg r de la tastatură. Se citesc apoi numere reale, până când suma lor depăşeşte valoarea lui r. Să se afişeze suma numerelor citite, cu o precizie de 2 zecimale la partea fracţionară şi numărul lor (câte s-au introdus).
13. Să se scrie un program care determină cmmmc a două numere citite de la tastatură.
14. Scrieţi un program care citeşte n numere întregi de la tastatură şi le afişează pe cele divizibile cu 3.
15. Să se scrie un program care citeşte de la tastatură un caracter, pe care îl afişează pe n rânduri, câte n caractere pe un rând, n citit de la tastatură.
16. Să se scrie o aplicaţie C/C++ în care se introduc de la tastatură numere întregi, până ce utilizatorul apasă tasta <Esc>, sau o altă tastă predefinită. Să se determine şi să se afişeze media numerelor impare pozitive care au fost introduse.

4.9. Individual work

1. Read from the keyboard 3 numbers a, b and c. Determine the area of the rectangle that has the sides equal to a and b. Verify if the rectangle's diagonal is equal to c with a precision eps=0.01 initial established.
2. Please verify if a natural number introduced from the keyboard is a perfect square or not.
3. Write a program that calculates an, by repeated multiplications of a where n is read from the keyboard (a is "hard coded", or introduced from the KB).
4. Write a program that reads from the keyboard an integer value n and calculates n! (n-factorial) not in a recursive mode.
5. Write a program that reads from the KB a given number and:
   - determines the greatest prime number that's smaller than the given number.
   - determines all the prime numbers smaller than the given number.
6. Write a program that determines the greatest common divider of 2 integer values read from the keyboard.
7. Write a program that determines all the divisors of a value introduced from the keyboard.
8. Calculate the product of 2 integer numbers using additions of the first number given by the second one.
9. Write a program that determines the integer quotient of 2 integer numbers introduced from the KB using a series of subtractions.
10. Write a program that determines the number of figures that compose an integer number read from the keyboard.
11. Write a program that reads from the keyboard n integer numbers. Display all the odd numbers and store them in an array.
12. Read from the keyboard an integer number r. Read a series of real numbers, until their sum is greater than r. Display the sum with a 2 digits precision at the fractional part and how many numbers were introduced.
13. Determine the least common multiple of 2 integer numbers read from the keyboard.
14. Write a program that reads n integer numbers from the keyboard and displays the values that can be divided by 3.
15. Write a program that reads from the keyboard a single character. The program should display that character on n rows, n times in a row, n read from the keyboard.
16. Write a C/C++ application that reads from the keyboard a series of integer numbers, until the user presses the <Esc> key, or another predefined key. Determine and display the average value of the odd positive numbers.

5. Aplicații cu tablouri în C/C++

C/C++ applications using arrays

5.1. Obiective

- Înțelegerea tablourilor unidimensionale și multidimensionale din C/C++
- Înțelegerea modului în care se lucrează cu tablourilor unidimensionale și multidimensionale din C/C++: declarare, inițializare, accesul la elemente, diferite operații
- Scrierea și rularea de programe simple în care sunt folosite tablouri

5.2. Objectives

- Understanding the one-dimensional and multi-dimensional arrays in C/C++
- Understanding the operations with arrays: declaration, initialization, access to elements, various other operations
- Writing and running some simple programs that use arrays

5.3. Breviar teoretic

Un tablou este o secvență de elemente de același tip stocate contiguu. Dimensiunea tabloului poate fi o expresie constantă întreagă pozitivă care specifică numărul de elemente. Tipul elementelor este tipul tabloului.

Orice tablou are un nume (care acționează ca un pointer constant la primul element, dar nu este efectiv un pointer). Deci, un nume de tablou este un identificator. Poate fi convenabil să-l considerăm ca un pointer constant în anumite contexte, deoarece comportamentul este în mare parte același și este implicit convertibil către pointer, dar nu este, în sine, un pointer.

5.3.1. Tablouri unidimensionale

Declarare:

```
tip_de_date nume_variabila_tablou[dimensiune];
```

Pentru referirea unui element se folosește operatorul de indexare, `[ ]`, precizând numele tabloului și poziția elementului în tablou (indexul sau indicele):

```
nume_variabila_tablou[index]
```

unde variabila `index` este de tip întreg și poate lua valori de la `0` până la `dimensiune-1`.

5.3.2. Tablouri multidimensionale

Declarare:

$$\texttt{tip\_de\_date nume\_variabila\_tablou[dimensiune}_1\texttt{][dimensiune}_2\texttt{]...[dimensiune}_N\texttt{]}$$

Un tablou cu mai multe dimensiuni este de fapt un tablou unidimensional având ca elemente alte tablouri, adică un tablou N-dimensional poate fi privit ca un tablou unidimensional având ca elemente tablouri N-1 dimensionale.

Pentru referirea unui element se folosește numele variabilei tablou urmată de operatorul de indexare de mai multe ori, odată pentru fiecare nivel.

$$\texttt{nume\_variabila\_tablou[index}_1\texttt{][index}_2\texttt{]...[index}_N\texttt{]}$$

5.3.3. Elemente de tablou

Un element de tablou poate să apară oriunde poată să apară o variabilă simplă cu același tip ca tipul tabloului: atribuiri, citiri de la consolă, afișări, calcule (expresii), apeluri de funcții.

5.3.4. Inițializarea tablourilor la declarare

Inițializarea se poate face la declarare, prin utilizarea unei liste de valori separate prin virgulă și delimitată de acolade (sau în noile versiuni C++ fară a specifica valorile, cu inițializare cu zero):

a) Tablouri unidimensionale:

```
tip_de_date nume_variabila_tablou[dimensiune] = { el0, el1, ..., eldimensiune-1 }
```

b) Tablouri multidimensionale:

```
tip_de_date nume_variabila_tablou[m][n] =
{ {el0,0, el0,1, ..., el0,n-1}, {el1,0, el1,1, ..., el1,n-1}, ..., {elm-1,0, elm-1,1, ..., elm-1,n-1} }
```

5.3.5. Prelucrarea tuturor elementelor unui tablou

Se utilizează cel mai frecvent instrucțiunea ciclică for( ) standard pentru a prelucra secvențial toate elementele. Variabila de ciclare este folosită ca index în tablou, iar o altă variabilă sau constantă simbolică ce conține dimensiunea tabloului se folosește în condiția asociată instrucțiunii ciclice. Când nu se face distincție între elemente se preferă folosirea unei instrucțiuni de tip for_range, introdusă ulterior in C++.

5.3.6. Transmiterea tablourilor către funcții

Se poate face prin declararea unui argument de tip tablou, și poate fi fără prima dimensiune. Uzual, se mai transmite și dimensiunea (sau dimensiunile) tabloului ca argument(e) separat(e).

Fiind o formă de transmitere de parametri **prin adresă**, orice modificare făcută în funcție a elementelor unui tablou transmis ca parametru se face asupra tabloului inițial. Dacă însă se folosește **modificatorul** `const` pentru tablou, atunci în funcție nu se mai pot modifica valorile elementelor tabloului original.

În cazul tablourilor multidimensionale, prima dimensiune din stânga poate lipsi, celelalte sunt obligatorii.

Exemplu de prototip de funcție ce primește ca argument un tablou:

```
int tab_fcn(char tablou[][25][50]); //prima dimensiune nespecificata
```

5.3.7. Căutarea în tablouri

De obicei se caută poziția unui element cu o anumită valoare.

Dacă tabloul nu este sortat (elementele nu sunt aranjate într-o anumită ordine) se aplică “căutarea liniară” care constă în parcurgerea secvențială (unul după altul) a elementelor tabloului și compararea acestora cu valoarea căutată. Dacă este găsit un element ce are acea valoare, se returnează indicele elementului, altfel se returnează valoarea -1. Acest algoritm permite găsirea primului element ce are o anumită valoare, însă pot fi mai multe elemente cu aceeași valoare.

Pentru căutarea în tablouri sortate există algoritmi mult mai eficienți.

5.3.8. Inserarea în tablouri

Pentru a insera elemente noi într-un tablou trebuie să ținem cont de următoarele:

- trebuie să existe poziții libere în tablou;
- pentru elementul nou se creează loc prin deplasarea elementelor de pe poziția unde vrem să inserăm 1 poziție spre dreapta.

În cazul tablourilor nesortate, cel mai simplu este să inserăm elementul pe prima poziție liberă dinspre dreapta tabloului.

În cazul tablourilor sortate trebuie găsită mai întâi poziția în care se va face inserarea, apoi trebuie mutate unele elemente spre dreapta tabloului pentru a-i face loc elementului de inserat, și apoi trebuie făcută inserarea.

5.3.9. Ștergerea din tablouri

Ca și inserarea, ștergerea presupune deplasarea elementelor din stânga poziției de șters 1 poziție spre stânga tabloului.

Se presupune că se cunoaște poziția în care se face ștergerea (dacă nu, se face căutarea elementului de tablou cu o anumită valoare).

5.4. Theoretical brief

An array is a sequence of elements of the same type stored contiguously. The size of the array can be a positive integer constant expression that specifies the number of elements. The elements type is the array type.

Any array has a name (which acts as a constant pointer to the first element but is not actually a pointer). So, an array name is an identifier. It may be convenient to think of it as a constant pointer in some contexts because the behavior is largely the same, and it is implicitly convertible to pointer, but it is not itself a pointer.

5.4.1. One-dimensional array

Declaration:

```
data_type array_variable_name[size];
```

To address an element of the array we use the indexing operator, `[ ]`, after the name of the array variable and we specify the position from the array that we want to address (also known as the index position):

```
array_variable_name[index]
```

where the variable `index` is an integer type and can take values from `0` up to `size-1`.

5.4.2. Multi-dimensional array

Declaration:

$\texttt{data\_type array\_variable\_name[dimension}_1\texttt{][dimension}_2\texttt{] ... [dimension}_N\texttt{]}$

An array with multiple dimensions is in fact a unidimensional array of arrays, meaning that an N-dimensional array can be looked at in the same way as a one-dimensional array of N-1 dimensional arrays.

To refer to an element from an array we use the array variable name followed by the indexing operator as many times as the number of dimensions in the array.

$\texttt{array\_variable\_name [index}_1\texttt{][index}_2\texttt{]...[index}_N\texttt{]}$

5.4.3. Array elements

An array element can be used anywhere we can use simple variables of the same type as the array: assignments, reading from the console, writing to the console, computations (expressions of various kind), function calls.

5.4.4. Initialization of arrays at declaration

The initialization of arrays can be done at declaration time by using a comma delimited list of values placed within curly braces (or in newer C++ versions without specifying the values, with zero-initialization):

a) One-dimensional array initialization:

```
data_type array_variable_name[dimension] = { el0, el1, ..., eldimensiune-1 }
```

b) Multi-dimensional array initialization:

```
data_type array_variable_name[m][n] =
{ {el0,0, el0,1, ..., el0,n-1}, {el1,0, el1,1, ..., el1,n-1}, ..., {elm-1,0, elm-1,1, ..., elm-1,n-1} }
```

5.4.5. Iterating through all array elements

In order to iterate through and access or modify all array elements we will use the for( ) instruction most of the time. The variable used in the for( ) cycle will usually be used as an array index and another variable or a constant that contains the array size will be used in the associated cycle condition. When the element position is not important during processing, we can use an instruction of type for_range to cycle through all elements. The latter was introduced in C++.

5.4.6. Passing an array to a function

This is done by declaring an array-type function argument that can be specified with or without explicitly mentioning the length of the first dimension and by always specifying the lengths of the rest of the dimensions.

The arrays that are passed to the function in this way are passed by address. This means that the same array as the one passed from the caller function is observed and modified inside the called function through the array-type function argument. When the `const` modifier is used when specifying the array-type function argument, array modifications are not permitted in the function.

Example prototype of a function with a multi-dimensional array argument specified:

```
//first dimension is not specified
int array_fcn(char array_param[][25][50]);
```

5.4.7. Finding array elements

One of the most frequently used arrays searching operations involves finding the position of an element having a given value.

If the array is not sorted (elements are not arranged in a certain order) we apply a ¨linear search¨ that involves iterating through elements sequentially (one after the other) and comparing each one of them with the given value. If an element having the given value is found, the index of the element is returned. If no element is found with the given value, -1 is returned. This algorithm allows finding the first element that has a given value, but an array might have multiple elements with the same value.

More efficient algorithms exist for finding items in sorted arrays.

5.4.8. Inserting elements into arrays

When inserting an element into an array, we need to ensure that the following prerequisites are met:

- the array needs to have free positions where the new element can be stored;

- once we determined the position into which the new element should be inserted, we move all elements from that position until the last one, 1 position to the right (towards the end of the array), and then perform the insert.

For unsorted arrays, the insert is done most easily done at the first available free position (no moving of elements required).

For sorted arrays, we need to find the index where the element should be inserted, proceed to moving of elements towards the end of the array and afterwards do the insert.

5.4.9. Removing elements from arrays

Like the insert, the deletion involves moving elements that are located after the index of the element to be deleted 1 position towards the beginning/left of the array.

Sometimes we may need to find the position/index of the element that will be deleted.

5.5. Exemple / Examples

5.5.1. Ex. 1

```
/* Program that reads the elements of a square matrix (a bidimensional array with
same size dimensions) and determines the sum of the elements located on the main
diagonal.  */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#define MAX  7
//constexpr int MAX=7;//C++

int main( ) {
  int i, j, n, sum = 0, tab[MAX][MAX];
// A read followed by value validation
    printf("\n Type the size of the matrix (size=lines=columns) <= %d: ", MAX);
    scanf("%d", &n);
    if (n <= 0 or n > MAX) {
        printf("\n\t Incorrect size (must be >0 and <=%d !", MAX);
        return 1;
    }//end_if

// Reading the elements of the matrix
    printf("\nType the elements of the matrix (integer numbers):");
    for (i = 0; i < n; i++) {
        printf("\n\t Line %d: \n", i);
        for (j = 0; j < n; j++) {
            printf("\t\tarray[%d,%d] = ", i, j);
            scanf("%d", &tab[i][j]);
        }//end_for_int
    }//end_for_ext

// Computing the sum of the elements from the main diagonal
    for (i = 0; i < n; i++)       // iterate through the first dimension
        for (j = 0; j < n; j++)   // iterate through the second dimension
            if (i == j)           // when indices are equal to each other
                sum += tab[i][j]; // add to the sum

    //for (i = 0; i < n; i++) sum+=tab[i][i]; // a more efficient version
```

```
// Print the result
    printf("\nThe sum of the elements on the main diagonal is: %d\n ", sum);
    return 0;
}//main
```

5.5.2. Ex. 2

```
/* Program for calculating the average value of a one-dimensional array */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#define DIM 7

float computeAverage(int x[], int dim); //prototype

int main( ){
    int i, dim;
    int x[DIM]={};//init_list, array values are 0
    printf("\n Type the size of the array to read <=%d: ", DIM);
    scanf("%d", &dim);
    if (dim <= 0 || dim > DIM) {
        printf("\n Incorrect size (must be >0 and <=%d ! ", DIM);
        return 1;
    }

    printf("\n Type the elements of the array (integer values):\n");
    for (i = 0; i<dim; i++)
    {
        printf("\tx[%d] = ", i);
        scanf("%d", &x[i]);
    }

    printf("\n The average value is: %.3f\n", computeAverage(x, dim));
    return 0;
}//main

// Function to determine the average value of the elements of an array
float computeAverage(int x[], int n){
    int i;
    int sum = 0;
    for (i = 0; i<n; i++)
        sum += x[i];
    return (float) sum / n;
}//computeAverage
```

5.5.3. Ex. 3

```
/* Program for finding the index of an element in a one-dimensional array */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#define DIM 100
int findElement(int[], int, int);//prototype
```

```
int main( ){
    int, dim, val, pos;
    int x[DIM];
    printf("\n Type the size of the array to read( <= %d): ", DIM);
    scanf("%d", &dim);
    if (dim <= 0 || dim > DIM) {
        printf("\n Incorrect size! (must be >0 and <=%d)", DIM);
        return 1;
    }

    printf("\n Type the elements of the array (integer values):\n");
    for (int i = 0; i<dim; i++)
    {
        printf("\tx[%d] = ", i);
        scanf("%d", &x[i]);
    }

    printf("\n Type the number to find in the array: ");
    scanf("%d", &val);
    pos = findElement(x, dim, val);
    if (pos < 0)
        printf("\n The number %d was not found in the array!\n", val);
    else
        printf("\n The number %d was found at position/index %d.\n", val, pos);
    return 0;
}//main

// Function to find a value in an array
// Returns the value index when successful or -1 otherwise
int findElement (int x[], int n, int val){
    for (int i = 0; i < n ; i++){
        if (x[i] == val)
            return i;
    }
    return -1;
}// findElement
```

5.5.4. Ex. 4

```
/* Program for inserting a value into a one-dimensional array with input data
validation */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define DIM 5

void insert(int arr[], int& n, int idx, int val);
void insertOrderedElement(int arr[], int& n, int val);

int main() {
    int i, dim, val;
   // int pos; //for unordered
    int x[DIM];
    do {
        printf("\nEnter the size of the array to read(<=%d): ", DIM - 1);
        scanf("%d", &dim);
```

```
        if (dim >= DIM)
            printf("\nThe dimension is too big for insertion!\n");
        if (dim <= 0)
            printf("\nThe dimension is invalid (zero or negative)!\n");
    } while (dim >= DIM || dim <= 0);

    printf("\nEnter the array elements (int)  :\n");
    //printf("\nEnter the array elements (ordered int)  :\n");
    for (i = 0; i < dim; i++) {
        printf("\tx[%d] = ", i);
        scanf("%d", &x[i]);
    }

    printf("\nEnter the number to insert : ");
    scanf("%d", &val);

    /*
    //unordered array
    do {
        printf("\nEnter the position to insert between 0 and %d: ", dim);
        scanf("%d", &pos);
        if (pos < 0 || pos > dim)
            printf("\nIt is not a valid position for insertion!\n");
    } while (pos < 0 || pos > dim);
    insert(x, dim, pos, val);
    */

    insertOrderedElement(x, dim, val);

    printf("\tThe array values are :\n");
    for (i = 0; i < dim; i++)
        printf("\t%d", x[i]);//end for
    printf("\n");
    return 0;
}//main

void insert(int arr[], int& n, int idx, int val) {
    if (idx == -1) arr[n] = val;  // append at the end
    else
    { // right shift (from the end)
        for (int i = n; i > idx; i--)
            arr[i] = arr[i - 1];
        // insert new value
        arr[idx] = val;
    }
    n++;
} //insert

void insertOrderedElement(int arr[], int& n, int val)
{
    int insertPosition;
    // insert position search
    insertPosition = -1;
    do
        insertPosition++;
    while (val > arr[insertPosition] && insertPosition != n - 1);
```

```
    if (val <= arr[insertPosition]) {
        // right shift
        for (int i = n - 1; i >= insertPosition; i--)
            arr[i + 1] = arr[i];
        // insertion
        arr[insertPosition] = val;
    }
    else
        arr[n] = val; // add to the end
    n++;
}//insertOrderedElement
```

5.5.5. Ex. 5

```
// 3D matrix display – three dimensional array from xOy perspective

#include<stdio.h>

#define Max1 3 //last limit - planes
#define Max2 2 //midle limit - rows
#define Max3 4 //first limit - columns

int main( ) {
    int i, j, k;
    double arr[Max1][Max2][Max3] = {
    {{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
     {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
     {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
    };
    printf(":::3D Array Elements:::\n\n");
    for (i = 0; i < Max1; i++)
    {
        for (j = 0; j < Max2; j++)
        {
            for (k = 0; k < Max3; k++)
            {
                printf("%6.2lf\t", arr[i][j][k]);
            }
            printf("\n");
        }
        printf("\n");
    }
}//main
```

5.5.6. Ex. 6

```
// Program for returning a static array declared in a function

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

#define MAX 10

int * getRandom( ); /* function to generate and return random numbers */

/* main function to call the getRandom( ) function */
```

```
int main( ){
    /* a pointer to an int */
    int *p;
    p = getRandom( );
    for ( int i = 0; i <MAX; i++ )
        printf("\nFrom main p[ %d] :  %d", i, p[i]);
}//main

int * getRandom( ){
    static int r[MAX];//must be static or allocated in dynamic mode
    /* set the seed */
    srand( (unsigned)time( NULL ) );
    for(int i = 0; i < MAX; ++i)  {
        r[i] = rand( );//generate random numbers
        printf( "From function r[%d] = %d\n", i, r[i]);
    }
    return r;
}//getRandom
```

5.6. Întrebări

1. Cum se pot accesa elementele unui tablou?
2. Cum se face inițializarea unui tablou?
3. Ce presupune inserarea, respectiv ștergerea, unui element dintr-un tablou?
4. Cum se realizează transmiterea unui tablou într-o funcție?

5.7. Questions

1. How can the elements of an array be accessed?
5. How can we initialize an array?
6. What does the insertion, removal, of an element from an array entail?
7. How do we pass an array as parameter/argument to a function?

5.8. Lucru individual

1. Scrieți un program pentru determinarea valorii medii a elementelor pozitive și ale celor negative dintr-un tablou unidimensional de maxim 10 elemente întregi cu validarea dimensiunii.
2. Scrieți un program pentru determinarea celui mai mic element pozitiv dintr-un tablou întreg unidimensional de maxim 10 elemente. Validați dimensiunea tabloului prin reintroducerea ei.
3. Scrieți o aplicație C/C++ care citește de la tastatură un tablou de maxim 10 valori întregi. Definiți o funcție care primește tabloul ca parametru și apoi îl afișează ordonat crescător (prin orice metodă de sortare).
4. Scrieți o aplicație C/C++ care definește o parolă (în format șir de caractere). Programul citește în mod repetat șirurile de caractere introduse de la tastatură, până când utilizatorul scrie parola corectă. Să se afișeze numărul de încercări până la introducerea parolei corecte.
5. Scrieți o aplicație C/C++ care citește de la tastatură două șiruri de caractere reprezentând numele și prenumele unei persoane. Afișați-le cu majusculă, separate printr-un spațiu de tabulare.
6. Scrieți o aplicație C/C++ care definește două matrice de valori întregi. Dimensiunea și elementele matricelor sunt citite de la tastatură și validate în program. Scrieți funcțiile care:
    - afișează pozițiile elementelor pare din fiecare matrice
    - afișează suma elementelor impare din ambele matrice
    - afișează suma matricelor (ajustare linii și coloane dacă nu sunt egale)

7. Citiți de la tastatura elementele unei matrice cu elemente de tip float, cu dimensiunea de maxim 5x5. Rearanjați coloanele matricei astfel ca suma elementelor de pe o anumită coloană să fie mai mică decât suma elementelor de pe coloana următoare.
8. Scrieți un program C/C++ care preia de la tastatură „n<=10” valori reale într-un tablou unidimensional, calculează cu o funcție valoarea medie a elementelor introduse și afișează cu o altă funcție doar valorile din tablou mai mari decât valoarea medie calculată. Validați dimensiunea tabloului.
9. Să se scrie o aplicație C/C++ în care se citesc într-un tablou unidimensional „n<=10” valori întregi și se determină numărul elementelor negative impare. Să se afișeze acest număr și elementele respective. Validați dimensiunea tabloului.
10. Scrieți programul C/C++ care citește elementele întregi ale unui tablou unidimensional „n<=10” și construiește într-o funcție un alt tablou unidimensional în care se vor stoca resturile împărțirii elementelor primului tablou la numărul elementelor pozitive din acesta. Afișați tabloul construit. Validați dimensiunea tabloului inițial.
11. Se citește de la tastatură un șir de caractere. Scrieți funcția care inversează șirul și apoi formează un alt șir de caractere ce va conține caracterele de pe pozițiile pare ale șirului inversat. Afișați șirurile obținute. Nu folosiți funcții de bibliotecă specifice șirurilor de caractere.
12. Să se citească de la tastatură elementele întregi ale unei matrice de dimensiune m x n, m și n <=7. Dacă matricea este pătratică să se afișeze elementele diagonalei secundare, altfel să se afișeze suma elementelor de pe o coloană valida dată, c. Valorile m, n și c se citesc de la tastatură și se vor scrie funcții pentru operațiile cerute.
13. Pornind de la exemplul 5 (tablouri 3D), în care se arată modul de parcurgere a unui tablou tridimensional din perspectiva xOy, să se scrie un program care face parcurgerea tabloului considerând perspectiva xOz și yOz. Afișați datele precizând din ce perspectivă au fost parcurse.
14. Afișați fiecare apariție a fiecărui caracter din alfabetul englez preluat dintr-un șir de testare (de exemplu „The quick brown fox jumps over the lazy dog.”), folosind cod morse (mecanism din vechea telegrafie). Pe o linie se va afișa un triplet format din: caracterul luat din șirul de test de la început, ofsetul din interiorul șirului unde a fost găsit (poziția relativă față de început), și codul morse aferent caracterului. Afișați șirul de test inițial cu prima apariție a fiecărui caracter convertit către caractere majuscule, dacă e cazul (nu caracterele majuscule, sau altele diferite de litere - THE QUICK BROWN FoX JuMPS oVer the LAZY DoG.).
15. Scrieți o aplicație C/C++ în care se citește de la tastatură un tablou de maxim 10 valori întregi cuprinse în plaja 2, ..., 10. Dimensiunea reală se introduce de la tastatură. Definiți două funcții care primesc tabloul și dimensiunea lui ca parametri și returnează valoarea minimă, respectiv maximă din tablou. Definiți o a treia funcție, filtru_olimpic( ), care primește ca parametrii șirul inițial și dimensiunea lui, valoarea minimă si cea maximă, returnând media elementelor din tablou fără prima apariție a valorii minime și a celei maxime. În main( ) afișați tabloul inițial, valoarea minimă și cea maximă, media returnată de filtrul olimpic cu o precizie de 2 zecimale la partea fracționară. Validați introducerea datelor (dimensiune tablou, elemente introduse, cu reluare).

5.9. Individual work

1. Write a C/C++ program that determines the mean value of the negative and positive elements from a one-dimensional int array of maximum 10 elements, with dimension validation.
2. Write a C/C++ program that determines the value of the smallest positive element of a one-dimensional int array of maximum 10 elements. Validate the array size by re-entering it.

3. Write a C/C++ application that reads from the keyboard an array of maximum 10 integer values. Define and implement a function that receives the array as parameter and then displays its elements, ordered increasingly (by any sorting method).
4. Write a C/C++ application that defines a password (as a string of characters). The program reads the entries repeatedly from the keyboard until the user enters the right password. Display the number of trials the user entered wrong passwords.
5. Write a C/C++ application that reads from the keyboard two arrays of characters representing the name and surname of a person. Display them with capital letters, separated by a <Tab> space.
6. Write a C/C++ application that defines two matrices of integer values. The dimensions of the matrices and their elements are read from the keyboard and validated in the program. Write the functions that display:
   - the positions of the odd elements in each matrix
   - the sum of all even elements in both matrices
   - the sum of the two initial matrices (adjusting rows and columns if are not equals)
7. Read from the keyboard the elements of a maximum 5x5 float matrix. Rearrange the columns so that the sum of each column's elements is smaller than the sum of the elements of the next column to the right.
8. Write a C/C++ program that reads from the keyboard n<=10 integer values into a one-dimensional array. Implement a function that calculates their average value. Another function will display the elements from the initial array that are greater than the computed average. Validate the array size.
9. Write a C/C++ program that reads from the keyboard n<=10 integer values into a one-dimensional array. Display the number of negative odd elements and their values. Validate the array size.
10. Write a C/C++ program that reads the integer elements of a one-dimensional array, n<=10. Implement a function that builds another array containing the remainders obtained by dividing the values in the initial array to the number of positive elements. Display the resulting array. Validate the array size of the initial array.
11. Read from the keyboard an array of characters. Define the function that reverses the array and populates another array with the elements from the even positions of the reverted array. Display the results. Do not use string library functions.
12. Read from the keyboard the elements of an m x n integer matrix, m and n <=7. If the matrix is square, display the elements from the secondary diagonal. If not, print the sum of all the elements from a certain valid column, c. The values of m, n and c are read from the keyboard. All the operations should be implemented in specific functions.
13. Starting from example 5 (3D arrays) which shows how to display a three-dimensional array from the perspective of xOy, write a program that goes through the array considering the perspective xOz and yOz. Display the data specifying from which perspective they were browsed.
14. Display each occurrence of each character in the English alphabet from a given string in old telegraphy related Morse code (example for testing purposes "The quick brown fox jumps over the lazy dog."). On a line there will be a triplet of the string character from the beginning of the given string, the offset inside the string where it was found (relative position to the beginning), and the Morse code. Display the initial test string with the first occurrence of each character converted to uppercase if necessary (not uppercase, or other non-alphabetical characters - THE QUICK BROWN FoX JuMPS oVer the LAZY DoG.).
15. Write a C/C ++ application in which an array of maximum 10 integer values in the range 2, ..., 10 is read from the keyboard. The actual size is entered from the keyboard. Define two functions that receive the array as a parameter and the actual dimension, and the functions return the minimum and maximum values of the array. Define a third function,

olympic_filter( ), which considers as parameters the initial array and its size, and the minimum, and maximum values, returning the average of the elements in the array without the first occurrence of the minimum and maximum value. In main( ) display the initial array, the minimum and maximum value, the average returned by the olympic_filter( ) with an accuracy of 2 decimals at the fractional part. Validate the data entry (array size, array elements entered, with resume).

6. Pointeri și operații cu ei. Transferul prin adresă a parametrilor către funcții. Referințe

Pointers. Operations using pointers. Parameter passing by address. References

6.1. Obiective

- Înțelegerea noțiunilor de pointer și referință și a modalității de definire și utilizare a lor;
- Familiarizarea cu operațiile cu pointeri;
- Scrierea și rularea de programe în care sunt folosiți pointeri, referințe și utilizarea lor în transferul argumentelor către funcții;

6.2. Objectives

- Understanding the concepts of pointer and reference and the method of defining and using them;
- Familiarization with pointer operations;
- Writing and running programs in which pointers, references, and their use in transferring arguments to functions are utilized;

6.3. Breviar teoretic

Un pointer (indicator) este o variabilă care are ca valoare o adresă, deci un pointer reprezintă adresa de început a unei zone de memorie asociate unui obiect (variabilă, tablou sau o structură de date mai complexă).

Tipuri de pointeri :

- **pointeri de date**: conțin adresa unor variabile sau constante din memorie;
- **pointeri spre funcții**: conțin adresa codului executabil din funcții;
- **pointeri generici** sau **pointeri void**: conțin adresa unui obiect oarecare.
- **pointeri spre obiecte:** conțin adresa unui obiect instanțiat
- **pointeri smart**: permit eliberarea automată a memoriei prin păstrarea unui contor de utilizare și dealocarea memoriei când acesta ajunge la 0. Au fost introduși recent, în C++1y.

Pointerii de date se declară astfel:

```
tip_date * nume_pointer;
```

ceea ce înseamnă că `nume_pointer` este numele unei variabile pointer care va conține adresa unei zone de memorie ce conține un obiect de tipul `tip_date`.

Pointerii de date pot fi asociați mai multor variabile la momente diferite de timp (reutilizați în program) și deci pot conține adrese diferite.

6.3.1. Operatori specifici pentru pointeri

Operatorul de adresare (referenţiere), *&*, asociat variabilei sau mai general obiectului, obține adresa acelei variabile sau obiect.

Exemplu:

```
char c, * pc;
//lui pc îi asignăm adresa variabilei c (pc devine pointer la variabila c)
pc=&c;
```

Operatorul de indirectare (sau dereferențiere), *, asociat unui pointer, permite accesul la conținutul locației de memorie a cărei adresă este conținută de pointer.

Exemplu:

```
int k=1, j=5, *p;
p=&k;       // asignează lui p adresa lui k
*p=2;       // valoarea variabilei k se schimba din 1 în 2
```

Indirectarea este un mecanism puternic de acces la memorie.

6.3.2. Operații cu pointeri

Cu pointerii se pot efectua operații de: atribuire, comparare, adunare, scădere, incrementare, decrementare. Se ține cont de tipul pointerilor și de faptul că adresele conținute de pointeri au valori numerice întregi fără semn.

Atribuirea: e permisă atribuirea valorii unui pointer la un alt pointer, dacă tipurile lor sunt identice sau dacă unul din pointeri este de tipul `void`, iar celălalt de tip oarecare.

Compararea a doi pointeri: se face cu operatorii relaționali, dar numai în cazul în care pointerii conțin adrese ale unor obiecte de același tip.

Adunare/scădere de întregi și **incrementare/decrementare:** la un pointer nu pot fi adunate sau scăzute decât valori întregi. Adunarea pointerilor între ei nu este permisă. Operațiile se efectuează relativ la tipul pointerului (`int`, `float`, `char`, etc.). De exemplu:

Fie declarația: `tip * id_ptr;`

Operațiile: `id_ptr + n` și `id_ptr - n`

înseamnă adunarea/scăderea la conținutul variabilei `id_ptr` a valorii: `n * sizeof(tip)`.

Analog se efectuează și operațiile de incrementare/decrementare, doar că n = 1.

Operația de incrementare/decrementare se poate aplica:

- asupra pointerului însuși (selectăm zona de memorie următoare/anterioară de aceeași dimensiune cu a tipului de date a pointerului; într-un tablou ar fi un element adiacent celui a cărui adresă se află stocată în variabila pointer);
- asupra obiectului stocat la adresa din variabila pointer.

Scăderea a doi pointeri: este permisă doar scăderea a doi pointeri la obiecte de același tip, rezultatul fiind o valoare care reprezintă diferența de adrese divizată la dimensiunea tipului de bază. (e recomandat ca aceste adrese sa fie specifice unui tablou). Practic, prin diferență obținem numărul de elemente dintre cele două adrese.

Datorită rolului tipului pointerului la adunare și scădere, operanzii nu pot fi pointeri `void` sau pointeri spre funcții.

6.3.3. Referințe

Referințele, la fel ca și pointerii, conțin adrese de memorie. Se pot utiliza doar în C++, nu și în C.

Acestea se declară cu ajutorul operatorului de adresare "&" și se inițializează obligatoriu la declarare cu o adresă (a unei variabile sau a unei constante):

```
int i;
int &r = i;
```

Accesul la o variabilă prin intermediul unei referințe se face simplu, fără a mai folosi operatorul de indirectare necesar în cazul pointerilo

```
int i;                              int i;
int *p;   //declarare              int &r = i; //definire (initializare)
...                                 ...
p = &i;   //definire prin asignare
```

```
*p = 100; //i = 100                    r = 1000;    //i = 1000
```

Referința conține tot timpul adresa aceleiași variabile, fiind de fapt o redenumire (alias) a variabilei.

Nici o referință nu poate avea valoarea `null`.

Uzual, referințele se folosesc la transmiterea parametrilor prin referință, caz în care indirectarea este ascunsă și nu este necesară dereferențierea în funcție (utilizarea operatorului *).

6.3.4. Apelul prin adresă utilizând parametri de tip pointeri și referință

În cadrul limbajului C/C++ transferul parametrilor către funcții este implicit efectuat prin valoare și constă în încărcarea valorilor parametrilor efectivi în zona de memorie a parametrilor formali. Dacă parametrul efectiv este o variabilă, orice operație efectuată în funcție asupra parametrului formal nu afectează variabila. Spunem că se lucrează cu copii ale parametrilor actuali, adică variabile locale pentru funcție. Acest lucru este o protecție utilă în cele mai multe cazuri.

Dacă vrem ca o funcție să modifice o variabilă parametru, atunci trebuie să transmitem funcției adresa variabilei, deci ca argumente se folosesc adrese, iar ca parametri formali se folosesc pointeri sau referințe (în C++) unde se vor copia aceste adrese.

6.4. Theoretical brief

A pointer is a variable that holds an address as its value. Thus, a pointer indicates the starting address of a memory zone associated with an object (variable, array, or a more complex data structure).

Types of pointers:

- **data pointers**: hold the address of variables or constants in memory.
- **function pointers**: hold the address of executable code within functions.
- **generic or void pointers**: hold the address of an arbitrary object.
- **object pointers:** hold the address of an instantiated object
- **smart pointers**: allow automatic memory release by keeping a reference count and automatic memory deallocation when the reference count reaches 0, recently introduced in C++1y.

Data pointers are declared as follows:

```
data_type *pointer_name;
```

This means that `pointer_name` is a pointer variable that will hold the address of a memory zone containing an object of type `data_type`. Data pointers can be associated with various variables at different times (reused in the program) and therefore can hold different addresses.

6.4.1. Specific Operators for Pointers

The address (referencing) operator, &, when associated with the variable or more generally with the object name, retrieves the address of that variable or object.

Example:

```
char c, * pc;
// assign the address of variable c to pc
// pc becomes a pointer to variable c
pc=&c;
```

The indirection (or dereferencing) operator, *, associated with a pointer, allows access to the content of the memory location whose address is held by the pointer.

Example:

```
int k=1, j=5, *p;
```

```
p=&k;         // assigns the address of k to p
*p=2;         // changes the value of variable k from 1 to 2
```

Indirection is a potent mechanism for memory access.

6.4.2. Operations with Pointers

Various operations can be performed with pointers, including: assignment, comparison, addition, subtraction, incrementing, and decrementing. It's crucial to consider the type of pointers and the fact that addresses contained by pointers have unsigned integer numeric values.

Assignment: The value of one pointer can be assigned to another pointer if their types are identical or if one of the pointers is of the void type and the other is of any type.

Comparison of two pointers: It is done with relational operators, but only when the pointers point to objects of the same type.

Addition/subtraction of integers and **incrementing/decrementing**: Only integer values can be added to or subtracted from a pointer. Adding pointers to each other is not allowed. Operations are performed relative to the type of the pointer (`int`, `float`, `char`, etc.). For example:

Given the declaration: `type *id_ptr;`

The operations: `id_ptr + n` and `id_ptr - n`

imply the addition/subtraction to the content of the variable `id_ptr` of the value `n * sizeof(type)`.

Analogously, the increment/decrement operations are performed, except that n = 1.

The increment/decrement operation can be applied:

- to the pointer itself (selecting the next/previous memory zone of the same size as the data type of the pointer; in an array, it would be an element adjacent to the one to which the pointer points);
- to the object to which it points.

Subtraction of two pointers: It is only permissible to subtract two pointers to objects of the same type, the result being a value representing the difference of addresses divided by the size of the base type. It is recommended that these addresses be specific to an array. Practically, through difference, we obtain the number of elements between the two addresses.

Due to the role of the pointer type in addition and subtraction, operands cannot be *void pointers* or *function pointers*.

6.4.3. References

References, like pointers, hold memory addresses. They can only be used in C++, not C.

They are declared using the "&" addressing operator and must be initialized with an address (of a variable or constant) upon declaration:

```
int i;
int &r = i;
```

Accessing a variable via a reference is straightforward, without needing to use the indirection operator as in the case of pointers:

```
int i;
int *p; // declaration
...
// definition through assignment
p = &i;
```

```
int i;
int &r = i; // definition (init)
...
```

```
*p = 100; // i = 100                    r = 1000;    // i = 1000
```

A reference always contains the address of the same variable, essentially being a renaming (alias) of the variable. No reference can have a null value. References are typically used for pass-by-reference parameter transmission, wherein indirection is hidden and dereferencing in the function (using the * operator) is not necessary.

6.4.4. Call by Address Using Pointers and Reference Type Parameters

In C/C++, parameter transfer to functions is implicitly performed by value, consisting of loading the values of the actual parameters into the memory area of the formal parameters. If the actual parameter is a variable, any operation performed within the function on the formal parameter does not affect the variable. We say that we work with copies of the actual parameters, i.e., local variables for the function. This is useful protection in most cases.

If we want a function to modify a parameter variable, then we must pass the address of the variable to the function. Therefore, addresses are used as arguments, and pointers or references (in C++), where these addresses will be copied, are used as formal parameters.

6.5. Exemple / Examples

6.5.1. Ex. 1

```
/* Pointer declaration, using addressing and indirection operators */
```

a) 4 bytes integer variant:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main( ) {
    int i = 10, j = 50;

    //  Pointer declaration
    int *iptr;

    //  Define pointer by address assignment
    iptr = &i;

    printf("Address of i stored by iptr, in hex: %llx and pointer: %p\n",
        iptr, iptr);

    printf("Value at the address stored by iptr: %d\n", *iptr);

    //  By changing the value stored at the address stored by iptr, we change
    //the value of variable i.
    *iptr = 25;
    printf("The new value of i is:  %d\n", i);//i=25

    // Reassign the pointer to point to j
    iptr = &j;
    printf("\nAddress of j stored by iptr, in hex: %llx and pointer: %p\n"
           , iptr, iptr);
    printf("Value at the address stored by iptr: %d\n", *iptr);

    // Using pointer operations to change the value stored by variable j
    *iptr = 25;//j=25
```

```
    printf("The new value of j is:  %d\n", j);
    return 0;
}//main
```

b) 8 bytes integer variant:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <inttypes.h>

int main( ) {
    int64_t i = 10, j = 50;

    // Pointer declaration
    _int64 *iptr;

    // Define pointer by address assignment
    iptr = &i;

    printf("Address of i stored by iptr, in hex: %llx and pointer: %p\n",
        iptr, iptr);

    printf("Value at the address stored by iptr: %lld\n", *iptr);

    //  By changing the value stored at the address stored by iptr, we change
    //the value of variable i.
    *iptr = 25;
    printf("The new value of i is:  %lld\n", i);

    // Reassign the pointer to point to j
    iptr = &j;
    printf("\nAddress of j stored by iptr, in hex: %llx and pointer: %p\n"
           , iptr, iptr);
    printf("Value at the address stored by iptr: %lld\n", *iptr);

    // Using pointer operations to change the value stored by variable j
    *iptr = 25;
    printf("The new value of j is:  %lld\n", j);
    return 0;
}//main
```

6.5.2. Ex. 2

```
/* Pointers and cast operator */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

void base2(unsigned int);//prototype recursive base 2 display

int main( ){
  int n, *pn;
  float r, *pr;

  printf("\nType an integer number: ");
  scanf("%d", &n);
```

```
  pn = &n;
  printf("\nThe in memory representation of %d is %x in hex and in bytes: 0x%02x
%02x %02x %02x\n",
    n,
    n,
    *((unsigned char *)pn+3),
    *((unsigned char *)pn+2),
    *((unsigned char *)pn+1),
    *((unsigned char *)pn));

  printf("\n The bits in base 2 (binary): ");
  base2(n);

  printf("\nType a floating point number (numar real): ");
  scanf("%f", &r);
  pr = &r;

  printf("\nThe in memory representation of %f is %a in hex and in bytes: 0x%02x
%02x %02x %02x\n",
    r,
    r,
    *((unsigned char *)pr+3),
    *((unsigned char *)pr+2),
    *((unsigned char *)pr+1),
    *((unsigned char *)pr));

  return 0;
}//main

void base2 (unsigned int a){
  if(a!=0)
    base2(a>>1);
  printf("%d ", a%2);
}//base2
```

6.5.3. Ex. 3

```
/* Declaring and making use of different types of pointers */

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

int main( ){
  int   *pt_int;
  float *pt_float;
  int   i = 10, j = 20;
  float x = 1.2345f, y = 32.14f;
  void *general;  // generic pointer

  pt_int = &i;
  *pt_int += j;
  cout << "The value of i becomes: " << *pt_int << "\n";
  general = pt_int;
  *(int *)general = 0;
  cout << "The last value for i is: " << i << "\n";
```

```
  pt_float = &x;
  y += 5.0f * (*pt_float);
  cout << "The value of y is:  " << y << "\n";

  general = pt_float; // assigning the address of x

  *(float *)general = 1.1f;
  cout << "The last value for x is: " << x << "\n";

  return 0;
}//main
```

6.5.4. Ex. 4

```
/* Passing parameters by value and addresses using functions */

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

void swap1 (int x, int y);  //pass by value
void swap2 (int *x, int *y);//pass by address using pointers
void swap3 (int &x, int &y);//pass by address using references

int main ( ){
  int i = 10, j = 20;

  swap1(i, j); // Pass by value
  cout << "Pass by value : i = " << i << ", " << "j = " << j << '\n';

  swap2(&i, &j); // Pass by address using pointers
  cout << "Pass by address using pointers: i=" << i << ", " << "j=" << j << '\n';

  swap3(i, j); // Pass by address using references
  cout << "Pass by address using references: i=" << i << ", " << "j=" << j << '\n';
  return 0 ;
}//main

void swap1 (int x, int y)
{
  int temp = x;
  x = y;
  y = temp;
}// swap1

void swap2 (int *x, int *y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}// swap2

void swap3 (int &x, int &y)
{
  int temp = x;
```

```
  x = y;
  y = temp;
}// swap3
```

6.6. Întrebări

1. Ce este un pointer?
2. Câte tipuri de pointeri există în limbajul C/C++?
3. Cum se declară și se inițializează un pointer?
4. Ce operator se folosește pentru a obține adresa unui obiect?
5. Ce operator se folosește pentru a afla valoarea memorată într-o locație de memorie dată prin adresă?
6. Ce este o referință?
7. Cum se declară o referință și cum se inițializează?
8. Care sunt asemănările între pointeri și referințe?
9. Care sunt diferențele între pointeri și referințe?
10. Ce operații se pot face asupra pointerilor?
11. Ce se înțelege prin transferul parametrilor prin adresă?
12. Cum se face transferul parametrilor prin adresă?

6.7. Questions

1. What is a pointer?
2. How many types of pointers are supported in C/C++?
3. How does one declare and initialize a pointer?
4. Which operator is used to obtain the address of an object?
5. Which operator is used to retrieve the value stored at a memory location, given by an address?
6. What is a reference?
7. How is a reference declared and initialized?
8. What are the differences between pointers and references?
9. What operations can be performed on pointers?
10. What is meant by the passing (or transfer) of parameters by address?
11. How is the transfer of parameters by address achieved?

6.8. Lucru individual

1. Să se scrie un program C/C++ care citește elementele a două tablouri unidimensionale de numere întregi cu dimensiunea <=7 și afișează produsul scalar al acestora. Se va folosi o funcție care preia elementele de la tastatură și o altă funcție, care calculează produsul scalar. Ambele vor utiliza pointeri. Citirea si validarea numărului de elemente ale tabloului și afișarea rezultatului se va face în funcția `main()`.
2. Să se scrie o aplicație C/C++ în care se generează aleator 20 de numere întregi cu valori mai mici decât 50 (folosiți `srand()`, `rand()` si operatorul `%`). Să se scrie o funcție care elimină din tabloul unidimensional creat numerele impare, returnând noul tablou. Funcția va utiliza pointeri. Afișați în `main()` tabloul inițial și cel obținut după eliminarea elementelor impare.
3. Să se scrie un program C/C++ în care se citesc de la tastatură numere reale, ce vor fi stocate într-un tablou unidimensional. Să se scrie o funcție care copiază într-un alt tablou toate valorile din primul tablou, care sunt mai mari decât valoarea medie a numerelor preluate. Se vor folosi pointeri și se vor afișa în `main()` valorile din cele două tablouri.
4. Să se scrie un program care citește de la tastatură un șir de caractere, apoi elimină din șir caracterele care se repetă și afișează în final șirul obținut, folosind pointeri.

5. Să se scrie un program care citește de la tastatură două șiruri de caractere și afișează numărul de caractere prin care ele diferă (adică numărul de caractere care există în primul și nu există în al doilea + numărul de caractere care există în al doilea și nu există în primul). Folosiți pointeri pentru accesul la elementele tablourilor.
6. Să se scrie o aplicație C/C++ care citește de la tastatură un șir de caractere. Să se scrie o funcție care afișează caracterele ce compun șirul și numărul de apariții ale fiecăruia, folosind pointeri.

6.9. Individual work

1. Write a C/C++ application that reads from the keyboard two one-dimensional arrays of integers, dimension <=7, and displays the scalar product of the two arrays. Use a function that reads the elements from the keyboard and another function that calculates the scalar product, using pointers in both functions. Reading and validation of array dimension and displaying the arrays' elements should be done in the main() function.
2. Write a C/C++ application that generates 20 random numbers, each smaller than 50 (use `srand()`, `rand()` and `%` operator). Write a function that eliminates the odd elements from the one-dimensional array (using pointers), returning the new array. Display both the initial and the final array in the `main()` function.
3. Write a C/C++ program that fills-up a float-type, one-dimensional array with values read from the keyboard. Write a function that copies into another array the values greater than the average of all elements from the array, by using pointers. Both arrays should be displayed in the `main()` function.
4. Write a C/C++ application that reads from the keyboard an array of characters and displays the string obtained by eliminating the characters that appear more than once in the original array using pointers.
5. Write a C/C++ application that reads from the keyboard two arrays of characters and displays the total number of individual characters (the number of characters that are in the first array and do not exist in the second one + the number of characters that are in the second array and do not exist in the first one). Use pointers for accessing the arrays of elements.
6. Write a C/C++ program that reads from the keyboard an array of characters. Write a function that displays the characters that are in the array and the number of times they appear. Use pointers.

7. Pointeri și tablouri. Transferul de argumente către funcția main(). Pointeri spre funcții

Pointers and arrays. Passing of arguments to main() function. Pointers and functions

7.1. Obiective

- Înțelegerea legăturii dintre pointeri și tablouri.
- Înțelegerea modului în care se transmit parametrii din linia de comandă.
- Utilizarea pointerilor spre funcții
- Scrierea și rularea de programe care exploatează aceste facilități.

7.2. Objectives

- Understanding the relation between pointers and arrays.
- Understanding the way, the command line arguments are transmitted.
- Usage of pointers to functions.
- Writing and running programs that make use of these functionalities.

7.3. Breviar teoretic

Un nume de variabilă tablou, atunci când este folosit fără operatorul de indexare, este un identificator care specifică adresa primului element din tablou. El poate fi tratat ca un pointer constant către primul element din tablou, deoarece comportamentul este în mare parte același.

Consecințe:

- Adresa tabloului, dată de numele lui, poate fi atribuită unui alt pointer (ne-constant, de același tip), pointer ce poate fi utilizat pentru accesul la elementele tabloului;
- Un pointer se poate indexa ca și când ar fi un tablou, dar dacă nu conține adresa unui tablou rezultatul este imprevizibil;
- Dacă `p` conține adresa unui tablou, compilatorul C/C++ generează cod executabil mai scurt pentru `*(p+i)` decât pentru `p[i]` (compilatoarele moderne – cu mai multe treceri, sunt capabile a optimiza codul automat, iar această regulă nu mai e neapărat valabilă);
- Numele unui tablou specifică adresa primului element din tablou, adresă care nu se poate modifica. Acesta poate fi folosit ca un pointer aritmetic (adunând la el o variabilă cu rol de index. Vezi Exemplul 1).

7.3.1. Tablouri de pointeri

Tablourile de pointeri se definesc similar cu tablourile altor tipuri predefinite și se folosesc mai ales la crearea de tabele de pointeri către șiruri de caractere care pot fi selectate în funcție de anumite valori întregi fără semn, cu rol de indice:

```
const char *p[] = { "Out of range", "Disk full", "Paper out" };
```

Tablourile de pointeri pot fi accesate și cu pointeri dubli.

7.3.2. Pointerii și modificatorul const

Există pointeri către constante și pointeri constanți, care nu pot fi modificați.

- pointeri către constante:

```
const char *str1 = "pointer catre sir de caractere constant";
//str1[0] = 'P';   // incorect: sirul fiind declarat imutabil (const)
```

```
str1 = "ptr la const"; //OK: pointerul nu e declarat const, poate fi
folosit pentru a indica spre un alt sir
```

- pointeri constanți către șiruri de caractere (nu sunt acceptate de noile compilatoare C++). O valoare de tip `const char*` nu poate fi folosită pentru a inițializa o entitate de tip `char * const`, acceptată de compilatoarele C:

```
char *const str2 = "pointer constant";
//str2 = "ptr to const";  // incorect, str2 fiind constant
str2[0] = 'P';    // ok, sirul nu e constant, dar functioneaza doar in C
```

- pointeri constanți către constante:

```
const char *const str3 = "pointer constant la constanta";
//str3 = "ptr const la const";      // incorect
//str3[0] = 'P';                    // incorect
```

7.3.3. Indirectarea multiplă

Când un pointer conține adresa unui alt pointer avem un proces numit indirectare dublă (multiplă). El arată astfel: Pointer către pointer -----> Pointer ----> Obiect. Primul pointer conține adresa celui de-al doilea pointer, care conține adresa de memorie unde se află stocat obiectul. Declararea se face utilizând un asterisc suplimentar în fața numelui pointerului.

7.3.4. Pointeri spre funcții

Unei funcții, ca și unei structuri sau unui tablou, sistemul de operare îi alocă o adresă de memorie fizică. Un pointer către o funcție va conține această adresă.

Declararea unui pointer spre o funcție trebuie să precizeze tipul returnat de funcție și lista de parametri formali:

```
tip_rezultat (*pf)(lista_param_formali);
```

Regulă practică: declarația se face ca și cum am scrie un prototip de funcție, numele funcției fiind substituit de construcția (`*nume_pointer_la_funcție`).

Numele unei funcții este sinonim cu adresa de început a codului său executabil în memorie, astfel că:

`pf = nume_functie;` este sinonimă cu `pf = &nume_functie;`

Apelul unei funcții prin intermediul unui pointer se face cu construcția:

```
(*pf)(lista_param_actuali);
```

iar dacă funcția returnează o valoare care se poate atribui unei variabile (funcție cu tip), atunci apelul poate fi:

```
var = (*pf)(lista_param_actuali);
```

Standardul C++ permite simplificarea apelului folosind pointeri la funcții astfel:

```
var = pf(lista_param_actuali);
```

În biblioteca C/C++ (`stdlib.h`) există funcția `qsort()` care este folosită pentru a sorta un tablou:

```
void qsort(
    void *base, size_t nitems, size_t size,
    int (*compar)(const void *, const void*)
);
```

unde:

base - este indicatorul către primul element al tabloului care urmează să fie sortat,

nitems - este numărul de elemente din tablou,

size - este dimensiunea în octeți a fiecărui element din tablou,

compar - este funcția care compară două elemente (se folosește ca pointer spre ea).

7.3.5. Transferul de argumente către funcția main()

La lansarea în execuție, multe aplicații permit specificarea unor argumente în linia de comandă.

Programele scrise în limbajul C/C++ permit introducerea de argumente în linia de comandă, prin definirea unor parametri pentru funcția `main()`:

```
int main([ int argc, char *argv[] [ ,char *env[] ]]) { ... }
```

- argc, „argument count", conține numărul argumentelor (>= 1)
- argv, „argument vector" este un tablou de pointeri către șiruri de caractere, unde:
  - argv[0] conține adresa șirului de caractere cu numele și calea programului care se lansează în execuție
  - argv[1] conține adresa șirului de caractere ce reprezintă primul argument din linia de comandă, ș.a.m.d.
- env, „environment variables" este un tablou de pointeri către șiruri de caractere care reprezintă o listă de parametri ai SO (tipul prompterului,...); ultimul pointer are valoarea NULL, marcând sfârșitul listei de parametri.

Caracteristici:

Între argumente se consideră ca separator caracterul spațiu și tab, iar dacă un argument va conține spațiu (tab), acel argument trebuie încadrat între ghilimele.

Numele `argc`, `argv`, `env` nu sunt impuse, utilizatorul poate folosi și alte nume.

Pentru că aceste argumente se primesc sub forma de șiruri de caractere, aceste șiruri pot fi convertite spre un format intern de reprezentare în memorie cu funcțiile `atoi()`, `atof()`, `atol()`.

Indiferent de modul de utilizare a funcției `main()` de către utilizator, cei 3 parametri sunt alocați pe stivă.

Testarea programelor ce transferă argumente din linia de comandă se poate face în două moduri:

1. din sistemul de operare, după ce s-a creat fișierul executabil, lansând programul folosind opțiunea Start->Run și apoi selectând fișierul executabil aferent programului, după care se tastează argumentele, separate prin spațiu; pentru fiecare lansare în execuție trebuie repetată această operație.
2. din mediul de programare, stabilind în setările proiectului curent argumentele ce vor ajunge la funcția main( ); setările respective și modul în care se ajunge la acestea depinde de mediul de programare cu care se lucrează.

7.4. Theoretical brief

The name of an array variable, when used without the indexing operator, is an identifier that specifies the address of the array's first element. It can be treated as a constant pointer to the first element of the array, as the behavior is largely the same.

Consequences:

- The address of the array, given by its name, can be assigned to another (non-constant, same-type) pointer, which can be used to access the array's elements;
- A pointer can be indexed as if it were an array, but if it doesn't point to an array, the result is unpredictable;

- If `p` points to an array, the C/C++ compiler generates shorter executable code for `*(p+i)` than for `p[i]` (modern compilers – multi-pass compilers, are capable of optimizing code automatically, so this rule is not necessarily applicable);
- The name of an array variable specifies the address of the first element in the array, an address that cannot be modified, and it can be used as an arithmetic pointer (adding to it a variable acting as an index). See Example 1.

7.4.1. Array of pointers

Arrays of pointers are defined similarly to arrays of other predefined types and are mainly used to create tables of pointers to strings that can be selected based on certain unsigned integer values, serving as indices:

```
const char *p[] = { "Out of range", "Disk full", "Paper out" };
```

Arrays of pointers can also be accessed with double pointers.

7.4.2. Pointers and the const modifier

In C/C++ there are pointer to constants and constant pointers, which cannot be modified:

- pointers to constants:

```
const char *str1 = "pointer to a constant string";
//str1[0] = 'P';  // incorrect: the string is declared immutable (const)
str1 = "ptr to const";//OK: the pointer is not declared const, it can be used for another string
```

- Constant pointers to a string (not accepted by new C++ compilers - A value of type “const char*” cannot be used to initialize an entity of type “char * const” – Accepted by C compilers)

```
char *const str2 = "constant pointer";
//str2 = "ptr to const";  // incorrect, str2 is constant
str2[0] = 'P';    // ok, the string is not constant, but only works in C
```

- constant pointers to constants:

```
const char *const str3 = "constant pointer to a constant";
//str3 = "const ptr to const";      // incorrect
//str3[0] = 'P';                    // incorrect
```

7.4.3. Multiple indirection

When one pointer points to another pointer, we have a process called double (multiple) indirection. It looks like this: Pointer to pointer -----> Pointer ----> Object. When a pointer points to another pointer, the first pointer contains the address of the second pointer, which points to the location containing the object. The declaration is made using an additional asterisk in front of the pointer name.

7.4.4. Pointers to functions

Similar to a structure or an array, an operating system allocates a physical memory address to a function. A pointer to a function will contain this address.

Declaring a pointer to a function must specify the type returned by the function and the list of formal parameters:

```
return_type (*pf)(list_of_formal_parameters);
```

Practical Rule: The declaration is made as if writing a function prototype, with the function name being replaced by the construct `(*function_pointer_name)`.

The name of a function is synonymous with the starting address of its executable code in memory, so:

`pf = function_name;` is synonymous with `pf = &function_name;`

Calling a function through a pointer is done with the construct:

```
(*pf)(list_of_actual_parameters);
```

and if the function returns a value that can be assigned to a variable (typed function), then the call can be:

```
var = (*pf)(list_of_actual_parameters);
```

C++ standard allows simplifying calls using function pointers as follows:

```
var = pf(list_of_actual_parameters);
```

In the C/C++ library (`stdlib.h`), there is the `qsort()` function, which is used to sort an array:

```
void qsort(
    void *base, size_t nitems, size_t size,
    int (*compar)(const void *, const void*)
);
```

where:

base is the pointer to the first element of the array to be sorted,

nitems is the number of elements in the array,

size is the size in bytes of each element in the array,

compar is the function that compares two elements (used as a pointer to it).

7.4.5. Transfer of arguments to main() function

Upon execution, many applications allow specifying arguments through the command line.

Programs written in C/C++ allow the introduction of command line arguments by defining parameters for the main() function:

int main([int argc, char *argv[] [, char *env[]]]) { ... }

- argc, or "argument count", contains the number of arguments (≥ 1).
- argv, or "argument vector", is an array of pointers to character strings, where:
  - argv[0] points to the name and path of the program being executed,
  - argv[1] points to the first argument from the command line, and so on.
- env, or "environment variables", is an array of pointers to character strings representing a list of OS parameters (like the type of the prompter...). The last pointer is NULL, marking the end of the parameter list.

Characteristics:

Space and tab characters are considered separators between arguments. If an argument contains a space (tab), it should be enclosed in quotes.

The names `argc`, `argv`, and `env` are not mandatory; users can utilize other names.

Since these arguments are received in the form of character strings, they can be converted to an internal memory representation format with functions such as `atoi()`, `atof()`, and `atol()`.

Regardless of how the user utilizes the `main()` function, the three parameters are allocated on the stack.

Testing of programs that transfer arguments from the command line can be done in two ways:

1. From the operating system: After the executable file has been created, launch the program using the Start -> Run option and then select the corresponding executable file. Afterward, type the arguments separated by space. This operation must be repeated for each execution.
2. From the development environment: Establish in the current project settings the arguments that will reach the main() function. These particular settings and how to access them depend on the development environment being used. Once set, these settings are valid for all programs being tested and for any execution. (See Visual Studio settings for instance).

7.5. Exemple / Examples

7.5.1. Ex. 1

```
/* Program for determining the minimum value in a one-dimensional array
*/

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#define DIM 20

// Prototype of function to determine the minimum value from an array
int detMin(int *, int);

int main( ){
    int i, dim, min;
    int x[DIM]={};//int x[DIM{};
    do
    {
    printf("\n Type the size of the one-dimensional array: ");
    scanf("%d", &dim);
    if (dim <= 0 || dim > DIM)printf("\nIncorrect size (must be >0 and <=%d !", DIM);
    } while (dim <= 0 || dim > DIM);

    printf("\n Type the elements of the one-dimensional array:\n");
    for(i=0; i<dim; i++)
    {
        printf("\tx[%d] = ", i);
        scanf("%d", (x+i));
    }
    min = detMin(x, dim);

    printf("\n The minimum value of the one-dimensional array is: %d\n", min);
    return 0;
}//main

int detMin(int *x, int n){
    int i, min;

    min = *x++;
    for(i=1; i<n; i++)
```

```
        if(*x < min) min = *x++;
        else x++;
    return min;
}//detMin
```

7.5.2. Ex. 2

```
/*  Program for showcasing the access with array of pointers to character arrays
(strings) using double pointers.
*/

#define _CRT_SECURE_NO_WARNINGS
#include <cstdio>
void err(const char **, int nr_err);
const int dim = 20;

int main( ) {
    const char* p[] = { "Ok\n", "Disk full\n", "Paper out\n" };
    err(p, 0); err(p, 1); err(p, 2);
    char s1[dim], s2[dim], s3[dim];
    printf("\nEnter first string: ");
    scanf("%s", s1);
    printf("\nEnter second string: ");
    scanf("%s", s2);
    printf("\nEnter third string: ");
    scanf("%s", s3);
    const char* pp[] = {s1,s2,s3};
    err(pp, 0); printf("\n");
    err(pp, 1); printf("\n");
    err(pp, 2); printf("\n");
}//main

void err(const char ** p, int nr_err) {
      printf(p[nr_err]);
}//err
```

7.5.3. Ex. 3

```
/*  Passing an array of pointers to character arrays to a function by double
pointer and by reference to a pointer.
*/

// double pointer C - pointer reference CPP
#include <iostream>
using namespace std;

const char* someFuncC(const char ** p2c); //double pointer parameter
const char* someFuncCPP(const char * &r); //reference to pointer parameter
const int dim = 10;

int main( ) {
    const char* sir[dim] =
        { "aa", "bbb", "cc", "dddd", "ee", "ff", "ggg", "hh", "iii", "jj" };
    const char* res;
    cout << "\nCall C double pointer function" << endl;
```

```
    res = someFuncC(sir);
    cout << res << " sir[0]: " << sir[0] << endl;
    cout << "\nCall CPP pointer reference function" << endl;
    res = someFuncCPP(*sir);
    cout << res << " sir[0]: " << sir[0] << endl;
    return 0;
}//main

const char* someFuncC(const char ** p2c){
    const char* q = "abc";
    *p2c = q;//Change the value of the pointer p2c itself by q which is an address
    return *p2c;
}// someFuncC

const char* someFuncCPP(const char * &r){
    const char* q = "cba";
    r = q;// Change the value of the reference r itself
    return r;
}// someFuncCPP
```

7.5.4. Ex. 4

```
/*  Program that makes use of pointers to functions.*/

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int get_result(int, int, int (*)(int, int));

int max(int, int);
int min(int, int);

int main( ){
    int result;

    // Call to get_result, passing the function max as parameter
    result = get_result(1, 2, max);
    printf("The maximum out of 1 and 2 is: %d\n", result);

    // Call to get_result, passing the function min as parameter
    result = get_result(1, 2, min);
    printf("The minimum out of 1 and 2 is: %d\n", result);
    return 0;
}//main

int get_result(int a, int b, int (*compare)(int, int))
{
    // Call to function using pointer
    return(compare(a, b));
}// get_result

int max(int a, int b)
{
    printf("Call to function max:\n");
    return((a > b) ? a: b);
}//max
```

```
int min(int a, int b)
{
    printf("Call to function min:\n");
    return((a < b) ? a: b);
}//min
```

7.5.5. Ex. 5

```
/*  Program that reads from the console multiple integers representing values of
resistors and computes the equivalent resistor for series or parallel grouping.
*/

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <ctype.h>
//#include <conio.h>

#define DIM 10

//functions prototypes
float calcul(int*, int, float (*)(int*, int));
float serie(int*, int);
float paralel(int*, int);

int main( ){
    int i, n, tab[DIM];
    char grp;
    float (*pf)(int*, int);

    printf("\n Type the number of resistors < =%d: ", DIM);
    scanf("%d", &n);
    if (n <= 0 || n>DIM){
        printf("\n Incorrect value supplied!\n");
        return 1;
    }

    printf("\n Type %d values representing values of resistors:\n", n);
    for (i = 0; i < n; i++)
    {
        printf("Resistor %d :", i);
        scanf("%d", &tab[i]);   //(tab+i)
    }

    printf("\n The type of the desired grouping is (s - series/p - parallel) ? ");
    //grp=_getche();
    //fflush(stdin);
    scanf(" %c", &grp);
    switch (grp) //toupper(grp) -  ctype.h
    {
        case 'S':
        case 's':    pf = serie; break;
        case 'P':
        case 'p':    pf = paralel; break;
        default:
            printf("\n Invalid value selected, quitting !");
```

```
       return 1; //non-zero exit code signals an error to the parent process
    }//end switch

    printf("\n The equivalent %s grouping has the resistance: %.2f\n",
        pf == serie ? "series" : "parallel",
        calcul(tab, n, pf));
    return 0;
}//main

float calcul(int* tab, int n, float (*pf)(int*, int))
{
    // we may add more common processing steps here (a validation or other)
    return(pf(tab, n));
}//calcul

float serie(int* tab, int n)
{
    float suma = 0.f;
    while (n)
        suma += tab[--n];
    return(suma);
}//serie

float paralel(int* tab, int n)
{
    float suma = 0.0f;
    while (n)
        suma += 1.0f / tab[--n];
    return(1.0f / suma);
}//paralel
```

7.5.6. Ex. 6

```
/*  Program that sorts an array of int or double using the qsort() function.
*/

//sorting array of integers/doubles with qsort( )
#include <stdio.h>
#include <stdlib.h>

const int dim = 10;
int comp_int(const void* a, const void* b);
int comp_double(const void* a, const void* b);

int main( ) {
    int i_numbers[dim] =
        {1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000};
    double d_numbers[dim] =
        {1892., 45., 200., -98., 4087., 5., -12345., 1087., 88., -100000.};
    int i;

    // Sort descending the int array
    qsort(i_numbers, dim, sizeof(int), comp_int);

    printf("\nOrdering descending int values:\n");
    for (i = 0; i < dim; i++)
```

```
        printf("Number = %d\n", i_numbers[i]);

    // Sort ascending the double array
    qsort(d_numbers, dim, sizeof(double), comp_double);
    printf("\nOrdering ascending double values:\n");
    for (i = 0; i < dim; i++)
        printf("Number = %.2lf\n", d_numbers[i]);

    return 0;
}//main

int comp_int(const void* a, const void* b) {
    return (*(int*)b - *(int*)a); //descending
}// comp_int

int comp_double(const void* a, const void* b) {
    if (*(double*)a > *(double*)b)
        return 1; //ascending
    else if (*(double*)a < *(double*)b)
        return -1;
    else
        return 0;
}// comp_double
```

7.5.7. Ex. 7

```
/*  Program that reads integers from the command line and shows their sum.
*/

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    int i, suma = 0, n;

    if (argc == 1) {
        printf("\n\n\tYou must specify the numbers to add in the command line!");
        exit(1);
    }//end if

    for (i = 1; i < argc; i++)
    {
        n = atoi(argv[i]);
        suma += n;
    }//end for
    printf("\nThe sum of the command line arguments is: %d\n", suma);
    return 0;
}//main
```

7.6. Întrebări

1. Care sunt operatorii specifici pentru pointeri?
2. Care sunt operațiile permise cu pointerii?
3. Care este diferența între un pointer constant și un pointer către o constantă?
4. Ce este un pointer către o funcție?
5. Cum se poate face apelul unei funcții folosind pointeri?

6. Ce reprezintă parametrii din linia de comandă?
7. Cum pot fi accesați parametrii din linia de comandă într-un program C/C++?

7.7. Questions

1. What are the operators that can be used with pointers?
2. What operations are allowed on pointers?
3. What is the difference between a constant pointer and a pointer to a constant?
4. What is a pointer to a function?
5. How can a function be called using pointers?
6. What do the parameters in the command line represent?
7. How can the parameters from the command line be accessed in a C/C++ program?

7.8. Lucru individual

1. Rezolvați minimum 3 probleme din laboratorul precedent de aplicații cu tablouri folosind accesul la elementele tabloului prin pointeri.
2. Se consideră doi parametri întregi și alți doi flotanți preluați din linia de comandă. Să se afișeze suma primelor 2 valori întregi și produsul ultimelor 2 valori de tip `float`.
3. Scrieți o aplicație care citește de la tastatură un șir de caractere cu lungimea mai mare decât 7. Folosiți un pointer pentru a accesa și afișa caracterele de pe pozițiile 1, 3, 5 și 7.
4. Citiți de la tastatură elementele a 2 matrice de valori întregi (pătrate). Scrieți o funcție care primește ca parametri pointerii la cele 2 matrice și returnează un pointer la matricea sumă (alocată dinamic). Rezultatul însumării matricelor va fi afișat în funcția main(). Afișați elementele de pe diagonala secundară a matricei sumă, folosind același pointer. Considerați și cazul generalizat în care matricile nu sunt pătrate și ajustați dimensiunile pentru a face operațiile cerute.
5. Definiți un tablou de pointeri către șiruri de caractere. Fiecare locație a tabloului conține adrese către unul din următoarele șiruri de caractere:
   - "valoare prea mică"
   - "valoare prea mare"
   - "valoare corectă"

   Aplicația generează un număr aleator între 1 și 100 și apoi citește în mod repetat intrarea de la tastatură până când utilizatorul introduce valoarea corectă. Folosiți mesajele definite pentru a informa utilizatorul, la fiecare pas, despre relația existentă între numărul generat și ultima valoare citită.
6. Scrieți un program în care să definiți un tablou de pointeri spre șiruri de caractere, pe care să-l inițializați cu diferite mesaje. Afișați mesajele.
7. Să se scrie un program care preia din linia de comandă șiruri de caractere și afișează șirul rezultat din concatenarea acestora.
8. Să se scrie un program care inversează șirul de caractere citit din linia de comandă.
9. Scrieți un program care citește de la tastatură elementele de tip float ale unui tablou unidimensional, elemente ce reprezintă mediile unei grupe de studenți. Să se scrie o funcție care determină numărul studenților cu media >= 8. Afișați rezultatul în `main()`. (lucrați cu pointeri, fără variabile globale).
10. Scrieți un program C/C++ în care se citesc de la tastatură elementele de tip întreg ale unui tablou unidimensional `arr_A`, utilizând o funcție. Scrieți o funcție care completează un alt tablou unidimensional `arr_B`, fiecare element al acestuia fiind obținut prin scăderea mediei aritmetice a elementelor din `arr_A` din elementul corespunzător din `arr_A`. Scrieți o altă funcție care permite afișarea unui tablou unidimensional și afișați tablourile unidimensionale `arr_A` și `arr_B`. (utilizând pointeri, fără variabile globale).

11. Scrieți un program în care se citesc de la tastatură elementele de tip întreg ale unei matrice pătratice, utilizând o funcție. Scrieți o funcție care determină numărul de elemente negative de deasupra diagonalei secundare. Afișați rezultatul în `main()` (fără variabile globale).
12. Scrieți un program în care se citesc de la tastatură elementele de tip întreg ale unei matrice pătratice, utilizând o funcție. Scrieți o funcție care interschimbă două linii ale matricei. Afișați cu o funcție matricea inițială și cea obținută. Dimensiunea matricei și numerele ce identifică liniile care vor fi interschimbate se citesc de la tastatură, în funcția `main()`. (fără variabile globale).
13. Considerând algoritmul care preia numerele de la tastatură direct ordonate crescător într-un tablou unidimensional, folosiți mecanismul de acces la elemente prin pointeri. Scrieți o aplicație C/C++ care implementează o funcție care are ca parametri formali un pointer la un tablou unidimensional de tip `float` și dimensiunea. (`void dir_sort(float *, int n);`)
14. Scrieți algoritmul care interclasează două tablouri unidimensionale sortate, de tip întreg. Folosiți pointeri.
15. Ordonați crescător un tablou unidimensional de numere întregi citit de la tastatură (sau generat aleator) folosind funcția de bibliotecă `qsort()` din `stdlib.h`. Folosiți același algoritm `qsort()` pentru valori de tip `float` pe care să le ordonați descrescător.

7.9. Individual work

1. Resolve minimum 3 problems referring to arrays from the previous array laboratory, using pointer access mechanism.
2. Considering two integers and two float parameters from the command line, display the sum of first two integers and the product of the last two float parameters.
3. Write a C/C++ application that reads from the keyboard an array of characters that has more than 7 elements. Use a pointer for displaying the characters that have the indexes 1, 3, 5 and 7.
4. Write a C/C++ application that reads from the keyboard the elements of 2 square integer matrices. Write a function that receives as parameters the pointers to the matrices and returns the pointer to the sum matrix (dynamically allocated). The result is to be displayed in function main(). Display the elements from the second diagonal of the sum matrix using the returned pointer. Also consider the generalized case where the matrices are not square and adjust the dimensions to do the required operations.
5. Define an array of character pointers. Each location will store one of the following messages:
   - "value too small"
   - "value too big"
   - "correct value"

   The application generates a random number between 0 and 100 and then reads repeatedly the keyboard until the user enters the right number. Use the previously defined messages for informing the client about the relation between the auto-generated number and the last value entered from the keyboard.
6. Write a C/C++ application that defines an array of pointers to character strings and initialize them with different messages. Display the messages.
7. Write a C/C++ program that reads some character arrays from the command line and displays the resulting concatenated string.
8. Write a C/C++ program that inverts a string of characters read from the command line.
9. Write a C/C++ program that reads from the keyboard the float type elements of a one-dimensional array. The values represent the average marks of a group of students. Write a function that determines the number of students having the average mark >= 8. Display the result in the main function. (use pointers, avoid global variables).

10. Write a C/C++ program that implements a function for reading from the keyboard some integer values. The function stores the values in a one-dimensional array named `arr_A`. Write another function that fills a different one-dimensional array `arr_B`, each element being obtained by subtracting the mean value of the initial values from the corresponding element located in the one-dimensional array `arr_A`. Write a function that displays a one-dimensional array and use it for `arr_A` and `arr_B` one-dimensional array. (use pointers, don't use global variables)
11. Write a C/C++ program that defines a function for reading from the keyboard the integer type values that form a [n x n] matrix. Write a function that determines the number of negative elements that are located above the secondary diagonal. Display the result in the main() function (don't use global variables).
12. Write a C/C++ program that defines a function for reading from the keyboard the integer type values that form a [n x n] matrix. Write a function that interchanges two lines of the matrix. Use another function for displaying the initial and the processed matrices. Read from the keyboard (in the main function) the dimension (n) of the matrix and the indexes that indicate the rows to be switched (do not use global variables).
13. Considering the algorithm that directly introduces the numbers from KB in a sorted mode in a one-dimensional array, use the mechanism to access by pointers the elements. Develop a C/C++ application considering a function having as formal parameters a pointer to a one-dimensional array of `float` type and the dimension. (`void dir_sort(float *, int n);`)
14. Develop the algorithm able to interclass two one-dimensional sorted arrays of int type. Use pointers.
15. Order in increasing mode a one-dimensional array of integer type introduced from the keyboard (or generated in random mode) using `qsort()` from `stdlib.h`. Use the same `qsort()` algorithm for float numbers and a decreasing order.

8. Alocarea dinamică a memoriei în C/C++. Gestiunea memoriei

Dynamic memory allocation in C/C++. Memory management

8.1. Obiective

- Înțelegerea noțiunii de alocare dinamică a datelor în memorie
- Scrierea și rularea de programe în care este folosită alocarea dinamică a datelor în memorie

8.2. Objectives

- Understanding how the data is dynamically allocated in memory
- Writing and running programs that make use of the dynamic memory allocation.

8.3. Breviar teoretic

Funcțiile cele mai utilizate în procesul de alocare dinamică a memoriei în limbajul C sunt:

`malloc()`

`calloc()`

`free()`

- Funcția *malloc()* are prototipul: `void * malloc(unsigned n);`
- Eliberarea zonei alocate cu funcția `malloc()` se face folosind funcția standard `free()` cu următorul prototip:

```
void free(void *p); // p este pointerul obținut la apelul funcției
malloc()
```

- O altă funcție standard utilizată pentru a aloca dinamic zona de memorie este funcția `calloc()` care și inițializează zona cu 0, cu prototipul:

```
void * calloc(unsigned nrelem, unsigned dimelem);
```

  Eliberarea acestei zone se face folosind tot funcția `free()`.

- Realocarea memoriei se poate face cu funcția `realloc()` declarată în `stdlib.h` având următorul prototip:

```
void *realloc(void *block, size_t size);
```

În limbajul C++ alocarea dinamică a memoriei mai poate fi efectuată și cu operatorii ***new*** și ***delete***.

- Operatorul `new`, permite alocarea în zona heap a memoriei. Acesta este un operator unar și are prioritate ca și ceilalți operatori unari. Rezultatul aplicării operatorului `new` este adresa de început a zonei de memorie alocată în memoria heap. Operatorul `new` nu returnează NULL (0), semnalizează în schimb o excepție `std::bad_alloc` dacă nu este posibilă realizarea alocării. Se poate dezactiva mecanismul de excepții prin:

```
tip * p = new (std::nothrow) tip;   //fără excepții active
//unde p este inițializat fie ca un pointer către o zona de memorie, fie 0 (în
cazul în care nu se poate aloca dimensiunea necesară, și astfel se poate verifica
ca la funcțiile de alocare din C valoarea returnată)
```

- Operatorul `delete` este folosit pentru eliberarea zonei de memorie alocată pe heap cu operatorul `new`. Dacă p este un pointer de un anumit tip și avem:

```
tip *p;
p = new (nothrow)tip;
```

atunci zona de memorie alocată în heap se eliberează cu:

```
delete p;
```

- Alocarea tablourilor cu operatorul `new` se face specificând între paranteze pătrate acest lucru astfel:

```
int *pi = new (nothrow)int[10];     //tablou unidimensional alocat
dinamic (initializat implicit cu 0 ?, functie de compilator)
int *pi = new (nothrow)int[10]{10}; //tablou unidimensional alocat
si initializat explicit cu 10 (C++0x/1y/2z)
```

Eliberarea se face cu:

```
delete [] pi ;
```

- Inițializarea zonei de memorie (diferă formal cu mici diferențe în funcție de mediile integrate folosite):

```
void * memset(void *dst, int c, size_t n);
```

- setează primii n octeți începând cu adresa dst la valoarea dată de conținutul din parametrul c și returnează adresa șirului pe care lucrează;

- Copierea zonelor de memorie:

```
void * memcpy(void *dest, const void *src, size_t n);
```

- copiază un bloc de memorie de lungime n de la adresa src la adresa dest., nu se garantează o funcționare corectă în cazul în care blocul sursă și cel destinație se suprapun, funcția returnează adresa blocului destinație;

```
void * _memccpy(void *dest, const void *src, int c, size_t n);
```

- copiază n octeți de la sursă (adresa src) la destinație (adresa dest), însă copierea se poate opri în următoarele situații:
  - la întâlnirea valorii c (care este copiată la destinație);
  - s-au copiat deja n octeți la adresa de destinație.

- Mutarea zonelor de memorie:

```
void * memmove(void *dest, const void *src, size_t n);
```

- mută un bloc de lungime n de la adresa sursă `src` la adresa destinație `dest`. Se garantează copierea corectă, chiar dacă cele două zone de memorie se suprapun; funcția returnează adresa zonei destinație;

- Compararea zonelor de memorie:

```
int memcmp(const void *s1, const void *s2, unsigned n);
int _memicmp(const void *s1, const void *s2, unsigned n);//valid doar
in C++
```

- funcția `memcmp()` compară primii n octeți ai zonelor de memorie s1 și s2 și returnează un întreg mai mic decît, egal cu, sau mai mare decât zero, dacă s1 este mai mic decat, coincide, respectiv este mai mare decât s2.
- funcția `_memicmp()` face același lucru, dar fără a ține cont de litere mari respectiv litere mici (ignoring case).

- Căutarea într-o zonă de memorie:

  ```
  void * memchr(const void *s, int c, unsigned n);
  ```

- funcția `memchr()` caută caracterul c în primii n octeți de memorie indicați de s; căutarea se oprește la primul octet care are valoarea c (interpretată ca unsigned char); funcția returnează un pointer la octetul găsit sau `NULL` dacă valoarea nu există în zona de memorie.

- Copierea cu inversarea octeților adiacenți

  ```
  void _swab(char *src, char *dest, int n);
  ```

- funcția `swab()` copiază n octeți de la adresa sursă `src` la o adresă destinație `dest`, cu interschimbarea octeților adiacenți. Este utilizată pentru a copia date pe o altă mașină, la care ordinea octeților este diferită. Argumentul n trebuie să fie un număr întreg pozitiv și par. Pentru n impar se vor copia doar n-1 octeți.

8.4. Theoretical brief

The functions most used in dynamic memory allocation process in C language are:

`malloc()`

`calloc()`

`free()`

- `malloc()` has the prototype `void * malloc( unsigned n);`
- freeing a `malloc()` allocated memory is performed using `free()` with the following prototype:

  ```
  void free(void *p); // p is the pointer obtained with malloc()
  ```

- Another standard function used to dynamically allocate the memory is `calloc()` which also initializes the are with 0, and has the prototype:

  ```
  void * calloc(unsigned nrelem, unsigned dimelem);
  ```

  Freeing this memory area is also performed using `free()`.

- Reallocating the memory can be done with the function `realloc()` defined in `stdlib.h` and having the prototype:

  ```
  void *realloc(void *block, size_t size);
  ```

In C++ the dynamic memory allocation can be performed with the ***new*** and ***delete*** operators.

- The `new` operator enables the memory allocation in the heap area. This is a unary operator and has the same priority as the other unary operators. `new` returns the address for the area within the heap region. `new` does not return NULL (0), but throws a `std::bad_alloc`

exception if the allocation cannot be performed. The exception mechanism can be deactivated through:

```
tip * p = new (std::nothrow) tip;    //no active exceptions
```

```
//p is initialized either as a pointer to a memory location or 0 (when the memory is not available and thus can be verified as with the allocation functions in C the return value)
```

- The `delete` operator is used to free the memory allocated on the heap with the `new` operator. If p is a pointer and we have:

```
tip *p;
```

```
p = new (nothrow)tip;
```

then the allocated memory on the heap can be freed with:

```
delete p;
```

- Array allocation with the `new` operator can be performed by using the square brackets:

```
int *pi = new (nothrow)int[10];      //dynamically allocated array (implicitly initialized with 0, depending on the compiler)
```

```
int *pi = new (nothrow)int[10]{10}; //dynamically allocated array and initialized with 10 (C++0x/1y/2z)
```

Freeing the memory is done by:

```
delete [] pi ;
```

- Memory initialisation (slight difference can be found across the different compilers):

```
void * memset(void *dst, int c, size_t n);
```

  - sets the first n bytes starting with the dst address to the value given in the c parameter and returns the address of the start of the sequence;

- Copying memory areas:

```
void * memcpy(void *dest, const void *src, size_t n);
```

- copies a memory block of length n from the address src to the address dst. There is no guarantee that the block is correctly copied if the dst and src areas overlap. It returns the address of the dst block;

```
void * _memccpy(void *dest, const void *src, int c, size_t n);
```

- copied n bytes from the src to dst, but it stops when: the value c is found (also copied in the dst); n bytes have already been copied:

- Memory moving:

```
void * memmove(void *dest, const void *src, size_t n);
```

- moves a block of length n from the src to the dest address. The correct copying of the data is guaranteed even if the two memory areas overlap. The function returns the address of the dest area;

- Comparing memory areas:

```
int memcmp(const void *s1, const void *s2, unsigned n);
int _memicmp(const void *s1, const void *s2, unsigned n);//valid doar in C++
```

- `memcmp()` compares the first n bytes of the s1 area and of the s2 area and returns an integer value which is smaller, equal or greater than 0 if s1 is smaller, coincides, or is greeted than s2..
- `_memicmp()` does the same thing, but ignores the case of the characters.

- Searching in a memory area:

```
void * memchr(const void *s, int c, unsigned n);
```

- `memchr()` searches for the c character in the first n bytes of the area indicated by s. The search stops at the first byte which has the value c (interpreted as unsigned char). The function returns a pointer to the found byte or NULL if the value does not exist in the memory area.

- Copying with neighboring byte swapping:

```
void _swab(char *src, char *dest, int n);
```

- `swab()` copies n bytes from the src address to the dest address by interchanging adjacent bytes. n has to be a positive even integer. For odd n only n-1 bytes will be copied.

8.5. Exemple/Examples

8.5.1. Ex. 1

```
/* Memory allocation for a one-dimensional array using library functions */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <malloc.h>

int main() {
    int i, n, * tab;
    printf("\nInsert the array dimension (int, >0): ");
    scanf("%d", &n);
    if ((tab = (int*)malloc(n * sizeof(int)))) {
        printf("\nInsert the array elements(int): \n");
        for (i = 0; i < n; i++) {
            printf("\t element %d: ", i);
            scanf("%d", tab + i);
            //scanf("%d", &tab[i];
        }
       //... processing data
        printf("\n\n The content of the array: ");
        for (i = 0; i < n; i++)
            printf(" \ntab[%d] = %d", i, *(tab + i));
            //printf(" \ntab[%d] = %d", i, tab[i]);
        }
    else
        printf("\nCannot allocate memory!");
    if (tab)
        free(tab);
}//main
```

8.5.2. Ex. 2

```
/* Memory allocation for a one dimensional array using C++ operators.*/

#include <iostream>
using namespace std;

int main() {
  int i, n, *tab;
  cout << "\n Insert the array dimension (int, >0): ";
  cin >> n;
  tab = new (nothrow) int[n];
  if (tab != 0) {
    cout << "\n Insert the array elements : ";
    for (i = 0; i < n; i++) {
      cout << "\n\t element " << i << " : ";
      cin >> tab[i];
      //cin >>*(tab+i);
    }
       //... processing data
    cout << "\n\n The elements of the array are: ";
    for (i = 0; i < n; i++)
      cout << tab[i] << ' ';
    //cout << *(tab+i) << ' ';
  }
  else
    cout << "Cannot allocate memory";
  if (tab)
    delete[] tab;
}//main
```

8.5.3. Ex. 3

```
/* Sequential memory allocation for a one dimensional array treated as a
bidimensional array.*/

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

void cit_mat(int*, int, int);
void afis_mat(int*, int, int);

int main(){
    int *tab, m, n;
    printf("\n Insert m and n, the number of rows and columns(int, >0): ");
    scanf("%d%d", &m, &n);
    //cin >> m >> n;
    if ((m <= 0) || (n <= 0)) {
        printf("\nInvalid numbers!");
        return 1;
    } // end if

    // Dynamic memory allocation and pointer testing
    tab = (int*)malloc(m * n * sizeof(int));
```

```
    //tab = new (nothrow) int[m * n];
    if (tab == 0) {
        //cout<<"\n Allocation error!";
        printf("\n Allocation error!");
        return 1;
    }
    printf("\nInsert the elements of the matrix %dx%d :", m, n);
    cit_mat(tab, m, n);
       //... some processing
    printf("\nThe contents of the matrix is:");
    afis_mat(tab, m, n);

    // deallocation
    if (tab)
        free(tab);
    //if (tab) delete [ ] tab;
} //main

void cit_mat(int* tb, int l, int c)
{
    int i, j;
    for (i = 0; i < l; i++) {
        //cout<<"\nLine "<< i <<": ";
        printf("\nLine %d: ", i);
        for (j = 0; j < c; j++)
            //cin >> *(tb + i * c + j);
            scanf("%d", tb + i * c + j);
    }
} //cit_mat

void afis_mat(int* tb, int l, int c)
{
    int i, j;
    for (i = 0; i < l; i++) {
        printf("\n\t");
        for (j = 0; j < c; j++)
       //cout << *(tb + i * c + j)<< ' ';
            printf("%d ", *(tb + i * c + j));
    } // end for
    printf("\n");
} //afis_mat
```

8.5.4. Ex. 4

```
/* Allocating a bidimensional array (matrix) treated line by line using double
pointers; reading and printing the elements. Integer array. */

#include <iostream>
using namespace std;

int main(){
    int i, j, m, n, **tab;
    cout << "\n Insert the number of lines (int, >0): ";
    cin >> m;
    if ((tab = new (nothrow) int*[m])) {
        cout << "\n Insert the number of columns (int, >0): ";
```

```
        cin >> n;
        for (i = 0; i < m; i++) {
            tab[i] = new (nothrow) int[n];
            if (tab[i] == 0) {
                cout << "\n Allocation error !";
                return 1;
            }
        }
        cout << "\n Insert the array elements:";
        for (i = 0; i < m; i++) {
            cout << "\n\t Linia " << i << ":\n";
            for (j = 0; j < n; j++) {
                cout << "\t Element tab[" << i << "][" << j << "]: ";
                cin >> tab[i][j];
            }
        }
       //.... processing data
           cout << "\n\n The elements of the array are: \n";
        for (i = 0; i < m; i++) {
            for (j = 0; j < n; j++)
                cout << '\t' << tab[i][j];
            cout << '\n';
        }
    }
    else
        cout << "Cannot allocate the array pointer!";
    if (tab) {
        for (i = 0; i < m; i++)
            delete[] tab[i]; // each line is deallocated
        delete[] tab; //deallocating the array pointer
    }
} //main
```

8.5.5. Ex. 5

```
/* Allocating a bidimensional array (matrix) treated line by line using double
pointers; reading and printing the elements. Characters array */

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include <string.h>

const int dim = 100;

int main() {
    int i, m, c_man = 0, l_man;
    char ** tab;
    char * man;// maneuver
    man = new (nothrow) char[dim];
    if(man==0) {
        cout << "\nCannot allocate! Exit!";
        return 1;
    }
    cout << "\n Enter the no. of strings (int, >0): ";
    cin >> m;
```

```
    tab = new (nothrow) char* [m]; //array of pointers to rows
    if (tab == 0) {
        cout << "\nCannot allocate! Exit!";
        return 1;
    }
    for (i = 0; i < m; i++) {
        cout << "\n Enter string " << i << ": ";
        cin >> man;
        l_man = (int)strlen(man);
        c_man += l_man;
               tab[i] = new (nothrow) char[l_man + 1]; //each row pointer allocated
       as man chars+1 for \0
        if (tab[i] == 0) {
            cout << "\n Allocation error!Exit! " << endl;
            return 1;
        }
        strcpy(tab[i], man);
    }
    delete[] man; //remove initial man char array
    man = new (nothrow) char[c_man + 1] {0};  //assign with 0 char array from C++0x
    if (man == 0) {
        cout << "\nCannot allocate! Exit!";
        return 1;
    }
   // memset(man, 0, c_man);//set memory of char array with 0
    cout << "\nStrings are: " << endl;
    for (i = 0; i < m; i++) {
        cout << tab[i] << "\n";
        strcat(man, tab[i]);
    }
    cout << "\nThe concatenated strings are: " << man << endl;
    if (tab) {
        for (i = 0; i < m; i++)
            delete[] tab[i]; //for each row remove the space of char values
       delete[] tab; //remove the array of pointers to rows - no. of strings
    }
    delete[] man; //remove man char array
    return 0;
} //main
```

8.5.6. Ex. 6

```
/* Optional - a function which returns the address of a dynamically allocated array
and throws a bad_alloc exception (exceptions will be introduced in OOP C++). */

#include <iostream>
using namespace std;

int* foo(int);

int main() {
    int dim;
    cout << "\nEnter the dimension of the array (int, >0): ";
    cin >> dim;
    int* p = foo(dim);
    cout << "\nThe values of the array are: ";
```

```
    for (int i = 0; i < dim; i++)
        cout << p[i]<<' ';
    delete[] p;
}//main

int* foo(int dim)
{
    try {
        //dynamic arrays or static arrays may be returned as address
        int* arr = new int[dim];
        for (int i = 0; i < dim; i++)
            *(arr + i) = i;
        return arr;
    }
    catch (const bad_alloc& exc) {
        cout << "\n Allocation problem!!!";
        exit(-10);
    }
}//foo
```

8.5.7. Ex. 7

```
/* The programme copies in memory the content of the string from the "alphabet"
address until it encounters the 'K' character. */

#include <stdio.h>
#include <string.h>

int main(){
    const char* alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    char target[27];
    char* result;
    printf("The content of the destination string: ");
    result = (char*)_memccpy(target, alphabet, 'K', strlen(alphabet));
    if (result)
        *result = NULL;
    printf(target);
    printf("\n");
} //main
```

8.5.8. Ex. 8

```
/* The programme compares the memory locations where the "AAAAA", "BBBBB" and
"aaaaaA" strings were stored and shows the difference between the memcmp() and
_memicmp()/memicmp() (only available in C++) functions. */


#include <stdio.h>
#include <string.h>

int main(){
    const char* a = "AAAAA";
    const char* b = "BBBBB";
    const char* c = "aaaaaA";

    printf("The result of comparing %s with %s using memcmp( ) is: %d\n", a, b,
memcmp(a, b, strlen(a)));
```

```
    printf("The result of comparing %s with %s using _memicmp( ) is: %d\n", a, c,
_memicmp(a, c, strlen(a))); //only in C++
    //printf("The result of comparing %s with %s using _memicmp( ) is: %d\n", a, c,
memicmp(a, c, strlen(a)));//only in C++ gcc, ...
} //main
```

8.5.9. Ex. 9

```
/* The programme exemplifies the memmove() function. */

#include <stdio.h>
#include <memory.h>
const int dim = 5;

int main( ) {
  float values[ ] = { 1.1f, 2.2f, 3.3f, 4.4f, 5.5f, 6.6f };
  float empty[dim];
  int i;
  printf("The content of the destination array: ");
  memmove(empty, values, sizeof(empty));
  for (i = 0; i < dim; i++)
    printf("%3.1f\t", empty[i]);
  printf("\n");
}//main
```

8.5.10. Ex. 10

```
/* The programme introduces the _swab() function. */

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

int main() {
    char source[] = "Sir de initializare12345";
    char target[64];
    memset(target, 0, sizeof(target));
    _swab(source, target, (int)strlen(source));
    //swab(source, target, strlen(source)); // Use this if _swab is not available
    printf("Source: %s\n", source);
    printf("Destination: %s\n", target);

    return 0;
}//main
```

8.6. Întrebări

1. Ce înseamnă memoria „heap”?
2. Prin ce diferă funcția `malloc()` de funcția `calloc()`?
3. Cum se poate face alocarea dinamică în limbajul C ? Dar în limbajul C++ ?
4. Faceți o comparație între funcțiile de alocare dinamică a memoriei în C și mecanismul din C++.

8.7. Questions

1. What does the heap memory mean?

2. What is the difference between the `malloc()` and `calloc()` functions?
3. How can dynamic memory allocation be performed in the C programming language? How about in C++?
4. Do a comparison between the dynamic memory allocation functions in C and C++ mechanism.

8.8. Lucru individual

1. Să se scrie un program care citește n (validați n prin reintroducerea valorii) numere reale, pe care le stochează într-un tablou alocat dinamic, afișează suma elementelor negative din tablou folosind o funcție, iar la sfârșit eliberează zona de memorie alocată.
2. Fie o aplicație de gestiune distribuită care consideră că tratează activitatea din m (>1,<=5) orașe diferite, în fiecare oraș fiind n (>1,<=3) birouri de vânzare pe teritoriul respectiv. Să se creeze în cadrul unei funcții un tablou de m pointeri către date de tip float, fiecare pointer din acest tablou referind o zonă în heap alocată dinamic de n date de tip float, ce reprezintă situația vânzărilor la sfârșitul unei zile. Se cere prelucrarea datelor din fiecare oraș, respectiv din fiecare birou de vânzare, prelucrare ce va include:
   - funcție care permite introducerea datelor pentru cele m orașe și respectiv pentru cele n magazine din fiecare oraș;
   - funcție ce permite determinarea totalului de vânzări pe fiecare oraș în parte, valoare pe care o va returna, iar în programul principal se va calcula și afișa media vânzărilor din toate cele m orașe;
   - funcție care va permite eliberarea spațiului de memorie alocat dinamic astfel încât dacă aceeași firmă are alte n magazine în cele m orașe de profil diferit să poată realoca un spațiu echivalent pentru noile prelucrări pe care le va efectua.
3. Să se scrie o aplicație C/C++, care alocă dinamic memorie pentru stocarea elementelor a două matrice de "m" linii și "n" coloane. Să se scrie o funcție care calculează suma celor două matrice și o funcție pentru afișarea unei matrice. Să se afișeze matricile inițiale și matricea obținută după adunare.
4. Să se scrie o aplicație C/C++ care alocă dinamic memorie pentru "n" șiruri de caractere, care se vor citi de la tastatură.
5. Declarați un pointer global de tip float. În funcția main() citiți o valoare întreagă m care reprezintă dimensiunea tabloului de numere reale. Alocați memoria necesară pentru stocarea tabloului și citiți elementele de la tastatură. Determinați valoarea medie a celor m elemente și calculați Mn=(sum(pow((xi-xmed),n))/m, unde n=1,2,3. Afișați rezultatele și apoi eliberați memoria. Folosiți funcțiile malloc() și free(). Generați numerele din tablou folosind funcția de bibliotecă care generează numere aleatoare și determinați pentru acestea media valorilor și Mn. Realizați aceeași aplicație folosind operatorii new și delete.
6. Folosiți alocarea dinamică pentru o matrice m x n cu valori întregi (m, n <7). Inițializați elementele matricei. Dacă matricea este pătratică, folosiți metoda lui Sarrus pentru a obține valoarea determinantului. Afișați rezultatul și eliberați memoria.
7. Să se scrie o aplicație C/C++ care alocă dinamic memoria necesară pentru stocarea a n<=10.000 numere întregi. Programul inițializează numerele cu valori aleatoare între 1 și Max<=100 (folosiți funcțiile srand() și/sau rand() în VC++). Scrieți o funcție care afișează cele mai frecvente k<=10 numere și numărul lor de apariții în tabloul inițial.
8. Să se scrie o aplicație C/C++ în care se alocă dinamic memorie pentru n numere întregi (validați n prin reintroducerea valorii), numere ce vor fi citite de la tastatură. Să se scrie funcția care extrage radicalul din fiecare număr întreg valid (>0) și stochează valorile obținute într-un alt tablou alocat dinamic. Să se afișeze numerele inițiale și valorile din tabloul nou format. Eliberați la sfârșit memoria alocată.
9. Scrieți un program în care se citesc m șiruri de caractere ce pot conține și spațiu, care se concatenează într-un alt șir, alocat dinamic. Afișați șirurile inițiale introduse (mai puțin de

256 caractere), cel concatenat (șirurile vor fi separate prin spațiu), cuvintele independente (separate prin spațiu folosind strtok( )), și numărul lor. Eliberați memoria alocată dinamic. Repetați procesul atât cât dorește utilizatorul. Generalizați considerând ca delimitatori caracterele din șirul: char delimiters[] = " .,;:!?-"; în loc de spațiu.

10. Să se scrie o aplicație C/C++, care alocă dinamic memorie pentru stocarea elementelor unei matrice pătrate de dimensiune nxn. Să se scrie o funcție care calculează suma numerelor pozitive pare de sub diagonala principală și o funcție pentru afișarea matricei. Să se afișeze matricea și suma cerută. Eliberați memoria alocată dinamic.
11. Generați un tablou de pointeri către șiruri constante folosind funcția strdup( ) sau o altă funcție specifică. Afișați valorile pare din tablou.
12. Scrieți o aplicație C/C ++ care citește de la tastatură un tablou de până la 10 numere întregi valori în intervalul 2, ..., 10 (vezi asemănător problema cu tablouri fără alocare dinamică la laboratorul despre tablouri). Dimensiunea reală este introdusă de la tastatură, tabloul fiind alocat dinamic. Definiți două funcții care consideră tabloul ca parametru și returnează valorile minime și maxime din tablou. Definiți o altă funcție care returnează valoarea medie a elementelor introduse în tablou.
    Definiți o altă funcție, olympic_filter(), care consideră tabloul inițial, returnând adresa unui tablou alocat dinamic, fără prima apariție a valorii minime și maxime, cu valorile ordonate crescător.
    În main() se afișează, de asemenea, tabloul inițial, valoarea minimă și maximă, media elementelor tabloului inițial, elementele tabloului nou obținut în funcția olympic_filter() și media obținută după aplicarea filtrului olimpic.
    Validați introducerea datelor în conformitate cu cerințele problemei (dimensiunea tabloului, elementele introduse cu reluare). Operațiile cu elementele tablourilor se vor realiza folosind pointeri.

8.9. Individual work

1. Write a program that reads from the keyboard n (validate n by re-entering the value) real numbers, stores them into a dynamically allocated array, displays the sum of the negative elements determined by a function and frees the allocated memory.
2. Consider an application for managing a company's distributed activity m (>1,<=5) different cities with n(>1,<=3) selling points in each one of them. Define a function that declares an array of float pointers, each pointer being used for referencing a heap zone of memory that stores n float values that represent the total sales at the end of day. The program needs to process the data using functions for:
    - reading the data for all the selling points from all the cities;
    - calculating and returning the total amount of sales in every city. The values will be centralized in the main function and the average value will be displayed.
    - freeing all the used memory
3. Write a program that allocates the necessary amount of memory for storing the elements from two [m x n] integer matrices. Write a function that calculates the sum of the initial matrices and another one for displaying both the original values and the result.
4. Write a program that reads n characters arrays from the keyboard and stores them into n dynamically allocated memory zones.
5. Declare a global float pointer. In function main(), read an integer value m that represents the dimension of an array of float numbers. Allocate the necessary amount of memory for storing the array and then read its elements from the keyboard. Determine the average value of those m elements and calculate Mn=(sum(pow((xi-xmed),n))/m, where n=1,2,3. Display the results and then free the allocated memory. Use the functions malloc() and

free(). In another function, generate the numbers in the array using a library function that generates random numbers and determine the same values. Use new and delete operators.

6. Use the dynamic allocation for a matrix of m x n integer values (m, n <7). Initialize the elements of the matrix. If n is equal to m, use the Sarrus method to obtain the value of the "determinant". Display the result and free the allocated memory.
7. Write a C/C++ application that allocates the necessary amount of memory for storing n<=10.000 integer numbers. The program automatically initializes the numbers with random values between 1 and Max<=100 (use the library functions srand() and/or rand() in VC++). Write a function that displays the most k<=10 frequent numbers and the number of their appearances in the initial array.
8. Write a C/C++ application that allocates memory for n integer numbers (validate n by re-entering the value) that will be read from the keyboard. Write a function that determines the square root of each number (>0) and stores the result into another dynamically allocated array. Display the initial and computed values. Free the allocated memory.
9. Write a program that reads strings that can contain space that are concatenated into another, dynamically allocated string. Display the initial strings entered (less than 256 characters each), the concatenated one (the strings will be separated by a space), the independent words (separated by a space using strtok()), and their number. Free up dynamically allocated memory. Repeat the process as many times as the user wants. Generalize considering as delimitators the string: char delimiters[] = " .,;:!?-"; instead of space.
10. Write a C/C++ application that allocates the necessary amount of memory for storing a NxN integer matrix. Write a function that calculates the sum of the positive numbers located below the main diagonal and another function that displays the matrix. Print the matrix and the calculated sum. Free the allocated memory.
11. Generate an array of pointers to constant strings using the strdup() method or a specific method. Display the even entries of the array.
12. Write a C/C ++ application that reads from the keyboard an array of up to 10 integers in range 2, ..., 10 (see the similar problem from arrays without dynamic allocation from array laboratory). The actual size is entered from the keyboard, the array being allocated dynamically. Define two functions that consider the array as a parameter and return the minimum and maximum values in the array. Define another function that returns the average value of the elements of the entered array.
    Define another function, olympic_filter(...), which considers the initial array, returning the address of a dynamically allocated array without the first occurrence of the minimum and maximum value ordered in increasing mode.
    In the main( ) function also display the initial array, the minimum and maximum value, the average of the elements of the initial array, the elements of the newly obtained array returned by the function olympic_filter() and the average obtained after the olympic filter. Validate the data entry according to the requirements of the problem (array size, elements entered with resumption). The operations with the elements of the arrays will be done using pointers.

9. Structuri. Structuri imbricate. Pointeri la structuri. Alte tipuri utilizator

Data structures. Nested structures. Pointers to structures. Other user data-types

9.1. Obiective

- Înțelegerea noțiunii de date definite de utilizator (user-datatypes)
- Înțelegerea modalității de lucru cu structuri: declarare, inițializare, acces la câmpuri
- Scrierea de programe folosind structuri simple și imbricate
- Exersarea accesului cu pointeri la structuri de date
- Declararea și utilizarea datelor utilizator diferite de cele de tip struct.
- Scrierea de programe folosind alte date utilizator.

9.2. Objectives

- Understanding the user-datatypes notion.
- Understanding how structures work: declaration, initialization, field access
- Writing simple programs using simple and nested structures
- Accessing the data structures with pointers
- Declaring and using data types other than struct.
- Writing programs using other data types.

9.3. Breviar teoretic

Nevoia pentru structuri era una dintre motivațiile principale pentru crearea limbajului C.

O structură reprezintă o mulțime ordonată de elemente grupate în vederea utilizării lor în comun. Ea se declară folosind cuvîntul cheie ***struct*** astfel:

```
struct [Nume_struct] {
lista_declarații;
}[nume1,nume2,…,numen];
```

unde:

- `Nume_struct`, este un nou tip de date, tip definit de utilizator, cu construcția `struct`;
- `lista_declarații`, este o listă prin care se declară componentele unei structuri, putând conține elemente de forma tip_i $element_i$, unde tip_i este un tip admis de limbajul C inclusiv o nouă structură (în acest ultim caz avem structuri imbricate);
- $nume_1,\ldots,nume_n$ este o listă de variabile/obiecte de tip `Nume_struct`, putând lipsi la definirea structurii, caz în care este obligatoriu să fie precizat `Nume_struct`.

Dacă avem `Nume_struct` și nu avem $nume_i$ atunci putem defini ulterior alte variabile/obiecte de tip `Nume_struct` cu declarația:

```
struct Nume_struct nume_obj;    //C/C++
```

sau

```
Nume_struct nume_obj;    //C++
```

unde `Nume_struct` reprezintă un nou tip de dată, un tip utilizator.

Elementele unei structuri se numesc câmpuri sau atribute, iar grupările de elemente pot fi numite variabile de tip structură, înregistrări sau obiecte.

Printre elementele unei structuri se pot găsi și alte structuri (inclusiv pointeri către structura însăși), care sunt referite din exterior spre interior.

Accesul la elementele unei structuri se face precizând numele variabilei/obiectului de tip structură, de exemplu `angajat`, și câmpul care ne interesează (inclusiv dacă e o structură imbricată), separate prin operatorul . (punct).

```
angajat.data_nast.zi = 9; //zi este camp al variabilei struct. data_nast,
care este câmp al variabilei-structura angajat
```

Acest mod de acces se numeste acces prin ***nume calificat***.

Accesul la elementele unei structuri se poate face și printr-un pointer la structură, folosind operatorul -> (săgeată).

```
p = &angajat;
```

(p->data_nast).an = 1995; //acces prin pointer la campul data_nast, și `sub-campul an`

Un tablou de structuri se declară considerând `struct Nume_struct` ca fiind tipul tabloului.

Structurile pot fi transferate ca parametri funcțiilor. În general, structurile nu se transferă prin valoare, ci prin pointeri spre structură. O funcție poate să returneze o structură.

9.3.1. Pointeri la structuri

Un pointer către o structură se declară la fel ca orice pointer către orice alt tip de variabilă. Pointerul va conține adresa primei componente a structurii la care pointează.

Pointerii la structuri sunt utili când dorim să transmitem o structură unei funcții, prin intermediul lor putând să transmitem doar adresa structurii.

În corpul funcției accesul la câmpurile structurii se face prin intermediul pointerului p, astfel:

```
(*p).zi          sau   p ->zi
(*p).luna        sau   p ->luna
(*p).an          sau   p ->an
```

9.3.2. Câmpuri de biți (Bit fields)

Un câmp de biți este un câmp al unei structuri care cuprinde unul sau mai mulți biți adiacenți. Cu ajutorul câmpurilor de biți se pot accesa prin nume, unul sau mai mulți biți dintr-un octet sau cuvânt oferind o granularitate perfect adaptabilă la periferice inteligente (ca ceasul CMOS, controlere de comunicație, etc.).

O structură care conține câmpuri de biți se declară astfel:

```
struct [Nume_struct] {
tip [ câmp1] : lungime1;
  ...
tip [câmpn] : lungimen;
} [nume1, nume2,..., numem];
```

unde `tip` este un cuvânt cheie, de obicei `int/unsigned (char, int)`, care specifică modul în care este interpretată valoarea acelui cîmp, iar `lungime` reprezintă numărul de biți pentru câmpul respectiv și este o constantă cu valori între 0 și o valoare nu mai mare decât numărul de biți alocat tipului respectiv (8 biți pentru `char`, 16 pentru `short int` și 32 pentru `int/long` pentru mediul Microsoft).

Exemplu:

```
struct oct_cda_82C54 {
      unsigned char BCD : 1;//LSB
      unsigned char ModDeLucru : 3;
      unsigned char RW_LowHigh: 2;
      unsigned char SlctChannel : 2;
} oct_cda_timer1, oct_cda_timer2;
```

Reguli:

Câmpurile se alocă de la primul câmp către ultimul, începând cu biții de ordin inferior ai cuvântului (LSB). Un câmp de biți trebuie să se poată aloca într-un cuvânt calculator.

Dacă un câmp nu se poate aloca în cuvântul curent, atunci compilatorul îl va aloca în cuvântul următor.

Mai multe câmpuri de biți, dacă încap, vor fi păstrate în același cuvânt calculator.

Dacă unul din câmpuri este `int` sau `unsigned int` câmpul de biți va fi alocat pe minim un cuvânt, adică patru octeți (bytes).

Un câmp fără nume nu poate fi accesat. Acesta definește o zonă neutilizată dintr-un cuvânt.

Lungimea în biți alocată pentru un câmp de biți o putem defini zero, dacă vrem ca următorul câmp de biți să fie alocat în următorul cuvânt-calculator.

Compilatorul admite lungime 0 numai pentru câmpuri de biți fără nume. Un câmp fără nume nu poate fi accesat.

Câmpurile de biți se pot referi folosind aceleași convenții ca și în cazul structurilor obișnuite: direct (prin operatorul .) sau indirect (prin operatorul `->`).

Observații:

O structură poate îngloba câmpuri de biți împreună cu alte variabile de alte tipuri. Octetul fiind cea mai mică unitate de memorie adresabilă la calculatoarele compatibile IBM PC, adresa de memorie a unui câmp de biți nu se poate obține cu ajutorul operatorului de adresare &, deci acesta nu se poate aplica la un câmp de biți.

Câmpurile de biți sunt folosite când lucrăm cu multe date booleene, permițând folosirea eficientă a memoriei, 8 date booleene pe octet. Prelucrarea datelor pe biți se poate face și cu ajutorul operatorilor logici pe biți, care duc însă în general la un efort mai mare în programare comparativ cu câmpurile de biți (necesită instrucțiuni suplimentare cum ar fi deplasări, setări, și/sau mascări de biți).

9.3.3. Reuniuni (Unions)

Reuniunea este un tip de dată definit de utilizator, asemănător din punct de vedere al construcției cu structura, dar în loc de cuvântul cheie `struct` este folosit cuvântul cheie `union` și la definirea unei variabile de acest tip nu se alocă memorie pentru toate câmpurile conținute de reuniune, ci doar memoria necesară câmpului cu dimensiunea cea mai mare. Putem spune că reuniunea este ca o structură în care toate câmpurile sunt stocate la aceeași adresă.

Exemplu:

```
union A {
      int x;
      long y;
      double r;
      char c;
} u;
```

unde A este un nou tip de dată de tip reuniune, iar u este o variabilă/obiect de tip reuniune A.

Accesul la elementele x, y, r, c ale reuniunii se face ca și la structuri folosind operatorul . *(punct)* sau operatorul -> *(săgeată)*.

9.3.4. Asignări de nume pentru tipuri de date

În limbajul C se poate atribui un nou nume unui tip deja existent, fie el predefinit în limbaj sau definit de utilizator, cu o construcție de forma:

```
typedef tip_initial nou_nume_tip;
```

unde:

- `tip_initial`, este tipul existent;
- `nou_nume_tip`, este noul nume, care poate fi utilizat în continuare pentru a declara date de același tip.

9.3.5. Enumerări (Enumeration, enum)

Tipul enumerare permite programatorului să definească o listă de constante întregi cu nume, în vederea folosirii de nume sugestive pentru valori numerice.

Tipul enumerare se declară printr-un format asemănător cu cel utilizat în cadrul structurilor și anume:

```
enum [Nume_enum] {
nume_i [=const_i],
 ...
```

}[en_1, en_2, ..., en_n];

unde:

- Nume_enum, este numele noului tip de date utilizator introdus prin această declarație
- $nume_i$, sunt nume care se vor utiliza în locul valorilor numerice și au valori implicite: prima are valoarea 0, iar restul au valoarea constantei precedente plus 1 ($nume_i$ va avea valoarea i);
- $const_i$ este constantă de inițializare a unui element; dacă este prezentă atunci:

  $nume_i$ = $const_i$, $nume_{i+1}$ = $const_i$+1,...

de exemplu:

```
enum Months {ian = 1, feb, mar, ......, dec}; // feb = 2
```

- en_1, en_2, ..., en_n, sunt variabile enum de noul tip nume_enum.

În limbajul C, unei variabile enumerare i se poate atribui o valoare întreagă și se pot face operații cu aceste variabile. În limbajul C++, unei variabile enumerare i se pot atribui numai constante de enumerare.

Tipul enumerare se folosește pentru variabile care pot lua un număr redus de valori întregi prin asocierea unor nume sugestive fiecărei valori.

9.4. Theoretical brief

The need for structures was one of the main motivations for creating the C programming language. A structure represents an ordered set of elements grouped for common use. It is declared using the struct construct as follows:

```
struct [Struct_Name] {
    declaration_list;
}[name1, name2, ..., nameN];
```

Where:

- Struct_Name is a new user-defined data type created using the struct construct.
- declaration_list is a list that declares the components of the structure. It can contain elements of the form type element, where type is a valid C data type, including another structure (in the case of nested structures).
- $name_1$, ..., $name_N$ is a list of variables or objects of type Struct_Name. It may be omitted during structure definition, in which case Struct_Name must be specified.

In C, structures provide a way to organize and group related data elements into a single user-defined data type. This allows for better code organization and management of complex data structures.

If we have a Nume_struct (Name_struct) and we don't have a name, we can later define other variables/objects of type Nume_struct using the following declarations:

```
struct Nume_struct nume_obj;  // în C/C++
```

then:

```
Nume_struct nume_obj;   //C++
```

where Nume_struct represents a new type of data, user-defined data.

The elements of a structure are called fields or attributes, and groupings of elements, variables of the structure type, or records are also known as objects.

Among the elements of a structure, you can find other structures (including pointers to the structure itself), which are referenced from the exterior to the interior.

Access to the elements of a structure is achieved by specifying the name of the variable or object of the structure type, for example, "employee," and the field of interest (even in the case of a nested structure), separated by the dot operator, ".".

```
employee.date_of_birth.day = 9; // 'day' is a field of the 'date_of_birth'
variable, which is a field of the 'employee' structure.
```

This method of access is called qualified name access.

Accessing the elements of a structure can also be done through a structure pointer using the arrow operator (->).

```
p = &employee;
(p->date_of_birth).year = 1995; // access through a pointer to the
'date_of_birth' field, and the sub-field 'year'
```

9.4.1. Pointers to structures

A pointer to a structure is declared in the same way as any pointer to any other variable type. The pointer will contain the address of the first component of the structure it points to.
Pointers to structures are useful when we want to pass a structure to a function, allowing us to pass only the address of the structure.
Inside the function, access to the structure's fields is done through the pointer 'p' as follows:

```
(*p).day     or     p->day
(*p).month   or     p->month
(*p).year    or     p->year
```

In C, both `(*p).field` and `p->field` are equivalent ways to access the field of the structure using a pointer.

9.4.2. Bit Fields

A bit field is a field within a structure that encompasses one or more adjacent bits. Bit fields are used to access, by name, one or more bits within a byte or word, providing a highly adaptable granularity for intelligent peripherals (such as the CMOS clock, communication controllers, etc.).

A structure that contains bit fields is declared as follows:

```
struct [Struct_Name] {
    type [field1] : length1;
    ...
    type [fieldn] : lengthn;
```

} [$name_1$, $name_2$, ..., $name_m$];

where type is a keyword, usually int/unsigned (char, int), that specifies how the value of that field is interpreted, and length is the number of bits for that field and is a constant with values between 0 and no greater than the number of bits allocated to that type (8 bits for char, 16 for short int, and 32 for int/long for where type is a keyword, usually int/unsigned(char, int), that specifies how interpreted the value of that field, and length represents the number of bits for that field and is a constant with values between 0 and a value no greater than the number of bits allocated to that type (8 bits for char, 16 for short int and 32 for int/ long for the Microsoft environment).

In this context:

- [Struct_Name] represents the name of the structure.
- [$field_1$], [$field_n$] are the names of the individual bit fields.
- [$length_1$], [$length_n$] represent the number of bits each bit field occupies.
- [$name_1$, $name_2$, ..., $name_m$] are optional variable names when defining the structure.

Example:

```
struct oct_cda_82C54 {
      unsigned char BCD : 1;//LSB
      unsigned char ModDeLucru : 3;
      unsigned char RW_LowHigh: 2;
      unsigned char SlctChannel : 2;
} oct_cda_timer1, oct_cda_timer2;
```

In this example:

- unsigned char represents the data type for the bit fields.
- BCD, OperationMode, ReadWrite_LowHigh, and SelectChannel are the names of individual bit fields.
- 1, 3, and 2 indicate the number of bits each bit field occupies.
- oct_cda_timer1 and oct_cda_timer2 are optional variable names when defining the structure.

Typically, unsigned char, int, or similar data types are used for bit fields, and the number of bits for each field should be a constant with values between 0 and the maximum size of the computer's word (e.g., 8 bits for char, 16 for short int, and 32 for int/long in the Microsoft environment).

Rules:

Bit fields are allocated from the least significant bit (LSB) of the computer word, starting with the first field and progressing towards the last.

A bit field must be allocatable within a single computer word.

If a bit field cannot fit in the current word, the compiler will allocate it in the next word.

Multiple bit fields, if they can fit, will be stored in the same computer word (e.g., a list of fields with lengths 1, 3, 2, already occupies two bytes).

If one of the fields is int or unsigned int, the bit field will be allocated in at least one computer word, which is four bytes (bytes).

An unnamed bit field cannot be accessed. It defines an unused area within a word.

The length in bits allocated for a bit field can be defined as zero if you want the next bit field to be allocated in the next computer word.

The compiler only allows a length of 0 for unnamed bit fields. An unnamed bit field cannot be accessed.

Bit fields can be accessed using the same conventions as with regular structures: directly (through the . operator) or indirectly (through the -> operator).

Observations:

A structure can contain bit fields along with other variables of different types.

Since the byte is the smallest addressable memory unit on compatible IBM PC computers, the memory address of a bit field cannot be obtained using the address operator &, and it cannot be applied to a bit field.

Bit fields are used when working with many boolean data, allowing efficient memory usage with 8 boolean data items per byte. Bitwise logical operations can be used for bit-level data processing, but they often require more programming effort compared to bit fields, involving additional instructions such as shifting, setting, and/or masking of bits.

9.4.3. Unions

A union is a user-defined data type that is structurally similar to a structure. However, instead of using the keyword "struct," the keyword "union" is used in its declaration. When a variable of this type is defined, memory is allocated only for the field with the largest size, not for all the fields contained within the union. It can be said that a union is like a structure in which all fields are stored at the same address.

Example:

```
union A {
      int x;
      long y;
      double r;
      char c;
} u;
```

In this example:

- `A` is a new user-defined data type of the union type.
- `u` is a variable or object of the union type A.

Access to the elements x, y, r, and c of the union is done similarly to structures using the . (dot) operator or the -> (arrow) operator.

9.4.4. Assigning Names to Data Types

In the C language, you can assign a new name to an already existing data type, whether it's predefined in the language or user-defined, using the following construct:

```
typedef initial_type new_type_name;
```

Where:

- `initial_type` is the existing data type.
- `new_type_name` is the new name, which can be used subsequently to declare data of the same type.

9.4.5. Enumerations

The enumeration type allows a programmer to define a list of integer constants with names, providing meaningful names for numeric values. The enum type is declared using a format similar to that used for structures, like this:

```
enum [EnumName] {
numei [=consti],
 ...
}[en1, en2, ..., enn];
```

where:

- `EnumName` is the name of the new user-defined data type introduced by this declaration.
- constant names are the names that will be used in place of numeric values and have default values: the first constant has a value of 0, and the rest have values equal to the previous constant's value plus 1 (i.e., constant_name will have a value of i).
- initial_value is an optional constant to initialize an element. If present, then:

  $nume_i = const_i$, $nume_{i+1} = const_i+1$,...

  for example:

  ```
  enum Months {jan = 1, feb, mar, ......, dec}; // feb = 2
  ```

- $en_1, en_2, ..., en_n$, are `enum` variables of the new type `EnumName`.

In the C language, an enumeration variable can be assigned an integer value, and operations can be performed with these variables. In C++, an enumeration variable can only be assigned enumeration constants.

The enumeration type is used for variables that can take a limited number of integer values by associating meaningful names with each value. This provides more readable and self-explanatory code by using suggestive names for specific values.

9.5. Exemple/Examples

9.5.1. Ex. 1

```
/* Read the elements of a Data_calendar structure from the console and display the
read date in a different format: day/month/year. */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
#include <stdlib.h> // for exit(int)
```

```
const int char_dim = 11;

struct Date_calendar {
    int day;
    char month[char_dim];
    int year;
};

// Global array of pointers to constant strings, defined
const char* months[] = { "jan", "feb", "mar", "apr", "may", "jun", "jul", "aug",
"sep", "oct", "nov", "dec" };

void puts_exit(const char*); // Keeps displaying the passed error message until a
key is pressed, then exits the process

int main() {
    struct Date_calendar dc = {}; // Generalized initialization of an object
    printf("\nInitial date is:\n %d/%s/%d", dc.day, dc.month, dc.year);
    // struct Date_calendar udc; // Declaration of an object - allowed
    // printf("\nInitial date is:\n %d/%p/%d", udc.day, udc.month, udc.year); - Not
allowed
    int i, month = 0; // 'month' is in a different namespace than 'dc.month'

    puts("\nEnter the current date:");
    printf("\n\t day (1,2,...31): ");
    scanf("%d", &dc.day);
    if ((dc.day <= 0) || (dc.day > 31))
        puts_exit("\nIncorrect day!");
    printf("\n\t month (jan, feb...): ");
    scanf("%s", dc.month);
    for (i = 0; i < 12; i++) {
        if (_stricmp(dc.month, months[i]) == 0) {
            month = i + 1;
            break;
        }
    }
    if (month == 0) // If the 'month' variable hasn't changed, the typed word
wasn't identical to any of the elements in 'months[]'
        puts_exit("\nIncorrect month!");
    printf("\n\t year: ");
    scanf("%d", &dc.year);
    if (dc.year <= 0)
        puts_exit("\nIncorrect year (negative)!");
    printf("\nEntered date is:\n %d/%d/%d", dc.day, month, dc.year);
}//main

void puts_exit(const char* ptr) {
    puts(ptr); // printf(ptr); -- Also OK
    exit(1);
}//puts_exit
```

9.5.2. Ex. 2

```
/* The program locally declares a new data type, Forma, within the main() function
and defines a variable/object of this type called circle. The fields of the
structure are accessed using the "qualified name" method */
#include <stdio.h>

int main() {
    struct Shape {
        int type;
        int color;
        float radius;
        float area;
        float circumference;
    } circle = { 2, 1, 5.0f, 78.37f, 31.42f };

    printf("\nCircle Info:\n");
    printf("circle.type: %d\n", circle.type);
    printf("circle.color: %d\n", circle.color);
    printf("circle.radius: %g\n", circle.radius);
    printf("circle.area: %g\n", circle.area);
    printf("circle.circumference: %g\n", circle.circumference);

    // Later on, we define another variable-structure of the same type, "Shape," as
"circle."
    struct Shape sphere = { 3, 1, 5.f };
    // Not all fields were initialized at declaration
    sphere.area = 4 * 3.14f * sphere.radius * sphere.radius;

    printf("\nSphere Info:\n");
    printf("sphere.type: %d\n", sphere.type);
    printf("sphere.color: %d\n", sphere.color);
    printf("sphere.radius: %g\n", sphere.radius);
    printf("sphere.area: %g\n", sphere.area); // 314
    printf("sphere.circumference: %g\n", sphere.circumference);
    // Field implicitly initialized to 0
} // main
```

9.5.3. Ex. 3

```
/* The program reads data for n people (name, surname, date of birth, personal
identification number) and then displays this information */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define MAX 20

const int char_dim = 20;

struct Calendar_Data {
    int day;
    char month[char_dim];
    int year;
};
```

```
struct Personal_Data {
    char last_name[char_dim];
    char first_name[char_dim];
    long code;
    struct Calendar_Data birth_date; // Nested structure
};

void readPersonalData(struct Personal_Data*); // Pointer to structure
void displayPersonalData(struct Personal_Data*);

int main() {
    struct Personal_Data pd[MAX]; // 'pd' is an array of structures
    int i, n;
    printf("\nNumber of employees: ");
    scanf("%d", &n);
    if (n <= 0 || n > MAX) {
        printf("\nInvalid number!");
        return 1;
    }
    printf("\nEnter personal data:");
    for (i = 0; i < n; i++) {
        printf("\nPerson %d:", i);
        readPersonalData(&pd[i]);
    }

    printf("\n\nEntered Persons: %d\n", n);
    for (i = 0; i < n; i++) {
        printf("\t");
        displayPersonalData(&pd[i]);
    }
    return 0;
} // main

void readPersonalData(struct Personal_Data* p) // When called, p <-- &pd[i]
{
    printf("\nLast Name: ");
    scanf("%s", p->last_name);
    printf("\nFirst Name: ");
    scanf("%s", p->first_name);
    printf("\nCode: ");
    scanf("%ld", &p->code);
    printf("\nBirth Date: ");
    printf("\n\tDay: ");
    scanf("%d", &(p->birth_date).day);
    printf("\n\tMonth: ");
    scanf("%s", (p->birth_date).month);
    printf("\n\tYear: ");
    scanf("%d", &(p->birth_date).year);
} // readPersonalData

void displayPersonalData(struct Personal_Data* p) {
    printf("\n Last Name: %s", p->last_name);
    printf("\n First Name: %s", p->first_name);
    printf("\n Code: %ld", p->code);
    printf("\n Birth Date: ");
    printf("\n\t Day: %d", (p->birth_date).day);
```

```
    printf("\n\t Month: %s", (p->birth_date).month);
    printf("\n\t Year: %d\n", (p->birth_date).year);
} // displayPersonalData
```

9.5.4. Ex. 4

```
/* The program takes input from the keyboard for data about n people (name,
surname, date of birth, personal identification number) and then displays the
information. It uses dynamic memory allocation for n data structures of the same
type. The data is sorted based on one field and two fields, and then it is
displayed. The program follows the Abstract Data Type (ADT) approach for this */

//DatePers.h
const int char_dim = 20;

struct Calendar_Data {
    int day;
    char month[char_dim];
    int year;
};

struct Personal_Data {
    char last_name[char_dim];
    char first_name[char_dim];
    long code;
    struct Calendar_Data birth_date;

    void readPersonalData(Personal_Data*); // Methods inside the struct
    void displayPersonalData(Personal_Data*);
};

void Personal_Data::displayPersonalData(Personal_Data* p) {
    cout << "\nLast Name: " << p->last_name;
    cout << "\nFirst Name: " << p->first_name;
    cout << "\nCode: " << p->code;
    cout << "\nBirth Date: ";
    cout << "\n\tDay: " << (p->birth_date).day;
    cout << "\n\tMonth: " << (p->birth_date).month;
    cout << "\n\tYear: " << (p->birth_date).year;
} // displayPersonalData

void Personal_Data::readPersonalData(Personal_Data* p) {
    cout << "\nLast Name: ";
    cin >> p->last_name;
    cout << "\nFirst Name: ";
    cin >> p->first_name;
    cout << "\nCode: ";
    cin >> p->code;
    cout << "\nBirth Date: ";
    cout << "\n\tDay: ";
    cin >> (p->birth_date).day;
    cout << "\n\tMonth: ";
    cin >> (p->birth_date).month;
    cout << "\n\tYear: ";
    cin >> (p->birth_date).year;
} // readPersonalData
```

```
//main() zone
#include <iostream>
using namespace std;
#include "DatePers.h"

int compareCode(const void* a, const void* b);
int compareNameYear(const void* a, const void* b);

int main() {
    struct Personal_Data *pd, person; // Declaration of the pointer pd, object for
method calls
    int i, n;
    cout << "\nEnter the number of employees (int, >0): ";
    cin >> n;
    if (n <= 0) {
        cout << "\nInvalid number (negative or zero)!" << endl;
        exit(1);
    }

    if (!(pd = new (nothrow) Personal_Data[n])) { // Initialize the pd pointer
        cout << "Allocation failed!";
        exit(1);
    }

    cout << "\nEnter personal data:";
    for (i = 0; i < n; i++) {
        cout << "\nPerson: " << i;
        person.readPersonalData(pd + i);
    }

    cout << "\nEntered Persons: ";
    for (i = 0; i < n; i++)
        person.displayPersonalData(pd + i);

    cout << "\n\nPersons sorted by code: ";
    qsort(pd, n, sizeof(Personal_Data), compareCode);
    for (i = 0; i < n; i++)
        person.displayPersonalData(pd + i);

    cout << "\n\nPersons sorted by name and year: ";
    qsort(pd, n, sizeof(Personal_Data), compareNameYear);
    for (i = 0; i < n; i++)
        person.displayPersonalData(pd + i);

    delete[] pd; // Release allocated memory
} // main

int compareCode(const void* a, const void* b) {
    Personal_Data* pa = (Personal_Data*)a;
    Personal_Data* pb = (Personal_Data*)b;
    return (pb->code - pa->code); // Descending order
} // compareCode

int compareNameYear(const void* a, const void* b) {
```

```
    int name_comparison;
    Personal_Data* pa = (Personal_Data*)a;
    Personal_Data* pb = (Personal_Data*)b;
    if ((name_comparison = strcmp(pa->last_name, pb->last_name)) == 0)
        return ((pa->birth_date).year - (pb->birth_date).year);
    // Ascending order by name and year
    return name_comparison;
} // compareNameYear
```

9.5.5. Ex. 5

```
/* The program takes real values from the keyboard and stores them in a one-
dimensional array. It uses a structure to return the sum of the positive values and
the sum of the negative values from the array. */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define MAX 20

struct Sums {
    float positiveSum;
    float negativeSum;
};

Sums calculateSums(float[], int);

int main() {
    int n;
    float values[MAX];
    printf("\nHow many values do you want to read? ");
    scanf("%d", &n);
    if (n <= 0 || n > MAX) {
        printf("\nIncorrect value for the number of values.");
        return 1;
    }
    printf("\nEnter the values:\n");
    for (int i = 0; i < n; i++) {
        printf("values[%d]=", i);
        scanf("%f", &values[i]);
    }
    Sums result =calculateSums(values, n);
    printf("The sum of positive elements is %.2f, and the sum of negative elements
is %.2f", result.positiveSum, result.negativeSum);
    return 0;
} // main

Sums calculateSums(float x[], int n) {
    Sums sums = {}; // Generalized initialization with 0 for numeric fields
    for (int i = 0; i < n; i++) {
        if (x[i] >= 0)
            sums.positiveSum += x[i];
        else
            sums.negativeSum += x[i];
    }
    return sums;
} // calculateSums
```

9.5.6. Ex. 6

```
/* Bit fields
The program uses the SYSTEMTIME structure, the declaration of which is presented
below. To access the mentioned structure, the Windows.h header file must be
included. */

typedef struct _SYSTEMTIME {
WORD wYear;
WORD wMonth;
WORD wDayOfWeek;
WORD wDay;
WORD wHour;
WORD wMinute;
WORD wSecond;
WORD wMilliseconds;
} SYSTEMTIME;


#include <iostream>
using namespace std;
#include <Windows.h>

const char* getWeekday(unsigned short); // Translate the day number into text

struct Date {
    unsigned short nWeekDay : 3; // 0...7 (3 bits)
    unsigned short nMonthDay : 5; // 0...31 (5 bits)
    unsigned short nMonth : 4; // 0...15 (4 bits)
    unsigned short nYear : 11; // 0...2047 (11 bits)
};

int main() {
    // Declare a pointer of type SYSTEMTIME
    Date date;
    SYSTEMTIME* p_st = new (nothrow) SYSTEMTIME;

    // Populate the elements of the SYSTEMTIME structure
    GetSystemTime(p_st);
    // GetLocalTime(p_st); // There's also a function for local time

    // Display the data stored in SYSTEMTIME
    cout << "Current year: " << p_st->wYear << endl;
    cout << "Current month: " << p_st->wMonth << endl;
    cout << "Current day: " << p_st->wDay << endl;
    cout << "Current time (GMT): " << p_st->wHour << endl;
    cout << "Current minute: " << p_st->wMinute << endl;
    cout << "Current second: " << p_st->wSecond << endl;
    cout << "Current millisecond: " << p_st->wMilliseconds << endl;
    printf("\nThe time can also be displayed in the format: %02d:%02d:%02d", p_st-
>wHour, p_st->wMinute, p_st->wSecond);

    // Copy the data from SYSTEMTIME into Date
    date.nYear = p_st->wYear;
    date.nMonth = p_st->wMonth;
```

```
    date.nMonthDay = p_st->wDay;
    date.nWeekDay = p_st->wDayOfWeek;

    // Display the data from Date
    cout << "\nDate displayed with values from Date:\n";
    cout << date.nMonthDay << "(" << date.nWeekDay << ")." << date.nMonth << "." <<
date.nYear << endl;

    // Display the data from Date with the day as text
    cout << getWeekday(date.nWeekDay) << ", " << date.nMonthDay << "." <<
date.nMonth << "." << date.nYear << endl;

    delete p_st;
} // main

const char* getWeekday(unsigned short nWeekday) {
    const char* weekdayStrings[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday" };
    // The array of pointers to character strings is local to the function but
persists after return.
    return weekdayStrings[nWeekday]; // or return *(weekdayStrings + nWeekday);
} // getWeekday
```

9.5.7. Ex. 7

```
/* Unions - example 1. */

#include <iostream>
using namespace std;

int main() {
    // Anonymous inner union
    union {
        int data1;
        float data2;
    };

    data1 = 3;
    cout << "The value in the data1 field is: " << data1;

    data2 = 1.2345;
    cout << "\nThe value in the data2 field is: " << data2;
    cout << "\nOnce again, the value in the data1 field is: " << data1 << endl;
}//main
```

9.5.8. Ex. 8

```
/* Unions - example 2. */

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>

using namespace std;
```

```
int main() {
    union {
        long l;
        double d;
        char s[4];
    } vr;

    vr.l = 100000;
    cout << vr.l << " "; // Print the value stored in the long field of the union.

    vr.d = 123.2342;
    cout << vr.d << " "; // Print the value stored in the double field of the
union.

    strcpy(vr.s, "hi\n");
    cout << vr.s; // Print the string stored in the char array field of the union.

    return 0;
}//main
```

9.5.9. Ex. 9

```
/* Unions - example 3. */

#include <stdio.h>

int main() {
    union EmployeeDates {
        int days_worked;

        struct Date {
            int month;
            int day;
            int year;
        } last_day;
    } emp_info;

    union Numbers {
        int a;
        float b;
        long c;
        double d; // the largest field
    } value;

    printf("The size of the EmployeeDates union is: %d bytes.\n",
sizeof(emp_info));
    printf("The size of the Numbers union is: %d bytes.\n", sizeof(value));
    return 0;
}//main

/* The size of the EmployeeDates union is 12 bytes.
   The size of the Numbers union is 8 bytes. */
```

9.5.10. Ex. 10

```
/* Enumeration - example 1. */
```

```
#include <stdio.h>
int main() {
    enum Weekdays { Luni = 1,
        Marti,
        Miercuri,
        Joi,
        Vineri,
        Sambata };
    printf("%d %d %d %d %d %d\n", Luni, Marti, Miercuri, Joi, Vineri, Sambata);
    return 0;
} //main
```

9.5.11. Ex. 11

```
/* Enumeration - example 2. */

#include <stdio.h>
int main() {
    enum Weekdays {
        Monday = 10,
        Tuesday = 20,
        Wednesday = 30,
        Thursday = 40,
        Friday = 50,
        Saturday = 60
    };
    printf("%d %d %d %d %d %d\n", Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday);
    return 0;
}//main
```

9.5.12. Ex. 12

```
/* The program translates between the abbreviated and full names of the months. */

#include <iostream>
using namespace std;
enum Month { Jan = 1,
    Feb,
    Mar,
    Apr,
    May,
    Jun,
    Jul,
    Aug,
    Sep,
    Oct,
    Nov,
    Dec };

const char* monthStr(Month month);

int main() {
    cout << "\nMonth " << Aug << " is: " << monthStr(Aug) << '\n';
```

```
    cout << "\nMonth " << Dec << " is: " << monthStr(Dec) << '\n';
    return 0;
}//main

const char* monthStr(Month month)
{
    switch (month) {
    case Jan:
        return "January";
    case Feb:
        return "February";
    case Mar:
        return "March";
    case Apr:
        return "April";
    case May:
        return "May";
    case Jun:
        return "June";
    case Jul:
        return "July";
    case Aug:
        return "August";
    case Sep:
        return "September";
    case Oct:
        return "October";
    case Nov:
        return "November";
    case Dec:
        return "December";
    default:
        return " ";
    }
}//monthStr
```

9.6. Întrebări

1. Cum se numesc elementele unei structuri ?
2. Cum se face accesul la elementele unei structuri ?
3. Ce sunt structurile îmbricate?
4. Cum se inițializează câmpurile unei structuri?
5. Ce înțelegeți printr-un câmp de biți?
6. Cum se pot referi câmpurile de biți?
7. Care sunt diferențele dintre structuri și reuniuni?
8. Ce înțelegeți prin tipul enumerare?
9. Cum se poate asigna un nume unui tip de date?

9.7. Questions

1. What are the elements of a structure called?
2. How is access to the elements of a structure done?
3. What are nested structures?
4. How are the fields of a structure initialized?

5. What do you understand by a bit field?
6. How can bit fields be referred to?
7. What are the differences between structures and unions?
8. What do you understand by the enumeration type?
9. How can a name be assigned to a data type?

9.8. Lucru individual

1. Să se scrie un program C/C++, în care definiți o structură și scrieți două funcții care permit afișarea unei variabile de tipul structurii. În cazul primei funcții se transferă ca parametru o variabilă de tip structură de date prin valoare, iar în cazul celei de a doua funcții variabila se va transmite prin adresă (folosind pointeri). În funcția main( ) inițializați câmpurile structurii cu date citite de la tastatură. În ambele funcții afișați datele din structură folosind un mesaj adecvat.
2. Să se scrie un program C/C++, în care o funcție returnează o structură de date adecvată. În acest fel vor fi returnate mai multe valori. Afișați rezultatul, valorile inițiale transferate funcției (puteți realiza orice operație în cadrul acelei funcții), cu mesaje adecvate. De exemplu, pornind de la exemplul 5 de la structuri, returnați suma numerelor pare și a celor prime dintr-un tablou de numere întregi pozitive dat ca parametru.
3. Să se scrie o aplicație C/C++, care utilizând o structură de tip Angajat, să afișeze toate datele persoanelor cu ocupația inginer dintr-o întreprindere (nume, prenume, ocupația, data nașterii, secția în care lucrează).
4. Folosind structura de la exemplul precedent , să se scrie un program care citește datele personale pentru n persoane (nume, prenume, data nașterii, codul numeric personal, data angajării) și apoi le afișează în ordinea datei angajării.
5. Să se scrie un program C/C++, care folosind o structură numită Student, având câmpurile {nume, prenume, țara de origine, grupa, anul nașterii}, să determine numărul de studenți străini din grupă (grupă de MAX=30 studenți) și să afișeze toate datele acestora. Datele pentru studenții din grupă se citesc de la intrarea standard, până la întâlnirea numelui AAA (litere mici sau mari).
6. Să se scrie un program care afișează numele, prenumele și media studentului cu cele mai bune rezultate din grupă în urma sesiunii de iarnă. Folosiți pentru aceasta un tablou de structuri, de un tip numit Student, alocarea dinamică, și o funcție care returnează înregistrarea Student cu media cea mai mare.
7. Să se scrie o aplicație C/C++, care alocă dinamic memorie pentru datele a n studenți (nume, prenume și gen), citește datele pentru fiecare dintre aceștia, afișează numărul studentelor și eliberează memoria alocată.
8. Declarați un nou tip de date O_struct, care să conțină o variabilă de tip întreg, una de tip caracter și un șir de 256 de caractere. Definiți în main( ) o variabilă statică de tip O_struct, căreia să-i inițializați câmpurile cu date citite de la intrarea standard. Declarați apoi un pointer de tip O_struct, numit po_struct, care va fi definit prin alocarea dinamică a unei zone de memorie care să conțină un articol de tip O_struct. Inițializați câmpurile structurii de date folosind pointerul po_struct. Afișați toate câmpurile și eliberați zona de memorie alocată.
9. Scrieți o aplicație C/C++, care alocă dinamic memorie pentru memorarea datelor a n produse, folosind o structură Produs, cu câmpurile denumire, pret, cantitate, citește datele pentru fiecare dintre produse și afișează produsul din care avem cel mai mult pe stoc. În final va elibera memoria alocată.
10. Să se definească o structură cu numele Masina, care are câmpurile {producător, anul fabricației, capacitatea cilindrică și culoare}. Să se aloce dinamic memorie pentru n date de

tip Masina și să se citească informațiile pentru acestea. Să se afișeze înregistrările mașinilor de culoare roșie, fabricate după anul 2020.

11. Folosind structura de date union denumită Grup, compusă din diferite câmpuri (int, long, double, char, etc.), scrieți o aplicație C/C++ care va inițializa un element de tipul Grup de la tastatură. Este posibil să afișăm în același timp toate câmpurile folosind pointeri sau nume calificate, având valori corecte? Afișați ceea ce este posibil și dimensiunea elementului de tip union. Realizați aceeași operație considerând o simplă structură struct cu aceleași câmpuri.
12. Definiți o dată de tip enumerare enum Lumina_Alba care va avea în componență culorile de bază (Roșu, Portocaliu, Galben, Verde, Albastru, Indigo și Violet). Inițializați câteva variabile/obiecte de tip Lumină_Alba. Urmăriți să generați culori secundare cu ajutorul operațiilor specifice unei enumerări prin combinații ale culorilor de bază. Traduceți printr-un mecanism folosind enumerări, numele culorilor în limba Franceză, Engleză sau Germană și afișați toate culorile, menționând numele inițial, noul nume și valoarea asociată.

9.9. Individual work

1. Write a C/C++ program in which you define a structure and create two functions that allow you to display a variable of that structure type. In the case of the first function, a data structure variable is passed as a value parameter, and in the case of the second function, the variable will be passed by reference (using pointers). In the main() function, initialize the structure's fields with data read from the keyboard. In both functions, display the data within the structure using an appropriate message.
2. Develop a C/C++ application considering an adequate data structure that will be returned by a function. In this way more values can be returned. Display the results, the initial values transferred to the function (doing whatever operation inside the function) using adequate messages. As an example, starting from example 5 from structures, return the sum of even and prime numbers from an array of positive integers given as a parameter to the function.
3. Using included structures, Data_calend with fields day, month, year and Personal_data with fields name, surname, occupation, code, department, birth_date and empl_date of type Data_calend, generate an array of structures of type Personal_data, containing couple of employees (max. 20), reading their data from the keyboard. Considering “engineer”, “teacher”, “student” and “manager” as possible values for the field occupation, display all engineer’s records.
4. Using the previous array of structures, generate a list of records being sorted in ascending order by their code, and in descending order by the empl_date.
5. Develop a C/C++ application considering an adequate data structure named Student having the fields: name, surname, country, group and birth_year. Count all the non- Romanian students from the group (MAX students in the group). The effective fields will be introduced from the keyboard generating an array of structures. A name AAA (upper or lower case) will finish the introduction process.
6. Develop a C/C++ program that displays the name, surname, and media of the student with the best results in the group after the winter exams. Define a user-type array of structures named Student, using dynamic memory allocation and a function that will return the record of the best student.
7. Write a C/C++ application that allocates dynamically memory for the data of n students (surname, name, gender), reading from the keyboard all the required info, the program displays the number of female students and frees up the allocated memory.
8. Declare a structure named A_structure that contains as fields one integer and one character-type variable and an array of 256 characters. Define in the main() function a variable of type A_structure and initialize its fields with data read from the keyboard.

Declare a pointer, named pa_structure and initialize-it by allocating memory for a single variable of type A_structure. Use this pointer to define all the fields of your variable with data read from the keyboard. Display all the fields of the structure, then free up the allocated memory.

9. Write a C/C++ application that allocates the necessary amount of memory for storing n products, using a structure named Product having the fields: name, price, quantity. After reading from the keyboard each product's data, display the item that has the biggest stock value. Free up the allocated memory.
10. Define a structure named Car that contains the following variables: producer, production_year, cylinder_volume and color. Store in a newly allocated zone of memory the data for n cars. Display the records for the red cars produced after 2020.
11. Write a C/C++ application that defines a union called group that contains various fields (int, log, double, char, etc.). Read from the keyboard the data associated with a group variable. Is it possible to display all the fields in the union using pointers or fully qualified field names, as correct values? Display the accessible information and the dimension of the union variable. Perform the same operations considering a regular structure with the same fields instead of the union.
12. Define an enumeration called White_Light that will contain the basic colors (red, orange, yellow, green, blue, indigo and violet). Initialize a few variables of type White_Light and try to generate some secondary colors by combining the basic colors mentioned above. Use an enumeration-based mechanism for translating the color names into French, English or German. Display all the colors mentioning the original and the translated names and the associated value.

10. Fișiere text. Fișiere binare. Fișiere în acces direct

Text files. Binary files. Direct access files

10.1. Obiective

- Scrierea de programe folosind fișiere text.
- Înțelegerea modalității de lucru cu fișiere binare
- Înțelegerea accesului aleator în fișiere
- Scrierea de programe folosind fișiere binare

10.2. Objectives

- Writing programs using text files.
- Understanding how the binary files work
- Understanding the random-access files
- Writing programs that use binary files.

10.3. Breviar teoretic

10.3.1. Fluxuri (Stream-uri) de date

Sistemul de intrări/ieșiri separă programatorul de echipament printr-o așa-zisă abstractizare a datelor care circulă în permanență între mașina de lucru și program.

Această formă abstractă de comunicare se numește flux (stream), iar instrumentul efectiv de comunicare se numește fișier (file). Stream-ul asigură un canal de comunicație între două componente hardware sau software. Una din componente e numită sursă, ea transmite date pe acest canal, iar cealaltă destinație, ea recepționând datele din canal.

10.3.2. Stream-uri de tip text

Acest tip de flux este practic o secvență de caractere ASCII sau UNICODE. Standardul ANSI C permite ca un stream de tip text (caractere ASCII) să fie organizat pe linii, fiecare linie fiind terminată cu un caracter de control, de linie nouă, LF (Line Feed), având codul ASCII 0*x0A*. Utilizarea acestui caracter este opțională, folosirea sa depinzând de programator și de modul de implementare a aplicației respective. La fișiere text apăsarea tastei ENTER este translatată într-o secvență de două caractere, CR și LF (*0x0D 0x0A*). În acest mod nu va exista o corespondență totală între ceea ce este transmis și ceea ce conține fișierul.

10.3.3. Stream-uri standard

Noțiunea de fișier în limbajul C/C++ este privită într-un mod mai larg. Se poate asocia un stream unui fișier prin operația de deschidere, urmând apoi a se transmite datele între program și celălalt capăt al conexiunii logice. La lansarea unei aplicații C/C++ automat devin active mai multe fluxuri standard și sunt închise automat la încheierea execuției. Acestea sunt:

- *stdin*: intrare standard; stream de tip text;
- *stdout*: ieșire standard; stream de tip text;

- *stderr*: ieșire standard erori; stream de tip text;

După atingerea scopului pentru care a fost folosit fișierul respectiv, urmează eliberarea conexiunii logice (și a resurselor alocate) prin efectuarea operației de închidere. În general, înaintea închiderii propriu-zise se golește stream-ul asociat (flushing) pentru a fi siguri că nu rămân date blocate în bufferul de disc asociat conexiunii. În cazul în care nu se realizează închiderea explicită prin program a unui fișier, el este închis automat în momentul terminării execuției programului care a deschis respectivul fișier. Dacă execuția programului este terminată prematur, există posibilitatea ca stream-urile respective să rămână deschise.

Funcții de intrare-ieșire la nivel consolă - definite în <conio.h> (lucrează fără buffer)

Aceste funcții nu mai sunt recunoscute de majoritatea compilatoarelor C/C++ moderne ce nu mai permit accesul direct de la consolă.

- `int getch(void);` //preia un caracter de la tastatură, VC++ `int _getch( )`
- `int getche(void);`

  la fel ca `getch()`, dar îl și afișează pe ecran (preluare cu ecou); //VC++ `int _getche( )`
- `int putch(int caracter);`
- scrie caracterul indicat pe ecran, valoarea returnată fiind fie caracterul de afișat (în caz de succes), fie caracterul EOF în caz de eșec. Codul ASCII al EOF e definit în <stdio.h> ca valoare -1, care în C2, pe un octet, este 0xFF.

Alte funcții standard de intrare-ieșire definite în `<stdio.h>` (lucrează cu buffer)

```
char* gets(char* sir);
```

- extrage un șir de caractere din stdin până la întâlnirea caracterului LF '\n', pe care îl depune în șirul de octeți șir, urmat de '\0'. În acest mod se pot citi șiruri de caractere care pot conține și spații. Funcția returnează adresa șirului citit în caz de succes sau un pointer NULL în caz contrar;

```
int puts(const char* sir);
```

- trimite șirul de octeți șir la fluxul stdout și returnează o valoare mai mare decât 0 în caz de succes și EOF în caz contrar;

```
int getchar(void);
```

- preia un caracter din fluxul stdin, în caz de succes returnează caracterul citit sau EOF în caz contrar.

```
int putchar(int caracter);
```

- inserează caracterul de afișat în fluxul de ieșire stdout și returnează caracterul inserat în caz de succes sau EOF în caz contrar.

Platforma Microsoft Visual Studio compilatoarele VC++1y/2z oferă funcția `gets_s()` ce permite optimizarea/securizarea introducerii șirurilor de caractere. Această funcție folosește un parametru suplimentar, lungimea șirului. La citiri combinate de șiruri de caractere și alte tipuri de date (int, float, …) e necesar a ignora ultimul caracter din buffer ce se face apelând `getchar()`, sau `fflush(stdin)`, sau `cin.get()`, în funcție de context.

10.3.4. Tratarea fișierelor la nivel inferior - opțional

Acest mod de abordare a fișierelor este dependentă de sistemul de operare și este în consecință, mult mai puțin folosită. Funcțiile de lucru la acest nivel sunt descrise în fișierele header: `<io.h>`, `<stat.h>`, `<fcntl.h>`.

Principalele funcții de la acest nivel sunt:

```
int open(const char* path, int access [, unsigned mode]);
```

- unde path reprezintă calea spre fișier, iar parametrul access specifica unul din modurile de deschidere *r, w, r+w, a*, etc. Returnează un descriptor de fișier, file handle, argument pentru funcțiile `read(), write()`

```
int create(const char *path, int amode);
```

- unde *path* reprezintă *calea* spre noul fișier ce va fi creat în unul din modurile date de *amode*; Exemplu: `handle = creat("FILENAME.EXT", S_IREAD | S_IWRITE);`

```
int read(int handle, void* buf, unsigned len);
```

- se încearcă citirea a *len* octeți în bufferul de memorie *buf* din fișierul asociat cu *handle*;

```
int write(int handle, void *buf, unsigned len);
```

- semnificația variabilelor este aceeași ca și în cazul de mai sus ;

```
long lseek(int handle, long offset, int fromwhere);
```

- poziționează cursorul în fișierul *handle* la distanța *offset* octeți, începând de la *fromwhere*. Ultima variabilă are aceleași posibile valori ca și în cazul funcției *fseek()*;

```
int close(int handle);
```

- închide fișierul descris de *handle* și returnează –1 în caz de eroare sau 0 dacă operația s-a efectuat cu succes. Exemplu: `close(handle);`

10.3.5. Tratarea fișierelor la nivel superior

Pentru ca funcțiile specifice lucrului cu fișiere să fie recunoscute în program, trebuie inclus fișierul header `<stdio.h>`. Acesta asigură prototipurile pentru funcțiile de I/O și definește următoarele tipuri de date: `size_t, fpos_t` și `FILE`. Primele două reprezintă varietăți de întreg fără semn, iar tipul `FILE` este o structură având ca elemente atributele fișierului.

Macrourile pentru lucrul cu fișiere, definite în `stdio.h`, sunt următoarele:

- **NULL**, definește un pointer nul;
- **EOF**, este definit în general ca fiind –1 și reprezintă valoarea returnată când o funcție încearcă să citească peste sfârșitul fișierului; în structura `FILE` se află un câmp, *unsigned flags;* în care sunt mai multe flag-uri, inclusiv cel legat de `EOF` care se poate poziționa doar după o operație efectuată asupra fișierului
- **FOPEN_MAX** definește numărul maxim de fișiere care pot fi deschise simultan;
- **SEEK_SET**, **SEEK_CUR**, **SEEK_END** sunt folosite împreună cu funcția `fseek()` și ajută la poziționarea în fișier la început (SEEK_SET), la poziția curentă (SEEK_CUR) sau la sfârșit (SEEK_END).

Pointerul fișierului este legătura dintre fișier și sistemul de I/O definit de standardul ANSI C. Prin intermediul acestuia se vehiculează toate informațiile în și dinspre fișier. Exemplu: `FILE* fp;`

Funcții standard C/C++ specifice lucrului cu fișiere în mod text:

Nume funcție	**Efect**
fopen()	Deschide un fișier
fclose()	Închide un fișier
fputc()	Scrie un caracter în fișier
fgetc()	Citește un caracter din fișier
fputs()	Scrie un șir de caractere în fișier
fgets()	Citește un șir de caractere din fișier
fseek()	Poziționează cursorul la un anumit octet în fișier
fprintf()	Scrie date formatate în fișier
fscanf()	Citește date formatate din fișier
ftell()	Permite citirea indicatorului de poziție
fgetpos()	Citește valoarea poziției curente
fsetpos()	Setează valoarea indicatorului de poziție
feof()	Returnează true dacă se ajunge la sfârșitul fișierului
ferror()	Returnează true dacă a apărut o eroare
rewind()	Readuce indicatorul de poziție la începutul fișierului
remove()	Șterge un fișier
fflush()	Golește un stream asociat unui fișier

Deschiderea unui fișier folosind standardul C/C++

Prototipul funcției de deschidere a unui fișier, fopen(), este următorul:

```
FILE* fopen(const char* file_name, const char* mod);
```

unde `file_name` reprezintă numele fișierului care va fi deschis în modul de acces definit de `mod`. În tabelul următor sunt prezentate modurile de deschidere ale unui fișier:

Mod de deschidere	Semnificație
r	Deschide un fișier text pentru citire
w	Deschide un fișier text pentru scriere
a	Adaugă într-un fișier text
rb	Deschide un fișier binar pentru citire
wb	Deschide un fișier binar pentru scriere
ab	Adaugă într-un fișier binar
r+	Deschide un fișier text pentru citire/scriere (nu creează dacă nu există, nu distruge dacă există)
w+	Creează un fișier text pentru citire/scriere (creează dacă nu există, distruge dacă există)
a+	Adaugă sau creează un fișier text pentru citire/scriere (creează dacă nu există, adaugă la sfârșit dacă există)
r+b sau rb+	Deschide un fișier binar pentru citire/scriere
w+b sau wb+	Creează un fișier binar pentru citire/scriere
a+b sau ab+	Adaugă sau creează un fișier binar pentru citire/scriere

La fișierele text se poate adăuga suplimentar în mod explicit și caracterul *t,* cum ar fi în loc de *r* să se specifice *rt*.

Pe platformele de la Microsoft trebuie folosită directiva următoare:

```
#define _CRT_SECURE_NO_WARNINGS
```

10.3.6. Închiderea unui fișier

Se realizează prin apelul funcției

```
int fclose(FILE* fp);
```

și are ca efect închiderea fluxului deschis în prealabil către fișierul respectiv.

Operația de închidere trebuie efectuată numai într-un context adecvat (după ce toate procesele de citire/scriere au fost încheiate), pentru că în caz contrar, pot apărea pierderi de date din fișier, poate fi distrus fișierul sau chiar pierdut. După închiderea unui fișier, se eliberează blocul de control al acestuia, fiind disponibil pentru a fi reutilizat.

Platforma Visual Studio, compilatoarele Visual C++1y/2z au introdus o altă sintaxă pentru funcția de deschidere a unui fișier `fopen_s();`

```
errno_t  fopen_s( FILE**  pFile,  const char *filename,  const char *mode
);
```

- `pFile` – e un pointer către file pointerul asociat fișierului deschis.
- `filename` – nume fișier
- `mode` – tipul de acces permis care permite și considerarea de fișiere UNICODE:

  ```
  fopen_s(&fp, "newfile.txt", "w+, ccs=UNICODE");
  ```

- `errno`, flag eroare (int); dacă e 0 e OK, dacă nu înseamnă că a apărut eroare la deschidere

Scrierea în fișier se realizează prin intermediul următoarelor funcții în mod standard:

```
int fputc(int c, FILE* stream);
```

- scrie caracterul c în fișierul identificat de pointerul stream. Funcția returnează caracterul c în caz de reușită sau EOF în caz contrar.

```
int fputs(const char* s, FILE* stream);
```

- scrie șirul pointat de s în fișierul stream. Returnează o valoare pozitivă în caz de reușită sau EOF în caz contrar.

```
int fprintf(FILE* stream, const char* format[, argument, ...]);
```

- are parametrii similari cu `printf()`, singura deosebire fiind apariția pointerului stream la fișierul unde va avea loc scrierea. Returnează numărul de octeți scriși sau, în caz de eșec, EOF. În specificatorul de format trebuie folosite și caracterele de formatare dorite cum ar fi '\n'.

Citirea din fișier are drept suport următoarele funcții în mod standard:

```
int fgetc(FILE* stream);
```

- citește un caracter de la poziția curentă din fișierul indicat de stream. În caz de succes, returnează caracterul citit, altfel EOF.

```
char* fgets(char* s, int n, FILE* stream);
```

- depune în șirul s un șir de caractere de lungime n citit din fișierul stream. Returnează șirul în caz de succes sau NULL în caz de sfârșit prematur de fișier sau în caz de eroare.

```
int fscanf(FILE* stream, const char* format[, address, ...]);
```

- citește din fișierul stream date formatate, analog cu funcția `scanf()`.

Platforma Microsoft Visual Studio compilatoarele VC++1y/2z oferă funcția `fscanf_s()`, ca un mecanism de securitate/optimizare în cadrul platformei. Folosind directiva:

```
#define _CRT_SECURE_NO_WARNINGS
```

se pot folosi funcțiile standard existente în C/C++ cu fișiere.

10.3.7. Funcțiile feof() și ferror()

```
int feof(FILE* fp);
```

verifică în structura FILE asociată fișierului la deschidere flag-ul _F_EOF, dacă la accesul anterior s-a atins sau nu sfârșitul de fișier. Returnează o valoare diferită de zero dacă s-a ajuns la sfârșitul fișierului sau zero în caz contrar.

```
int ferror(FILE* fp);
```

returnează o valoare pozitivă dacă s-a produs o eroare la citire/scriere sau 0 în caz de operație reușită.

10.3.8. Stream-uri binare

Un stream binar este o secvență de octeți aflată într-o corespondență biunivocă cu cei de la echipamentul extern. La tratarea la nivel binar aceste secvențe de octeți sunt completate ("padding") sau segmentate, pentru a aduce pachetul de date la o lungime standard cerută de protocolul de comunicare, (de exemplu un disc poate avea 512 octeți/sector sau un cadru IP poate transporta 1500 octeți "payload"), dar aceste operații sunt transparente pentru utilizator. În cazul fluxurilor binare nu apar translatări de date (de tipul 0A <---> 0D 0A, cum erau la fluxuri de tip text).

10.3.9. Lucrul cu fișiere în mod binar

În cazul fișierelor binare, informația stocată în interiorul acestora nu este lizibilă. Dacă totuși cineva forțează deschiderea unui fișier binar cu un editor de texte (de ex. Notepad), "caracterele" care vor fi afișate pe ecran par un amestec de caractere alfanumerice fără sens, incluzând caractere semigrafice (cod ASCII > 7Fh) și caractere de control.

Citirea/scrierea fișierelor binare se face cu următoarele două funcții, declarate în <stdio.h>:

```
size_t  fread(void* ptr, size_t size, size_t n, FILE* stream);
size_t  fwrite(const void* ptr, size_t size, size_t n, FILE* stream);
```

`ptr` - un pointer către un buffer de memorie care va primi datele din/spre fișier,

`size` - dimensiunea în octeți a unui articol de tipul celor care vor fi citite/scrise din/în fișier.

Articol în acest context înseamnă un pachet de octeți, de o dimensiune ce depinde de aplicație. Poate fi de exemplu un element al unui tablou de structuri, numită înregistrare (în cazul unei baze de date), un cadru (în cazul transmiterii datelor la distanță), etc.

`n` - numărul articolelor care urmează să fie citite/scrise

`stream` - specifică fișierul deschis în prealabil, este acel "file-handle" returnat de `fopen()`

Deschiderea și închiderea fișierelor binare se face cu aceleași funcții ca și la fișierele de tip text, cu `fopen()/fopen_s()` și `fclose()`.

10.3.10. Funcții specifice accesului aleator

Accesul la un octet înregistrat poate fi secvențial (ca de exemplu la "tape drive", cu organizare unidimensională) sau direct, numit aleator (de exemplu, prin coordonate cilindru, cap magnetic, pistă, sector, etc.). Pentru sistemul de fișiere conținutul fișierului e transparent. Programatorul însă va utiliza funcții diferite pentru a accesa conținutul fișierelor text sau a celor binare.

```
long int ftell(FILE* fp);
```

- permite citirea indicatorului de poziție, care conține offset-ul octetului de început pentru următorul acces. Primul octet de date este la offset zero. Indicatorul de poziție se actualizează automat după fiecare acces. Funcția returnează poziția curentă din fișierul specificat de pointerul fp în caz de reușită sau valoarea -1L (valoarea lui EOF, -1, extinsă cu semn pe 4 octeți, în C2), în caz de eșec

```
int fgetpos(FILE* fp, long int* poz);
```

- citește valoarea poziției curente și o înscrie în variabila `poz`. Returnează valoarea 0 în caz de succes, sau una diferită de zero în caz de eroare, setând variabila globală `errno` la o valoare pozitivă, care poate fi interpretată cu functia `perror()`

```
int fsetpos(FILE* fp, const long int* poz);
```

- se atribuie valoarea variabilei `poz` la indicatorul de poziție asociat fișierului indicat de `fp`. Altfel spus poziționează cursorul în fișier. Se returnează valoarea 0 în caz de succes, ca și la `fgetpos()`.

```
int fseek(FILE* fp, long nr_oct, int origine);
```

- Deplasează indicatorul de poziție din poziția origine cu un număr de octeți `nr_oct` numit și "offset" (număr cu semn). Returnează valoarea 0 în caz de reușită sau o valoare nenulă în caz de eșec. Poziția de referință "`origine`" poate avea trei valori: 0 = SEEK_SET : început de fișier; 1 = SEEK_CUR : poziția curentă a cursorului; 2 = SEEK_END : sfârșit de fișier; unde SEEK_SET etc. sunt constante simbolice de limbaj, definite în `<stdio.h>`.

```
void rewind(FILE *fp);
```

- permite poziționarea indicatorului de poziție la începutul fișierului. Se șterge și indicatorul de eroare asociat fișierului.

10.3.11. Alte funcții referitoare la fișiere

```
FILE * freopen(const char *filename, const char *mode, FILE *stream);
```

Asociază un nou fișier generic la fluxul sursă deschis (de exemplu redirectează stdout la stdprn sau către un fișier pe disc)

```
int rename(const char *oldname, const char *newname);
```

Redenumește un fișier prin program

```
void clearerr(FILE *fp);
```

Șterge indicatorii de eroare și de sfârșit fișier

10.4. Theoretical brief

10.4.1. Data streams

The input/output system separates the programmer from the equipment through a so-called abstraction of data that constantly flows between the workstation and the program. This abstract form of communication is called a stream, and the actual communication tool is called a file. The stream provides a communication channel between two hardware or software components. One of the components is called the source, which transmits data on this channel, and the other is the destination, which receives the data from the channel.

10.4.2. Text streams

Text streams are essentially a sequence of ASCII or UNICODE characters. The ANSI C standard allows a text stream (ASCII characters) to be organized into lines, with each line terminated by a control character, a newline character, LF (Line Feed), with ASCII code 0x0A. The use of this character is optional, depending on the programmer and the application's implementation. In text files, pressing the ENTER key is translated into a sequence of two characters, CR and LF (0x0D 0x0A). This way, there won't be a complete correspondence between what is transmitted and what the file contains.

10.4.3. Standard streams

In the C/C++ programming languages, the concept of a file is seen more broadly. A stream can be associated with a file through the process of opening it, and then data can be transmitted between the program and the other end of the logical connection. When a C/C++ application is launched, several standard streams become active automatically and are closed automatically at the end of execution. These are:

- stdin: standard input; a text stream.
- stdout: standard output; a text stream.
- stderr: standard error output; a text stream.

After the file has served its purpose, the logical connection (and allocated resources) is released by performing the closing operation. Generally, before the actual closing, the associated stream is emptied (flushed) to ensure that no data remains stuck in the disk buffer associated with the connection. If a file is not explicitly closed by the program, it is automatically closed when the program that opened it terminates. If the program execution terminates prematurely, there is a possibility that the respective streams remain open

Console input-output functions defined in `<conio.h>` (work without buffering):

These functions are no longer recognized by most modern C/C++ compilers, as they no longer allow direct console access.

- `int getch(void);` retrieves a character from the keyboard.
- `int getche(void);` similar to `getch()`, but it also displays the character on the screen (echoing).
- `int putch(int character);` writes the specified character to the screen, and the returned value is either the character to be displayed (in case of success) or the EOF character (in case of failure). The ASCII code of EOF is defined in `<stdio.h>` with a value of -1, which in C2, on one byte, is 0xFF.

Other standard input-output functions defined in `<stdio.h>` (work with buffering).

```
char* gets(char* sir);
```

- extracts a string of characters from stdin until it encounters the LF character '\n', then stores it in the byte string 'str,' followed by '\0'. This way, you can read strings of characters that can also contain spaces. The function returns the address of the read string in case of success or a NULL pointer in case of failure.

```
int puts(const char* sir);
```

- It sends the byte string 'str' to the stdout stream and returns a non-negative value in case of success and EOF in case of failure.

```
int getchar(void);
```

- It retrieves a character from the stdin stream, and in case of success, it returns the read character, or EOF in case of failure. (It's a macro function).

```
int putchar(int caracter);
```

- It inserts the character to be displayed in the stdout output stream and returns the inserted character in case of success or EOF in case of failure. (It's a macro function).

Microsoft Visual Studio platform and VC++1y/2z compilers provide the `gets_s()` function, which allows for optimizing and securing the input of character strings. This function uses an additional parameter, the length of the string. When reading a combination of character strings and other data types (int, float, etc.), it's necessary to ignore the last character in the buffer by using `getchar()`, `fflush(stdin)`, or `cin.get()`, depending on the context.

10.4.4. Handling files at a lower level - optional

This approach to file handling is dependent on the operating system and is, therefore, much less commonly used. The functions for working at this level are described in the header files: `<io.h>`, `<stat.h>`, `<fcntl.h>`.

The main functions at this level are:

```
int open(const char* path, int access [, unsigned mode]);
```

- where '`path`' represents the file path, and the '`access`' parameter specifies one of the opening modes, such as 'r,' 'w,' 'r+w,' 'a,' etc. It returns a file descriptor, file handle, which can be used as an argument for `read()` and `write()` functions.

```
int create(const char *path, int amode);
```

- where '`path`' represents the path to the new file that will be created in one of the modes given by 'amode.' For example: 'handle = creat("FILENAME.EXT", S_IREAD | S_IWRITE);

```
int read(int handle, void* buf, unsigned len);
```

- an attempt is made to read '`len`' bytes into the memory buffer '`buf`' from the file associated with '`handle`'

```
int write(int handle, void *buf, unsigned len);
```

- The meaning of the variables is the same as in the case above.

```
long lseek(int handle, long offset, int fromwhere);
```

- It positions the cursor in the file associated with '`handle`' at a distance of '`offset`' bytes, starting from '`fromwhere`' The last variable has the same possible values as in the `fseek()` function.

```
int close(int handle);
```

- it closes the file described by '`handle`' and returns -1 in case of an error or 0 if the operation was successful. For example: `close(handle);`

10.4.5. Handling files at a higher level.

In order for the functions specific to working with files to be recognized in the program, the `<stdio.h>` header file must be included. This header file provides prototypes for I/O functions and defines the following data types: `size_t`, `fpos_t`, and `FILE`. The first two represent unsigned integer types, and the `FILE` type is a structure with file attributes as its elements.

The macros for working with files, defined in <stdio.h>, are as follows:

- NULL, defines a null pointer.
- EOF, is generally defined as -1 and represents the value returned when a function tries to read past the end of the file. In the `FILE` structure, there is a field of unsigned flags that contains multiple flags, including the one related to `EOF`, which can be set only after an operation on the file.
- FOPEN_MAX, defines the maximum number of files that can be opened simultaneously.
- SEEK_SET, SEEK_CUR, SEEK_END are used together with the `fseek()` function and help in positioning within a file at the beginning (SEEK_SET), at the current position (SEEK_CUR), or at the end (SEEK_END)."

The file pointer is the link between the file and the I/O system defined by the ANSI C standard. All information to and from the file is conveyed through it. For example: `FILE* fp;`

Standard C/C++ functions specific to working with text files:

Function name	**Effect**
fopen()	Opens a file
fclose()	Closes a file
fputc()	Writes a character into a file
fgetc()	Reads a character from a file
fputs()	Writes a string into a file
fgets()	Reads a string from a file

fseek()	Positions the cursor at a specific byte in the file
fprintf()	Writes formatted data into a file
fscanf()	Reads formatted data from a file
ftell()	Enables reading the cursor position
fgetpos()	Reads the value from the current position in the file
fsetpos()	Sets the value of the position cursor
feof()	Returns true when reaching the end of the file
ferror()	Returns true if an error occurs
rewind()	Resets the cursor to the start of the file
remove()	Deletes a files
fflush()	Empties a stream associated to a file

Opening a file using the C/C++ standard.

The prototype of the file opening function, fopen(), is as follows:

```
FILE* fopen(const char* file_name, const char* mod);
```

Where `'file_name'` represents the name of the file to be opened in the access mode defined by 'mode.' The following table presents the file opening modes:

Opening mode	Meaning
r	Open a text file for reading
w	Truncate to zero length or create a text file for writing
a	Append; open or create text file for writing at end-of-file
rb	Open binary file for reading
wb	Truncate to zero length or create a binary file for writing
ab	Append; open or create binary file for writing at end-of-file
r+	Open text file for update (reading and writing)
w+	Truncate to zero length or create a text file for update
a+	Append; open or create text file for update
r+b or rb+	Open binary file for update (reading and writing)
w+b or wb+	,Truncate to zero length or create a binary file for update
a+b or ab+	Append; open or create binary file for update

For text files, an additional 't' character can be explicitly added, for example, instead of 'r,' you can specify 'rt.'

On Microsoft platforms, the following directive should be used:

```
#define _CRT_SECURE_NO_WARNINGS
```

10.4.6. Closing a file

Is done by calling the function

```
int fclose(FILE* fp);
```

and has the effect of closing the previously opened stream to the respective file.

The closing operation should only be performed in a suitable context (after all read/write processes have been completed) because otherwise, data loss from the file can occur, the file can be corrupted, or even lost. After closing a file, its control block is released, making it available for reuse

The Visual Studio platform and Visual C++1y/2z compilers have introduced another syntax for the file opening function, `fopen_s();`

```
errno_t  fopen_s( FILE**  pFile,  const char *filename,  const char *mode
);
```

- `pFile` – It's a pointer to the file pointer associated with the opened file.
- `filename` – file name
- `mode` – The type of access permitted that also allows for the consideration of Unicode files:

  `fopen_s(&fp, "newfile.txt", "w+, ccs=UNICODE");`
- `errno`, error flag (int); if it's 0, it's okay, if not, it means there was an error in opening.

Writing to a file is done through the following functions in the standard way.:

`int fputc(int c, FILE* stream);`

- It writes the character 'c' to the file identified by the 'stream' pointer. The function returns the character 'c' in case of success or EOF in case of failure.

`int fputs(const char* s, FILE* stream);`

- It writes the string pointed to by 's' to the file identified by the 'stream.' It returns a positive value in case of success or EOF in case of failure.

`int fprintf(FILE* stream, const char* format[, argument, ...]);`

- It has parameters similar to `printf()`, with the only difference being the appearance of the 'stream' pointer to the file where the writing will take place. It returns the number of bytes written, or in case of failure, EOF. The desired formatting characters, such as '\n,' should be used in the format specifier.

Reading from a file is supported by the following standard functions:

`int fgetc(FILE* stream);`

- It reads a character from the current position in the file indicated by the 'stream.' In case of success, it returns the read character; otherwise, EOF.

`char* fgets(char* s, int n, FILE* stream);`

- It deposits in the string 's' a string of length 'n' read from the file identified by 'stream.' It returns the string in case of success or NULL in case of premature end of file or an error.

`int fscanf(FILE* stream, const char* format[, address, ...]);`

- It reads formatted data from the file identified by 'stream,' analogous to the `scanf()` function.

The Microsoft Visual Studio platform and VC++1y/2z compilers offer the function fscanf_s() as a security/optimization mechanism within the platform. Using the directive:

```
#define _CRT_SECURE_NO_WARNINGS
```

you can use the existing standard functions in C/C++ with files

10.4.7. Functions feof() and ferror()

```
int feof(FILE* fp);
```

It checks in the FILE structure associated with the file at the opening for the _F_EOF flag to determine if the end of the file was reached in the previous access. It returns a non-zero value if the end of the file has been reached or zero otherwise.

```
int ferror(FILE* fp);
```

It returns a positive value if an error occurred during read/write or 0 in case of a successful operation.

10.4.8. Binary streams

A binary stream is a sequence of bytes that is in a one-to-one correspondence with those from the external equipment. In binary-level processing, these byte sequences are padded or segmented to bring the data packet to a standard length required by the communication protocol (for example, a disk can have 512 bytes/sector or an IP frame can carry 1500 bytes of payload), but these operations are transparent to the user. In the case of binary streams, there are no data translations (such as 0A <--> 0D 0A, as in text streams).

10.4.9. Working with files in binary mode

In the case of binary files, the information stored inside them is not human-readable. If someone attempts to open a binary file with a text editor (e.g., Notepad), the 'characters' displayed on the screen appear as a mixture of nonsensical alphanumeric characters, including semigraphic characters (ASCII code > 7Fh), and control characters.

The reading/writing of binary files is done with the following two functions, declared in `<stdio.h>`:

```
size_t  fread(void* ptr, size_t size, size_t n, FILE* stream);
size_t  fwrite(const void* ptr, size_t size, size_t n, FILE* stream);
```

`ptr` - a pointer to a memory buffer that will receive data from/to the file,

`size` - the size in bytes of an item of the type that will be read/written from/to the file. An item in this context means a packet of bytes, of a size that depends on the application. It could be, for example, an element of an array of structures, called a record (in the case of a database), a frame (in the case of remote data transmission), etc.

`n` - the number of items to be read/written

`stream` - specifies the file previously opened, it is the 'file-handle' returned by `fopen()`.

Opening and closing binary files is done with the same functions as text files, using `fopen()/fopen_s()` and `fclose()`.

10.4.10. Functions specific to random access.

Access to a recorded byte can be sequential (as in a 'tape drive,' with one-dimensional organization) or direct, known as random access (for example, through cylinder coordinates, magnetic head, track, sector, etc.). For the file system, the content of the file is transparent. However, the programmer will use different functions to access the content of text files or binary files.

```
long int ftell(FILE* fp);
```

- It allows reading the position indicator, which contains the offset of the first byte for the next access. The first data byte is at offset zero. The position indicator is updated automatically after each access. The function returns the current position in the file specified by the 'fp' pointer in case of success, or the value -1L (the value of EOF, -1, sign-extended to 4 bytes in C2), in case of failure.

```
int fgetpos(FILE* fp, long int* poz);
```

- It reads the value of the current position and writes it to the variable 'pos.' It returns the value 0 in case of success, or a non-zero value in case of an error, setting the global variable 'errno' to a positive value, which can be interpreted using the 'perror()' function.

```
int fsetpos(FILE* fp, const long int* poz);
```

- It assigns the value of the variable '`poz`' to the position indicator associated with the file indicated by '`fp`' In other words, it positions the cursor in the file. It returns the value 0 in case of success, similar to `fgetpos()`.

```
int fseek(FILE* fp, long nr_oct, int origin);
```

- It moves the position indicator from the '`origin`' position by a number of bytes '`nr_oct`' called '`offset`' (a signed number). It returns the value 0 in case of success or a non-zero value in case of failure. The reference position 'origin' can have three values: 0 = SEEK_SET: the beginning of the file; 1 = SEEK_CUR: the current cursor position; 2 = SEEK_END: the end of the file; where SEEK_SET, etc., are symbolic language constants defined in <stdio.h>.

```
void rewind(FILE *fp);
```

- Enables positioning the position indicator at the beginning of the file. It also clears the error indicator associated with the file.

10.4.11. Other file-related functions

```
FILE * freopen(const char *filename, const char *mode, FILE *stream);
```

Associates a new generic file with the open source stream (for example, redirects `stdout` to `stdprn` or to a file on disk).

```
int rename(const char *oldname, const char *newname);
```

Renames a file through the program.

```
void clearerr(FILE *fp);
```

Clears the error and end-of-file indicators.

10.5. Exemple/Examples

10.5.1. Ex. 1

```
/* Copying a file character by character. */

#include <stdio.h>
#include <stdlib.h>
// fopen_s – starting from VS 2010

int main() {
    FILE *fps, *fpd;
    char c;
    char src_file[] = "file1.txt", dest_file[] = "file2.txt"; //current directory
    errno_t err;

    // create the source file programmatically
    err = fopen_s(&fps, src_file, "w");
    if (err != 0) {
        puts("Error creating the source file!");
        exit(1);
    }
    fprintf(fps, "ttt");
    fclose(fps);

    err = fopen_s(&fps, src_file, "r");
    if (err != 0) {
     // file1.txt will be in the directory where the *.cpp source file is located
        printf("Error opening the source file %s\n", src_file);
        exit(1);
    }

    // open the destination file
    err = fopen_s(&fpd, dest_file, "w");
    if (err != 0) {
        puts("Error opening the destination file!");
        exit(1);
    }
    while ((c = fgetc(fps)) != EOF) {
        fputc(c, fpd); // write characters to the destination file
        putc(c, stdout); // characters being copied are displayed in the terminal
    }
    // close files
    fclose(fpd);
    fclose(fps);
    printf("\nCopy completed.\n\The destination file, %s, is saved in the current
directory.\n", dest_file);
    return 0;
} // main
```

10.5.2. Ex. 2.

```
/* The standard C/C++ version (uses fopen() instead of fopen_s()) that is
compatible with other C/C++ development environments, including compilers like gcc,
clang, etc. */

#define _CRT_SECURE_NO_WARNINGS // for Microsoft
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fps, *fpd;
    char c;
    char src_file[] = "file1.txt", dest_file[] = "file2.txt";
    if ((fps = fopen(src_file, "r")) == NULL) {
        // file1.txt is in the directory where the *.cpp source file is located.
        // It can be created in a VC++ project with Source Files > Add > New Item >
Utility > Text File
        puts("Error opening the source file!");
        exit(1);
    }
    // open the destination file
    if ((fpd = fopen(dest_file, "w")) == NULL) {
        puts("Error opening the destination file!");
        exit(1);
    }

    while ((c = getc(fps)) != EOF) // read characters from the source file
        putc(c, fpd); // write characters to the destination file

    // close files
    fclose(fpd);
    fclose(fps);
    puts("Copy completed in file2.txt");
    return 0;
} // main
```

10.5.3. Ex. 3

```
/* Reading the file in an infinite loop with an exit condition when reaching the
end of the file. The EOF flag in the pointed structure of fp is updated only after
an attempted access, for example, with fgetc(). */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE* fp;
    int c;
    fp = fopen("file.txt", "r");
    //file.txt is created in advance in the same directory as the *.cpp source
file.
    if (fp == NULL) {
        perror("Error in opening 'file.txt'");
        exit(1);
```

```
    }
    while (1) {
        c = fgetc(fp);
        if (feof(fp))
            break;
        printf("%c", c);
    }
    fclose(fp);
    return 0;
} // main
```

10.5.4. Ex. 4

```
/* Copying a file line by line. Absolute and relative paths */

#include <stdio.h>
#include <stdlib.h>
#define DIML 81

int main() {
    FILE* fps, *fpd;
    errno_t err;
    char buf[DIML], *p;
    // Any OS and Windows OS
    char src_file[] = "E:/CPP/Source.cpp", dest_file[] = "E\\CPP\\file3.txt";
    // The appearance of the new file will be monitored on drive E:, directory CPP

    // char src_file[] = "Source.cpp", dest_file[] = "file3.txt";
    // Management will be done in the current directory

    // open source file
    if ((err = fopen_s(&fps, src_file, "r")) != 0) {
        puts("Error opening the source file!");
        exit(1);
    }

    // open destination file
    if ((err = fopen_s(&fpd, dest_file, "w")) != 0) {
        puts("Error opening the destination file!");
        exit(1);
    }

    // read data from the source file
    while ((p = fgets(buf, DIML, fps)) != NULL)
        fputs(buf, fpd); // write data to the destination

    // close files
    fclose(fpd);
    fclose(fps);
    puts("Copy completed");
    return 0;
}//main
```

10.5.5. Ex. 5

```
/* Example of fprintf() - "Write formatted data to stream" and fscanf_s() with
_countof(). */

#include <stdio.h>
#include <stdlib.h>
#define DIML 81

int main() {
    char str[DIML];
    float f;
    FILE* pFile;
    int err;
    err = fopen_s(&pFile, "myfile.txt", "w");
    if (err != 0) {
        perror("\nError creating the file 'myfile.txt'");
        exit(1);
    }
    fprintf(pFile, "%f %s", 3.1416f, "PI-Omega");
    fclose(pFile);

    err = fopen_s(&pFile, "myfile.txt", "r");

    if (err == 0) {
        fscanf_s(pFile, "%f", &f);
        fscanf_s(pFile, "%s", str, (unsigned int)_countof(str));
        fclose(pFile);
        printf("I have read: %g and %s \n", f, str);
        return 0;
    }
    else {
        puts("\nError opening the file 'myfile.txt'");
        exit(1);
    }
} // main
```

10.5.6. Ex. 6

```
/* An application that reads integer values from the 'test.txt' file, created
'manually,' displays the square root extracted from the positive ones, and adds the
number of positive values found in the file at the end of the file. */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    FILE* fp;
    int x, count = 0;
    fp = fopen("test.txt", "r");
    if (!fp) {
        perror("\n Error opening the file...");
        exit(1);
    }
```

```
    printf("\n Extracting square roots of positive values read from the file:\n");
    while (fscanf(fp, "%d", &x) != EOF)
        if (x > 0) {
            count++;
            printf("Read value = %d, square root = %.2lf\n", x, sqrt((float)x));
        }
    fclose(fp);
    fp = fopen("test.txt", "a");
    if (!fp) {
        printf("\n Error opening the file for appending...\n");
        exit(1);
    }
    fprintf(fp, " %d", count);
    fclose(fp);
    printf("\n Closed with append");
    return 0;
}//main
```

10.5.7. Ex. 7

```
/* A program that generates random numbers, writes them to a binary file, then
reads them from the file and displays them on the screen. The filename and the
number of values to generate are entered from the keyboard.*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h> // Library functions for real-time clock

FILE* generate(); // This function creates a binary file, writes numbers to it, and
leaves the file open. It returns a file handle.

int main() {
    FILE* f1;
    int buf1; // An integer buffer for displaying with fread()
    f1 = generate(); /* When returning from the function, the file position
indicator is on the next byte to write. Here, fclose() will place an EOF if nothing
more is written to this file. If the file has N bytes, the offset of this byte = N
(the first one being at offset zero). So ftell() will actually return the length of
the file. */
    printf("Length of the binary file = %d[bytes]\n", ftell(f1));

    // We are about to display the file's content
    fseek(f1, 0L, SEEK_SET); // Bring the file pointer to the first byte.
    printf("The values written in the file are: ");
    while (fread(&buf1, sizeof(int), 1, f1) == 1)
        printf("%4d", buf1);

    fclose(f1);
    puts("\nThe file is closed, press a key to finish.\n");
    return 0;
}//main

FILE* generate()
{ // Return the identifier for the file created in the function
    int n, i;
    char sp[10]; // Character array, the name of the file to create
```

```
    int buf2; // Workspace for a single integer to be written to the file
    FILE* pf; // Declare fp as a pointer to a control structure of FILE type
    int OFFSET = 1;
    int RANGE_MAX = 100;
    time_t t;

    printf("Enter the binary file name (max 10 chars): ");
    gets_s(sp, _countof(sp));
    printf("How many values? : ");
    scanf_s("%d", &n);
    errno_t err;
    err = fopen_s(&pf, sp, "w+b");
    // Access mode w+b: if a file with this name already exists, it will be
overwritten
    if (err != 0) {
        puts("\n Could not open the file.");
        exit(1);
    }
    else {
        srand((unsigned)time(&t));
        for (i = 0; i < n; i++) {
            buf2 = (int)(((double)rand() / RAND_MAX) * RANGE_MAX + OFFSET);
            fwrite(&buf2, sizeof(int), 1, pf);
        }
    }
    printf("...done, all numbers have been written to %s, but the file is still
open, and the file pointer is at the end of the file\n", sp);
    return pf;
}// generate
```

10.5.8. Ex. 8

```
/* To test this program, we can use the binary file created in the previous
example. The program determines the length of a binary file using random access
functions, fseek(), and ftell(). The filename is taken by the program at runtime */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DIML 81

int main() {
    char c[DIML]; // The name/path of the file to be read, max. 80 characters
    FILE* fp; // Declaration of a pointer to a control structure of the predefined
FILE type. The stream specified by fp will be connected to our file via fopen()
    printf("Enter the file name: ");
    gets_s(c); // Reads a string from the keyboard and stores it in the array
variable c, automatically adding the string terminator '\0'

    errno_t err;
    if ((err = fopen_s(&fp, c, "rb")) != 0) { // Open a binary file for reading
        printf("\n The file cannot be opened!");
        exit(1);
    }
    // fseek() allows random (direct) access to any byte in the file
    fseek(fp, 0L, SEEK_END); // We positioned ourselves at the end of the file
```

```
    // ftell() returns the offset of the current position from the beginning of the
file in bytes
    printf("The file %s has %d bytes,\nplease verify with your OS file manager.\n",
c, ftell(fp));
    fclose(fp); // Close the file
    return 0;
} //main
```

10.5.9. Ex. 9

```
/* The program concatenates 'n' binary files into a new destination file. The
filenames are taken by the program, similar to an archiving program. To test this
program, we use several binary files created with Example 6 */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#define DIML 81

typedef unsigned char DATE; // User-defined data type

int main() {
    DATE Buff; // Temporary storage for the byte to be transferred
    int nr; // Number of files to concatenate
    int cont_planif, cont_realiz; // Counters for source files
    char nmf_s[DIML], nmf_d[DIML]; // Names of source and destination files
    FILE* fs, * fd; // Pointers associated with files
    // Destination file
    printf("Enter the destination file name: ");
    scanf("%s", nmf_d);
    errno_t err;
    if ((err = fopen_s(&fd, nmf_d, "wb")) != 0) {
        // Open the destination file for writing in binary mode
        printf("\n The file %s cannot be written!", nmf_d);
        exit(1);
    } // The destination file now appears in File Manager but with zero length
    printf("Enter the number of binary files to concatenate: ");
    fscanf(stdin, "%d", &nr);
    for (cont_planif = 0, cont_realiz = 0; cont_planif < nr; cont_planif++) {
        printf("Enter the name of source file #%d: ", cont_planif + 1);
        scanf("%s", nmf_s);
        errno_t err; // Open the current source file in binary mode for reading
        if ((err = fopen_s(&fs, nmf_s, "rb")) != 0) {
            printf(":-( The file %s cannot be opened\n", nmf_s);
            continue; // Now, cont_realiz is not incremented
        } // if()
        // Take one byte at a time from the source file and append it to the
destination file
        while (1) {
            /* feof() returns nonzero only if a previous operation, such as fread()
or fseek(), has set the EOF flag in the FILE structure, a control structure of the
stream attached to this file. */
            fread(&Buff, sizeof(DATE), 1, fs);
            // After fread(), the Position indicator is placed on the next byte to
be read.
```

```
            if (feof(fs)) // Now the byte at the current position is EOF
                break; // Exit the infinite loop
            fwrite(&Buff, sizeof(DATE), 1, fd);
        } // while()
        fclose(fs); // Close the current source file
        cont_realiz++; // Success
        printf("Current position in %s = %d\n", nmf_d, ftell(fd));
        // Display the length of the destination file as a progress indicator.
    } // for()
    fclose(fd); // Finally, close the destination file
    printf("Concatenated %d binary files into %s\n", cont_realiz, nmf_d);
    return 0;
} //main
```

10.5.10. Ex. 10

```
/* Writing and reading elements of a structure to/from a file. First, you should
create a CPP directory on a valid disk on your computer. */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

const int char_dim = 20;
const int np = 3; // np = number of people

typedef struct {
    char last_name[char_dim];
    char first_name[char_dim];
    int age;
} PERSON;

int main() {
    FILE* fp;
    PERSON table[np]; // Array of structures (Static allocation for 3 people)
    PERSON* p_person; // Pointer of the structure type
    PERSON x; // Structure for handling data read from the file
    char file_name[] = "data_file.bin";
    p_person = table;

    for (int i = 0; i < np; i++) {
        printf("\nEnter the last name of person %c: ", char('a' + i));
        scanf("%s", table[i].last_name);
        printf("Enter the first name of person %c: ", char('a' + i));
        scanf("%s", (p_person + i)->first_name);
        printf("Enter the age of person %c: ", char('a' + i));
        scanf("%d", &table[i].age);
    }

    errno_t err;
    err = fopen_s(&fp, file_name, "wb"); // Create and open the file
    if (err != 0) {
        printf("\nError creating the file...\n");
        exit(1);
    }
```

```
    for (int i = 0; i < np; i++)
        fwrite(&table[i], sizeof(PERSON), 1, fp); // Write data to the file
    fclose(fp);

    err = fopen_s(&fp, file_name, "rb"); // Reopen the file for reading
    if (err != 0) {
        printf("\nError opening the file...\n");
        exit(1);
    }
    p_person = &x;

    while (fread(p_person, sizeof(PERSON), 1, fp)) {
        printf("\nLast Name: %s, First Name: %s, Age: %d\n", x.last_name,
x.first_name, x.age);
    }
    fclose(fp);

    return 0;
} // main
```

10.5.11. Ex. 11

```
/* Writing and reading elements of a structure to/from a file, followed by sorting
based on multiple fields. */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const int char_dim = 20;
const int np = 3; // np = number of people

int compare_name_first_name_age(const void* a, const void* b);

struct Person {
    char last_name[char_dim];
    char first_name[char_dim];
    int age;
};

int main() {
    int i;
    FILE *fp, *fps;
    Person table[np]; // Array of structures (Static allocation for np people)
    Person* p_person; // Pointer of the structure type
    Person x; // Structure for handling data from the file
    char file_name[] = "data_file.bin"; // Current directory
    char sorted_file_name[] = "sorted_data_file.bin";
    p_person = table;

    printf("\nEnter data for %d people", np);
    for (i = 0; i < np; i++) {
        printf("\nEnter the last name of person %c: ", char('a' + i));
        scanf("%s", table[i].last_name);
```

```
        printf("Enter the first name of person %c: ", char('a' + i));
        scanf("%s", (p_person + i)->first_name);
        printf("Enter the age of person %c: ", char('a' + i));
        scanf("%d", &table[i].age);
    }

    errno_t err;
    err = fopen_s(&fp, file_name, "wb"); // Create and open data_file.bin
    if (err != 0) {
        printf("\nError creating the file...\n");
        exit(1);
    }

    for (i = 0; i < np; i++)
        fwrite(&table[i], sizeof(Person), 1, fp); // One by one
    fclose(fp);

    err = fopen_s(&fp, file_name, "rb"); // Reopen the file for reading
    if (err != 0) {
        printf("\nError opening the file...\n");
        exit(1);
    }
    p_person = &x;
    printf("\nThe file with the initial data is:\n");
    while (fread(p_person, sizeof(Person), 1, fp)) {
        printf("\nLast Name: %s, First Name: %s, Age: %d\n", x.last_name,
x.first_name, x.age);
    }
    fclose(fp);

    err = fopen_s(&fps, sorted_file_name, "wb"); // sorted_data_file.bin
    if (err != 0) {
        printf("\nError creating the sorted file...\n");
        exit(1);
    }
    printf("\n\nThe file with sorted data is:\n");
    qsort(table, np, sizeof(Person), compare_name_first_name_age);
    fwrite(table, sizeof(Person), np, fps);
    fclose(fps);

    err = fopen_s(&fps, sorted_file_name, "rb");
    if (err != 0) {
        printf("\nError opening the sorted file...\n");
        exit(1);
    }
    while (fread(p_person, sizeof(Person), 1, fp)) {
        printf("\nLast Name: %s, First Name: %s, Age: %d\n", p_person->last_name,
p_person->first_name, p_person->age);
    }
    fclose(fps);
} //main

// Sorting by last name, first name, and age
int compare_name_first_name_age(const void* a, const void* b) {
    int last_name_diff, first_name_diff;
    Person* pa = (Person*)a;
```

```
    Person* pb = (Person*)b;
    if ((last_name_diff = strcmp(pa->last_name, pb->last_name)) == 0)
        if ((first_name_diff = strcmp(pa->first_name, pb->first_name)) == 0)
            return (pa->age - pb->age);
        else
            return first_name_diff;
    return last_name_diff;
} // compare_name_first_name_age
```

10.6. Întrebări

1. Ce înțelegeți printr-un stream de tip text?
2. Ce fluxuri standard se deschid automat la lansarea în execuție a unui program C/C++?
3. Ce funcții utilizați pentru scrierea/citirea unui fișier text, caracter cu caracter?
4. Ce funcții utilizați pentru scrierea/citirea unor înregistrări de tip float dintr-un fișier text?
5. Ce funcție utilizați pentru citirea unui fișier text linie cu linie?
6. Ce este un stream binar?
7. Care sunt funcțiile utilizate pentru citirea/scrierea fișierelor binare?
8. Cum se realizează accesul aleator la fișiere?

10.7. Questions

1. What do you understand by a text stream?
2. Which standard streams are automatically opened when a C/C++ program is executed?
3. Which functions do you use for reading/writing a text file character by character?
4. Which functions do you use for reading/writing float records from/to a text file?
5. Which function do you use for reading a text file line by line?
6. What is a binary stream?
7. Which functions are used for reading/writing binary files?
8. How is random access to files achieved?

10.8. Lucru individual

1. Să se scrie un program care citește și apoi afișează date întregi preluate dintr-un fișier text al cărui nume este citit de la consolă. Creați în prealabil fișierul prin program.
2. Să se scrie un program care citește dintr-un fișier text 10 numere întregi (generat în prealabil prin program sau extern). Să se scrie funcțiile care:
   - aranjează crescător/descrescător șirul și afișează rezultatul;
   - numără câte elemente sunt pare și afișează rezultatul.

   Adăugați în fișierul original noile rezultate obținute.
3. Scrieți un program care citește de la consolă n numere întregi pe care le scrie într-un fișier text cu numele citit de la tastatură. Citiți apoi numerele din fișier, determinați media lor aritmetică, pe care o adăugați la sfârșitul fișierului și o afișați și pe ecran.
4. Scrieți un program C/C++ care citește de la tastatură un caracter, apoi scrie acest caracter într-un fișier text pe n linii, câte n caractere pe fiecare linie, n citit de la consolă.
5. Să se scrie o aplicație C/C++ care citește un fișier text linie cu linie și îl afișează pe ecran. Se va folosi un fișier existent din sistem sau se va genera în prealabil unul prin program.
6. Scrieți un program C/C++ care citește de la tastatură valori reale în format float, cu confirmare. Valorile citite vor fi scrise într-un fișier text cu numele citit din linia de comandă. Citiți apoi fișierul și afișați valorile mai mari decât o valoare dată, citită de la tastatură.

7. Scrieți o aplicație C/C++ care citește caracter cu caracter un fișier text și convertește primul caracter al fiecărui cuvânt în majusculă.
8. Scrieți un program care citește valori reale dintr-un fișier creat în prealabil și scrie într-un alt fișier partea întreagă a numerelor pozitive citite.
9. Să se scrie o aplicație care:
   - citește de la consolă un număr întreg n;
   - citește de la consolă, cu o funcție, "n" numere reale, într-un tablou unidimensional, alocat dinamic în memorie;
   - scrie aceste valori din tablou într-un fișier binar, al cărui nume este citit tot de la consolă;
   - citește apoi conținutul fișierului și afișează numerele din 3 în 3 poziții, folosind funcții specifice accesului aleator la fișiere.
10. Să se scrie o aplicație care:
    - definește o structură numită Student, cu câmpurile numele, prenumele, grupa, media;
    - citește de la consolă un număr întreg n (numărul studenților)
    - pentru fiecare înregistrare de tip Student, citește cu o funcție datele aferente și le scrie într-un fișier, cu numele preluat de la consolă;
    - citește conținutul fișierului și afișează studenții ce au media mai mare decât o valoare citită de la consolă.
11. Să se scrie o aplicație care:
    - citește de la consolă un număr întreg n
    - citește de la consolă, cu o funcție, n numere întregi, memorându-le într-un tablou unidimensional, alocat dinamic
    - scrie valorile din acest tablou într-un fișier binar, al cărui nume este citit de la consolă;
    - citește conținutul fișierului și afișează conținutul și offsetul pozițiilor pe care s-au găsit numere pare.
12. Să se scrie un program, care:
    - preia din linia de comandă două nume de fișiere.
    - citește prin program 8 numere întregi, pe care le scrie în primul fișier, în mod binar.
    - citește înapoi valorile din acest fișier și calculează media aritmetică a numerelor mai mari decât 4
    - scrie rezultatul în al doilea fișier, sub forma: Media aritmetica a numerelor…, este… unde în locul punctelor de suspensie va scrie valorile a căror medie a fost calculată, respectiv valoarea mediei, cu o precizie de 2 zecimale.
13. Definiți o structura Prajitura, cu câmpurile: nume, nr_bucati, preț. Citiți de la tastatură datele pentru un număr n de prăjituri și salvați aceste date într-un fișier binar. Citiți apoi înregistrările din fișier și afișați toate informațiile despre prăjitura cea mai ieftină.
14. Scrieți într-un fișier binar câteva numere întregi, citite de la tastatură. Citiți apoi numerele de pe poziții impare și afișați pentru fiecare, dacă este valoare pară sau impară.

10.9. Individual work

1. Write a program that displays the integer elements read from a text file. The filename is entered from the keyboard. The file has to be created in the program in advance.
2. Write a program that reads from a text file 10 integer numbers. The file has to be previously created by program or externally by using the operating system's facilities. Write the functions that:
   - order the integers array in ascending/descending order and displays the result
   - count the number of even numbers in the array and display the result

   Write the generated results into the original file.

3. Write a program that reads from the keyboard n integer values and then stores them into a text file. The filename has to be read from the keyboard. Then read the numbers from the file, calculate their average value, display-it and append-it to the end of the text-file.
4. Write a program that reads from the keyboard a single character and an integer value n. Generate a text file that will contain n lines and on each line write the character n times, n being read from the keyboard.
5. Develop a C/C++ application that will display (line by line) the content of a previously created text file then one created by program.
6. Write a C/C++ program that reads from the keyboard (with confirmation) a series of float values. Write the values into a text file that has the name entered from the command line. Read the file's content and display all the values greater than a given number, read from the keyboard.
7. Write a C/C++ application that reads a text file character by character and converts the first letter of each word into its uppercase equivalent.
8. Write a program that reads real values from a previously created file and writes to another file the entire part of the positive numbers read.
9. Write an application that:
   - reads from the keyboard an integer value n;
   - reads from the keyboard (using a function) n floating-point numbers, storing them into a dynamically allocated one-dimensional array;
   - writes out the floating-point values into a binary file, the filename being read from the keyboard;
   - reads back the file's content, displaying the numbers with indexes 0, 3, 6, etc. using the random file-access methods;
10. Write an application that:
    - defines a structure called Student, having fields name, surname, group, average mark ;
    - reads from the keyboard an integer value n;
    - for each Student it reads from the keyboard (in a function) the personal data (all fields),
    - stores the information for all the n students and into a binary file, the filename being read from the keyboard;
    - reads back the file's contents, displaying the data related to the students who have the average mark >= than a specific value given from the keyboard.
11. Write an application that:
    - reads from the keyboard an integer value n;
    - reads from the keyboard, with a function, n integers, storing them into a dynamically allocated one-dimensional array;
    - writes these values into a binary file (the filename is also read from the keyboard);
    - reads the file's content and displays the offset and content of all positions where even numbers are found.
12. Write a C/C++ program that reads from the command line two file names. The program should ask the user to introduce eight integer values from the KBD, saving them into the first file, in binary mode. Read back these values from the file and determine the arithmetical media of the values greater than 4. Write the result into the second file in text mode, using the following format: "The average value of ... is ...". The first space needs to be replaced with the values used for calculating the average, the second with the average value itself, using a two digits precision.
13. Define a structure called Cookie that contains as variables the name, no_of_pieces and price. Read from the keyboard the data for n cookies and save it into a binary file. Read back the file and display the information belonging to the cheapest cookie.

14. Write into a binary file a series of integer numbers read from the keyboard. Read back the numbers stored in the file on odd positions displaying whether they are odd or even numbers.

Bibliografie/References

- Mircea-Florin Vaida, Ligia-Domnica Chiorean, Lenuța Alboaie, Petre Gavril Pop, Cosmin Strileţchi, Kuderna-Iulian Benţa, Programarea în limbajul C/C++ cu elemente C++1y. Programare web C++, Casa Cărții de Stiință, Cluj-Napoca, 2016, pp.336, ISBN 978-606-17-1015-7

- Ligia-Domnica Chiorean, Kuderna-Iulian Benţa, Mircea-Florin Vaida, Petre Gavril Pop, Cosmin Strileţchi, C/C++ - Ghid teoretic si practic, Casa Cărții de Stiință, Cluj-Napoca, 2016, pp. 464, ISBN 978-606-17-1016-4

- Resurse web:

  https://en.cppreference.com/w/
  https://cplusplus.com/reference/


HELIOS
software