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

Algoritmi de bază și programarea orientată pe obiecte folosind C/C++. Elemente practice

Varianta bilingvă

Basic algorithms and object-oriented programming using C/C++. Practical elements.

Bilingual version

Editura UTPRESS
Cluj-Napoca, 2024
ISBN 973-606-737-727-9

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

Algoritmi de bază și programarea orientată pe obiecte folosind C/C++. Elemente practice.

Varianta bilingvă

Basic algorithms and object-oriented programming using C/C++. Practical elements.

Bilingual version

UTPRESS
Cluj-Napoca, 2024
ISBN 978-606-737-727-9

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

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

Pregătire format electronic on-line: Gabriela Groza

ISBN 978-606-737-727-9

Cuvânt înainte

În cadrul cărții au fost tratate cele mai importante concepte necesare pentru integrarea unor elemente specifice legate de funcții și macrofunctii, a recursivității și a algoritmilor de bază folosite în programare precum și algoritmii de căutare și sortare în limbajul C/C++. Se introduc de asemenea elementele de bază ale limbajului de programare C++ orientate către mecanismul obiectual. Sunt introduse gradat noțiunile legate de : clase, obiecte, încapsulare, polimorfism, moștenire, abstractizare, intrări/ ieșiri specifice C++.

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, 2024

Cuprins/Contents

1. Aplicații bazate pe macrofuncții, aserțiuni, funcții inline, funcții cu parametri impliciți, funcții cu număr variabil de parametri, supraîncărcarea funcțiilor.

Macro functions, assertions, inline functions, functions with implicit parameters, functions with a variable number of parameters, overloading functions.

1.1. Obiective

- Înțelegerea noțiunilor legate de funcții macro, funcții inline prin aplicarea lor în practică în programe C/C++,
- Asimilarea modului de lucru cu funcții cu parametri impliciți și cu parametri variabili,
- Mecanismul de folosire a supraîncărcării funcțiilor.

1.2. Objectives

- Understanding the notions related to macro functions, inline functions by applying them in practice in C/C++ programs,
- Assimilating the way of working with functions with default parameters and with variable numbers of parameters,
- Overloading functions using mechanism.

1.3. Breviar teoretic

Funcțiile macro (macrodefiniții cu parametri)

Au următoarea sintaxă:

```
#define nume(p1, p2,...,pn)   text
```

unde:

- p_i sunt parametri formali (fără tip);
- `text` este textul de substituție care va conține parametri formali.

Apelarea unei macrodefiniții cu parametri se face într-un mod similar cu apelarea unei funcții: se scrie numele macro-ului urmat de parametrii efectivi între paranteze, separați prin virgule. Totuși, spre deosebire de funcții, la momentul apelului, macro-ul este expandat în textul definit inițial, cu mențiunea că parametri formali sunt înlocuiți cu parametri efectivi.

Macrodefinițiile cu parametri nu returnează valori.

Macrourile predefinite sunt:

```
__cplusplus, __LINE__, __DATE__, __FILE__, __TIME__, __STDC__.
```

Anularea unei macrodefiniții se face cu directiva:

```
#undef nume
```

Aserțiunile

Aserțiunile, ar trebui folosite doar pentru a verifica condiții care ar trebui din punct de vedere logic să fie imposibil a fi false. Aceste condiții ar trebui să se bazeze numai pe intrările generate de propriul cod. Orice verificări bazate pe intrări externe ar trebui să utilizeze excepții. Aserțiunile în limbajul C/C++ sunt introduse prin macrofuncția `assert(...)`.

Macrofuncția `assert (...)`, definită în fișierul antet `assert.h`, testează valoarea unei expresii.

Dacă valoarea expresiei este 0 (fals), atunci `assert (...)` afișează un mesaj de eroare și apelează funcția `abort()`, definită în antetul `stdlib.h`, pentru a termina executarea programului.

Când aserțiunile nu mai sunt necesare, linia

```
#define NDEBUG
```

este inserată în fișierul programului, mai degrabă decât ștergerea manuală a fiecărei aserțiuni.

Din *C ++ 1y* putem folosi `static_assert` care este aplicată în timpul compilării.

Funcțiile inline

Funcțiile inline sunt funcții pentru care compilatorul substituie apelul funcției cu codul acesteia (textul funcției). Ele sunt specifice limbajului C++.

Sunt definite similar celorlalte funcții, dar cuvântul cheie "inline" se adaugă înaintea definiției.

Funcțiile inline păstrează toate proprietățile funcțiilor obișnuite, cum ar fi verificarea numărului și tipului parametrilor la apel, modul de transfer al parametrilor, precum și domeniul de valabilitate al declarațiilor locale și al parametrilor.

Aceste funcții se folosesc de obicei atunci când codul lor ocupă un număr mic de linii, pentru a optimiza performanța, evitându-se astfel operațiile specifice apelului unei funcții (cum ar fi încărcarea și descărcarea stivei etc.).

Funcțiile cu parametri impliciți

Funcțiile cu parametri impliciți sunt funcții pentru care programatorul a declarat valori implicite pentru unii parametri.

La apel, se poate omite specificarea parametrilor efectivi pentru parametrii formali care au valori implicite, caz în care compilatorul va utiliza automat aceste valori.

Valorile implicite trebuie specificate o singură dată, fie în prototipul funcției, fie în definiția acesteia.

Parametrii cu valori implicite trebuie să fie plasați la sfârșitul listei de parametri.

Funcțiile cu un număr variabil de parametri

Funcțiile cu un număr variabil de parametri sunt utile în situații similare cu cele ale funcțiilor standard `printf()` și `scanf()`.

Aceste funcții au un prototip de forma:

```
tip_returnat nume_functie(tip arg1, ...);
```

Ele pot accesa argumentele (parametrii efectivi) utilizând un set de funcții definite în fișierul antet stdarg.h:

- `void va_start(va_list ap, lastfix);`
  Această funcție inițializează lista `ap`, de tipul `va_list`, cu argumentele corespunzătoare parametrilor variabili. Parametrul `lastfix` este ultimul parametru fix declarat înainte de punctele de suspensie `...`, care indică începutul listei de parametri variabili. Această funcție trebuie să fie apelată înainte de a accesa orice parametru variabil.
- `tip_argument va_arg(va_list ap, tip_argument);`
  Această funcție returnează următorul argument din lista `ap`, având tipul specificat prin `tip_argument`.
- `void va_end(va_list ap);`
  Această funcție curăță lista de parametri furnizată și trebuie apelată după ce toate argumentele au fost procesate. Este necesară pentru a încheia corect utilizarea listei de argumente variabile.

Supraîncărcarea funcțiilor

În C++, este posibil ca funcții diferite să aibă același nume, dar să difere prin semnătură. Acest mecanism se numește supraîncărcarea funcțiilor sau overloading.

Semnătura unei funcții este definită de următoarele elemente: numele funcției, numărul de parametri, tipul parametrilor și ordinea acestora.

Valoarea de retur nu este considerată parte a semnăturii funcției.

1.4. Theoretical brief

Macro functions (macro-definitions with parameters)

They have the following syntax:

```
#define name(p1, p2,...,pn)   text
```

where:

- p_i are formal parameters (no type);
- `text` is the substitution text that will contain formal parameters.

The call of a macro-definition with parameters is analogous to the call of a function, but by expanding: the name of the macro followed by actual parameters in parentheses and separated by commas. Macro-definitions with parameters return no values.

Predefined macro functions are:

```
__cplusplus, __LINE__, __DATE__ , __FILE__, __TIME__ , __STDC__
```

The cancellation of a macro-definition is done with the directive:

```
#undef name
```

Assertions

Assertions should only be used to verify conditions that should logically be impossible to be false. Those conditions should only be based on inputs generated by their own code. Any checks based on external inputs should use exceptions. Assertions in the C/C++ language are introduced by the macro function `assert(...)`.

Macro function `assert(...)`, is defined in header file `assert.h`, and will test the value of an expression.

If the value of the expression is 0 (false), then `assert (...)` will display an error message and will call the `abort()` function (from - `stdlib.h`) to terminate the program execution.

When assertions are no longer needed, the line:

#define NDEBUG

is inserted into the program file, rather than manually deleting each assertion.

Starting with *C ++ 1y* we may use `static_assert` which is applied during compilation.

Inline functions

Inline functions are functions for which the compiler substitutes the function call with its actual code (the text). The are specific to the C++ language.

They are defined similarly to other functions, but the keyword "inline" is added before the function definition.

Inline functions retain all the usual properties of regular functions, such as checking the number and types of parameters at the call, the method of parameter passing, and the scope of local declarations and parameters.

These functions are typically used when their code consists of a small number of lines, in order to optimize performance by avoiding the overhead associated with a function call (such as stack loading and unloading, etc.).

Functions with implicit (default) parameters

Functions with default parameters are functions where the programmer has declared default values for some parameters.

When calling such a function, the actual parameters for the formal parameters that have default values can be omitted, in which case the compiler will automatically use those default values.

Default values must be specified only once, either in the function prototype or in its definition.

Parameters with default values must be placed at the end of the parameter list.

Functions with a variable number of parameters

Functions with a variable number of parameters are useful in situations similar to those of the standard functions `printf()` and `scanf()`.

These functions have a prototype in the following form:

```
return_type function_name(type arg1, ...);
```

They can access the arguments (actual parameters) using a set of functions defined in the `stdarg.h` header file:

- `void va_start(va_list ap, lastfix);`
  This function initializes the `ap` list, of type `va_list`, with the arguments corresponding to the variable parameters. The parameter `lastfix` is the last fixed parameter declared before the ellipsis `...`, which marks the beginning of the list of variable parameters. This function must be called before accessing any variable parameter.
- `type va_arg(va_list ap, type);`
  This function returns the next argument from the `ap` list, having the type specified by `type`.
- `void va_end(va_list ap);`
  This function cleans up the list of parameters and must be called after all arguments have been processed. It is necessasry to properly end the use of the variable argument list.

Function overloading

In C++, it is possible for different functions to have the same name but differ by their signature. This mechanism is called *function overloading*, or simply *overloading*.

The signature of a function is defined by the following elements: the function name, the number of parameters, the types of the parameters, the order of the parameters.

The return value is not considered part of the function's signature.

1.5. Exemple/ Examples

Ex. 1 - Macro and inline functions

A) Macro function with implicit result in the expression:

```
#define MAX2(a,b) ((a)>(b)?(a):(b))
```

Call:

```
cin>>a; cin>>b;
cout<< "\nMax (2 values) = "<< MAX2(a,b);
```

B) Macro function with explicit result stored in parameter `a`:

```
#define MAX2I(a,b) {if(a<b)\
    a=b;\
}
```

Call:

```
cin>>a; cin>>b;
MAX2I(a,b);
cout<< "\n Max (with if) = "<< a;
```

C) Inline function example:

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

inline int max(int a, int b);

int main()
{
    int v1, v2, max_v;
    cout << "Enter 2 int values: ";
    cin >> v1; cin >> v2;
    max_v = max(v1, v2);
    cout << "\n The maximum is= " << max_v << endl;
}//main

inline int max(int a, int b)
{
    if (a > b) return a;
    return b;
}//max
```

Ex. 2 - Assertions, data validation

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

constexpr int DIM = 50;//compile time
const int val = 5;

int main( ) {
    int a;
    char str[DIM];
    printf("Enter an integer value: ");
    scanf("%d", &a);
```

```
    assert(a >= val);
    printf("Integer entered is %d\n", a);
    printf("Enter string: ");
    scanf("%s", str);
    assert(strlen(str)>val);
    printf("String entered is: %s\n", str);
    return 0;
}//main
```

Ex. 3 - Disabled assertions (NDEBUG)

```
#include <iostream>
//assert( ) disabled
#define NDEBUG
#include <cassert>
//static_assert(sizeof(int) == 4, "int must be 4 bytes");//compilation time
using namespace std;

int main( ){
    assert(2 + 2 == 3 + 1);
    cout << "Expression valid... Execution continues\n";
    assert(2 + 2 == 1 + 1);
    cout << "Assert disabled... Execution continues despite invalid expression\n";
}//main
```

Ex. 4 - Functions with default parameters - itoa()

```
/* Illustrates the use of function itoa( ) with a default parameter (here
   radix)- deprecated, itoa is not part of the standard (works on MSVC only) */
#define _CRT_SECURE_NO_WARNINGS
#define _CRT_NONSTDC_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

#define DIM 10
//We declare the itoa() function as having a default parameter numeration base = 10
char * itoa(int value, char * string, int radix=10);

int main( )
{
    int i;
    char sir_baza10[DIM] , sir_baza16[DIM];
    /*We call the function with the parameter radix=10, that is, the integer that
      will be transformed in the string of numbers corresponding to it in base ten
    */
    printf("The integer number is: ");
    scanf("%d", &i);
    //The call was made without specifying the third parameter,
    //so it defaults to 10
    itoa(i, sir_baza10);
    printf("The string corresponding to the integer in base 10 is: %s\n",
           sir_baza10);
    //the call to itoa was made with a third argument having the value 16
    itoa(i, sir_baza16, 16);
    printf("The string corresponding to the integer in base 16 is: %s",
           sir_baza16);
}//main
```

Ex. 5 - Functions with default parameters - _itoa_s()

```
/* Illustrates the use of a function (_itoa_s( )) from Microsoft VC++ no implicit
   parameters from stdlib.h */
#include <stdio.h>
#include <stdlib.h>

constexpr int DIM = 10;
char* _itoa_s(int value, char* string, int length, int radix = 10);
// radix is accepted as a param but must be specified

int main()
{
    int i;
    char sir_baza10[DIM], sir_baza16[DIM];
    printf("The integer number is: ");
    scanf_s("%d", &i);
    /* the call was made with the fourth argument of the function _itoa_s( )
       having value 10 */
    //_itoa_s(i, sir_baza10, _countof(sir_baza10));  //linker error
    _itoa_s(i, sir_baza10, _countof(sir_baza10), 10);
    printf("The string corresponding to the integer in base 10 is: %s\n",
           sir_baza10);
    /* the call was made with the fourth argument of the function _itoa_s( )
       having value 16 */
    _itoa_s(i, sir_baza16, _countof(sir_baza16), 16);
    printf("The string corresponding to the integer in base 16 is: %s",sir_baza16);
}//main
```

Ex. 6 - Function with variable number of arguments (polinom)

```
/* polinom( ) is a function that calculates the value of a polynomial, having as
   first parameter the value of the independent variable and then a variable number
   of parameters corresponding to the coefficients of the polynomial.
   The polinom( ) function will determine the degree of the polynomial from the
   number of parameters. */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdarg.h>
#include <math.h>

double polinom(double x, ...);

int main( )
{
    double x;
    printf("Enter the value of x: ");
    scanf("%lf", &x);
      printf("\nValue of P(x) = 5*x^3+34*x^2+20*x-5 este: %lf",
             polinom(x, -5.0, 20.0, 34.0, 5.0, HUGE_VAL));
      /* HUGE_VAL is a particular value for which the search stops in the
         parameter list, has as value the largest representable real number
         (double)*/
      printf("\n\n");
      printf("\nValue of P(x) = x^5+3*x^4-x^2+2*x+15 este: %lf\n",
             polinom(x, 15.0, 2.0, -1.0, 0.0, 3.0, 1.0, HUGE_VAL));
}//end main
```

```
double polinom(double x, ...) //the order of coefficients is from grade 0 to n
{
    /*Holds the current power being processed. After all powers
      are processed, its value will be equal to the degree of the polynomial*/
    int grad_polinom = 0;
    /*coef will contain the coefficient for the power being processed*/
    double rezultat = 0., coef;
    va_list ap;     // pointer declaration
    va_start(ap, x);// pointer definition
    while ((coef = va_arg(ap, double)) != HUGE_VAL)
    {
        rezultat += coef * pow(x, grad_polinom);
        grad_polinom++;
    }//end while

    va_end(ap);

    printf("\nThe polinom degree is: %d\n", grad_polinom - 1);
    return rezultat;
}//end_polinom
```

Ex. 7 - Function overloading, call by value and by reference

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

int abs(int n); //by value
int abs(int &n, int b); //by reference

int main() {
    int a = -8;
    cout << "\nAbs. by value =" << abs(a);
    cout << "\nAbs. by reference =" << abs(a, 2);
}//main

int abs(int n) {
    return ((n < 0) ? (-n) : n);
}//abs

int abs(int &n, int b) {
    return ((n < 0) ? (-n) : n);
}//abs by reference
```

1.6. Lucru individual

1. Definiți funcții macro `MAXi (unde i=2,3)` care determină și afișează maximul dintre 2, respectiv 3 numere introduse de la tastatură. Folosiți variante diferite (operator condițional, instrucțiuni `if`*, etc.*).
2. Definiți o funcție `inline min( )` care determină și afișează minimul dintre 2 intregi și alta pentru minimul dintre 3 numere întregi, valorile fiind introduse de la tastatură. Considerați supraîncărcarea funcțiilor.
3. Considerați o structură de date `Student`, care conține un câmp de tip șir de caractere (maxim 30) pentru `nume_prenume` și un alt câmp `nota` de tip `int`. Definiți un obiect de tip `Student` la care datele vor fi citite de la tastatură. Validați folosind aserțiuni ca `nume_prenume` să aibă cel puțin 5 caractere, iar `nota` să fie >=5 și <=10. Afișați câmpurile obiectului în caz de introducere corectă.
4. Considerați o funcție cu 3 parametrii toți impliciți (`int, float, double`) care returnează produsul acestor valori. Apelați funcția considerând mai multe variante de apel concrete (fără parametri, 1 parametru, 2 parametri, 3 parametri).
5. Folosind supraîncărcarea funcțiilor, definiți trei funcții cu același nume, dar cu tipuri diferite de parametri (`int, int *, int &`), care returnează radicalul unei valori intregi, daca e pozitiva. Analizați cazul transmiterii parametrilor prin valoare și prin referință.
6. Determinați elementul minim dintr-un tablou unidimensional de 10 numere flotante (introduse de la tastatură/inițializate) folosind funcții cu un număr variabil de parametri. Se vor considera primele 7 valori din șir, apoi următoarele 3, după care se afișează minimul din cele 10 folosind valorile determinate anterior.
7. Scrieți un program care face o codare simplă prin adaugarea la codul ASCII al caracterului a unei valori întregi, *n=3*, folosind macro funcții. Exemplu: `'a'` devine în urma codării `'d'`.
8. Să se scrie un program care afișează numele programului, data și ora compilării și numărul de linii pe care îl are acest program, folosind macrofuncții standard.
9. Realizați o aplicație C/C++ care aplică un cod binar (mască) fiecărui element al unui șir printr-o funcție de codare și invers îl decodează într-o funcție de decodare, folosind funcții macro (cu sau-exclusiv).
   Exemplu: fie caracterul `'a'`, codul mască `11001010`:
   rezultatul codării ar fi:
   `'a'` -> 97 (0x61) ASCII -> 01100001 SAU EXCLUSIV logic pe biți
   codul binar (masca) 11001010

   10101011
   decodarea se face:
   rezultatul codării ->10101011 SAU EXCLUSIV logic pe biți
   codul binar 11001010

   01100001 codul ASCII al lui `'a'`

1.7. Individual work

1. Define macro functions `MAXi (i=2,3)` that determines and display the maximum between 2 and other, among 3 numbers introduced from the KB. Use different options (conditional operator, `if` statement, etc.).
2. Define an `inline function min()` that determines and display the minimum among 2 and other among 3 numbers introduced from the KB. Consider overloading functions.
3. Consider a `Student` data structure, which contains a string field (maximum 30 characters) for `name_surname` and another `mark` field of type `int`. Define a `Student` object where the data will be read from the keyboard. Validate width assertions that `name_surname` has at least 5 characters and the `mark` should be> = 5 and <=10. Display the object fields if entered correctly.
4. Consider a function with 3 implicit parameters (all) (int, float, double) that returns the product of the values. Call that function with different variants for effective parameters(no pram, 1 param, 2 params, 3 params).
5. Using functions overloading define 3 functions with the same name but with different prams type (`int or, int*, or int&`) that will return the square root of the *int* value if it is positive. Analyze the calling mechanism by value and reference.
6. Determine the minimum of a `10 float` numbers from a one dimensional array (implicit values or from the KB) using a function with a variable number of parameters. The first 7 values will be considered initially, next the last 3, and at the end these 2 values.
7. Write a program that performs a simple coding operation by increasing with a value, as *n=3*, the value of the ASCII code of a character using macro functions. For example, `'a'` becomes `'d'`, etc.
8. Write a program that displays the name of the program, the compilation date and time and the number of code lines included in the program using standard macrofunctions.
9. Implement a C/C++ application that applies (using a macro function with XOR) a binary mask to each element located in an array of characters. Define the decoding function, too. Example: considering the character `a` and the mask code `11001010`:
   a. the coding result will be obtained as it follows:

      `'a'` -> 97 (0x61) ASCII ->01100001 XOR

      Binary code (mask) 11001010

      10101011

   b. the decoding process:

      the coded result -> 10101011 XOR

      binary code mask 11001010

      01100001 ⇔the ASCII code of `'a'`

2. Funcții recursive. Elemente de programare funcțională.

Recursive functions. Functional programming elements.

2.1. Obiective

- Abilitatea de a proiecta algoritmi recursivi simpli și de a-i implementa în limbajul C/C++;
- Înțelegerea mecanismului de încărcare și eliberare a stivei la apelul funcțiilor recursive.

2.2. Objectives

- Ability to design simple recursive algorithms and implement them in the C/C++ language;
- Understanding the stack loading and unloading mechanism when recursive functions are called.

2.3. Breviar teoretic

Funcții recursive

O funcție este recursivă dacă executarea ei implică cel puțin un apel către ea însăși (autoapel). Autoapelarea se poate realiza:

- direct, prin ea însăși (*recursivitate directă*);
- prin intermediul altor funcții care se apelează circular (*recursivitate indirectă*).

Caracteristici:

- Autoapelul se face pentru o dimensiune mai mică a problemei. Există o dimensiune a problemei pentru care nu se mai face autoapel, ci se revine din funcție (eventual, cu returnarea unui rezultat);
- Pentru evitarea situației în care funcția se apelează pe ea însăși la infinit, este obligatoriu ca autoapelul să fie legat de îndeplinirea unei condiții care să asigure oprirea din acel ciclu de autoapeluri;
- Orice apel de funcție provoacă în limbajul C/C++ memorarea pe stivă a parametrilor acelei funcții și a adresei de revenire la prima instrucțiune de după apel. La apeluri repetate, spațiul ocupat pe stivă poate crește în afara limitelor admise. De aceea se urmărește ca funcțiile recursive să aibă un număr minim de parametri, în acest caz recomandându-se a folosi parametrii globali dacă e posibil.

La apelul recursiv al unei funcții se creează o nouă instanță de calcul pentru acea funcție. La fiecare apel pe stiva locală funcției se pun un nou set de variabile locale (inclusiv parametri formali), cu aceleași nume, dar diferite ca zonă de memorie și cu valori diferite la fiecare apel. În fiecare moment e accesibilă doar variabila din vârful stivei (LIFO).

Utilizarea tehnicilor recursive nu conduce de obicei la soluții optime, fiind recomandabilă analiza eficienței soluției iterative (nerecursive) și a celei recursive, alegându-se soluția cea mai avantajoasă. Soluția recursivă duce însă la soluții mai elegante și mai ușor de urmărit.

Exemple de algoritmi recursivi simpli:

- calculul factorialului;
- calculul șirului lui Fibonacci;
- funcția Ackermann;
- cel mai mare divizor comun;
- calculul combinărilor;
- calculul sumei cifrelor unui numar, etc.

Opțional - Programarea funcțională (FP)

Programarea funcțională este programarea folosind funcții, o paradigmă de programare *declarativă* folosită pentru a crea programe cu o secvență de funcții simple mai degrabă decât instrucțiuni. Unele limbaje notabile ce utilizează această paradigmă sunt: Lisp, Haskell, ML, F#, etc.

În programarea convențională (iterativă) (C/C++, Java, C# etc.), instrucțiunile sunt luate ca un set de declarații într-o anumită sintaxă sau format, dar în cazul programării funcționale, toate calculele sunt considerate ca o combinație a funcțiilor matematice separate.

Utilizarea programării funcționale este în creștere. Este utilizată pentru baze de date mari, programare paralelă (CUDA pentru fiecare pas nou, procesul e executat pe un procesor nou) și învățarea automată. Programarea funcțională duce la infrastructuri stabile, care sunt fiabile, tolerante și mai puțin predispuse la erori.

Programarea funcțională facilitează utilizarea recursivității. Recursivitatea directă este o modalitate de definire a funcțiilor în care funcția este apelată în interiorul propriei definiții. Definițiile în matematică sunt adesea date recursiv.

2.4. Theoretical brief

Recursive functions

A function is recursive if its execution involves at least one self-call.

The recursion (self-call) can be:

- **Direct:** When the function calls itself directly.
- **Indirect:** When the function calls and is called back by other functions, in a circular manner.

A self-call is made for a smaller problem size. There exists a base case or a condition for which no self-call is made, and the function returns a result instead or stops triggering the recursion.

To avoid infinite recursion, it is essential that the self-call is tied to a condition that ensures termination, i.e., a stopping condition that breaks the cycle of recursive calls.

Any function call causes C/C++ to store the parameters of that function on the stack and the return address to the first instruction after the call. With repeated calls, the stack's usage may grow beyond allowable limits. To minimize stack usage, it is advisable to limit the number of parameters in recursive functions. If necessary, global variables can be used to reduce the number of parameters (although global variables should be used cautiously).

On each recursive call, a new instance of the function is created. Each instance has its own set of local variables (including formal parameters) that have the same names but occupy different memory spaces and may hold different values. At any given time, only the variables on the top of the stack (the current function call) are accessible, following the Last In, First Out (LIFO) principle.

Recursive techniques do not always lead to the most efficient solutions. It is advisable to compare the efficiency of iterative (non-recursive) and recursive solutions and choose the more advantageous approach. However, recursive solutions often result in more elegant and easier-to-understand code.

Examples of simple recursive algorithms:

- calculation of factorial;
- Fibonacci sequence calculation;
- Ackermann function;
- the greatest common divisor;
- calculation of combinations;
- calculation of the sum of digits of a number, etc.

Optional - Functional Programming (FP)

Functional programming is programming using functions, a declarative programming paradigm used to create programs with a sequence of simple functions rather than statements (instructions). Some notable functional programming languages are: Lisp, Haskell, ML, F#, etc.

In conventional (iterative) programing (C/C++, Java, C#, etc.), instructions are taken as a set of declarations in a specific syntax or format, but in the case of functional programing, all the computation is considered as a combination of separate mathematical functions.

The use of functional programming is growing. It's great for big databases, parallel programming (CUDA for each new step to be processed on a new processor), and machine learning. Functional programming leads to stable infrastructures that are reliable, fault-tolerant, and less prone to errors.

Recursion is a way of defining functions in which the function is applied inside its own definition. Definitions in mathematics are often given recursively.

2.5. Exemple / Examples

Sugestii de activități de realizat la parcurgerea exemplelor (opțional)

Urmăriți, pentru fiecare exemplu de mai jos, valorile parametrilor la apel, prin rularea pas cu pas a programelor folosind tasta F11 (specific Visual Studio). Urmărirea valorilor se va face în ferestrele Autos, Locals, Watch, iar fereastra Quick-Watch poate fi de folos pentru evaluarea de expresii în contextul curent al programului.

Faceți schimbările necesare pentru ca programul să verifice valorile parametrilor sau valoarea returnată pentru a asigura funcționarea corectă și evitarea de valori nepotrivite.

Re-implementați funcțiile recursive în mod nerecursiv acolo unde este posibil.

Calculați numărul de pași necesari în cazul utilizării algoritmului recursiv și în cazul utilizării algoritmului nerecursiv echivalent și comparați rezultatele.

Suggested activities while going through examples (optional)

For each of the following examples, watch the value of call parameters (arguments) while running the program in a step by step manner using the F11 key (specific to Visual Studio). The parameter values can be inspected using the Autos, Locals, Watch windows, and Quick-Watch can be used to evaluate expressions in the current context of the program.

Make the necessary changes for the program to check the call parameter values or the value returned by the function so that it performs well and for inappropriate values.

Write the same recursive function in iterative mode where possible.

Count the number of steps required in the case of recursion and non-recursion and make a comparison.

Ex. 1 - Factorial

```
//Factorial  (recursive implementation in C)  n! = n * (n-1)!
#include <iostream>
using namespace std;

int factorial(int n);

int main( ) {
    int step;
    cout << "\nEnter the step: "; cin >> step;
```

```
    cout << "\nFactorial of step = " << step << endl<<factorial(step) << endl;
    cout << "address 1";
}//main

int factorial(int n) {
    cout << "\nStep = " << n;
    if (n == 0)
        return 1;//exit condition
    else
        return n * factorial(n - 1);  //address 2
}//factorial
```

Ex. 2 - Fibonacci (recursiv ineficient / inefficient recursive)

```
//Fibonacci inefficient (exponential) recursive version
#include <stdio.h>
const int f=7;

int fib(int);

int main( ){
       printf("Fibonacci de %d =%d",  f, fib(f));//call for f = 7
}//main

int fib(int n){
       if (n < 2)
              return n;//so, fib(1)=1, fib(0)=0
       else
              return (fib(n-1) + fib(n-2));     //recursive call
}//fib_recursive
```

Ex. 3 - Fibonacci (echivalent iterativ / iterative solution)

```
/*The iterative linear solution for generating the Fibonacci number for entered n
  values no array */
#include <iostream>
using namespace std;

int main( ) {
    int l = 1, j = 0, k, n;
    cout << "Wished number is: ";
    cin >> n;
    cout<<"f[0]= 0\n";
    for (k = 1; k < n; k++)
    {
        j = l + j;
        l = j - l;
        cout <<"f[" <<k <<"]= " <<j <<endl;
    }//end for
}//main
```

Ex. 4 - Suma cifrelor unui număr în baza 10 / Sum of digits of a base 10 number

```
/*Sum of digits of a number in base 10 (recursive implementation)
  V(n)=0, if n=0,
  V(n/10)+n%10
*/
```

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

int sum_digits_recursive(int);

int main( ){
    int n;
    cout<<"Enter the number whose digits will be summed:";
    cin>>n;
    cout<<" The sum of the digits is:" << sum_digits_recursive(n);
}//main

int sum_digits_recursive(int n){
    if (n==0)
        return 0;
    else
        return sum_digits_recursive(n/10)+n%10;
}//sum_digits_recursive
```

Homework: Integrate the following iterative implementation:

```
int sum_digits_iterative(int n){
       int s=0;
       while(n != 0) {
              s += n%10;
              n /= 10;
       }
       return s;
}//sum_digits_iterative
```

Ex. 5 - Media și suma numerelor pare / Average and sum of even numbers

```
/* Average of even values from an array with a recursive method and a recursive sum
and the number of even elements obtained */
#include<iostream>
using namespace std;
const int dim = 20;

double mediaEven(int* a, int n);
int sumEven(int* a, int n, int& k);

int main() {
    int a[dim], s = 0, n;
    cout << "How many elements? (n>0, n<=" << dim << ") "; //C equivalent below
    //printf("How many elements ? (n>0, n<= %d) ", dim) ;
    cin >> n; // scanf("%d", &n) ;
    cout << "\nEnter " << n << " int elements:" << endl; //C equivalent below
    //printf("\nEnter %d  elements:\n", n) ;

    for (int i = 0; i < n; i++)
        cin >> a[i];//scanf("%d", &a[i])

    cout << "Media of even elements (recursive)= " << mediaEven(a, n);
    //printf("Media of even elements (recursive)= %.2lf ", mediaEven(a, n));
```

```
    int k=0; //init with 0
    s = sumEven(a, n, k);

    if (k != 0)
        cout << "\nSum of even elements (recursive)= " << s
             << " even numbers = " << k << " Media with sum= "
             << s / (double)k;
        /* printf("\nSum of even elements (recursive)= %d even numbers =  %d Media
with sum = %.2lf", s, k, s/(double)k); */
}//main

double mediaEven(int* a, int n) {
    static int k;
    if (n == 0) return 0;
    else
    if (a[n - 1] % 2 == 0) {
        k++;
        return mediaEven(a, n - 1) + (double)a[n - 1] / k;
    }
    else {
        return mediaEven(a, n - 1);
    }
}//mediaEven

int sumEven(int* a, int n, int& k) {
    if (n == 0) return 0;
    else if (a[n - 1] % 2 == 0) {
        k++;
        return sumEven(a, n - 1, k) + a[n - 1];
    }
    else return sumEven(a, n - 1, k);
}//sumEven
```

Ex. 6 - Opțional - programare funcțională / Optional - functional programming (Haskell)

In this example, we use Haskell to create a replication function. The function takes two inputs: a number `i` (which acts as an integer iterator) and an element `x` to be replicated. The function returns a list with `i` repetitions of the element `x`. For example, calling the function with `replicate ' 3 5` would return `[5, 5, 5]`.

Let's think about the exit condition. The base case for the recursion is when the iterator `i` is 0 or negative. If the iterator is 0 or less, the function returns an empty list, as it doesn't make sense to replicate an element a non-positive number of times.

The function in Haskell would look like this:

```
replicate ' :: ( Num i, Ord a) => i -> a -> [a]
replicate ' n x
| n <= 0 = [ ]
| otherwise = x: replicate ' (n -1) x
```

2.6. Lucru individual

1. Construiți o funcție recursivă care calculează A_n^k, unde *n, k* sunt citite de la tastatură, *k<n*. ($A_n^k = n \cdot A_{n-1}^{k-1}$; la *k=1,* A_n^k este *n*). Verificați rezultatul folosind și metoda bazată pe factorial. Validați datele de intrarea prin aserțiuni.
2. Calculați C_n^k, *n* și *k* fiind preluate de la tastatură, *k<n*, utilizând o funcție recursivă.
   ($C_n^k = \frac{n}{n-k} \cdot C_{n-1}^k$; pentru *n=0, k=0* sau *n=k*, C_n^k este 1, sau folosiți altă relație recursivă)
   Verificați rezultatul folosind și metoda bazată pe factorial. Validați datele de intrarea prin aserțiuni.
3. Calculul celui mai mare divizor comun a două numere folosind o funcție recursivă.
4. Se consideră recursivitatea indirectă (seria de medii arimetico-geometrice a lui Gauss):
   $a_n=(a_{n-1}+b_{n-1})/2.$, și
   $b_n=sqrt(a_{n-1}*b_{n-1})$, determinați a_n și b_n, pentru *n,* a_0, b_0 introduse de la tastatură.
5. Citiți un șir de caractere de la tastatură, caracter cu caracter (sau un șir de caractere într-un tablou de caractere), cu ajutorul unei funcții bazate pe caracter (inclusiv *getchar()*) (sau cu specificator adecvat). Afișați șirul în ordine inversă folosind o funcție recursivă.
6. Determinați printr-o funcție recursivă, și alta nerecursivă produsul scalar a doi vectori folosind tablouri unidimensionale de tip *int*. Considerați tablouri unidimensionale de aceeași lungime.
7. Să se calculeze suma numerelor impare dintr-un tablou unidimensional de numere întregi în mod recursiv, tablou citit dintr-un fișier unde, ca primă valoare, avem numărul de elemente ale tabloului.
8. Analog cu problema precedentă, dar se calculează produsul elementelor aflate pe poziții impare într-un tablou unidimensional, respectiv să se calculeze suma numerelor prime din tablou.
9. Folosind o funcție recursivă, calculați suma valorilor de tip *int* introduse de la tastatură cu confirmare in *main()*, adică cereți utilizatorului să indice dacă mai dorește să mai introducă o nouă valoare sau nu. Adăugați o nouă funcționalitate funcției recursive pentru a calcula și afișa și media valorilor date de utilizator. Semnalați printr-un mesaj când suma valorilor depășește o anumită valoare prestabilită.
10. Considerați un tablou unidimensional de *n* (<=30) de valori întregi. Determinați în mod recursiv și nerecursiv numărul de apariții în tablou ale unei valori întregi *x* citite de la tastatură.
11. Considerați un număr *n* întreg pozitiv în baza 10 introdus de la tastatură. Folosind o funcție recursivă convertiți valoarea *n* într-o altă bază de numerație 1<*b<10* citită de la tastatură.
12. Fie ecuația de gradul 2: $x^2 - sx + p = 0$. Fără a calcula rădăcinile x_1 și x_2 determinați, daca e posibil, $S_n = x_1^n + x_2^n$, folosind reprezentarea recursivă a sumei: *Sum(n) = { 2, dacă n=0; s, dacă n=1; s*Sum(n-1) – p*Sum(n-2), dacă n>1; }* unde *s* și *p* sunt valori reale, iar *n* întreagă, introduse de la tastatură.
13. Scrieți un program care să calculeze în mod recursiv și în mod nerecursiv valoarea seriei armonice $s_n=1/1+1/2+1/3+\ldots 1/n$, unde *n* este un număr natural, cu două funcții diferite in același program. Apelați cele două funcții cu diferite valori ale lui *n*.

Opțional, activitate suplimentară:

- Aplicații folosind numărul lui Fibonacci, cu numărul de aur, secțiunea de aur, unghiul de aur. Semnificație, utilizare (inclusiv codare), optimabilitate privind implementarea.

- Formula lui Moivre, ce face legătura între numerele complexe și trigonometrie. Dacă avem un număr complex *z* dat sub forma trigonometrică: *z=ρ(cosϕ+isinϕ),* atunci: $z^n=[\rho(\cos\phi+i\sin\phi)]^n=\rho^n(\cos n\phi+i\sin n\phi)$. Formula lui Moivre poate fi folosită pentru a exprima *cosnϕ* și *sinnϕ* ca puteri ale *cosϕ* și *sinϕ* în mod recursiv. Aplicații practice la transformata Fourier.
- Analiza corectitudinii unor expresii și obținerea formei poloneze postfixate, prefixate etc.
- Tehnici recursive de generare și desenare a fractalilor, etc.
- Analizați soluțiile oferite de *chatGPT* pentru problemele de la teme, în variante recursive și non-recursive

2.7. Individual work

1. Write a recursive function that calculates A_n^k , where *n* and *k* are read form the keyboard, *k<n*. ($A_n^k = n \cdot A_{n-1}^{k-1}$; *if k=1* A_n^k is *n*). Verify the result using the factorial definition. Validate input data by assertions.
2. Write a recursive function that calculates C_n^k , where *n* and *k* are read from the keyboard, *k<n*.

   ($C_n^k = \frac{n}{n-k} \cdot C_{n-1}^k$; *if n=0, k=0 or if n=k,* C_n^k is 1, or another definition). Verify the result using the factorial definition. Validate input data by assertions.
3. Calculate the greatest common divider of 2 numbers using a recursive function.
4. Considering the following indirect recursive formulas (Gauss arithmetical-geometrical media):
   $a_n=(a_{n-1}+b_{n-1})/2.$, and $b_n=sqrt(a_{n-1}*b_{n-1})$,
determine a_n and b_n for *n, a0* and *b0* read from the standard input.
5. Read a string of characters from the keyboard, one character at a time, using a character function (inclusiv *getchar()*), (or a complete string in a character array, using a specific mechanism). Reverse the string using a recursive function character by character or the entire character string.
6. Determine the scalar product of 2 vectors using a recursive function and a non-recursive function. Consider one-dimensional arrays of *int* type, same dimension.
7. Calculate the sum of the odd numbers from an array of integer values, using a recursive function. The numbers from the array are read from a file. The first value in the array represents the array's length.
8. Using the code developed for the previous problem, calculate the product of the elements located on odd positions in the one-dimensional array, and also calculate the sum of the prime numbers in the array.
9. Using a recursive function, calculate the sum of a series composed of keyboard entered int values. The values are read as long as the user desires so. Add other functionality to the function in order to determine and display the average value of the entered numbers. Print on the screen a significant message when the sum is greater than a predefined value.
10. Consider a one-dimensional array of *n* (<=30) integer values. Determine (recursively and non-recursively) the number of times a certain value *x* read from the keyboard appears in the array.
11. Read from the keyboard a positive integer value *n* (base 10). Use a recursive function for converting *n* into another base 1< *b<10*, also read from the keyboard.
12. Consider the 2-nd degree equation $x^2 - sx + p = 0$. Without calculating the solutions x_1 and x_2 dete1rmine, if it is possible, $S_n = x_1^n + x_2^n$ using this sum's recursive definition: *Sum(n) =*

*{ 2, if n=0; s, if n=1; s*Sum(n-1) – p*Sum(n-2), if n>1; }*. *s* and *p* are *float* values, *n* is an integer value, all read from the keyboard.

13. Write a program that calculates recursively and non-recursively (two distinct functions in the same program) the value of the harmonic series $s_n=1/1+1/2+1/3+…1/n$, where *n* is a natural number. Call the functions with 2 different values for *n*.

Optional additional activity:

- Applications using Fibonacci numbers, with golden number, golden section, golden angle. Significance, use (including coding), optimality concerning implementation.
- Moivre's formula, which makes the connection between complex numbers and trigonometry. If we have a complex number z is given in trigonometric form: $z = \rho (\cos\phi + i\sin\phi)$, then:
$z^n=[\rho(\cos\phi+i\sin\phi)]^n=\rho^n(\cos n\phi+i\sin n\phi)$. Moivre's formula can be used to express $\cos n\phi$ and $\sin n\phi$ as power of $\cos\phi$ and $\sin\phi$ recursively. Fourier transform applications are used.
- Analysis of the correctness of some expressions and obtaining the *Polish* form postfixed, prefixed, etc.
- Recursive techniques for generating and drawing fractals, etc.
- Analyze the solutions obtained with *chatGPT* for the individual work, in recursive and non-recursive variants

3. Metode de programare recursive și nerecursive. Metoda Backtracking. Metoda Divide et Impera. Tehnici de căutare.

Recursive and non-recursive programming methods. The Backtracking method. The Divide et Impera method. Searching techniques.

3.1. Obiective

- Înțelegerea tipurilor de probleme la care se pretează folosirea metodelor „backtracking” și „divide et impera”.
- Înțelegerea mecanismului de implementare a unor aplicații folosind metode de tip „backtracking” și „divide et impera”.
- Înțelegerea tehnicilor de căutare.

3.2. Objectives

- Understanding the problem types that can be solved using the „backtracking” and “divide et impera” techniques.
- Understanding the programming mechanism for the applications that use „backtracking” and „divide et impera”.
- Understanding the searching techniques.

3.3. Breviar teoretic

Metoda backtracking

Soluția are mai multe componente x[k], k=0,n-1 și fiecare componentă poate lua valori x[k]=1,…,h.

a. VARIANTA NERECURSIVĂ

```
//initializari
k=0;
x[k]=0;
/*repeta pana cand ne-am intors
inapoi mai mult decat trebuia*/
do{
        //atata timp cat mai sunt valori de ales
      while  ( x[k]<h )
                              {
      //trec la urmatoarea valoare
      x[k]=x[k]+1;

      if (posibil(k))   //daca este solutie partiala corecta

         if   ( k==n-1)    //daca am ajuns la o solutie completa
                              afiseaza  solutia(X);
                    else  //trec la urmatoarea componenta
                                    {
                                          k=k+1;
                                          x[k]=0;
                                    }
        }//while
      k=k-1; /*fac pasul inapoi*/
}while(! (k<0));// (k>=0)
```

b. VARIANTA RECURSIVĂ

```
void Backtracking(int k)
{
//pentru toate valorile pe care le poate lua X[k]
for(int i=1;   i <=   h  ;  i++)
{
   x[k]=i;
        if  ( posibil(k)) //sol. partiala posibila

        if (k==n-1)  //solutie completa

                afiseaza  solutia(X);
        else
                Backtracking(k+1);
                //apel recursiv
  }//for
}//Backtracking
```

Apel in main():

```
//initializari
Backtracking(0);
```

Soluția este reprezentată ca un *vector* (implementat printr-un tablou unidimensional) ***X***:

$$\boldsymbol{X}(x_1, x_2,..., x_n) \in \boldsymbol{S}=S_1xS_2x...xS_n$$

unde:

S e spațiul soluțiilor posibile care este finit (produs cartezian a S_i)

x_i, *i=1,...n,* sunt componentele soluției S, cu valori în S_i

Funcția ***posibil(k)*** returnează**:**

- **TRUE** dacă e soluție parțială corectă, adică posibilă și se poate continua sau
- **FALSE** în caz contrar.

Se poate să nu fie o funcție pentru cazurile mai simple sau funcția poate avea mai mulți parametri în cazul în care condiția de continuitate este mai complexă sau ea oferă valori în etapa verificării dacă avem o soluție completă sau nu (ce se poate implementa dacă e mai complicată cu o funcție ***gata_solutie()***).

Funcția ***afiseaza_solutia(X)*** are rolul de a afișa o soluție completă obținută la un moment dat. Funcție de problema care e rezolvată poate să aibă diferite forme de implementare.

La orice problemă standard de backtracking se stabilesc:

1. *k = 0,…n-1*, adică numărul de componente ale unei soluții
2. *x[k]=1,…h*, adică domeniul valorilor posibile ale unei componente
3. *posibil(k)*, condiția de continuitate - este o condiție internă care stabilește ce relații trebuie respectate între componentele unei soluții.

Atât în varianta recursivă cât și în cea nerecursivă inițial se determină (citesc) date referitoare la:

- Numărul componentelor, *n*
- Domeniul valorilor, *h*
- Alte date necesare pentru elaborarea algoritmului

De asemenea, sunt unele aplicații în care se impun unele restricții de pornire. Există variante de backtracking în care soluțiile nu au aceeași lungime (număr variabil de componente) sau se cere obținerea unor soluții optimale, etc.

Metoda divide et impera

Principiul propus de metodă este:

- Descompune problema în subprobleme în mod recursiv, până când ajungem la o subproblemă pe care o putem rezolva.
- Soluțiile subproblemelor se vor combina obținând soluția finală a problemei.

Tehnici de căutare

În general se pune problema căutării unor obiecte pe baza valorii unui câmp (cheie) asociat fiecărui obiect. Vom considera că obiectele sunt grupate în tablouri unidimensionale și că pentru cheie este definită o relație de ordine.

În cazul în care obiectele nu sunt ordonate, nu este posibilă decât o căutare directă, adică parcurgerea liniară a tabloului și compararea valorii cheii cu valoarea de căutat. Dacă însă obiectele sunt ordonate, găsirea unui obiect se poate face mai rapid. Un algoritm în acest sens este cel de căutare binară, atât în variantă recursivă, cât și nerecursivă, ce folosește înjumătățirea intervalului, în fond un algoritm "Divide et impera".

În cazul unor tablouri neordonate, se pot folosi următoarele funcții de bibliotecă (există și **alte variante** ale acestor funcții în noile variante ale mediului **VC++** care pot fi folosite):

```
void * lfind(const void *key, const void *base, unsigned *num, unsigned
width, int (*fcmp)(const void *, const void*));

void * lsearch(const void *key, const void *base, unsigned *num, unsigned
width, int (*fcmp)(const void *, const void *));
```

În cazul tablourilor ordonate există funcția de bibliotecă `bsearch()` pentru căutarea binară (înjumătățirea intervalului):

```
void * bsearch(const void *key, const void *base, unsigned num, unsigned
width, int (*fcmp)(const void*, const void*));
```

3.4. Theoretical brief

The backtracking method

The solution has a series of components x[k], k=0, n-1 and each component can have as possible values x[k]=1,…,h.

a. NON-RECURSIVE VARIANT

```
//init
k=0;
x[k]=0;
/* repeat until we are back
 more than it should */
do{
        //as long as there are still values to choose from
      while   ( x[k]<h )

                                {
      //pass to next value
      x[k]=x[k]+1;

      if (posibil(k))    //if it is a correct partial solution

        if    ( k==n-1)    // if it is ready_solution()
                                display_solution(X);
                    else  //pass to next component
                                {
                                                k=k+1;
                                                x[k]=0;
                                }

          }//while
        k=k-1; /*back step*/
}while(! (k<0));// (k>=0)
```

b. RECURSIVE VARIANT

```
void Backtracking(int k)
        {
        // for all the values it can take X[k]
        for(int i=1;   i <=   h ;  i++)
{
          x[k]=i;
                if ( posibil(k)) //partial possible sol.

                if (k==n-1)  //ready_solution ()

                display_solution(X);
                else
                        Backtracking(k+1);
                //recursive call
    }//for
}//Backtracking
```

Apel in main():

```
        //init
        Backtracking(0);
```

The solution is represented as a *vector* (implemented through a one-dimensional array) ***X***:

$$\boldsymbol{X}(x_1, x_2, \ldots, x_n) \in \boldsymbol{S}=S_1 x S_2 x \ldots x S_n$$

where:

S is the space of possible solutions and is finite (cartesian product of S_i)

x_i, *i=1,...n,* are the components of solution S, having the values in S_i

The function ***posibil(k)*** returns**:**

- **TRUE** if it is a correct partial solution, i.e. possible and can be continued, or
- **FALSE** otherwise.

In simpler cases the function can be omitted or the function can have more parameters for the more complex cases. The function can offer values during the checkup phase and if neccesary a ***ready_solution()*** function may be implemented).

The function ***display_solution(X)*** has the role of displaying a complete solution obtained at a given time. Depending on the problem that is solved, it can have different forms of implementation.

In each standard backtracking problem, the following shall be established:

1. *k = 0,…n-1*, the number of components of a solution
2. *x[k]=1,…h*, The range of possible values for a solution component
3. *posibil(k)*, the continuity condition, an internal condition that determines what relationships must be respected between the components of a solution.

In both the recursive and non-recursive versions, the following data is determined (or read):

- Number of components, *n*
- Value range, *h*
- Other data necessary for the development of the algorithm

There are also some applications where some startup restrictions are imposed. There are backtracking variants where the solutions don't have the same length (variable number of components) or optimal solutions are required, etc.

Divide and conquer method (divide et impera)

The principle proposed by the method is:

- Break down the problem into subproblems recursively until we reach a subproblem that we can solve.
- The solutions of the subproblems will be combined to obtain the final solution of the problem.

Searching techniques

In general, they solve the problem of finding if some objects are to be found in a collection, based on a specific key's value. The objects are usually organized in unidimensional arrays.

If the objects are not ordered, only a direct search is possible in which the array is browsed liniarly and the key of each element is compared with the searched value. However, if objects are ordered, finding an object can be done faster. An algorithm for this case is the ***binary search*** in both recursive and non-recursive variants, that works with halving the searched interval, being basically a "Divide et impera" algorithm.

In the case of unordered arrays, the following library functions can be used (there are **other variants** of these functions in some variants of **the VC++** environment):

```
void * lfind(const void *key, const void *base, unsigned *num, unsigned
width, int (*fcmp)(const void *, const void*));
```

```
void * lsearch(const void *key, const void *base, unsigned *num, unsigned
width, int (*fcmp)(const void *, const void *));
```

In the case of ordered arrays, there is a library function `bsearch()` for binary search (halving the interval):

```
void * bsearch(const void *key, const void *base, unsigned num, unsigned
width, int (*fcmp)(const void*, const void*));
```

3.5. Exemple/ Examples

Ex. 1 - Backtracking (Colorare steag cu n dungi / Coloring a flag with n stripes)

```
/* Coloring a flag with n (<=5) stripes, each stripe can have h colors */

//a) non-recursive variant
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define DIM 5

int posibil(int);
void afis_sol(int n);

int x[DIM];

int main()
{
    int k,h,n;
    printf("\nEnter the number of flag stripes (<=%d): ", DIM);
    scanf("%d",&n);
    printf("\nEnter the maximum number of possible colors for a stripe: ");
    scanf("%d",&h);
    k=0;
    x[k]=0;
    do
    {
        while(x[k] < h) //There are still possible values for component k
        {
            x[k]++; //go to the next position
            if(posibil(k))
            if(k==(n-1))
                afis_sol(n);  //solution is ready
            else {
                k++;
                x[k]=0;
            } //move to the next component and start from zero
        }//while
        k--; //no more values for component k; return to k-1 component
    }while(!(k<0));//turning back is no longer possible
}//main

int posibil(int k){
    if(k==0) return 1;     // initially everything is possible
    if(x[k] == x[k-1])     // 2 identical side-by-side colors (not possbile)
        return 0;
    return 1; //ok
}//posibil

void afis_sol(int n){
    //display the current solution
    for(int i=0;i<n;i++)
        printf("%d ",x[i]);
    printf("\n");
}//afis_sol
```

```
//b) recursive variant
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define DIM 5

int posibil(int);
void afis_sol(int n);
void dr_rec(int k);
int x[DIM], n;
int h;

int main() {
    printf("\nEnter the number of flag stripes (<=%d)\n", DIM);
    scanf("%d", &n);
    printf("Enter the maximum number of possible colors for a stripe\n");
    scanf("%d", &h);
    dr_rec(0);  //call of recursive function
}//main

void dr_rec(int k){
    for (int i = 1; i <= h; i++){
        x[k] = i;
        if (posibil(k))
            if (k == (n - 1)) afis_sol(n);    //final solution
            else dr_rec(k + 1);    //move to next component
       }
}//dr_rec

int posibil(int k){
    if (k == 0)return 1;  // initially everything is possible
    if (x[k - 1] == x[k]) // 2 adjacent identical colors (not allowed)
        return 0;
    return 1; // ok
}//posibil

void afis_sol(int n){
    //display the solution
    for (int i = 0; i < n; i++)
        printf("%d ", x[i]);
    printf("\n");
}//afis_sol
```

Ex. 2 - Backtracking (Număr exprimat ca sumă / Number expressed as a sum)

```
/* Expressing a number n as a sum of natural numbers (backtracking with variable
number of components in the solution) */

//a) recursive variant
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define DIM 20

void back_SubSet(int);
int posibil(int, int&);
void afis_sol(int);
```

```
int x[DIM],n;

int main() {
    printf("\nEnter the number to be decomposed (<=%d): ", DIM);
    scanf("%d",&n);
    printf("\n\tSolutions :\n");
    back_SubSet(0);
}//end main

void back_SubSet(int k) {
int sum;
    x[k] = 0;
    while(x[k] < n) {
        x[k]++;
        if(posibil(k, sum)) {
            if(sum==n)
                afis_sol(k);
            else
                back_SubSet(k+1);
            }//end else
    }//end while
}//end back_SubSet

int posibil(int k, int &s){
    s=0;
    if(k==0)
        return 1;
    if(x[k] >= x[k-1]) {
        for( int i=0;i<=k;i++)
            s += x[i];
        if(s <= n)
            return 1;
    }//end if
    return 0;
}//end posibil

void afis_sol(int k){
    printf("\n\t");
    for(int i=0;i<=k;i++)
         printf("%d  ",x[i]);
}//end afis_sol

//b) iterative variant
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define DIM 20

int posibil(int, int&);
void afis_sol(int);

int x[DIM], n;

int main(){
    int k,sum;
    printf("\nEnter the number to be decomposed (<=%d): ", DIM);
    scanf("%d",&n);
```

```
    printf("\n\tSolutions :\n");

    k=0;
    x[k]=0;
    do  {
        while(x[k] < n) {
            x[k]++;
            if(posibil(k, sum)){
                if(sum==n)
                    afis_sol(k);
                else {
                    k++;
                    x[k]=0;
                }//end else
            }//end if
        }//end while
        k--;
    }
    while(!(k<0)); //end do…while
}//main

The functions:
int posibil(int k, int &s);
void afis_sol(int k);
are identical with the ones presented in the recursive variant.
```

Ex. 3 - Backtracking (Problema comis-voiajorului / Traveling salesman problem)

```
/* Traveling salesman problem. Optimal solution.
n cities with connections represented as a cost matrix. The costs (val!=0)
represent direct links, otherwise detour. Starting from town i, the salesman has to
visit all the cities only once with minimum cost */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define MAX 7

int x[MAX], y[MAX];//x solution components, result y
int  cost[MAX][MAX];//cost matrix

void cit(int[][MAX], int);// read cost matrix
void afis(int[][MAX], int);//display cost matrix
int max_cost(int[][MAX], int);//det. max cost
void afis_sol(long, int);//display optimum solution
int posibil(int);// direct route between cities and cities not already visited

int main( ) {
    int k,n;
    long cost_M, n_C;
    printf("Enter dim of cost matrix(no. of cities) <=%d\n", MAX);
    scanf("%d", &n);
    printf("Enter and display the cost matrix\n");
    cit(cost, n);
    afis(cost, n);
    cost_M = n * (long)max_cost(cost, n) + 1;//initial max cost
```

```
    printf("initial max cost is= %ld\n", cost_M);
    k = 0;//first component
    printf("Enter initial city (0, ..., %d)", n-1);
    scanf("%d", &x[k]);
    k = 1;//next component
    x[k] = -1;//possible values from 0 therefore start from -1
    do {
        while (x[k] < n) // there are still possible values for component k
        {
            x[k]++; //pass on to the next value
            if (posibil(k))
                if((k == (n - 1))&&(cost[x[n - 1]][x[0]] != 0)) {
                    /* that is, he reached the city n-1
                       and can return from where he left */
                    n_C = 0;//new cost
                    for (int i = 0; i < n - 1; i++) n_C += cost[x[i]][x[i + 1]];
                    n_C += cost[x[n - 1]][x[0]];//total cost total including return
                    if (n_C < cost_M)
                        for (int i = 0; i < n; i++) {
                            y[i] = x[i]; cost_M = n_C;
                        }
                    //save new obtained solution , change cost_M
                }
                else { k++; x[k] = -1; } // pass on to the next component
        }//while
        k--; // there are no more values for component k. Return the k-1 component
    } while (!(k < 1));
    afis_sol(cost_M, n);  //solution ready
}//main

int posibil(int k){
    if (k == 0)return 1;
    if (cost[x[k - 1]][x[k]] != 0) {//direct road
        for (int i = 0; i < k; i++)// the city was not yet chosen
            if (x[k] == x[i])   return 0;
        return 1;
    }
    return 0;
}//posibil

void afis_sol(long fc, int n){
    for (int i = 0; i < n; i++) //display the current solution and cost
        printf("%d ", y[i]);
    printf("\nThe cost is: %ld", fc);
}//afis_sol

void cit(int mat[][MAX], int n) {
    for (int i = 0; i < n - 1; i++){
        mat[i][i] = 0;//same city
        for (int j = i + 1; j < n; j++){
            printf("Enter elem[%d][%d]= ", i, j);
            scanf("%d", &mat[i][j]);
            mat[j][i] = mat[i][j];//symetric main diagonal
        }
    }
}//cit
```

```
void afis(int mat[][MAX], int n){
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++)
            printf("%4d ", mat[i][j]);
        printf("\n");
    }
}//afis

int max_cost(int mat[][MAX], int n){
    int max = mat[0][0];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (mat[i][j] > max)
                max = mat[i][j];
    return max;
}//max_cost
```

Ex. 4 - Backtracking (Generator de note posibile / Generation of possible grades)

```
/* Generator de note posibile pornind de la 3 ponderi si nota finala dorita.
   Generate some weighted partial grades starting from a final grade.
*/

#include <iostream>
using namespace std;
#include <math.h>//gcc, clion

#define NRNOTE 3

void afis_sol(void);

int x[NRNOTE], n = 1, cont;
float p1, p2, p3;
int nmin = 4, nmax = 10;

int main( ){
    int k;
    float p;
    cout << "\nEnter 3 weights \nWeight 1 : ";
    cin >> p1;
    cout << "\nWeight 2 : ";
    cin >> p2;
    cout << "\nWeight 3 : ";
    cin >> p3;
    p = p1 + p2 + p3;
    if (p != 1.0f) {
        cout << "\nWrong weights...";
        exit(0);
    }
    cout << "\nEnter the final grades (0 - 10, 0 ends the cycle)";
    while (n != 0)
    {
        cout << "\n Grade : ";
        cin >> n;
        if (n <= nmax)
        {
```

```
            cont = 0;
            k = 0;
            x[k] = 0;
            do{
                while (x[k]<nmax) {
                    x[k] = x[k] + 1;
                    if (x[k] > nmin - 1) {
                        if (k == (NRNOTE - 1)) {
                            afis_sol();
                        }
                        else {
                            k++;
                            x[k] = 0;
                        }
                    }
                }
                k--;
            } while (!(k<0));
        }
        else cout << "\nInvalid grade";
    }
}//main

void afis_sol(void){
    int l;
    double fraction, integer;
    double number;
    number = (double)(p1*x[0] + p2 * x[1] + p3 * x[2]);
    fraction = modf(number, &integer);
    if (fraction >= .5)
        integer++;
    //if (fraction >=.5) number=number+1;
    // l=(int)number;
    l = integer;
    if (l == n){
        printf("\n N1 = %d N2 = %d N3 = %d", x[0], x[1], x[2]);
        cont++;
        if (cont >20) {
            cont = 0;
            cout << "\nIntrodu un caracter pentru continuare";
            cin.ignore();
              cin.get( );
        }
    }
}//afis_sol
```

Ex. 5 - Divide et impera / Divide and Conquer (Turnurile din Hanoi / Tower of Hanoi)

```
/* Divide et impera. Tower of Hanoi, recursive variant */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

double mutari; // number of moves
void hanoi(int n, char a, char b, char c);
```

```
int main(){
    int n;
    printf("Number of disks = ");
    scanf("%d",&n);
    mutari=0;
    hanoi(n, 'A', 'B', 'C');
    printf("Number of moves = %lf\n", mutari);
}  //main

void hanoi(int n, char a, char b, char c)
 //n-nr. of disks, a-source, b-destination, c-maneuver
{
    if (n==1){
        printf("Disk 1 is moved from pole %c on pole %c\n",a,b);
        mutari++;
        return;
    }

    // n>1, n-1 disks, a-source, c-destination, b-maneuver
    hanoi(n-1,a,c,b);

    // Disk n goes from pole a to pole b
    printf("Disk %d goes from pole %c to pole %c\n",n,a,b);
    mutari++;

    // n>1, n-1 disks c-source, b-destination, a-maneuver
    hanoi(n-1,c,b,a);
} // hanoi
```

Ex. 6 - Sortare / Sorting (_lsearch, _lfind, bsearch)

```
/* complete examples with searching library functions */

//*************************************************************************
/* The program uses the lsearch() library function to find the name of a month; if
not found, add it to the end */

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
#include <search.h>

const int dim = 12;
const int dim_c = 20;

int cmp(const void* arg1, const void* arg2);
int addelem(const char* key, const char** tab, int nelem);

int main( ) {
    const char* luni[dim] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun" };
    int nluni = 6;
    int i;
    //const char* key = "Jul";
    char key[dim_c];
    printf("\nEnter the searched month: ");
```

```
    scanf("%s", key);
    if (addelem(key, luni, nluni))
        printf("Month %s is already stored.\n", key);
    else {
        nluni++;
        printf("Month \"%s\" was added to the array: ", key);
        for (i = 0; i < nluni; i++)
            printf("%s, ", luni[i]);
    }
}

int addelem(const char* key, const char** tab, int nelem){
    int oldn = nelem;

    _lsearch(
        &key, tab, (unsigned*)&nelem, sizeof(tab[0]),
        (int(*)(const void*, const void*))cmp
    );
    return(nelem == oldn);
}

int cmp(const void* arg1, const void* arg2){
    char ** f1 = (char**)arg1;
    char ** f2 = (char**)arg2;
    return(_stricmp(*f1, *f2));
    //return(_stricmp(*(char**)arg1, *(char**)arg2));
}

//*************************************************************************
// using the _lfind() library function, VC++22

#define _CRT_SECURE_NO_WARNINGS
#include <stdlib.h>
#include <stdio.h>
#include <search.h>

int compare_int(const void * a, const void * b);

int main( ){
    int int_values[ ] = { 1, 3, 2, 4, 5 };
    int* int_ptr, key_value, num; //key_value is the searched value
    printf("Enter the searched value: ");
    scanf("%d", &key_value);
    num = sizeof(int_values) / sizeof(int_values[0]);
    //call _ lfind()
    int_ptr = (int*)_lfind(&key_value, int_values, (unsigned*)&num,
                           sizeof(int_values[0]),
                           (int (*) (const void*, const void*)) compare_int);
    if (int_ptr != 0)
        printf("Number %d is in array on pos. : %lld and address %p\n",
                key_value,
                (_int64)(int_ptr - int_values + 1),
                int_ptr);
    else
        printf("Number %d is not in the array\n", key_value);
}//end main
```

```
int compare_int(const void * a, const void * b){
    return(*(int *)a - *(int *)b);
}//end compare_int

//*****************************************************************************
// using the bsearch() library function

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

int compare_int(const void * a, const void * b);
int compare_float(const void * a, const void * b);

int main( ){
    int int_values[ ] = { 1, 2, 3, 4, 5 };
    float float_values[ ] = { 1.1f, 2.2f, 3.3f, 4.4f, 5.5f };

    int* int_ptr, int_value = 2, num;
    float* float_ptr, float_value = 33.3f;
    num = sizeof(int_values) / sizeof(int_values[0]);
    //call bsearch() for an array of integer values
    int_ptr = (int*)bsearch(&int_value, int_values, num, sizeof(int_values[0]),
                            (int (*) (const void*, const void*)) compare_int);
    if (int_ptr)
        printf("Value %d was found!\n", int_value);
    else
        printf("Value %d was not found...\n", int_value);

    num = sizeof(float_values) / sizeof(float_values[0]);
    //call bsearch() for an array of real numbers
    float_ptr = (float*)bsearch(&float_value, float_values, num,
                           sizeof(float_values[0]),
                           (int (*) (const void*, const void*)) compare_float);

    if (float_ptr)
        printf("Value %3.1f was found!\n", float_value);//end if
    else
        printf("Value %3.1f was not found...\n", float_value);//end else
}//end main

int compare_int(const void * a, const void * b){
    return(*(int *)a - *(int *)b);
}//end_compare_int

int compare_float(const void * a, const void * b){
    float* fa = (float*)a;
    float* fb = (float*)b;
    if (*fa < *fb) return -1;
    if (*fa > *fb) return 1;
    return 0;
}//end_compare_float
```

3.6. Lucru individual

1. Testați aplicațiile privind metodele de programare oferite în cadrul laboratorului. Folosiți aceste aplicații pentru a implementa problemele 2 și 4 propuse, considerând în acest sens și aplicațiile adiționale din laborator.
2. Fie un sistem de calcul care urmărește controlul unui proces de transmisie a datelor pe o linie principală având un debit maxim de 40MB/s. Fluxul de date de pe această linie este împărțit de către maxim 10 utilizatori, traficul pe liniile oferite lor putându-se efectua cu debite între 2 și 40 MB/s (valori intregi). Impărțirea debitelor pe cele maxim 10 linii se face în mod dinamic de către sistemul de control considerând pentru fiecare linie i, o pondere subunitară p_i, asociată la configurarea sistemului funcție de utilizator. Introduceți inițial cele maxim 10 ponderi p_i astfel încât suma lor să fie egală cu 1. Dacă această condiție este verificată generați toate soluțiile posibile considerând că pe linia principală vom avea cel puțin un debit de 2MB/s, deci de 2...40MB/s determinând în aceste cazuri debitele posibile pe cele maxim 10 linii de intrare ale utilizatorilor (ajustate la intregi MB/s). Se pune pe 0, dacă nu se poate asigura minimul de 2MB/s. Afișați aceste soluții. Implementarea poate fi recursivă sau nerecursivă.
3. Considerând exemplul legat de problema comis voiajorului, rezolvați această problemă considerând cazul în care matricea de costuri nu este una simetrică față de diagonala principală și costurile de deplasare în acelaș oras pot fi diferite de zero (caz oferte de prețuri).
4. În cadrul unei companii de software se pune problema difuzării unor pachete de date la mai multe filiale, maxim 7, valoarea fiind introdusă de la intrarea standard. Filialele sunt dispuse în mai multe puncte din lume, costurile de transmisie fiind dependente de poziția lor geografică. Pachetele trebuie să ajungă la fiecare filială chiar dacă nu există o legătură directă între toate filialele ci prin intermediul unei alte filiale caz în care pachetul va urma o rută ocolitoare. Costurile necesare transmiterii datelor în rețea se introduc inițial la generarea sistemului. Să se determine și afișeze ruta pe care trebuie transmise pachetele de date astfel încât pornind de la filiala *i*, citită de la intrarea standard pachetul să fie transmis cu cost minim (eventual cu confirmare adică pachetul poate să se întoarcă înapoi). Implementarea poate fi recursivă sau nerecursivă. Folositi un *LUT* (Look Up Table) pentru a asocia valorilor soluției obținute, șiruri de caractere adecvate cu numele orașelor.
5. Problema investirii optime de capital: pentru un investitor cu un capital *C* și *n* oferte la care trebuie avansate fondurile f_i și care aduc beneficiile b_i, se cere selectarea acelor oferte care îi aduc beneficiul maxim. Folosiți backtracking și standard C pentru implementare. Afișați ofertele alese, beneficiile și beneficiul maxim.
6. Creați un fișier în care stocați un șir de numere întregi (<100), generate in mod aleator, valori <1000. Preluați prin program aceste valori. Folosiți metoda *divide et impera* pentru a determina minimul și maximul din acest fisier și afisați rezultatele pe ecran. Actualizați fișierul inițial adăugând aceste două valori la final, pe rând nou.
7. Scrieți un program C care generează maximum 200 de numere întregi (număr par) într-un mod aleator cu valori nu mai mari de 500 valori ce vor fi stocate într-un fișier de intrare. Folosind metoda „divide et impera” determinați cel mai mare divizor comun pereche după pereche din aceste numere întregi care sunt generate într-un mod aleatoriu. Afișați valorile perechilor și cel mai mare divizor comun, rând cu rând. Aceste valori vor fi stocate într-un fișier de ieșire în același mod.
8. Să se calculeze $\int_a^b 1/(1+x^2)dx$, cu ajutorul metodei trapezelor, astfel încât înălțimea fiecărui trapez a cărui arie se însumează să fie mai mică decât $\varepsilon=0.0001$. Aria trapezului cu vârfurile în punctele *(a,0), (b,0), (a,f(a))* și *(b,f(b))* este *(b-a)*(f(a)+f(b))/2*, iar

f(x)=1/(1+x²). Se citesc de la tastatură numerele reale *a* și *b, a≤b.* Utilizați metoda divide et impera.

9. Să se scrie o aplicație C/C++ care să genereze aleator maxim 10 valori întregi, ce vor fi memorate într-un tablou unidimensional. Să se verifice dacă o altă valoare generată aleator aparține acestui tablou, utilizând funcția *_lsearch()*.
10. Scrieți o aplicație C/C++ care să găsească imaginile cele mai apropiate folosind pe rând o cheie de căutare din antetul imaginii. Antetul este reprezentat printr-o structură ce conține un nume de fișier (șir de caractere), o cale (șir de caractere), o rezoluție de intensitate (întreg), o dimensiune în octeți (întreg). Pentru fiecare cheie folosiți o metodă diferită de căutare. Antetul imaginilor se află stocat într-un fișier sau se citește într-un tablou de structuri de la tastatură.
11. Citiți de la intrarea standard un tablou unidimensional de maxim 20 de valori întregi. Folosind mecanismul de căutare binară, determinați dacă o nouă valoare, *a,* introdusă de la tastatură există în șir. Dacă da, determinați toți factorii primi ai acestei valori pe care îi veți afișa pe ecran.
12. Folosind un fișier care conține numere reale ordonate, căutați o valoare reală *a* introdusă de la tastatură în cadrul șirului, folosind algoritmul de căutare binară iterativ, recursiv și funcția de bibliotecă *bsearch()*. Afișați șirul citit, valoarea *a* și poziția (dacă a fost gasită) folosind cei trei algoritmi specificați.

3.7. Individual work

1. Test the applications related to the programming methods described in this document. Use these applications for implementing problem 2 and problem 4.
2. Consider a computing system that monitors the data transfer on a main communication channel that has a maximum flow capacity of 40 MBps. The data stream can be shared among max 10 users. The channel's data flow is divided automatically by the monitoring program using max 10 (0...9) sub-unitary weights, each of them being associated with a user. The weights are entered from the keyboard when the program starts, and their sum must be 1. If the weights are entered correctly, generate all the possible solutions for each one of the max 10 individual channels, considering that the main channel has the debit between 2MBps and 40 MBps. Calculate the solutions with a precision of MBps. The 0 MBps will be assigned for those channels that cannot have the minimum of 2MBps. Display the solutions. The implementation can be both recursive and non-recursive.
3. Considering the example related to the problem of salesman, solve the problem considering the case when the cost matrix is not symmetrical with respect to the main diagonal and the travel costs in the same city can be different from zero (case of price offers).
4. In a software company, some data packets must be broadcasted to *n (<=7)* subsidiaries, *n* being entered from the keyboard. The subsidiaries are located at different distances and the transmission costs depend on those distances. The data packets must reach their destination, irrespective of the existence of a direct connection between the main company and a certain subsidiary (use other intermediate subsidiaries if necessary). The costs associated to each direct link are entered from the keyboard. Any subsidiary can be considered as the transmitting company packets. Determine and display the route that the data packets have to follow in order to minimize the total transmission cost. The implementation can be recursive or non-recursive. Use a *LUT* (Look Up Table) to match the values of the obtained solution with matching strings of characters with city names.

5. The optimum capital investment: an investor has a capital *C* and *n* offers; he has to invest it into f_i investment funds each of them being associated with a corresponding benefit b_i. Generate all the possible investing solutions. Indicate the solution that brings the maximum profit. Use backtracking and standard C for implementation. Display the chosen offers, benefits and maximum benefit.
6. Generate in a random mode not more than 100 integers with values less than 1000 into file. Using the "divide et impera" programming method, determine the minimum and maximum values from the file that will be displayed on the screen. Update the original file by adding the determined minimum and maximum values at the end on a new row.
7. Write a C program that generates maximum 200 (even number) integer numbers in a random mode with values not more than 500 and store it an input file. Using the „divide et impera" method determine the greatest common divider pair by pair from these integer numbers that are generated in a random mode. Display the pair values and the greatest common divider, row by row. These values will be stored in an output file in the same mode.
8. Calculate $\int_a^b 1/(1+x^2)dx$, using the trapezes method. The height of each considered trapeze must be smaller than $\varepsilon=0.0001$. The area of a trapezes defined by the points *(a,0), (b,0), (a,f(a))* and *(b,f(b))* is $(b-a)*(f(a)+f(b))/2$, where $f(x)=1/(1+x^2)$. The float values of a and b are read from the keyboard (a must be smaller or equal to b). Use the „divide et impera" method.
9. Write a C/C++ application that generates max 10 integer random values and stores them in an array. Check if another randomly generated value belongs to this array, using *_lsearch()* function.
10. Write a C/C++ application that finds and displays the images that represent the closest match to a certain searching key. The searching key is stored in each image's header and it is represented using a structure with the following fields: a filename (string of characters), a path (string of characters), an intensity resolution (integer value) and a dimension in bytes (integer value). Use a different searching technique for each key type. The headers are either stored into a file or the correspondent data is read from the keyboard.
11. Read from the keyboard a one-dimensional array of maximum 20 integer values. Using the binary search mechanism, determine if another value *a* (also read from the standard input) is part of the array. If so, determine and display all its prime factors.
12. Using a file that contains real ordered numbers, look for a value read from the keyboard. The searching method will rely on an iterative binary search, the recursive algorithm and the *bsearch()* library function. Display the values read from the file, the value to be searched and the position where it was found (if any).

4. Tehnici de sortare. Complexitatea algoritmilor

Sorting techniques. Algorithms Complexity

4.1. Obiective

- Înțelegerea tehnicilor de sortare.
- Familiarizarea cu metodele specifice de implementare a acestor tehnici în limbajul C/C++
- Funcții de bibliotecă pentru sortare: *qsort( )*
- Înțelegerea complexității algoritmilor

4.2. Objectives

- Understanding the sorting techniques.
- Using the sorting techniques in specific C/C++ implementations
- Library sorting functions: qsort( )
- Understanding the algorithms complexity

4.3. Breviar teoretic

Metode de sortare

Pentru un set de obiecte $S=\{a_1, a_2, ..., a_n\}$ sortarea constă în rearanjarea obiectelor într-o ordine specifică prin permutarea acestora, rezultând setul $S1=\{a_{k1}, a_{k2}, ..., a_{kn}\}$, astfel că dându-se o funcție de ordonare *f* (metoda de sortare) și o relație <, să obținem:

$$f(a_{k1}) < f(a_{k2}) <...< f(a_{kn})$$

Sortarea se face în raport cu o cheie asociată obiectelor și relația < .

Eficiența unei metode de sortare se evaluează în general prin următoarele operații:

- numărul de comparații ale cheii
- numărul de permutări ale unui obiect

Aceste operații sunt dependente de numărul de elemente din set și metoda de sortare folosită. O metodă avansată de sortare necesită până la $n\log_2(n)$ operații.

Există metode de sortare:

- simple – efectuează n^2 operații; sortarea se face "in situ", adică pe loc, tabloul inițial fiind modificat
- avansate – permit reducerea numărului de operații până la *n*log*(n)*, însă prezintă o complexitate mai mare

Biblioteca standard *(stdlib.h)* pune la dispoziție funcția:

```
void qsort(
    void *base, unsigned nelem, unsigned width,
    int(*fcmp)(const void *v1, const void *v2));
```

Funcția de comparare, definită de utilizator, trebuie să returneze pentru ordonare crescătoare :

< 0 dacă *v1 < *v2

0 dacă *v1 == *v2

> 0 dacă *v1 > *v2

Pentru ordonare descrescătoare valorile negativă și pozitivă returnată se vor inversa.

Tabloul unidimensional de ordonat poate să conțină și alte tipuri de date agregate (șiruri de caractere sau structuri).

Framewok-ul STL pune la dispozitie o metodă de sortare sort(…) bazată pe un algoritm numit introsort care combină sortarea quick cu sortarea heap și sortarea prin inserție pentru a oferi beneficiile fiecărui algoritm, evitând în același timp complexitatea de timp pentru cel mai rău caz.

Complexitatea algoritmilor

Cea mai obișnuită modalitate de a exprima complexitatea computațională este utilizarea notației big-O, care a fost introdusă de matematicianul german Paul Bachmann în 1892.

Notația Big-O constă din litera O urmată de o formulă care oferă o evaluare calitativă a timpului de rulare în funcție de dimensiunea problemei, notată în mod tradițional ca n (sau N).

De exemplu, complexitatea computațională a căutării liniare este O(n), iar complexitatea computațională a metodei de sortare prin selecție simplă este O(n2).

Compararea problemelor de tip n^2 și $n\log(n)$ ($n\log_2(n)$)

Diferența dintre O(n2) și O(n log(n)) este enormă pentru valori mari ale lui n, așa cum se arată în acest tabel (cu n, n2, nlog (n)):

O(n)	$O(n^2)$	O(n log(n))
10	100	33
100	10,000	664
1,000	1,000,000	9,966
10,000	100,000,000	132,877
100,000	10,000,000,000	1,660,877
1,000,000	1,000,000,000,000	19,931,569

Pe baza acestor numere, avantajul teoretic al utilizării sortării de tip merge față de sortarea prin selecție pe un tablou unidimensional de 1.000.000 de valori ar fi un factor mai mare de 50.000.

Clase standard de complexitate

Complexitatea unui algoritm particular general tinde să se încadreze într-una din următoarele clase standard de complexitate:

constant	O(1)	Accesarea primului element dintr-un vector
logaritmic	O(log n)	Căutare binară într-un vector sortat
liniar	O(n)	Suma elementelor dintr-un vector, căutare liniară
n log(n)	O(n log(n))	Sortare prin interclasare (merge)
pătratică	$O(n^2)$	Sortare prin selecție
cubică	$O(n^3)$	Algoritm naiv pentru înmulțirea matricilor
exponențială	$O(2^n)$	Soluția problemei turnurilor din Hanoi

4.4. Theoretical brief

Sorting methods

For a set of objects $S=\{a_1, a_2, ..., a_n\}$ sorting means rearranging the objects in a specific order using permutations, resulting the set $S1=\{a_{k1}, a_{k2}, ..., a_{kn}\}$, so that given an ordering function f (sorting method) and the following relation, $<$, that will be respected, being the imposed relation, so that:

$f(a_{k1}) < f(a_{k2}) <...< f(a_{kn})$

Sorting is done using a key associated with the objects and the associated < relation.

The efficiency of a sorting method is evaluated generally according to the following operations:

- the number of key comparisons
- the number of permutations.

These operations depend on the number of elements in the set. A good sorting technique uses down to $n\log_2(n)$ operations.

The sorting methods are:

- simple - n^2 operations are made; the sorting is done "in situ" (in the same place), the operations being performed in the initial array.
- advanced - allow reducing the number of operations down to $n\log(n)$. These methods' complexity is usually higher.

The standard library *(stdlib.h)* provides the function:

```
void qsort(
    void *base, unsigned nelem, unsigned width,
    int(*fcmp)(const void *v1, const void *v2));
```

The comparison function, defined by the programmer, has to return (for increasing order):

< 0 if *v1 < *v2

0 if *v1 == *v2

> 0 if *v1 > *v2

For decreasing order, the positive and negative value will be interchanged.

The one-dimensional array to be ordered can contain aggregate date (structures, arrays of characters, etc.).

The STL framework provides a sorting method *sort(…)* based on an algorithm named *introsort* which combines the *quick sorting* with *heap sorting* and *insertion sorting* for using the benefits offered by each individual algorithm, avoiding the time complexity of the worst-case scenario.

Algorithm Complexity

The most common way to express computational complexity is to use big-O notation, which was introduced by the German mathematician Paul Bachmann in 1892.

Big-O notation consists of the letter O followed by a formula that offers a qualitative assessment of running time as a function of the problem size, traditionally denoted as n (or N).

For example, the computational complexity of linear search is $O(n)$, and the computational complexity of a simple selection sort is $O(n^2)$.

Comparing n^2 and $n\log(n)$ ($n\log_2(n)$) problems

The difference between O(n2) and O(nlog (n)) is enormous for large values of n, as shown in this table (with n, n2, nlog (n)):

O(n)	$O(n^2)$	O(n log(n))
10	100	33
100	10,000	664
1,000	1,000,000	9,966
10,000	100,000,000	132,877
100,000	10,000,000,000	1,660,877
1,000,000	1,000,000,000,000	19,931,569

Based on these numbers, the theoretical advantage of using merge sort over selection sort on a one-dimensional array of 1,000,000 values would be a factor of more than 50,000.

Standard Complexity Classes

The complexity of a general particular algorithm tends to fall into one of the following standard complexity classes:

constant	O(1)	Finding the first element in a vector
logarithmic	O(log n)	Binary search in a sorted vector
linear	O(n)	Summing a vector; linear search
n log(n)	O(n log(n))	Merge sort
quadratic	$O(n^2)$	Selection sort
cubic	$O(n^3)$	Obvious algorithms for matrix multiplication
exponential	$O(2^n)$	Tower of Hanoi solution

4.5. Exemple/ Examples

Ex. 1 - Metode de sortare / Sorting methods (Algoritmi de sortare / Sorting algorithms)

```
/* The sorting functions used in a main program that reads the initial array and
   displays it after ordering.
*/
//*************************************************************************
// insertion sort
void sortIns(int *p, int n){
    int i, j, temp;
    for(i=1; i<n; i++)  {
        temp = p[i];
        for(j=i-1; j>=0; j--) {
            if(p[j] > temp)
                p[j+1] = p[j];//right shifting the element
            else
                break;
        }//end for
        p[j+1] = temp;
    }//end for
}//end sortIns
```

```
//*****************************************************************************
// selection sort
void sortSel(int *p, int n)
{
    int i, j, pozmin, temp;
    for(i=0; i<n; i++) {
        // the minimum value's position
        pozmin = i;
        for(j=i+1; j<n; j++) {
            if(p[pozmin] > p[j])
                pozmin = j;
        }//end for
        // interchange
        temp = p[pozmin];
        p[pozmin] = p[i];
        p[i] = temp;
    }//end for
}//end sortSel


//*****************************************************************************
// Bubble sort (without flag)
void sortBubble(int *p, int n)
{
    int i, j, temp;
    for(i=1; i<n; i++) {
        for(j=n-1; j>=i; j--) {
            if(p[j-1] > p[j]) {
                temp = p[j];
                p[j] = p[j-1];
                p[j-1] = temp;
            }//end if
        }//end for
    }//end for
}//end sortBubble


//*****************************************************************************
// Bubble sort (with flag, optimized do-while)
void sortBubbleDO(int *p, int n)
{
    int j, k=0,temp, flag;
    do{
        flag = 0;
        for(j=0; j<n-k-1; j++) {
            if(p[j] > p[j+1]) {
                temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
                flag = 1;
            }
        }//for
        k++;
    }while(flag != 0);
}//sortBubbleDO
```

```
//*************************************************************************
// quicksort (classic)
void quickSort(int *p, int prim, int ultim){
    int i, j, pivot, temp;
    i = prim; j = ultim;
    pivot = p[ultim];
    // partitioning
    do  {
        while(p[i] < pivot)
            i++;
        while(p[j] > pivot)
            j--;
        if(i < j) {
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
        }//end if
        if(i <= j) {
            j--;
            i++;
        }//end if
    }while(i < j);//end do-while

    // recursive call
    if(prim < j) quickSort(p, prim, j);
    if(i < ultim) quickSort(p, i, ultim);
}//end quickSort
```

Ex. 2 - Generare și sortare șir cu cronometrare / Timed array generation and sorting

```
/* An array of values is ordered using quick sort and bubble sort. The time
   intervals required for various code sequences are displayed.
*/
#include <iostream>
using namespace std;
#include <time.h>//for certain compilers

void init(int numere[ ], int);
void afis(int numere[ ], int);
void bubble(int numere[ ], int);
void quick(int numere[ ], int);
int comparare(const void* arg1, const void* arg2);

int main( ) {
    int dim, * numere;
    cout << "\nNumber of elements: ";
    cin >> dim;
    numere = new (nothrow) int[dim];
    if(numere == 0)  {cout<<"\nMemory allocation error. \n";exit(1);}
    init(numere, dim);
    afis(numere, dim);
    bubble(numere, dim);
    afis(numere, dim);
    init(numere, dim);
    afis(numere, dim);
```

```
    quick(numere, dim);
    afis(numere, dim);

    delete[ ]numere;
}//main

void init(int numere[ ], int dim){
    clock_t start, end;
    double dif;
    time_t t;

    start = clock();
    srand((unsigned)time(&t));//srand((unsigned)time(0));

    for (int i = 0; i < dim; i++) {
        numere[i] = rand();
    }

    end = clock();
    dif = (double)(end - start);
    cout<< "\nGenerating the numbers took " << dif << " (clock ticks),  "
        << dif / CLOCKS_PER_SEC << " (seconds)" << endl;
}//init

void afis(int numere[ ], int dim){
    clock_t start, end;
    double dif;

    start = clock();
    for (int i = 0; i < dim; i++)
        cout << " " << numere[i];

    end = clock();
    dif = (double)(end - start);
    cout<< "\nDisplaying the numbers took " << dif << " (clock ticks),  "
        << dif / CLOCKS_PER_SEC << " (seconds)" << endl;
}//afis

void bubble(int numere[ ], int dim)
{
    clock_t start, end;
    double dif;
    int aux, ok;

    start = clock();
    do {
        ok = 1;
        for (int i = 0; i < dim - 1; i++) {
            if (numere[i] > numere[i + 1]) {
                aux = numere[i];
                numere[i] = numere[i + 1];
                numere[i + 1] = aux;
                ok = 0;
            }
        }
    } while (ok == 0);
```

```
    end = clock();
    dif = (double)(end - start);
    cout<< "\nBubble sort took " << dif << " (clock ticks),  "
        << dif / CLOCKS_PER_SEC << " (seconds)" << endl;
}//bubble

void quick(int numere[ ], int dim) {
    clock_t start, end;
    double dif;

    start = clock();
    qsort(numere, dim, sizeof(numere[0]), comparare);

    end = clock();
    dif = (double)(end - start);
    cout<< "\nQuick sort took " << dif << " (clock ticks),  "
        << dif / CLOCKS_PER_SEC << " (seconds)" << endl;
}//quick

int comparare(const void* arg1, const void* arg2){
    return (*(int*)arg1 - *(int*)arg2);
}//comparare
```

Ex. 3 - Sortare șir de cuvinte prin metoda bulelor / Sort array of words using bubble sort

```
/* Bubble sort, arrays of characters
*/

#include <iostream>
using namespace std;

int fcmp(const void* s1, const void* s2);
void bubbleSort(const char** names, const int size);

int main( )
{
    int dimc;
    const char* tabc[ ] = { "abc", "xyz", "acd", "axyz", "bc", "eltcti" };
    dimc = sizeof(tabc) / sizeof(tabc[0]);
    bubbleSort(tabc, dimc); //sorting

    cout << "\nSorted strings: ";
    for (int i = 0; i < dimc; i++)
        cout << tabc[i] << ", ";
    cout << endl;
}//main

void bubbleSort(const char** names, const int size) {
    int k = 0;//optimize the end of the sorted array
    bool swapped;
    do {
        swapped = false;
        for (int i = 0; i < size-k-1; ++i) {
            if (fcmp(names[i], names[i + 1]) > 0) {
                const char* temp = names[i]; //address interchange
```

```
                names[i] = names[i + 1];
                names[i + 1] = temp;
                swapped = true;
            }
        }
        k++;
    } while (swapped);
}// bubbleSort

int fcmp(const void* s1, const void* s2){
    //return((int)strlen((const char *)s1) -
    //    (int)strlen((const char *)s2));//by length
    return _stricmp((const char*)s1, (const char*)s2);//ignore case
}//fcmp
```

Ex. 4 - Sortare șir de structuri cu qsort / Sorting arrays of structures using qsort

```
/* qsort( ) for sorting structures */

/* A) no dynamic allocation
 */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const int dim = 12;

struct Datac {
    int an;
    int luna;
    int zi;
};
struct Pers{
    char numep[dim];
    struct Datac datan;
};

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

int main( ){
    struct Pers angaj[ ] = {
        {"x3", {2001, 6,15}},
        {"x2", {1960, 5, 5}},
        {"x3", {2002, 9, 22}},
        {"x2", {1961, 12, 31}},
        {"x1", {1980, 2, 28}}
    };

    int  i;
    int nang = sizeof(angaj) / sizeof(angaj[0]);

    // sorting function call
    qsort(angaj, nang, sizeof(angaj[0]), cmp);
    printf("Datele sortate :\n");
```

```
    for (i = 0; i < nang; i++)
        printf("\t%s, %d, %d, %d\n", angaj[i].numep, angaj[i].datan.an,
                angaj[i].datan.luna, angaj[i].datan.zi);
}//main

int cmp(const void* aa, const void* bb) {
    Pers* a = (Pers*)aa;
    Pers* b = (Pers*)bb;
    if ((a->datan).an > (b->datan).an) return 1;
    if ((a->datan).an < (b->datan.an)) return -1;
    //an egal, verificam luna
    if ((a->datan).luna > (b->datan).luna) return 1;
    if ((a->datan).luna < (b->datan).luna) return -1;
    //luna egala, verificam ziua
    if ((a->datan).zi > (b->datan).zi) return 1;
    if ((a->datan).zi < (b->datan).zi) return -1;
    //ziua egala, verificam numele
    if ((strcmp(a->numep, b->numep) > 0)) return 1;
    if ((strcmp(a->numep, b->numep) < 0)) return -1;
    //totul egal
    return 0;
}//cmp

/* B) Dynamic memory allocation, implementation with header file
 */

//DatePers.h
struct Data_calend {
    int ziua;
    char luna[11];
    int anul;
};

struct Date_pers {
    char nume[16];
    char prenume[20];
    long cod;
    struct Data_calend data_nast;
    void citDatePers(Date_pers*);//methods inside struct
    void afisDatePers(Date_pers*);
};
//No typdef necessary in C++ for declaring and using structures

void Date_pers:: afisDatePers(Date_pers* p) {
    cout << "\n Name: " << p->nume;
    cout << "\n Surname: " << p->prenume;
    cout << "\n Code: " << p->cod;
    cout << "\n Birth day: ";
    cout << "\n\t Day: " << (p->data_nast).ziua;
    cout << "\n\t Month: " << (p->data_nast).luna;
    cout << "\n\t Year: " << (p->data_nast).anul;
}// afisDatePers

void Date_pers:: citDatePers(Date_pers* p){
    cout << "\nName: ";
    cin >> p->nume;
```

```
    cout << "\nSurname: ";
    cin >> p->prenume;
    cout << "\nCode: ";
    cin >> p->cod;
    cout << "\nBirth Day: ";
    cout << "\n\tDay: ";
    cin >> (p->data_nast).ziua;
    cout << "\n\tMonth: ";
    cin >> (p->data_nast).luna;
    cout << "\n\tYear: ";
    cin >> (p->data_nast).anul;
}// citDatePers

//DatePers.cpp - main() zone
#include <iostream>
#include "DatePers.h"

using namespace std;

int comp_cod(const void* a, const void* b);
int comp_nume_prenume_an(const void* a, const void* b);

int main( ){
    struct Date_pers* dp, man;
    int i, n;
    cout << "\nNumber of employees: "; cin >> n;
    if (n <= 0) {
        cout << "\n Invalid number(negative or zero) !" << endl; exit(1);
    }
    //cout << "length Data_calend: \n" << sizeof(Data_calend) << endl;
    //cout << "length Date_pers: \n" << sizeof(Date_pers) << endl;
    if (!(dp = new (nothrow) Date_pers[n])) { //init pointer dp
        cout << " Allocation failed!";
        exit(1);
    }
    cout << "\n Enter your personal data :";
    for (i = 0; i < n; i++) {
        cout << "\n Person:" << i+1;
        man.citDatePers(dp + i);
    }
    cout << "\nPersons introduced: ";
    for (i = 0; i < n; i++) man.afisDatePers(dp + i);
    cout << "\n\nPersons sorted in descending order by code: ";
    qsort(dp, n, sizeof(Date_pers), comp_cod);
    for (i = 0; i < n; i++) man.afisDatePers(dp + i);
    cout << "\n\nPersons sorted in ascending order by last name and year: ";
    qsort(dp, n, sizeof(Date_pers), comp_nume_prenume_an);
    for (i = 0; i < n; i++) man.afisDatePers(dp + i);
    delete[ ] dp; // free the allocated memory
}//main

int comp_cod(const void* a, const void* b) {
    Date_pers* pa = (Date_pers*)a;
    Date_pers* pb = (Date_pers*)b;
    return (pb->cod - pa->cod); //descendent
}// comp_cod
```

```
int comp_nume_prenume_an(const void* a, const void* b) {
    int fl_nume, fl_pre;
    Date_pers* pa = (Date_pers*)a;
    Date_pers* pb = (Date_pers*)b;
    if ((fl_nume = strcmp(pa->nume, pb->nume)) == 0)
        if ((fl_pre = strcmp(pa->prenume, pb->prenume)) == 0)
            //ascending by nume, prenume and an
            return ((pa->data_nast).anul - (pb->data_nast).anul);
        else
            return fl_pre;//difference prenume
    return fl_nume;//difference nume
}// comp_nume_prenume_an

/* C) Dynamic allocation C++, qsort. ADT implementation. */
//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;
    Calendar_Data birth_date;
    void readPersonalData(); //methods no params.
    void displayPersonalData();
};

void Personal_Data:: displayPersonalData() {
    cout << "\nLast Name: " << last_name;
    cout << "\nFirst Name: " << first_name;
    cout << "\nCode: " << code;
    cout << "\nBirth Date: ";
    cout << "\n\tDay: " << birth_date.day;
    cout << "\n\tMonth: " << birth_date.month;
    cout << "\n\tYear: " << birth_date.year;
} // displayPersonalData

void Personal_Data::readPersonalData() {
    cout << "\nLast Name: ";
    cin >> last_name;
    cout << "\nFirst Name: ";
    cin >> first_name;
    cout << "\nCode: ";
    cin >> code;
    cout << "\nBirth Date: ";
    cout << "\n\tDay: ";
    cin >> birth_date.day;
    cout << "\n\tMonth: ";
    cin >> birth_date.month;
    cout << "\n\tYear: ";
    cin >> birth_date.year;
} // readPersonalData
```

```
//DatePers.cpp - 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( ) {
    Personal_Data* pd;
    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;
        (pd+i)->readPersonalData();
    }
    cout << "\nEntered Persons: ";
    for (i = 0; i < n; i++) (pd + i)->displayPersonalData();

    cout << "\n\nPersons sorted by code: ";
    qsort(pd, n, sizeof(Personal_Data), compareCode);
    for (i = 0; i < n; i++) (pd + i)->displayPersonalData();

    cout << "\n\nPersons sorted by name and year: ";
    qsort(pd, n, sizeof(Personal_Data), compareNameYear);
    for (i = 0; i < n; i++) (pd + i)->displayPersonalData();

    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
```

4.6. Lucru individual

1. Implementați metoda bulelor (Bubble-Sort) care folosește un indicator flag și optimizează ciclul interior. Se cere atât scrierea funcției, cât și partea de program care face citirea și afișarea șirului inițial și a celui ordonat.
2. Modificați programul care exemplifică metoda de sortare rapidă explicita (quickSort - clasic) așa încât să ordoneze șirul inițial în ordine descrescătoare. Comparați timpul de răspuns pentru mai multe valori *n* cu funcția *qsort( )* din biblioteca standard.
3. Folosiți funcțiile de bibliotecă pentru sortări (*qsort( )*) pentru a aranja un tablou unidimensional de înregistrări cu nume, prenume, cod numeric personal, data angajării după două câmpuri la alegere (un exemplu ar fi: crescător după nume și descrescător după data angajării).
4. Scrieți o aplicație C/C++ în care plecând de la două tablouri (unidimensionale) de numere naturale să se obțină un al treilea tablou care să conțină toate elementele tablourilor sursă fără a se repeta, aranjate în ordine crescătoare.
5. Completați codul problemei date ca exemplu (2) cu alte metode de sortare (sortarea prin selecție, sortarea shell, etc.). Citiți de la tastatură numărul de elemente al șirurilor de valori si apoi trimiteți-l ca parametru la fiecare funcție. Comparați timpii de lucru ai fiecărui algoritm folosind tablouri cu dimensiune mare. Analizați în acest caz partea de afisare. Introduceți un timer (1-5 sec.) între sortări în *main( ).* Gestiunea timpului puteți să o faceți folosind funcții din *<chrono>* la nivel de nanosecunde (vezi activ. suplimentară).
6. Citiți de la tastatură *m* elemente de tip întreg într-un tablou unidimensional și o valoare întreagă *n<m*. Impărțiți tabloul citit în două sub-tablori astfel:
   a) primul sub-tablou va contine primele *n* elemente din tabloul inițial
   b) al doilea sub-tablou va contine restul elementelor din tabloul initial.
   Să se realizeze următoarele operații:
   - să se ordoneze crescător cele două sub-tablouri
   - să se sorteze tabloul inițial, prin interclasarea celor două sub-tablori ordonate (merge-sort).
7. Să se scrie un program care permite sortarea unui stoc de calculatoare. Acestea să se reprezinte în program ca o structură formată din caracteristicile calculatoarelor (nume (șir caractere), tip de procesor (șir caractere), frecvența de tact (l*ong int*), dimensiunea memoriei RAM (*in*t), preț (*float*). Sortarea se va face, la alegerea utilizatorului, după: preț, memorie, tact sau tip de procesor. Folosiți, de preferință, funcția de bibliotecă pentru sortări *qsort( )* sau o altă metodă la alegere. Sortați apoi considerând un câmp șir de caractere și unul numeric. Afișați rezultatele.
8. Preluați din două fișiere două tablouri unidimensionale ce conțin valori reale. Generați un al treilea tablou care să conțină toate valorile din cele doua tablouri și toate valorile obținute prin medierea valorilor de pe aceeași poziție din cele doua tablouri inițiale. Dacă numărul de elemente ale tablourilor diferă, media se va face considerând valoarea 0.0 pentru elementele lipsă. Ordonați al treilea tablou și numărați câte valori neunice sunt în șir.
9. Generați în mod aleatoriu un tablou de maxim 200 valori întregi, valori nu mai mari de 100. Determinați și afișati valoarea minima, mediană și maximă generată, sortand elementele printr-o metodă la alegere. Determinați valoarea medie și comparați această valoare cu cea mediană (afisați diferența). Verificați dacă valoarea medie este în tabloul inițial generat.
10. Generați printr-un mecanism aleatoriu un tablou de maxim 200 de valori reale (prin două tablouri de aceiași dimensiune, primul fiind partea întreagă (nu mai mare de 100), al doilea partea fracționară (limitată la 20 ca întreg ce devine .20 fracționar), tabloul real

fiind obținut prin combinarea părții întregi și fracționare. Afișati tablourile generate, cel real obținut.
Sortați folosind functia *qsort( )* tabloul real și afisați rezultatul obținut.

11. Alocați în mod dinamic un tablou de *n* numere întregi, care vor fi citite și afișate. Citiți o valoare cheie de la intrarea standard. Folosind funcția *_lfind( )/lfind( )* căutați și afișați toate pozițiile în care această cheie se găsește în tabloul citit. Tratați cazul în care sunt valori multiple, sau valoarea nu e în tablou. Folosind funcția *qsort( )* sortați apoi acest tablou, pe care îl afișați. Căutați folosind funcția *bsearch()* aceiași valoare în tabloul sortat și afișati poziția ei. Analizați modul de acțiune al funcției *bsearch()(adresa/poziția valorii returnate dacă sunt valori multiple).*

Activitate suplimentară

Analizați mecanismul de contorizare a timpului și mecanismul de analiză a complexității algoritmilor pentru a efectua sortări folosind funcții avansate de timp cum ar fi cele din biblioteca *<chrono>*. (*https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9)*

4.7. Individual work

1. Implement the Bubble-Sort method using a flag indicator and optimize the inner loop. Write the function that orders an array of integer values read from the keyboard. Display the original and the sorted arrays.
2. Modify the program that implements the classic quickSort algorithm so that it will sort the initial array of values in decreasing order. Compare the response time for multiple *n* values with the *qsort( )* function from the standard library.
3. Use the library function "*qsort( )*" for sorting an array of records that contain a name, a surname, a personal identification code and an employment date. The sorting is based on the data stored in some specific fields (like name, employment date, etc.).
4. Write a C/C++ application that reads from the keyboard 2 arrays of positive numbers. The program determines a 3-rd array that contains all the elements in the initial arrays, increasingly ordered. The elements that have the same value must appear a single time in the ordered array.
5. Add some new sorting methods to the code presented in the examples area (example 2 - selection sort, shell sort, etc.). Read from the keyboard the number of elements from the array and pass it as parameter to each sorting function. Compare the working times scored by each implemented sorting algorithm using huge arrays. Use comments for displaying method. Enter a timer (1-5 seconds) among the sorting methods in *main( ).* Time management can be done using functions from *<chrono>* at the nanosecond level (see additional activity).
6. Read from the keyboard a one-dimensional array of m integers and an integer value n<m. Split the array in 2 subarrays as it follows:
   a) the first array will contain the first n elements of the initial array
   b) the second subarray will contain the rest elements of the initial array
   Realize the following operations:
   - sort increasingly the subarrays
   - sort the initial array, by interlacing the sorted subarrays (merge-sort)
7. Write a program that sorts a stock of computers, represented in the program as objects created from a structure that stores the computers' characteristics: name (string of characters), processor type (string of characters), frequency (long int), RAM memory (int), price (float). The sorting is performed, as specified by the user, by price, memory amount, frequency, or processor type. Use the qsort( ) library function or any other

sorting technique. Next sort the data using a character string and a numerical field. Display the results.

8. Read from two files two one dimensional arrays composed of real values. Generate a third array that contains all the values from the initial arrays and all the values obtained by calculating the average of the corresponding numbers. If the initial arrays have different numbers of values, the average is calculated between the existent values and 0.0. Order the last array and count the number of non-unique elements.
9. Generate in a random mode a maximum of 200 smaller than 100 integer numbers and store them into an array. Determine and display the minimum, the median and the maximum value and sort the array to accomplish that. Determine the average value and display the difference between it and the calculated median. Check if the average value is part of the initial array.
10. Generate 200 random float values. Store the integer parts (not bigger than 100) into an array. The fractional parts (limited at 20 as an integer value representing a 0.20 fractional part) are stored into another array with the same size as the first one. The initial values re-calculated by combining the elements stored into the previously described arrays are stored into another vector. Display all the arrays. Use the qsort( ) library function for storing the float values and display the final result.
11. Allocate an array of n integer numbers that will be read from the keyboard. Read another value that represents a searching key. Using the _lfind( )/lfind( ) function, search and display all the positions in which the key appears in the initial array. Handle the cases in which the key is found on more than 1 position or is not found at all. Sort the array using the qsort( ) function and display the result. Use the bsearch( ) function for searching the same key in the ordered array and print on the screen its positions. Analyze the mode of action of the bsearch() function (address / position of the returned value if there are multiple values).

Additional activity

Use advanced timing functions (such as those from the *<chrono>* library) and complexity analysis mechanism for measuring the sorting algorithms' effectiveness.
(*https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9*)

5. Clase, obiecte, membrii clasei

Classes, objects, class members

5.1. Obiective

- Înțelegerea teoretică și practică a noțiunilor de clasă, obiect, membru al unei clase, specificatori de acces (vizibilitate).
- Scrierea de programe simple, după modelul programării orientate pe obiecte (POO - OOP) folosind ADT (Abstract Data Type), care exemplifică noțiunile menționate mai sus.

5.2. Objectives

- Theoretical and practical undestanding of classes, objects, class members and access specifiers.
- Writing simple programs, following the Object Oriented Programming (OOP) model using ADT (Abstract Data Type), that exemplify the aspects mentioned above.

5.3. Breviar teoretic

Programarea orientată pe obiecte implică definirea *tipurilor abstracte de date, ADT,* în vederea definirii obiectelor, a mesajelor, precum și a mecanismului de moștenire. Noțiunile de bază ale OOP se referă în principal la *definirea claselor de obiecte*, a *moștenirii,* a *polimorfismului, încapsulării, abstractizării* și ulterior *a excepțiilor.*

Clasele C++ reprezintă tipuri noi de date, care conțin metode (funcții membre) și atribute (variabile din clasă), un fel de „șablon" folosit pentru a defini metodele și atributele unui obiect anume, creat după modelul clasei.

Procesul de creare al unui obiect se numește *instanțiere,* caz în care pentru fiecare atribut se vor preciza niște valori concrete (explicit sau implicit).

Metodele unei clase pot să fie definite în interiorul ei (metode *inclass*) sau în afară, caz în care este necesară indicarea clasei prin operatorul rezoluție sau scop (::).

Accesul la atributele și metodele membre ale unei clase se poate controla prin utilizarea *specificatorilor de acces (vizibilitate)*: ***public, private, protected***, care definesc mecanismul de *încapsulare*.

Metodele membre *constructor* sunt metode care sunt apelate în momentul instanțierii unor obiecte, ele nu au specificat nici un fel de tip returnat, au numele identic cu cel al clasei din care fac parte. Rolul constructorilor este de a inițializa atributele din acea clasă și de a aloca spațiu de memorie corespunzător obiectului instanțiat și atributelor folosite în cadrul clasei.

Destructorul clasei este o metodă care, analog cu constructorul, poate fi recunoscută prin faptul că are același nume cu clasa din care face parte, este precedat de caracterul ~ și are rolul de a elibera spațiul de memorie alocat pentru obiectul instanțiat și a atributelor folosite în cadrul clasei.

Trecerea de la programarea structurată la cea prin abstractizarea datelor si apoi obiectuală în C++ se face generalizând noțiunile legate de tipuri utilizator definite prin structuri și reuniuni, astfel încât metodele care în general asigură mecanismul de interfațare (public) aparțin claselor ce accesează atributele ce definesc mecanismul de implementare (private).

5.4. Theoretical brief

Object-oriented programming involves defining *abstract data types, ADTs,* in order to define objects, messages, and the mechanism of inheritance. The basic notions of OOP refer mainly to *defining classes*, *inheritance, polymorphism, encapsulation, abstraction* and subsequently *exceptions.*

C++ classes represent new data types containing methods (member functions) and attributes (class variables), a kind of "template" that are used to define the methods and attributes of a particular object, created according to the class model.

The process of creating an object is called *instantiation,* in which case for each attribute some concrete values will be specified (explicitly or implicitly).

The methods of a class can be defined inside it (*inclass methods*) or outside, in which case it is necessary to indicate the class by the resolution or scope operator (::).

The access to a class's member attributes and methods can be controlled by using *access* (*visibility) specifiers*: ***public, private, protected***, which define the *encapsulation* mechanism.

Member *constructor* methods are called when instantiating objects, they don't specify any returned type, have the same name as the class to which they belong. The constructors initialize the class attributes and allocate the necessary memory.

The *class destructor* is a method that, analogous to the constructor, can be recognized by having the same name as the class to which it belongs, is preceeded by the ~ character. It usually frees the memory allocated for the object and for its attributes.

The transition from structured programming to abstract data type and next object oriented programming in C++ is made by generalizing the notions related to user types defined by structures and unions, so that the methods that generally provide the interface mechanism (public) belong to classes that access the attributes that define the implementation mechanism (private).

5.5. Exemple/ Examples

Ex. 1 - Operații elementare cu un dreptunghi / Elementary operations with a Rectangle

```
/* The Rectangle class implemented for performing elementary operations with
   rectangular geometric shapes
*/
// Rectangle.h - class Rectangle with header
class Rectangle {
    //private members by default
    int height;
    int width;
public:
    //public members
Rectangle(int h = 10, int w = 10);//explicit cons. with all implicit parameters
    int det_area(void);
    void setHeight(int);//setter
    void setWidth(int);//setter
    ~Rectangle(void); //explicit  destructor
};//class_Rectangle

// explicit constructor implementation
Rectangle:: Rectangle(int h, int w){
    height = h;
    width = w;
}//constructor
```

```
//explicit destructor implementation
Rectangle:: ~Rectangle() { cout << "\nDestructor called..."; }//destructor

int Rectangle:: det_area( ) {
    return height * width;
}//det_area

//setter
void Rectangle:: setHeight(int init_height){
    height = init_height;
}//setHeight

//setter
void Rectangle:: setWidth(int init_width){
    width = init_width;
}//setWidth

// Rectangle.cpp - main()
#include <iostream>
using namespace std;
#include "Rectangle.h"

int main( ) {
// objects of Rectangle type
    Rectangle rect, square;//instantiated with implicit values
    cout << "\nObjects(access with operator .): \n";
    cout << "\tImplicit values: \n";
    cout << "\t\tRectangle area: " << rect.det_area( ) << endl;
    cout << "\t\tSquare area: " << square.det_area( ) << endl;
    cout << "\tAny value: \n";
    rect.setHeight(12);
    rect.setWidth(8);
    cout << "\t\tRectangle area : " << rect.det_area( ) << endl;
    square.setHeight(8);
    square.setWidth(8);
    cout << "\t\tSquare area : " << square.det_area( );
    cout << "\n.................................\n\n";
// pointer to Rectangle
    Rectangle *point;
    cout << "\nPointers to objects (access with  operator ->):";
    point = new (nothrow) Rectangle;//inst. with implicit values
    cout << "\n\tImplicit values: \n";
    cout << "\t\tArea rectangle(square):" << point->det_area( ) << endl;
    cout << "\tAny values: \n";
    point->setHeight(12);
    point->setWidth(9);
    cout << "\t\tArea (rectangle)square: " << point->det_area( ) << endl;
    delete point;
    cout << "\n.................................\n\n";
// reference Rectangle
    Rectangle &ref_square = square;
    ref_square.setHeight(17);
    ref_square.setWidth(17);
    cout << "\t\t Square area  using references: " << ref_square.det_area( );
    cout << "\n.................................\n\n";
}//main
```

Ex. 2 - Operații elementare cu un punct 2D / Elementary operations with a 2D point

```
/* Point class  without header*/

#include <iostream>
using namespace std;

class Point {
    //attributes
    int x, y;
public :
    //explicit constructor with all implicit parameters
    Point(int a=0, int b=0) {
        x=a; y=b;
    }

    //setters and getters
    void setX(int a){
        x=a;
    }
    int getX(){
        return x;
    }
    void setY(int b){
        y=b;
    }
    int getY(){
        return y;
    }
};//class_Point

int main( ){
Point p1;     // equivalent with Point p1(0,0);
    cout << "x = " << p1.getX() << ", y = " << p1.getY();
    Point p2(10);   // equivalent with Point p2(10,0);
    Point p3(10,10);
    p1.setX(20);
    p1.setY(30);
    //display the values associated to the x and y attributes for p1, p2, p3
    //...
}//main
Requirement: Add and test a method for calculating the distance between two points.
```

Ex. 3 - Clasa Complex - operații de bază / Complex class - basic operations

```
/* class Complex */

//Complex.h
class Complex{
    //Complex real and imag parts as private attributes
    double re, im;
public:
    //explicit constructor with all implicit parameters
    Complex(double x=0.0, double y=0.0) {
        re = x;
        im = y;
    }
```

```
    double modulus(){
        return sqrt(re*re + im*im);
    }

//to implement:
    //double phase();

    //setters and getters
    void setRe(double real){ re=real; }
    double getRe(){ return re; }

    void setIm(double imaginar){ im=imaginar; }
    double getIm(){ return im; }

    //addition
    void ad_complex(Complex b){
        re += b.re;
        im +=b.im;
    }//ad_complex

    //subtraction
    void sc_complex(Complex b){
        //... to implement
    }//sc_complex
};//class

//Complex.cpp - main
#include <iostream>
using namespace std;
#include "Complex.h"

int main( ){
    Complex object_1,object_2;
    double aux;
    cout <<"\nThe real part of object 1: ";
    cin >> aux;
    object_1.setRe(aux);
    cout <<"The imaginary part of object 1: ";
    cin >> aux;
    obiect_1.setIm(aux);

    //reading the values for object 2
    //....

    object_1.ad_complex(object_2);//the result is stored in object_1
    cout << "\nThe real part of the sum: " <<object_1.getRe();
    cout << "\nThe imaginary part of the sum: " <<object_1.getIm();

    //call the subtraction method
    //...
}//main
```

Requirement: add and test a method for *phase* calculation and implement it outside the class.

Ex. 4 - Metode ce returnează obiecte / Methods that return objects

```
/* class Comp lex (partial implementation); methods that return objects */

...

class Complex {
    double re, im;
public:
    void setRe(double x);
    void setIm(double x);
    double getRe();
    double getIm();
    Complex sum(Complex c);
    Complex difference(Complex c);
    Complex multi(Complex c);
    Complex div(Complex c);
}; //class

Complex Complex:: sum(Complex c) {
    Complex rez;
    rez.re=(re+c.re);
    rez.im=(im+c.im);
    return rez;
}//sum

...
int main( ){
    Complex object_1,object_2;
    ...
    Complex sum_object = object_1.sum(object_2);//implicit copy_constructor
    ...
}//main
```

5.6. Lucru individual

1. Scrieți o aplicație C/C++ care într-un fișier antet definește o structură de date numită *Scerc* care conține raza ca variabilă întreagă. Definiți două funcții pentru a calcula aria și circumferința, având ca parametru un obiect de tip *Scerc*. Într-un program C/C++, declarați două obiecte *c1, c2* de tip *Scerc* și introduceți valorile razei de la tastatură. Aceleași cerințe vor fi implementate în aceeași aplicație într-un alt fișier antet folosind o clasă numită *Cerc* cu atributul raza privată de tip *int*, clasă care va conține, pe lângă metodele de calcul a ariei și perimetrului, un constructor explicit cu parametru, un destructor. Introduceți în clasă o metodă de tip accesor, *getRaza()*, care permite accesul la raza datelor private și pe care o puteți utiliza pentru a afișa în *main( )* raza obiectelor. Definiți, de asemenea, o funcție globală *void displayR(...)* care utilizează un obiect *Cerc* și afișează raza acestuia. Definiți mai multe obiecte *Cerc* cu valori de rază introduse de la tastatură, cu care să apelați metodele definite, folosind obiecte instanțiate, pointeri la obiecte, referințe la obiecte. Afișați raza cu și cu funcția globală definită.
2. Să se definească o clasă numită MyString într-un fișier numit strClass.h care să fie compusă din metodele specifice care efectuează următoarele operații pe șiruri de caractere:
   - determină lungimea șirului primit la intrare;
   - determină ultima poziție de apariție a unui anumit caracter din șirul de intrare ;
   - returnează șirul primit la intrare, scris cu caractere majuscule;

- returnează șirul primit la intrare, scris cu caractere minuscule;
- returnează numărul de apariții ale unui anumit caracter din șirul primit;

Să se scrie programul care citește de la tastatură un șir de maxim 10 caractere și care, pe baza clasei implementate anterior, efectuează asupra șirului de intrare operațiile definite în cadrul clasei.

3. Să se scrie programul care implementează clasa *Numar* cu un atribut de tip *int* val și care, în cadrul funcției *main( )*, declară un obiect de tipul clasei și apoi un pointer la acesta, prin intermediul cărora se va afișa pe ecran rezultatul adunării a două numere de tip *Numar* cu valorile preluate de la tastatură în cadrul unor obiecte *Numar*. Implementați metoda *int suma_nr(Numar)* care realizează suma în cadrul clasei și o returnează ca un *int*, metoda care însumează cele două obiecte (curent si parametru). Implementați metoda în cadrul clasei și altă metoda cu același scop, dar nume diferit, în afara clasei.
4. Să se definească o clasă care implementează metodele:
   - int plus(int x, int y), care returnează suma valorilor primite la apelul metodei;
   - int minus(int x, int y), care returnează diferența valorilor primite la apelul metodei;
   - int inmultit(int x, int y), care returnează produsul valorilor primite la apelul metodei;
   - float impartit(int x, int y), care returnează catul valorilor primite la apelul metodei;

   Să se scrie aplicația care utilizează această clasă. Considerați și cazul în care în cadrul clasei aveți atributele de tip int x și y, caz în care metodele nu vor mai avea parametrii.

   Observație: În cazul împărțirii, trebuie verificată validitatea operației (împărțitor diferit de zero). În cazul în care operația este imposibilă, trebuie afișat un mesaj de eroare.

5. Să se creeze o clasă care să modeleze numerele complexe. Scrieți un program care utilizează această clasă și defineste două obiecte afișând caracteristicile obiectelor și rezultatele operațiilor definite (Folosiți exemplul 3 cu rezultat in obiectul curent).
6. Pornind de la clasa Complex, (exemplul 4), să se implementeze operațiile de adunare, scădere, înmulțire și împărțire pentru numere complexe prin metode corespunzătoare implementate la alegere în clasă și/sau în afara ei. Testați aceste metode prin instanțierea unor obiecte. Metodele vor returna obiecte de tip Complex și în main( ) vor fi afișate rezultatele folosind metode accesor.
7. Să se scrie un program care implementează clasa Aritmetica cu două atribute a și b de tip numeric (int, float sau double) și metode setter și getter adecvate. Implementați metoda suma() în interiorul clasei și metoda diferenta() ce apartine de asemenea clasei, dar e definită în afara clasei, metode care vor fi apelate prin intermediul unui obiect al clasei Aritmetica. În funcția principală main() instanțiați trei obiecte de tip Aritmetica. Modificați atributele a și b la fiecare obiect în parte folsind metodele de tip setter. Aplicați asupra lor operațiile de adunare și scădere pe care le-ați implementat prin metodele suma() și diferenta(). Metodele returnează valorile numerice corespunzătoare operației folosind cele două atribute ale clasei valori ce le veti afișa în main( ). La fiecare grup de operații adunare/scadere afișați valorile atributelor obiectului folosind metodele de tip getter.
8. Declarați o clasă Fractie care are două atribute întregi de tip private a și b pentru numărător și numitor. Definiți două metode de tip set(), respectiv două de tip get() pentru atributele clasei. Instanțiați două obiecte de tip *Fractie* și afișați atributele inițiale și cele obținute după folosirea metodelor *set()*. Definiți o metodă *simplifica()* apelată cu un obiect pentru care au fost utilizate metodele de tip *set(),* care determină divizorii numitorului și numărătorului, îi afișează și realizează simplificarea fracției, afisând în metodă și rezultatul obținut (noua fracție, numărător_simplificat/numitor_simplificat).

5.7. Individual work

1. Write a C/C++ application that defines, in a header file, a data structure named Scircle, which contains the radius as an integer variable. Define two functions to calculate the area and the circumference, having as a parameter an object of type Scircle. In a C/C++ program, declare two objects c1, c2 of type Scircle and enter radius values from the keyboard.
   The same requirements will be implemented in the same application in another header file using a class called Circle with the private radius attribute of type int. The class will implement two methods for calculating the circle's area and perimeter, an explicit constructor with 1 parameter, and a destructor.
   Add to the class an accessor type method, getRadius(), which returns the value of the private attribute radius. Define a global method named void displayR(...) that uses a Circle object and displays its radius.
   Define multiple Circle objects with radius values entered from the keyboard, and call the specific methods using the instantiated objects, pointers to the objects, and object references. Display radius supplementary with the global function defined.
2. Declare a class called MyString and store it in a file named strClass.h. The methods in the class perform the following tasks:
   - determine the length of the array of characters received as parameter;
   - determine the index of the last occurrence of a certain character in the array;
   - return the array of characters received as parameter, all letters being transformed into capital letters;
   - return the number of occurrences of a certain character;

   Write a program that reads from the keyboard an array of maximum 10 characters and performs all the operations implemented in the class defined at problem 2.
3. Write a program that implements a class called Number with an attribute of int type, val. In function main(), declare two Number objects and a pointer to one object of this type, used to call a method int sum_nr(Number) from the class. The method will calculate and return the sum of two int numbers read from the keyboard, associated to the instantiated objects (current, parameter). Implement the method inside / outside the class.
4. Define a class that implements the following methods:
   - int plus(int x, int y), which returns the sum of x and y;
   - int minus(int x, int y), which returns the difference between x and y;
   - int multiply(int x, int y), which returns the result of x multiplied by y;
   - float divide(int x, int y), which returns the quotient of x and y;

   Write the application that uses this class. Also consider the case that the class contains two int attributes, x and y, and the class methods have no parameters.

   Remark: prevent the division of a number by 0. If this situation occurs, display an error message.
5. Define a class that manages complex numbers. Write the program that uses this class considering two objects and displays the object's characteristics and the results of the defined operations.
6. Starting with the Complex class presented in ex.4, implement the addition, subtraction, multiplication, and division of complex numbers. The methods can be implemented inside or outside the class. Test the defined methods by using them upon some created objects. The methods will return Complex objects that will be displayed in main( ) using the getter methods.

7. Write a program that implements a class named Arithmetics that has two numeric (int, float or double) attributes a and b. The class will provide setter and getter methods for the attributes. Inside the class, implement a method named sum( ). The method difference( ) also belongs to the class and is implemented outside the class. The methods will be called using Arithmetics objects.
   Create 3 objects instantiated from the Arithmetics class and use the defined methods. The results will be displayed in main( ). For each addition/subtraction, display the values of the object attributes using the getter methods.
8. Declare a class named Fraction that has two private integer attributes a and b representing a fraction's numerator and denominator. Define two setter and getter methods for the class's attributes. Create two Fraction instances and display the initial attributes and the ones established after using the setter methods. Define a method named simplify( ) that determines and displays all the common dividers of the numerator and denominator, simplifies the fraction and prints in the method the final result (new fraction, numerator simplified/ denominator simplified).

6. Accesul la membrii unei clase

The access to a class members

6.1. Obiective

- Înțelegerea teoretică și practică a specificatorilor de acces (vizibilitate), a modalităților de acces la membrii privați sau protejați ai unei clase, a accesului prin intermediul obiectelor (instanțelor), pointerilor sau adreselor.

6.2. Objectives

- Theoretical and practical understanding of the access specifiers, of private and protected members' access, of using the objects (instances), pointers and addresses for accessing the class's members.

6.3. Breviar teoretic

Utilizarea *specificatorilor de acces (vizibilitate)*: *public, private, protected* servește la controlul accesului la atributele și metodele membre ale unei clase.

În cazul în care nu este menționat nici un specificator de acces, compilatorul consideră implicit toate atributele și metodele din clasă ca fiind *private.*

Semnificația acestor specificatori în C++ este:

public: atributele și metodele care urmează sunt vizibile (accesibile) din orice zonă a programului;

private: atributele și metodele care urmează pot fi accesate doar de către membrii clasei;

protected: atributele și metodele care urmează pot fi accesate în cadrul clasei curente sau în clasele derivate din ea.

De obicei se preferă ca atributele dintr-o anumită clasă să fie *private*, pentru un mai bun control al acestora (ele pot fi modificate doar prin intermediul unor metode publice din clasă) și deci pentru o protecție a lor.

Metodele și atributele publice ne dau modul de interfațare al clasei, iar cele private ne dau modul de implementare a clasei.

Metodele și atributele protected sunt folosite la moștenire.

Metodele publice care au ca scop modificarea unor atribute *private* sau *protected* poartă numele de *metode mutator/setter*.

Metodele publice care au ca scop returnarea unor atribute *private* sau *protected* poartă numele de *metode accesor/getter*.

Compilatoarele moderne generează la cerere metode *setter* și *getter* pentru fiecare din atributele *private* ale unei clase în mod individual, având ca și nume *set/getNumeAtribut*, prima literă din numele atributului fiind *uppercase*.

6.4. Theoretical brief

The use of *access (visibility) specifiers*: *public, private, protected* serves at controlling the access to a class attributes and methods.

If no access specifier is mentioned, the compiler considers all attributes and methods in the class as *private.*

The significance of these specifiers in C++ is:

> *public:* the following attributes and methods are visible (accessible) from any area of the program;
>
> *private:* the following attributes and methods can only be accessed by class members;
>
> *protected:* The following attributes and methods can be accessed within the current class or in the classes derived from it.

It is usually preferred that attributes of a class be *private*, for a better control and protection (they can only be modified by public class methods).

Public methods and attributes form a class interface and private ones contribute to the class implementation.

Protected methods and attributes are used in inheritance.

Public methods aimed at modifying *private* or *protected* attributes are called *mutator/setter methods*.

Public methods aimed at returning *private* or *protected* attributes are called *accessor/getter methods*.

Current IDEs generate on demand individual *setter* and *getter* methods for each of the *private* attributes, named *set/getAttributeName*, the first letter of the attribute name being mentioned in *uppercase*.

6.5. Exemple/ Examples

Ex. 1 - Acces la atribute publice și private / Accessing public and private attributes

```
/* Accessing the private and public attributes of a class using getter and setter
   methods
*/
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
const int Max=20;

class Test1 {
    int x;
public:
    char sir[Max]; //not recommended to be public
    Test1() {//explicit constructor without parameters
        x = 0;
        strcpy(sir, "Text implicit.");
    }
```

```
//setter methods:
    //method that changes the private x attribute
    void setX(int a) {
        x = a;
    }
    //method that changes the public attribute (obsolete)
    void setSir(char * new_sir) {
        strcpy(sir, new_sir);
    }

//getter methods
    //method that returns the value of the private attribute x
    int getX() {
        return x;
    }
    //method that returns the value of the public attribute sir
    char* getSir() { //sir is accessible without this method
        return sir;
    }
};//class

int main( ) {
    Test1 ob1; //instantiation
    int a;
    char s[Max];

    //reading and setting the attributes
    cout << "\nEnter the value of the \"int\" attribute: ";
    cin >> a;
    ob1.setX(a); //setter call

    cout << "\nEnter the value of the \"char array\" attribute: ";
    cin >> s;
    ob1.setSir(s); //(obsolete) setter call

    //displaying the attributes
    //getter for the private attribute
    cout << "\nThe \"int\" attribute returned by the getter is: "
        << ob1.getX()
        << endl;
    //the public attribute accessed directly
    cout << "\nThe  \"char array\" attribute accessed directly is: "
        << ob1.sir
        << endl;
    //getter for the public attribute
    cout << "\nThe \"char array\" attribute returned by the getter is: "
        << ob1.getSir()
        << endl;
}//main
```

Requirement:
Consider the *sir* attribute from the class private and modify the program accordingly.

Ex. 2 - Specificatorul de acces implicit / Implicit access specifier

```
/* The implicit access specifiers
*/
#include <iostream>
using namespace std;

class Test2{
    int x;
    Test2( ){
        x=0;
        cout<<"\nExplicit empty constructor.";
    }
};//class

int main( ){
    //Test2 ob1; //instantiation impossible, the constructor is private!
    int a;
    cout<<"\nEnter an \"int\" value: ";
    cin>>a;
    //impossible private attribute access
    //ob1.x = a;
}//main
```

Ex. 3 - Exemplu didactic, clasa Matrix / Didactical example, Matrix class

```
/* Didactical example - Matrix class with private attributes accessed using public
   methods
*/
//Matrix.h

const int Max1=10;
const int Max2=10;

class Matrix{

    //attributes
    int matrix[Max1][Max2], dim1, dim2;
    //private method declaration (will return an element)
    int getElement(int row, int column);

public:
    //explicit constructor with parameters
    Matrix(int dim1, int dim2){
        //local variables
        int i, j;
        this->dim1=dim1;
        this->dim2=dim2;
        cout<<"\nEnter the matrix elements: ";
        for(i=0; i<dim1; i++){
            for(j=0; j<dim2; j++){
                cout<<"\nmatrix["<<i<<"]["<<j<<"] = ";
                cin>>matrix[i][j];
            }
        }
    }
```

```
    //matrix display method
    void displayMatrix(){
        //local variables
        int i, j;
        cout<<"\nThe matrix elements are: ";
        for(i=0; i<dim1; i++){
            cout<<"\n";
            for(j=0; j<dim2; j++){
                cout<<getElement(i, j)<<" ";
            }
        }
        cout<<endl;
    }//displayMatrix
    //method declaration (will display a column)
    void displayColumn(int col);
};//class

//external implementation of the methods declared inside the class

void Matrix::displayColumn(int col){
    if(col<0||col >=dim2){ //validation
        cout<<"\nThe column "<<col<<" is outside the matrix!\n";
    }
    else{
        cout<<"\nThe elements in column "<<col<<": ";
        for(int i=0; i<dim1; i++){
            cout<<getElement(i, col)<<" ";
        }
    }
}// displayColumn

int Matrix:: getElement (int row, int column){
    return matrix[row][column];
}// getElement

//main zone
#include <iostream>
using namespace std;
#include "Matrix.h"

int main( ){
    int dim1,dim2;
    cout<<"\nEnter the matrix dimensions: (>0, <=10):\n";
    cin>>dim1;
    cin>>dim2;

    Matrix m1(dim1,dim2); //instantiation
    m1.displayMatrix();
    int c;
    cout<<"\nEnter a column number: < " <<  dim2;
    cin>>c;
    m1.displayColumn(c);
    //impossible access to a private member
    //m1. getElement(0, 0);
}//main
```

Ex. 4 - Obiecte, adrese, pointeri și referințe / Objects, addresses, pointers and references

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

class Test4{
    //all the class members are public (like a structure)
public:
    float f;//recomended to be private

    Test4( ){
        cout<<"\nExplicit empty constructor.";
        //implicit value for the class attribute
        f = 0.0f;
    }

    float getF(){//f is not private, the getter is obsolete
        return f;
    }
};//class

int main(  ){
    Test4 ob1; //instantiation
    float a;

    //direct access to the public attribute
    cout<<"\nThe value of the \"float\" class attribute: "<<ob1.f;

    cout<<"\nEnter a new \"float\" value: ";
    cin>>a;

    ob1.f = a; //modifying the attribute
    cout<<"\nThe new value of the \"float\" class attribute: "<<ob1.f;

// Using a pointer to a class object
    Test4 *pob; //declaration
    pob = new(nothrow) Test4; //initialization

    cout<<"\n Enter a \"float\" value: ";
    cin>>a;

    pob->f = a; //accessing a public member using a pointer

    //displaying the attribute using the getter method
    cout<<"\nThe value of the \"float\" attribute: "<<pob->getF();
    delete pob;
// Using a reference to an object
    Test4 &rob1 = ob1;

    //displaying the attribute using the reference
    cout<<"\nThe \"float\" attribute: "<<rob1.getF();
}//main
```

Requirement: Make the *f* attribute private and define a setter method with an appropriate name. Check the functionality.

Ex. 5 - Clasa complex - modul si faza / Complex class - modulus and phase

```
//complex numbers
//Complex.h

class Complex {
    // private members
    int re;
    int im;
    double modul;//not recommended
    double faza;//not recommended
public:
    // public members
    void setRe(int r) {    re = r; }
    void setIm(int i) {    im = i; }

    int getRe( ) { return re; }
    int getIm( ) { return im; }

    void det_modul_faza( ) {//should be split in 2 different methods
        modul = sqrt((double)re*re + (double)im*im);
        faza = atan2((double)re, (double)im);//radians
    }

    double getModul(void) {
        return modul;
    }

    double getFaza(void) {
        return faza;
    }
};//class

//main zone
#include <iostream>
using namespace std;
#include "Complex.h"

int main() {
    Complex c1;
    c1.setRe(12);
    c1.setIm(5);
    c1.det_modul_faza( );

    cout << "Object data: " << endl;
    cout << "\tRe = " << c1.getRe() << ", Im=" << c1.getIm() << endl;
    cout << "\tModul: " << c1.getModul() << endl;
    cout << "\tPhase (radians): " << c1.getFaza( ) << "\tPhase (degrees):"
        << c1.getFaza( ) * 180 / 3.14 << endl;
}//main
```

Requirement: Remove the module and phase attributes and adapt the methods to determine these parameters independently.

Ex. 6 - Validare Cod Numeric Personal / Personal numerical code validation

```
//a) personal numerical code for persons born before 2000 – didactical example

//Person.h
//#include <time.h>

class Person {
    // private members
    char nume[16];
    char prenume[24];
    char cnp[14];
/*personal numerical code (CNP) format :
S YY MM DD ... (another 6 digits); 13 digits in total
*/
public:
    // public members
    void setNume(char *n);
    void setPrenume(char *p);
    int validCnp(char *c);

    char* getNume(void) { return nume; }
    char* getPrenume(void) { return prenume; }
    char* getCnp(void) { return cnp; }

    char get_gender(void);
    int get_an_nast(void);
    int get_luna_nast(void);
    int get_zi_nast(void);
    int get_varsta(void);
};//class

void Person:: setNume(char *n) {
    if (n != 0)//validation should be done in main()
        strncpy(nume, n, 15);
    else
        strcpy(nume, "not specified");
}

void Person:: setPrenume(char *p) {
    if (p != 0) strncpy(prenume, p, 23);
    else strcpy(prenume, "not specified");
}

int Person::validCnp(char* c) {
    char buf[3];
    int n;
    if (c != 0) {
        cnp[13] = '\0';
        if (strlen(c) != 13)
            return 1;
        if (c[0] != '1' && c[0] != '2')
            return 2;
        strncpy(buf, c + 1, 2);
        buf[2] = '\0';
        n = atoi(buf);
```

```
        if (n > 99) return 3;
        strncpy(buf, c + 3, 2);
        buf[2] = '\0';
        n = atoi(buf);
        if (n <= 0 || n > 12)
            return 4;
        strncpy(buf, c + 5, 2);
        buf[2] = '\0';
        n = atoi(buf);
        if (n <= 0 || n > 31)
            return 5;
        strcpy(cnp, c);
        return 0;
    }
    else return -1;
}

char Person:: get_gender(void) {
    if (cnp[0] == '1') return 'M';
    if (cnp[0] == '2') return 'F';
    return 'X';
}

int Person:: get_an_nast(void) {
    char buf[3];
    strncpy(buf, cnp + 1, 2);
    buf[2] = '\0';
    return(1900 + atoi(buf));
}

int Person:: get_luna_nast(void) {
    char buf[3];
    strncpy(buf, cnp + 3, 2);
    buf[2] = '\0';
    return(atoi(buf));
}

int Person:: get_zi_nast(void) {
    return((cnp[5] - '0') * 10 + (cnp[6] - '0'));
}

int Person:: get_varsta(void) {
    struct tm *newTime;
    time_t szClock;
    time(&szClock);
    newTime = localtime(&szClock);
    int an_c = 1900 + newTime->tm_year;
    int an_n = get_an_nast();
    int n = an_c - an_n;
    int lu_c = newTime->tm_mon + 1;
    int lu_n = get_luna_nast();
    if (lu_c < lu_n) n--;
    else {
        if (lu_c == lu_n) {
            int zi_c = newTime->tm_mday;
            int zi_n = get_zi_nast();
```

```
            if (zi_c < zi_n)
                n--;
        }
    }
    return n;
}

// Person.cpp - main zone
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

#include "Person.h"

int main( ) {
    Person p1;
    char aux_string[30];
    cout << "\nEnter the name: ";
    cin >> aux_string;
    p1.setNume(aux_string);
    cout << "\nEnter the given name: ";
    cin >> aux_string;
    p1.setPrenume(aux_string);
    cout << "\nEnter personal numerical code (CNP): ";
    cin >> aux_string;//1890403120671
    p1.validCnp(aux_string);

    cout << "Object data: " << endl;
    cout << "\tName: " << p1.getNume() << ", given name: " << p1.getPrenume()
        << ", CNP: " << p1.getCnp() << endl;
    cout << "\tSex: " << p1.get_gender() << endl;
    cout << "\tBirthdate: " << p1.get_an_nast()
        << "/" << p1.get_luna_nast() << "/" << p1.get_zi_nast() << endl;
    cout << "\tAge: " << p1.get_varsta() << endl;
}//main

//*****************************************************************************
//b)same problem, variant with VC++ library functions

//Person.h
//#include <time.h>
const int dim_sir = 24;

class Person {
    char nume[dim_sir];
    char prenume[dim_sir];
    char cnp[14];
public:
    void setNume(char *n);
    void setPrenume(char *p);
    int setValidCnp(char *c);

    char* getNume(void) {
        return nume;
    }
```

```
    char* getPrenume(void) {
        return prenume;
    }

    char* getCnp(void) {
        return cnp;
    }

    char get_gender(void);
    int get_an_nast(void);
    int get_luna_nast(void);
    int get_zi_nast(void);
    int get_varsta(void);
};//class

void Person:: setNume(char *n)//validation should be done in main()
{
    if (n != 0)
        strncpy_s(nume, n, dim_sir-1);
    else
        strcpy_s(nume, "Necunoscut");
}

void Person:: setPrenume(char *p) {
    if (p != 0)
        strncpy_s(prenume, p, dim_sir-1);
    else
        strcpy_s(prenume, "Necunoscut");
}

int Person:: setValidCnp(char *c) {
    char buf[3];
    int n;
    if (c != 0) {
        if (strlen(c) != 13) //length
            return 1;
        if (c[0] != '1' && c[0] != '2') //sex
            return 2;
        strncpy_s(buf, c + 1, 2); //year
        buf[2] = '\0';
        n = atoi(buf);
        if (n > 99)
            return 3;
        strncpy_s(buf, c + 3, 2);//month
        buf[2] = '\0';
        n = atoi(buf);
        if (n == 0 || n > 12)
            return 4;
        strncpy_s(buf, c + 5, 2);//day
        buf[2] = '\0';
        n = atoi(buf);
        if (n == 0 || n > 31)
            return 5;
        strcpy_s(cnp, c);
        return 0;
    }
```

```
    else
        return -1;
}

char Person:: get_gender(void) {
    if (cnp[0] == '1')
        return 'M';
    if (cnp[0] == '2')
        return 'F';
    return 'X';
}

int Person:: get_an_nast(void) {
    char buf[3];
    strncpy_s(buf, cnp + 1, 2);
    buf[2] = '\0';
    return(1900 + atoi(buf));
}

int Person:: get_luna_nast(void) {
    char buf[3];
    strncpy_s(buf, cnp + 3, 2);
    buf[2] = '\0';
    return(atoi(buf));
}
int Person:: get_zi_nast(void) {
    return((cnp[5] - '0') * 10 + (cnp[6] - '0'));
}

int Person:: get_varsta(void) {
    struct tm newTime;
    time_t szClock;

    time(&szClock);
    localtime_s(&newTime, &szClock);

    int an_c = 1900 + newTime.tm_year;
    int an_n = get_an_nast();
    int n = an_c - an_n;

    int lu_c = newTime.tm_mon + 1;
    int lu_n = get_luna_nast();
    if (lu_c < lu_n)
        n--;
    else {
        if (lu_c == lu_n) {
            int zi_c = newTime.tm_mday;
            int zi_n = get_zi_nast();
            if (zi_c < zi_n)
                n--;
        }
    }
    return n;
}
```

```
//Person.cpp - main zone
#include <iostream>
using namespace std;

#include "Person.h"

int main( ) {
    Person p1;
    char aux_string[dim_sir];

    cout << "\nEnter name: ";
    cin >> aux_string;
    p1.setNume(aux_string);

    cout << "\nEnter given name: ";
    //cin >> aux_string;
    cin.ignore();
    gets_s(aux_string, dim_sir);//reading with whitespaces
    p1.setPrenume(aux_string);

    cout << "\nEnter CNP: ";
    cin >> aux_string;//1890403120671
    int t_cnp=p1.setValidCnp(aux_string);

    switch (t_cnp) {
        case 0:
            cout << "\nCNP valid\n";
            cout << "\nObject data: " << endl;
            cout << "\tName: " << p1.getNume() << ", given name: "
                << p1.getPrenume() << ", CNP: " << p1.getCnp() << endl;
            cout << "\tGender: " << p1.get_gender() << endl;
            cout << "\tBirthdate: " <<p1.get_an_nast() << "/" << p1.get_luna_nast()
                << "/" << p1.get_zi_nast() << endl;
            cout << "\tAge: " << p1.get_varsta() << endl; break;
        case 1:
            cout << "\nWrong CNP length"; break;
        case 2:
            cout << "\nWrong sex indicator"; break;
        case 3:
            cout << "\nWrong birth year"; break;
        case 4:
            cout << "\nWrong birth month"; break;
        case 5:
            cout << "\nWrong birth day"; break;
        default:
            cout << "\nCNP problems..."; break;
    }
}//main
```

6.6. Lucru individual

1. Să se scrie o aplicație C++ care implementează o clasă numită *PilotF1*. Clasa definește atributele *private nume* (tablou unidimensional de caractere), *echipa* (tablou unidimensional de caractere), *varsta* (*int*), *record* (*int*), *nr_pole_position* (*int*). Ca și membri *public*, clasa conține metode accesor/getter și mutator/setter distincte pentru fiecare din atributele clasei.
   În funcția *main( )*, să se creeze 3 instanțe distincte ale clasei *PilotF1* și să se folosească metodele mutator/setter pentru a inițializa datele din fiecare obiect cu informația corespunzătoare citită de la tastatură. Folosind metodele accesor/getter, să se afișeze toate datele pilotului cu cel mai bun record.
2. Să se modifice exemplul 2 astfel încât codul să poată fi lansat în execuție considerând atributul clasei *private* și metode *public* get/set adecvate. În *main()* instanțiați un obiect din clasă care va fi modificat și apoi accesat, afisând rezultatul.
3. Pornind de la exemplul care tratează lucrul cu matrice, considerați ca și atribute *private* matricea dată printr-un pointer dublu pentru a aloca un tablou de pointeri către liniile matricii (sau, pointer simplu la alocarea dinamică contiguă a matricii), și două atribute *private* de tip *int* pentru numărul de linii și coloane. Constructorul cu doi parametrii va aloca dinamic spațiu pentru matrice inițializând elementele cu 0. Implementați un destructor explicit pentru a elibera spațiul alocat. Definiți o metodă publică *void setElement(int l, int c, int v)* care va seta un element cu valoarea *v*, de pe linia *l* și coloana *c*. Declarați o funcție globală *void readMatrix(Matrix m)* care va avea ca și parametru o matrice instanțiată la care se va valida in *main()* ca dimensiunile să fie corecte, >=0 si <dim=10, și care va citi elementele matricii de la tastatură (va folosi și metoda publică *setElement()*). Metoda din clasă, *displayMatrix(),* va fi definită și ea ca și o funcție globală la fel ca și *readMatrix(...)*, numai că matricea va fi transferată prin referință în absența constructorului de copiere, considerând metoda *getElement()* ca și metodă *public*. Metoda de afișare a unei coloane, nu va valida numărul coloanei, coloana fiind validată în *main( )* înainte de apelul metodei. Completați codul scris cu metode specifice pentru:
   - afișarea elementelor de pe diagonala secundară a matricei, dacă matricea este pătratică; în caz contrar se afișează un mesaj corespunzător;
   - afișarea elementelor de sub diagonala principală în aceleași condiții ca mai sus;
   - afișarea unei matrice de dimensiunea celei inițiale ale cărei elemente pot avea valori de 0 (dacă elementul corespunzător este mai mic decât o valoare în prealabil citită de prag) sau 1 (în caz contrar), și verifică dacă matricea este rară, >= 67% din elemente sunt zero);
4. Refaceți aplicația anterioară în care considerați metodele *getElement()* și *setElement()* *private* iar funcțiile globale *readMatrix(...)* și *displayMatrix(...)* vor fi definite ca și metode membre *public* în cadrul clasei.
5. Să se scrie o clasă care are ca atribut *private* un câmp de tip *data*, definit într-o structură externă clasei (*zi – int, luna – int, an - int*). Clasa conține metode mutator/setter și accesor/getter (*public*) pentru acces la informația privată, ca și structură de tip *data*, pentru fiecare câmp din structură. In clasă se mai află doua metode *public* care:
   - testează validitatea datei stocate;
   - scrie într-un fișier toate datele până la anul curent care preced (cronologic) data stocată în clasă considerând doar atributul *an* ca fiind variabil; dacă data succede data curentă se va scrie un mesaj adecvat.

   În functia main( ), după instanțierea clasei și citirea de la tastatură a componentelor unei date, să se apeleze metodele membre și apoi să se verifice rezultatele obținute.
6. Folosiți accesul prin pointeri și prin referință în rezolvarea problemelor 1...5 la accesul la membrii clasei.

7. Scrieți o aplicație C++ care definește o clasă numită Triunghi. Clasa conține ca și atribute private de tip int laturile triunghiului a, b și c, un constructor cu parametri și metode adecvate de tip setter și getter pentru fiecare atribut în parte. Clasa va conține metode care vor calcula aria și perimetrul formei. Scrieți o metodă distinctă care va afișa un anumit mesaj dacă triunghiul este dreptunghic. Introduceți în clasă o metodă public care va determina dacă valorile lui a, b și c ale unui obiect formează un triunghi. Metoda va fi apelată în main() după instanțierea unui obiect și în metodele setter, dacă sunt modificate laturile unui obiect existent. Setter-ii vor păstra valoarea laturii vechi dacă cea nouă nu este o latură validă.
   Analizați și cazul în care validarea este înainte de a instanția un obiect. Preluați de la tastatură 3 valori int pentru cele 3 laturi, cu confirmare prin reintroducerea valorilor, dacă este necesar.
8. Să se scrie clasa Seif, cu atributele private cifru (pin - șir de caractere, 4/6 cifre int) și suma de tip double. Definiți metodele private getSuma() și setSuma( ) și metodele public puneInSeif( ) și scoateDinSeif( ) cu care să accesați suma de bani care se află în seif. Metoda puneInSeif( ) poate apela getSuma() și setSuma( ), metoda scoateDinSeif( ) poate apela getSuma( ) și setSuma( ). Instanțiați obiecte din clasa Seif, iar metodele puneInSeif( ) și scoateDinSeif( ) vor putea accesa suma doar dacă parametrul de tip cifru utilizat corespunde obiectului instanțiat. În caz de diferență de cifru, se va da un mesaj.
9. Dezvoltați aplicația prezentată în exemplul 6 varianta a) sau b) prin:
   - utilizarea valorilor returnate de metoda setValidCnp( ) pentru a valida suplimentar (luna și ziua) CNP-ul în main( ) funcție de an bisect sau nu și numărul lunii.
   - permiterea introducerii de coduri CNP care încep cu alte cifre decât 1 și 2, cu analiza semnificației noilor valori (5 - masculin, 6 – feminin, 7- CD). Modificați validarea anului, întrucât valoarea e specifică persoanelor născute după 2000.

6.7. Individual work

1. Write a C++ application that implements a class called *F1Pilot*. The class defines the *private* attributes *name* (one dimensional array of characters), *team* (one dimensional array of characters), *age* (*int*), *best_time* (*int*) and *pole_position_no*(*int*). As *public* members, the class contains mutator/setter and accessor/getter methods for each of the class's attributes.
   In function *main( )*, create 3 different instances of the *F1Pilot* class and use the setter methods for initializing each object's data with the corresponding information read from the keyboard. Using the getter methods, display all the data related to the pilot that has the best time.
2. Modify example 2 so that the code can be launched in execution considering the *private* class attribute and appropriate *public* get/set methods. In *main(),* instantiate an object from the class that will be modified and then accessed, displaying also the result.
3. Starting from the example that uses *Matrix* class, consider as *private* attributes the matrix, represented as a double pointer, and two *private* attributes of type *int* for the number of *rows* and *columns*. The constructor with 2 parameters will dynamically allocate the memory for storing the matrix and will initialize the elements with 0. Implement an explicit destructor that frees the memory occupied by the matrix. Define a *public* method *void setElement(int r, int c, int v)* that will set an element with the value *v*, on row *r* and column *c*. Declare a global function *void readMatrix(Matrix m)* that will have as parameter a matrix object with the dimensions validated in *main()* (>=0 and <=10). The function will read the elements of the matrix from the keyboard (it will also use the public method *setElement()*). The method *displayMatrix()* implemented in the example will be defined as a global function just like *readMatrix(),* only that the matrix will be transferred by reference in the absence of the copy constructor*,* and will use the

getElement() public method. The method that displays a column will not check the column number, the column being validated in *main()* before calling the method.
Add specific methods for:
- displaying the elements from the main diagonal of the matrix, in case the matrix is square; if not, display a significant message.
- displaying the elements below the main diagonal under the same conditions as above;
- displaying a matrix that has identical dimensions with the original matrix populated with 0's (if the corresponding element is less than a previously read value of the threshold) or 1 (otherwise). Check if the matrix is rare (> = 67% of the elements are zero);

4. Reimplement the previous application by considering the *getElement()* and *setElement()* as private methods and the *readMatrix(…)* and *displayMatrix(…)*, previously defined as global functions, will be defined as public member methods implemented inside the class.
5. Write a C++ class that has as *private* attribute a *date* field. The *date* is defined as a structure declared outside the class and it contains the fields *day – int, month – int, year – int*. The class contains *public* accessor/getter and mutator/setter methods that can use the *private* information as *date* structure, for all fields from the structure. The class also contains two *public* methods that:
   - test the validity of the stored date.
   - write into a file all the dates till the current *year* that precede chronologically the class stored *date* based only on the *year* attribute as a modified attribute; if the date succeeds the current date, an appropriate message will be written.

   In the main( ) function, after instantiating the class and after reading from the keyboard all the components of a date, call the member methods and then verify the obtained results.
6. Use pointers and references for solving the problems 1...5.
7. Write a C++ application that defines a class called *Triangle*. The class contains as *private int* attributes the triangle's sides *a, b* and *c*, and in the public zone a constructor with parameters and adequate setter and getter methods for each attribute (individual methods). The class will contain methods that will calculate the shape's area and perimeter. Write a distinct method that will print a specific message if the triangle contains a 90 degree angle. Implement a *public* method that will determine whether the values of *a, b* and *c* of an object form a triangle. The method will be called in *main()* after instantiating an object. Use the setters for changing the attributes. The setters will keep the old side values if the new ones are not valid.
   Also analyze the case in which the sides are validated before instantiating an object. Read from the keyboard 3 *int* values for the 3 sides. Re-enter the values, if necessary.
8. Write a class named *Safe* that has as *private* attributes the *cipher (*pin, 4/6 *int* digits*)* and the amount of *money* of *double* type. Implement the *private* methods *getMoney()* and *setMoney()*. The *public* methods *putInSafe( )* and *getFromSafe( )* will call the previous *private* methods only if the *cipher* sent as parameter matches the value stored inside the class. Display a message if the cipher is not correct.
9. Modify the application from example 6, variant a) or b):
   - using the values returned by the method *setValidCnp( )*, check if the CNP *month* and *day* are valid if the year is leap.
   - process CNP codes that start with numbers different than 1 or 2 and analyze the significance of the new values (5 - male, 6 – female, 7 - CD). Modify the year processing, since the CNP refers to persons born after 2000.

7. Constructori. Destructori. Tablouri de obiecte.

Constructors. Destructors. Object arrays.

7.1. Obiective

- Înțelegerea teoretică și practică a noțiunilor de constructori, destructor, tablouri de obiecte.
- Scrierea de programe simple, după modelul programării prin abstractizare a datelor, care exemplifică noțiunile menționate mai sus.
- Înțelegerea noțiunii de liste de inițializare

7.2. Objectives

- Understanding the constructors, destructor and object arrays.
- Writing some simple OOP programs that use the notions mentioned above.
- Understanding the initializer list notion

7.3. Breviar teoretic

Tipuri de constructori, după numărul de parametri

Constructorul implicit vid este adăugat automat de compilator dacă nu există un constructor explicit și este echivalentul unui constructor public fără parametri și fără instrucțiuni.

Constructorii expliciți pot fi de mai multe feluri**:**

- ***Vizi***: nu conțin instrucțiuni, utili pentru a schimba vizibilitatea celui implicit și pentru a preveni crearea de obiecte prin constructori generați automat, care nu sunt în controlul programatorului.
- ***Fară parametri***: care inițializează anumite atribute din clasă (uneori pe toate).
- ***Cu parametri***: inițializează anumite atribute din clasă și/sau efectuează anumite operații conform argumentelor primite ca parametrii.

Începând cu *C++1y* s-a introdus posibilitatea de a păstra un constructor capabil a fi apelat fără parametri, ca și constructor *default*. Poate fi implementat cu o listă vidă de parametri sau cu o listă de parametri având toate valorile implicite:

```
ClassName( ) = default;
```

sau

```
ClassName(lista_param_toti_impliciti ) = default;
```

unde, *ClassName* e numele clasei.

De asemenea, în C++1y, un constructor poate apela un alt constructor definit, folosind ***liste de inițializare***.

Constructori speciali (copiere, mutare, conversie)

Un tip special de constructor este ***constructorul de copiere*** (copy constructor), a cărui menire este de a crea copia unui obiect, într-un obiect nou. Constructorii de copiere sunt necesari doar în cazul în care clasa respectivă conține atribute de tip pointeri ce necesită alocare dinamică pentru a se rezerva spațiu. Altfel, compilatorul oferă un constructor de copiere implicit.

Forma generală a copy constructorului este:

```
ClassName(const ClassName &Ob){…};
```

Începând cu versiunile *C++1y* a apărut ca și constructor special și ***move constructorul***, care permite definirea unui constructor care să dea adresa obiectului de copiat fără a face o copie explicită a lui. Acesta e definit cu ajutorul referințelor *r-value* și *semantici move* (*semantici move* - implică pointarea către un obiect care deja există în memorie) de forma:

```
ClassName (ClassName && obj) {...} ;
```

Compilatorul este responsabil pentru eliberarea resurselor alocate obiectului sursă.

Există și ***constructori expliciți de conversie***.

C++1y a adăugat capacitatea de a apela un constructor de alt constructor folosind o listă de inițializare.

Datele membre (atributele), pot fi inițializate:

- în cadrul constructorilor (prin atribuiri)
- în ***listele de inițializare*** (initializer list) care se află între antetul constructorului și corp:

```
Point::Point(int a, int b) : x(a), y(b) {
    // corp constructor
    // x și y sunt atribute de clasă, a și b sunt parametri
}
```

În acest caz, procesul de initializare este realizat înainte de a executa corpul constructorului.

Datele membre *const* și referințele către datele membre trebuie inițializate folosind liste de inițializare. Motivul se datorează faptului că nu le este alocată memorie suplimentară, ele fiind echivalate în tabele de simboluri.

Destructorul

Destructorul este o metodă specială care:

- nu are parametri.
- poate fi definit dacă se dorește efectuarea anumitor operații în momentul distrugerii obiectului. În cazul atributelor de tip pointeri alocați dinamic, se definește un destructor care realizează eliberarea acestor pointeri.
- poate să lipsească, caz în care se apelează de către compilator un destructor implicit.

Pointerul this

Pointerul ***this*** pointează spre membrii instanței curente a clasei (este apelabil doar din interiorul clasei!!!). Este util mai ales când în interiorul unei metode dintr-o clasă este necesar să facem distincție între variabile ce aparțin unor obiecte diferite, pointerul *this* indicând întotdeauna obiectul curent.

Aspecte legate de constructori și inițializarea tablourilor

Pentru a crea ***tablouri de obiecte***, avem nevoie de un constructor fără parametri (implicit vid sau unul explicit echivalent, inclusiv *explicit default*). Dacă se folosește un constructor cu parametri, fiecare obiect din tablou poate fi inițializat indicând o listă de inițializare.

7.4. Theoretical brief

Types of constructors by number of parameters

The ***implicit empty constructor*** is added automatically by the compiler when an explicit constructor does not exist, and it is the equivalent of a public parameterless constructor without any instructions.

The ***explicit constructors*** can be classified as:

- ***Empty***: they do not contain instructions and are useful for changing the visibility of the default one or to prevent the creation of objects through the use of the implicit constructor, whose implementation is not controlled by the programmer.
- ***Parameterless***: they initialize attributes from the class (some or all of them).
- ***Parameterized***: they initialize specific attributes of the class and/or perform certain operations, according to the parameters received as arguments.

Starting with *C++1y*, the possibility of keeping a constructor capable of being called without parameters as a default constructor was introduced. It can be implemented with an empty parameter list or with a parameter list having default values for all the parameters:

```
ClassName( ) = default;
```

or,

```
ClassName(parameter_list_with_default_values) = default;
```

Also in *C++1y* a constructor can call another defined constructor using ***initializer lists***.

Special constructors (copy, move, convert)

A special type of constructor is the ***copy constructor***, whose purpose is to clone an existing object into a new one. They are only needed if the class contains pointer attributes that require dynamic memory allocation. Otherwise, the compiler provides an implicit copy constructor.

The general form of the copy constructor:

```
ClassName(const ClassName &Ob){…};
```

Starting with *C++1y* the ***move constructor*** was introduced. It allows defining a constructor that provides the address of a copyied object without creating a clone. It is implemented using *r-value* references and *move semantics* that involve addressing an object that already has allocated memory:

```
ClassName (ClassName && obj) {...} ;
```

The compiler is responsible for freeing the resources allocated to the source object.

Explicit ***conversion constructors*** can also be defined.

C++1y introduced the possibility of calling a constructor from another constructor, using intializer lists.

The class attributes are able to be initialized:

- Inside constructors (by assignments)
- in ***initializer lists***, mentioned between the constructor's header and body:

```
Point::Point(int a, int b) : x(a), y(b)
{
    // constructor body
    // x and y are class attributes, a and b are parameters
}
```

In this case, the initialization process is realized before executing the constructor's body.

const data members and *reference* data members must be initialized using initializer lists. They don't have any distinct allocated memory, being instead inserted in lookup tables.

The destructor

The destructor is a special method that:

- has no parameters
- can be explicitly implemented if specific operations have to be performed when an object is destroyed. If there are attributes represented by dynamically allocated pointers, the destructor has to free the associated memory.
- can be omitted, in which case a default destructor is called by the compiler.

The pointer named "this"

this pointer indicates the current class instance. It can be called only from within the class. It is useful in implementing the methods that have to distinguish the class attributes from the values received as parameters.

Aspects related to constructors and the initialization of arrays

For creating object arrays a constructor with no parameters is necessary (implicit empty constructor or an explicit one without any parameters, including the *default* ones). If a parameterized constructor is used, zthe initializer lists cand be used for each object in the array.

7.5. Exemple/ Examples

Ex. 1 - Setarea atributelor folosind constructori / Using constructors to set attributes

```
/* 1.a The Rectangle class, constructors and elementary  operations
*/

//Rectangle.h
class Rectangle {
    //implicit private members
    int height;
    int width;
public:
    // public members
    //explicit constructor with all implicit values
    Rectangle(int h = 10, int w = 10);
    int det_area();
    void setHeight(int);
    int getHeight() { return height; }
    void setWidth(int);
    int getWidth() { return width; }
    ~Rectangle(); // explicit destructor, not necessary
};//class

Rectangle:: Rectangle(int h, int w){
    height = h;
    width = w;
}

Rectangle:: ~Rectangle(){ cout << "\nCall destructor..."; }

void Rectangle:: setHeight(int init_height){ height = init_height; }

void Rectangle:: setWidth(int init_width){ width = init_width; }

int Rectangle:: det_area() {
    return height * width;
}
```

```
//main( )
#include <iostream>
using namespace std;
#include "Rectangle.h"

int main( ){
    int i;
    cout << "\n\nArray of objects initialized at declaration\n";
    Rectangle group1[4] = {
        Rectangle(),
        Rectangle(5),
        Rectangle(30,20),
        Rectangle(40,30)
    };

    for (i = 0; i < 4; i++)
        cout << "\nRectangle area: " << group1[i].det_area()
            <<" with sides, width= " << group1[i].getWidth()
            << " and height= " << group1[i].getHeight() << endl;

    cout << "\nEnd-Array of objects initialized at declaration\n";
    cout << ".....................................\n";

// array of objects
    Rectangle group2[4];
    cout << "\nArray of objects initialized with set methods\n";
    for (i = 1; i < 4; i++)
        group2[i].setHeight(i + 10);
    //group2[i].setWidth(10);//not necessary
    for (i = 0; i < 4; i++)
        cout << "\nRectangle area : " << group2[i].det_area()
            <<" with sides, width= " << group2[i].getWidth()
            << " and height= " << group2[i].getHeight() << endl;

    cout << "\nEnd-Array of objects assigned with set methods\n";
    cout << ".....................................\n";
// dynamic array
    Rectangle* group3 = new (nothrow) Rectangle[4];
    cout << "\nDynamic Array of objects assigned with set methods \n";
    for (i = 1; i < 4; i++)
         (group3 + i)->setHeight(i + 10);
        //(group3 + i)->setWidth(10);//not necessary
    for (i = 0; i < 4; i++)
        cout << "\nRectangle area : " << group3[i].det_area()
            <<" with sides, width= " << group3[i].getWidth()
            << " and height= " << group3[i].getHeight() << endl;
    delete[ ]group3;
    cout << "\nEnd-Dynamic Array of objects assigned with set methods\n";
    cout << ".....................................\n\n";
// dynamic array
    Rectangle* group4 = new (nothrow)  Rectangle[4];
    cout << "\n Array of objects assigned with parameterized constructor\n";
    group4[0] = Rectangle(5);//5,10
    group4[1] = Rectangle(15, 20);
    group4[2] = Rectangle(25, 30);
    group4[3] = Rectangle(35, 40);
```

```
    for (i = 0; i < 4; i++)
        cout << "\nRectangle area : " << (group4 + i)->det_area()
            <<" with sides, width= " << (group4 + i)->getWidth()
            << " and height= " << (group4 + i)->getHeight() << endl;
    delete[ ]group4;
    cout << "\nEnd-Dynamic Array of objects assigned with parameterized constr.\n";
    cout << "..................................\n\n";
    cout << "\nEnd program\n\n";
}//main

/* 1.b Rectangle class, various constructor variants */

// Rectangle.h
class Rectangle {
    int height;
    int width;
public:
    Rectangle( ) = default;

    Rectangle(int a) //explicit constructor with 1 parameter
    {
        height = width = a;
    }

    Rectangle(int h, int w);//explicit constructor, 2 parameters
    int det_area();
    void setHeight(int);
    int getHeight() { return height; }
    void setWidth(int);
    int getWidth() { return width; }
    ~Rectangle(); // explicit destructor
};//class

Rectangle::Rectangle(int h, int w){
    height = h;
    width = w;
}
Rectangle::~Rectangle(){ cout << "\nCall destructor..."; }

void Rectangle::setHeight(int init_height){ height = init_height; }
void Rectangle::setWidth(int init_width){ width = init_width; }
int Rectangle::det_area() { return height * width; }

//main
#include <iostream>
using namespace std;
#include "Rectangle.h"

int main( )
{
    int i;
    cout << "\n\nArray of objects initialized at declaration\n";
    Rectangle group1[4] = {
        Rectangle(),       //default values (0 for int ?)
        Rectangle(20), Rectangle(30, 20), Rectangle(40, 70) };
```

```
    for (i = 0; i < 4; i++)
        cout << "\nRectangle " << i<< " area: " << group1[i].det_area()
            << " with sides, width = " << group1[i].getWidth()
            << " and height = " << group1[i].getHeight() << endl;
    cout << "\nEnd-Array of objects initialized at declaration\n";
    cout << "................................\n";

// array of objects assigned with seters
    Rectangle group2[4];
    cout << "\nArray of objects assigned with set methods\n";
    for (i = 0; i < 4; i++) {
        group2[i].setHeight(i + 10);
        group2[i].setWidth(i + 20);
    }
    for (i = 0; i < 4; i++)
        cout << "\nRectangle " << i << " area: " << group2[i].det_area()
            << " with sides, width= " << group2[i].getWidth()
            << " and height= " << group2[i].getHeight() << endl;
    cout << "\nEnd-Array of objects assigned with set methods\n";
    cout << "................................\n";

// dynamic array assigned with set methods
    Rectangle* group3 = new (nothrow) Rectangle[4];
    cout << "\nDynamic Array of objects assigned with set methods \n";
    for (i = 0; i < 4; i++) {
         (group3 + i)->setHeight(i + 10);
         (group3 + i)->setWidth(2 * i + 10);
    }
    for (i = 0; i < 4; i++)
        cout << "\nRectangle " << i << " area: " << group3[i].det_area()
            << " with sides, width= " << group3[i].getWidth()
            << " and height= " << group3[i].getHeight() << endl;
    delete[ ]group3;
    cout << "\nEnd-Dynamic Array of objects assigned with set methods\n";
    cout << "................................\n\n";

// dynamic array objects assigned with constructor with parameters
    Rectangle* group4 = new (nothrow)  Rectangle[4];
    cout << "\n Array of objects assigned with constructor with parameters\n";
    group4[0] = Rectangle(5, 10);
    group4[1] = Rectangle(15, 20);
    group4[2] = Rectangle(25, 30);
    group4[3] = Rectangle(35, 40);

    for (i = 0; i < 4; i++)
        cout << "\nRectangle " << i << " area: " << (group4 + i)->det_area()
            << " with sides, width= " << (group4 + i)->getWidth()
            << " and height= " << (group4 + i)->getHeight() << endl;
    delete[ ]group4;
    cout << "\nEnd-Dynamic Array of objects assigned with parameterized constr.\n";
    cout << "................................\n\n";
    cout << "\nEnd program\n\n";
}//main
```

Ex. 2 - Stivă cu constructori speciali şi destructor / Stack with special constructors and destructor

```
// Stack class

//Stiva.h
const int dim_char = 256;//255+1 for \0
const int dim_arr = 25;

class Stiva {
    int dim;
    char* stack;
    int next;
public:
    Stiva( );
    Stiva(int);
    Stiva(const Stiva&);//copy constructor
    ~Stiva();
    int push(char c);
    int pop(char& c);
    int isEmpty(void);
    int isFull(void);
};//class

// constructors
Stiva::Stiva( ) {
    next = 0;
    dim = dim_char;
    stack = new (nothrow) char[dim];
}
Stiva::Stiva(int dim_i) {
    next = 0;
    dim = dim_i;//inclusive /0
    stack = new (nothrow) char[dim];
}

// copy constructor
Stiva::Stiva(const Stiva& instack) {
    next = instack.next;
    dim = instack.dim;
    stack = new (nothrow) char[dim];
    for (int i = 0; i < dim; i++)
        stack[i] = (char)instack.stack[i];
    cout << "\nCopy  constructor\n";
}

//destructor
Stiva :: ~Stiva() {
    delete[ ] stack;
    cout << "\nDestructor\n";
}

//isEmpty
int Stiva::isEmpty() {
    if (next <= 0)return 1;
    else return 0;
}
```

```
//isFull
int Stiva::isFull() {
    if (next >= dim) return 1;
    else return 0;
}
// push
int Stiva::push(char c) {
    if (isFull())return 0;
    *(stack + next) = c;
    next++;
    return 1;
}
// pop
int Stiva::pop(char& c) {
    if (isEmpty())return 0;
    next--;
    c = *(stack + next);
    return 1;
}

// test program -  main
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include "Stiva.h"

int main( ) {
    unsigned int i;
    char buf1[dim_char] = "Bafta in sesiune !", buf2[dim_char], buf3[dim_char];
    //strcpy(buf1, "Bafta in sesiune !");
    cout << endl << "Initial string: " << buf1 << endl;
    Stiva mesaj1;
    for (i = 0; i < (strlen(buf1)); i++)
        mesaj1.push(buf1[i]);
    i = 0;
    while (!mesaj1.isEmpty())
        mesaj1.pop(buf2[i++]);
    buf2[i] = '\0';
    cout << endl << " push() -> pop() -> reverse: " << buf2 << endl;//LIFO

    // re-populate mesaj1
    for (i = 0; i < (strlen(buf1)); i++)
        mesaj1.push(buf1[i]);
    // copy constructor -init
    Stiva mesaj2(mesaj1);
    i = 0;
    while (!mesaj2.isEmpty())
        mesaj2.pop(buf3[i++]);
    buf3[i] = '\0';
    cout << endl << "Copy previous object: " << buf3 << endl;

    char sTest[dim_arr] = "Sir_de_test";
    cout << endl << "Test string: " << sTest << endl;

    Stiva mesaj3((int)strlen(sTest) + 1);//second constructor
```

```
    for (i = 0; i < (strlen(sTest)); i++)
        mesaj3.push(sTest[i]);

    // copy constructor -init with assign
    Stiva mesaj4 = mesaj3;
    i = 0;
    while (!mesaj4.isEmpty()) mesaj4.pop(sTest[i++]);
    sTest[i] = '\0';
    cout << endl << "Copy initial test string extracted with pop(): "
        << sTest << endl;
}//main
```

Ex. 3 - Constructor de copiere / Copy constructor

```
/* Copy constructor */

//CPunctText.h
const int dim_sir = 21;//+1 for \0

class CPunctText {
    int x;
    int y;
    int lungime_sir;//redundant attribute
    char *sNume;
public:
    //explicit empty constructor
    CPunctText( );
    //parameterized constructor, last parameter with implicit value
    CPunctText(int ix, int iy, const char *sText = "Punct");
    //copy constructor
    CPunctText(const CPunctText &pct);
    //destructor
    ~CPunctText( );
    void afis() {
        cout << "\nObject has x= " << x;
        cout << "\nObject has y= " << y;
        cout << "\nObject has the array of characters =  " << sNume;
    }//afis
};//class

CPunctText::CPunctText( ) {
    cout << "\n empty explicit constructor";
    x=y=0;
    lungime_sir = dim_sir;
    sNume = new (nothrow) char[lungime_sir];
    strcpy(sNume, "Unknown");
}

CPunctText::CPunctText(int ix, int iy, const  char *sText) {
    cout << "\n parameterized constructor";
    lungime_sir = strlen(sText) + 1;// pentru \0
    sNume = new (nothrow) char[lungime_sir];
    x = ix;
    y = iy;
    strcpy(sNume, sText);
}
```

```
CPunctText::CPunctText(const CPunctText &pct) {
    cout << "\n copy constructor";
    sNume = new (nothrow)  char[pct.lungime_sir];
    x = pct.x;
    y = pct.y;
    lungime_sir = pct.lungime_sir;
    strcpy(sNume, pct.sNume);
}

CPunctText::~CPunctText() {
    cout << "\n destructor";
    delete[ ] sNume;
}

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

#include "CPunctText.h"

int main( ) {
    CPunctText cpt1(1, 2, "Punct1");//parameterized constructor call
    CPunctText cpt2(cpt1);          //copy constructor call
    CPunctText cpt3 = cpt2;         //copy constructor call
    CPunctText cpt4(4, 5);          //parameterized constructor call

    cpt3.afis();
    cpt4.afis();
}//main
```

Recommendation: introduce getter methods to use in main() instead of the afis() method and modify what it will be necessary

Ex. 4 - Implementare linie poligonală / Implementation of a polygonal line

```
/* PolygonalLine class can manage a variabil number of points
*/

#pragma once
struct Point
{
    int x, y;
};

class PolygonalLine
{
private:
    // private data
    char name[DIM];
    unsigned int npoints;
    unsigned int poz;            // first point has position 1,...
    Point* tab;
```

```
public:
    // explicit constructor
    PolygonalLine(const char* den, unsigned int np) {
        if (np == 0) np = 1;
        strcpy(name, den);
        npoints = np;
        poz = 0;
        tab = new (nothrow) Point[np];
        // display message for teaching purposes only!!!
        cout << endl << "Call explicit constructor ...";
    }

    // copy constructor
    PolygonalLine(const PolygonalLine& p) {
        strcpy(name, p.name);
        npoints = p.npoints;
        poz = p.poz;
        tab = new (nothrow) Point[npoints];
        if (tab != 0 && poz > 0 && poz < npoints) {
            for (unsigned int i = 0; i < poz; i++)
                tab[i] = p.tab[i];
            //memcpy((char*)tab, (char*)p.tab, poz * sizeof(Punct));

        }
        // display message for teaching purposes only!!!
        cout << endl << "Call copy constructor...";
    }

    // destructor
    ~PolygonalLine() {
        delete[] tab;
        // display message for teaching purposes only!!!
        cout << endl << "Call destructor...";
    }

    // accessor methods
    void setName(const char* den) { strcpy(name, den); }
    char* getName() { return name; }
    unsigned int getCapacity() { return npoints; }
    unsigned int getNPoints() { return poz; }

    void setPoint(unsigned int poz, Point p) {
        if (poz > 0 && poz <= npoints)
            tab[poz - 1] = p;
        // What happens if poz is not valid?
    }

    Point getPoint(unsigned poz) {
        if (poz > 0 && poz <= npoints)
            return tab[poz - 1];
        else
            return Point{ 0,0 };
    }
```

```
    void addPoint(Point p) {
        if (poz < npoints) { tab[poz++] = p; }
      // What happens if the point to be added already exists in the dynamic table?
    }
};

void display(PolygonalLine& lp) {
    cout << endl << "Name: " << lp.getName();
    cout << endl << "Points of the polygonal line: " << endl;
    for (unsigned int i = 1; i <= lp.getNPoints(); i++) {
        cout << "\t" << lp.getPoint(i).x << ", " << lp.getPoint(i).y << endl;
    }
}

// main
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string>
using namespace std;
const int DIM = 30;

#include "PolygonalLine.h"

int main() {
    Point p1 = { 1,1 }; Point p2 = { 2,2 }; Point p3 = { 3,3 };

    PolygonalLine lp1("Point", 1);      //  explicit constructor call
    lp1.addPoint(p1);
    display(lp1);

    PolygonalLine lp2("Segment", 2);    // explicit constructor call
    lp2.addPoint(p1);
    lp2.addPoint(p2);
    display(lp2);

    PolygonalLine lp3("Triangle", 3);    // explicit constructor call
    lp3.addPoint(p1);
    lp3.addPoint(p2);
    lp3.addPoint(p3);
    display(lp3);

    PolygonalLine lp4(lp3); // copy constructor call
    lp4.setPoint(3, Point{ 4,4 });
    display(lp4);

    // dynamic object                   //  explicit constructor call
    PolygonalLine* lp31 = new (nothrow) PolygonalLine("New triangle", 3);
    if (lp31 != 0) {
        lp31->addPoint(p1); lp31->addPoint(p3); lp31->addPoint(Point{ 5,5 });
        display(*lp31); //pointer dereferencing to obtain the reference of the
                        //object addressed by the pointer
        delete lp31; // the destructor for the lp31 object will be called
    }
    // At the end of the execution of the main() function, the destructor
    // for the objects lp4, lp3, lp2, lp1 will be called
}//main
```

Ex. 5 - Apel de constructor din alt constructor / Constructor call from another constructor

```
//constructor called from another constructor

//Header.h

class MyClass {
    int x;
    double y;
public:
    //constructor with 1 param
    MyClass(int c) {
        x = c;
        y = c;
    }
//constructor with 2 params that calls constructor with 1 param in an init list
    MyClass(int a, double b): MyClass(a)
    {
        y = b;
    }

    int getX() { return x; }
    double getY() { return y; }

    void setX(int a) { x = a; }
    void setY(double b) { y = b; }
};//class

//main
#include <iostream>
using namespace std;
#include "Header.h"

int main( ) {
    int a;
    double b;

    cout << "\nEnter an int: ";
    cin >> a;
    cout << "\nEnter a double: ";
    cin >> b;

    MyClass ob1(a, b);
    cout << "\nValues of ob1 are: " << ob1.getX() << "  " << ob1.getY();

    cout << "\nEnter another int: ";
    cin >> a;

    MyClass ob2(a);
    cout << "\nValues of ob2 are: " << ob2.getX() << "  " << ob2.getY();
    ob2.setY(7.7);
    cout << "\nValues of modified ob2 are: " << ob2.getX() << "  " << ob2.getY();
}//main
```

Ex. 6 - Constructori de copiere și de mutare / Copy and move constructors

```
//a) copy constructor, move constructor
#include <iostream>
#include <vector>
using namespace std;

class A {
    int* ptr;
public:
    A() {  // as default constructor
        cout << "\nCalling Explicit constructor no param\n";
        ptr = new (nothrow) int;
        *ptr = 0;
    }
    A(int a) {
        cout << "\nCalling Explicit Constructor with param";
        ptr = new (nothrow) int;
        *ptr = a;
    }
    A(const A& obj) {  // Copy Constructor, copy of object is created
        this->ptr = new (nothrow) int;
        // Deep copying
        *ptr = *(obj.ptr);
        cout << "\nCalling Copy constructor";
    }

    A(A&& obj) noexcept {  // Move constructor, no copy created
        cout << "\nCalling Move constructor";
        this->ptr = obj.ptr;
        obj.ptr = nullptr;
    }
    ~A() {  // Destructor
        if (ptr != nullptr)  cout << "\nCalling Destructor";
        else  cout << "\nDestructor is called" << " for nullptr with move";
        delete ptr;
    }

    A sum(A ob) {//used with copy constructor
        A s;
        *s.ptr = *(this->ptr) + *ob.ptr;
        return s;
    }
    A sum_m(A ob) {//used with copy and move constructor
        *ob.ptr = *(this->ptr) + *ob.ptr;
        return ob;
    }

    void setX(int a) { *ptr = a; }
    int getX() { return *ptr; }
};//class A

// inserter for A class - will be detailed in a later lab
ostream& operator<< (ostream& stream, A ob) {
    stream << "\nContent of A = " << ob.getX() << '\n';
    return stream;
```

```
}

int main() {
    A ob1;
    ob1.setX(10);
    cout << "\n Simple object modified with set: " << ob1.getX();
    A ob2(20);
    cout << "\n Simple object with explicit constructor with param: "
        << ob2.getX();
    cout << "\nObject previously created: ";
    A rez1 = ob1.sum(ob2);
    cout << "\n Copy constructor: rez1 sum of ob1 and ob2: " << rez1.getX();
    cout << "\n\nParameter is anonymous object:";
    A rez2 = ob1.sum(A(77));
    cout << "\n Copy constructor: rez2 sum of ob1 and anonymous object: "
        << rez2.getX();
    A rez3 = ob1.sum_m(A(17));
    cout << "\n Copy and move constr.: rez3 sum of ob1 and temp anonymous object: "
        << rez3.getX();

    cout << "\n\npush_back objects -  Move constructor, and copy constructor\n";
    vector <A> vec;//will be specified later the vector container
    vec.push_back(A());
    vec.push_back(A(7));
    vec.push_back(A(17));
    for (int i = 0; i < vec.size(); i++)
        cout << "\n vector object from position: " << i << " is: " << vec[i];

    return 0;
}//main

//b) Explicit move constructor, and copy constructor
#include <iostream>
using namespace std;

const int dim = 10;

class MyClass {
    int* data;
public:
    MyClass( ) {
        cout << "Default constructor called." << endl;
        data = new int[dim];
    }
    MyClass(const MyClass& other) {
        cout << "Copy constructor called." << endl;
        data = new int[dim];
        for (int i = 0; i < dim; i++) {
            data[i] = other.data[i];
        }
    }
    MyClass(MyClass&& other) {
        cout << "Move constructor called." << endl;
        data = other.data;
        other.data = nullptr;
    }
```

```
    ~MyClass() {
        if (data != nullptr)  cout << "\nCalling Destructor.";
        else  cout << "\nDestructor is called" << " for nullptr with move";
        delete[ ] data;
    }
};//MyClass

int main( ) {
    MyClass a;
    MyClass b = move(a);//move constructor
    MyClass c = b;//copy constructor
    return 0;
}//main
```

Ex. 7 - Utilizare liste de inițializare / Using initializer lists (const, reference)

```
/* C++ program to demonstrate the use of  initializer list for const and reference
data member */

#include<iostream>
using namespace std;

class Test {
    const int c;
    int& r;
public:
    //Initializer list must be used for const and reference
    Test(int t, int &u) :c(t), r(u) { }
    int getC() { return c; }
    int getR() { return r; }
};//Test

int main( ) {
    int x = 20;
    Test t1(10, x);
    cout << "\n Const value: "<<t1.getC();
    cout << "\n Reference value: " << t1.getR();
    return 0;
}//main
```

7.6. Lucru individual

1. Modificați exemplul 3 astfel încât să permită obținerea unui nou punct, având coordonatele determinate prin adunarea coordonatelor a două astfel de puncte. Numele noului punct va fi rezultat prin concatenarea numelor celor două puncte. Adăugați și testați o metodă care calculează distanța de la un punct la origine. Modificați clasa astfel încât să eliminați metoda *afis()* folosind în schimb metode accesor adecvate. Eliminați de asemenea atributul *lungime_sir* modificând adecvat metodele clasei. Adaugați și un move constructor. Testați și altă variantă utilizând și funcții specifice șirurilor de caractere din VC++1y/2z (*strcpy_s( )* și *strcat_s( )*).
2. Să se scrie o aplicație C++ care să modeleze obiectual un tablou unidimensional de numere reale de dimensiunea *dim*, ca și atribut, tabloul fiind dat printr-un pointer. Creați două instanțe ale clasei și afișați valorile unui al 3-lea tablou, obținute prin scăderea elementelelor corespunzătoare din primele 2 tablouri. Dacă tablourile au lungimi diferite,

tabloul rezultat va avea lungimea tabloului cel mai scurt. Inițializarea, scăderea și afișarea tablourilor se vor face cu metode din clasă, și se pot face și cu funcții globale, la alegere.

3. Modelați clasa *Student* care să conțină atributele private *nume*, *prenume*, *note (tablou 7 valori int)*, *grupa*. Alocați dinamic memorie pentru *n* studenți în *main().* Calculați media cu o metodă din clasă și sortați studenții după medie, afișând datele fiecărui student (*nume, prenume, grupa, medie*). Implementați și destructorul clasei care să afișeze un mesaj.
4. Să se scrie o aplicație în care se modelează clasa *Student* cu *nume*, *prenume*, *numar_note* și *notele* din sesiunea din iarnă declarat printr-un pointer de tip *int*. Să se afișeze numele studenților din grupă care au restanțe și apoi numele primilor 3 studenți din grupă în ordinea mediilor, care se va afisa și ea.
5. Să se scrie o aplicație C++ în care se citește de la tastatură un punct 3D prin coordonatele *x, y, z*. Să se scrie o metodă prin care să se facă translația punctului cu o anumită distanță pe fiecare dintre cele trei axe. Să se verifice dacă dreapta care unește primul punct (netranslatat) și cel rezultat în urma translației trec printr-un al treilea punct dat de la consolă.
6. Definiți o clasă *Complex* modelată prin atributele de tip double *real*, *imag* și un pointer de tip *char* către numele fiecărui număr complex. În cadrul clasei definiți un constructor explicit cu doi parametri care au implicit valoarea 1.0 și care alocă spațiu pentru *nume* un șir de maxim 15 caractere, de exemplu *"c1"*. De asemenea, definiți un constructor de copiere pentru clasa *Complex.* Clasa va mai conține metode mutator/setter și accesor/getter pentru fiecare membru al clasei, metode care permit operațiile de bază cu numere complexe și un destructor explicit. Definiți cel mult 10 numere complexe într-un tablou. Calculați suma numerelor complexe din tablou, valoare ce va fi folosită pentru a inițializa un nou număr complex, cu numele "Suma_c". Realizați aceleași acțiuni făcând diferența și produsul numerelor complexe.
7. Considerați o clasă Pair care are două atribute de tip pointeri spre int, *x și *y. Definiți un constructor fără parametri care inițializează zonele cu 0, cu doi parametri care inițializează zonele cu valorile specificate ca și parametri, un constructor cu un parametru care apelează constructorul cu doi parametri cu valori date de parametru și 0. Clasa are un copy constructor, un move constructor, un destructor (care distinge obiectele care sunt temporare) și seteri și geteri care gestionează adecvat atributele. Se definește o metodă sum() care are ca și parametru un obiect Pair și returnează un obiect Pair sumă a obiectului curent cu cel dat ca și parametru. Clasa mai contine o metodă media() care returnează media aritmetica de tip double a conținutului de la cele două atribute. Instanțiați 3 obiecte folosind constructorii fără parametri, cu un parametru, cu doi parametri. Modificați valorile la obiectul instanțiat fără parametri cu seteri. Afișati valorile de la adresa atributelor și media. Afișați valorile de la adresele atributelor celorlalte obiecte. Adunați primele două obiecte într-un nou obiect rezultat și afișați rezultatul. Considerați mesaje adecvate în constructori și destructor. Urmăriți apelurile realizate.
8. Considerăm clasa Fractie care are două atribute întregi private a și b pentru numărător și numitor, doua metode de tip set( ) respectiv get( ) pentru atributele clasei publice și o metodă simplifica( ) publică care simplifică obiectul curent Fractie de apel, returnând un alt obiect simplificat. Metoda simplifica( ) va apela o metodă private cmmdc( ) pentru simplificarea fracției. Definiți un constructor explicit fără parametri care inițializează a cu 0 și b cu 1, și un constructor explicit cu doi parametri care va fi apelat după ce s-a verificat posibilitatea definirii unei fracții (b!=0). Definiți o metodă aduna_fractie( ) care are ca și parametru un obiect de tip Fractie și returnează suma obiectului curent de apel cu cel dat ca și parametru, ca și un alt obiect de tip Fractie. Analog definiți metode pentru scadere, inmultire și impartire. Instanțiati două obiecte de tip Fractie cu date citite de la tastatură. Afisați atributele inițiale și cele obținute după apelul metodei simplifica( ). Efectuați operațiile implementate prin metodele clasei și afisați rezultatele.

9. Considerând problema precedentă adăugați în clasa *Fractie* un atribut pointer la un șir de caractere, *nume* care va identifica numele unei fractii. Constructorul fără parametri va aloca dinamic un șir de maxim 20 de caractere inițializat cu numele, *"Necunoscut"*, cel cu parametri va conține un parametru suplimentar cu numele implicit, *"Necunoscut"* care va fi copiat in zona rezervată ce va fi de două ori dimensiunea șirului implicit. Pentru acest atribut se vor crea metode accesor și mutator, care să afișeze numele unui obiect de tip *Fractie* respectiv care să poată modifica numele cu un nume specificat (*Sectiune_de_aur, Numar_de_aur*, etc.). De asemenea se va implementa și un copy constructor și un destructor. In programul principal instanțiați două obiecte de tip *Fractie*, unul folosind constructorul fără parametri, celălalt folosind constructorul cu parametri, valorile parametrilor fiind introduse de la tastatură. Modificați atributele primului obiect, folosind metode de tip *setter*. Inițializați un al treilea obiect de tip *Fractie* folosind copy constructorul. Afisați atributele acestui obiect obținut folosind metodele de tip *getter.*

7.7. Individual work

1. Modify example 3 in order to allow the addition of two *CPunctText* points. The name of the new point will be created from the names of the compounding points by concatenation. Add a method that returns the distance from a point to origin. Modify the class so that you remove the *afis()* method by using appropriate getter methods instead. Also remove the *lungime_sir* attribute by appropriately modifying the class methods. Also add a move constructor. Test using as other variant the character string specific functions of VC ++ 1y/2z (*strcpy_s ( )* and *strcat_s ( )*).
2. Write a C++ application that models as objects one-dimensional arrays of real numbers of size *dim*, as an attribute, the array being given by a pointer. Instantiate two objects of this class and store in a third one the result of subtracting each of the two real number arrays' elements. If the source arrays have different lengths, the result has the length of the shortest array. Initialization, subtraction, and display may be implemented with class methods, but can also be done with global functions.
3. Create a class named *Student* that has as private attributes the *name*, *surname*, some *marks (array 7 int values)*, the *group*. Allocate the necessary amount of memory for storing *n* students. Determine the average mark with a method from the class for each student and use it for sorting the students. Display the ordered array (*name, surname, group, average_mark*). The destructor will display a message.
4. Model in C++ a class named *Student* containing *name*, *surname* the *number of marks* and the *marks* from the winter session exams specified as an *int* pointer. Display the name of the students who have arears exams and the first three students in the group based on the average mark, that will be also displayed.
5. Write a C++ application that reads a 3D point from the keyboard by giving the *x, y* and *z* coordinates. Write a method that moves the point with a given distance on each of the three axes. Verify if the line between the first (initial position of the point) and the second position of the this point (after translation on all directions) crosses a third given point.
6. Define a class called *Complex* that stores the *double* variables for *real*, *imag* and a pointer of character type that holds the name of the complex number. Define an explicit constructor with 2 parameters that have 1.0 as implicit value. The constructor also initializes the pointer with a 15 characters wide memory zone. Define a copy constructor for this class. Implement the setter and getter methods for each attribute stored inside the class. All the operations related to complex numbers are also emulated using some specific methods. An explicit destructor method is also part of the class. Define an array of not more than 10 complex numbers. Determine the sum of all the numbers in this array and use this value for

initializing a new instance of the class named *complex_sum*. Repeat this action for all the rest of the operations implemented inside the class.

7. Consider a *Pair* class that has two pointers to *int* type, ** x* and ** y*. Define a constructor no parameters that initializes the memory zones with 0, a constructor with two parameters that initializes the memory zones with the values specified as parameters, a constructor with one parameter that calls the two-parameter constructor with the parameter value, and 0. The class has a copy constructor, a move constructor, a destructor (which distinguishes objects that are temporary) and setters and getters that properly manage attributes. A *sum()* method is defined that has a *Pair* object as a parameter and returns a *Pair* object representing the sum of the current object with the one given as a parameter. The class also contains an *average()* method that returns the *double*-type arithmetic mean of the content stored in the attributes. Instantiate 3 objects using the no-parameter, one-parameter, two-parameter constructors. Change the values for the no-parameter object with setters. Display the values from the addresses of the parameters and the average value. Display the values from the attribute addresses of other objects. Add the first two objects into a new result object and display the result. Consider appropriate messages in constructors and destructor. Track the calls.
8. Consider a class named *Fraction* that has two private integer attributes *a* and *b* for the denominator and nominator, two *set( )* and *get( )* methods and a method *simplify( )* that will simplify the current calling *Fraction* object and will return as result a *Fraction* object. *simplify( )* method will call a private *greatestCommonDivider( )* method to simplify the fraction. Define an explicit constructor without parameters that initializes *a* with 0 and *b* with 1. Define another explicit constructor that receives 2 integer parameters. For this constructor check if *b*!=0 before calling it. Define a method named *addFraction( )* that returns the object obtained by *adding* the current object with the one received as parameter, as a *Fraction* object. Define in the same manner the methods that *subtract, multiply* and *divide* two fractions. Instantiate two *Fraction* objects having the corresponding data read from the keyboard. Display the initial attributes and the ones obtained after simplifying the fractions. Call the methods that apply the implemented arithmetical operations and display the results.
9. Considering the previous task, add in the *Fraction* class another attribute consisting in a character array pointer (*name*) that identifies a fraction. The constructor without parameters will allocate a max 20 characters memory zone initialized with "*Unknown*", the parameterized constructor will have another last implicit parameter initialized with "*Unknown*" that will represent the fraction's name and the reserved space will be twice the string dimension. Implement setter and getter methods for the *name* attribute. Implement a copy constructor and a destructor. In the *main( )* function create two *Fraction* objects, one using the constructor without parameters and the other using the parameterized constructor. Modify the attributes of the first object using *setter* methods. Create a third object using the copy constructor. Display the attributes of this last object using the getter methods.

8. Funcții și clase prietene. Membri statici și *const*.

Friend functions and classes. Static and *const* members.

8.1. Obiective

- Capacitatea de a utiliza funcții și clase prietene (friend) și membri statici și *const* (atribute și metode) ai unei clase;
- Înțelegerea practică a acestor noțiuni prin implementarea de programe.

8.2. Objectives

- Using friend functions and classes, static and *const* class members (attributes and methods);
- Practical understanding of the notions mentioned above (programs implementation).

8.3. Breviar teoretic

Funcția prietenă (*friend*) este o funcție care poate accesa oricare atribut al unei clase, inclusiv cele private, din afara acelei clase prin intermediul unui obiect (sau pointer, referință) al/a clasei.

Existența lor se justifică deoarece în unele situații scutesc programatorul de munca derivării și adăugării unei noi funcționalități. Funcția *friend* poate fi la rândul ei membră a altei clase.

O clasă prietenă (*friend*) este o clasă care are acces la toți membrii unei alte clase.

Specificatorului *static* aplicat unui membru face ca acesta să existe o singură dată pentru toate obiectele de tipul clasei în care există un astfel de membru.

O particularitate în cazul folosirii atributelor statice este necesitatea redeclarării (alocării și inițializării) lor în exteriorul clasei. Unele compilatoare noi *C++1y* permit declararea atributelor statice și în clasă, de tip *inline* fără a mai fi redeclarate în afara clasei:

```
inline static int counter = 0;  //compilator gcc
```

Metodele statice au o serie de caracteristici restrictive:

- au acces doar la alți membri de tip static ai clasei și pot lucra cu membri globali;
- pointer-ul *this* nu poate fi utilizat în corpul lor de instrucțiuni;
- nu pot exista două versiuni ale aceleiași metode, una statică și una non-statică.

Membrii statici aparțin clasei și nu obiectului.

Atributele unei clase pot fi declarate constante cu modificatorul *const* (inspectați exemplul următor).

```
clasa Point {
    const int x, y;
    ...
};
```

Atributele constante au următoarele caracteristici:

- sunt inițializate numai cu un constructor, printr-o listă de inițializare
- nici o altă metodă nu poate schimba aceste date membre.

Metodele din clasă pot fi definite ca metode constante pentru a specifica faptul că aceste metode pot accesa obiectele constante (cu atribute *const*).

Când se declară o metodă (funcție membră) drept *const* (după parametrii metodei), aceasta indică faptul că **metoda promite să nu modifice niciun membru al clasei**. Această metodă este capabilă să acceseze obiecte *const* și alte obiecte (normale) (automate, anonime etc.), dar nu să modifice atributele clasei. Pointerul *this* din interiorul metodei devine un pointer către un obiect *const*.

În consecință, nu se pot modifica datele membrilor care nu pot fi de tip *mutable* în cadrul acestei metode.

Cu toate acestea, **citirea** atributelor clasei **este permisă**.

Uneori este necesară modificarea unui atribut al unui obiect chiar și în cadrul unei metode *const* (de exemplu, pentru contoarele interne). Pentru a realiza acest lucru, începînd cu C++1y/2z se poate marca atributul ca fiind mutabil (*mutable)*.

8.4. Theoretical brief

A friend function can access any attribute of a class, including the private ones, from outside that class via an object (or pointer, reference) of the class.

The friend functions' existence is justified because in some situations they relieve the programmer of the work of deriving the existent classes for adding new functionalities. The *friend* functions can be member of other classes.

A *friend* class can access all the members of another class.

The *static* specifier refers to creating a single copy for the corresponding member, no matter how many class instances are created.

A peculiarity in the case of using static attributes is the need to redeclare (assign and initialize) them outside the class. Some new *C++1y* compilers allow declaring the static attributes only inside the class. The *inline* keyword is used and the external initialization is no longer necessary:

```
inline static int counter = 0;//gcc compiler
```

The static methods have some restrictions:

- they can access only other static class members (besides accessing the global ones);
- they can't use the *this* pointer;

There cannot be two versions of the same method, one static and one non-static.

The static members belong to the class and not the object.

The attributes of a class can be declared constant with the *const* modifier (check the example).

```
class Point {
    const int x, y;
    …
};
```

The constant attributes have the following properties:

- these data are initialized only with a constructor using an *initialization list;*
- no method can change these data members.

Class methods can be marked as *const* to specify that they can access the constant objects (with *const* attributes).

When you declare a method (member function) as *const* (after the method parameters), it indicates that the method **promises not to alter any members of the class**. This method is able to access

const objects and other (normal) objects (automatic, anonymous, etc.) but not to modify the attributes. The *this* pointer inside the method becomes a pointer to a *const* object. Consequently, you cannot modify any ***non-mutable*** member data within that method.

However, **reading** the class attributes **is permissible** inside the method.

Sometimes, you need to modify an attribute of an object even within a *const* method (e.g., for internal counters). To achieve this from *C++1y/2z*, you can mark the attribute as ***mutable***.

8.5. Exemple/ Examples

Ex. 1 - Funcții friend, accesarea atributelor private / Friend functions, accessing private attributes

```
/* Friend functions, global functions */
//Media.h
class MyClass {
    int x, y;
public:
    void setX(int x) { this->x = x; }
    int getX( ) { return x; }
    void setY(int y) { this->y = y; }
    int getY( ) { return y; }

    double media();//member method
    friend double media_friend(MyClass); //friend function declaration
};//MyClass

double MyClass:: media(){
    return (x + y) / 2.;
}

double media_friend(MyClass x) {  //friend function implementation, direct access
    double rez;
    rez = (x.x + x.y) / 2.;
    return rez;
}

double media_global(MyClass x) {  //global function, access with getters
    double rez;
    rez = (x.getX( ) + x.getY( )) / 2.;
    return rez;
}

//main()
#include <iostream>
using namespace std;
#include "Media.h"

int main( ) {
    MyClass ob1;

    int x, y;
    cout << "\nValoarea lui x: ";
    cin >> x;
    cout << "\nValoarea lui y: ";
    cin >> y;
```

```
    ob1.setX(x);
    ob1.setY(y);
    cout << "\nThe average computed with the member method: " << ob1.media( );
    cout << "\nThe average computed with the friend function: "
        << media_friend(ob1);
    cout << "\nThe average computed with the global function: "
        << media_global(ob1);
}//main
```

Ex. 2 - Adunare obiecte Complex cu funcție friend / Adding Complex objects using a friend function

```
/* Friend functions applied on complex numbers */
//Complex.h

class Complex {
    double r, i;
public:
    Complex( ) {
        r = 0.;
        i = 0.;
    }
    void setR(double re) { r = re; }

    void setI(double im) {
        i = im;
    }
    double getR( ) {
        return r;
    }
    double getI( ) {
        return i;
    }
    friend Complex operator_plus(Complex, Complex);
};//class

Complex operator_plus(Complex j, Complex k) {
    Complex l;
    l.r = k.r + j.r;
    l.i = k.i + j.i;
    return l;
}

//main()
#include <iostream>
using namespace std;
#include "Complex.h"

int main( ) {
    Complex a, b, c;
    a.setR(10.);
    a.setI(20.);
    b.setR(20.);
    b.setI(30.);
    c = operator_plus(a, b);//c=a+b, if we overload + operator
    cout << "Sum is: " << c.getR( ) << "+i*" << c.getI( ) << endl;
}//main
```

Ex. 3 - Lucrul cu atribute statice publice / Working with public static attributes

```
/* 3.a – public static attributes */
//MyClass.h
class MyClass{
    int x;
public:
    static int n; //public static attribute
    MyClass(int v) {
        cout << "\nConstructor called with: " << v << endl;
        x = v;
        n++;
    }
    int getX( ) {
        return x;
    }
    ~MyClass( ) {
        cout << "\nDestructor called: n = " << n << endl;
        n--;
    }
};//MyClass

int MyClass:: n;   //static attribute initialization (will be 0)

//main( )
#include <iostream>
using namespace std;
#include "MyClass.h"

int main( ) {
    cout << "Access using the class name: n = " << MyClass:: n << endl;
    MyClass a(3);
    cout << "Atrributes: " << "x = " << a.getX( )
        << ", n = " << MyClass:: n << endl;
    cout << "\nAccess using the first object (not recommended), a: n = " << a.n
        << endl;
    MyClass b(5);
    cout << "Access using the second object: " << "x = " << b.getX( )
        << ", n = " << b.n << endl;
    cout << "\nAccess using the class name: n = " << MyClass:: n
        << " and the first object, a = " << a.n << endl;
}//main

//3.b – static private attributes, static public methods
//MyClass.h

class MyClass {
    int x;
    static int n;//private static attribute
public:
    void setX(int a) {
        x = a;
    }
    static void setN(int b) {
        n = b;
    }
```

```
    int getX( ) {
        return x;
    }
    static int getN( ) {//static attribute accessed
        return n;
    }
};//MyClass

int MyClass:: n;   //static attribute initialization (=0)

//main( )
#include <iostream>
using namespace std;
#include "MyClass.h"

int main( ) {
    MyClass ob1, ob2;
    ob1.setX(1);
    ob1.setN(2);//not recommended
    cout << "\t n(static) using an object: " << ob1.getN( );//not recommended
    MyClass:: setN(3);
    cout << "\n x(nonstatic): " << ob1.getX( );
    cout << "\t n(static) using the class name: " << MyClass:: getN( );
    cout << "\n n(static) using the first object: " << ob1.getN( );
    ob2.setX(5);
    ob2.setN(6);
    MyClass:: setN(7);
    cout << "\n x(nonstatic) from ob2: " << ob2.getX( );
    cout << "\n n(static) from ob2: " << ob2.getN( );
    cout << "\t n(static) using the class name: " << MyClass:: getN( );
    cout << "\n x(nonstatic) from ob1: " << ob1.getX( );
    cout << "\n n(static) from ob1: " << ob1.getN( );
    cout << "\t n(static) using the class name: " << MyClass:: getN( );
}//main
```

Ex. 4 - Atribute statice, metode statice / Static attributes, static methods

```
/*the static method v_calend that works with the static attributes zz,ll,aa from
the Dc (Date from Calendar) */
//Dc.h
class Dc {
    static int zz, ll, aa;
    public: Dc(int z = 15, int l = 4, int a = 2024) {
        zz = z;
        ll = l;
        aa = a;
    }
    static int v_calend( );

    // friend  void afis_data(Dc);
    friend  void afis_data( );
};//Dc

int Dc::zz, Dc::ll, Dc::aa;//static attributes that belong to the class
```

```
int Dc::v_calend( ) //validate data calend including leap year and days of months
{
    static int t_nrz[ ] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
    if (Dc::aa < 1600 || Dc::aa>4900) return 0;
    if (Dc::ll < 1 || Dc::ll>12) return 0;
    if (Dc::zz < 1 ||
        Dc::zz > t_nrz[Dc::ll] +
        (Dc::ll == 2 && (Dc::aa % 4 == 0 && Dc::aa % 100 || Dc::aa % 400 == 0))
       )
        return 0;
    return 1;
}//v_calend

/*
void afis_data(Dc d) {
    cout << "\nDay= " << d.zz << " Month= " << d.ll << " Year= " << d.aa;
}
*/
void afis_data( ) {
    cout << "\nDay= " << Dc::zz << " Month= " << Dc::ll << " Year= " << Dc::aa;
}

//main( )
#include <iostream>
using namespace std;
#include "Dc.h"

int main() {
    int z, l, a;
    cout << "\n Enter a day: ";
    cin >> z;
    cout << "\n Enter a month: ";
    cin >> l;
    cout << "\n Enter a year: ";
    cin >> a;
    Dc data_calend(z, l, a);//pass to static attributes
    if (Dc::v_calend() == 0) { cout << "\n Wrong date"; exit(1); }
    //if (data_calend.v_calend() == 0){ cout << "\n Wrong date"; exit(1);}
    //afis_data(data_calend);
    afis_data();
    return 0;
}//main
```

Ex. 5 - Număr de apeluri de constructor / Counting constructor calls

```
//the Object class with a counter named atr_static, as static attribute

class ObJect {
    int value;
    static int atr_static;
    public:
    ObJect(int v) {
        value = v;
        atr_static ++;
    }
    int getValue( ) { return value; }
```

```
    static int getAtr_static( ) {
        return atr_static;
    }
};

int ObJect:: atr_static;//external intialization and allocation
```

Ex. 6 - Obiecte și metode constante / Constant objects and methods

```
//  const methods and const objects
// hPoint.h:
class Point {
    const int x, y;//const attributes
public :
    Point(int a, int b) : x(a), y(b){}
    void print1(void) const;//const method
    void print2(void);
};//Point class

// The source code App_Point.cpp
#include <iostream>
using namespace std;
#include "hPoint.h"

void Point::print1(void) const {
    cout<<"\n const method"<<endl;
    cout << "\n Abscissa (const) : " << x;
    cout << "\n Ordinate (const): " << y;
}// print1

void Point::print2(void) {
    cout<<"\n normal method"<<endl;
    cout << "\n Abscissa : " << x;
    cout << "\n Ordinate : " << y;
}// print2

int main( ){
const Point pct(5,1); //const Point object
    pct.print1( );
    //pct.print2( );//error not const method
}//main
```

Ex. 7 - Atribute mutabile / Mutable attributes

```
// mutable attributes
#include <iostream>
using namespace std;

class MyClass {
    mutable int counter;
    // int counter;
public:
    MyClass() : counter(0) { }//init counter to 0
    void foo() {
        counter++; // This works
        cout << "\nFrom foo: " << counter<< endl;
    }
```

```
    void foo_c() const {
        counter++; // allowed because `counter` is `mutable`, otherwise error
        cout << "\nFrom foo_c const: " << counter<< endl;
    }
    int getCounter() const { return counter; }
};//MyClass

int main() {
    MyClass obj;
    obj.foo();
    cout << "\nThe counter with foo() and getter is: " << obj.getCounter();
    obj.foo_c();
    cout << "\nThe counter with foo_c const and getter is: " << obj.getCounter();
}//main
```

8.6. Lucru individual

1. Implementați o aplicație C++ în care clasa *OraCurenta* are ca și atribute *private* ora, minutele și secundele și metode *publice* de tip *set/get* pentru atributele clasei. Adăugați o funcție *friend* clasei prin care să se poată copia conținutul unui obiect *OraCurenta* dat ca și parametru, într-un alt obiect instanță a aceleiași clase care va fi returnat de funcție, ora fiind însă modificată la Greenwich Mean Time. Utilizați timpul curent al calculatorului.
2. Scrieți o aplicație C++ în care clasa *Calculator* are un atribut privat *memorie_RAM* (*int*) și o funcție prietenă *tehnician_service( )* care permite modificarea valorii acestui atribut. Funcția *friend* va fi membră într-o altă clasă, *Placa_de_baza* care are o componentă *denumire_procesor* (șir de caractere). Scrieți codul necesar care permite funcției prietene *tehnician_service( )* să modifice (schimbe) valoarea atributului *denumire_procesor* și *memorie_RAM*.
3. Definți o clasă numită *Repository* care are două atribute *private* de tip întreg. Clasa mai conține un constructor explicit vid și unul cu 2 parametri și metode accesor care afișează valorile atributelor din clasă. Scrieți o clasă numită *Mathemathics*, *friend* cu prima clasă, care implementează operațiile aritmetice elementare (+, -, *, /) asupra atributelor din prima clasă. Fiecare metodă primește ca și parametru un obiect al clasei *Repository*.
4. Scrieți o aplicație C++ care definește într-o clasă atributul *public* contor *atr_static* de tip *static* întreg. Aceasta se va incrementa în cadrul constructorului. După o serie de instanțieri, să se afișeze numărul de obiecte create (conținutul variabilei *atr_static*).
5. Rezolvați problema 4 în cazul în care atributul *static* este de tip *private*. Definiți o metodă accesor care returnează valoarea contorului. Analizați cazul în care metoda accesor e statică sau nestatică și modul în care e apelată.
6. Scrieți o aplicație C++ care definește o clasă numită *Triunghi*. Clasa conține ca și atribute statice *private int* laturile *a, b* și *c* ale triunghiului, iar în zona publică un constructor cu parametri și metode adecvate setter și getter pentru fiecare atribut, cu metode separate. Clasa va conține metode care vor calcula aria și perimetrul formei. Scrieți o metodă statică distinctă fără parametri care va afișa un mesaj specific dacă triunghiul este dreptunghic. Scrieți o metodă publică statică care va determina dacă valorile laturilor date ca și parametri formează un triunghi. Metoda va fi apelată în *main( )* înainte de instanțierea unui obiect și în metodele setter, dacă schimbăm laturile unui obiect existent. Setter-ul va păstra valoarea laturii vechi dacă cea nouă nu este o latură validă. Înainte de a instanția un obiect se preiau de la tastatură 3 valori *int* pentru cele 3 laturi, cu confirmare prin reintroducerea valorilor, dacă este cazul, și se verifică prin metoda statică (cu parametri) valabilitatea să fie laturi ale unui triunghi.

7. Scrieți o aplicație C++ în care să implementați clasa *Punct* cu atributele *private x* și *y*. Implementați două funcții *friend* care să calculeze și să afișeze aria și perimetrul a mai multe forme geometrice echilaterale (triunghi, patrat, hexagon, etc.) ale căror latură este definită de două puncte. Punctele *P0(x0,y0)* și *P1 (x1,y1)*, precum și numărul de laturi sunt trimise ca parametri către funcțiile friend. Coordonatele punctelor și apoi selecția figurii geometrice se va realiza introducând de la tastatură valorile corespunzătoare.
8. La un chioșc se vând ziare, reviste și cărți. Fiecare publicație are un nume, o editură, un număr de pagini, un număr de exemplare per publicație și un preț fără TVA. Scrieți clasa care modelează publicațiile. Adăugați un membru static *private valoare_tva* (procent) și o metodă statică pentru modificarea valorii TVA-ului. Să se calculeze suma totală cu TVA pe fiecare tip de publicație (ziare, reviste și carți) și prețul mediu pe pagină la fiecare publicație în parte. Modificați TVA-ul și refaceți calculele. Afișați editurile ordonate în funcție de încasări.
9. Considerați clasa *Fractie* care are două atribute întregi *private a* și *b* pentru numărător și numitor, două metode de tip *set( )* respectiv *get( )* pentru fiecare din atributele clasei. Declarați o funcție *friend simplifica( )* care are ca și parametru un obiect al clasei, returnând un alt obiect simplificat. Considerați o variabila *private* statică întreagă *icount*, care va fi inițializată cu *0* și incrementată în cadrul constructorilor din clasa. Definiți un constructor explicit fară parametri care inițializează *a* cu 0 și *b* cu 1, și un constructor explicit cu doi parametri care va putea fi apelat daca se verifică posibilitatea definirii unei fracții (*b*!=0). Definiți un destructor explicit care afișează și decrementează contorul *icount*. Definiți o funcție *friend _f_aduna_fractie(...)* care are ca și parametri două obiecte de tip *Fractie* și returnează suma obiectelor în alt obiect *Fractie*. Analog definiți funcții *friend* pentru scădere, înmulțire și împărțire. Instanțiați două obiecte de tip *Fractie* cu date citite de la tastatură. Afisați atributele inițiale și cele obținute după apelul funcției *simplifica( )*. Printr-o metodă accesor, afișați contorul *icount*. Efectuați operațiile implementate prin funcțiile *friend* ale clasei, inițializând alte 4 obiecte cu rezultatele obținute. Afisați rezultatele și contorul după ultima operație folosind o metodă accesor adecvată.
10. Considerați problema referitoare la gestiunea CNP-ului dintr-un laborator anterior (lab. 6), la care să validați complet CNP-ul (zi corectă funcție de luna corectă și an corect, inclusiv anii bisecți), în care să introduceți cu confirmare mai multe date de tip *Person,* afișând la final câte obiecte au data introdusă corect. Contorizați folosind atribute statice private.

8.7. Individual work

1. Implement a C++ application that defines the class called *CurrentHour* with *hour*, *minute*, *second* as *private* attributes. The class has *public* setter/getter methods for each attribute. Add a *friend* function that copies the content of a *CurrentHour* object used as parameter into another instance of the class that will be returned by the function, the hour being modified to Greenwich Mean Time. Use the computer local current time.
2. Write a C++ application in which the class *Calculator* has a private attribute called *RAM_memory* (int) and a *friend* function named *service_technician( )* that can modify the attribute's value. The *friend* function will be member in the class *Motherboard*, that encapsulates the *processor_type* attribute (one dimensional array of characters). Write the code that allows the modifying of the *processor_type*'s value and the *RAM_memory* from the friend function.
3. Define a class called *Repository* that has 2 integer private attributes. The class contains an empty constructor and another one with 2 parameters. An accesor method that displays the attributes values is also included in the class. Write another class called *Mathematics* which is friend to the first one. This class contains the implementation of the elementary arithmetical operations (+, -, *, /) applied to the values stored in the first class. Each arithmetical method receives as parameter an object instantiated from the first class.

4. Write a C++ application that stores inside a class a public static integer attribute called *static_counter*. The attribute is incremented each time the class's constructor is called. After instantiating several objects, display their number using the value of the static attribute.
5. Implement the 4-th problem by changing the static attribute's access modifier to *private*. Define a method that returns the counter's value. Consider the case in which the getter method is static or non-static and access it accordingly.
6. Write a C++ application that defines a class called *Triangle*. The class contains as *private static int* attributes the triangle's sides *a, b* and *c*, and in the public zone a constructor with parameters and adequate setter and getter methods for each attribute in separate methods. The class will contain methods that will calculate the shape's area and perimeter. Write a distinct static method (no parameters) that will print a specific message if the triangle has a 90 degree angle. Develop a *static public* method that will determine whether the values of *the 3 sides* as parameters can form a triangle. The method will be called in *main( ) before instantiating* an object and in setter methods, if we change the sides of an existing object. The setter will keep the old side value if the new one is not valid.
   Before instantiating an object, read from the keyboard 3 *int* values corresponding to the 3 triangle sides. Verify them with the static method and, if necessary, ask the user to write them again.
7. Write a C++ application that defines a class named *Poin*t with the private attributes *x* and *y*. Implement two *friend* functions that calculate and display the area and perimeter of different equilateral geometrical shapes (triangle, square, pentagon, etc.) whose side is defined by two *Point* parameters *P0(x0, y0)* and *P1(x1, y1)*. The number of sides is also sent as parameter. The points' coordinates and the shape selection will be realized using parameters introduced from KB.
8. A kiosk sells newspapers, magazines, and books. Each publication has a name, an editorial house, a number of pages, the number of copies and a price (without VAT). Write the class that models the publications. Introduce a *private* static member named *VAT_value* (percentage) and a static method that modifies the value of this variable. Determine the total income and the average price per page for each publication type. Modify the VAT and redo the calculations. Order the printing houses by the total income and display the result.
9. Consider the *Fraction* class that has two *private* integer attributes *a* and *b* for the denominator and nominator. Use two setter and getter methods for all the class's attributes. Declare a *friend* function named *simplify( )* that receives as parameter a *Fraction* object and returns the corresponding simplified object. Consider a *private* static integer variable *icount* that will be initialized with 0 and incremented in the class's constructors. Define an explicit constructor without parameters that initializes *a* with 0 and *b* with 1 and another explicit constructor with two integer parameters. Before calling this constructor check if b!=0. Define an explicit destructor that displays and decrements the value of *icount.* Define a *friend* function *f_add_fraction(...)* that returns an object reflecting the sum of the objects received as parameters. Implement similar functions for fractions subtraction, multiplication and division. Instantiate two *Fraction* objects and read the appropriate data from the keyboard. Display the initial attributes and the ones obtained after simplifying. Call the implemented *friend* functions and store the results into another different four objects. Display the results and the objects counter using the corresponding accesor methods.
10. Consider the CNP management problem from a previous laboratory (lab. 6) where you fully validate the CNP (correct day based on the correct month and the correct year, including the leap years). Read and check (in a loop, as long as continuation is confirmed) the data corresponding to several *Person* objects. Use private static attributes as counters.

9. Supraîncărcarea metodelor și operatorilor.

Methods and operators overloading.

9.1. Obiective

- Capacitatea de a utiliza metode specifice din cadrul clasei și funcții și clase prietene (*friend*) ale unei clase pentru supraîncărcări de operatori;
- Înțelegerea practică a noțiunii de polimorfism prin implementarea de programe cu supraîncărcări de metode și operatori.

9.2. Objectives

- The ability of using specific member methods and friend classes and functions for operators overloading;
- Practical understanding of polymorphism by implementing programs that overload methods and operators.

9.3. Breviar teoretic

Supraîncărcarea metodelor (funcțiilor membre) (*overloading*) este unul dintre atributele esențiale ale limbajului C/C++, conferindu-i flexibilitate în implementarea programelor; ea implementează conceptul de *polimorfism* static.

Termenul de *polimorfism* denotă acea caracteristică a unei anumite componente soft de a avea "mai multe forme". Frecvent supraîncărcarea este realizată la nivelul constructorilor. Metodele care au acces la atributele clasei nu sunt în general supraîncărcate. Metodele virtuale pot fi fi redefinite într-un mecanism de polimorfism dinamic.

Operatorii se supraîncarcă prin crearea metodelor/funcțiilor-*friend* cu nume *operator.* O metodă/funcție *operator* definește operațiile specifice pe care le va efectua operatorul respectiv, relativ la clasa în care a fost destinat să lucreze.

Se pot supraîncărca aproape toți operatorii, cu unele excepții. Pot fi supraîncărcați și operatorii *new* și *delete,* însă cu metode (funcții membre) statice, fără a specifica explicit aceasta. Scopul supraîncărcării acestor operatori este de a efectua anumite operații speciale de alocare de memorie în momentul creării sau distrugerii unui obiect din clasa în care aceștia au fost supraîncărcați.

Tabelul următor prezintă o sinteză a mecanismului de supraîncărcare a operatorilor.

Expression	Operator	Member function - method	Global function - friend
`@A`	`+ - * & ! ~ ++ --`	`A::operator@()`	`operator@(A)`
`A@`	`++ --`	`A::operator@(int)`	`operator@(A,int)`
`A@B`	`+ - * / % ^ & | < > == != <= >= << >> && || ,`	`A::operator@(B)`	`operator@(A,B)`
`A@B`	`= += -= *= /= %= ^= &= |= <<= >>= []`	`A::operator@(B)`	-
`A(B,C,...)`	`()`	`A::operator()(B,C...)`	-
`A->X`	`->`	`A::operator->()`	-

9.4. Theoretical brief

Overloading methods (member functions) is one of the essential attributes of C++, giving it flexibility in implementing programs; It implements the concept of *static polymorphism*.

The term *polymorphism* denotes that characteristic of a particular software component to have "several forms". Frequently the overloading is performed at constructors level. Methods that have access to class attributes are generally not overloaded. Virtual methods can be redefined into a dynamic polymorphism mechanism.

Operators get overloaded by creating methods/friend-functions with *operator names.* An operator method/function defines the specific operations that the operator will perform, relative to the class in which it is intended to work.

Almost all operators can be overloaded, with some exceptions. *new* and *delete* operators can also be overloaded, but with static methods (member functions), without explicitly specifying this. The purpose of overloading these operators is to perform certain special memory allocation operations when creating or destroying an object of the class in which they were overloaded.

The following table summarises the operator overloading mechanism.

Expression	Operator	Member function - method	Global function - friend
`@A`	`+ - * & ! ~ ++ --`	`A::operator@()`	`operator@(A)`
`A@`	`++ --`	`A::operator@(int)`	`operator@(A,int)`
`A@B`	`+ - * / % ^ & | < > == != <= >= << >> && || ,`	`A::operator@(B)`	`operator@(A,B)`
`A@B`	`= += -= *= /= %= ^= &= |= <<= >>= []`	`A::operator@(B)`	-
`A(B,C,...)`	`()`	`A::operator()(B,C...)`	-
`A->X`	`->`	`A::operator->()`	-

9.5. Exemple/ Examples

Ex. 1 - Supraîncărcarea constructorilor și a operatorilor / Overloading constructors and operators

```
/*  1.a  Constructors overloading;
         Overloading the operators +, -, and [ ] for the String class
*/
//String_static.h

const int dim = 31;//30+1 for \0

class String {
    char sir[dim];
public:
    String( )=default;
    String(char x[dim]) {
        strcpy(sir, x);
    }
```

```
    char* getSir( ) {
        return sir;
    }

    String operator+ (String x1) { // overloading with method (member function)
        String rez;
        strcpy(rez.sir, sir);//attention at sir length
        strcat(rez.sir, x1.sir);
        return rez;
    }
    // [ ] overloading for returning a character from a certain position
    char operator[ ](int poz) {
        char rez;
        rez = sir[poz];
        return rez;
    }
    //friend function for overloading - operator
    friend String operator- (String& x1, String& x2);
};//String

// operator- overloading friend function
String operator- (String& x1, String& x2) {
    char* pp;
    //the position where x2.sir is found in x1.sir
    pp = strstr(x1.sir, x2.sir);
    if (pp == NULL) //not found
        return String(x1.sir);
    else {
        char buf[dim];
        strncpy(buf, x1.sir, pp - x1.sir);
        strcpy(buf + (pp - x1.sir), pp + strlen(x2.sir));
        //strcpy_s(buf + (pp - x1.sir), dim - (pp - x1.sir), pp + strlen(x2.sir));
        return String(buf);
    }
}

//main( )
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include "String_static.h"

int main( ) {
    char xx[dim];
    cout << "\n Write an array of chars (object 1): ";
    cin.getline(xx, sizeof(xx));// //gets_s(xx, sizeof(xx));
    String ob1(xx);
    cout << "\n Write an array of chars (object 2): ";
    cin.getline(xx, _countof(xx));//gets_s(xx, _countof(xx));
    String ob2(xx);

    String ob3;//default constructor called
    ob3 = ob1 + ob2;//implicit assign, + overloading
    cout << "\nThe array after addition: " << ob3.getSir();
    cout << "\n The array to be subtracted from the 1-st one: ";
    cin.getline(xx, _countof(xx));//gets_s(xx, _countof(xx));
```

```
    String ob4(xx);
    String ob5 = ob1 - ob4;//implicit copy constructor, cloning, - overload
    cout << "\nThe subtraction result: " << ob5.getSir( );
    char c = ob5[0];
    cout<<"\nFirst character: "<<c;
    String obx = move(ob3);
    cout << "\nThe array from obx after moving ob3: " << obx.getSir();
    }//main

//1.b  constructors overloading; dynamic memory allocation
//StringA.h
const int dim = 31;//30+1 for \0

class String {
    char* sir;
public:
    String( ) { //default constructor
        sir = new (nothrow) char[dim]; //default memory zone
    }

    String(char* x) {
        sir = new (nothrow)  char[strlen(x) + 1];//exact memory zone +\0
        strcpy(sir, x);
    }

    String(const String& x) {//copy constructor
        sir = new (nothrow) char[strlen(x.sir) + 1];
        strcpy(sir, x.sir);
        cout << "\nCopy constructor";
    }

    String& operator= (const String& x) {//assign overloading
        if (this == &x)
            return *this;
        delete[ ] sir;//free initial memory
        sir = new (nothrow) char[strlen(x.sir) + 1];//exact memory +\0
        strcpy(sir, x.sir);
        cout << "\nAssign overload";
        return *this;
    }

    String(String&& a) {//move constructor
        sir = a.sir;
        cout << "\nMove constructor";
        a.sir = nullptr;
    }

    ~String( ) {
        if (sir != nullptr) cout << "\nCall Destructor\n";
        else  cout << "\nDestructor is called for nullptr with move\n";
        delete[ ] sir;
    }

    void setSir(char* x) {
        strcpy(sir, x);
    }
```

```
    char* getSir( ) {
        return sir;
    }

    String operator+ (String x1) {  //overload with method (member function)
        String rez;
        rez.~String();//destructor call for freeing the memory
        rez.sir = new (nothrow) char[strlen(this->sir) + strlen(x1.sir) + 1];
        strcpy(rez.sir, sir);
        strcat(rez.sir, x1.sir);
        return rez;
    }

    char operator[ ](int poz) { // [ ] overloading
        char rez;
        rez = sir[poz];
        return rez;
    }

    //friend function for operator- overloading
    friend String operator- (const String& x1, const String& x2);
};//String

// operator- overloading friend function
String operator- (const String& x1, const String& x2) {
    char* pp;
    pp = strstr(x1.sir, x2.sir);
    if (pp == NULL)
        return String(x1.sir);
    else {
        //char buf[dim];
        char* buf = new (nothrow) char[strlen(x1.sir) + 1];
        strncpy(buf, x1.sir, pp - x1.sir);
        strcpy(buf + (pp - x1.sir), pp + strlen(x2.sir));
        return String(buf);
    }
}

//main( )
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include "StringA.h"

int main( ) {
    char xx[dim];
    cout << "\nArray of chars (ob1), smaller than " << dim << " :";
    cin.getline(xx, sizeof(xx));
    String ob1(xx);
    cout << "Array of chars (ob2), smaller than " << dim << " :";
    cin.getline(xx, sizeof(xx));
    String ob2(xx);
    String ob3;
    ob3 = ob1 + ob2;//assign, + overloading
    cout << "Addition result: " << ob3.getSir();
```

```
    cout << "\n Array of chars (ob4) to be subtracted, smaller than " << dim
        << " :";
    cin.getline(xx, sizeof(xx));
    String ob4(xx);
    String ob5 = ob1 - ob4;//copy constructor, - overloading
    cout << "Subtraction result: " << ob5.getSir();
    String ob6;
    cout << "\n Array of chars (ob6) smaller than " << dim << " :";
    cin.getline(xx, _countof(xx));
    ob6.setSir(xx);
    cout << "Sirul : " << ob6.getSir();
    ob3.setSir(xx);
    cout << "\nThe array from ob3 set to ob6: " << ob3.getSir();
    char c = ob3[0];
    cout<<"\nFirst character: "<<c;

    String obx = move(ob3);
    cout << "\nThe array from obx after moving ob3: " << obx.getSir();
}//main
```

Note: For correct operation and efficiency at ex. 1.b it is necessary to overload the assign operator for the String class, copy and move constructor and implement a destructor to free the memory

Ex. 2 - Supraîncărcarea operatorilor (clasa Complex) / Operator overloading (Complex class)

```
/* v1.  class for modelling complex numbers, display complex object using a method
that returns the object formatted as a string
string_for_display */

// Complex.h
#include <iostream>
#include <string>
using namespace std;

class Complex {
    float re, im;
public:
    // Because the constructor has default values for the parameters
    // it will also be used for conversions from float to Complex.
    Complex(float re = 0, float im = 0) { this->re = re; this->im = im; }

    float getRe() const { return re; }
    float getIm() const { return im; }
    void setRe(const float r) { re = r; }
    void setIm(const float i) { im = i; }

    string string_for_display() const {
        return "(" + to_string(re) + ") + j*(" + to_string(im) + ")\n";
    }

    // Operator + overloading using member method
    Complex operator+(Complex) const;
```

```
    // Operator - overloading using friend function
    friend Complex operator-(Complex, Complex);

    // Operator ~ (modulus) overloading using member method
    float operator~() const {
        return sqrt(re * re + im * im);
    }

    // Prefixed ++ operator overloading: modifies the operand before it is used
    Complex& operator++() { re += 5.0f; im += 5.0f; return *this; }

    // Postfixed ++ operator overloading: modifies the operand after it is used
    Complex operator++(int) {
        const Complex temp = *this;     // local copy of the current object
        ++(*this);       // current object increment using the previous version
        return temp;     // return the copy of the object before incrementation
    }
};//Complex_class

// Method operator+
Complex Complex::operator+(const Complex x) const {
    return { im + x.im, re + x.re };
}

// Friend function operator-
Complex operator-(const Complex x, const Complex y) {
    return Complex(x.re - y.re, x.im - y.im);
}

// External function operator*
Complex operator*(const Complex x, const Complex y) {
    Complex rez;
    const float r1 = x.getRe(); const float i1 = x.getIm();
    const float i2 = y.getIm(); const float r2 = y.getRe();
    rez.setRe(r1 * r2 - i1 * i2);
    rez.setIm(r1 * i2 + r2 * i1);
    return rez;
}

// main
#include <iostream>
#include <string>
using namespace std;

#include "Complex.h"

int main() {
    Complex c1(1, 2), c2(3, 4), c3;
    cout << "(c1) = " << c1.string_for_display();
    cout << "(c2) = " << c2.string_for_display();

    c3 = c1 + c2;
    cout << "(c3= c1 + c2) = " << c3.string_for_display();

    c3 = c1 - c2;
    cout << "(c3= c1 - c2) = " << c3.string_for_display();
```

```
    c3 = c1 * c2;
    cout << "(c3= c1 * c2) = " << c3.string_for_display();

    cout << "(module c1) = " << ~c1 << endl << endl;

    c2 = c1++;
    cout << "(c1++) = " << c1.string_for_display();
    cout << "(c2 = c1++) = " << c2.string_for_display();

    Complex c4 = ++c1;
    cout << "(++c1) = " << c1.string_for_display();
    cout << "(c4 = ++c1) = " << c4.string_for_display();

    c2++;
    cout << "(c2++) = " << c2.string_for_display();
}//main



// v2. class for modelling complex numbers (without string pseudocontainer)

// Complex.h
class Complex {
    float re, im;
public:
    // Because the constructor has default values for the parameters
    // it will also be used for conversions from float to Complex.
    Complex(float re = 0, float im = 0) { this->re = re; this->im = im; }

    // Naming convention
    float getRe() const { return re; }
    float getIm() const { return im; }
    void setRe(const float r) { re = r; }
    void setIm(const float i) { im = i; }

    void displayComplex(void) const {
        cout << "(" << re << ")+ j*(" << im << ")\n";
    }

    // Operator + overloading using member method
    Complex operator+(Complex) const;

    // Operator - overloading using friend function
    friend Complex operator-(Complex, Complex);

    // Operator ~ (modulus) overloading using member method
    float operator~() const {
        return sqrt(re * re + im * im);
    }

    // Prefixed ++ operator overloading: modifies the operand before it is used
    Complex& operator++() { re += 5.0f; im += 5.0f; return *this; }

    // Postfixed ++ operator overloading: modifies the operand after it is used
    Complex operator++(int) {
```

```
        const Complex temp = *this;     // local copy of the current object
        ++(*this);     // current object increment using the previous version
        return temp;   // return the copy of the object before incrementation
    }
};//Complex_class

// Method operator+
Complex Complex::operator+(const Complex x) const {
    return { im + x.im, re + x.re };
}

// Friend function operator-
Complex operator-(const Complex x, const Complex y) {
    return Complex(x.re - y.re, x.im - y.im);
}

// External function operator*
Complex operator*(const Complex x, const Complex y) {
    Complex rez;
    const float r1 = x.getRe(); const float i1 = x.getIm();
    const float i2 = y.getIm(); const float r2 = y.getRe();
    rez.setRe(r1 * r2 - i1 * i2);
    rez.setIm(r1 * i2 + r2 * i1);
    return rez;
}

// main
#include <iostream>
using namespace std;

#include "Complex.h"

int main() {
    Complex c1(1, 2), c2(3, 4), c3;
    cout << "(c1) = ";
    c1.displayComplex();
    cout << "(c2) = ";
    c1.displayComplex();

    c3 = c1 + c2;
    cout << "(c3= c1 + c2) = ";
    c3.displayComplex();

    c3 = c1 - c2;
    cout << "(c3= c1 - c2) = ";
    c3.displayComplex();

    c3 = c1 * c2;
    cout << "(c3= c1 * c2) = ";
    c3.displayComplex();

    cout << "(modul c1) = " << ~c1 << endl << endl;

    c2 = c1++;
    cout << "(c1++) = ";
    c1.displayComplex();
```

```
    cout << "(c2 = c1++) = ";
    c2.displayComplex();

    Complex c4 = ++c1;
    cout << "(++c1) = ";
    c1.displayComplex();
    cout << "(c4 = ++c1) = ";
    c4.displayComplex();

    c2++;
    cout << "(c2++) = ";
    c2.displayComplex();
}//main
```

Ex. 3 - Supraîncărcarea new și delete / Overloading new and delete

```
/* new and delete operators overloaded in class Pozitie by default as static
methods */
//Pozitie.h

class Pozitie {
    int x;
    int y;
public:
    Pozitie( )=default;
    Pozitie(int oriz, int vert) {
        x = oriz;
        y = vert;
    }
    int getX( ) {
        return x;
    }
    int getY( ) {
        return y;
    }

    void *operator new  (size_t marime);
    void operator delete (void *p);
};//class

void *Pozitie :: operator new  (size_t marime) {
    cout<<"\n new overloaded";
    return malloc(marime);//a dedicated overloaded can be done with LSI or LDI
}

void Pozitie :: operator delete (void *p) {
    cout<<"\n delete overloaded";
    free(p);
}

//main( )
#include <iostream>
using namespace std;
#include "Pozitie.h"
```

```
int main( ) {
    Pozitie *p1, *p2;
    p1 = new Pozitie(100, 200);
    if (!p1) {
        cout<<"\n p1 was not allocated.";
        exit(0);
    }
    p2 = new  Pozitie(10, 20);
    if (!p2) {
        cout<<"\n p2 was not allocated.";
        exit(1);
    }
    cout << "\np1 has the coordinates: " << p1->getX( ) << " " << p1->getY( );
    cout << "\np2 has the coordinates: " << p2->getX( ) << " " << p2->getY( );
    delete(p1); delete(p2);
}//main
```

Ex. 4 – Supraîncărcarea ++ și -- (clasa Time) / Overloading ++ and -- (Time class)

```
//overloading ++ and --, other operators
//Time_inc_dec.h

class Time{
    int hh;
    int mm;
public:
    Time(int m) {
        if (m >= 0) {//validation before call in main( )
            hh = m / 60;
            mm = m % 60;
        }
        else {
            hh = mm = 0;
        }
    }

    Time(int h = 0, int m = 0) {
        //validation before call in main( )
        if (((h >= 0) && (h < 24)) && ((m >= 0) && (m < 60))) {
            hh = h;
            mm = m;
        }
        else {
            hh = mm = 0;
        }
    }

    int getH() { return hh; }

    void setH(int h) {
        if ((h >= 0) && (h < 24)) {//validation before call in main( )
            hh = h;
        else
            hh = 0;
    }
```

```
int getM( ) {
    return mm;
}

void setM(int m) {
    if ((m >= 0) && (m < 60)) {//validation before call in main( )
        mm = m;
    else
        mm = 0;
}

void show( ) {//displaying method, may be replaced with getters
    cout << hh << ":" << mm << endl;
}

// prefixed ++ operator
Time& operator++( ) {
    mm++;
    if (mm == 60)
    {
        hh++;
        if (hh == 24)
            hh = 0;
        mm = 0;
    }
    return *this;
}

// postfixed ++ operator
Time operator++(int) {
    Time temp = *this;
    ++(*this);
    return temp;
}

// prefixed -- operator
Time& operator--( ) {
    if (mm == 0)
    {
        hh--;
        if (hh < 0)
            hh = 23;
        mm = 59;
    }
    else
        mm--;
    return *this;
}

// postfixed -- operator
Time operator--(int) {
    Time temp = *this;
    --(*this);
    return temp;
}
```

```
    // compound assignment += overloading
    Time& operator+=(int min) {
        int h1, m1;
        h1 = min / 60;
        m1 = min % 60;
        hh = (hh + h1 + ((mm + m1) / 60)) % 24;
        mm = (mm + m1) % 60;
        return *this;
    }

    // compound assignment -= overloading
    Time& operator-=(int min) {
        int h1, m1;
        h1 = min / 60;
        m1 = min % 60;
        hh = hh - h1;
        if (mm < m1)
            hh--;
        if (hh < 0)
            hh = 24 + hh;
        mm = mm - m1;
        if (mm < 0)
            mm = 60 + mm;
        return *this;
    }

    // addition + overloading
    Time operator+(Time tm) {
        Time rez;
        int nm = mm + tm.mm;
        rez.hh = hh + tm.hh + nm / 60;
        if (rez.hh > 23)
            rez.hh = rez.hh - 24;
        rez.mm = nm % 60;
        return rez;
    }

    // subtraction - overloading
    Time operator-(Time tm) {
        Time rez;
        if (mm < tm.mm)
        {
            rez.hh = hh - tm.hh - 1;
            rez.mm = 60 - (tm.mm - mm);
        }
        else
        {
            rez.hh = hh - tm.hh;
            rez.mm = mm - tm.mm;
        }
        if (rez.hh < 0)
            rez.hh = 24 + rez.hh;
        return rez;
    }
};//class
```

```
//main( )
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include "Time_inc_dec.h"

int main( )
{
    Time tm1(1, 29);
    cout << "Time 1: ";
    tm1.show( );
    cout << endl;

    Time tm2;
    tm2 = --tm1;
    cout << "Time 1 (tm2 = --tm1): ";
    tm1.show( );
    cout << "Time 2 (tm2 = --tm1): ";
    tm2.show( );
    cout << endl;

    tm1 = ++tm2;
    cout << "Time 1: (tm1 = ++tm2)";
    tm1.show( );
    cout << "Time 2: (tm1 = ++tm2)";
    tm2.show( );

    cout << endl;
    tm1 = tm2++;
    cout << "Time 1 (tm1 = tm2++): ";
    tm1.show( );
    cout << "Time 2 (tm1 = tm2++): ";
    tm2.show( );
    tm2--;
    cout << "Time 2 (tm2--): ";
    tm2.show( );

    cout << endl;
    Time tm3(22, 15);
    cout << "Time 3: ";
    tm3.show( );

    cout << endl;
    tm3 += 75;
    cout << "Time 3 (+=75): ";
    tm3.show( );
    tm3 -= 86;
    cout << "Time 3 (-=86): ";
    tm3.show( );

    Time tm4(23, 18);
    cout << endl;
    cout << "Time 4: ";
    tm4.show( );
    cout << "Time 3 + Time 4 : ";
    (tm3 + tm4).show();
```

```
    cout << "Time 3 - Time 4 : ";
    (tm3 - tm4).show( );
}//main
```

Note: Replace the show() method with getter methods called in main() or with an appropriate friend function

Ex. 5 - Supraîncărcarea indexării [] / Overloading the indexing operator [] (Dictionary class)

```
// v1. class Dictionary, using dynamic char arrays
// Dictionary.h

struct assoc {
    char* word;
    char* definition;
};

class Dictionary {
    assoc* dict;
    unsigned dim;
    unsigned next;    // the last position occupied in the dictionary
public:
    Dictionary(const unsigned size) {
        dict = new(nothrow) assoc[size];
        dim = size;
        next = 0;
    }

    ~Dictionary() { delete[] dict; }

    int getDim() const { return dim; }
    int getNext() const { return next; }
    int addWord(const char* cuv, const char* def);
    void setCapacity(unsigned new_size);

    const char* operator[](const char*) const;
    assoc operator[] (unsigned int) const;
};//Dictionary_class

int Dictionary::addWord(const char* cuv, const char* def)
{
    if (next < dim) {
        dict[next].word = new(nothrow) char[strlen(cuv) + 1];
        strcpy(dict[next].word, cuv);
        dict[next].definition = new(nothrow) char[strlen(def) + 1];
        strcpy(dict[next].definition, def);
        next++;
        return 0;
    }
    return 1;
}

void Dictionary::setCapacity(const unsigned new_size)
{
    assoc* p;
    p = new(nothrow)assoc[new_size];
```

```
    if (p != 0) {
        unsigned int nowordsmin = (new_size < dim) ? new_size : dim;
        for (unsigned i = 0; i < nowordsmin; i++)
            p[i] = dict[i];
        dim = new_size;
        delete[] dict;
        dict = p;
    }
}

const char* Dictionary::operator[](const char* cuv) const
{
    assoc* pAssoc = dict;
    for (unsigned i = 0; i < dim; i++) {
        if (strcmp(cuv, pAssoc->word) == 0)
            return pAssoc->definition;
        pAssoc++;
    }
    return notFound;
}

assoc Dictionary::operator[](unsigned int pos) const
{
    if (pos >= 0 && pos < dim)
        return dict[pos];

    return assoc{ NULL, NULL };
}

void show(Dictionary& d) {
    cout << endl << "Dictionary content:" << endl;
    for (int i = 0; i < d.getNext(); i++) {
        cout << "\t" << "Entry no " << i + 1 << "->" << d[i].word << ":"
            << d[i].definition << endl;
    }
}


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

const char* notFound = "Not found...";

#include "Dictionary.h"

int main()
{
    int size = 4;
    Dictionary dict(size);
    dict.addWord(
      "Car",
      "A vehicle with an engine, wheels, and seats for a small number of people.");
    dict.addWord(
```

```
        "Tree",
        "A plant that has a wooden trunk and branches that grow.");
    dict.addWord(
        "Animal",
        "A living thing that can move, eat, sense and react to the world.");

    show(dict);

    cout << "Animal: " << dict["Animal"] << endl;
    show(dict);

    dict.addWord("Student", "A person who is learning...");
    cout << "Student: " << dict["Student"] << endl;
    show(dict);

    cout << "Engineer: " << dict["Engineer"] << endl;

    dict.setCapacity(5);
    dict.addWord("Laptop", "A small computer");
    cout << "Student: " << dict["Student"] << endl;
    cout << "Laptop: " << dict["Laptop"] << endl;
    show(dict);

    dict.addWord(
        "Smartphone",
        "a mobile phone that can be used as a small computer");
    cout << endl << "Smartphone: " << dict["Smartphone"] << endl;
}//main


// v2. class Dictionary, using pseudocontainer string from STL
// Dictionary.h
#pragma once
struct assoc {
    string cuvant;
    string definitie;
};

class Dictionary {
    assoc* dict;
    unsigned dim;
    unsigned next;    // the last position occupied in the dictionary
public:
    Dictionary(const unsigned size) {
        dict = new(nothrow) assoc[size];
        dim = size;
        next = 0;
    }

    ~Dictionary() { delete[] dict; }

    int getDim() const { return dim; }

    int addWord(string cuv, string def);
    void setCapacity(unsigned new_size);
```

```
    string& operator[](const string) const;
    assoc operator[] (unsigned int) const;
};//Dictionary_class

int Dictionary::addWord(string cuv, string def)
{
    if (next < dim) {
        dict[next].cuvant = cuv;
        dict[next].definitie = def;
        next++;
        return 0;
    }
    return 1;
}

void Dictionary::setCapacity(const unsigned new_size)
{
    assoc* p;
    p = new(nothrow) assoc[new_size];
    if (p != 0) {
        unsigned int nowordsmin = (new_size < dim) ? new_size : dim;
        for (unsigned i = 0; i < nowordsmin; i++)
            p[i] = dict[i];
        dim = new_size;
        delete[] dict;
        dict = p;
    }
}

string& Dictionary::operator[](const string cuv) const
{
    assoc* pAssoc = dict;
    for (unsigned i = 0; i < dim; i++) {
        if (cuv == pAssoc->cuvant)
            return pAssoc->definitie;
        pAssoc++;
    }
    return notfound;
}

assoc Dictionary::operator[](unsigned int pos) const
{
    if (pos >= 0 && pos < dim)
        return dict[pos];
    return assoc{ "", "" };
}

void show(Dictionary& d) {
    cout << endl << "Dictionary content:" << endl;
    for (int i = 0; i < d.getDim(); i++) {
        cout << "\t" << "Entry no " << i + 1 << "->" << d[i].cuvant << ":"
            << d[i].definitie << endl;
    }
}

// main
```

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

string notfound = "Not found...";

#include "Dictionary.h"

int main()
{
    int size = 4;
    Dictionary dict(size);
    dict.addWord(
        "Car",
      "A vehicle with an engine, wheels, and seats for a small number of people.");
    dict.addWord(
        "Tree",
        "A plant that has a wooden trunk and branches that grow.");
    dict.addWord(
        "Animal",
        "A living thing that can move, eat, sense and react to the world.");

    show(dict);

    cout << "Animal: " << dict["Animal"] << endl;
    show(dict);

    dict.addWord("Student", "A person who is learning...");
    cout << "Student: " << dict["Student"] << endl;

    dict["Student"] = "A person who is learning at a college or university";

    cout << "Student: " << dict["Student"] << endl;
    show(dict);

    cout << "Engineer: " << dict["Engineer"] << endl;

    dict.setCapacity(5);
    dict.addWord("Laptop", "A small computer");
    cout << "Student: " << dict["Student"] << endl;
    cout << "Laptop: " << dict["Laptop"] << endl;
    show(dict);

    dict.addWord(
        "Smartphone",
        "a mobile phone that can be used as a small computer");
    cout << endl << "Smartphone: " << dict["Smartphone"] << endl;
}//main
```

Ex. 6 - Supraîncărcarea indexării [] / Overloading the indexing operator [] (Analize class)

```
/*4.a [ ] overloading for indexed access, friend classes*/
//Pers_analize.h
const int maxx = 31;//implicit nr. of characters
const int dim = 5;  //implicit nr. of objects
```

```
class Analize;
class Pers{
    //personal data
    char nume[maxx];
    double greutate;
    int varsta;
    friend class Analize;
public:
    void display( );
};// Pers

void Pers::display( ){
       if(this) cout << "\nPerson : " << nume << "\tWeight : " << greutate <<
"\tAge : "<<varsta;
    else cout << "\nNo such object...";
}

class Analize{
    Pers *p;
    int n;
public:
    //constructors
    Analize( ){
        n = dim;
        p = new (nothrow)  Pers[dim];
    }
    Analize(int nr){
        n = nr;
        p = new (nothrow)  Pers[n];
    }
    // [ ] overloading
    Pers* operator[ ](char *);
    Pers* operator[ ](double);
    Pers* operator[ ](int);

    void introdu( );
};//Analize

void Analize::introdu( ){
    cout << "\nEnter data :\n";
    for (int i = 0; i<n; i++)
    {
        cout << "Name" << (i + 1) << " : ";
        cin >> p[i].nume;
        cout << "Weight : ";
        cin >> p[i].greutate;
        cout << "Age : ";
        cin >> p[i].varsta;
    }
}//introdu

//name indexing
Pers* Analize:: operator[ ](char *nume){
    for (int i = 0; i<n; i++)
        if (strcmp(p[i].nume, nume) == 0)return &p[i];
```

```
    return NULL;
}//op[ ]nume

 //weight indexing
Pers* Analize:: operator[ ](double gr){
    for (int i = 0; i<n; i++)
        if (p[i].greutate == gr)return &p[i];
    return NULL;
}//op[ ]greutate

 //position indexing
Pers* Analize:: operator[ ](int index){
    //to verify before in main()
    if ((index >= 1) && (index <= n))return &p[index - 1];
    else { cout << "\nWrong index"; return NULL; }
}//op[ ]index

//main()
#include <iostream>
using namespace std;
#include "Pers_analize.h"

int main( ){
    char c;
    int nr;
    cout << "\nNr. of objects : ";
    cin >> nr;
    Analize t(nr);
    t.introdu( );
    while (1) {
        cout << "\nOption (w-weight, n-name, i-index, e-exit) ?";
        cin >> c;
        switch (toupper(c)) {
            case 'W':double g;
                cout << "Weight: ";
                cin >> g;
                t[g]->display( );
                break;
            case 'N':char n[maxx];
                cout << "Name: ";
                cin >> n;
                t[n]-> display( );
                break;
            case 'I':int i;
                cout << "Index: ";
                cin >> i;
                t[i]-> display( );
                break;
            case 'E':return 0;

        }//end switch-case
    }//end while
}//main
```

```
/*4.b variant 2: [ ] overloaded for indexed access, association relation */
//Pers_analize.h

const int maxx = 31;//implicit nr. of chars.
const int dim = 5;  //implicit nr. of objects

class Persoana {
    char nume[maxx];
    double greutate;
    int varsta;
public:
    Persoana( ) {
        strcpy(nume, "Unknown");
        greutate = 0.0;
        varsta = 0;
    }

    Persoana(char* nume, double greutate, int varsta) {
        strcpy(this->nume, nume);
        this->greutate = greutate;
        this->varsta = varsta;
    }
    char* getNume() { return nume; }
    double getGreutate() { return greutate; }
    int getVarsta() { return varsta; }

    void display( ) {
        cout << "\nName: " << nume;
        cout << "\nWight: " << greutate;
        cout << "\nAge: " << varsta;
    }
};//class

//Analize and Persoana in association
class Analize {
    Persoana* p;
    int n;
public:
    Analize( ) {
        p = new (nothrow)  Persoana[dim];
        n = dim;
    }

    Analize(int nr) {
        p = new (nothrow)  Persoana[nr];
        n = nr;
    }
    void introdu( ) {
        int j;
        char nume[maxx];
        double greutate;
        int varsta;
        for (j = 0; j < n; j++) {
            cout << "\nData for person : " << j + 1;
            cout << "\nName: ";
            cin >> nume;
```

```
            cout << "\nWeight: ";
            cin >> greutate;
            cout << "\nAge: ";
            cin >> varsta;
            p[j] = Persoana(nume, greutate, varsta);
        }
    }

    void operator[ ](char* nume) {
        int j;
        for (j = 0; j < n; j++)
            if (strcmp(nume, p[j].getNume()) == 0) p[j].display();
    }

    void operator[ ](double greutate) {
        int j;
        for (j = 0; j < n; j++)
            if (greutate == p[j].getGreutate()) p[j].display();
    }

    void operator[ ](int varsta) {
        int j;
        for (j = 0; j < n; j++)
            if (varsta == p[j].getVarsta()) p[j].display();
    }
};//class

//main( )
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include "Pers_analize.h"

int main( ) {
    int n;
    char c;
    char nume[maxx];
    double greutate;
    int varsta;
    cout << "\nHow many persons? ";
    cin >> n;
    Analize a(n);
    a.introdu();
    cout << "Searching after (a = age, w = weight, n = nume, e=exit)? ";
    cin >> c;
    switch (toupper(c)) {
        case 'A': { cout << "\nAge: "; cin >> varsta; a[varsta]; break; }
        case 'W': { cout << "\nWeight: "; cin >> greutate; a[greutate]; break; }
        case 'N': { cout << "\nName: "; cin >> nume; a[nume]; break; }
        case 'E': return 0;
    }//end switch
}//main
```

Ex. 7 - Supraîncărcarea =,+,-,*,() (clasa Matrix) / Overloading =,+,-,*,() (Matrix class)

```
/* Matrix class, overloads for operators =, +, -, *, () , data validation in main()
*/

class Matrix {
    int rows;
    int cols;
    int* elems;

public:
    Matrix( );
    Matrix(int rows, int cols);
    Matrix(const Matrix&);
    ~Matrix( ) { delete[ ] elems; }
    int& operator ( ) (int row, int col);
    Matrix& operator=(const Matrix&);
    //friend Matrix operator+(Matrix&, Matrix&);
    Matrix operator+(Matrix&);//method
    friend Matrix operator-(Matrix&, Matrix&);
    friend Matrix operator*(Matrix&, Matrix&);
    int getRows( ) { return rows; }
    int getCols( ) { return cols; }
    void init(int r, int c);
    void citire( );
    void afisare( );
};//Matrix

Matrix::Matrix( ) : rows(linii), cols(coloane)
{
    elems = new (nothrow) int[rows * cols];
}

Matrix::Matrix(int r, int c) : rows(r), cols(c)
{
    elems = new (nothrow) int[rows * cols];
}

Matrix::Matrix(const Matrix& m) : rows(m.rows), cols(m.cols)
{
    int n = rows * cols;
    elems = new (nothrow) int[n];
    for (int i = 0; i < n; i++)
        elems[i] = m.elems[i];
}

void Matrix::init(int r, int c) {
    rows = r;
    cols = c;
    elems = new (nothrow) int[rows * cols];
}

int& Matrix:: operator( )(int row, int col){
    return elems[row * cols + col];
}
```

```
Matrix& Matrix:: operator=(const Matrix& m) {
    if (this != &m) {
        delete[ ] elems;
        rows = m.rows; cols = m.cols;
        int n = rows * cols;
        elems = new (nothrow) int[n];
        for (int i = 0; i < n; i++)
            elems[i] = m.elems[i];
    }
    return *this;
}

/*
Matrix operator+(Matrix &p, Matrix &q) {
    Matrix m(p.rows, p.cols);
    for (int r = 0; r < p.rows; ++r)
        for (int c = 0; c < p.cols; ++c)
            m(r, c) = p(r, c) + q(r, c);
    return m;
}op+ friend */

Matrix Matrix::operator+(Matrix& p) {
    Matrix m(p.rows, p.cols);
    for (int r = 0; r < p.rows; ++r)
        for (int c = 0; c < p.cols; ++c)
            m(r, c) = p(r, c) + elems[r * cols + c];
    return m;
}//op+ methods

Matrix operator-(Matrix& p, Matrix& q) {
    Matrix m(p.rows, p.cols);
    for (int r = 0; r < p.rows; ++r)
        for (int c = 0; c < p.cols; ++c)
            m(r, c) = p(r, c) - q(r, c);
    return m;
}//op-

Matrix operator*(Matrix& p, Matrix& q) {
    Matrix m(p.rows, q.cols);
    for (int r = 0; r < p.rows; ++r)
        for (int c = 0; c < q.cols; ++c) {
            m(r, c) = 0;
            for (int i = 0; i < p.cols; ++i)
                m(r, c) += p(r, i) * q(i, c);
        }
    return m;
}//op*

void Matrix::citire( ) {
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++) {
            cout << "Elem. [" << i << "][" << j << "]  ";
            cin >> elems[cols * i + j];
        }
}//citire
```

```
void Matrix::afisare( ) {
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
            cout << elems[cols * i + j] << "\t";
        cout << endl;
    }
}//afisare

//main
#include<iostream>
#include "Matrix.h"
using namespace std;

const int linii = 2;
const int coloane = 3;

int main( ) {
    int i, j;
    Matrix m(linii, coloane);

    for (int i = 0; i < linii; i++)
        for (int j = 0; j < coloane; j++)
            m(i, j) = i + (j + 1) * 10;

    for (i = 0; i < linii; i++){
        for (j = 0; j < coloane; j++)
            cout << m(i, j) << "\t";
        cout << endl;
    }

    int l, c;
    cout << " ( ) overload" << endl;
    cout << "Line number (>=1): ";
    cin >> l;
    cout << "Column number (>=1): ";
    cin >> c;
    if ((l >= 1 && l <= m.getRows( )) && (c >= 1 && c <= m.getCols( )))
        cout << "Element m[" << l << "," << c << "]=" << m(l - 1, c - 1) << endl;
    else cout << "Wrong indexes!" << endl;
    cout << endl << "Copy constructor:" << endl;
    if (m.getRows( ) > 0 && m.getCols( ) > 0) {
        Matrix mcopy = m;
        cout << "Matrix \"mcopy\" is:" << endl;
        mcopy.afisare( );
    }
    else cout << "Wrong dimentions for copied matrix!" << endl;

    cout << endl << "New matrix \"n\" ";
    Matrix n(linii, coloane);
    cout << endl << "Enter the elements:" << endl;
    n.citire( );
    cout << endl << "Matrix \"n\" is:" << endl;
    n.afisare( );

    cout << endl << " = overloaded, copying matrix \"m\" in matrix \"n\"" << endl;
```

```
    if (m.getRows( ) == n.getRows( ) && m.getCols( ) == n.getCols( )) {
        n = m;
        //n.afisare( );
        for (i = 0; i < linii; i++) {
            for (j = 0; j < coloane; j++)
                cout << n(i, j) << "\t";//operator( ) overloaded
            cout << endl;
        }///end for
    }
    else
        cout << "Matrices have different dimensions, copying not possible" << endl;
    cout << endl << "New matrix \"m1\" ";
    Matrix m1(linii, coloane);
    cout << endl << "Enter the elements:" << endl;
    m1.citire( );
    cout << endl << "Matrix \"m1\" is:" << endl;
    m1.afisare( );

    Matrix m2(linii, coloane);
    cout << endl << "overloading +" << endl;
    if (m.getRows( ) == m1.getRows( ) && m.getCols( ) == m1.getCols( )) {
        m2 = m + m1;
        cout << endl << "The sum m+m1 is: " << endl;
    }
    m2.afisare( );
    cout << endl << "overloading - " << endl;
    if (m.getRows( ) == m1.getRows( ) && m.getCols( ) == m1.getCols( )) {
        m2 = m - m1;
        cout << endl << "The subtraction m-m1 is: " << endl;
    }
    m2.afisare( );

    /*matrix m has 2 lines and 3 columns so for multiplication m3 must have 3 lines
and 2 columns*/
    cout << endl << "The product \"m3\" matrix" << endl;
    cout << "Nr. of lines: ";
    cin >> l;
    cout << "Nr. of columns: ";
    cin >> c;
    Matrix m3;
    if (l > 0 && c > 0) m3.init(l, c);
    else cout << endl << "Negative dimensions! "
            << "The matrix will have 2 lines and 3 columns." << endl;
    m3.citire( );
    cout << endl << "Matrix \"m3\" is:" << endl;
    m3.afisare( );

    cout << endl << " * overloading";
    if (m.getCols( ) == m3.getRows( )){
        Matrix m4(m.getRows( ), m3.getCols( ));
        m4 = m * m3;
        cout << endl << "The product m*m3 is: " << endl;
        m4.afisare( );
    }
    else cout << endl << "Matrices cannot be multiplied.";
}//end_main
```

9.6. Lucru individual

1. Să se implementeze clasa *Complex* care supraîncarcă operatorii aritmetici cu scopul de a efectua adunări, scăderi, înmulțiri și împărțiri de numere complexe (folosind metode membre (+, -) și funcții friend (*, /)).
   Observație: numerele complexe vor fi definite ca având o parte reală și una imaginară, ambii coeficienți fiind reprezentați prin numere reale.
2. Pornind de la exemplul 1b, modificați supraîncărcarea operatorului –, care să permită scăderea a mai multor apariții din șirul inițial. Analizați funcționarea aplicației.
3. Pornind de la exemplul 4b, introduceți metode de tip set la atributele clasei *Persoana*, astfel încât introducerea datelor să fie făcută cu metoda *introdu( )* și metode setter în loc de constructor. Preluați opțiunile cu confirmare, la fel ca la exemplul 4a. Continuați la opțiune greșită dând un mesaj adecvat. Asigurați consistența supraîncărcării operatorilor de indexare (când nu se găsește obiectul). Considerați atributul *nume* de tip *char *,* alocarea spațiului fiind făcută în constructori. Definiți copy constructorul și supraîncărcați operatorul de asignare în cadrul clasei *Persoana*. Introduceți destructori în ambele clase. Considerați acum procesul de sortare după aceleași chei ca și la căutare cu afișarea rezultatelor în ordine descrescătoare. Verificați funcționalitatea elementelor introduse.
4. Pornind de la exemplul 5 referitor la matrici, verificați/implementați următoarele cerințe:
   a. citirea/scrierea unei matrici, unde dimensiunile sunt preluate de la tastatură
   b. testați toți operatorii supraîncărcați. Implementați variante în care se folosesc metode membre la supraîncărcare.
   c. afișați elementele de pe diagonala principală și secundară
   d. implementați operațiile cu matrici folsind metode membre.
5. Să se supraîncarce operatorul *[ ]* astfel încât, folosit fiind asupra unor obiecte din clasa *Departament*, ce conține un tablou de obiecte de tip *Angajat* (clasa *Angajat* conține atributele *nume* (șir de caractere) și *salariu* (double)), să returneze toată informația legată de angajatul al cărui număr de ordine este trimis ca și parametru.
6. Să se supraîncarce operatorii *new* și *delete* într-una din clasele cu care s-a lucrat anterior, în vederea alocarii și eliberarii de memorie pentru un obiect din clasa respectivă.
7. Să se scrie programul care consideră o clasă *MyClass* cu trei atribute de tip *int*. Clasa consideră pe baza mecanismului de supraîncărcare metode publice *int myFunction(…),* care în funcție de numărul de parametri primiți, returnează fie valoarea primită (1 parametru), fie produsul atributelor de intrare (>= 2 parametri). Instanțiați un obiect din clasă în *main( ),* setați atributele cu metode setter adecvate din clasă și afișați valorile la apelurile metodelor.
8. Să se scrie programul care utilizează o clasă numită *Calculator* și care are în componența sa metodele publice supraîncărcate:
   - *int calcul(int x)* care returnează pătratul valorii primite;
   - *int calcul(int x, int y)* care returnează produsul celor două valori primite;
   - *double calcul(int x, int y, int z)* care returnează rezultatul înlocuirii în formula *f(x,y,z) = (x-y)(x+z)/2.* a valorilor primite;

   Programul primește din linia de comandă toți parametrii necesari pentru toate aceste metode ale clasei.

   Considerați și cazul în care toate aceste metode sunt statice. E posibil să aveți în același timp metode publice statice și non-statice? Analizați și cazul în care clasa are 3 atribute *private* de tip *int, x, y, z,* care sunt modificate cu metode setter adecvate. Ce trebuie să modificați pentru a putea efectua operațiile cerute?
9. Să se definească clasa *Student* având ca și date membre private: *numele* (pointer șir de caractere), *note* (pointer de tip întreg) și *nr_note* (*int*). Clasa mai conține un constructor cu

parametri, un constructor de copiere/move, o metodă de supraîncărcare a operatorului de atribuire, o metodă de setare a notelor, o metodă de afișare a atributelor și un destructor. Să se instanțieze obiecte folosind constructorul cu parametri, un alt obiect ca și clonă va fi obținut folosind constructorul de copiere, afișând de fiecare dată atributele obiectului creat. Realizați o operație de copiere a unui obiect în alt obiect, ambele fiind create în prealabil. Afișati rezultatul copierii. Analizați metodele utilizate. Realizați o altă implementare în care numele e dat printr-un șir de caractere fix sau pseudo container *string,* iar *note* e un tablou de dimensiune fixă specificat printr-o constantă, atributul *nr_note* fiind eliminat.

10. Definți o clasă numită *Number* care are un atribut *private* de tip *double*. Clasa mai conține un constructor explicit vid și unul cu un parametru și o metodă accesor care returnează valoarea atributului din clasă. Scrieți o clasă numită *Mathemathics*, care supraîncarcă operatorii specifici operațiilor aritmetice elementare (+, -, *, /). Clasa are ca și atribut un obiect instanțiat din prima clasă. Operațiile aritmetice se efectuează asupra datelor obținute din obiectul curent și din unul primit ca parametru.
11. Considerați clasa *Fractie* care are două atribute întregi private *a* și *b* pentru numărător și numitor, două metode de tip *set()* respectiv *get()* pentru fiecare din atributele clasei. Declarați o metodă *simplifica()* care simplifică un obiect *Fractie*. Considerați o variabilă privată statică întreagă *icount*, care va fi inițializată cu *0* și incrementată în cadrul constructorilor din clasă. Definiți un constructor explicit fără parametri care inițializează *a* cu 0 și *b* cu 1, și un constructor explicit cu doi parametri care înainte de apel va verifica posibilitatea definirii unei fracții (*b*!=0). Definiți un destructor explicit care afișează un mesaj și contorul *icount*. Supraîncărcați operatorii de adunare, scădere, înmulțire și împărțire (+,-,*,/) a fracțiilor folosind funcții *friend* care și simplifică dacă e cazul rezultatele obținute. Instanțiați două obiecte de tip *Fractie* cu date citite de la tastatură. Afișați atributele inițiale ale obiectelor. Printr-o metodă accesor, afișați contorul *icount*. Efectuați operațiile implementate prin funcțiile *friend*, inițializând alte 4 obiecte cu rezultatele obținute. Afișați rezultatele și contorul după ultima operație folosind o metodă accesor adecvată.
12. Folosind aceeași clasa *Fractie*, definiți supraîncărcarea operatorilor compuși de asignare și adunare, scădere, înmulțire și împărțire (+=, -=, *=, /=) cu metode membre. Supraîncărcați operatorii de incrementare și decrementare post/prefixați care adună/scade valoarea *1* la un obiect de tip *Fractie* cu funcții membre (metode). Instanțiați două obiecte de tip *Fractie* cu date citite de la tastatură. Realizați o copie a lor în alte două obiecte. Efectuați operațiile compuse implementate prin metodele clasei folosind copiile obiectelor, asignând rezultatele obținute la alte 4 obiecte. Afișați cele 4 rezultate, iar apoi afișați rezultatele după incrementare/decrementare post/prefixată la cele 4 obiecte obținute.

9.7. Individual work

1. Implement a class called *Complex* that overloads the arithmetical operators (+, -, *, /) for performing the corresponding operations when applied to *Complex* instances (use both friend functions (*, /) and member methods (+, -)).
   Note: The *Complex* numbers will have a real and an imaginary part, both coefficients being represented as real numbers.
2. Modify the operator – overload, allowing multiple occurrences of the initial string from example 1 to be dropped. Test the functionalities.
3. Starting from examples *4b*, enter *set*-type methods for the attributes of the *Person* class, so entering data is done in *introdu( )* method using setter methods instead of the constructor. Read the options with confirmation as in 4a example. If a wrong option is entered, continue and display an adequate message. Ensure consistency indexing operators overloading (when no object is found, display a specific message). Consider the *name* as *char** attribute, the memory being allocated in the constructors. Define the copy constructor and overload

the assign operator within the *Person* class. Define destructors in both classes. Consider now a reverse sorting process based on the same keys as the ones used for searching. Verify all introduced functionalities.

4. Starting with the matrix 5-th example, verify/resolve the following tasks:
    a. reading/writing a matrix with dimensions from KB.
    b. test all the overloaded operators. Implement variants in which member methods are used for overloading of operators.
    c. displays the elements located on both diagonals.
    d. implement matrix operations using member methods.
5. Overload the *[ ]* operator for the *Department* class that contains an array of *Employee* objects (that has as attributes the *name* (character array) and the *salary* (float)). When the operator is applied to a *Department* object, it returns (or displays) all the data related to the *Employee* object with that index.
6. Overload the *new* and *delete* operators for one of the classes implemented before, in order to allocate / de-allocate the necessary amount of memory.
7. Write the program that considers a *MyClass* class with three *int*-type attributes. The class overloads the *public* method *int myFunction (…),* which depending on the number of parameters received, returns either the value received (1 parameter) or the product of the input parameters (>= 2 parameters). Instantiate an object of the class in *main(), set* the attributes using dedicated setter methods from the class, and display the values returned after calling *myFunction()*.
8. Write the program that implements a class called *Calculator* that has as overloaded public methods:
    - *int calcul(int x)* that returns the square value of *x*;
    - *int calcul(int x, int y)* that returns the product of *x* and *y*;
    - *double calcul(int x, int y, int z)* that returns the result of *f(x,y,z) = (x-y)(x+z)/2.;*

    The program receives the parameters from the command line.
    Consider the case when all the methods are static. Is it possible to have in the same time static and non-static public methods? Analyze the case in which the class has 3 *private* attributes of type *int, x, y, z,* that are modified with setter adequate methods. What must be modified to perform the required operations?
9. Define a class called *Student*, containing as private members: *name* (pointer to character array), *marks*(integer pointer) and *no_marks* (integer). The class also contains a constructor with parameters, a copy/move constructor, a method for assign operator overloading, a method for marks setting, a display method and a destructor. Create some objects using the constructor with parameters, another one using the copy constructor for cloning an object, displaying each time the attributes of the created object. Copy an object into another one. Display the result of the copy operation. Analyze the used methods. Make another implementation in which the name is given by a fixed character string or pseudo container string, and *marks* is a fixed size array specified by a constant, so the *no_marks* attribute can be removed.
10. Define a class named *Number* that has as private attribute a *double* variable. The class contains an explicit empty constructor, a constructor with a parameter and an accessor method that displays the value of the stored variable. Write a class called *Mathematics* that has as attribute an instance created from the first class and overloads the arithmetical operators (+, -, /, *). Each method calculates the appropriate result by considering the data obtained from the current object and from another one received as parameter.
11. Consider a class named *Fraction* that has two integer private attributes *a* and *b* for the denominator and nominator. Define *set( )* and *get( )* methods for the class's attributes. Declare a method named *simplify( )* that simplifies a fraction. Consider a *private* static integer variable *icount* that will be initialized with 0 and incremented in the constructors.

Declare two explicit constructors, one without parameters that initializes *a* with 0 and *b* with 1 and the other that has two integer parameters which will verify before if the fraction can be defined (b! =0). Define an explicit destructor that will display a message and the *icount* counter. Overload the arithmetic operators (+, -, *, /) using *friend* functions. The results will be displayed after being simplified. Instantiate 2 *Fraction* objects and read the appropriate data from the keyboard. Display the original values of the nominators and denominators. Using a specific accessor method, display the value of *icount*. Apply the implemented *friend* functions and *initialize* other 4 objects with the obtained results. Display the characteristics of the final objects and the value of *icoun*t.

12. Using the same *Fraction* class, overload the compound arithmetical operators (+=, -=, *=, /=) using member functions (methods). Overload the *pre*-de/incrementation and *post*-de/incrementation operators that will subtract/add the value 1 to an already existent *Fraction* object. Instantiate 2 *Fraction* objects and read the appropriate data from the keyboard. Copy the objects in the other 2 temporary objects. Apply the overloaded operators using the copied objects and *assign* the results to the other 4 objects. Display the characteristics of the final objects. Display the results obtained after decrementing/incrementing post/prefixed of the final 4 results.

10. Moștenirea simplă și multiplă

Simple and multiple inheritance.

10.1. Obiective

- Înțelegerea teoretică a noțiunii de moștenire simplă în limbajul C++; implementarea practică a diferitelor tipuri de moștenire simplă;
- Utilizarea facilităților moștenirii multiple;

10.2. Objectives

- Theoretical understanding of the simple C++ inheritance; practical implementation of different simple inheritance types;
- Using the multiple inheritance facilities;

10.3. Breviar teoretic

Moștenirea este un principiu al programării obiectuale care recomandă crearea de modele abstracte (clase de bază), care ulterior sunt concretizate (clase derivate) în funcție de problema specifică pe care o avem de rezolvat.

Aceasta duce la crearea unei ierarhii de clase și implicit la reutilizarea codului, la multe dintre probleme soluția fiind doar o particularizare a unor soluții deja existente.

Ideea principală în cadrul moștenirii este aceea că orice *clasă derivată* dintr-o *clasă de bază* "moștenește" toate atributele și metodele permise ale acesteia din urmă.

În procesul de moștenire, putem restricționa accesul la componentele clasei de bază sau putem modifica specificatorii de vizibilitate ai membrilor clasei, din punctul de vedere al clasei derivate:

Clasa de bază	**Moștenirea**		
	private (implicită)	**public**	**protected**
Membru „private"	inaccesibil	inaccesibil	inaccesibil
Membru „public"	private	public	protected
Membru „protected"	private	protected	protected

Moștenirea simplă se face după modelul:

```
class ClasaDerivata : [modificator_de_acces] ClasaDeBaza{
      //corp clasa
}
```

În cazul *moștenirii multiple* o clasă poate să moștenească două sau mai multe clase de bază. Sintaxa generală de specificare a acestui tip de moștenire este următoarea:

```
class ClasaDerivata : [modificator_de_acces1] ClasaDeBaza1,
                      [modificator_de_acces2] ClasaDeBaza2
                      [, modificator_de_accesN ClasaDeBazaN]{...};
```

Dacă o clasă este derivată din mai multe clase de bază, constructorii acestora sunt apelați în ordinea derivării, iar destructorii sunt apelați în ordinea inversă derivării.

Implicit `modificator_de_acces` la moștenire este `private`.

10.4. Theoretical brief

Inherintance is a principle of object oriented programming that recommends the creation of abstract models (base classes), which are subsequently materialized (derived classes) depending on the specific problem that has to be solved.

This leads to the creation of a hierarchy of classes and implicitly to code reusing, in many cases the solution being just a customization of another existing one.

The main idea in inheritance is that any *derived class* "inherits" from a *base class* all the permitted attributes and methods.

In the inheritance process, we can restrict access to core class components or we can modify the access specifiers in the derived class.

Base class	**Inheritance**		
	private (implicit)	**public**	**protected**
"private" member	inaccessible	inaccessible	inaccessible
"public" member	private	public	protected
"protected" member	private	protected	protected

The simple inheritance follows the following syntax:

```
class DerivedClass : [access_specifier] BaseClass{
      //class body
}
```

In the case *of multiple inheritance*, a class may inherit two or more base classes. The general syntax specifying this type of inheritance is as follows:

```
class DerivedClass : [access_specifier1] BaseClass1,
                     [access_specifier2] BaseClass2
                     [, access_specifierN BaseClassN]{...};
```

If a class is derived from more than one base class, their constructors are called in order of derivation, and the destructors are called in reverse order of derivation.

The implicit `access_specifier` for inheritance is `private`.

10.5. Exemple/ Examples

Ex. 1 - Moştenire publică, atribute protejate / Public inheritance, protected members

```
//Propagation of protected members in public inheritance
//Baza_deriv.h

class Baza {
protected:
    int i, j;
public:
    void setI(int a) {
        i = a;
    }
    void setJ(int b) {
        j = b;
    }
    int getI( ) {
        return i;
    }
    int getJ( ) {
        return j;
    }
};//Baza_class

class Derivata : public Baza {
public:
    int inmulteste( ) {
        return (i * j);    // correct, i and j remain protected
    }
};//Derivata_class

//main
#include <iostream>
using namespace std;
#include "Baza_deriv.h"

int main( ) {
    Derivata obiect_derivat;
    cout << "\n Attribute values (undefined): i, j: " << obiect_derivat.getI( )
        << ", " << obiect_derivat.getJ( ) << endl;
    //obiect_derivat.i = 5;       // wrong, i is protected not public
    obiect_derivat.setI(5);  // setI( ) is public from Baza
    obiect_derivat.setJ(17); // setJ( ) is public from Baza
    cout << "\n Attribute values (from Baza): i, j: " << obiect_derivat.getI( )
        << ", " << obiect_derivat.getJ( ) << endl;
    cout << "\n Product is: " << obiect_derivat.inmulteste( );//from Derivat
} //main
```

Ex. 2 - Moştenire protejată / Protected inheritance

```
// protected inheritance
//Baza_deriv.h
class Baza {
    int x;
protected:
    int y;
```

```
public:
    int z;
    Baza(int x = 0, int y = 0) {
        this->x = x;
        this->y = y;
    }//Baza
    int getX( ) {
        return x;
    }
    void setX (int a) {
        x=a;
    }
    int getY( ) {
        return y;
    }
};//Baza_class

class Derivata : protected Baza {
public:
    void do_this( ) {
        cout << "\n --------Derived class, do_this( )------";
        //cout << "\n The value of private x: " << x << endl; //private
        cout << "\n The value of private x: "<< getX( )<< endl; //public, 0
        cout << "\n The value of protected y: " << y << endl;// protected, 0
        cout << "\n The value of z: " << z << endl;// protected, undefined
        setX(5);
        cout << "x= " << getX( ) << endl; //ok, getX/setX( ) become protected
        y = 7; cout << "y= " << y << endl;//correct, y remains protected
        z = 9; cout << "z= " << z << endl;//correct, z becomes protected
    }//do_this
};//Derivata_class

//main

#include <iostream>
using namespace std;
#include "Baza_deriv.h"

int main( ) {
    int x, y;
    cout << "\n x= ";      cin >> x;
    cout << "\n y= ";      cin >> y;
    Baza ob1(x, y);
    Derivata ob2;//first the base constructor is called, x and y are implicit 0
    cout << "\n -From base class-\n private x: " << ob1.getX( )
        << "\n protected y: " << ob1.getY( ) << endl;

    //// getX( ) and getY( ) are protected in ob2, code below will not work
    //cout << "\n -From derived class-\n private x: " << ob2.getX( )
    //    << "\n protected y: " << ob2.getY( ) << endl;

    ob2.do_this( );// is public in the derived class, so is accessible
}//main
```

Ex. 3 - Moștenire privată / Private inheritance

```
// private inheritance
//Baza_deriv.h
class Baza {
protected:
    int a, b;
public:
    Baza( ) { a = 1, b = 1; }
    void setA(int a) { this->a = a; }
    void setB(int b) { this->b = b; }
    int getA( ) { return a; }
    int getB( ) { return b; }
    int aduna( ) { return a + b; }
    int scade( ) { return a - b; }
};//Baza_class

class Derivata : private Baza
{
public:
    int inmulteste( ) {
        return a * b;
    }
};//Derivata_class

//main
#include <iostream>
using namespace std;
#include "Baza_deriv.h"

int main( )
{
    Baza obiect_baza;
    cout << "\nBase class (init. values): " << obiect_baza.getA( ) << " "
        << obiect_baza.getB( ) << '\n';
    cout << "\nSum = " << obiect_baza.aduna( ); // ok aduna( ) is public
    cout << "\nDifference = " << obiect_baza.scade( ); //ok scade( ) is public
    obiect_baza.setA(2);
    obiect_baza.setB(3);
    cout << "\nBase class (modified values): " << obiect_baza.getA( ) << " "
        << obiect_baza.getB( ) << '\n';
    cout << "\nSum / Dif = " <<obiect_baza.aduna( ) << "/"
        << obiect_baza.scade()<<'\n';

    Derivata obiect_derivat;
    cout << "\nProduct (derived class, init. values) = "
        << obiect_derivat.inmulteste( )<< '\n'; // ok implicit values

    //// incorrect code below, aduna( ) becomes private
    //cout << "\nSum = " << obiect_derivat.aduna( );
    //// incorrect code below, scade( ) becomes private
    //cout << "\nDifference = " <<obiect_derivat.scade( );
}//main
```

Ex. 4 - Moștenire privată (caz particular) / Private inheritance (edge case)

```
//individual exceptions previous example – private inheritance
class Derivata : Baza
{
public:
    int inmulteste() {
        return a * b;
    }
    using Baza::aduna; //individual exception, allows access with derived objects
    Baza::scade;   //individual exception, allows access with derived objects
};//class_Derivata

//main
#include <iostream>
using namespace std;
#include "Baza_deriv.h"

int main( )
{
    Baza obiect_baza;
    cout << "\From base (initial values): " << obiect_baza.getA( ) << " "
        << obiect_baza.getB( ) << '\n’;

    // code below is correct, aduna( ) and scade( ) are public
    cout << "\nSum is (with initial val., base) = " << obiect_baza.aduna( );
    cout << "\Difference is (with initial val., base)= " << obiect_baza.scade( );

    obiect_baza.setA(2);
    obiect_baza.setB(3);
    cout << "\nFrom base (modified): " << obiect_baza.getA( ) << " "
        << obiect_baza.getB( ) << '\n’;
    cout << "\nSum/Difference after set= " <<obiect_baza.aduna() << "/"
        << obiect_baza.scade ( )<<'\n';
    Derivata obiect_derivat;
    cout << "\nProduct is (from derived with initial values) = "
        << obiect_derivat.inmulteste() << '\n';
    // corect val. implicite
   cout << "\nSum is (from derived with initial values, from base) = "
       << obiect_derivat.aduna( ); // correct aduna( ) keeps public from Baza
   cout << "\nDifferece is (from derived with initial values, from base) = "
       << obiect_derivat.scade( );  // correct, scade( ) keeps public from Baza
}//main
```

Ex. 5 - Moștenirea multiplă / Multiple inheritance

```
//Multiple inheritance
//Baza12_deriv.h
class Baza1 {
protected:
    int x;
public:
    int getX( ) {
        return x;
    }
    void arata( ) { cout << "From Baza1: x = " << x << endl; }
};//Baza1_class
```

```
class Baza2 {
protected:
    int y;
public:
    int getY( ) {
        return y;
    }
    void arata( ) { cout << "From Baza2: y = " << y << endl; }
};//Baza2_class

class Derivata : public Baza1, public Baza2 {
public:
    void setX(int i) { x = i; }
    void setY(int j) { y = j; }
    void arata( ) {
        cout << "\nFrom Derivata: \n";
        Baza1::arata( );
        Baza2::arata( );
    }
}; //Derivata_class

//main
#include <iostream>
using namespace std;
#include "Baza12_deriv.h"

int main( ) {
    Derivata obiect_derivat;
    obiect_derivat.setX(100); obiect_derivat.setY(200);
    cout << "B1: x is: " << obiect_derivat.getX( ) << endl;//Baza1
    cout << "B2: y is: " << obiect_derivat.getY( ) << endl;//Baza2
    obiect_derivat.arata( );//from Derivata class
}//main
```

Ex. 6 - Constructori în procesul de moștenire / Constructors in inheritance

```
// Constructors in inheritance
// Baza_deriv.h

class Base{
protected:
    int m_no;
public:
    Base(int no = 0) : m_no(no){
        cout << "\nBase class constructor";
    }
    int getM_no( )  {
        return m_no;
    }
};//Base_class

class Derived : public Base
{
protected:
```

```
    double m_cost;
public:
    //Call Base(int) constructor with value no
    Derived(double cost = 0, int no = 0) : Base(no), m_cost(cost) { }
    // Derived(double cost = 0, int no = 0) :  m_cost(cost) {
    //     m_no = no;//access to base class attribute
    //     cout << "\nDerived class constructor";
    // }//not recomended

    double getM_Cost( ) {
        return m_cost;
    }
    double multiplyNoCost( ) {
        return m_no * m_cost;
    }
};//Derived_class



//main
#include <iostream>
using namespace std;

#include "Baza_deriv.h"

int main( )
{
    Derived derived(1.3, 15);
    cout << "\nNo: " << derived.getM_no( ) << '\n';
    cout << "Cost: " << derived.getM_Cost( ) << '\n';
    cout << "Total_Cost: " << derived.multiplyNoCost( ) << '\n';
    return 0;
}//main
```

Ex. 7 - Problemă forme geometrice / Geometric shapes problem

```
//  Geometric shapes, problem 6 from individual work
//a)    No inheritance, ADT
//Shape_Circle.h
class Shape{
    char name[dim];
    int r;
public:
    Shape(char* s, int rr){
        strcpy(name, s);
        r = rr;
    }

    char* getName( )    {
        return name;
    }

    double areaCircle( ) {
        return pi * r * r;
    }
```

```
    double perCircle( ) {
        return 2. * pi * r;
    }
};//Shape_Circle

//Shape_Circle_Square.h
class Shape
{
    char name[dim];
    int r;
    int s;
public:
    Shape(char* str, int rr, int ss){
        strcpy(name, str);
        r = rr;
        s = ss;
    }
    char* getName( ) {
        return name;
    }
    double areaCircle( ) {
        return pi * r * r;
    }
    double perCircle( ) {
        return 2. * pi * r;
    }
    double areaSquare( ) {
        return (double)s * s;
    }
    double perSquare( ) {
        return 4. * s;
    }
};//Shape_Circle_Square

//other classes can be added, like Rectangle, etc.

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

const int dim = 30;
const double pi = 3.14;

#include "Shape_Circle.h"
//#include "Shape_Circle_Square.h"

//#include "Shape_Circle_Square_Rectangle.h"
// to be developed

int main( ) {
    int r;
    //int l;
    //…
    char s[dim];
    cout << "\nRead the name of the shape: ";
```

```
    cin >> s;
    cout << "\nSpecify the Circle radius: ";
    cin >> r;
    //cout << "\nSpecify the Square side: ";
    //cin >> l;

    Shape sh(s, r);
    //Shape sh(s, r, l);

    cout << "\nShape Name: " << sh.getName( );
    cout << "\nCircle Perimeter: " << sh.perCircle( );
    cout << "\nCircle Area: " << sh.areaCircle( );

    //cout << "\nSquare Perimeter: " << sh.perSquare( );
    //cout << "\nSquare Area: " << sh.areaSquare( );
    //…
}//main

//b)    Inheritance variant, OOP
//Shape.h
class Shape{
protected:
    char name[dim];
public:
    Shape(char* s) {
        strcpy(name, s);
    }
    char* getName( ) {
        return name;
    }
};//Shape_Base_class

//Circle.h
class Circle: public Shape{
    int r;
public:
    Circle(char* s, int rr) :Shape(s), r(rr)
    { }
    double area( ) {
        return pi * r * r;
    }
    double per( ) {
        return 2. * pi * r;
    }
};//Circle_Derived_class

//Square.h
class Square :public Shape{
    int s;
public:
    Square(char* n, int l) :Shape(n), s(l)
    { }
    double area( ) { return (double)s * s; }
    double per( ) { return 4. * s; }
};//Square_Derived_class
```

```
//Rectangle.h
class Rectangle :public Shape{
    int w, h;
public:
    Rectangle(char* s, int m, int l) :Shape(s), w(m), h(l)
    { }
    double area( ) {
        return (double) w * h;
    }
    double per( ) {
        return 2. * (w + h);
    }
};//Rectangle_Derived_class

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

const int dim = 30;
const double pi = 3.14;

#include "Shape.h"
#include "Circle.h"
#include "Square.h"
#include "Rectangle.h"

int main( ) {
    int r, n, a, b;
    char s[dim];
    cout << "How many shapes do you want to process? ";
    cin >> n;
    for (int i = 0; i < n; i++) {
        cout << "\n\nRead the name of the shape number" << i + 1 << ": ";
        cin >> s;
        if (_stricmp(s, "circle") == 0) {
        //if (stricmp(s, "circle") == 0) {//Other non VC++ compilers
            cout << "\nNow specify the radius: ";
            cin >> r;
            Circle c(s, r);
            cout << "\nName: " << c.getName();
            cout << "\nPerimeter: " << c.per();
            cout << "\nArea: " << c.area();
        }
        else if (_stricmp(s, "square") == 0) {
            cout << "\nNow specify the side: ";
            cin >> r;
            Square x(s, r);
            cout << "\nName: " << x.getName();
            cout << "\nPerimeter: " << x.per();
            cout << "\nArea: " << x.area();
        }
        else if (_stricmp(s, "rectangle") == 0) {
            cout << "\nSpecify the length: ";
            cin >> a;
            cout << "\nSpecify the width:";
```

```
            cin >> b;
            Rectangle d(s, a, b);
            cout << "\nName: " << d.getName();
            cout << "\nPerimeter: " << d.per();
            cout << "\nArea: " << d.area();
        }
        else
            cout << "\nInvalid shape name";
    }//for
}//main

/*c) Variant with selection menu, OOP inheritance, just the main( ) function,
classes the same as in b)*/
//main
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

const int dim = 30;
const double pi = 3.14;

#include "Shape.h"
#include "Rectangle.h"
#include "Circle.h"
#include "Square.h"

int main() {
    char ss[dim], cc;
    int r, l, a, b;

    while (1)
    {
        cout << "\nOption (c-circle, s-square, r-rectangle, e-exit) ?";
        cin >> cc;
        switch (toupper(cc))
        {
            case 'C':
            {
                cout << "\n\nEnter the name of the shape: ";
                cin >> ss;
                cout << "\nNow specify the radius: ";
                cin >> r;
/*
                Circle* cp = new Circle(ss, r);//memory allocation

                cout << "\nName: " << cp->getName();
                cout << "\nPerimeter: " << cp->per();
                cout << "\nArea: " << cp->area();
*/
                Circle c(ss, r);
                cout << "\nName: " << c.getName();
                cout << "\nPerimeter: " << c.per();
                cout << "\nArea: " << c.area();
                //delete cp;//free memory
                break;
            }//local space name
```

```
/*
            case 'S':
                cout << "\n\nEnter the name of the shape: ";
                cin >> ss;
                cout << "\nNow specify the side length: ";
                cin >> l;
                Square* xp = new Square(ss, l);
                cout << "\nName: " << xp->getName();
                cout << "\nPerimeter: " << xp->per();
                cout << "\nArea: " << xp->area();
                delete xp;
                break;
        //error C2360
*/

            case 'S':
            {
                cout << "\n\nEnter the name of the shape: ";
                cin >> ss;
                cout << "\nNow specify the side length: ";
                cin >> l;
                Square x(ss, l);
                cout << "\nName: " << x.getName();
                cout << "\nPerimeter: " << x.per();
                cout << "\nArea: " << x.area();
                break;
            }
/*eliminates error C2360, with a local namespace and works without allocation*/

            case 'R':
            {
                cout << "\n\nEnter the name of the shape: ";
                cin >> ss;
                cout << "\nSpecify the length: ";
                cin >> a;
                cout << "\nSpecify the width:";
                cin >> b;
                Rectangle* dp = new Rectangle(ss, a, b);
                cout << "\nName: " << dp->getName();
                cout << "\nPerimeter: " << dp->per();
                cout << "\nArea: " << dp->area();
                delete dp;
                break;
            }

            case 'E':
                cout << "\nExit";
                return 0;

            default:
                cout << "\nInvalid shape name";
        }//end switch-case
    }//end while
}//main
```

10.6. Lucru individual

1. Implementați programul prezentat în exemplul 3 și examinați eventualele erori date la compilare, dacă există, prin eliminarea comentariilor. Modificați programul astfel încât să se poată accesa din funcția *main( )*, prin intermediul obiectului *obiect_derivat*, și metodele *aduna( )* și *scade( )* din clasa de bază păstrând moștenirea de tip *private*. (vezi exemplul 4)
2. Folosind modelul claselor de la moștenire publică, exemplul 1, implementați două clase, astfel:
    a. clasa de bază conține metode pentru:
    b. codarea unui șir de caractere (printr-un algoritm oarecare - recomandat XOR cu o mască fixă) => public;
    c. afișarea șirului original și a celui rezultat din transformare = > public;
    d. clasa derivată conține o metodă pentru:
    e. scrirea rezultatului codării într-un fișier, la sfârșitul acestuia (întâi șirul inițial și apoi cel codat). Fiecare înregistrare are forma: *nr_inregistrare: șir_initial șir_codat*;

    Accesul la metodele ambelor clase se face prin intermediul unui obiect rezultat prin instanțierea clasei derivate. Programul care folosește clasele citește un șir de caractere de la tastatură și apoi, în funcție de opțiunea utilizatorului, afișează rezultatul codării și/sau îl scrie în fișier.
3. Să se implementeze o clasă de bază cu două atribute *protected* de tip întreg care conține o metodă mutator pentru fiecare atribut al clasei, parametrii metodelor fiind preluați în *main()* de la tastatură și metode accesor pentru fiecare atribut care returnează atributul specific. Să se scrie o a doua clasă, derivată din aceasta, care implementează operațiile matematice elementare: +, -, *, / asupra atributelor din clasa de bază, rezultatele fiind returnate de metode. Să se scrie o a treia clasă, derivată din cea de-a doua, care implementează în plus o metodă pentru extragerea rădăcinii pătrate dintr-un număr (*mul*, ca rezultat al operației * din prima clasă derivată) și de ridicare la putere (atât baza (*plus*, ca rezultat al operației + din prima clasă derivată) cât și puterea (*minus*, ca rezultat al operației - din prima clasă derivată) sunt trimiși ca și parametri). Verificați apelul metodelor considerând obiecte la diferite ierarhii.
4. Definiți o clasă numită *Triangle* care are 3 atribute protected pentru laturi și o metodă care calculează perimetrul unui triunghi ale cărui laturi sunt citite de la tastatură (se va valida existența unui triunghi înainte de a folosi un constructor adecvat) și apoi o clasă derivată în mod public din *Triangle, Triangle_extended*, care în plus, calculează și aria triunghiului. Folosind obiecte din cele două clase apelați metodele specifice.
5. Adaugați în clasa derivată din programul anterior o metodă care calculează înălțimea triunghiului. Apelați metoda folosind un obiect adecvat.
6. Definiți o clasă numită *Forme* care consideră o figură geometrică cu un *nume* ca și atribut de tip pointer la un șir de caractere. Clasa va conține un constructor fără parametri, unul cu parametri, copy/move constructor și se va supraîncărca operatorul de asignare. Clasa va avea și metode getter/setter și un destructor. Derivați în mod public o clasă *Cerc* care adaugă un atribut de tip *int* pentru rază și constructori adecvați considerând și atributele (*nume, raza*) și o metodă getter pentru rază și alte metode care calculează aria și perimetrul cercului de raza *r*, valoare introdusă în *main( )* de la tastatură. Similar definiți o clasă *Patrat* și *Dreptunghi* care permit determinarea ariei și perimetrului obiectelor specifice. Instanțiați obiecte din clasele derivate și afișați aria și perimetrul obiectelor. Datele specifice vor fi introduse de la tastatură. Definiți un obiect de tip *Cerc* cu parametri, care să îl copiați într-un nou obiect clonă la care să îi afișați atributele. Definiți un obiect de tip *Patrat* cu parametri și altul fără parametri. Asignați celui fără parametri obiectul instanțiat cu parametri și afișați atributele.
7. Considerați o clasă de bază *Cerc* definită printr-un atribut *protected raza*, care are un constructor cu parametri și o metodă care determină aria cercului. Considerați o altă clasă de

bază *Patrat* cu un atribut protected latura similar clasei *Cerc*. Derivați un mod public clasa *CercPatrat* care are un constructor ce apelează constructorii claselor de bază și o metodă care verifică dacă pătratul de latură *l* poate fi inclus în cercul de rază *r*. De asemenea clasa derivată determină și perimetrul celor două figuri geometrice. Instanțiați un obiect din clasa derivată (datele introduse de la tastatură), determinați aria și perimetrul cercului și al pătratului. Afișați dacă pătratul cu latura introdusă poate fi inclus în cercul de rază specificat.

8. Considerați clasa *Fracție* care are două atribute întregi *protected a* și *b* pentru numărător și numitor, două metode de tip *set( )* respectiv *get( )* pentru fiecare din atributele clasei. Declarați o metodă publică *simplifica( )* care simplifică un obiect *Fracție*. Definiți un constructor explicit fără parametri care inițializează *a* cu 0 și *b* cu 1, și un constructor explicit cu doi parametri care va putea fi apelat dacă se verifică posibilitatea definirii unei fracții (*b!=0*). Supraîncărcați operatorii de adunare, scădere, înmulțire și împărțire (+,-,*,/) a fracțiilor folosind metode membre care și simplifică dacă e cazul rezultatele obținute, apelând metoda *simplifica( )* din clasă. Definiți o clasă *Fractie_ext* derivată public din *Fracție*, care va avea un constructor cu parametri (ce apelează constructorul din clasa de bază). Supraîncărcați operatorii de incrementare și decrementare prefixați care adună/scade valoarea 1 la un obiect de tip *Fractie_ext* cu metode membre.
   Instanțiați două obiecte de tip *Fracție* fără parametri. Setați atributele obiectelor cu date citite de la tastatură. Afișați atributele inițiale ale obiectelor și noile atribute definite. Efectuați operațiile implementate prin metodele membre, inițializând alte 4 obiecte cu rezultatele obținute. Simplificați și afișați rezultatele. Instanțiați două obiecte de tip *Fractie_ext* cu date citite de la tastatură. Efectuați operațiile disponibile clasei, asignând rezultatele obținute la alte obiecte *Fracti_ext*. Simplificați și afișați rezultatele.

10.7. Individual work

1. Implement the program presented in the third example and examine the compilation errors if the existing comments are un-commented? Modify the program so the object *obiect_derivat* will be able to access the *aduna( )* and *scade( )* methods, from the *main( )* function keeping the *private* inheritance. (see example 4)
2. Using the classes from public inheritance example 1, implement 2 classes with the following requests:
   a. the base class has the methods for:
   b. coding a one-dimensional array of characters- string (using a user-defined algorithm - recommended XOR with a fix mask) => public;
   c. displaying the original and the coded array = > public;
   d. the derived class has a method for:
   e. appending the coded one-dimensional array at the end of a previously created text file (first the original string and then the encoded one). Each record respects the format: *record_number: initial_string coded_string*;

   The methods located in both classes are accessed using an instance of the derived class. The program that uses the classes reads from the keyboard a one-dimensional array of characters and allows the user to choose whether the input will be coded or will be appended at the end of the text file.
3. Implement a class that has 2 protected integer variables, that contains setter and getter methods for each attribute. Write a second class that inherits the first defined class and implements the elementary arithmetic operations (+, -, *, /) applied on the variables mentioned above the results being returned by methods. Write a third class derived from the second one that implements the methods for calculating the *square root* of a number (*mul* result obtained by the previous derived class) received as parameter, and for raising a numeric value to a certain power (the *base* (*plus*, result obtained by the previous derived class) and

the *power* (*minus*, result obtained by the previous derived class) are sent to the method as parameters). Verify the methods's calling using objects at different hierchies levels.

4. Define a class called *Triangle* with 3 attributes for the triangle sides that has a method that calculates the perimeter of the triangle with the sides introduced from the KB. Another class, *Triangle_extended*, is derived in public mode from *Triangle* and defines a method for calculating the triangle's area. Using objects from both classes are called the allowed methods. Verify before instantiating the objects the possibility to define a *Triangle* object.
5. Extend the second class from the previous problem with a method that can compute the triangle's height. Call the method using an adequate object.
6. Define a class *Shape* that considers a shape with a *name* attribute as a pointer to character string. The class will contain a constructor without parameters, one with parameters, copy/move constructors and the assign operator will be overloaded. The class will also have getter/setter methods and a destructor. Derive in public mode a *Circle* class that adds an *int* type attribute to the *radius* and appropriate constructors also considering the attributes (*name, radius*) and a getter method for the radius and other methods that calculate the area and perimeter of the circle of radius *r*, value entered in main () from the keyboard.
   In the same mode define other classes (*Square, Rectangle*, etc.) Instantiate objects from the derived classes and display the area and the perimeter. The data will be introduced from the KB. Define a *Circle* object with parameters introduced from the KB, to copy to a new object and display its attributes. Define a Square object with parameters and another without parameters. Assign the instantiated object with the parameters to the one without parameters and display the attributes.
7. Consider a base class *Circle* defined by a protected attribute *radius*, that contains a constructor with parameters and a method that will determine the area of the circle. Consider other base class, *Square* with a protected attribute, *length*, like *Circle* class. Derive in public mode the class *RoundSquare* from both classes that will contain a constructor which will call the constructors from base classes and a method that will verify if the square of length *l* may be included in the circle of radius *r*. The derived class will also determine the perimeter of both shapes. Instantiate an object from the derived class (data from the KB) and determine the area and perimeter of the composed shapes. Display a message if the square may be included in the circle.
8. Consider the *Fraction* class that has two protected attributes *a* and *b* for the nominator and denominator and two corresponding setter and getter methods for all attributes. Declare a public method named *simplify( )* that simplifies a fraction. Define an explicit constructor without parameters that initializes a with 0 and b with 1 and another explicit constructor with two integer parameters. For this constructor is verified if b!=0 before to be called. Overload the addition, subtraction, multiplication, and division operators (+, -, *, /) using member methods that simplify (if necessary) the obtained results. Define a class named *Fraction_ext* that inherits in a public mode the *Fraction* class and has a parameterized constructor that calls the constructor from the base class. Use member methods for overloading the pre-incrementation and pre-decrementation operators that will add/subtract 1 to the value of a *Fraction_ext* instance. Instantiate two *Fraction* objects without parameters. Set the attributes using values read from the keyboard. Perform the implemented operations and initialize the other four objects with the obtained results. Simplify the results. Instantiate two objects of *Fraction_ext* type with data from the KB. Perform the available operations. Assign the operation results to other existing *Fraction_ext* objects. Simplify and display the obtained results.

11. Clase și metode virtuale. Clase abstracte.

Virtual classes and methods. Abstract classes.

11.1. Obiective

- Înțelegerea moștenirii virtuale, a modalității de declarare, definire și utilizare a metodelor virtuale și a claselor abstracte în limbajul C++;
- Upcasting/downcasting in C++;

11.2. Objectives

- The understanding of C++ virtual inheritance, of virtual classes and methods' declaration, definition and usage, of abstract classes;
- Upcasting/downcasting in C++;

11.3. Breviar teoretic

Moștenirea virtuală are loc atunci când mai multe clase moștenesc virtual o clasă de bază comună. Ea constă în a crea o nouă instanță (virtuală) a clasei părinte în fiecare dintre clasele copil (derivate), independentă de celelalte instanțe generate în clasele "paralele". Obiectele din clasa de bază vor fi accesate de o altă clasă derivată prin mostenire multiplă (și hibridă) din clasele derivate virtual din clasa de bază folosind un singur obiect de acest tip din clasa de bază.

Metodele virtuale sunt definite folosind specificatorul ***virtual*** în clasa de bază și apoi sunt redefinite în clasele derivate. **Comportamentul specific apare atunci când sunt apelate printr-un pointer**. Un pointer al clasei de bază poate fi folosit pentru a indica spre orice clasă derivată din aceasta. Când un astfel de pointer indică spre un obiect derivat ce conține o metodă virtuală redefinită, compilatorul C++ determină care versiune a metodei va fi apelată, în funcție de tipul obiectului spre care indică acel pointer (către un obiect din clasa derivată, sau către un obiect din clasa de bază). Varianta cu care se va lucra se stabilește la apel, motiv pentru care se folosește denumirea de "legatură dinamică sau târzie" (dynamic (late) binding) spre deosebire de "legătura statică sau timpurie" (static (early) binding) în care varianta cu care se lucrează este stabilită în etapa de compilare. Astfel se pot executa versiuni diferite ale metodei virtuale în funcție de tipul obiectului (o formă de polimorfism dinamic) referit de pointer.

C++1y recomandă utilizarea specificatorului `override` după numele metodei care este redefinită în clasa derivată pentru o verificare a semnăturii metodei de către compilator.

Mecanismul de **moștenire de tip metadata** (inheritance metadata) e prezent în C++ fiind specificat selectiv și explicit prin `virtual` și `override`, față de limbajul Java unde el e implicit.

O **clasă abstractă** este o clasă care are cel puțin o **metodă virtuală pură** (adică, metoda nu are implementare). Aceste clase nu sunt utilizate direct, ci furnizează un schelet pentru alte clase ce vor fi derivate din acestea. De obicei, toate metodele membre ale unei clase abstracte sunt virtuale și au implementări vide, urmând să fie redefinite în clasele derivate. O clasă abstractă nu poate fi instanțiată (nu se pot declara obiecte având acest tip), în schimb se pot declara pointeri la o clasă abstractă.

Metodele statice și constructorii nu pot fi virtuali, dar destructorii pot, caz în care se garantează distrugerea obiectelor din clasele derivate.

Upcasting/downcasting

Un pointer de tipul unei clase de bază este compatibil ca și tip cu un pointer de tipul oricărei clase derivate din ea; pointerii de tipul clasei de bază pot fi folosiți pentru a accesa metode membre și din clasele derivate (moștenite din clasa de bază care au fost declarate virtuale) proces numit **upcasting**.

Pointerul de tipul clasei de bază poate fi utilizat cu obiecte din clasa de bază, în acest caz metodele clasei de bază sunt apelate.

La *upcasting*, obiectul derivat nu se schimbă. Prin *upcast* pointerul de tipul clasei de bază este asociat către un obiect din clasa derivată, dar se pot accesa numai metodele și datele membre care sunt definite în clasa de bază și cele redefinite, fiind virtuale. **Doar metodele virtuale sunt supuse legării dinamice**.

Un obiect derivat poate fi asignat unui obiect din clasa de bază, fără a specifica ceva suplimentar, proces numit **object slicing**, proces ce poate fi considerat periculos.

Un pointer de tipul unei clase derivate poate fi asociat unui un obiect din clasa de bază folosind un *cast* explicit. Procesul se numește **downcasting**:

```
Pozitie pp0(7,7);//base class object
Pozitie *p=new (nothrow) Punct(100,100,'Z');//base class pointer obtained
from a derived class with upcasting
 …
cout<<"\nDowncasting:\n ";
Punct *pdown;//derived pointer
pdown=(Punct*)&pp0;//base class object
pdown->afisare( );//base class method if pp0 refers to base class
pdown = (Punct*)p;//downcasting by derived class object
pdown->afisare( );//derived class method
```

11.4. Theoretical brief

Virtual inheritance occurs when multiple classes virtually inherit a common base class. It consists in creating a new (virtual) instance of the parent class in each of the child (derived) classes, independent of the other instances generated in the "parallel" classes. Base class objects will be accessed by another class derived through multiple (and hybrid) inheritance from the classes derived virtually from the base class using a single object created from it.

Virtual methods are methods defined with the ***virtual*** specifier in the base class and then redefined in the derived classes. **The specific behavior occurs when they are called through a pointer**. A base class pointer can be used to point to any class derived from it. When such a pointer points to a derived object containing a redefined virtual method, the C++ compiler determines which version of the method will be called, depending on the type of object to which that pointer points (to an object of the derived class or to an object of the base class). The variant to be worked with is established during the call, which is why the term "dynamic (late) binding" is used, as opposed to the "static (early) binding" in which the called variant is established during compiling. This way Different versions of the virtual method can be run depending on the object's type (a form of dynamic polymorphism).

C++1y recommends using the specifier `override` after the name of the method that is redefined in the derived class for informing the compiler to verify the method signature.

The C++ **metadata inheritance** is being selectively and explicitly specified by `virtual` and `override`, compared to the Java language where it is an implicit mechanism.

An **abstract class** is a class that has at least one **pure virtual method** (the method has no implementation). These classes are not used directly, but provide a skeleton for other classes to be derived from. Usually, all member methods of an abstract class are virtual and have empty implementations in order to be redefined in the derived classes. An abstract class cannot be instantiated (declaring objects of this type is not allowed). Instead, declaring pointers to an abstract class is allowed.

Static methods and constructors cannot be virtual but destructors can, in which case destruction of the derived class objects is guaranteed.

Upcasting/downcasting

A pointer to a base class is compatible as a type with a pointer to any class derived from it. The base class pointers can be used to access member methods from the derived classes (inherited from the base class, where they have been declared virtual) process called **upcasting**.

The base class pointer can be used to indicate objects of the base class, in which case the base class methods are called.

When *upcasting*, the derived object does not change. By *upcasting* the base class pointer is associated to an object in the derived class, but only member methods and data that are defined in the base class can be accessed, along with the **virtual redefined** ones. **Only virtual methods are subject to dynamic binding**.

A derived object can be assigned to a base class object without specifying anything additional. This is named **object slicing**, and is considered dangerous.

A pointer to a derived class can be assigned to a base class object using an explicit cast. The process is called **downcasting**:

```
Pozitie pp0(7,7);//base class object
Pozitie *p=new (nothrow) Punct(100,100,'Z');//base class pointer obtained
from a derived class with upcasting
 …
cout<<"\nDowncasting:\n ";
Punct *pdown;//derived pointer
pdown=(Punct*)&pp0;//base class object
pdown->afisare( );//base class method if pp0 refers to base class
pdown = (Punct*)p;//downcasting by derived class object
pdown->afisare( );//derived class method
```

11.5. Exemple/ Examples

Ex. 1 - Upcasting şi metode virtuale / Upcasting and virtual methods

```
//upcasting and virtual methods

/*Students.h*/

const int dim = 20;
class Student {
protected:
```

```
    char nume[dim];
    int anul;
public:
    Student(const char* n, int a) {
        strcpy(nume, n);
        anul = a;
    }
    void  setNume(const char* n) {
        strcpy(nume, n);
    }
    char* getNume() {
        return  nume;
    }
    void  setAnul(int  a) {
        anul = a;
    }
    int  getAnul() {
        return  anul;
    }
    virtual  void  mesaj() {
        cout << "Healthy student";
    }
};//Student

class  StudentCovid : public  Student
{
    int  grupa;
public:
    StudentCovid(char* n, int  a, int  g) :Student(n, a) {
        grupa = g;
    }
    void  setGrupa(int  g) {
        grupa = g;
    }
    int  getGrupa() {
        return  grupa;
    }
    void  mesaj() override {//virtual method redefinition
        cout << "Quarantined student";
    }
};//StudentCovid

/*Source.cpp*/
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace  std;
#include "Students.h"

int  main( ){
    char  maria[ ] = "Maria", ana[ ] = "Ana", ioana[ ] = "Ioana";
    Student  ob(maria, 1), ob1(ana, 2);
    StudentCovid  ob2(ioana, 1, 2113);
    cout << "\n Base class objects\n";
    cout << ob.getNume() << "  Year:  " << ob.getAnul() << "  ";
    ob.mesaj();
    cout << endl;
```

```
    cout << ob1.getNume() << "  Year:  " << ob1.getAnul() << "  ";  ob1.mesaj();


    cout << "\nDerived class object\n";
    cout << ob2.getNume() << "  Year:  " << ob2.getAnul() << "  "
        << ob2.getGrupa() << "  ";
    ob2.mesaj();

    Student* p;//base class pointer
    p = &ob;
    cout << "\nBase class pointer" << endl;
    cout << p->getNume() << "  Year:  " << p->getAnul() << "  "; p->mesaj();
    p = &ob2;//upcasting
    cout << "\nPointer to derived class upcasted to base class" << endl;
    cout << p->getNume() << "  Year:  " << p->getAnul() << "  ";

    // cout << "\nGroup:  "<< p->getGrupa( ) << endl;/*not accesible – belongs to
the derived class*/
    p->mesaj();//is virtual, dynamic binding
    p->setNume("Ioana-Delia");//only for common attributes
    p->setAnul(2);
    ob2.setGrupa(2214);//specific to derived class
    cout << "\nUpdate derived class object\n"<<ob2.getNume() << "  Year:  "
        << ob2.getAnul() << "  " << ob2.getGrupa() << "  ";
    ob2.mesaj();
    return  0;
}//main
```

Ex. 2 - Moștenirea virtuală hibridă / Virtual hybrid inheritance

```
//virtual hybrid inheritance

//variant char array dynamic allocation: Persons_type.h
const int dim = 20;

class Person {
protected:
    char* ptype;
    char* name;
public:
    Person(const char* t, const char* n)
    {
        ptype = new (nothrow) char[strlen(t) + 1];
        name = new (nothrow) char[strlen(n) + 1];
        strcpy(ptype, t);
        strcpy(name, n);
    }
    Person(const Person& s) {//copy constructor
        ptype = new (nothrow) char[strlen(s.ptype) + 1];
        strcpy(ptype, s.ptype);
        name = new (nothrow) char[strlen(s.name) + 1];
        strcpy(name, s.name);
        cout << "\nPerson copy constructor";
    }
    Person& operator= (const Person& x) {//assign
        if (this == &x) return *this;
```

```
        this->~Person();//not compulsory
        ptype = new (nothrow) char[strlen(x.ptype) + 1];//memory for type
        strcpy(ptype, x.ptype);
        name = new (nothrow) char[strlen(x.name) + 1];//memory for name
        strcpy(name, x.name);
        cout << "\nPerson assign";
        return *this;
    }

    virtual ~Person()
    //~Person()
    {
        delete[] ptype;
        delete[] name;
        cout << "\nPerson destructor";
    }
    const char* getName() { return name; }
    void setName(const char* n) { strcpy(name, n); }
    const char* getPtype() { return ptype; }
    void setPtype(const char* t) { strcpy(ptype, t); }
}; //Person

class Student : virtual public Person
{
protected:
    int creditHours;
public:
    Student(const char* n, int ch) : Person("Student", n), creditHours(ch)
    { }
    Student(const Student& s) : Person(s) {//copy constructor
        creditHours = s.creditHours;
        cout << "\nStudent copy constructor";

    }
    Student& operator= (const Student& x) {
        if (this == &x) return *this;
        ptype = new (nothrow) char[strlen(x.ptype) + 1];//memory for type
        strcpy(ptype, x.ptype);
        name = new (nothrow) char[strlen(x.name) + 1];//memory for name
        strcpy(name, x.name);
        creditHours = x.creditHours;
        cout << "\nStudent assign";
        return *this;
    }
    int getCreditHours() { return creditHours; }
    void setCreditHours(int ch) { creditHours = ch; }
    ~Student() {
        cout << "\nStudent destructor";
    }
}; //Student

class Employee : virtual public Person
{

protected:
    int vacationHours;
```

```
public:
    Employee(const char* n, int vh) : Person("Employee", n), vacationHours(vh)
    { }
    Employee(const Employee& s) : Person(s) {//copy constructor
        vacationHours = s.vacationHours;
        cout << "\nEmployee copy constructor";
    }
    Employee& operator= (const Employee& x) {
        if (this == &x) return *this;
        ptype = new (nothrow) char[strlen(x.ptype) + 1];//memory for type
        strcpy(ptype, x.ptype);
        name = new (nothrow) char[strlen(x.name) + 1];//memory for name
        strcpy(name, x.name);
        vacationHours = x.vacationHours;
        cout << "\nEmployee assign";
        return *this;
    }
    int getVacationHours() { return vacationHours; }
    void setVacationHours(int vh) { vacationHours = vh; }
    ~Employee() {
        cout << "\nEmployee destructor";
    }
}; //Employee

class StudentEmployee : public Student, public Employee
{
public:
    StudentEmployee(const char* n, int ch, int vh) :
        Person("StudentEmployee", n),
        Student("ignored", ch),
        Employee("ignored", vh)
    { }

    //copy constructor
    StudentEmployee(const StudentEmployee& s) : Person(s), Employee(s), Student(s){
        cout << "\nStudentEmployee copy constructor";
    }

    StudentEmployee& operator= (const StudentEmployee& x) {
        if (this == &x)
            return *this;
        ptype = new (nothrow) char[strlen(x.ptype) + 1];//memory for type
        strcpy(ptype, x.ptype);
        name = new (nothrow) char[strlen(x.name) + 1];//memory for name
        strcpy(name, x.name);
        creditHours = x.creditHours;
        vacationHours = x.vacationHours;
        cout << "\nStudentEmployee assign";
        return *this;
    }

    ~StudentEmployee() {
        cout << "\nStudentEmployee destructor";
    }
};//StudentEmployee
```

```
//main
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include "Persons_type.h"

int main( )
{
    StudentEmployee se("Popescu", 60, 15);
    cout << "Name: " << se.getName() << " Type: " << se.getPtype()
        << " Credits: " << se.getCreditHours()
        << " Vacation: " << se.getVacationHours() << endl;

    StudentEmployee clone_se = se;//copy constructor, all hierachy
    cout << "\nClone_object\nName: " << clone_se.getName()
        << " Type: " << clone_se.getPtype()
        << " Credits: " << clone_se.getCreditHours()
        << " Vacation: " << clone_se.getVacationHours() << endl;
    se.setName("Ionescu");
    se.setPtype("Other_SE");
    se.setCreditHours(33);
    se.setVacationHours(77);
    clone_se = se;//assign only StudentEmployee
    cout << "\nSame modified Clone_object by assign\nName: " << clone_se.getName()
        << " Type: " << clone_se.getPtype()
        << " Credits: " << clone_se.getCreditHours()
        << " Vacation: " << clone_se.getVacationHours() << endl;
}//main
```

Ex. 3 - Moștenire simplă, tipuri de casting / Simple inheritance, types of casting

```
//Simple inheritance, up/down-casting, virtual methods – didactic example
//Poz_punct.h

// clasa de baza
class Pozitie {
protected:
    int x, y;
public:
    Pozitie(int = 0, int = 0);
    Pozitie(const Pozitie&);
    ~Pozitie( );
    //void afisare();
    //void deplasare(int, int);

    virtual void afisare( );
    virtual void deplasare(int, int);
};//CB

// constructor
Pozitie::Pozitie(int abs, int ord) {
    x = abs; y = ord;
    cout << "Base class constructor \"Pozitie\", ";
    afisare();
}
```

```
//copy constructor
Pozitie::Pozitie(const Pozitie& p) {
    x = p.x;
    y = p.y;
    cout << "Base class copy constructor \"Pozitie\", ";
    afisare();
}

// destructor
Pozitie::~Pozitie( ) {
    cout << "Destructor base class \"Pozitie \", ";
    afisare();
}

void Pozitie::afisare( ) {
    cout << " Base class display: coords.: x = " << x
        << ", y = " << y << "\n";
}

void Pozitie::deplasare(int dx, int dy) {
    cout << "Base class movement" << endl;
    x += dx; y += dy;
}

// derived class
class Punct : public Pozitie {
    int vizibil;// flag
    char culoare;
public:
    Punct(int = 0, int = 0, char = 'A');
    Punct(const Punct&);
    ~Punct();
    void arata( ) { vizibil = 1; }
    void ascunde( ) { vizibil = 0; }
    void coloreaza(char c) { culoare = c; }

    void deplasare(int, int)  override;
    void afisare( )  override;
};//CD

// constructor
Punct::Punct(int abs, int ord, char c) :Pozitie(abs, ord) {
    vizibil = 0;
    culoare = c;
    cout << "Derived class constructor \"Punct\", ";
    afisare( );//CD
}

// copy constructor
Punct::Punct(const Punct& p) :Pozitie(p.x, p.y) {
    vizibil = p.vizibil;
    culoare = p.culoare;
    cout << "Copy constructor derived class \"Punct\", ";
    afisare();//CD
}
```

```
// destructor
Punct::~Punct() {
    cout << "Destructor derived class \"Punct\", ";
    afisare();//CD
}

// moving function redefinition
void Punct::deplasare (int dx, int dy) {
    if (vizibil) {
        cout << " Derived class display\n";
        x += dx;
        y += dy;
        afisare();//derived class method
    }
    else {
        x += dx;
        y += dy;
        cout << "Derived class display using the base class method\n";
        Pozitie::afisare();
    }
}

// displaying method redefinition
void Punct::afisare( ){
    cout << "Pozitie: x = " << x << ", y = " << y;
    cout << ", culoare: " << culoare;
    if (vizibil) cout << ", vizibil \n";
    else cout << ", invizibil \n";
}

// program de test
#include <iostream>
using namespace std;
#include "Poz_punct.h"

int main( ) {
    Pozitie pp0(7, 7);//base class object
    cout << "\n Base class methods \n";
    pp0.afisare();
    pp0.deplasare(6, 9);
    pp0.afisare();
    cout << "\n Derived class methods \n";
    Punct p0(1, 1, 'v');//derived class object
    p0.afisare();
    Punct p1(p0);
    p1.arata();
    p1.deplasare(10, 10);
    cout << "\nUpcasting - objects:\n";
    pp0 = p0;//upcasting by objects
    pp0.afisare();
    cout << "\nUpcasting - pointers:\n ";
    Pozitie * p;//base class pointer
    p = new Punct(100, 100, 'Z');//derived object to base class pointer
    //cout<<"\nBase class display: \n"; non virtual
    cout << "\nDerived class display: "
        << "derived class object if virtual, else base class\n";
```

```
    p->afisare();//invizibil
    p = &pp0;
    cout << "\nBase class display: always base class object\n";
    p->afisare();
    p = &p1;
    cout << "\nDerived class display: "
        << "derived class object if virtual, else base class\n";
    p->afisare();
    Punct * pp;
    pp = &p1;
    cout << "\nDerived class display: always derived class object\n";
    pp->afisare();
    cout << "\n Derived class movement with 10, 10 \n";
    pp->deplasare(10, 10);
    cout << "\nDerived class display: derived class object with ascunde()\n";
    pp->ascunde();
    pp->afisare();
    cout << "\n Derived class movement with 10, 10 and ascunde()\n";
    pp->deplasare(10, 10);
    cout << "\nBase class display: "
        << "derived object always displayed with base class method\n";
    pp->Pozitie::afisare();
    cout << "\nDowncasting:\n ";
    Punct * pdown;//derived pointer
    pdown = (Punct*)&pp0;//downcasting by base class object
    cout << "\nBase class display: "
        << "base class object using a derived pointer, else derived class \n";
    pdown->afisare();
    pdown = (Punct*)p;//downcasting by derived class object
    cout << "\n Derived class display" << endl;
    pdown->afisare();
    return 0;
}//main
```

Ex. 4 - Definirea și utilizarea de metode virtuale / Defining and using virtual methods

```
//defining and using virtual methods
//Vehicul.h

class Vehicul {
protected:
    int roti;
    float greutate;
    int incarcatura_pasageri;
    int volum_date;

public:
    virtual void mesaj() {
        cout << "Vehicul message\n";
    }
    int getRoti() {
        return roti;
    }
    virtual float getGreutate() {
        return greutate;
    }
```

```
    void setRoti(int r) {
        roti = r;
    }
    void setGreutate(float g) {
        greutate = g;
    }
    int getIncarcatura_pasageri() {
        return incarcatura_pasageri;
    }
    void setIncarcatura_pasageri(int ip) {
        incarcatura_pasageri = ip;
    }
};//Vehicul

class Automobil : public Vehicul {
public:
    void mesaj() override {
        cout << "Automobil message\n";
    }
    float getGreutate() override{
        cout << " Auto-greutate ";
        return greutate;
    }
};//Automobil

class Camion : public Vehicul {
    float incarcatura_utila;
public:
    float getGreutate() override {
        cout << " Camion-greutate ";
        return greutate;
    }
    float getIncarcatura_utila() {
        return incarcatura_utila;
    }
    void setIncarcatura_utila(float iu) {
        incarcatura_utila = iu;
    }
};//Camion

class Barca : public Vehicul {
public:
    void mesaj() override {
        cout << "Barca message\n";
    }
    float getGreutate() override {
        cout << " Barca-greutate ";
        return greutate;
    }
    int getVolum_date() {
        return volum_date;
    }
    void setVolum_date(int v) {
        volum_date = v;
    }
};//Barca
```

```
//main
#include<iostream>
using namespace std;
#include "Vehicul.h"

int main( ) {
    // direct call through specific methods
    Vehicul monocicleta;
    Automobil ford;
    Camion semi;
    Barca barca_de_pescuit;

    monocicleta.mesaj();
    ford.mesaj();
    semi.mesaj();//from Vehicul
    barca_de_pescuit.mesaj();

    // Call via a specific pointer
    Vehicul* pmonocicleta;
    Automobil* pford;
    Camion* psemi;
    Barca* pbarca_de_pescuit;

    cout << "\n";
    pmonocicleta = &monocicleta;
    pmonocicleta->mesaj();

    pford = &ford;
    pford->mesaj();

    psemi = &semi;
    psemi->mesaj();//from base class

    pbarca_de_pescuit = &barca_de_pescuit;
    pbarca_de_pescuit->mesaj();

    // call using a base class pointer
    cout << "\n";
    pmonocicleta = &monocicleta;
    pmonocicleta->mesaj();//Vehicul

    pmonocicleta = &ford;//upcasting
    pmonocicleta->mesaj();//Automobil

    pmonocicleta = &semi;//upcasting
    pmonocicleta->mesaj();//Camion- Vehicul

    pmonocicleta = &barca_de_pescuit;//upcasting
    pmonocicleta->mesaj();//Barca
    //base class pointer for calling the defined methods
    Vehicul* pb;
    pb = &monocicleta;
    pb->setRoti(1);
    pb->setGreutate(10);
```

```
    cout << "\nBase class attributes: Wheels= " << pb->getRoti()
        << " Weight= " << pb->getGreutate() << endl;
    pb = &ford;
    pb->setRoti(4);
    pb->setGreutate(1000);
    pb->setIncarcatura_pasageri(200);
    cout << "\nDerived class attributes - Automobil: Wheels= " << pb->getRoti()
        << " Weight = " << pb->getGreutate()
        <<" Passengers= "<< pb->getIncarcatura_pasageri() << endl;
    pb = &barca_de_pescuit;
    pb->setRoti(0);
    pb->setGreutate(1000);
    pb->setIncarcatura_pasageri(200);
    barca_de_pescuit.setVolum_date(300);
    //pb->setVolum_date(300);//belongs to the derived class
    cout << "\nDerived class attributes - Barca_de_pescuit: Wheels= "
        << pb->getRoti() << " Weight= " << pb->getGreutate()
        << " Passengers= " << pb->getIncarcatura_pasageri() << endl;
    cout << "\t Volume (object access) = " << barca_de_pescuit.getVolum_date()
        << endl;
    return 0;
}//main
```

Homework: Associate the pointer of the base class to all the objects obtained from the derived classes and check the functionality. Perform the same operations considering the virtual methods from the base class as non-virtual and the variant all virtual methods in the base class. Analyze the results.

Ex. 5 - Clase abstracte și metode pur virtuale / Abstract classes and pure virtual methods

```
//Abstract classes and pure virtual methods
//Header.h
enum Color {Co_red, Co_green, Co_blue};

// abstract base class
class Shape {

protected:
    int xorig;
    int yorig;
    Color co;

public:
    Shape(int x, int y, Color c) : xorig(x), yorig(y), co(c) { }

    virtual ~Shape( ) { } // virtual destructor
    virtual void draw( ) = 0; // pure virtual method
};//Shape

// class Line (between origin and a destination point)
class Line : public Shape {
    int xdest, ydest;
public:
    Line(int x, int y, Color c, int xd, int yd) :
                xdest(xd), ydest(yd), Shape(x, y, c) { }
```

```
    ~Line( ) {cout << "~Linie\n";} // virtual destructor

    void draw( ) override // implementation of pure virtual method
    {
        cout << "Linie" << "(";
        cout << xorig << ", " << yorig << ", " << int(co);
        cout << ", " << xdest << ", " << ydest;
        cout << ")\n";
    }
};//Line

// Class Circle
class Circle : public Shape {
    int raza;

public:
    Circle(int x, int y, Color c, int r) : raza(r), Shape(x, y, c) { }

    ~Circle( ) { cout << "~Cerc\n"; } // virtual destructor
    void draw( ) override // implementation of pure virtual method
    {
        cout << "Cerc" << "(";
        cout << xorig << ", " << yorig << ", " << int(co);
        cout << ", " << raza;
        cout << ")\n";
    }
};//Circle

// Clasa Text : text
class Text : public Shape {
    char* str;
public:
    Text(int x, int y, Color c, const char* s) : Shape(x, y, c)
    {
        str = new (nothrow)  char[strlen(s) + 1];
        strcpy(str, s);
    }

    ~Text( ) {delete [ ] str; cout << "~Text\n";} // virtual destructor

    void draw( ) override // implementation of pure virtual method
    {
        cout << "Text" << "(";
        cout << xorig << ", " << yorig << ", " << int(co);
        cout << ", " << str;
        cout << ")\n";
    }
};//Text

//main( )
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
#include "Header.h"
const int n = 5;
```

```
int  main( )
{
    int i;
    Shape* sptrs[n];//base class pointer array, used through upcasting
//upcasting
    sptrs[0] = new (nothrow)  Line(1, 1, Co_blue, 4, 5);//upcasting
    sptrs[1] = new (nothrow)  Line(3, 2, Co_red, 9, 75);
    sptrs[2] = new (nothrow)  Circle(5, 5, Co_green, 3);
    sptrs[3] = new (nothrow)  Text(7, 4, Co_blue, "Hello!");
    sptrs[4] = new (nothrow)  Circle(3, 3, Co_red, 10);
    for (i = 0; i < n; i++)
        sptrs[i]->draw();
    for (i = 0; i < n; i++)
        delete sptrs[i];
    return 0;
}//main
```

Homework: Insert a copy constructor in the Text class and overload the assign operator. Test the functionality using objects from the Text class.

11.6. Lucru individual

1. În cazul exemplului 3 (care exemplifică moștenirea simplă, cu clasa de bază *Pozitie* și derivată *Punct*) se cer următoarele:
    a. urmariți și verificați ordinea de apel pentru constructori/destructori
    b. extindeți funcția *main( )* pentru a utiliza toate metodele din clasa de baza și din clasa derivată
    c. introduceți o nouă clasă *Cerc* (date-atribute și metode), derivată din clasa *Pozitie*
    d. scrieți un program ce utilizează aceste clase.
2. La exemplul al patrulea extindeți clasa de bază cu alte metode virtuale, redefinite în clasele derivate, cum ar fi metode *get( )* și *set( )* pentru greutatea vehiculului (variabila *greutate*).
3. Să se scrie un program C++ în care se definește o clasă *Militar* cu o metodă publică virtuală *sunt_militar( )* care indică apartenența la armată. Derivați clasa *Militar* pentru a crea clasa *Soldat* și clasa *Ofiter*. Derivați mai departe clasa *Ofiter* pentru a obtine clasele *SubLocotenent, Locotenent*, *Capitan, Maior, Colonel, General*. Redefiniți metoda *sunt_militar( )* pentru a indica gradul militar pentru fiecare clasă specifică. Instanțiați fiecare clasă *Soldat, Locotenent,...,General*, și apelați metoda *sunt_militar( ).*
4. Declarați o clasă *Animal*, care va conține *o metodă pur virtuală*, *respira( )* și două metode virtuale *manaca( )* și *doarme( )*. Derivați în mod *public* o clasă *Caine* și alta *Peste*, care vor defini metoda pur virtuală, iar clasa *Caine* va redefini metoda *mananca( )*, iar *Peste* metoda *doarme( )*. Instanțiați obiecte din cele două clase și apelați metodele specifice. Definiți apoi un tablou de tip *Animal*, care va conține obiecte din clasele derivate, dacă e posibil. Dacă nu, găsiți o soluție adecvată.
5. Definiți o clasă *Shape* care consideră o formă cu un atribut *nume* ca șir de caractere. Clasa va conține un constructor cu parametri și *metode virtuale pure* pentru *area(), perimeter()* și *display().* Derivați în modul *public* o clasă *Circle* care adaugă un atribut de tip *int r* ca și rază și un constructor adecvat având în vedere atributele (*nume, raza*) și definește metodele care calculează aria, perimetrul cercului cu raza *r*, și *display()* care le afișează,valoare pentru *r* fiind introdusă în *main( )* de la tastatură.
   În același mod definiți altă clasă *Square* care adaugă un atribut de tip *int l* ca latură și metode care calculează aria și perimetrul pătratului de latură *l,* valoare pentru *l* fiind introdusă în *main()* de la tastatură și *display()* care le afișează. În *main( )* folosind un meniu cu instrucțiunea

switch-case cu opțiuni pentru *Circle, Square* și *Exit*, instanțiați obiecte din clasele derivate și afișați aria și perimetrul până când este selectat *Exit*. Datele vor fi introduse de la tastatură.

6. Definiți o clasă abstractă care conține 3 declarații de *metode pur virtuale* pentru concatenarea, întrețeserea a două șiruri de caractere și inversarea unui șir de caractere primit ca parametru. O subclasă implementează corpurile metodelor declarate în clasa de bază. Instanțiați clasa derivată și afișați rezultatele aplicării operațiilor implementate în clasă asupra unor șiruri de caractere citite de la tastatură. Examinați eroarea dată de încercarea de a instantia clasa de bază.
7. Definiți o clasă numită *Record* care stochează informațiile aferente unei melodii (artist, titlu, durata). O clasă abstractă (*Playlist*) conține ca atribut privat un pointer spre un șir de obiecte de tip înregistrare. În constructor se alocă memorie pentru un număr de înregistrări definit de utilizator. Clasa conține metode accesor și mutator pentru datele componente ale unei înregistrări și o metodă pur virtuală cu un parametru (abstractă), care poate ordona șirul de înregistrări după un anumit criteriu codat în valoarea întreagă primită ca parametru (1=ordonare după titlu, 2=ordonare după artist, 3=ordonare după durată). Într-o altă clasă (*PlaylistImplementation*) derivată din *Playlist* se implementează corpul metodei abstracte de sortare.
   În funcția *main( )*, să se instanțieze un obiect din clasa *PlaylistImplementation* și apoi să se folosească datele și metodele aferente.
8. Scrieți o aplicație *C/C++* în care să implementați clasa de bază abstractă *PatrulaterAbstract* având ca atribute protected patru instanțe ale clasei *Punct* (o pereche de coordonate *x* și *y*, accesori și mutatori) reprezentând coordonatele colțurilor patrulaterului. Declarați două metode membre pur virtuale pentru calculul ariei și perimetrului figurii definite. Derivați clasa *PatrulaterConcret* care implementează metodele abstracte moștenite și care conține o metodă proprie care determină dacă patrulaterul este pătrat, dreptunghi, patrulater oarecare (convex/concav). În programul principal instanțiați clasa derivată și apelați metodele implementate. Ariile se vor calcula funcție de tipul patrulaterului. La patrulaterul convex oarecare aria va fi dată de următoarea formulă care exprimă aria funcție de laturile *a, b, c, d*, semiperimetrul *s*, și de diagonalele *p, q*:

   A=sqrt {(s-a)(s-b)(s-c)(s-d)-1/4(ac+bd+pq)(ac+bd-pq)}.

   La patrulaterul concav se va determina doar perimetrul. Bonus, pentru calculul ariei în acest caz.
9. Considerați clasa *Fractie* care are două atribute întregi protected *a* și *b* pentru numărător și numitor, două metode de tip *set( )* respectiv *get( )* pentru atributele clasei. Declarați o *metodă virtuală simplifica( )* care simplifică un obiect *Fractie* folosind *cmmdc-ul* determinat prin operatorul *%*. Definiți un constructor explicit fără parametri care inițializează *a* cu 0 și *b* cu 1, și un constructor explicit cu doi parametri care va putea fi apelat dacă se verifică posibilitatea definirii unei fracții (*b*!=*0*). Supraîncărcați operatorii de adunare, scădere, înmulțire și împărțire (+, -, *, /) a fracțiilor folosind funcții friend care și simplifică dacă e cazul rezultatele obținute, apelând metoda *simplifica( )* din clasă. Definiți o clasă *Fractie_ext* derivată public din *Fractie*, care va avea un constructor cu parametrii (ce apelează constructorul din clasa de bază) și redefinește metoda *simplifica( )* folosind pentru *cmmmdc* algoritmul prin diferență. Afișați un mesaj adecvat în metodă. Definiți de asemenea supraîncărcarea operatorilor compuși de asignare și adunare, scădere, înmulțire și împărțire (+=, -=, *=, /=) cu metode membre. Supraîncărcați operatorii de incrementare și decrementare postfixați care adună/scade valoarea *1* la un obiect de tip *Fractie_ext* cu metode membre.
   Instanțiați două obiecte de tip *Fractie* fără parametrii. Setați atributele obiectelor cu date citite de la tastatură. Afișați atributele inițiale ale obiectelor și noile atribute definite. Efectuați operațiile implementate prin funcțiile *friend* din clasa de bază, inițializând alte 4 obiecte cu rezultatele obținute. Simplificați și afișați rezultatele. Instanțiați două obiecte de tip

Fractie_ext cu date citite de la tastatură. Efectuați operațiile implementate prin metodele clasei, asignând rezultatele obținute la alte 4 obiecte *Fractie_ext*. Folosiți pentru operații copii ale obiectelor inițiale. Simplificați și afișați rezultatele. Verificați posibilitatea utilizării celor două metode de tip *simplifica( )* (din clasa de bază și derivată) folosind instanțe din clasa de bază și derivată folosind un pointer către clasa de bază *Fractie*.

11.7. Individual work

1. Considering the third example (simple inheritance, the base class *Pozitie* and the derived class *Punct)*, resolve the following tasks:
    a. verify the order in which the constructors and destructors are called.
    b. extend the *main()* function in order to use all the methods from the base and derived class.
    c. write a new class called *Circle* (attributes and methods) derived from *Pozitie*
    d. write a program that uses the classes mentioned before.
2. Extend the base class from the fourth example by adding some other virtual methods, which will be implemented in the derived classes (like the *setter* and *getter* for the value of *greutate*).
3. Write a C++ program that defines a class called *Military* that has a public virtual method *im_military()*. Define the classes *Soldier* and *Officer*, both being derived from the first class. Extend further the *Officer* class by implementing the classes *SubLieutenant, Lieutenant, Captain, Major, Colonel, General*. Override the method *im_military( )* for indicating the military degree represented by each class. Instantiate each of the classes *Soldier, Lieutenant, ..., General* and call the *im_military( )* method.
4. Declare a class called *Animal* that contains a *pure virtual method* (*breath( )*) and two *virtual methods* (*eat( )* and *sleep( )*). The classes *Dog* and *Fish* inherit the first class in a *public* mode and implements the pure virtual method. The class *Dog* overrides the *eat()* method. The class *Fish* overrides the *sleep()* method. Instantiate the derived classes and call the specific methods. After that, define an array of *Animal* objects that will contain instances of the derived classes, if that's possible. If not, find an appropriate solution.
5. Define a class *Shape* that considers a shape with a *name* attribute as character string. The class will contain a constructor with parameters and *pure virtual methods* for *area(), perimeter()* and *display().* Derive in public mode a *Circle* class that adds an *int* type attribute *r* to the *radius* and an appropriate constructor considering the attributes (*name, radius*) and defines the methods that calculate the *area(), perimeter()* and *display()* of the circle of radius *r*, value for *r* entered in *main( )* from the keyboard.
   In the same mode define other class *Square* that adds an *int* type attribute *l* as the *side* and methods that calculate the *area(), perimeter()* and *display()* of the square of side *l*, value for *l* entered in *main()* from the keyboard. In *main( )* using a menu with *switch-case* instruction with options for *Circle, Square*, and *Exit,* instantiate objects from the derived classes and display the area and the perimeter with the *display()* method at each option till is selected *Exit*. The data will be introduced from the KB.
6. Define an abstract class that contains 3 pure virtual methods declarations for concatenating, interlacing two one dimensional arrays of characters and for reverting the character array received as parameter. A subclass implements the methods declared in the base class. Instantiate the 2-nd class and display the results produced by applying the methods mentioned above upon some data read from the keyboard. Examine the error given by the attempt of instantiating the base class.
7. Define a class called *Record* that stores the data related to a melody (*artist, title, duration*). An abstract class (*Playlist*) contains as private attribute a pointer to an array of records. The pointer is initialized in the constructor by a memory allocation process (the number of records is defined by the user). The class contains accessor and mutator methods for each of a record's

fields and an abstract method (pure virtual) that sorts the records array according to a criterion coded in the received parameter (*1=sorting by title, 2=sorting by artist, 3=sorting by duration*). The abstract method is implemented inside another class (*PlaylistImplementation*) that inherits the *Playlist* class.
In the *main( )* function, instantiate the *PlaylistImplementation* class and initialize and use all the related data and methods.

8. Write a C++ application that defines the abstract base class *AbstractQuadrilateral* having as protected attributes four instances of the *Point* class (a pair of *x* and *y* coordinates, *getter* and *setter* methods) that represent the quadrilateral's corners. Declare two pure virtual methods for determining the area and the perimeter of the shape. Implement the derived class *ActualQuadrilateral* that implements the inherited abstract methods and has another method for determining whether the quadrilateral is a square, rectangle, or irregular quadrilateral. Instantiate the derived class and call the defined methods. The area will be determined depending on the quadrilateral type. The irregular convex quadrilateral are will be determined considering the following formula that express the area in terms of the sides *a, b, c, d*, the semiperimeter *s*, and the diagonals *p, q:*

   A=sqrt {(s-a)(s-b)(s-c)(s-d)-1/4(ac+bd+pq)(ac+bd-pq)}.

   At the irregular concave quadrilateral only the perimeter will be determined. Bonus for determining the area.
9. Consider the *Fraction* class that has two protected attributes *a* and *b* for the nominator and denominator and two corresponding setter and getter methods. Declare a *virtual method* named *simplify( )* that simplifies a fraction using the greatest common divider determined using the *%* operator. Define an explicit constructor without parameters that initializes *a* with 0 and *b* with 1 and another explicit constructor with two integer parameters. For this constructor is verified if *b!=0* before to be called. Overload the addition, subtraction, multiplication and division operators (+, -, *, /) using *friend* functions and simplify with a virtual method *simplify( )* (if necessary) the obtained results. Define a class named *Fraction_ext* that inherits in a public mode the *Fraction* class and has a parameterized constructor that calls the constructor from the base class. The derived class redefines the implementation of *simplify( )* by determining the greatest common divider using the differences based algorithm. Display an appropriate message in this method. Overload the composed addition, subtraction, multiplication, and division operators (+=, -=, *=, /=) using member methods. Use member methods for overloading the post-increment and post-decrement operators that will add 1 to the value of a *Fraction_ext* instance. Instantiate 2 *Fraction* objects without parameters. Set the attributes using values read from the keyboard. Perform the operations implemented with *friend* functions from the base class and initialize another 4 objects with the obtained results. Simplify the results. Instantiate two objects of *Fraction_ext* type with data from the KB. Perform the implemented operations with the member functions and methods. Assign the operation results to the other 4 existing *Fraction_ext* objects. Use for operations copies of the initial objects. Simplify and display the obtained results. Verify the possibility of using both *simplify( )* methods (base and derived class) using instances of the base and derived classes and a pointer of *Fraction* type.

12. Intrări/ieșiri C++. Supraîncărcarea operatorilor de I/E.

Input/Output in C++. The I/O operators overloading.

12.1. Obiective

- Înțelegerea noțiunilor teoretice legate de sistemul de intrare/ieșire prin aplicarea lor în practică, dezvoltarea de programe C++ cu intrari/ieșiri.

12.2. Objectives

- Understanding the theoretical aspects of the I/O system by applying them into the developed programs.

12.3. Breviar teoretic

Sistemul de Intrare/Ieșire (I/E) din C++ operează prin stream-uri.

Un stream este un dispozitiv logic, care fie produce, fie consumă informație și este cuplat la un dispozitiv fizic prin intermediul sistemului de I/E din C++. La execuția unui program în C++ se deschid în mod automat următoarele patru stream-uri predefinite: `cin`, `cout`, `cerr`, `clog`.

Sistemul de I/E din C++ ne permite să **formatăm** datele. Fiecare stream din C++ are asociat un număr de "indicatori de format" (flag-uri), care determină modul de afișare. În clasa `ios_base` este definit tipul `fmtflags`, pentru flagurile de formatare ale fluxului. În această clasă există mai multe constante publice de acest tip, care pot fi folosite pentru formatarea fluxurilor și care sunt enumerate în continuare (detalii curs):

`skipws` – ignorarea caracterelor de tip whitespace

`left` – aliniere la stanga

`right` - aliniare la dreapta

`internal` - o valoare numerică se extinde pentru completarea câmpului

`dec` - afisare în zecimal

`oct` – afisarea în octal

`hex` – afisare în hexazecimal

`showbase` – afișarea simbolului bazei de numerație în care se face afișarea

`boolalpha` – citește/scrie elementele booleene ca stringuri (true și false).

`showpoint` - afișarea tuturor zerourilor și a punctului zecimal pentru o valoare în virgulă mobilă

`showpos` - determină afișarea semnului înaintea valorilor numerice pozitive

`scientific` - afișarea numerelor în virgulă mobilă în format științific (cu exponent)

`fixed` – afișarea în forma obișnuită a numerelor în virgulă mobilă parte întreagă/ fracționară

`unitbuf` - streamul este eliberat după fiecare operație de ieșire

`uppercase` - afișarea cu majusculă a caracterelor generate de stream

Formatarea datelor poate fi realizată prin **intermediul flag-urilor, manipulatorilor standard și a manipulatorilor utilizator.**

Stabilirea formatului prin intermediul flag-urilor se poate realiza prin utilizarea metodei:

```
long setf (long f );
```

unde flag-urile sunt codificate în parametrul de tip long al metodei.

```
fmtflags setf (fmtflags  mask);
fmtflags setf (fmtflags  mask, fmtflags unset);
```

Resetarea flagurilor se realizează cu metoda:

```
long unsetf (long flags);
void unsetf (fmtflags flags);
```

Starea curentă a indicatorilor de format poate fi obținută cu metoda `flags()`:

```
fmtflags flags();
fmtflags flags(fmtflags f);
```

Resetarea la starea implicită, default, a flag-urilor se face apelând:

```
flags(0) ;
```

Există trei metode membre ale clasei ios, care **stabilesc lățimea câmpului, precizia și caracterul de înserat**. Acestea sunt `width(), precision()` și `fill()`. Prototipul metodei `width()` este următorul:

```
int width(int w);
```

unde w – este lățimea câmpului, iar valoarea returnată este valoarea anterioară a câmpului.

La afișarea unei valori în virgulă mobilă, se folosesc în mod implicit (default) **6 cifre pentru întregul număr** (parte întreagă și fracționară). Totuși **doar partea fracționară** o putem fixa dacă alegem o reprezentare specifică pentru valori reale (`fixed` sau `scientific`) și apelăm metoda `precision()`, altfel se va referi la precizia specifică părții întregi și fracționare (numărul de digiti):

```
int precision(int p);
```

unde p – stabilește precizia , metoda returnează vechea valoare.

Când un câmp trebuie completat, se folosește în mod implicit caracterul “spațiu”. Se poate specifica un caracter alternativ cu ajutorul metodei `fill()`, cu prototipul:

```
char fill(char ch);
```

unde ch – noul caracter cu care se completează câmpul, metoda returnează vechiul caracter.

Metodele de tip “manipulator” sau “manipulatorii standard” permit formatarea operațiilor de I/E. În limbajul C++ sunt :

```
dec, endl, ends, flush, hex, oct, resetiosflags(long f), setbase (int
base), setfill(int ch), setioflags(long f),
setprecision(int p), ws, setw(int w).
```

Este necesară includerea fișierului header `<iomanip>` pentru utilizarea manipulatorilor cu parametri.

Manipulatorii utilizator sunt de două tipuri:

- cei care operează cu stream-uri de intrare;
- cei care sunt asociați stream-urilor de ieșire.

Forma generală a manipulatorilor utilizator de ieșire este următoarea:

```
ostream& nume-manipulator (ostream& stream) {
    // cod
    return stream;
}
```

Sintaxa metodelor manipulator utilizator de intrare este următoarea:

```
istream& nume manipulator (istream& stream) {
    // cod
    return stream;
}
```

Noile versiuni C++ 1y/2z au introdus **noi elemente legate de formatarea datelor** cu manipulatori:

`https://www.tutorialspoint.com/cpp_standard_library/iomanip.htm`

`https://en.cppreference.com/w/cpp/io/manip`

printre care amintim:

- `hexfloat, defaultfloat (C++11)`, pentru numerele reale,
- `emit_on_flush, no_emit_on_flush (C++20)`, controlează dacă stream-ul `baza_syncbuf` al unui flux se emite în flux
- `get_money (C++11)`, parsare valoare monetară
- `put_money (C++11)`, formatează și produce o valoare monetară
- `get_time (C++11)`, parsare dată/timp la format specificat
- `put_time (C++11)`, formatează și scoate o valoare de tip dată/timp în funcție de formatul specificat
- `quoted (C++14)`, inserează și extrage șiruri între ghilimele cu spații încorporate

Supraîncărcarea operatorilor de intrare-ieșire

Operatorii de intrare/ieșire pot fi supraîncărcați într-o anumită clasă **prin funcții friend și/sau funcții globale**. Supraîncărcarea operatorului de ieșire se realizează printr-o **funcție inserter**, care are următoarea formă generală:

```
ostream& operator <<(ostream& stream, Nume_clasa ob)
{
    // corp inserter
    return stream;
}
```

Supraîncărcarea operatorului de intrare se realizează printr-o **funcție extractor**, care are următoarea formă generală:

```
istream& operator >>(istream& stream, Nume_clasa& ob)
{
    // corp extractor
    return stream;
}
```

12.4. Theoretical brief

The Input/Output (I/O) system in C++ operates via streams.

A stream is a logical device that either produces or consumes information and is connected to a physical device via the I/O system in C++. When executing a program in C++, the following four predefined streams are automatically opened:: cin, cout, cerr, clog.

The I/O system in C++ allows data formatting. Each stream in C++ has a number of "format indicators" (flags) associated with it, that controls the displaying mode. In `ios_base` class (or `ios`) the `fmtflags` type is defined, for setting the stream formatting flags. In this class there are several other public constants of this type, used at streams formatting, listed below (more details in the lecture material):

`skipws` – ignoring the whitespace characters

`left` - left align

`right` - right align

`internal` - a numeric value is expanded to fill the available field

`dec` - decimal representation

`oct` - octal representation

`hex` - hexadecimal representation

`showbase` - the numeration base symbol is shown

`boolalpha` - reads/writes the boolean elements as strings (true and false).

`showpoint` - all the zeroes and the point are displayed for float values

`showpos` - the sign is displayed in front of the positive numeric values

`scientific` - the float values are represented using exponents

`fixed` - the float values are displayed in the regular shape integer part / decimal part

`unitbuf` - the stream is cleaned after each I/O operation

`uppercase` - the characters in the stream are represented in uppercase

Data formatting can be achieved by means of flags, standard manipulators and user defined manipulators.

Establishing the format using the flags can be done using the method:

```
long setf (long f);
```

where flags are encoded in the method's long type parameter.

```
fmtflags setf (fmtflags  mask);
fmtflags setf (fmtflags  mask, fmtflags unset);
```

Resetting the flags can be done by using the mehtods:

```
long unsetf (long flags);
void unsetf (fmtflags flags);
```

The current state of format indicators can be obtained with the method `flags()`:

```
fmtflags flags();
fmtflags flags(fmtflags f);
```

Resetting the flags to the default value is done by calling:

```
flags(0);
```

There are three member methods of the ios class, which determine the width of the field, precision and filling character: `width()`, `precision()` and `fill()`.

The `width()` method has the following prototype:

```
int width(int w);
```

where w represents the field width, and returns the previous value.

When displaying a real (floating point) value, 6 digits are used by default for the entire number (integer and fractional part). However, only the fractional part can be fixed if we choose a specific real type (`fixed` or `scientific`) and call the `precision()` (method), otherwise it will refer to the precision specific to the integer and fractional part.

The `precision()` method allows setting the precision length:

```
int precision(int p);
```

When a field needs filling, the whitespace character is used by default. Changing the filling character can be done by calling the `fill()` method:

```
char fill(char ch);
```

where ch is the new filling character.

The "manipulator" methods or "standard manipulators" allow formatting the I/O operations. In the C++ language, the manipulators are :

```
dec, endl, ends, flush, hex, oct, resetiosflags(long f),
setbase (int base), setfill (int ch), setioflags(long f),
setprecision(int p), ws, setw(int w).
```

The **<iomanip>** header file needs to be used for manipulators with parameters.

The user defined manipulators are divided into:

- manipulators that configure the input streams;
- manipulators that configure the output streams.

The generic form of the output manipulator is:

```
ostream& nume-manipulator (ostream& stream){
    // code
    return stream;
}
```

The generic form of the input manipulator is:

```
istream& nume manipulator (istream& stream){
    // code
    return stream;
}
```

The new C++ 1y/2z versions introduced new elements related to formatting data using manipulators:

```
https://www.tutorialspoint.com/cpp_standard_library/iomanip.htm
https://en.cppreference.com/w/cpp/io/manip
```

among which there are:

- `hexfloat, defaultfloat (C++11)`, for real numbers,
- `emit_on_flush, no_emit_on_flush (C++20)`, control if the `base_syncbuf` stream is emitted in the associated stream
- `get_money (C++11)`, money value parsing
- `put_money (C++11)`, formats a money value
- `get_time (C++11)`, parses the date/time according to a specified format
- `put_time (C++11)`, formats and outputs a date/time value
- `quoted (C++14)`, inserts / extracts arrays of characters delimited by quotes, with embedded witespaces

Overloading the I/O operators

The Input/Output operators can be overloaded in a given class using friend functions and/or global functions. Overloading the output operator is done by an inserter function, which has the following general form:

```
ostream& operator <<(ostream& stream, Class_name ob){
    // inserter code
    return stream;
}
```

Overloading the input operator is done by an extractor function, which has the following general form:

```
istream& operator >>(istream& stream, Class_name& ob){
    // extractor code
    return stream;
}
```

12.5. Exemple/ Examples

Ex. 1 - Formatarea datelor cu flag-uri / Data formatting using flags

```
// setf(),  unsetf()
#include <iostream>
using namespace std;
const int dim =30;

int main( ){
    // implicit displaying
    cout << 123.33 << " Hello! " << 100 << '\n';
    cout << 10 << ' ' << -10 << '\n';
    cout << 100.01 << '\n';
    cout << 100.0 << '\n';

    // changing the format
    cout.unsetf(ios::dec);
    cout.setf( ios_base::hex);
    cout << 123 << " Hello! " << 100 << '\n';
    cout.setf (ios::showpos|ios::showbase);
    cout << 10 << ' ' << -10 << '\n';
    cout.setf(ios::scientific);
    cout << 100.1 << '\n';
```

```
    cout.unsetf(ios::scientific);
    cout.setf(ios::dec|ios::showpoint);
    cout << 100.0 << '\n';

    //alignment
    cout.width(dim);
    cout.fill('*');
    cout.setf(ios::right);//default
    cout << "Right align" << '\n';

    cout.unsetf(ios::right);
    cout.width(dim);
    cout.fill('$');
    cout.setf(ios::left);
    cout << "Left align" << '\n';
    return 0;
}//main
```

Ex. 2 - Manipulatori standard / Standard manipulators

```
//standard I/O manipulators

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

int main( ){
    cout << hex << 100 << endl;
    cout << oct << 100 << endl;
    cout << setfill( 'Y') << setw (8);
    cout << 10 << endl;
    cout << setprecision(8);//integer and fractional part
    cout << 100.120 << endl;
    cout << setprecision(4);
    cout << -100.120 << endl;

    cout.setf(ios::fixed);
    cout << setprecision(8);//only for fractional part
    cout << 100.120 << endl;
    cout << setprecision(4);
    cout << -100.120 << endl;
    return 0 ;
} //main
```

Ex. 3 - Manipulatori definiți de utilizator / User-defined manipulators

```
//user defined manipulators

#include<iostream>
using namespace std;

ostream& init(ostream& stream);//user defined manipulator

int main( ){
    //cout.setf(ios::uppercase);
    cout << "Value to be displayed: ";
    cout << init << 111.123456;
```

```
    return 0;
}//main

ostream& init(ostream& stream){
    stream.width(20);
    stream.precision(5);
    stream.fill('$');
    stream.setf(ios::showpos | ios::scientific | ios::uppercase);
    return stream;
}//init
```

Ex. 4 - Supraîncărcarea operatorilor de I/O / Overloading the I/O operators

```
//overloading the I/O operators using friend functions
//MyClass.h
const int dim = 10;

class MyClass {
    int x;
    char text[dim];
public:
    MyClass( ) {
        x = 0;
        strcpy(text, "no text");
    }
    MyClass(int x, const char* text) {
        this->x = x;
        strcpy(this->text, text);
    }
    friend istream& operator>>(istream& stream, MyClass& obj);
    friend ostream& operator<<(ostream& stream, MyClass obj);
};//MyClass

//extractor overloading
istream& operator>>(istream& stream, MyClass& obj) {
    cout << "\nInteger value: ";
    stream >> obj.x;
    cout << "\nText value: ";
    stream >> obj.text;
    return stream;
}

//inserter overloading
ostream& operator<<(ostream& stream, MyClass obj) {
    stream << "\nThe attributes are: ";
    stream << obj.x << ", ";
    stream << obj.text;
    return stream;
}

//main
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#include "MyClass.h"
```

```
int main( ) {
    MyClass test_obj1(7, "aaa");
    cout << test_obj1;
    MyClass test_obj2;
    cout << "\nEmpty constructor object: "<< test_obj2;
    cin >> test_obj2;
    cout << "\n Object modified after reading: " << test_obj2;
    return 0;
}//end_main
```

Ex. 5 - Supraîncărcarea operatorilor de I/O / Overloading the I/O operators

a)//overloading the I/O operators using **friend functions**

```
//Coord.h

class Coord {
    int x, y;
public:
    Coord ( ) { x=0; y=0; }
    Coord (int i, int j) { x=i; y=j; }
    friend ostream& operator<<(ostream&  , Coord ob);
    friend istream& operator>>(istream& , Coord &ob);
};// Coord class

// inserter
ostream& operator<< (ostream& stream, Coord ob) {
    stream << ob.x <<" "<< ob.y << '\n';
    return stream;
}

// extractor
istream& operator>> (istream& stream, Coord & ob){
    cout<< "Enter the coordinates: ";
    stream>>ob.x >>ob.y;
    return stream;
}

//main
#include <iostream>
using namespace std;
#include "Coord.h"

int main (){
    Coord A(2,2), B(10,20);
    cout<< A << B;
    cin >> A;
    cout << A;
}//main
```

b)//overloading the I/O operators using **global functions**

```
//Coord.h

class Coord {
    int x, y;
public:
```

```
    Coord ( ) { x = 0; y = 0; }
    Coord (int i, int j) { x = i; y = j; }
    int getX( ) {
        return x;    }
    int getY( ) {
        return y;    }
    void setX(int a) {
        x = a;    }
    void setY(int b) {
        y = b;    }
};// Coord class

// global inserter
ostream& operator<< (ostream& stream, Coord ob) {
    stream << ob.getX() << " " << ob.getY() << '\n';
    return stream;
}

// global extractor
istream& operator>> (istream& stream, Coord& ob) {
    int a, b;
    cout << "Introduceti coordonatele: ";
    stream >> a >> b;
    ob.setX(a);
    ob.setY(b);
    return stream;
}

//main
#include <iostream>
using namespace std;

#include "Coord.h"

int main( ) {
    Coord A(2, 2), B(10, 20);
    cout << A << B;
    cin >> A;
    cout << A;
    cin >> B;
    cout << B;
}//main
```

12.6. Lucru individual

1. Scrieți un program C++ în care afișați diferite valori în zecimal, octal și hexazecimal. Afișați valorile aliniate la dreapta, respectiv la stânga într-un câmp de afișare cu dimensiunea 15. Utilizați manipulatorul `setfill()` pentru stabilirea caracterului de umplere și metodele `width()` și `precision()` pentru stabilirea dimensiunii câmpului de afișare și a preciziei.
2. Scrieți o aplicație C++ în care se citesc de la tastatură date de diferite tipuri, urmând a fi afișate pe ecran utilizând manipulatorii standard.
3. Considerați achiziția de date cu valori reale de la un dispozitiv electronic (10 date). Afișați folosind un mesaj adecvat datele primite considerând un format minimal (partea întreagă).

Determinați media acestor valori (partea întreagă), iar dacă depășește un prag stabilit anterior (definit sau citit), afișați aceste date în format detaliat considerând că avem date de tip real, cu o precizie de 3 digiți la partea fracționară.

4. Definiți o clasă numită `MiscareAccelerata`, care conține atributele private `dc` (distanța curentă, `vc` (viteza curentă) și `a` (accelerația), atributele `dc`, `vc` și `a` sunt initializate în constructor iar valoarea lor este cea dată de `d0` și `v0`, și `a0` ca și parametri. În clasă sunt supraîncărcați operatorii de extracție și de inserție pentru a se putea inițializa și afișa caracteristicile unei instanțe. Implementați metoda `determinaMiscarea()` care re-calculează atributele `dc` și `vc`, pe baza unui număr de secunde primit ca și parametru și având în vedere legea mișcării rectilinii uniform accelerate cu accelerație `a0`. Instanțiați clasa și apoi folosiți membrii definiți.
5. Supraîncărcați operatorii de extracție și de inserție pentru clasa `Complex`, în care părțile reale și imaginare sunt ambele *protected* de tip real. Derivați public o clasă `Punct` din clasa `Complex`, adăugând atributul culoare pentru punctul de coordonate x și y corespunzător părții reale, respectiv imaginare a numărului complex. Supraîncărcați din nou aceiași operatori de intrare-ieșire. Instanțiați obiecte de tip `Complex` și `Punct` și verificați funcționalitatea supraîncărcării operatorilor. Asignați un obiect de tip `Punct` la unul de tip `Complex` prin upcasting și afișați atributele lui.
6. Considerați clasa `Fractie` care are două atribute întregi private `a` și `b` pentru numărător și numitor, două metode de tip `set()` respectiv `get()` pentru atributele clasei. Declarați o metodă `simplifica()` care simplifică un obiect `Fractie` returnând o valoare reală. Considerați un atribut privat static întreg `icount`, care va fi inițializată cu 0 și incrementat în cadrul constructorilor din clasă. Definiți un constructor explicit fară parametri care inițializează `a` cu 0 și `b` cu 1, și un constructor explicit cu doi parametri care va putea fi apelat dacă se verifică posibilitatea definirii unei fracții (b!=0). Definiți un destructor explicit care afișează un mesaj. Supraîncărcați operatorii de adunare, scădere, înmulțire și împărțire (+,-,*,/) a fracțiilor folosind funcții *friend* fără a simplifica rezultatele obținute. Instanțiați două obiecte de tip `Fractie` cu date citite de la tastatură. Afișați atributele inițiale ale obiectelor pe linii diferite iar fiecare membru al fracției vă fi afișat pe o lățime de 10 digiți, caracter de umplere *, primul număr aliniat la stânga iar al doilea aliniat la dreapta. Printr-o metodă accesor, afișați contorul `icount` ca și un întreg cu semn, pe 15 poziții, caracter de umplere $, aliniat la stânga. Efectuați operațiile implementate prin funcțiile *friend*, inițializând alte 4 obiecte cu rezultatele obținute. Afișați rezultatele (numărător/numitor) folosind supraîncărcarea operatorul de ieșire (<<, inserție) și contorul (ca și un întreg cu semn, pe 20 de poziții, caracter de umplere #, aliniat la dreapta) după ultima operație folosind o metodă accesor adecvată. Simplificați rezultatele obținute pe care le veți afișa ca numere reale de tip *fixed* cu o precizie de 4 digiți la partea fracționară.

12.7. Individual work

1. Write a C++ program that displays some numerical values in decimal, octal and hexadecimal. Display the values left and right aligned, inside a field that can hold 15 characters. Use the `setfill()` manipulator for setting the filling character and the `width()` and `precision()` methods for setting the displaying field size and the values representation precision.
2. Write a C++ application that reads from the keyboard a series of values of various types. Display the values using the standard manipulators.
3. Consider a data acquisition process from a hardware device (10 variables real type). Display, using an appropriate message, the data in a minimal format (the integer part). Determine the average value of the displayed numbers (the integer part) and if it is greater than a

previously defined (or entered) threshold, display the data in a detailed format (`float` variables, 3 digits precision at the fractional part).

4. Define a class called `AcceleratedMovement` that contains the private attributes `dc` (the current distance), `vc` (the current speed) and `a` (the acceleration). The values of `dc, vc` and `a` are initialized in the constructor and their values are equal to `d0, v0` and `a0` (as parameters).The class overloads the extraction and insertion operators for initializing and displaying the characteristics of a certain instance.
   Implement the method `determineMovement` that re-calculates the values of `dc` and `vc`, considering a number of seconds (received as paramenter) and the law of uniformly accelerated linear motion with `a0` acceleration. Instantiate the class and use the defined members.
5. Overload the extraction and insertion operators for the `Complex` class (both the imaginary and real parts are represented as `protected` real values). Create another class named `Point` that inherits the first class and introduces the color attribute for a point that has as coordinates the real and imaginary parts of the complex number. Overload again the extraction and insertion operators. Create some instances of the defined classes and verify the functionality of the overloaded operators. Assign an object of `Point` type to an object of `Complex` type and dispaly the attributes of the obtained object.
6. Consider the `Fraction` class that has two private attributes `a` and `b` for the nominator and denominator and two corresponding setter and getter methods. Declare a method named `simplify()` that simplifies a fraction and returns a real value. Define an explicit constructor without parameters that initializes `a` with 0 and `b` with 1 and another explicit constructor that receives two parameters representing the values of the nominator and denominator. For this constructor is verified if `b!=0` before to be called. Define a destructor that displays a message. Consider a static variable `icount` that will be initialized with 0 and incremented in the class's constructors. Overload the addition, subtraction, multiplication and division operators (+, -, *, /) using friend functions, without simplifying the obtained results. Instantiate two `Fraction` objects and read the corresponding data from the keyboard. Display the initial attributes of the objects, on different lines, in 10 digits placeholders using * as filling character. The denominator will be left aligned while the nominator will pe positioned in the right part of the displaying field. Using an accesor method display the value of `icount` as a signed integer, on 15 characters, left aligned, using *'$'* as filling character. Perform the operations implemented with *friend* functions initializing another four objects with the obtained results. Display the data (denominator/nominator) by overloading the output (<<, insertion) operator and the counter (as a signed integer, on 20 characters, right aligned, using '#' as filling character) after the last operation. Simplify the results and display the resulting values as *fixed* float numbers with 4 digits precision.

13. Fișiere în C++

Files in C++

13.1. Obiective

- Utilizarea claselor, metodelor și operatorilor de intrare/ieșire pentru intrările și ieșirile standard și pentru lucrul cu fișiere.

13.2. Objectives

- Using classes, methods and I/O operators for standard input and output and for working with files.

13.3. Breviar teoretic

Pentru a realiza **operații de I/E cu fișiere**, trebuie să includem în program fișierul de tip antet `fstream`, care definește mai multe clase, incluzând: `ifstream, ofstream și fstream`. Aceste clase sunt derivate din clasele `istream și ostream`. În C++, un fișier este deschis prin legarea lui la un `stream`. Înainte de deschiderea fișierului trebuie să obținem mai întâi un stream. Pentru a crea un stream de intrare, el trebuie declarat ca fiind de tip `ifstream`, iar pentru a crea unul de ieșire, streamul trebuie declarat de tip `ofstream`. Streamurile care realizează ambele tipuri de operații trebuie declarate de tip `fstream`.

Metoda `open()` este folosită pentru a asocia un fișier unui stream. Această metodă este membră a tuturor celor trei clase de tip stream și are prototipul:

```
void open(char *nume_fișier, int mod, int acces);
```

unde primul parametru reprezintă un pointer la numele fișierului, al doilea este o valoare care specifică modul de deschidere a fișierului, al treilea e implicit setat pe mecanismul de acces din SO DOS.

Se poate utiliza un **mecanism implicit simplificat** de deschidere a fisierelor de forma:

```
ifstream fin ("test.dat");  // in file
ofstream fout ("test.dat");  // out file
fstream finout ("test.dat");  // in-out file
```

Pentru **închiderea** unui fișier se folosește metoda membră `close()`. Metoda `close()` nu are parametri și nu returnează nici o valoare. Pentru a închide un fișier cuplat la un stream denumit stream_propriu, se folosește instrucțiunea:

```
stream_propriu.close( );
```

Limbajul C++ asigură numeroase metode de I/E de **tip binar**; `get()` și `put()` sunt metode binare de I/E de nivel fizic. Cu ajutorul metodei membre `put()` scriem un octet, iar cu metoda `get()` citim un octet. Prototipul metodei `get()` este următorul:

```
istream& get(char& car);
```

Metoda citește din stream-ul asociat un singur caracter și plasează valoarea sa în variabila `car`, returnează o referință a stream-ului de intrare.

Metoda `put()` are prototipul:

```
ostream& put(char car);
```

Metoda scrie caracterul `car` în stream și returnează o referință a stream-ului de ieșire. Există și alte variante ale metodelor.

În cadrul operațiilor de I/E se mai folosește metoda `ignore()` care extrage și sare peste cel mult *n* caractere dar se oprește la întâlnirea delimitatorului care e extras din stream (al doilea parametru). Formatul general este:

```
istream& ignore (int n = 1, int delim = EOF);
```

Metoda o găsim deseori legată de metoda `get()` astfel:

```
cin.ignore( );
cin.get( );
```

E de remarcat că funcția are toți parametrii impliciți și poate fi apelată fără parametri.

Pentru a citi/scrie **blocuri de date binare**, se utilizează metodele `read() și write()`, care au următoarele prototipuri:

```
istream& read(unsigned char *buf, int numar);
ostream& write(const unsigned char *buf, int numar);
```

Metoda `read()` citește `numar` de octeți din stream-ul asociat și îi plasează în bufferul pe care îl indică pointerul `buf`.

Metoda `write()` scrie din zona tampon, pe care o indică `buf`, `numar` octeți în stream-ul asociat.

În sistemul de I/E din C++, putem avea **acces de tip *aleator*** la fișiere, prin utilizarea metodelor `seekg()` și `seekp()`. Formele cele mai uzuale pentru aceste metode sunt:

```
istream& seekg (streamoff offset, seek_dir origine) ;
ostream& seekp (streamoff offset, seek_dir origine) ;
```

Tipul `streamoff` este definit în fișierul antet `iostream` și el poate conține valoarea maximă a offset-ului, iar `seek_dir` este un tip de enumerare care are trei valori:

```
ios::beg, ios::cur, ios::end.
```

Aflarea poziției curente de citire/scriere se poate face cu metodele `tellg()` sau `tellp()`, care au următoarele sintaxe :

```
long tellg();
long tellp();
```

Sistemul de I/E al limbajului C++ conține informații de **stare cu privire la operațiile de I/E**. Starea curentă a sistemului de I/E este conținută într-o variabilă de tip întreg, în care se codifică următorii indicatori: `goodbit, eofbit, failbit, badbit`. Acești indicatori sunt obținuți prin apelul metodei `std::ios::rdstate()`.

O a doua metodă de obținere a stării sistemului de I/E este prin utilizarea metodelor specializate**:** `int bad(), int eof(), int fail(), int good()`.

Metodele `bad()` și `fail()` returnează o valoare de tip `bool`: `true` dacă `badbit` respectiv `failbit` sunt activate, și metoda `good()` returnează `true` dacă nici unul din flagurile de stare (`eofbit, failbit și badbit`) nu este setat.

Metoda `eof()` returnează o valoare diferită de zero dacă se atinge sfârșitul fișierului (marcat de *EOF=-1* în versiunile anterioare legate de fișiere). Flag-ul `eofbit` va fi verificat **doar după accesul la fișier** și când nu mai sunt date, deci este EOF!!!

Metoda `getline()` permite **citirea unui șir de caractere** (inclusiv spații) până la întâlnirea delimitatorului sau până la citirea a *nr_max*-1 caractere. Este adecvată pentru citirea liniilor de fișiere text. Sintaxa ei este:

```
istream& getline( char* sir,int nr_max,  char delim = '\n' );
```

Metoda `gcount()` numără câte caractere s-au introdus de la ultima citire. Se apelează după utilizarea metodelor `get(), getline()` și `read()`. Prototipul metodei este:

```
int gcount();
```

În cazul metodei `peek()` aceasta returnează următorul caracter fără a-l extrage din stream.

```
int peek();
```

Metoda `putback()` pune înapoi în flux un caracter:

```
istream& putback(char c);// put back character c in the stream
```

13.4. Theoretical brief

In order to perform **I/O operations with files**, the header file `fstream` must be included. It defines several classes, including: `ifstream, ofstream` and `fstream`. These classes are derived from `istream` and `ostream`. In C++, a file is opened by connecting it to a `stream`. Before opening the file a stream must be obtained. A reading stream will have the `ifstream` type, a writing stream will have `ofstream` type. The streams able to perform both I/O operations are of `fstream` type.

The `open()` method is used for associating a file to a stream. It is common to all 3 stream classes and has the following prototype:

```
void open(char *nume_fișier, int mod, int acces);
```

where the first parameter represents a pointer to the filename, the second is a value specifying how the file will be opened, the third is default access mechanism in DOS OS.

A **simplified default mechanism** for opening form files can be used:

```
ifstream fin ("test.dat");  // in file
ofstream fout ("test.dat");  // out file
fstream finout ("test.dat");  // in-out file
```

The `close()` method is used for **closing a file**. It has no parameters and returns no returned value. For closing a file connected to a named stream, the following instruction is used:

```
stream_name.close();
```

The C++ language provides various methods that work with **binary data**. `get()` and `put()` are I/O methods that work on physical level. The `put()` method writes a byte, the `get()` method reads a byte.

The `get()` method's prototype:

```
istream& get(char& car);
```

The method reads from the associated stream of a single character and places its value in the variable *car*. It returns a reference of the input stream.

The `put()` method's prototype:

```
ostream& put(char car);
```

It writes the `car` character in the stream and returns a reference to the output stream.

There are other variants of these methods.

The method `ignore()` extracts and jumps over a maximum of *n* characters but stops if a certain delimiter is met (the second parameter).

The generic format it:

```
istream& ignore (int n = 1, int delim = EOF);
```

It is usually used in conjuction with `get()` as:

```
cin.ignore();
cin.get();
```

It is noteworthy that the function has all parameters with default values, and can be called without any parameters.

For reading/writing **blocks of binary data**, the methods `read()/write()` are used, with the following prototypes:

```
istream&  read(unsigned char *buf, int numar);
ostream& write(const unsigned char *buf, int numar);
```

`read()` reads `numar` bytes from the associated stream and places them in the buffer identified by `buf` pointer.

`write()` writes from the data stored in `buf`, `numar` bytes in the associated stream.

C++ provides **random file access mechanisms**, by using the `seekg()` and `seekp()` methods. The most common forms are:

```
istream& seekg (streamoff offset, seek_dir origine) ;
ostream& seekp (streamoff offset, seek_dir origine) ;
```

The `streamoff` type is defined in the `iostream` header file and can contain the maximum value of the offset, `seek_dir` is an enumeration containing the values:

```
ios::beg, ios::cur, ios::end.
```

Determining the current reading/writing position depends on the `tellg()` or `tellp()` methods:

```
long tellg();
long tellp();
```

The C++ language contains **status I/O data**. The current state of the I/O system is memorized in an integer variable, coded in the following indicators: `goodbit, eofbit, failbit, badbit`. They are obtained by calling the method `std::ios::rdstate()`.

Another variant of obtaining the I/O system state refers to using the **specialized methods:**

```
int bad(), int eof(), int fail(), int good().
```

`bad()` and `fail()` return a `bool`: `true` if `badbit` respectively `failbit` are activated, `good()` returns `true` if none of the status indicators (`eofbit, failbit și badbit`) is activated.

The `eof()` method returns a non-zero value if the end of file is encountered (marked by *EOF=-1* in previous versions). The flag `eofbit` will be verified only after the file is accessed and there are no more available informations, so EOF!!!

`getline()` allows reading **an array of character** (spaces included) until the new line delimiter is encountered or *nr_max*-1 characters have been read. It is used for reading the text files, line by line. Syntax:

```
istream& getline( char* sir,int nr_max,  char delim = '\n' );
```

`gcount()` counts how many characters have been introduced from the previous reading. It is called after using `get()`, `getline()` and `read()`.

The prototype:

```
int gcount( );
```

The `peek()` method returns the next character without extracting it from the stream.

```
int peek();
```

`putback()` puts a character back in the stream:

```
istream& putback(char c);
```

13.5. Exemple/ Examples

Ex. 1 - Scriere de caractere în fișier text cu put() / Using put() to write chars in a text file

```
//  using put( ) for writing characters in a text file
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char*argv[ ]){
char car;
    if (argc!=2)  {
        cout <<"Filename not specified! \n" ;
        exit(1);
    }

    ofstream out;
    out.open(argv[1]);
    if (!out) {
        cout  <<"File cannot be opened";
        exit(1);
    }

    cout << "Write characters. $ stops the process\n";
    do {
        cin.get(car);
        out.put(car);
    } while (car! ='$' );
    out.close( );
    cout << "\nEnd program";
    return 0;
}//main
```

Ex. 2 - Citire din fișier text cu read() / Reading a text file using read()

```
// reading a file using read( )

#include <iostream>
#include <fstream>
using namespace std;
const int dim = 11;
```

```
int main( ){
    char sir[dim] = " ";
    ifstream in;
    in.open("Text.txt");//minimum 10 characters in the file
    if (!in) {
        cout << "File cannot be opened\n";
        exit(1);
    }

    while (in) { //while (1) {
        in.read(sir, dim - 1);//reads sequences of 10 characters
        if (in.eof())break;
        cout << sir;
    }
    in.close();
    cout << "\nEnd program";
    return 0;
}//main
```

Ex. 3 - Scriere de date într-un fișier text / Writing data to a text file

```
//writing a string, double and int, space delimited; reading until EOF
#include <iostream>
#include <fstream>
using namespace std;
const int dim =20;

int main( ){
    ofstream out("test.txt"); // output, normal file

    if (!out) {
        cout << "Cannot open test.txt file.\n";
        exit(1);
    }

    out << "R " << 9.9 << " " << 10 << endl;// implicit overloading
    out << "T " << 9.9 << " " << 9 << endl;
    out << "M " << 4.8 << " " << 4 << endl;
    out<<std::ifstream::traits_type::eof();// write EOF
    out.close( );

    ifstream in("test.txt"); // input
    if (!in) {
        cout << "Cannot open test.txt file.\n";
        exit(1);
    }
    char item[dim];
    double cost;
    int mark;
/* not so used variant
    int eof_m;
    while (in) { //while (1) {
        in >> item >> cost >> mark;
        if (eof_m = in.peek( ), eof_m == -1)break;
        cout << item << " " << cost << " " << mark << "\n";
    };*/
```

```
    while (in) { //while (1) {
        in >> item >> cost >> mark;// implicit overloading
        if (in.eof( )) break;
        cout << item << " " << cost << " " << mark << "\n";
    }
    in.close( );
    cout << "\nEnd program";
    return 0;
}//main
```

Ex. 4 - Acces aleator la datele din fișier / Random access to data in a file

```
// random access files
#include<iostream>
#include<fstream>
using namespace std;

int main(int argc, char *argv[ ]){
    if(argc!=4){
        cout<<"Usage: <exe filename> <file_to_read> <position> <character>";
        exit(1);
    }
    fstream out;
    out.open(argv[1],ios::in|ios::out|ios::binary);
    if(!out){
        cout<<"File cannot be opened";
        exit(1);
    }
    out.seekp(atoi(argv[2]),ios::beg);
    out.put(*argv[3]);
    out.close( );
    cout << "\nEnd program";
    return 0;
}//main
```

Ex. 5 - I/O cu operatori de inserție și extracție / I/O using inserter and extractor operators

```
// a)extractor and inserter for keyboard reading and file writing

//Coord.h

class Coord {
    int x, y;
public:
    Coord( ) { x=0; y=0; }
    Coord(int i, int j) { x=i; y=j; }

    friend ostream& operator<< (ostream &, Coord ob);
    friend istream& operator>> (istream &, Coord& ob);
};//class

// inserter
ostream& operator<< (ostream &stream, Coord ob){
    stream<<"Coordinates:";
    stream << ob.x <<", "<< ob.y << '\n';
```

```
    return stream;
}

// extractor
istream& operator>> (istream &stream, Coord& ob){
    cout<< "Enter coordinates: ";
    stream>> ob.x >> ob.y;
    return stream;
}

// main program
#include <iostream>
using namespace std;
#include "Coord.h"

int main ( ){
    Coord A(2,2), B(10,20);
    cout<< A << B;
    cin >> A;
    cout << A;
/*
    ofstream fout;//error if working with files from the console
    fout.open("test.txt", ios::out);
    fout << A;
    fout.close( );
*/
    cout << "\nEnd program";
    return 0;
}//main

//b) different overloading for console / file

//Coord.h

class Coord {
    int x, y;

public:
    Coord( ) { x = 0; y = 0; }
    Coord(int i, int j) { x = i; y = j; }

    friend ostream& operator<< (ostream &, Coord ob);
    friend istream& operator>> (istream &, Coord& ob);
    friend ofstream& operator<< (ofstream &, Coord ob);
    friend ifstream& operator>> (ifstream &, Coord& ob);
};//class

// inserter for Coord
ostream& operator<< (ostream &stream, Coord ob){
    stream << "Coordinates are:";
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}

// extractor for Coord
istream& operator>> (istream &stream, Coord &ob){
```

```
    cout << "Enter the coordinates: ";
    stream >> ob.x >> ob.y;
    return stream;
}

// file inserter for Coord
ofstream& operator<< (ofstream &stream, Coord ob){
    stream << ob.x << " " << ob.y << '\n';
    return stream;
}

// file extractor for Coord
ifstream& operator>> (ifstream &stream, Coord &ob){
    stream >> ob.x >> ob.y;
    return stream;
}

// main program
#include <iostream>
#include <fstream>
using namespace std;
#include "Coord.h"

int main( ){
    ofstream fout;
    ifstream fin;
    fout.open("test.txt", ios::out | ios::trunc);
    if (!fout) {
        perror("Cannot open test.txt file.\n");
        exit(1);
    }
    Coord A(7, 2), B(17, 20);
    cout << A << B;
    //cout << "\nWrites A (7,2) and B(17,20) in the file\n";
    fout << A << B;
    cin >> A;
    //cout << "\nThe entered values are: \n";
    cout << A;
    fout << A;
    //fout << std::ifstream::traits_type::eof( );// write EOF
    fout.close( );
    fin.open("test.txt");
    if (!fin) {
        perror ( "Cannot open test.txt file.\n");
        exit(1);
    }
    while ((fin)) { //while (1) {
        fin >> A;
        if (fin.eof( )) break;
        cout << A;
    }
/*
    while (!fin.eof( )) {//read twice the last values - past the end
        fin >> A; cout << A;
    }
*/
```

```
    fin.close( );
    cout << "\nEnd program";
    return 0;
}//main
```

Ex. 6 - Citire cu testare folosind funcția good() / Reading and testing using good() function

```
/*end of file managed using good()*/

#include <iostream>     // std::cin, std::cout
#include <fstream>      // std::ifstream
//using namespace std;
#include <stdlib.h>

const int dim1 =256;
const int dim2 =25;

int main( ) {
    char str[dim1];
    std::fstream io;

    std::cout << " File to generate \n";
    io.open("test.dat", std::ios::out);
    if (!io) {
        std::cout << "Output file cannot be opened\n";
        exit(1);
    }
    io.write("Test array", dim2);
    io << "\njhgfs jdgahjs";
    io.close( );
    std::cout << "Enter the name of an existing test.dat file: ";
    std::cin.get(str, dim1);    // get c-string
    std::ifstream is(str);    // open file
    if (!is) {
        std::cout << "Input file cannot be opened\n";
        exit(1);
    }
    while (is)   // loop while extraction from file is possible
    {
        char c = is.get();   // get character from file
        if (is.good()) std::cout << c;
    }
    is.close();  // close file
    std::cout << "\nEnd program";
    return 0;
}//main
```

Ex. 7 - Utilizarea flag-urilor de status și de erori / Using status and error flags

```
/* status flags */
// error state flags
#include <iostream>     // std::cout, std::ios
#include <sstream>      // std::stringstream

void print_state(const std::ios& stream);
```

```
int main( ) {
    std::stringstream stream;
    stream.clear(stream.goodbit);
    std::cout << "goodbit:"; print_state(stream); std::cout << '\n';
    stream.clear(stream.eofbit);
    std::cout << " eofbit:"; print_state(stream); std::cout << '\n';
    stream.clear(stream.failbit);
    std::cout << "failbit:"; print_state(stream); std::cout << '\n';
    stream.clear(stream.badbit);
    std::cout << " badbit:"; print_state(stream); std::cout << '\n';
    std::cout << "\nEnd program";
    return 0;
}//main

void print_state(const std::ios& stream) {
    std::cout << " good( )=" << stream.good( );
    std::cout << " eof( )=" << stream.eof( );
    std::cout << " fail( )=" << stream.fail( );
    std::cout << " bad( )=" << stream.bad( );
}
```

13.6. Lucru individual

1. Să se scrie un program care folosește metoda `seekg()` pentru poziționare la mijlocul fișierului și apoi afișează conținutul fișierului începând cu această poziție. Numele fișierului se citește din linia de comandă.
2. Scrieți un program care utilizează metoda `write()` pentru a scrie într-un fișier șiruri de caractere. Afișați apoi conținutul fișierului folosind metoda `get()`. Numele fișierului se va citi de la tastatură.
3. Scrieți o aplicație C++ care citește un fișier utilizând metoda `read()`. Verificați starea sistemului după fiecare operație de citire. Numele fișierului se va citi din linia de comandă. Afișați pe ecran conținutul fișierului.
4. Scrieți o aplicație C++ în care deschideți un fișier în mod binar pentru citire. Afișați un mesaj corespunzător dacă fișierul nu a fost creat în prealabil și cereți reintroducerea numelui fișierului. Presupunând că în fișierul deschis există înregistrări de tip agendă (*nume, localitate, număr de telefon*), utilizați supraîncărcarea operatorilor de inserție și extracție pentru afișarea pe ecran a conținutului fișierului.
5. Considerați clasa `Fractie` care are două atribute întregi private `a` și `b` pentru numărător și numitor, două metode de tip `set()` respectiv `get()` pentru atributele clasei. Declarați o metodă `simplifica()` care simplifică un obiect `Fractie`. Definiți un constructor explicit fără parametri care inițializează `a` cu 0 și `b` cu 1, și un constructor explicit cu doi parametri care va verifica posibilitatea definirii unei fracții *(b!=0).* Definiți un destructor explicit care afișează un mesaj. Supraîncărcați operatorii de adunare, scădere, înmulțire și împărțire (+,-,*,/) a fracțiilor folosind metode membre care și simplifică dacă e cazul rezultatele obținute. Supraîncărcați operatorii de intrare (>>, extracție) și ieșire (<<, inserție) cu funcții *friend* care permit citirea și scrierea obiectelor. Instanțiați două obiecte de tip `Fractie` cu date citite de la tastatură. Afișați atributele inițiale ale obiectelor folosind supraîncărcarea operatorului de ieșire. Citiți alte 4 obiecte de tip `Fractie` folosind supraîncărcarea operatorului de intrare. Efectuați operațiile implementate prin metodele membre (adunarea și scăderea primelor două, respectiv înmulțirea și împărțirea ultimelor două), stocând rezultatele în alte 4 obiecte. Afișați rezultatele. Scrieți într-un fișier cu numele introdus de la tastatură cele 4 obiecte inițiale precum și rezultatele obținute, pe rânduri diferite.

13.7. Individual work

1. Write a program that uses the `seekg()` method for mid-file positioning and then displays the file's content, starting with this position. The filename is read from the command line.
2. Write a program that uses the `write()` method for writing some character arrays into a file. Display the file's content using the `get()` method. The filename is read from the keyboard.
3. Write a C++ application that reads a file's content using the `read()` method. The obtained data is displayed on the screen. Check the system's state after each reading operation. The filename is read from the command line.
4. Write a C++ application that opens a binary file for reading. The filename is read from the keyboard. Display a message if the file doesn't exist and ask the user to re-enter the filename. Assuming that the file contains some agenda records (*name, city, phone number*) overload the insertion and extraction operators for reading the file's content and for displaying it on the screen.
5. Consider the `Fraction` class that has two private attributes `a` and `b` for the nominator and denominator and two corresponding setter and getter methods. Declare a method named `simplify()` that simplifies a fraction. Define an explicit constructor without parameters that initializes `a` with 0 and `b` with 1 and another explicit constructor that receives two parameters representing the values of the nominator and denominator. This constructor verifies if the fraction can be defined (`b` != 0). Overload the addition, subtraction, multiplication and division operators (+, -, *, /) using member methods that simplify (if necessary) the obtained results.
   Overload the input (>>, extraction) and output (<<, insertion) operators using friend functions that allow reading and writing the data related to an entire object. Instantiate two `Fraction` objects with data read from the keyboard. Display the initial attributes of the objects by usign the insertion operator. Read another four objects using the extraction operator. Perform the operations implemented with member methods (the addition and subtraction of the first two objects, the multiplication and division of the last ones) and store the results into another four objects. Display the results. Write into a file the original values and the obtained results, on different rows.

Bibliografie/References

- Mircea-Florin Vaida, Ligia-Domnica Chiorean, Lenuța Alboaie, Petre Gavril Pop, Cosmin Strilețchi, Kuderna-Iulian Bența, 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 Bența, Mircea-Florin Vaida, Petre Gavril Pop, Cosmin Strilețchi, C/C++ - Ghid teoretic si practic, Casa Cărții de Stiință, Cluj-Napoca, 2016, pp. 464, ISBN 978-606-17-1016-4

- 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

- Resurse web:

  https://en.cppreference.com/w/
  https://cplusplus.com/reference/


HELIOS
software