

POO – C++ - Laborator 1

Recapitularea unor elemente ale limbajului C

1. Structuri

Definire

Se foloseste cuvantul cheie *struct* urmat de numele structurii si enumerarea membrilor acesteia:

```
struct X
{
    int i;
    char c;
};
```

Utilizare

Elementele membre ale variabilelor de tip structura pot fi accesate folosind operatorul `:`:

```
struct X x1, x2[4];
x1.i = 4;
x2[2].c=getch();
```

2. Pointeri

Declarare

```
int *pi; /* pointer la o variabila de tip int */
char *pc; /* pointer la o variabila de tip char */
struct X *px; /* pointer la o variabila de tip struct X */
struct X x1, x2[4];
```

Utilizare

```
pi=&x1.i; /* atribuirea adresei unei variabile de */
pc=&x2[2]; /* acelasi tip cu al pointerului */
pc=&px->c;
*pi=4; /* referirea zonei de memorie adresate de */
*pc=getch(); /* pointer cu ajutorul operatorului */
```

3. Alocare dinamica

Exista aplicatii in care necesarul de memorie nu este cunoscut din faza de compilare si rezervarea unor zone de memorie de dimensiuni acoperitoare pentru toate datele ar depasi capacitatea memoriei disponibile. Singura solutie in acest caz o reprezinta alocarea/eliberarea interactiva de zone de memorie chiar in timpul executiei programului - **alocarea dinamica** de memorie. In acest scop, in programele C se folosesc functii de biblioteca ale caror prototipuri (descrieri ale tipului si argumentelor functiilor) se gasesc in fisierul header **alloc.h**. Cele mai utilizate functii pentru alocarea de memorie sunt:

```
void* malloc(unsigned size);
void* calloc(unsigned nelem, unsigned size);
```

Prima functie primeste ca unic argument numarul de octeti ce trebuie alocati si returneaza adresa de inceput a zonei de memorie alocate in caz de succes sau *NULL* (0) in caz de esec. A doua functie se comporta identic, incercand insa alocarea a *nelem* blocuri successive de *size* octeti.

Toate alocările dinamice se fac într-o zona de memorie destinată special acestui scop, zona numita **heap** (gramada). În funcție de modelul de memorie folosit, dimensiunea heap-ului variază de la dimensiunea

unui segment (64KO) minus dimensiunea programului pâna la dimensiunea întregii memorii disponibile minus aceeași dimensiune a programului.

Pentru eliberarea unei zone de memorie alocate dinamic se folosește funcția complementara

```
void free(void* addr);
```

unde *addr* reprezintă un pointer ce contine adresa de început a unei zone de memorie alocate dinamic.

```
int *pi;
struct X *px;
...
/* alocare memorie pentru o variabila de tip int */
pi = (int *)malloc(sizeof(int));
*pi=3;

/* alocare pentru un vector de 4 elemente de tip struct X */
px=(struct X *)calloc(4, sizeof(struct X));
px[2]->i=4;
...
/* eliberare memorie */
free(pi);
free(px);
```

4. Crearea unui proiect în MS Visual Studio 2013

Visual Studio 2013 este mediul de dezvoltare integrat (**IDE – Integrated Development Environment**) ce se va folosi pentru scrierea programelor C++. Mediul permite dezvoltarea de aplicații în mai multe limbi de programare, de diverse tipuri. În cadrul laboratorului se vor scrie programe C++ de tip consolă. Pentru a scrie un program în VS2013, la început trebuie să creem un proiect.

Pentru a crea un proiect, se vor parcurge următorii pași :

1. Se lansează în execuție mediul de dezvoltare VS2013
2. În meniul principal, *File* → *New* → *Project...* figura 1.1:

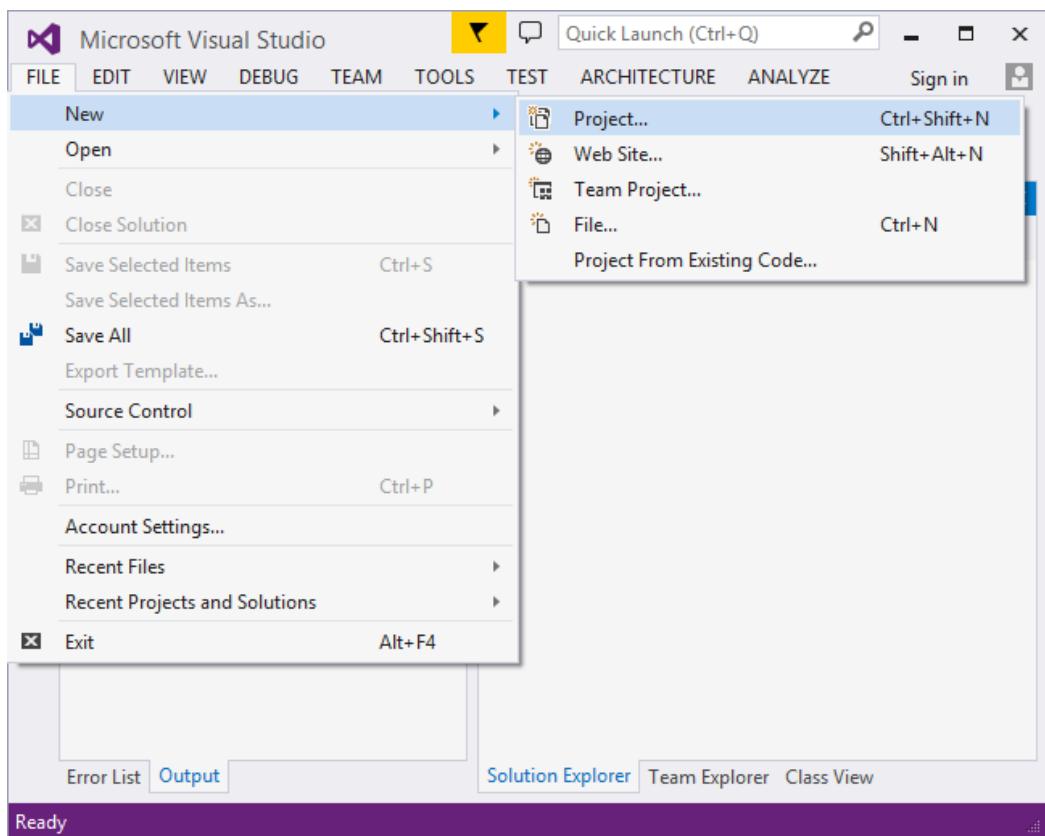


Fig. 1.1 Crearea unui nou proiect

3. În fereastra *New Project*, în stânga se observă caseta *Project types* (figura 1.2). Se alege *Other languages* → *Visual C++*

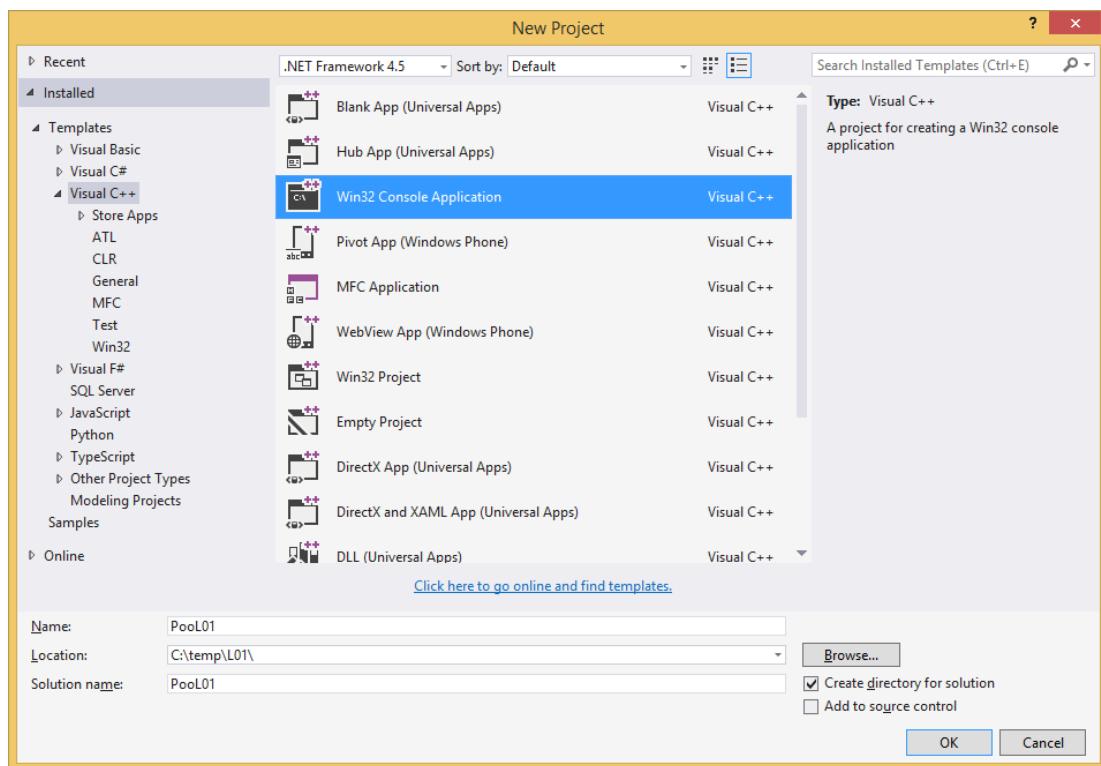


Fig. 1.2 Fereastra New Project

4. În partea dreaptă se observă caseta *Templates* din care se alege *Win32 Console Application*.

5. În partea de jos se observă casetele *Name* și *Location*. Se alege un nume pentru proiect. La *Location* se selectează directorul de lucru: *C:\temp* sau *D:\temp* după caz. (figura. 1.2)
6. Se apasă **OK**.
7. În continuare, apare fereastra *Welcome to the Win32 Application Wizard* (figura 1.3). Se apasă **Next, NU Finish!**

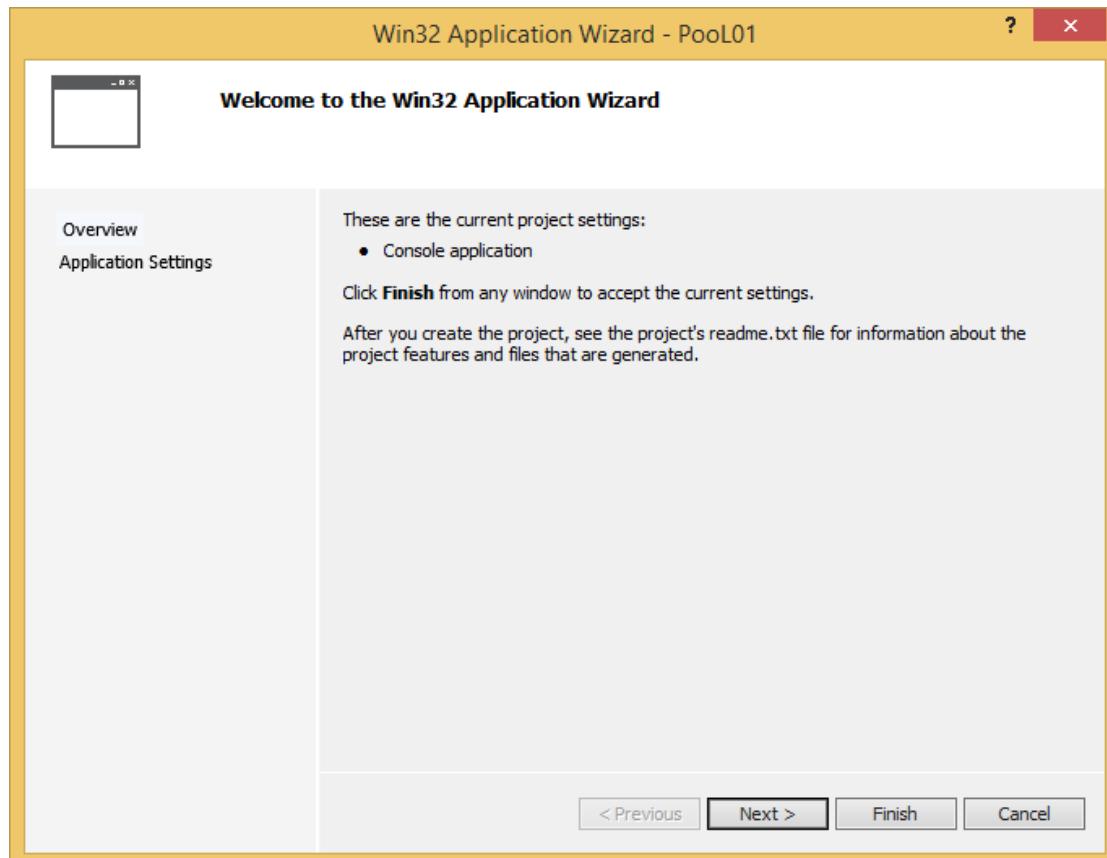


Fig. 1.3 Fereastra Welcome...

8. În noua fereastră apărută, la rubrica *Additional options*, se bifează *Empty project*. Se lasă celelalte setări neschimbate (figura 1.4):

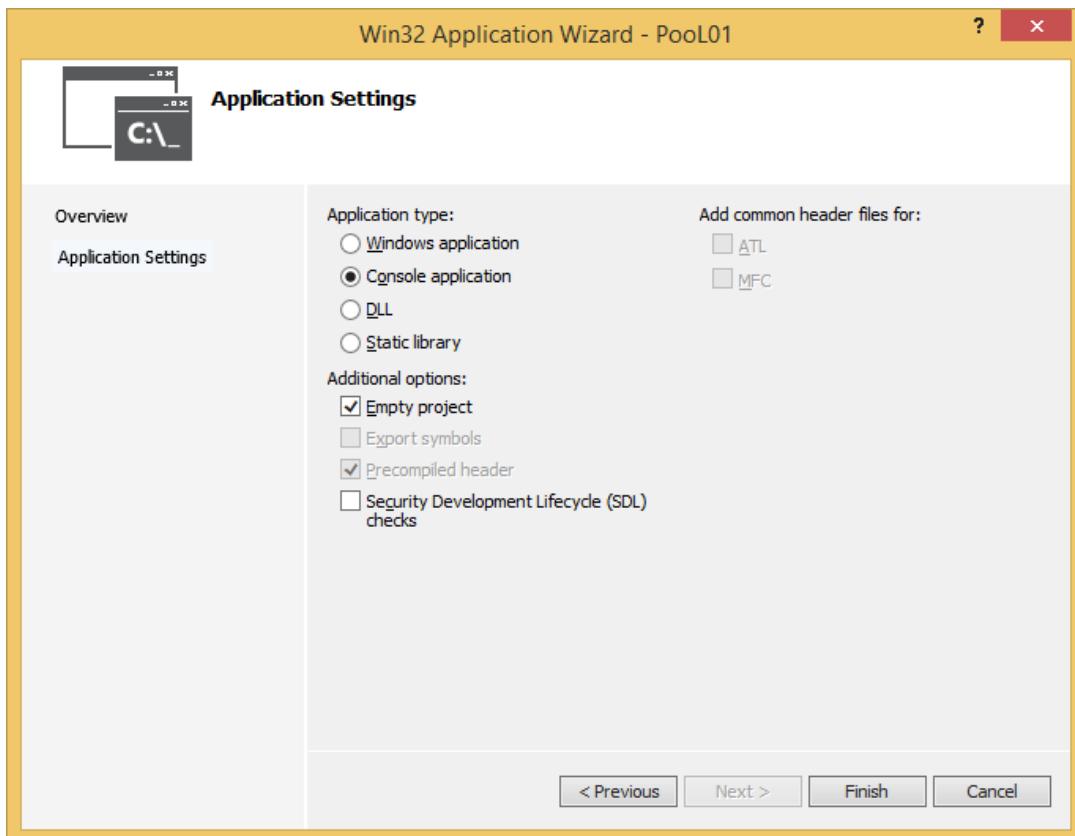


Fig.1.4 Fereastra Application Settings

9. Se apasă *Finish*.

Proiectul este creat și deschis în mediul de dezvoltare (figura 1.5).

Se pot observa următoarele ferestre:

- *Solution explorer* – în partea stânga. De aici se pot crea sau deschide fișierele proiectului. Inițial proiectul nu conține nici un fișier.
- *Start Page* – în restul ecranului. Această fereastră nu este utilă, ea poate fi închisă.

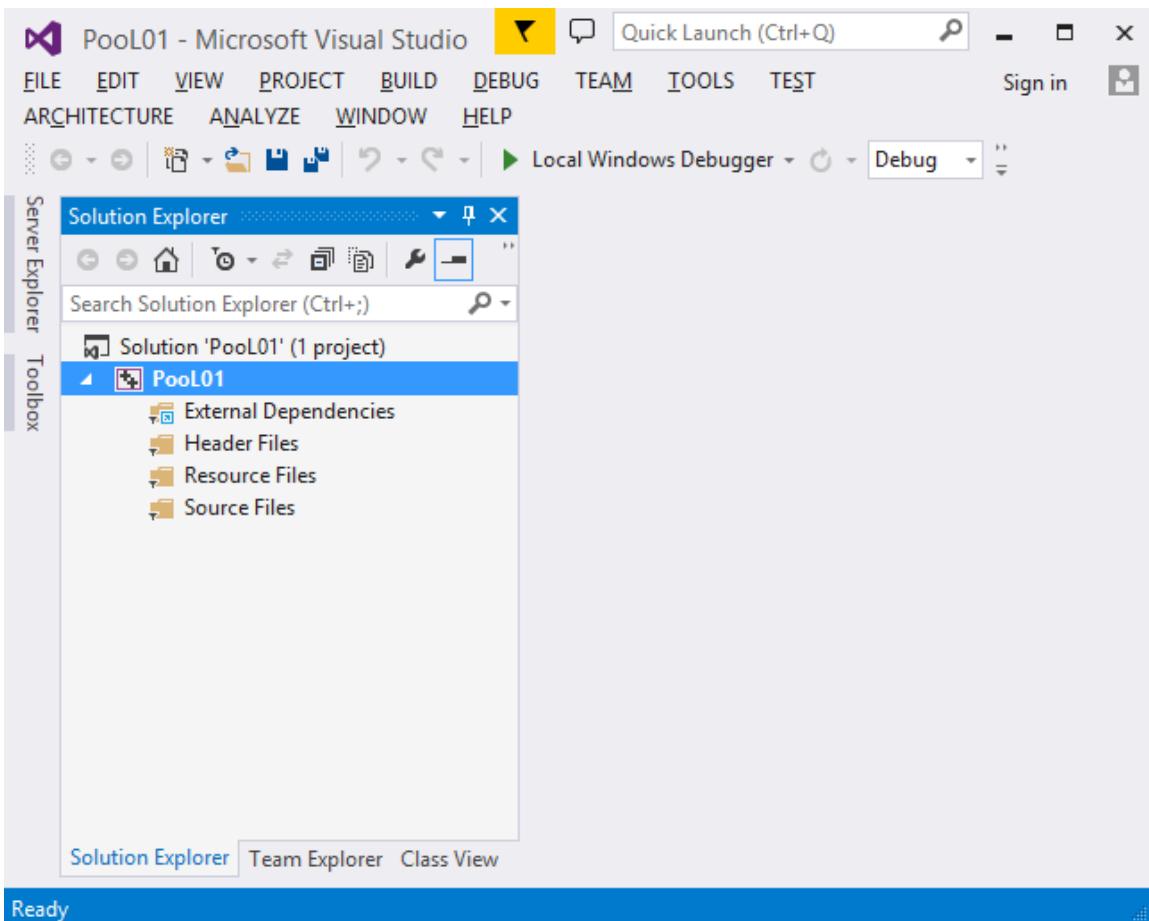


Fig. 1.5 Proiectul nou creat

5. Adăugarea unui fișier la proiect

Pentru a scrie un program în VS2013, trebuie adăugat un fișier sursă la proiect. Pentru aceasta se vor efectua următorii pași:

1. În *Solution Explorer*, click dreapta pe grupul *Source Files* → *Add* → *New Item...*(figura 1.6)

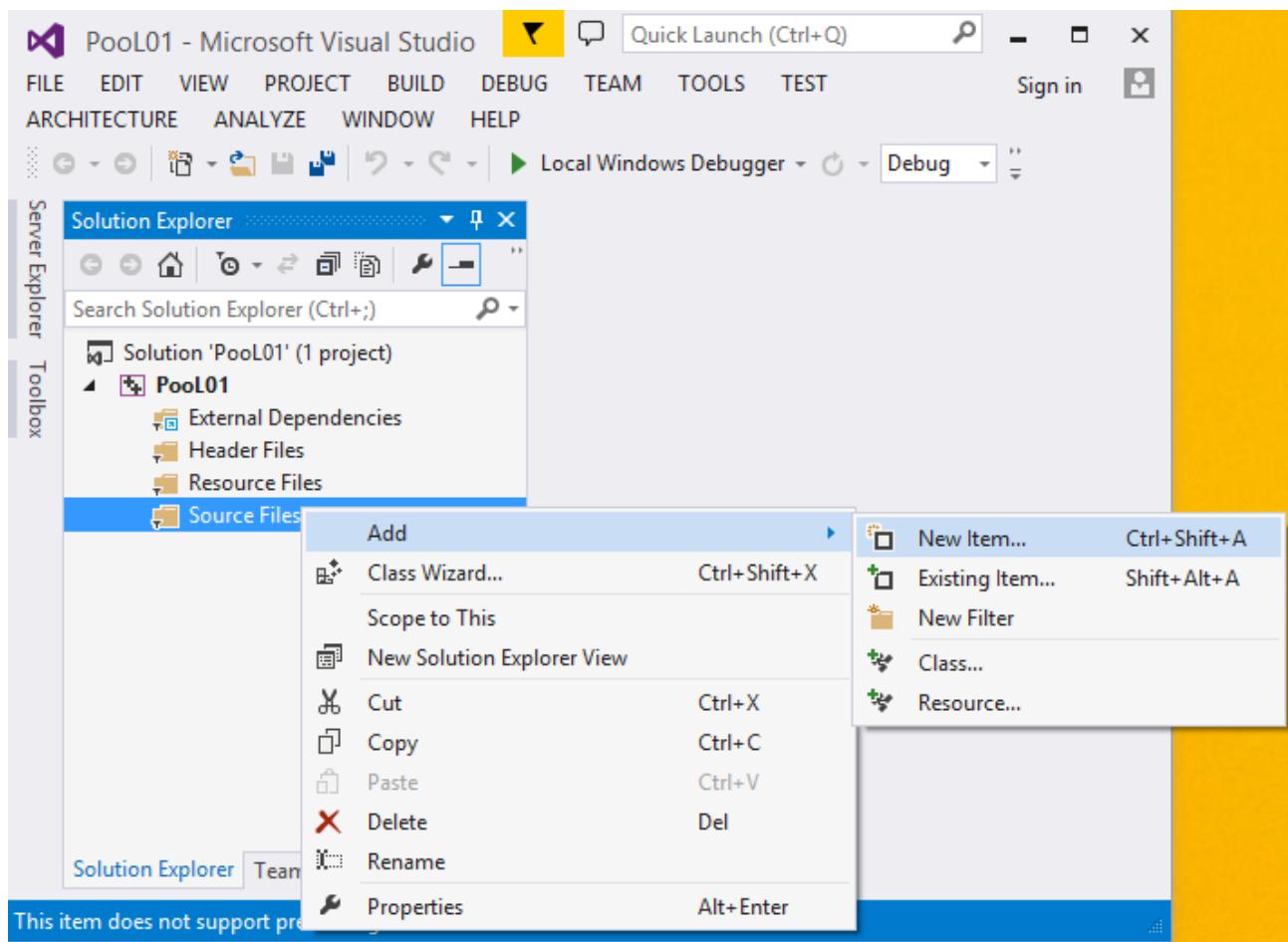


Fig. 1.6 Adăugare fișier sursă

2. Apare fereastra Add New Item (figura 1.7).
3. În caseta Templates se alege C++ File (.cpp)
4. La Name se introduce numele fișierului. Ca regulă ne scrisă, se va denumi fișierul care conține funcția `main()` `<numeProject>Main`; în cazul de față – `salutMain`.

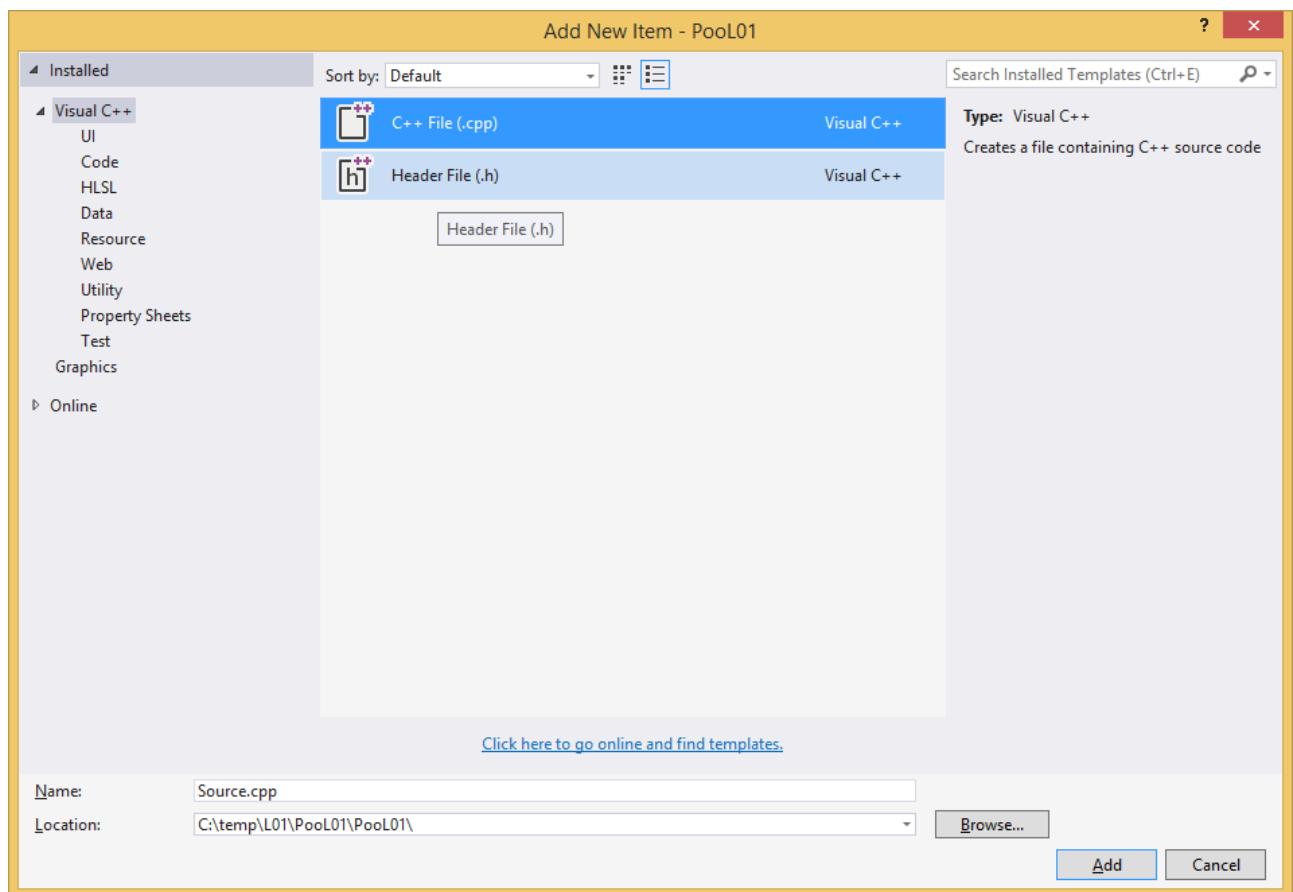


Fig. 1.7 Adăugare fișier sursă – partea 2

5. Se apasă *Add*. Noul fișier va fi creat și deschis în editor.

6. Scrierea programului

Se va scrie un program simplu care va afișa un mesaj la consolă (vezi figura 1.8):

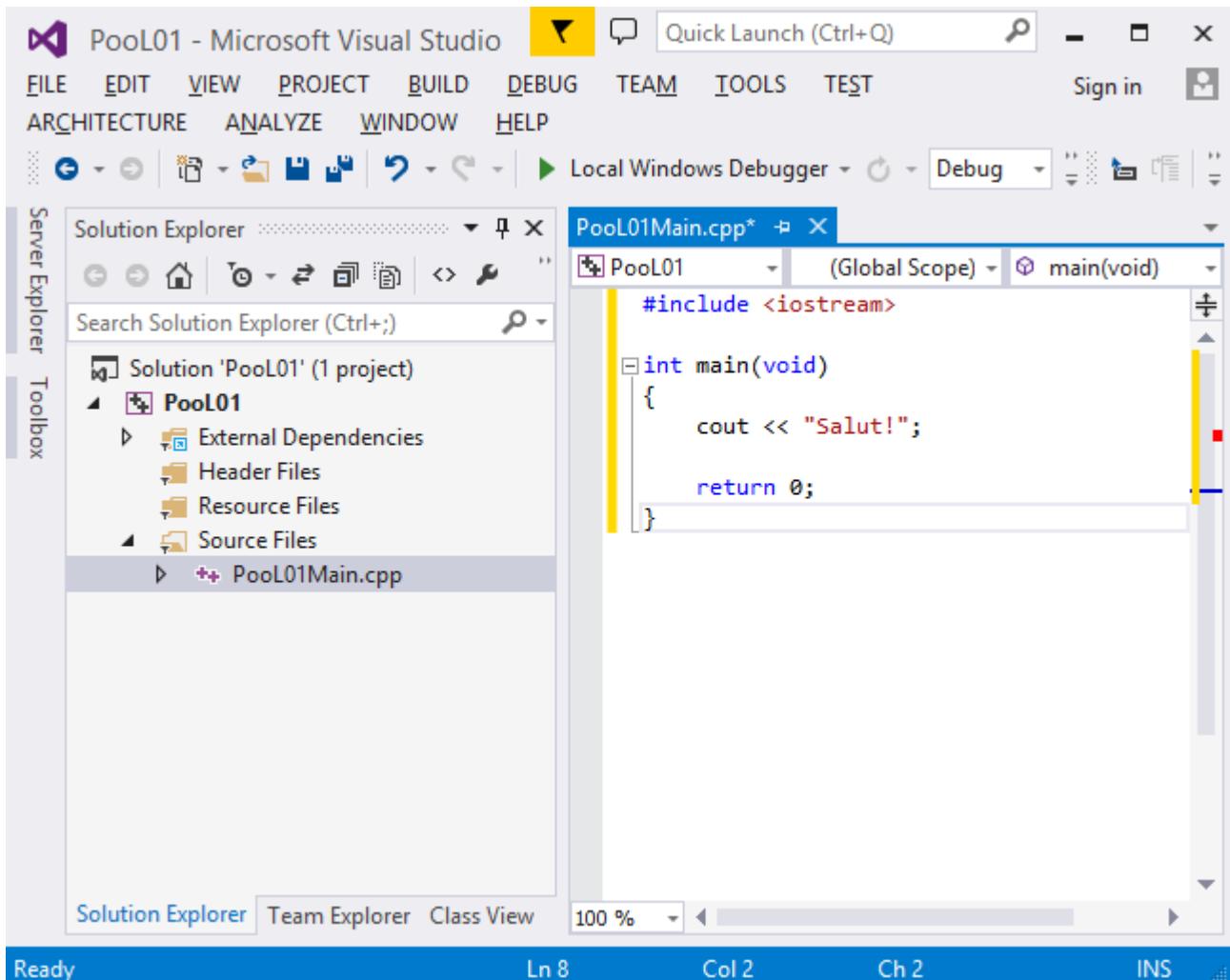


Fig. 1.8 Scrierea codului sursă în editor

În Visual Studio codul este formatat în mod automat în timp ce este scris. Poate fi formatat și ulterior apăsând **Ctrl+A**, iar apoi **Ctrl+K, Ctrl+F**.

7. Compilarea, rularea și detectarea erorilor

Pentru a compila proiectul se apasă **Ctrl+Alt+F7**, sau din meniul principal se selectează **Build → Rebuild Solution**.

Dacă sunt detectate erori de compilare acestea vor fi afișate în fereastra *Error List*. Se va introduce intenționat o eroare pentru a vedea facilitățile acestei ferestre. Astfel va înlocui linia `cout<<"Salut!";` cu `cout<<"Salut!"<<x;`. La compilare, deoarece variabila `x` nu este declarată se va genera o eroare și se va afișa în fereastra *Error List* următorul mesaj (figura 1.10):

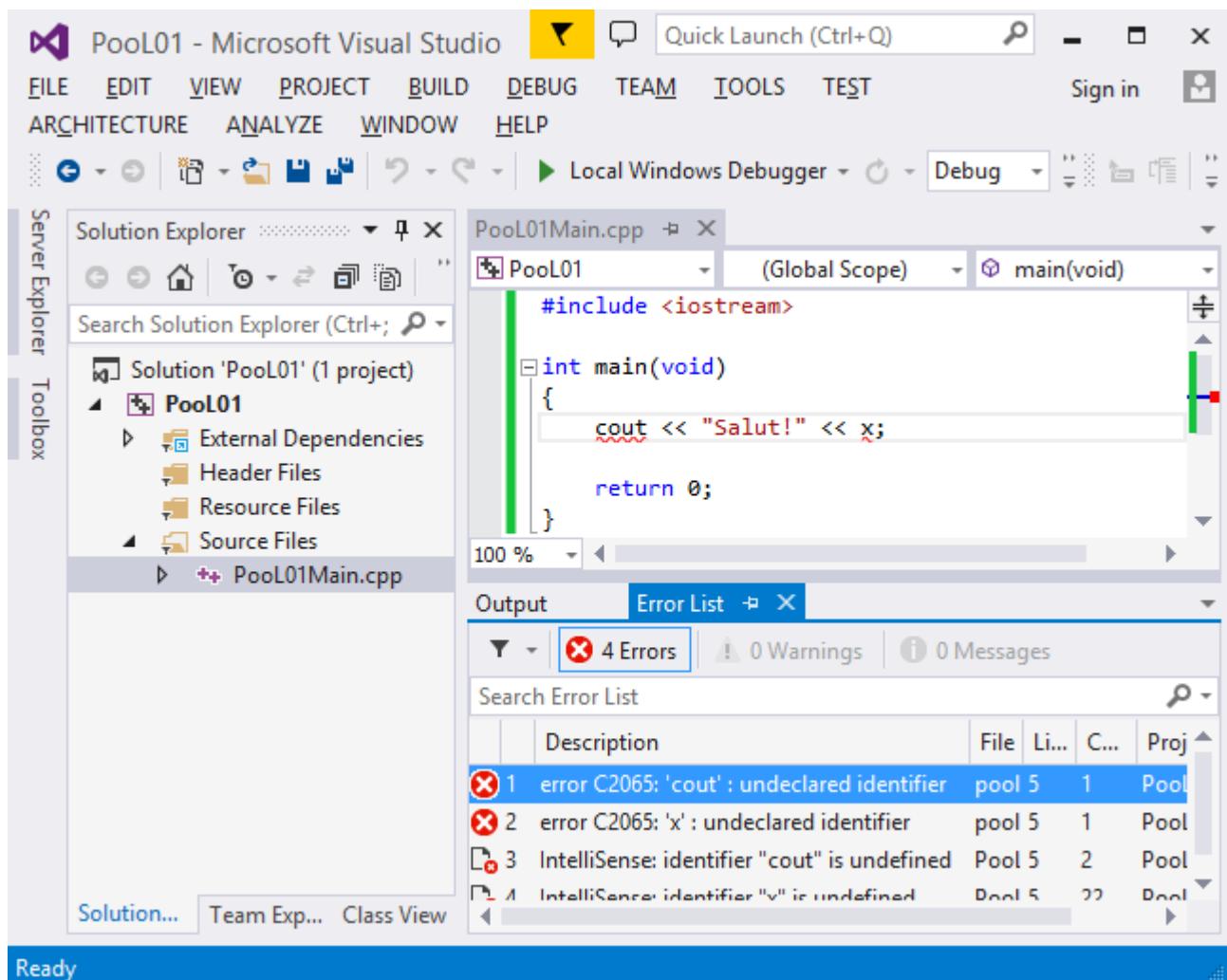


Fig. 1.10 Mesaj de eroare

Prin dublu-click peste mesajul de eroare, cursorul se va deplasa la linia de cod care conține eroarea respectivă. Alte tipuri de erori pot duce la mesaje de eroare neclare și chiar linia de cod a erorii poate fi indicată greșit.

În general se aplică următoarea regulă: eroarea trebuie căutată ori pe linia afișată în *Error List*, ori cu o linie mai sus. De exemplu, dacă în program, se șterge caracterul „;” de la sfârșitul uneia dintre linii, se observă că eroarea este localizată cu o linie mai jos.

Chiar dacă, după compilare, în fereastra *Error List* nu apar erori, dar apar warning-uri, acestea trebuie eliminate din codul sursă. De cele mai multe ori warning-urile duc la erori de execuție și se recomandă eliminarea acestora înainte de rularea proiectului.

Pentru a executa programul se apasă tasta **F5**. Se deschide o fereastră de tip consola în care rulează programul.

8. Scrierea / citirea cu cin și cout

În C++ s-a elaborat o modalitate mai simplă de scriere/citire la/de la consolă având același efect ca și funcțiile de bibliotecă `scanf()`/`printf()` din C. La începutul execuției fiecărui program sunt instantiate automat două variabile globale speciale `cin` și `cout`. Ele sunt folosite pentru citirea, respectiv scrierea la/de la consola.

Pentru a citi o variabilă de la consolă, se va scrie următoarea sintaxă:

```
int a;
cin >> a;
```

Operatorul `>>` are un rol special pentru variabila `cin`. Expresia:

```
cin >> a;
```

semnifică faptul că de la consolă este citită o valoare și depozitată în variabila `a`. Tipul variabilei din dreapta poate fi oricare din tipurile fundamentale: `int`, `char`, `float`, `double` sau `char*` care reprezintă un sir de caractere. Pentru fiecare tip enumerate mai sus, implicit citirea se va face corect.

Pentru a scrie o variabilă la consolă, folosim sintaxa:

```
char str[] = "abc";
cout << str;
```

În mod similar, operatorul `<<` are o semnificație specială pentru variabila `cout`. Expresia:

```
cout << str;
```

semnifică faptul că variabila `str` este scrisă la consolă. Variabilele scrise pot avea aceleași tipuri ca și cele citite cu `cin`. Aceste tipuri au fost enumerate mai sus pentru variabila `cin`.

Se observă că în exemplul de mai sus a fost scrisă la consolă o variabilă de tip `char[]`, tip care nu a fost menționat în lista de tipuri suportate pentru operandul dreapta. Totuși, utilizarea lui a fost posibilă într-o expresie cu `cout`. De ce?

Variabilele `cin` și `cout` sunt definite în header-ul `<iostream>`. Pentru a fi utilizate trebuie să adăugăm la începutul programului următoarele linii:

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

Aceste variabile speciale (`cin` și `cout`) sunt **obiecte**. Obiectele vor fi studiate în detaliu ulterior.

Iată un exemplu complet folosind noile facilități de scriere/citire din C++:

```
// exemplu: cin si cout
#include <iostream>
using namespace std;

int main() {
    int iVal;
    char sVal[30];

    cout << "Introduceti un numar: ";
    cin >> iVal;
    cout << "Si un sir de caractere: ";
    cin >> sVal;
    cout << "Numarul este: " << iVal << "\n"
        << "Sirul este: " << sVal << endl;
    return 0;
}
```

Exemplu de rulare:

```
Introduceti un numar: 12
Si un sir de caractere: abc
Numarul este: 12
Sirul este: abc
```

Un element nou este cuvântul `endl`. Acesta este o funcție specială numită și manipulator care trimită o nouă linie (echivalent cu `"\n"`) la ieșirea standard, golind și bufferul acesteia.

Atât expresiile cu `cin` cât și cele cu `cout` pot fi înlántuite.

Expresia

```
cout << a << " " << b;
```

este echivalentă cu

```
cout << a;
cout << " ";
cout << b;
```

Comparativ cu funcțiile `printf()` / `scanf()` din C, expresiile cu `cin` și `cout` sunt mai simple și mai ușor de înțeles. Nu mai sunt necesari specifiicatorii de format. Dezavantajul constă în faptul că nu se pot face afișări formatare pe un anumit număr de caractere. O afișare de genul:

```
printf("%7.2f", f);
```

nu are echivalent folosind `cout`, decât dacă se folosesc manipulatori speciali pentru formatare.

9. Un program mai complex

Fie următorul program: să se citească de la consolă un vector de `n` siruri de caractere, se alocă spațiu de memorie strict necesar și sortează sirurile în ordine crescătoare.

În Visual Studio, se poate crea o soluție nouă (o soluție reprezintă o mulțime de proiecte deschise simultan), sau se poate adăuga un proiect nou în cadrul soluției existente. Se va crea un proiect nou și se vor adăuga trei fișiere: `SortareSiruri.h`, `SortareSiruri.cpp`, `SortareSiruriMain.cpp`. Codul sursă este prezentat mai jos.

SortareSiruri.h

```
#ifndef _SortareSiruri_
#define _SortareSiruri_

char **citireVSiruri(int n);
void sortareVSiruri(char **vsiruri, int n);
void afisareVSiruri(char **vsiruri, int n);
void dealocareVSiruri(char **vsiruri, int n);

#endif
```

SortareSiruri.cpp

```
#include<iostream>
#include<string.h>
#include"SortareSiruri.h"
using namespace std;

char **citireVSiruri(int n) {
    char buffer[100];
    char **vsiruri = (char**)calloc(n, sizeof(char*));
    cin.ignore(100, '\n');
    for(int i=0; i<n; i++) {
        int len;
        cin.getline(buffer, 100);
        len = strlen(buffer);
        vsiruri[i]=(char*)calloc(len+1,sizeof(char));
        strcpy(vsiruri[i], buffer);
    }
    return vsiruri;
}

void sortareVSiruri(char **vsiruri, int n) {
    int suntPerm = 1;
    while(suntPerm) {
        suntPerm = 0;
        for(int i=0; i<n-1; i++) {
            if(strcmp(vsiruri[i],
                      vsiruri[i+1]) > 0) {
                char *aux = vsiruri[i];
                vsiruri[i] = vsiruri[i+1];
                vsiruri[i+1] = aux;
                suntPerm = 1;
            }
        }
    }
}
```

```

        }
    }

void afisareVSiruri(char **vsiruri, int n) {
    cout << "Sirurile sortate sunt:" << endl;
    for(int i=0; i<n; i++) {
        cout << vsiruri[i] << endl;
    }
}

void dealocareVSiruri(char **vsiruri, int n) {
    for(int i=0; i<n; i++) {
        free(vsiruri[i]);
    }
    free(vsiruri);
}

```

SortareSiruriMain.cpp

```

#include<iostream>
#include"SortareSiruri.h"
using namespace std;

int main() {
    int n;
    char** vsiruri;

    cout << "n=";
    cin >> n;
    vsiruri = citireVSiruri(n);
    sortareVSiruri(vsiruri, n);
    afisareVSiruri(vsiruri, n);
    dealocareVSiruri(vsiruri, n);
    return 0;
}

```

Acest program conține câteva elemente noi:

- În fișierul *SortareSiruri.cpp*, avem următoarea linie în funcția *citireVSiruri()*:

```
cin.ignore(100, '\n');
```

Acest apel este necesar înainte de apelarea funcției *cin.getline()*, care citește o linie de caractere de la consolă. Rolul apelului este să ignore caracterele din bufferul de intrare rămase de la citirea anterioară. Pentru detalii, se poate studia documentația acestei funcții. Ea are același rol ca și funcția *fflush(stdin)* în C, cu observația că *fflush(stdin)* golește bufferul de intrare.

- În aceeași fișier *SortareSiruri.cpp*, este apelul:

```
cin.getline(buffer, 100);
```

Funcția *cin.getline()* citește o linie de la consolă, sub forma unui sir de caractere și o depune în buffer. Al doilea-lea parametru – 100 – este numărul maxim de caractere care pot fi citite. Funcția este echivalentă cu *fgets()* din C.

A fost necesar utilizarea acestor funcții noi, în loc de *fflush()*, *fgets()* din C, deoarece nu este bine combinarea funcțiilor de lucru cu consola din C cu cele din C++. La utilizarea lor mixtă pot apărea incompatibilități.

Recomandare.

Toate programele realizate în cadrul laboratoarelor de POO vor fi concepute ca proiecte, respectând aceleași standarde ca și la obiectul Programarea Calculatoarelor. Adică unul sau mai multe fișiere header, unul sau mai multe fișiere cu funcții și un fișier cu funcția *main*.

10. Depanare

Depanarea (debug) este facilitatea oferită de mediile de dezvoltare de a analiza procesul de execuție a unui program în scopul de a detecta erorile. În modul de lucru *Debug*, se poate rula programul instrucțiune cu instrucțiune, urmărind valoarea unor variabile după fiecare instrucțiune executată. De asemenea, se pot stabili anumite linii de cod la care dorim ca programul să se oprească și astfel, să vizualizăm valoarea variabilelor alese de noi doar în acele puncte din program. Aceste linii la care se dorește întreruperea execuției programului se numesc puncte de oprire (breakpoint-uri).

De exemplu, se va considera cazul unei erori des întâlnite și se presupune că în programul *SortareSiruri.cpp* afișarea este greșită. Se va verifica în primul rând dacă alocarea memoriei și citirea a fost efectuată corect iar apoi, dacă sortarea a fost corectă. Pentru aceasta, se vor plasa două breakpoint-uri în funcția *main*, unul după funcția de citire și altul după funcția de sortare. Pentru plasarea unui breakpoint, se va poziționa cursorul în dreptul liniei de cod la care se dorește oprirea execuției programului și se apasă tasta **F9**. Se poate scoate un breakpoint tot prin apăsarea tastei **F9**. În partea dreaptă a editorului va apărea o bulină roșie pentru fiecare breakpoint astfel plasat (figura 1.11):

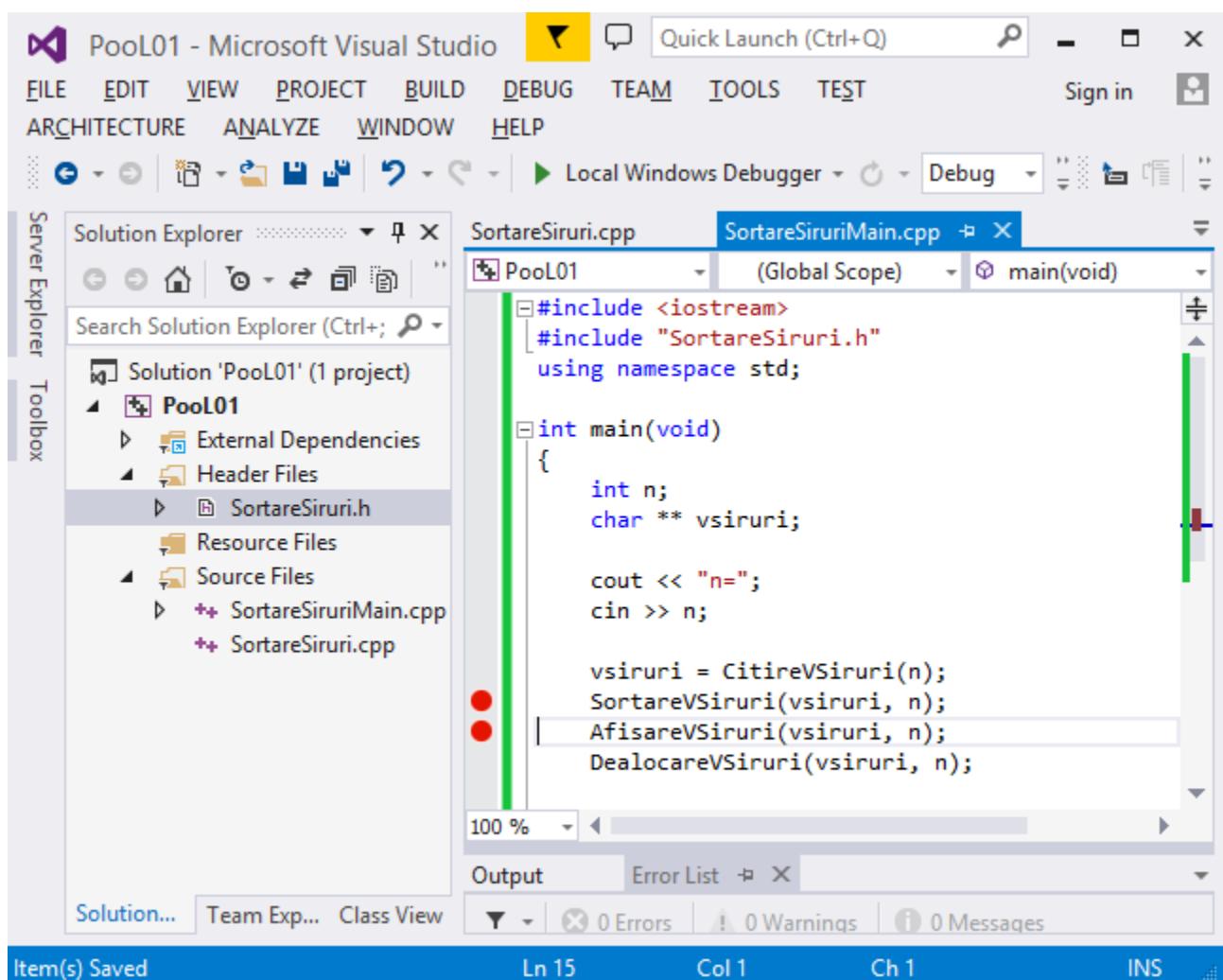


Fig. 1.11 Breakpoint-uri

În continuare, se apasă tasta **F5** pentru a rula programul în mod Debug. Se observă că programul își începe execuția normal, sunt cerute datele de intrare, iar după ce le introducem consola se blochează. În acel moment se revine la Visual Studio și se observă că aranjamentul ferestrelor s-a schimbat, iar în dreptul primului breakpoint a apărut o săgeată (figura 1.12), săgeata indică instrucțiunea la care s-a oprit execuția programului.

Atenție! *Instrucțiunea la care se află săgeata încă nu s-a executat, dar instrucțiunea anterioară a fost executată!*

În acest moment, pot fi vizualizate valorile unor variabile din program. În mod evident, interesează valoarea celor două variabile definite în `main`: `n` și `vsiruri`. Pentru a le vizualiza, alege meniul principal → `Debug` → `Windows` → `Watch` → `Watch 1`.

În partea de jos a mediului de dezvoltare, apare fereastra `Watch 1` (figura 1.12). În această fereastră, se dă click pe coloana `Name` și se introduc numele variabilelor ce se doresc să fie vizualizate – întâi `n`, se apasă `ENTER`, pe urmă `vsiruri`, se apasă din nou `ENTER`:

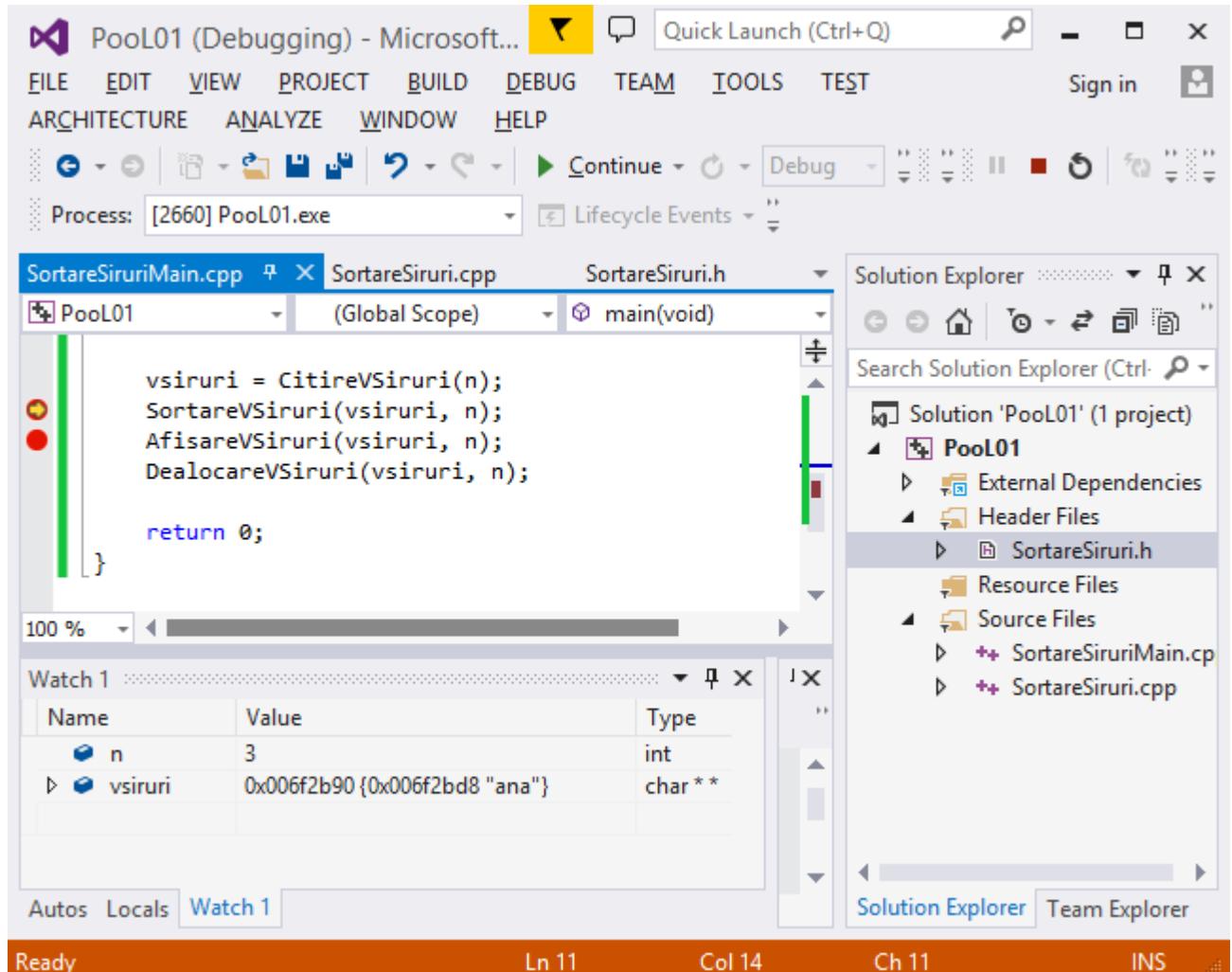


Fig. 1.12 Fereastra Watch

În a doua coloană (`Value`), va fi afișată valoarea variabilelor la acel moment al execuției programului, iar în cea de-a treia coloană (`Type`), tipul acestora. La variabila `n`, valoarea este un număr în baza zece. Însă la variabila de tip pointer la pointer, `vsiruri`, IDE-ul afișează o valoare în hexazecimal ce reprezintă adresa de memorie a primului element din sir și care nu este așa de utilă.

Fereastra `Watch` permite vizualizarea nu doar a variabilelor, ci și a expresiilor. De exemplu, se poate vizualiza elementele vectorului `vsiruri` (figura 1.13):

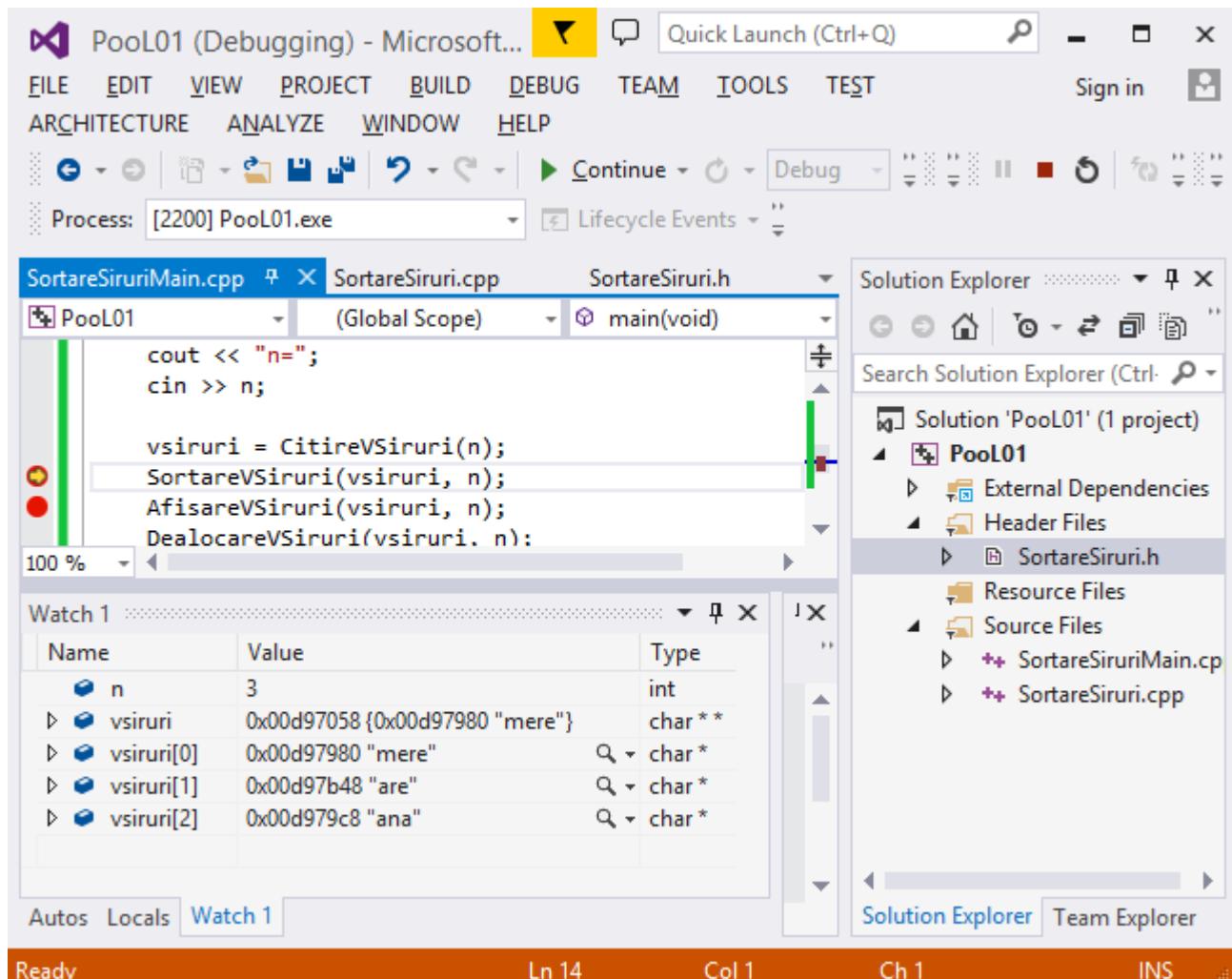


Fig. 1.13 Vizualizarea elementelor unui vector

Se observă că de data aceasta fiecare sir de caractere este afișat corect.

Citirea s-a efectuat aşa cum era de așteptat. Până în acest punct programul se execută corect. Se apăsăm tasta **F5** pentru a continua execuția programului până la următorul breakpoint. Se observă următoarele (figura 1.14):

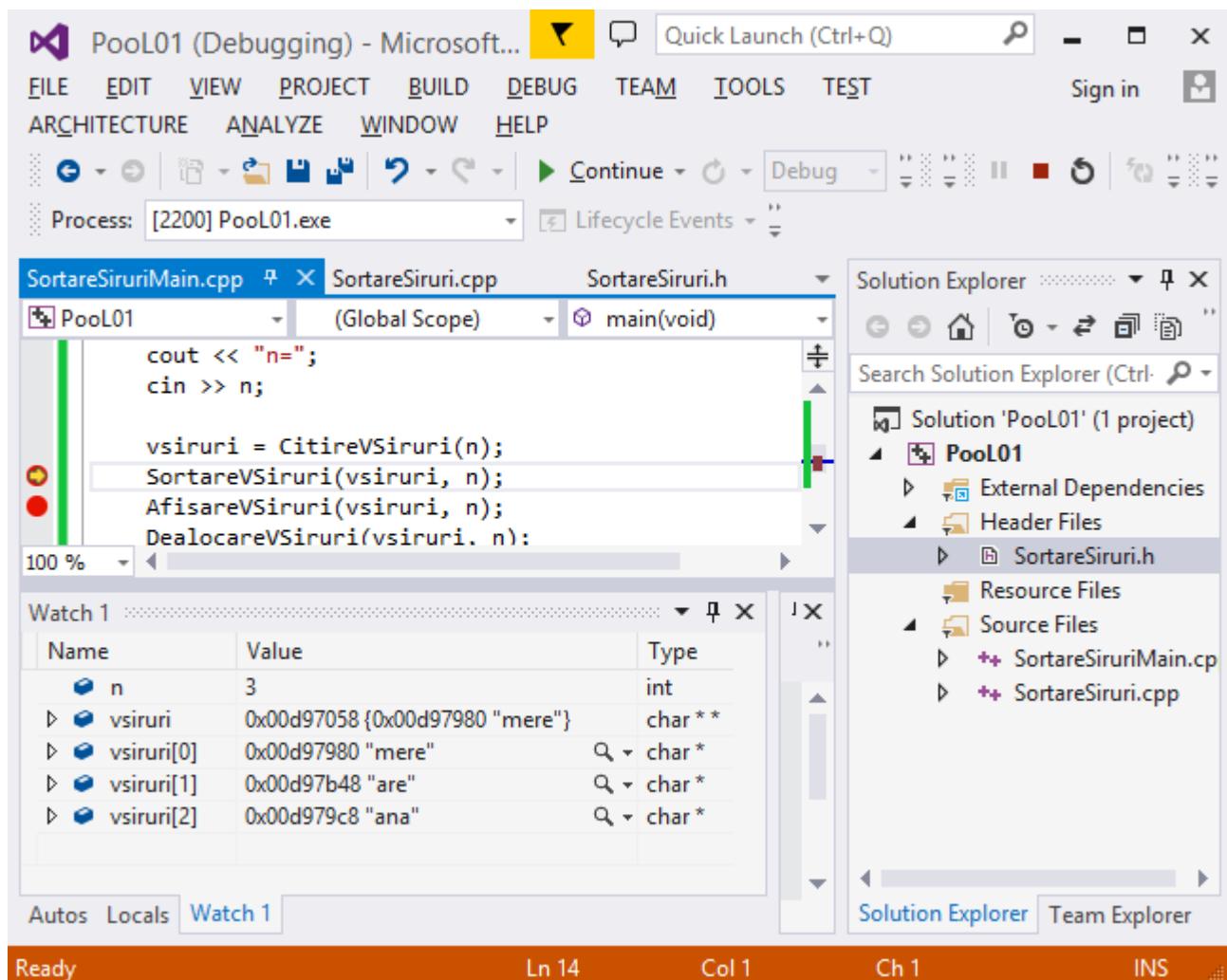


Fig. 1.14 Breakpoint după sortare

Ordinea elementelor în `vsiruri` s-a schimbat, elementele sunt sortate crescător, aşa cum era de așteptat. Unele dintre elementele schimbate sunt afișate cu roșu. Atât citarea cât și sortarea sunt corecte.

Explorați meniul *Debug* în timp ce vă aflați într-un breakpoint pentru a afla și alte facilități de debug.

Mai jos sunt prezentate combinații de taste utile în Visual Studio 2013.

Combinată de taste	Efect
Ctrl + C	Copy - copiere
Ctrl + V	Paste - afișare
Ctrl + A	Select all – selectare totală
Ctrl + K, F	Format selected – formatare selecție
Ctrl + A, K, F	Format all – formatare totală
Ctrl + Shift + B	Build all – compilare proiect
F5	Debug, continue after breakpoint – Intrare în depanare sau continuarea execuției după breakpoint
F9	Insert / remove breakpoint Adăugarea / eliminare breakpoint

11. Documentație

Sursa recomandată pentru documentare este situl <http://www.cplusplus.com/>.

În particular, sunt recomandate următoarele link-uri:

• <http://www.cplusplus.com/reference/clibrary/> unde se poate găsi documentația completă a tuturor fișierelor de bibliotecă din C și a funcțiilor din acestea. Documentația este similară cu cea din Borland C.

• <http://www.cplusplus.com/reference/iostream/> unde se poate găsi documentația claselor ce realizează operații de intrare/ieșire (intrare/ieșire standard – cin, cout și fișiere).

• <http://www.cplusplus.com/doc/tutorial/> unde se poate găsi un material didactic alternativ despre limbajul C++.

12. Exerciții

1. Creeți un nou proiect care să conțină programul prezentat în secțiunea 7.
2. Compilați și rulați programul de la exercițiul 1.
3. Executați pas cu pas programul.
4. Introduceți în mod intenționat o eroare în program, comentând linia:

```
suntPerm = 1;
```

din funcția sortareVSiruri().

Plasați două breakpoint-uri conform indicațiilor din secțiunea 8. Testați programul cu o intrare de cinci siruri și verificați dacă la al doilea breakpoint sirurile sunt sau nu sunt sortate corect.

5. Încercăți să detectați eroarea, presupunând că nu știți unde este. Localizați prima linie de cod care se execută după ce se detectează că două siruri trebuie inversate. Introduceți un breakpoint pe acea linie. Vizualizați toate variabilele locale și cele două siruri care urmează să fie inversate. Ce expresii veți introduce în fereastra *Watch* pentru cele două siruri?

De câte ori ar trebui să se realizeze inversarea sirurilor și de câte ori are loc în realitate? Ce puteți spune despre comportamentul programului?

6. Introduceți un breakpoint pe linia while și demonstrați cu ajutorul lui că bucla while se execută doar o dată în programul eronat.

7. Corectați eroarea în program. Numărați de câte ori se execută bucla while și de câte ori are loc inversarea a două siruri consecutive în funcția de sortare.

8. Creati un tip de date structura numit catalog care sa contine numarul de studenti, un pointer pe siruri de caractere (vector alocat dinamic de siruri) si doi pointeri pe functii pentru sortare alfabetica si sortare dupa lungime si un pointer pe functie pentru citirea datelor. Definiti:

- a. Cele două functii de sortare
- b. Functia de citire a sirurilor de caractere de la intrarea standard
- c. O functie de creare a unui catalog (aloca memorie pentru un tip de date catalog, initializeaza pointerii pe functii cu adresele functiilor definite mai sus, apeleaza functia de citire a datelor)
- d. O functie de distrugere a unei variabile de tip catalog (dealoca spatiul de memorie ocupat)

Scripti un program care sa utilizeze un astfel de tip de date folosind compilatorul de C.

Modificati programul pentru a utiliza facilitatile oferite de compilatorul de C++.

POO – C++ - Laborator 2

Definirea si utilizarea tipurilor de data structură in C și C++

1. (ANSI C) Creați un tip de date structură numit *student* care să conțină: *număr matricol* de tip *int*, *nume* de tip sir de caractere *char[]*, *gen* de tip *char*, *nota* de tip firgula mobilă simplă precizie *float*, doi pointeri pe funcții pentru citire date *void (read*)(student*st)* și unul pentru afisare date *void (write*)(student*st)*.

Citiți de la tastatura (folosind *cin*) numărul de studenți și alocați dinamic memorie pentru un vector de studenți (folosiți *v = new student[n]*).

Definiți două funcții: *void ReadData(student*st){...}* și *void WriteData(student*st){...}* care citesc și afisează membrii unei variabile de tip structură *student*. Parcurgeți vectorul cu elemente de tip *student* și initializați pointerii la funcții *void (read*)(student*st)* și *void (write*)(student*st)* cu funcțiile *void ReadData(student*st)* și *void WriteData(student*st)*.

Parcurgeți vectorul cu elemente de tip *student* și apelați funcțiile: *v[i].read(&v[i])* și *v[i].write(&v[i])* pentru citirea datelor și respectiv pentru afisarea datelor.

2. (C++) Declarați două funcții membre ale structurii *student* *void Read(void)* și *void Write(void)* și definiți aceste funcții în afara structurii: *void student::Read(void){...}* și *void student::Write(void){...}*.

Parcurgeți vectorul cu elemente de tip *student* și apelați funcțiile: *v[i].Read()* și *v[i].Write()* pentru citirea datelor și respectiv pentru afisarea datelor.

Analizați și discutați cele două abordări.

3. Rezolvați problema 8 din laboratorul 1 declarand ca membre ale structurii funcțiile pentru sortare alfabetică, sortare după lungime, citire și afișare de date.

obs. nu uitati dealocare zonelor de memorie (folosiți *delete []v*)

POO - C++ - Laborator 3

Cuprins

Namespace.....	1
Mecanisme de transfer a parametrilor.....	5
Domeniul de valabilitate și vizibilitate	5
Funcții cu parametri implicați	5
Supraîncărcarea funcțiilor.....	6
Tema de laborator:	7

Namespace

O problemă care ar putea să apară în programele C, este legată de faptul că, pe măsură ce dimensiunea programului crește, este din ce în ce mai greu să se evite duplicarea numelor pentru funcții și variabile. C++-ul standard oferă un mecanism pentru evitarea acestor coliziuni de nume, și anume prin utilizarea namespace-urilor.

Se vor grupa declarații / definiții la nivelul unor namespace-uri, și în acest caz, numele unor variabile sau funcții se poate repeta dar la nivelul unor spații de nume diferite. Datorită faptului că ele vor fi plasate în namespace-uri (spații de nume) diferite nu vor exista coliziuni.

Librările limbajului C++ standard sunt plasate în spațiul de nume **std (standard)**. De aceea în momentul în care vor fi folosite aceste librării C++ standard se va utiliza directiva **using**.

Exemplu1.cpp

```
#include <conio.h>
#include <iostream>
using namespace std;

namespace C1_1
{
    int x = 10;
    float y = .5;
}

namespace C1_2
{
    double x = 2.5;
    char y = 'x';
}

int main()
{
```

```

cout << C1_1::x << "\t";
cout << C1_2::x << endl;

using C1_1::y; //using declaration
cout << y << "\t";
cout << C1_2::y << endl;

using namespace C1_1; //using directive
cout << x << "\t";
cout << y << endl;

_getch();
return 0;
}

```

Rulare:

```

10  2.5
0.5  x
10  0.5

```

Un namespace poate fi continuat în mai multe fișiere header, nu se consideră o redefinire a spațiului de nume, ci doar o continuare a celui deja definit:

Header1.h

```

#ifndef _HEADER1_H
#define _HEADER1_H

namespace lib
{
    extern int x;
    void f();
    //.....
}
#endif

```

Header2.h

```

#ifndef _HEADER2_H
#define _HEADER2_H

#include "Header1.h"

namespace lib
{
    extern float y;
    void g();
    //.....
}
#endif

```

Exemplu2.cpp

```

#include <conio.h>
#include <iostream>
using namespace std;

#include "Header1.h"
#include "Header2.h"

using namespace lib;

```

```

int lib::x = 5;
float lib::y;

void lib::f()
{
    cout << "f() :" << x << endl;
}

void lib::g()
{
    cout << "g() :" << y << endl;
}

int main()
{
    int x = 10;
    cout << x << endl;
    f();
    g();

    getch();
    return 0;
}

```

Rulare:

```

10
f():5
g():0

```

Spre deosebire de o **directive using**, care tratează numele introduse ca fiind globale scopului, o **declarație using**, este o declarație care se face în interiorul scopului curent. Practic, prin utilizarea **declarației using** se poate suprascrie un nume introdus prin intermediu unei **directive using**.

Header.h
<pre> #ifndef _HEADER_H #define _HEADER_H namespace Functii1 { void f(); void g(); } namespace Functii2 { void f(int x); void f(); void g(); } #endif </pre>

Functii.cpp
<pre> #include <iostream> using namespace std; #include "header.h" </pre>

```

void Functii1::f()
{
    cout<<"Functii1: f()" << endl;
}

void Functii1::g()
{
    cout<<"Functii1: g()" << endl;
}

void Functii2::f(int x)
{
    cout<<"Functii2: f(" << x << ")" << endl;
}

void Functii2::f()
{
    cout<<"Functii2: f()" << endl;
}

void Functii2::g()
{
    cout<<"Functii2: g()" << endl;
}

```

Exemplu3.cpp

```

#include <conio.h>
#include "header.h"

void h()
{
    using namespace Functii1; //directive using
    using Functii2::f; //declaratia using
    f(5);
    f();
    Functii1::f();
}
int main()
{
    h();
    _getch();
    return 0;
}

```

Rulare:

```

Functii2: f(5)
Functii2: f()
Functii1: f()

```

Notă: La declarația using: `using Functii2::f;` s-a folosit numai numele identificatorului (funcția **f**) fără nici o informație despre tipul argumentelor funcției.

Dacă namespace-ul conține un set de funcții supraîncărcate cu același nume, prin declarația using se introduc toate funcțiile care se află în acest set.

Se va încerca să se evite cazurile de ambiguitate care pot să apară în aceste situații.

Mecanisme de transfer a parametrilor

Referința este un alias pentru o anume variabilă. Dacă în C, transmiterea parametrilor unei funcții se face prin valoare (inclusiv și pentru pointeri), în C++ se adaugă și transmiterea parametrilor prin referință.

Dacă tipul pointer se introduce prin construcția: **tip ***, tipul referință se introduce prin **tip &**.

O variabilă referință trebuie să fie inițializată la definirea sau declararea ei cu numele unei alte variabile.

```
int i;  
int &j = i;
```

Variabila **j** este un nume alternativ pentru **i**, cu această referință se poate accesa întregul păstrat în zona de memorie alocată lui **i**.

În limbajul C++, parametri pot fi transferați în două moduri: prin valoarea (valoare directă sau adresă) și prin referință. În transferul prin valoare parametri actuali (specificați în momentul apelului) sunt copiați în zona de memorie rezervată pentru parametri formali (specificați în momentul definiției funcției). Orice modificare efectuată asupra parametrilor formali nu va implica și modificarea parametrilor actuali (de apel).

Modificarea parametrilor actuali poate fi realizată dacă în momentul apelului se transmite adresa de memorie a acestora. Astfel secvențele de instrucțiuni ale funcție pot modifica conținutul memoriei de la adresele transmise și implicit valorile parametrilor actuali (de apel).

În cazul transferului prin referință, funcției i se transmit nu valorile parametrilor actuali ci un alias al acestora. În acest fel secvența de instrucțiuni a funcției poate modifica valorile parametrilor actuali.

Domeniul de valabilitate și vizibilitate

Prin domeniu de valabilitate (vizibilitate) se înțelege zona de program în care este valabilă declararea unui identificator (variabilă, funcție). Astfel, toți identificatorii declarați într-un bloc (secvență de program delimitată de accolade) sau modul (fișier sursă) sunt cunoscuți blocului/modulului respectiv și se numesc variabile locale sau globale. Dacă în interiorul unui bloc se definește un alt bloc atunci variabilele locale ale blocului părinte devin variabile globale pentru blocul fiu iar variabilele locale blocului fiu nu au valabilitate în blocul părinte. Dacă în blocul fiu este declarat (redefinit) un identificator identic ca denumire cu unul din blocul părinte atunci se ia în considerare ultima declarație. Se poate face apel la declarația din blocul părinte prin utilizarea operatorului ::.

Functii cu parametri implicați

Este posibil să se apeleze o funcție cu un număr de parametri actuali mai mic decât numărul parametrilor formali. Parametrii care pot lipsi se numesc implicați (cu valori implicate); ei trebuie să se regăsească în extrema dreaptă a listei parametrilor formali din antetul funcției.

```
1. void f(int m=0, int n=1) {}  
2. f();           //f(0,1)  
3. f(1);         //f(1,1)  
4. f(1,2);  
  
5. void g(float x, int m=0, int n=1) {}  
6. g(2.5);      //g(2.5, 0,1)  
7. g(2.5, 1);   //g(2.5, 1,1)  
8. g(2.5, 1,2);
```

Dacă se lucrează cu funcții ce au parametri implicați, trebuie să se evite situațiile de ambiguitate ce pot apărea la supraîncărcarea funcțiilor.

```
void F1(void)  
{  
    cout << "Apel F1\n";  
}  
void F2(double re=0, double im=0)  
{  
    cout << "Apel F2\n";  
}
```

Supraîncărcarea funcțiilor

În C++ pot exista mai multe funcții cu același nume, dar cu liste diferite de argumente. Aceste funcții sunt **supraîncărcate**. Supraîncărcarea se utilizează pentru a da același nume tuturor funcțiilor care fac același tip de operație.

Numele unei funcții și ansamblul argumentelor sale, ca număr și tipuri, se numește **semnătura** acelei funcții. Se observă că funcțiile supraîncărcate au semnături diferite. Să luăm ca exemplu funcția care calculează valoarea absolută a unui număr care poate fi int, long sau double. În C_ANSI, pentru această operație există trei funcții:

```
int abs (int); // se calculeaza valoarea absoluta a unui intreg  
long labs (long); // se calculeaza valoarea absoluta a unui intreg lung  
double fabs (double); // se calculeaza valoarea absoluta a unui numar real
```

În C++ se folosește numele “abs” pentru toate cele trei cazuri, declarate astfel:

```
int abs (int); // functie de biblioteca  
long abs (long); // functie de biblioteca  
double abs (double); // functie utilizator
```

```
#include <iostream>  
using namespace std;  
// #include <stdlib.h>
```

```

#include <conio.h>

double abs(double x)
{
    return x < 0 ? -x : x;
}

int main()
{
    int a = -5;
    long b = -5L;
    float f = -3.3f;
    double d = -5.5;
    cout << "a=" << abs(a) << endl;
    cout << "b=" << abs(b) << endl;
    cout << "c=" << abs(f) << endl;
    cout << "c=" << abs(d) << endl;
    getch();
    return 0;
}

```

Compilatorul C++ selectează funcția corectă prin compararea tipurilor argumentelor din apel cu cele din declarație.

Când facem supraîncărcarea funcțiilor, trebuie să avem grijă ca numărul și/sau tipul argumentelor versiunilor supraîncărcate să fie diferite. Nu se pot face supraîncărcări dacă listele de argumente sunt identice:

```

int calcul (int);
double calcul (int);

```

O astfel de supraîncărcare (numai cu valoarea returnată diferită) este ambiguă. Compilatorul nu are posibilitatea să discearnă care variantă este corectă și semnalează eroare.

Tema de laborator:

-revizuirea aplicatiei din laboratorul 1:

Crearea unui catalog pentru mai multe grupe de studenți, permitând citirea componentelor, afisarea, sortarea conform unor criterii (alfabetic, crescator după lungimea numelui sau descrescator după nota), având în vedere alocarea dinamică de memorie și eliberarea acesteia când nu mai este necesară.

//exemplu de continut Catalog.h:

```

#pragma once //pentru a nu fi inclus de mai multe ori
/*
echivalent cu cele 3 linii de directive pentru preprocessare:
#ifndef _CATALOG_
#define _CATALOG_
//macrodefinițiile, declaratiile de tipuri noi de date și de functii
#endif

```

```

*/
//putem seta aici toate incluziunile, astfel incat in fisierul "main" si cel de implementare a functiilor sa
avem de scris o singura
//linie de forma #include "Catalog.h"
#include <iostream>
#include <stdlib.h> //sau se poate folosi cstdlib
#include <string.h>

using namespace std;

typedef struct _Student{
    private: //aceste campuri NU vor fi "vizibile" (accesibile) direct din afara structurii
        char *nume;
        int nota;
    public:
        int getNota(void); //functie accesator prin care se permite "vederea" notei luate de
studentul curent
        void setNota(int v); //functie mutator prin care se seteaza valoarea notei pentru
studentul curent
        char* getNume(void); //functie accesator prin care
            //se permite accesul la pointerul la nume (deja alocat!) pentru studentul curent
        void setNume(char unNume[]); //functie mutator prin care se aloca dinamic Z.M. si se va
copia continutul sirului
            //obs.: atentie la numararea caracterelor! Trebuie inclus SI terminatorul de sir
'\0'
        void elibMem(void); //eliberez Z.M. ocupata de nume si setam pe NULL pointerul
        void citire(void); //citim, apelam setarile...
        void afisare(void); //afisarea datelor - efectuata cu ajutorul functiilor accesator
}Student;

typedef int (*PFnComparare)(Student a, Student b);

typedef struct _Grupa{
    //DUPA realizarea sarcinilor de lucru din cadrul laboratorului se pot
    // "ascunde" campurile de tip date astfel incat accesul catre ele sa fie realizate prin metode
accesor/mutator
    int nrStud;
    Student *tabStudenti; //alocat dinamic in cadrul functiei de citire. eliberat de catre functia de
eliberare
    char* numeGrupa; //denumirea grupei: 1208B, 1207A etc.
    void citire(void); //DECLARATIA metodei de citire => PROTOTIPUL functiei
    void afisare(void); //DECLARATIA metodei de afisare => PROTOTIPUL functiei
    PFnComparare comparator; //pointer catre o functie de comparatie, definita global
        // acest pointer va fi setat de catre un "obiect" de tip structura Catalog, pentru fiecare
grupa in parte.
    void bSort(void); //nu mai e necesar pointerul la functia de comparatie ca parametru,
        //deoarece este camp al "obiectului" curent de tip grupa!
}

```

```

        void elibMem(void); //a se vedea implementarea!
} Grupa;

typedef struct _Catalog{
    int nrGrupe;
    Grupa *tabGrupe;
    void setComparator(PFnComparare comparator);//iteram fiecare grupa din tablou si facem
initializarea campului
    //cu parametrul formal primit
    void citire(void);//citim numarul de grupe, alocam dinamic tabloul, apoi apelam citirea pentru
fiecare grupa in parte
    void afisare(void);//afisam, pe rand, continutul fiecarei grupe. A se vedea functia de afisare din
structura Grupa!
    void sortare(void); //considerand acel comparator care a fost setat pentru fiecare grupa
    //se apeleaza bSort pentru fiecare grupa in parte
    void elibMemorie(void);//in care apelam elibMem pentru fiecare grupa in parte
} Catalog;

//declaratii de functii "globale":
int comparNumeAlfabetic(Student a, Student b);
int comparNoteDescrescator(Student a, Student b);
int comparNumeDupaLungimeCrescator(Student a, Student b);

//exemplu de continut fisier Catalog.cpp
//va trebui sa completati cu implementarile (definitiile) tuturor functiilor declarate si pe care va trebui sa
le utilizati
#include "Catalog.h"
    //exemplu de DEFINIRE a unei metode
void Catalog::setComparator(PFnComparare comparator)
{
    int i; //contor pentru a parcurge grupele
    for(i=0;i<nrGrupe;i++)
        tabGrupe[i].comparator=comparator;
    //setam pentru fiecare grupa in parte pointerul la functia de comparatie
}
void Grupa::elibMem(void)
{
    int s; //contor pentru parcurgerea studentilor din cadrul unei grupe
    if(numeGrupa)
    {
        cout<<"eliberarea memoriei pentru grupa "<<numeGrupa<<endl;
        free(numeGrupa);
        numeGrupa=NULL;
    }
    for(s=0;s<nrStud;s++) //nrStud este CAMP al structurii grupa
        tabStudenti[s].elibMem(); //eliboram Z.M. ocupata de numele fiecarui student
in parte
    free(tabStudenti);
}

```

```

        tabStudenti=NULL;
    }

int comparNumeDupaLungimeCrescator(Student a, Student b)
{
    int rez=strlen(a.nume)-strlen(b.nume);
    if(rez>0)
        rez=1;
    else
        if(rez<0)
            rez=-1;
    //daca e 0 ramane 0
    return rez;
}

//exemplu de continut L02Main.cpp
#include "Catalog.h"
int main(void)
{
    int operatie;
    PFnComparare unPointerLaOFunctieDeComparare;
    Catalog catalogulAnului1, catalogulAnului2, catalogulAnului3, catalogulAnului4;
    Catalog catalogMaster[2];

    catalogulAnului2.citire();
    catalogulAnului2.afisare();
    do{
        do{
            cout<< "Ce doriti sa efectuati?"<<endl;
            cout<< "0.iesire din program;"<<endl;
            cout<< "1 - sortare alfabetica a numelor;"<<endl;
            cout<< "2 - sortare descrescatoare dupa nota;"<<endl;
            cout<< "3 - sortare dupa lungimea numelui - crescator."<<endl;
            cin>>operatie;
        }while((operatie<0) || (operatie>3));
        switch(operatie)
        {
            case 1:
                cout<<"1 - sortare alfabetica a numelor;"<<endl;
                unPointerLaOFunctieDeComparare=comparNumeAlfabetic;
                //se putea face apelul si direct, dars-a dorit evidențierea atribuirii pentru o
variabila
                //de tip pointer la functie!
                catalogulAnului2.setComparator(unPointerLaOFunctieDeComparare);
                //se putea apela si direct
catalogulAnului2.setComparator(comparNumeAlfabetic);
                break;
            case 2:

```

```

cout<< "2 - sortare descrescatoare dupa nota;"<<endl;
unPointerLaOFuncieDeComparare=comparNoteDescrescator;
catalogulAnului2.setComparator(unPointerLaOFuncieDeComparare);
break;

case 3:
    cout<< "3 - sortare dupa lungimea numelui - crescator."<<endl;
    unPointerLaOFuncieDeComparare=comparNumeDupaLungimeCrescator;
    catalogulAnului2.setComparator(unPointerLaOFuncieDeComparare);
    break;

default:
    cout<< "Sfarsitul executiei programului."<<endl;
}

}while(operatie);
catalogulAnului2.elibMemorie();
//folositi si cataloagele celorlalti ani de studiu sau stergeti declaratiile variabilelor nefolosite in
program!

return EXIT_SUCCESS;
}
//de completat apelurile necesare in main!
//de completat declaratiile de functii

```

-Cerinte (scenariu):

- *.Completarea programului cu implementarile functiilor declarate, conform logicii programului!
- *.pentru implementarea functiei bSort (bubbleSort) - preluati logica implementarii de la laboratorul 1, faceti modificarea a.i. sa folositi comparatorul dat ca pointer si interschimbarea - realizati-o cu ajutorul unei functii "swap2" care are ca parametri doua referinte la structuri Student
- *.citirea numarului de grupe =>alocare
- *.citirea numarului de studenti pentru fiecare grupa in parte =>alocare tablou
- *.citirea datelor pentru fiecare student in parte
- *.afisarea datelor citite pana acum (pentru verificarea corectitudinii citirilor efectuate)
- *.citirea operatiei dorite:
 - 0 - iesire din program => apelul eliberarii Zonelor de Memorie care au fost alocate dinamic
 - 1 - sortare alfabetica a numelor (setare pointer la functia de comparatie corespunzatoare,)
 - 2 - sortare descrescatoare dupa nota (apoi apelul pentru fiecare grupa in parte a)
 - 3 - sortare dupa lungimea numelui - crescator (sortarii bubbleSort, avand criteriul de comparatie setat)

POO - C++ - Laborator 4

Cuprins

1.	Clase.....	1
2.	Obiecte.....	2
3.	Specificatori de acces.....	3
4.	Constructor	4
5.	Constructor de inițializare.....	5
6.	Supraîncărcarea constructorilor	6
7.	Constructorul implicit	6
8.	Destructor	10
9.	Exerciții.....	11

1. Clase

Clasa este o extensie a conceptului de structură din C. Prin crearea unei clase se definește, de fapt, un nou tip de dată care reprezintă concretizarea unui anumit concept. Ca și structura, o clasă poate avea mai mulți membri, care pot fi:

- variabile, care conțin datele clasei și care se numesc **proprietăți** sau **câmpuri**
- funcții, care se numesc **metode**

Declararea unei clase se realizează folosind cuvântul cheie **class** și are loc într-un fișier header. Declararea clasei presupune, de asemenea, și declararea variabilelor și a metodelor proprii ei. Definirea metodelor se face de cele mai multe ori separat, într-un fișier sursa, .cpp . Crearea unei clase permite integrarea atât a datelor care trebuie prelucrate, cât și a funcțiilor și operațiilor ce realizează prelucrarea.

În exemplul de mai jos s-a declarat o clasă care încorporează atât dimensiunile unui dreptunghi, cât și metodele care permit modificarea datelor și prelucrarea lor:

dreptunghi.h

```
class Dreptunghi
{
private:
    int lungime, latime;

public:
```

```
    void SetLungime(int lung);
    void SetLatime(int lat);
    int Aria();
};
```

Atenție! După declararea clasei (după acolada care închide clasa) se pune ; (punct și virgulă)!

Observăm că, deocamdată, metodele din cadrul clasei au fost doar declarate. Ele urmează să fie definite într-un fișier sursă unde prima dată trebuie inclus headerul în care s-au făcut declarațiile. Apoi, pentru metodele definite, trebuie specificată individual clasa din care fac parte, folosind operatorul de rezoluție ::

dreptunghi.cpp

```
#include "dreptunghi.h"

void Dreptunghi::SetLungime(int lung)
{
    lungime = lung;
}

void Dreptunghi::SetLatime(int lat)
{
    latime = lat;
}

int Dreptunghi::Aria()
{
    return lungime * latime;
}
```

2. Obiecte

Clasele astfel declarate constituie un tip de dată. Declararea unei variabile de tipul unei clase se numește **instantierea clasei**, iar variabila astfel declarată se numește **instanță** sau **obiect**.

main.cpp

```
int main()
{
    Dreptunghi d; // crearea statică a unui obiect

    Dreptunghi *p; // pointer la clasa
```

```

p = new Dreptunghi; // crearea unui obiect prin alocare dinamica
delete p; // eliberarea memoriei

Dreptunghi dv[10]; // un vector de obiecte alocat static

Dreptunghi *pv;
pv = new Dreptunghi[10]; // vector de obiecte alocat dinamic
delete[] pv; // eliberarea memoriei

return 0;
}

```

Accesul la membrii clasei se realizează la fel ca în cazul structurilor: se folosește operatorul . (punct) pentru obiectele definite static (variabilele obișnuite) și operatorul -> (săgeată) în cazul pointerelor:

```

Dreptunghi d;

d.SetLungime(2);
d.SetLatime(3);
cout << d.Aria();

Dreptunghi *p = new Dreptunghi;
p->SetLatime(4);

```

3. Specificatori de acces

În exemplele precedente apar cuvintele cheie **private** și **public**. Acestea se numesc **specificatori de acces** și, prin utilizarea lor, se decide gradul de accesibilitate al variabilelor și metodelor clasei. Specificatorii se aplică tuturor membrilor aflați sub ei, până la apariția unui nou specificator sau până la încheierea declarării clasei.

- **private** permite accesul la membrii clasei doar membrilor aceleiași clase.
- **public** permite accesul la membrii clasei de oriunde (membrii clasei, funcții ce nu aparțin clasei, funcția main())

În mod implicit, toți membrii unei clase au modul de acces *private*.

dreptunghi.h
<pre> class Dreptunghi { int lungime, latime; public: void SetLungime(int lung); </pre>

```

    void SetLatime(int lat);
    int Aria();
private:
    int Perimetru();
};
```

dreptunghi.cpp

```

#include "dreptunghi.h"

void Dreptunghi::SetLungime(int lung)
{
    lungime = lung;

    /*
    membrul lungime este implicit private
    si este accesibil aici, deoarece metoda SetLungime apartine clasei Dreptunghi
    */
}
```

main.cpp

```

int main()
{
    Dreptunghi d;

    cout << d.lungime; //eroare: campul lungime este implicit private, se incercă
                      //accesarea lui din afara clasei Dreptunghi

    cout << d.Aria(); //metoda este public, poate fi accesată din afara clasei

    int p = d.Perimetru(); // eroare: metoda este private, nu se poate utiliza în
                          // afara clasei unde a fost declarată

    return 0;
}
```

4. Constructor

Atunci când se construiesc, obiectele au nevoie să li se aloce memorie pentru membrii de tip dată și eventual aceștia să fie inițializați. Inițializarea face obiectele operabile și reduce posibilitatea returnării unor valori nedorite în timpul execuției. În lipsa inițializării, există posibilitatea de a fi luate valori aleatoare prezente în locațiile de memorie asignate câmpurilor.

Pentru a evita acest tip de situații, o clasă poate conține o metodă specială numită **constructor**.

Constructorul are următoarele caracteristici:

- numele este același cu al clasei;
- nu are tip, deci nu returnează nimic;

- este apelat automat la începerea duratei de viață a oricărui obiect de tipul respectiv.

Declarația constructorului are următoarea formă:

Nume_clasa(lista parametri);

Constructorul este apelat când se crează un obiect al clasei. Pentru obiectul deja creat constructorul nu mai poate fi reapelat, ca orice altă metodă obișnuită.

Există o serie de constructori tipici pentru o clasă:

- **constructorul implicit**: nu are nici un parametru și initializează membrii de tip dată cu valori implicate;
- **constructorul de inițializare**: are de obicei câte un parametru pentru fiecare membru de tip dată din clasă și îl initializează cu valorile primite ca parametri;
- **constructorul de copiere**, acesta având un regim special în cadrul clasei.

Observații:

- 1) **Obiectele globale** (declarate în exteriorul oricărei funcții) au mod de alocare static și sunt plasate în segmentul de date; durata lor de viață începe înainte de intrarea în funcția `main()`.
- 2) **Obiectele locale** funcțiilor au mod de alocare auto și sunt plasate în segmentul de stivă; durata lor de viață începe în momentul în care execuția programului ajunge în locul în care sunt declarate.
- 3) **Obiectele instanțiate dinamic** sunt plasate în zona de heap; durata lor de viață începe în momentul alocării dinamice.

5. Constructor de inițializare

Vom implementa clasa Dreptunghi din laboratorul precedent folosind un constructor de inițializare:

```
#include<conio.h>
#include<iostream>
using namespace std;

class Dreptunghi {
    int lungime, latime;
public:
    Dreptunghi (int, int);
    int arie () {return (lungime*latime);}
};

Dreptunghi::Dreptunghi(int L, int l) {
    lungime = L;
    latime = l;
}

int main () {
    Dreptunghi dr(10,20);
    cout << "arie: " << dr.arie() << endl;
    _getch();
    return 0;
}
```

Ieșire:

arie: 200

Observăm că rezultatul rulării acestui exemplu este identic cu cel din laboratorul anterior. De data aceasta am eliminat funcțiile membru `setLungime()` și `setLatime()` și le-am înlocuit cu un constructor care face același lucru: inițializează valorile câmpurilor `lungime` și `latime` cu parametrii pe care îi primește, adică `L` și `l`.

Remarcăm că acești parametri sunt transmiși către constructor în momentul creerii obiectului:

`Dreptunghi dr(10, 20);`

6. Supraîncărcarea constructorilor

Să ne amintim că **semnatura unei funcții** reprezintă numele funcției împreună cu lista de parametri. În C++, două sau mai multe funcții care au același nume dar semnături diferite se consideră că sunt supraîncărcate. **Supraîncărcarea** se utilizează pentru a da același nume tuturor funcțiilor care fac același tip de operație.

De exemplu, funcția `abs()` care returnează valoarea absolută, se va numi la fel, indiferent dacă data de intrare este de tip `int`, `long` sau `double`.

```
int abs(int);
long abs(long);
double abs (double);
```

Atât funcțiile globale cât și metodele pot fi supraîncărcate. Prin urmare, este admisă și supraîncărcarea constructorilor.

7. Constructorul implicit

Atunci când într-o clasă nu există nici un constructor, compilatorul generează automat un constructor special pentru acea clasă, numit **constructor implicit**. Rolul acestui constructor este de a aloca memorie pentru datele membre. Dacă obiectele construite cu acest constructor implicit sunt globale, inițializarea membrilor de tip dată se face cu:

- **0** dacă sunt numerice;
- '**\0**' dacă sunt de tip caracter sau sir de caractere;
- **NULL (0)** dacă sunt pointeri.

Dacă în clasă avem membri de tip referință, atunci compilatorul nu generează constructorul implicit, generând o eroare de compilare care ne atrage atenția că implementarea unui constructor care să inițializeze referințele este obligatorie.

Dacă obiectele sunt locale, constructorul implicit generat de compilator nu face nici o inițializare pentru câmpuri.

Să analizăm următorul exemplu:

```
#include<conio.h>
```

```

#include<iostream>
using namespace std;

class Numar
{
    int nr;
public:
    void setNr(int n) {
        nr = n;
    }
    int getNr() {
        return nr;
    }
};

Numar nrGl;
int main()
{
    Numar nrLoc;
    cout << "Obiect global "
        "initializat implicit: "
        << nrGl.getNr() << endl;

    cout << "Obiect local "
        "neinitializat implicit: "
        << nrLoc.getNr() << endl;

    nrLoc.setNr(100);
    cout <<"Obiect local initializat explicit"
        " cu metoda membru: "
        << nrLoc.getNr() << endl;

    _getch();
    return 0;
}

```

Ieșire

```

Obiect global initializat implicit: 0
Obiect local neinitializat implicit: -858993460
Obiect local initializat explicit cu metoda membru: 100

```

Observăm că deși nu există un constructor în clasă, se pot crea obiecte, cu ajutorul constructorului implicit generat de compilator.

În acest caz, putem crea obiecte numai folosind următoarea sintaxă:

```

/*nu avem lista de initializare pentru obiect*/
Numar nrLoc;

```

Atunci când un obiect este initializat folosind constructorul implicit, trebuie folosită declarația fără paranteze. De exemplu:

```

Numar nr1; // declaratie corecta
Numar nr2(); // declaratie gresita

```

Constructorul implicit poate fi implementat și de către programator, acesta putând să facă alocări și / sau inițializări pentru membrii de tip dată. Dacă programatorul implementează un constructor pentru o clasă, compilatorul nu mai generează constructorul implicit.

Dacă într-o clasă, programatorul implementează un **constructor fără parametri**, acesta se numește tot **constructor implicit**.

```
class Numar
{
    int nr;
public:
    Numar() {
        // constructor implicit vid
    }

    void setNr(int n) {
        nr = n;
    }
};
```

Să analizăm următorul exemplu și să observăm modul în care se inițializează câmpurile obiectelor instantiate.

```
#include<conio.h>
#include<iostream>
using namespace std;

class Numar
{
    int nr;
public:

    Numar() {
        //constructor implicit cu initializare
        nr = 10;
    }
    void setNr(int n) {
        nr = n;
    }
    int getNr() {
        return nr;
    }
};

Numar nrGl;

int main()
{
    Numar nrLoc;
    cout << nrGl.getNr() << endl;
    cout << nrLoc.getNr() << endl;

    nrLoc.setNr(100);
    cout << nrLoc.getNr() << endl;
```

```
    _getch();
    return 0;
}
```

Ieșire

```
10
10
100
```

Un caz de ambiguitate care poate să apară la implementarea constructorului implicit este prezentat în următorul exemplu:

```
#include<conio.h>
#include<iostream>
using namespace std;

class Numar
{
    int nr;
public:
    //constructor implicit vid
    Numar() {}

    //constructor cu parametru predefinit
    Numar(int n=0) {
        nr = n;
    }
    void setNr(int n) {
        nr = n;
    }
};

int main()
{
    Numar nr;
    _getch();
    return 0;
}
```

La crearea obiectului nr, ambiguitatea apare din cauză că, în această situație, compilatorul poate apela ambi constructori care au fost definiți. Pentru rezolvarea ambiguității este recomandată eliminarea constructorului implicit, deoarece constructorul de inițializare cu parametru predefinit poate fi utilizat ca și constructorul implicit.

Să studiem următorul exemplu care utilizează supraîncărcarea constructorilor:

```
// supraincarcarea constructorilor
#include<iostream>
#include<conio.h>
using namespace std;

class Dreptunghi {
    int lungime, latime;
public:
```

```

//constructor implicit
Dreptunghi ();

//constructor cu parametri
Dreptunghi(int,int);
int arie() {return lungime * latime;}
};

Dreptunghi::Dreptunghi () {
    lungime = 10; latime = 10;
}

Dreptunghi::Dreptunghi(int L, int l) {
    lungime = L;
    latime = l;
}

int main () {
    Dreptunghi dr1 (10,20);
    Dreptunghi dr2;
    cout << "arie dr1: "
        << dr1.arie() << endl;
    cout << "arie dr2: "
        << dr2.arie() << endl;
    _getch();
    return 0;
}

```

Ieșire

```

arie dr1: 200
arie dr2: 100

```

8. Destructor

Destructorul unei clase este complementarul constructorului și are următoarele caracteristici:

- numele este același cu al clasei, precedat de caracterul '~';
- nu are tip, deci nu returnează nimic și nu are argumente, deci nu poate fi supraîncărcat;
- este apelat automat la terminarea duratei de viață a obiectelor nealocate dinamic. Pentru obiectele alocate dinamic apelarea destructorului trebuie să se facă explicit;
- trebuie să dezaloce zonele de memorie alocate dinamic pentru membrii de tip dată.

Declarația destructorului are următoarea formă:

<code>~Nume_clasa();</code>

Observații:

Există situații când nu este necesară implementarea destructorului:

- atunci când în clasă nu există date membre alocate dinamic, ele având tipuri fundamentale. Dacă nu se implementează destructorul, compilatorul va genera un **destructor implicit**, pe care îl apelează când se încheie durata de viață a obiectului.

- atunci când în clasă există tipuri abstracte de date (instantieri de structuri sau clase). Dacă clasa obiectului imbricat are un destructor explicit, compilatorul îl apelează pentru distrugerea acestui obiect. În caz contrar, compilatorul apelează destructorul implicit din clasa obiectului imbricat.

```
#include<conio.h>
#include<iostream>
using namespace std;

class Numar
{
    int *nr;
public:

    Numar() {
        nr = new int;
    }
    ~Numar() {
        cout << "~Numar()" << endl;
        delete nr;
        _getch();
    }
    void setNr(int n) {
        *nr = n;
    }
};

class Intreg{
    Numar n;
public:
    Intreg(int nr){
        n.setNr(nr);
    }
};

int main() {
    Intreg i(10);
    return 0;
}
```

Ieșire

~Numar()

Durata de viață a obiectelor se termină:

- pentru obiectele globale cu mod de alocare static, la ieșirea din funcția `main()`;
- pentru obiecte locale cu mod de alocare auto, la ieșirea din blocul în care au fost declarate.
- pentru obiectele cu alocare dinamică, la apelarea explicită a operatorului `delete`.

9. Exerciții

1. Să se implementeze clasa `Multime`, care reprezintă o mulțime de întregi.

Clasa va conține următoarele câmpuri private:

- `int *date` – vectorul de numere. Se va aloca în constructor și se va dezaloca în destructor;
- `int dim` – dimensiunea vectorului `date`; totodată reprezintă numărul maxim de elemente din mulțime;
- `int n` – numărul curent de elemente din mulțime; în orice moment de timp, elementele mulțimii vor fi primele `n` elemente din vectorul `date`.

Pe parcursul existenței mulțimii, numărul `n` și elementele din `date` se pot modifica, dar câmpul `dim` rămâne neschimbăt.

Membrii publici:

- constructorul implicit care initializează câmpurile private ale mulțimii; dimensiunea maximă a mulțimii va fi o valoare oarecare prestabilită;
- constructorul cu un parametru pentru câmpul `dim`, reprezentând dimensiunea maximă a mulțimii;
- destructorul care va elibera memoria alocată dinamic;
- metoda `void adauga(int)` care adaugă un element în mulțime; în cazul în care elementul deja există, mulțimea rămâne nemodificată; în cazul în care vectorul `date` este plin, se va afișa un mesaj de eroare;
- metoda `void extrage(int)` care extrage un element din mulțime; în cazul în care mulțimea nu conține elementul, ea rămâne neschimbătă;
- metoda `void afisare()` care afișează mulțimea.

Folosiți următorul program pentru testarea mulțimii:

```
int main() {
    Multime m(10);
    m.adauga(4);
    m.adauga(3);
    m.afisare();
    m.extrage(4);
    m.extrage(4);
    m.afisare();
    m.adauga(9);
    m.adauga(2);
    m.adauga(2);
    m.afisare();
    _getch();
    return 0;
}
```

2. Să se implementeze clasa `Stiva` având următoarele metode:

- constructorul implicit care initializează câmpurile private ale stivei;
- metoda `push` care adaugă un element în stivă;
- metoda `pop` care extrage un element din stivă;
- metoda `top` care returnează vârful stivei, fără să îl extragă;

- metoda `print` care afișează stiva.

Folosiți următorul program pentru testarea stivei:

```
int main() {
    Stiva s;
    s.push(4);
    s.push(3);
    cout << s.top() << endl;
    s.push(9);
    cout << s.pop() << endl;
    s.push(2);
    s.print();
    _getch();
    return 0;
}
```

POO - C++ - Laborator 5

Cuprins

1.	Membri statici ai claselor	1
2.	Cuvântul cheie <i>this</i>	2
3.	Constructorul de copiere	4
4.	Constructorul explicit de copiere.....	6
5.	Pointeri la clase	7
6.	Exerciții.....	8

1. Membri statici ai claselor

Folosind cuvântul cheie **static** unii membri ai clasei pot fi declarați de tip static. Membrii de tip dată declarați statici au următoarele proprietăți:

- există într-un singur exemplar indiferent de numărul de instanțieri ale clasei; ca o consecință, dimensiunea datelor membre statice nu participă la dimensiunea unei instanțieri a clasei;
- pot fi referiți prin numele clasei urmat de operatorul de rezoluție `::` și de numele membrului dată static;

Metodele declarate static pot fi referite similar chiar în cazul în care nu există instanțieri ale clasei.

```
#include<iostream>
#include<string.h>
#include<conio.h>
using namespace std;

class Persoana {
    char *nume;
    static unsigned nrPersoane;
public:
    Persoana(char *nume) {
        if(nume != NULL) {
            this->nume =
                new char[strlen(nume)+1];
            strcpy(this->nume, nume);

        } else {
            this->nume= NULL;
        }
        nrPersoane++;
    };
    ~Persoana() {
```

```

        if (nume != NULL) {
            delete[] nume;
        }
        cout<<"~Persoana()" << endl;
        nrPersoane--;
        cout << "Numar persoane ramase = "
            << nrPersoane << endl;
        _getch();
    }
    static void printNrPers() {
        cout <<"nrPers="
            << nrPersoane << endl;
    };
};

//initializare membru static
unsigned Persoana::nrPersoane = 0;

int main() {
    Persoana ionescu("ionescu");
    Persoana popescu("popescu");
    Persoana *simionescu
        = new Persoana("simionescu");
    Persoana::printNrPers();
    delete simionescu;
    return 0;
}

```

Ieșire

```

nrPers=3
~Persoana()
Numar persoane ramase = 2
~Persoana()
Numar persoane ramase = 1
~Persoana()
Numar persoane ramase = 0

```

2. Cuvântul cheie *this*

Cuvântul cheie *this* este o variabilă locală predefinită în C++, accesibilă în corpul oricărei **metode non-static** a unui obiect și care nu trebuie declarată. Această variabilă este de tip pointer la obiectul curent (obiectul a cărui metodă este executată) și conține adresa acestui obiect. Situația se prezintă ca și cum metodei i s-ar transmită implicit adresa instanțierii prin care a fost apelată.

Considerăm o clasă *X* și o instanțiere *x* a acestei clase. Există două posibilități de utilizare a variabilei predefinite *this*. Acestea sunt exemplificate prin *func1()* și *func2()* în exemplul următor:

- când se face apelul metodei *x.func1()*, lui *this* î se dă valoarea adresei lui *x* (*&x*) și poate fi folosit în corpul funcției *func1*, după care se poate returna ca și pointer.
- când se face apelul metodei *x.func2()*, se returnează conținutul pointerului *this*.

```

#include <conio.h>
#include<iostream>
using namespace std;

```

```

class X
{
public:
    X* func1()
    {
        cout<<"Test1 pentru this."<<endl;
        return this;
    }

    X& func2()
    {
        cout<<"Test2 pentru this."<<endl;
        return *this;
    }
};

int main()
{
    X x;
    X *x1 = x.func1();
    X& x2 = x.func2();
    x1->func1();
    x2.func2();
    _getch();
    return 0;
}

```

Ieșire

```

Test1 pentru this.
Test2 pentru this.
Test1 pentru this.
Test2 pentru this

```

Orice metodă non-statică a unei clase are ca prim parametru variabila `this` transmisă implicit.

În cadrul unei metode (mai ales în constructor), cuvântul `this` este util pentru a deosebi câmpurile clasei de parametrii cu același nume. Să urmărim următorul exemplu:

```

// exemplu: cuvantul cheie this
#include<iostream>
using namespace std;

class Complex {
    int re, im;
public:
    Complex(){re = im = 0;}
    Complex(int re, int im) {
        this->re = re;
        this->im = im;
    }
    void afisare() {
        cout << "("<<re<<","<<im<<") "<<endl;
    }
};
int main() {
    Complex c1(5,3);
    Complex c2(2,-3);
}

```

```
    c1.afisare();
    c2.afisare();
    return 0;
}
```

Ieșire:

```
(5, 3)
(2, -3)
```

Observăm parametrii constructorului, având același nume ca și câmpurile clasei:

```
Complex(int re, int im) {
    this->re /*campul clasei*/
        = re /*parametrul*/;
    this->im = im;
}
```

În acest caz variabilele locale `re` și `im` ascund câmpurile clasei. Totuși, câmpurile pot fi accesate fără probleme folosind cuvântul cheie `this`. Utilizarea acestei tehnici oferă o mai bună claritate a codului. Dispără necesitatea de a da nume diferite pentru parametrii constructorilor și ai metodelor de setare a câmpurilor. Tehnica se folosește pe larg în limbajele C++, Java și C#.

3. Constructorul de copiere

Constructorul de copiere este un constructor special, folosit pentru a crea un nou obiect, copie a unui obiect existent. Acest constructor are un singur argument – o referință către obiectul ce va fi copiat. Dacă în clasă nu există constructorul de copiere scris de programator, compilatorul generează implicit un constructor de copiere. Forma generală a constructorului de copiere este următoarea:

```
NumeClasa::NumeClasa(const NumeClasa &
obiectSursa);
```

Constructorul de copiere implicit copie fiecare membru a obiectului parametru în membrul corespunzător al obiectului de inițializat. De exemplu, pentru clasa `Complex` din exemplele noastre, compilatorul va genera următorul constructor de copiere:

```
Complex(const Complex &obiectSursa) {
    this->re = obiectSursa.re;
    this->im = obiectSursa.im;
}
```

Constructorul de copiere are un rol special în C++. El este apelat automat în următoarele situații:

1. La declararea unui obiect, inițializat dintr-un alt obiect. Exemplu:

```

Complex x(3,4); /* constructorul este folosit pentru a
crea obiectul x */
Complex y(x); /* constructorul de copiere este folosit
pentru a crea obiectul y*/
Complex z = x; /* constructorul de copiere este folosit
pentru initializare in */
// declaratie
z = x; /* Operatorul de atribuire (=), nu se
apeleaza constructori */

```

2. O funcție primește ca parametru un obiect transmis prin valoare.

3. O funcție returnează un obiect.

Să urmărim programul de mai jos:

```

#include<iostream>
using namespace std;
class Complex {
    int re, im;
public:
    Complex(){}
    Complex(int re, int im) {
        this->re = re;
        this->im = im;
    }

    Complex aduna(Complex c2) {
        Complex rez;
        rez.re = this->re+c2.re;
        rez.im = this->im+c2.im;
        return rez;
    }

    void afisare() {
        cout<<"(" << re << ", " << im << ")" << endl;
    }
};

int main() {
    Complex c1(5,3);
    Complex c2(2,-3);
    Complex c3;
    c3 = c1.aduna(c2);
    c1.afisare();
    c2.afisare();
    c3.afisare();
    return 0;
}

```

La apelul funcției `aduna()`, constructorul de copiere este apelat de 2 ori. O dată la transmiterea parametrului `c2`, și a doua oară la returnarea rezultatului `rez`.

În total, există 3 membri generați de compilator în mod implicit:

1. Constructorul implicit, în cazul în care nu este definit nici un alt constructor;
2. Constructorul de copiere;
3. Operatorul de atribuire.

Dacă programatorul are nevoie ca oricare dintre acești trei membri să fie definit altfel decât în modul implicit, îl poate defini în forma dorită.

4. Constructorul explicit de copiere

Un caz special este atunci când clasa conține câmpuri de tip pointer. În acest caz utilizatorul trebuie să definească constructorul de copiere în mod explicit. Constructorul de copiere explicit va aloca memoria necesară pentru câmpuri de tip pointer, și va inițializa memoria alocată.

Revenim la exemplul clasei Persoana:

```
#include<iostream>
#include<string.h>
using namespace std;
#pragma warning(disable : 4996)

class Persoana {
    char *nume;
public:
    Persoana(char *nume) {
        this->nume =
            new char[strlen(nume)+1];
        strcpy(this->nume, nume);
    };
    Persoana(const Persoana &p) {
        nume = new char[strlen(p.nume)+1];
        strcpy(nume, p.nume);
        cout<<"Constructor de copiere: "
            << nume<< endl;
    }
    ~Persoana() {
        if (nume != NULL) {
            delete[] nume;
        }
        cout<<"~Persoana()" << endl;
    }
};

void main() {
    Persoana ionescu("ionescu");
    Persoana popescu("popescu");
    Persoana popescu2=popescu;
}
```

Ieșire:

```
Constructor de copiere: popescu
~Persoana()
~Persoana()
~Persoana()
```

În constructorul de copiere se alocă memorie pentru câmpul nume, după care se copiează sirul de caractere din obiectul argument.

5. Pointeri la clase

Se pot crea și pointeri pe obiectele instanțiate din clase.

De exemplu:

```
Complex *c1, *c2;
```

c1 și c2 sunt pointeri la tipul Complex.

În vorbirea curentă, datorită faptului că obiectele pointate sunt instanțieri ale unei clase se folosesc și exprimarea „pointer la clasă”.

Putem folosi operatorul special săgeată (`->`) pentru a accesa membrul unui obiect referit de un pointer. Iată un exemplu cu câteva combinații posibile:

```
// exemplu cu pointeri la clase
#include<iostream>
using namespace std;
class Complex {
    int re, im;
public:
    Complex(int re, int im) {
        this->re = re;
        this->im = im;
    }
    void afisare() {
        cout<<"(" << re << ", " << im << ")" << endl;
    }
};

int main () {
    Complex a(1,2);
    Complex *b, *c;
    Complex **d = new Complex*[2];

    b = &a;
    c = new Complex(3,4);
    d[0] = new Complex(5,6);
    d[1] = new Complex(5,6);
    a.afisare();
    b->afisare();
    c->afisare();
    d[0]->afisare();
    d[1]->afisare();
    delete d[0];
    delete d[1];
    delete[] d;
    delete b;
    return 0;
}
```

Ieșire:

```
(1,2)
(1,2)
```

(3, 4)
(5, 6)
(5, 6)

Mai jos sunt sumarizate operațiilor posibile cu pointeri la clase:

Expresie	Semnificație
<code>*x</code>	obiect referit de x
<code>&x</code>	adresa lui x
<code>x.y</code>	membrul y al obiectului x
<code>x->y</code>	membrul y al obiectului referit de x
<code>(*x).y</code>	membrul y al obiectului referit de x (echivalent cu expresia anterioară)
<code>x[0]</code>	primul obiect din vectorul referit de x (echivalent cu <code>*x</code>)
<code>x[1]</code>	al 2-lea obiect din vectorul referit de x
<code>x[n]</code>	al (n+1)-lea obiect din vectorul referit de x

6. Exerciții

1. Să se implementeze o clasă `Complex`, similar cu exemplele din laborator. La această clasă să se adauge o metodă `egal()`, care va realiza compararea a 2 numere complexe. Metoda va avea următorul prototip:

```
int Complex::egal(Complex c2);
```

Metoda va compara complexele `this` și `c2` și va returna:

1, dacă `this == c2`

0, în caz contrar

Să se scrie o metodă `Complex::citire()`, care va citi numărul complex de la tastatură.

Scriți un program care citește de la tastatură 2 numere complexe și afișează rezultatul comparării lor.

2. Să se implementeze clasa `MultimeComplex`, care să păstreze o mulțime de numere complexe, reprezentate de clasa `Complex`.

Clasa va avea următoarele câmpuri private:

`Complex *v` - un vector de elemente `Complex`, care va păstra elementele mulțimii.

`int dim` – numărul de elemente alocate în vectorul `v`, dimensiunea maximă a mulțimii.

`int n` – numărul de elemente a mulțimii.

Implementarea este similară cu cea a mulțimii de întregi din lab. 2, doar că elementele stocate de mulțime vor fi de tip `Complex` în loc de `int`. Pentru compararea a 2 numere complexe în scopul determinării dacă sunt egale sau diferite, se va folosi funcția `egal()` de la problema 1.

Următoarele metode publice sunt similare cu cele ale mulțimii din lab. 2:

- constructorii, care inițializează câmpurile private ale mulțimii;

- `void adauga(Complex)` care adaugă un element în mulțime. În cazul în care elementul deja există, mulțimea rămâne nemodificată;
- `void extrage(Complex)` care extrage un element din mulțime. În cazul în care elementul nu este prezent, mulțimea rămâne neschimbată;
- `void afisare()` care afișează mulțimea.

Folosiți următorul program pentru a testa mulțimea:

```
int main() {
    MultimeComplexe m;
    Complex c1(2,3), c2(3,4), c3(2,-1);
    m.init();
    m.adauga(c1);
    m.adauga(c2);
    m.afisare();
    m.extrage(c1);
    m.extrage(c3);
    m.afisare();
    m.adauga(c3);
    m.adauga(c3);
    m.afisare();
    return 0;
}
```

3. Scrieți un program care testează clasa `MultimeComplexe` cu numere citite de la consolă. Programul va rula în buclă și va afișa la fiecare iterare un meniu, și va cere utilizatorului să aleagă una din următoarele operații:

- 1 – adăugare element
- 2 – extragere element
- 0 – ieșire din program.

În cazul în care se alege 1 sau 2, urmează citirea numărului complex de la tastatură, realizarea operației, și afișarea mulțimii rezultate.

POO - C++ - Laborator 6

Cuprins

1.	Funcții prietene. Clase prietene.....	1
2.	Supraîncărcarea operatorilor.....	2
3.	Supraîncărcarea operatorilor ++ și —.....	5
2.	Supraîncărcarea operatorului de atribuire (=).....	7
3.	Exerciții.....	8

1. Funcții prietene. Clase prietene

În C++ putem accesa membrii privați sau protejați ai unui obiect folosind funcții globale sau metode aparținând altor clase. Condiția este ca aceste funcții, respectiv clase, să fie declarate **prietene** cu clasele ale căror membri privați sau protejați le accesează. În acest scop se utilizează cuvântul cheie **friend**.

Relația de prietenie se poate declara între :

- O funcție globală și o clasă ;
- O funcție membră a unei clase și o altă clasă ;
- Între 2 clase diferite.

Considerăm următorul exemplu în care am exemplificat relația (a) și (c):

```
class Punct {  
    int x, y;  
public:  
    Punct(int x1, int y1) {  
        x = x1;  
        y = y1;  
    };  
    friend void f(void); // functie prietena  
    friend class X; // clasa prietena  
};  
void f() {  
    Punct a(2,0);  
    a.y = 9; /* acces la un membru privat al  
obiectului*/  
}  
class X {  
    Punct p;  
public:  
    void g() {
```

```
        p.x++; // acces la membru privat  
    }  
};
```

În următorul exemplu prezentăm relația de prietenie dintre metoda unei clase și o altă clasă:

```
class Y {  
public:  
    void f();  
};  
  
class X {  
private:  
    int a, b;  
    friend void Y::f();  
};  
  
void Y::f() {  
    X x;  
    x.a = 1;  
}
```

Notă: Relația de prietenie între clase nu este simetrică. Dacă A este prietena lui B, aceasta nu implică direct că B este prietena lui A. Dacă dorim, putem să declarăm 2 clase să fie reciproc prietene – A prietenă cu B și B prietenă cu A. Relația de prietenie nu este nici tranzitivă. Dacă A este prietenă cu B, iar B prietenă cu C, aceasta nu înseamnă în mod automat că A este prietenă cu C.

2. Supraîncărcarea operatorilor

Limbajul C++ permite ca acțiunea operatorilor să fie redefinită pentru noi tipuri de date. De exemplu putem defini clasa matrice, și operatorii + și * , care să efectueze adunarea și produsul a 2 matrici. Codul care ar face adunarea între 2 matrici ar arăta exact ca și adunarea între 2 numere:

```
Matrice a,b,suma,produs;  
  
//... inițializare a și b  
  
suma = a + b;  
produs = a * b;
```

Pentru a supraîncărca un operator care să poată fi aplicat obiectelor, trebuie să definim **funcțiile operator**, care pot avea domeniu local (declarate în cadrul clasei) sau global (declarate și definite în afara clasei). Funcțiile operator sunt funcții a căror nume este format din cuvântul cheie `operator` urmat de simbolul operatorului. De exemplu `tip operator+(...)`.

În continuare, vom dezvolta clasa `Complex` din laboratoarele anterioare. Vom adăuga operatorul de adunare (+), supraîncărcat pentru numere complexe.

```
//supraincarcare operator +  
#include<iostream>
```

```

using namespace std;
class Complex {
    int re,im;
public:
    Complex () {};
    Complex (int,int);
    Complex operator + (Complex);
    void afisare();
};

Complex::Complex (int re, int im) {
    this->re = re;
    this->im = im;
}

Complex Complex::operator+(Complex c2) {
    Complex temp;
    temp.re = this->re + c2.re;
    temp.im = this->im + c2.im;
    return temp;
}

void Complex::afisare() {
    cout << "(" << re << "," << im << ")" << endl;
}

int main () {
    Complex a(3,1);
    Complex b(1,2);
    Complex c;
    c = a + b;
    c.afisare();
    return 0;
}

```

Iesire:

(4,3)

Metoda `operator+` din clasa `Complex` realizează supraîncărcarea operatorului de adunare (`+`). Observăm că metoda are ca parametru un singur număr complex, deși realizează adunarea a 2 numere complexe. Acest lucru se datorează faptului că primul parametru al metodei este implicit și este pointerul `this`, fiind și primul operand din operația de adunare.

Această metodă poate fi apelată atât implicit folosind simbolul `+`, cât și explicit, folosind numele funcției:

<code>c = a + b;</code>
<code>c = a.operator+ (b);</code>

Cele două expresii sunt echivalente.

Funcțiile operator pot fi definite atât ca metode membre (ca în exemplul de mai sus) cât și sub formă de funcții globale. În cazul în care sunt definite ca metode, operandul din stânga va fi chiar `this`, iar operandul din dreapta va fi transmis ca parametru. În cazul în care funcția operator este globală, aceasta va avea 2 parametri reprezentând cei doi operanzi.

Pentru a putea accesa membrii privați ai clasei care a instanțiat obiectele parametri avem 2 posibilități:

1. Declarăm funcția globală prietenă a clasei a cărei instanță sunt obiectele parametru.
2. Să creăm metode prin care să citim valorile câmpurilor private.

De exemplu, pentru clasa `Complex`, putem supraîncărca operatorul `+`, ca funcție globală, și prietenă a clasei `Complex`:

```
class Complex {  
    int re,im;  
public:  
    Complex () {};  
    Complex (int,int);  
    void afisare();  
    friend Complex operator+(Complex, Complex);  
};  
  
//... Definirea celorlalți membri  
Complex operator+(Complex c1, Complex c2) {  
    Complex temp;  
    temp.re = c1.re+c2.re;  
    temp.im = c1.im + c2.im;  
    return temp;  
}  
  
int main () {  
    Complex a(3,1);  
    Complex b(1,2);  
    Complex c;  
    c = a + b;  
    c.afisare();  
    return 0;  
}
```

În locul funcțiilor prietene putem să ne garantăm accesul la membrii privați, definind două metode pentru citirea câmpurilor private.

În general, parametrul funcției operator poate fi de orice tip. De exemplu, putem să supraîncărçăm din nou operatorul `+`, care va avea parametru un număr întreg. Funcția va aduna numărul întreg atât la partea reală cât și la partea imaginară. și valoarea returnată de o funcție operator poate fi de orice tip, `char` și `void`. Similar cu operatorul `+` poate fi implementat orice alt operator binar.

și operatorii unari, gen `++`, pot fi supraîncărcați.

În tabelul de mai jos este prezentat modul cum pot fi declarate diverse funcții operator (înlocuiți @ cu operatorul în fiecare caz):

Expresie	Operator	Metodă	Funcție globală
@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)	-

Pe baza tabelului de mai sus, putem implementa operatorul unar !, care va realiza afișarea numărului complex:

```
void Complex::operator!() {
    cout << "(" << re << "," << im << ")" << endl;
}
```

Operatorul poate fi utilizat astfel:

```
Complex a (3,1);
!a;
```

Echivalentul operatorului ! implementat ca funcție globală are următorul prototip:

```
void operator! (Complex c1);
```

3. Supraîncărcarea operatorilor ++ și --

Operatorii unari ++ și -- au particularitatea că se pot utiliza în două moduri:

1. Ca prefix, de exemplu ++obiect, caz în care întâi se face incrementarea cu unu, apoi se utilizează obiectul;
2. Ca sufix, de exemplu obiect++, caz în care întâi se utilizează obiectul iar apoi se face incrementarea cu unu.

Implicit, supraîncărcarea operatorilor unari ++ și -- se face pentru forma prefixată. Să exemplificăm pentru operatorul ++, considerând o clasă Numar:

```
#include<iostream>
using namespace std;
class Numar {
    int nr;
public:
    Numar() { nr = 0; }
    Numar &operator++() {
        ++nr;
        return *this;
    }
}
```

```

        void afisare() {
            cout<<nr<<endl;
        }
    };

    int main() {
        Numar nr;
        ++nr;
        nr.afisare();
        return 0;
    }
}

```

Ieșire:

1

Dar dacă se dorește implementarea operatorului `++` ca sufix, vom proceda astfel:

```

#include<iostream>
using namespace std;
class Numar {
    int nr;
public:
    Numar() { nr = 0; }
    Numar &operator++(int a) {
        nr++;
        return *this;
    }
    void afisare() {
        cout<<nr<<endl;
    }
};

int main() {
    Numar nr;
    nr++;
    nr.afisare();
    return 0;
}

```

Ieșire:

1

Se observă că funcția operator are un parametru de tip întreg. Acest parametru nu este folosit. El servește pentru a face diferența între funcția `operator++()` prefixat, și funcția `operator++()` postfixat.

Funcțiile `operator++` prefixat, și postfixat, supraîncărcate ca funcții globale, mai au un parametru în plus. Prototipurile sunt următoarele:

<code>Numar &operator++(Numar &nr);</code>
<code>Numar &operator++(Numar &nr, int a);</code>

2. Supraîncărcarea operatorului de atribuire (=)

Un operator special este cel de atribuire (=), care se apelează ori de câte ori se întâlnește o expresie de tipul $a = b$. Acesta este singurul operator definit de compilator implicit. Implementarea implicită copiează valoarea fiecărui câmp al operandului din dreapta în câmpul corespunzător al obiectului operand stânga. Forma generală a operatorului = este:

```
X &operator=(const X&); //X - numele clasei
```

Pentru clasa Complex de mai sus, compilatorul generează următoarea **implementare implicită**:

```
Complex &operator=(const Complex &sursa) {
    this->re = sursa.re;
    this->im = sursa.im;
    return *this;
}
```

Operatorul = returnează o referință la obiect pentru a permite atribuirii înlănțuite ($a = b = c = d$).

În continuare prezentăm un exemplu de cod în care se apelează operatorul de atribuire:

```
Complex d(2,3);
Complex e;
e = d;// apelare operator de atribuire
```

Supraîncărcarea de către programator a operatorului = este absolut necesară atunci când clasa care instanțiază obiectul **conține câmpuri de tip pointer**. În supraîncărcarea operatorului egal se vor copia zonele de memorie referite de pointeri, și în caz de nevoie, se va face reallocarea.

Prezentăm următorul exemplu:

```
#include<iostream>
#include<string.h>
using namespace std;
#pragma warning(disable : 4996)

class Persoana {
    char *nume;
public:
    Persoana(char *nume) {
        this->nume = new char[strlen(nume)+1];
        strcpy(this->nume, nume);
    };
    Persoana(const Persoana &p) {
        nume = new char[strlen(p.nume)+1];
        strcpy(nume, p.nume);
        cout<<"Constructor de copiere: "
            << nume<< endl;
    }
    Persoana &operator=(const Persoana &p) {
        if (nume != NULL) {
```

```

        delete[] nume;
    }
    nume = new char[strlen(p.nume)+1];
    strcpy(nume, p.nume);
    cout<<"Operatorul= " << nume<< endl;
    return *this;
}
~Persoana() {
    if (nume != NULL) {
        delete[] nume;
    }
    cout<<"~Persoana()" << endl;
}
};

void main() {
    Persoana popescu("popescu");
    Persoana pop("pop");
    Persoana pop2=pop;
    pop2 = popescu;
}

```

Ieșire:

```

Constructor de copiere: popescu
Operatorul= popescu
~Persoana()
~Persoana()
~Persoana()

```

3. Exerciții

1. Completăți clasa Complex din laborator cu următorii operatori:

1.1. –

1.2. *

1.3. ==

1.4. ~ - modulul numărului complex

Scrieți un program care citește de la tastatură 2 numere complexe, și returnează rezultatul celor 4 operatori aplicați asupra numerelor.

2. Completăți clasa Multime din laboratorul 3, problema 1, cu următorii membri:

- Operatorul “+=” cu parametru int – adaugă un element la mulțime, echivalent cu funcția `Multime::adauga()`.
- Operatorul “-=” cu parametru int – extrage un element din mulțime, echivalent cu funcția `Multime::extrage()`.
- Operatorul “=” . Atenție la datele alocate dinamic.
- Constructorul de copiere.
- Operatorul “+=” cu parametru `Multime`. Va adăuga la mulțimea curentă elementele mulțimii primite ca parametru. Practic, după această operație mulțimea curentă va deveni reuniunea dintre cele 2 mulțimi operanzi. Va fi utilizat într-o expresie de genul `a+=b`

- Operatorul “+” cu parametru Multime. Va realiza reuniunea dintre cele 2 mulțimi operanzi. Spre deosebire de operatorul “+=” , mulțimea reuniune va fi un obiect nou, returnat de funcția operator. Mulțimile operanzi nu vor fi modificate. Se va utiliza într-o expresie de genul $a=b+c$ Realizați un program care testează toți acești operatori.
3. Implementați o clasă `String` care să reprezinte un sir de caractere și operațiile aferente. Definiți următorii membri:
- 3.1. Operatorul `+`, concatenarea sirurilor.
 - 3.2. Operatorul `=`
 - 3.3. Operatorul `==`
 - 3.4. Metoda

```
int cauta(String subsir)
realizează căutarea unui subșir într-un sir. Returnează prima poziție în sirul curent, în care a fost găsit subsir. Sau -1 dacă subșirul nu a fost găsit. De exemplu
sir.cauta(subsir) va returna 3, dacă sir reprezintă "alabala" iar subsir - "ba".
```
 - 3.5. Metoda

```
void afisare()
Afisează sirul.
```
 - 3.6. Metoda

```
int compara(String sir2)
Realizează compararea a 2 siruri în ordine lexicografică. Returnează -1 dacă sirul curent (this) este mai mic decât sir2, 0 dacă sirurile sunt egale, și 1 dacă sirul curent este mai mare.
```
 - 3.7. Constructorul `vid` – crează un sir `vid`.
 - 3.8. Constructorul cu argument un sir de caractere (`char *`).
 - 3.9. Constructorul de copiere.
 - 3.10. Destructorul.

POO - C++ - Laborator 7

Cuprins

1.	Moștenirea.....	1
2.	Membrii moșteniți din clasa de bază	4
3.	Crearea și ștergerea obiectelor instantiatе dintr-o clasă derivată	4
4.	Ierarhii de clase.....	5
5.	Proiectul Persoana	6
6.	Exerciții.....	9

1. Moștenirea

Moștenirea este posibilitatea de a crea clase noi, definite ca extensii ale altor clase. Clasa inițială se numește **clasa de bază**, iar clasa nouă, care extinde clasa de bază – **clasă derivată**. Proprietatea cea mai importantă a unei clase deriveate este aceea de a **moșteni** toți membrii definiți în clasa de bază. Deasemenea, clasa derivată poate să includă și membri noi, acesta fiind unul din scopurile pentru care este creată. Moștenirea este cel de-al 2-lea principiu al programării orientate obiect, următorul după încapsulare.

De exemplu putem să declarăm clasa Persoana, care să reprezinte orice persoană. Din Persoana putem să derivăm clasa Student. Clasa Persoana va avea câmpurile private nume și prenume, și metodele publice setValoriPersoana() și afisare(). Clasa Student va moșteni toți membrii clasei Persoana, și va avea în plus câmpul privat grupa, și metodele publice setValoriStudent() și o altă metodă afisare(), despre care vom discuta mai jos.

Ca să declarăm o clasă derivată, folosim următoarea sintaxă:

```
class nume_clasa_derivata: public nume_clasa_baza {  
    /*corful clasei deriveate*/  
};
```

Clasa de bază trebuie să fie declarată mai sus, folosind sintaxa obișnuită.

Prezentăm un exemplu complet de moștenire:

```

//clase derivate
#pragma warning(disable : 4996)
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;

class Persoana {
protected:
    char nume[20], prenume[20];
public:
    void setValoriPersoana(char nume[], char
prenume[]);
    void afisare();
};

class Student: public Persoana {
private:
    int grupa;
public:
    void setValoriStudent(char nume[], char prenume[],
int grupa);
    void afisare();
};

void Persoana::setValoriPersoana(char nume[], char
prenume[]) {
    strcpy(this->nume, nume);
    strcpy(this->prenume, prenume);
}

void Persoana::afisare() {
    cout << nume << " " << prenume << endl;
}

void Student::setValoriStudent(char nume[], char
prenume[], int grupa) {
    setValoriPersoana(nume, prenume);
    this ->grupa = grupa;
}

void Student::afisare() {
    cout << nume << " " << prenume << ", grupa: " <<
grupa << endl;
}

int main() {
    Persoana persoana;
    Student student;
    persoana.setValoriPersoana("Vasile", "Dumitrescu");
    student.setValoriStudent("Ion", "Ciubotaru", 1104);
    persoana.afisare();
    student.afisare();
    getch();
    return 0;
}

```

Iesire:

Vasile Dumitrescu
Ion Ciubotaru, grupa: 1104

Acest exemplu conține mai multe aspecte noi.

1. Modul de acces protected. Observăm că membrii nume și prenume din clasa Persoana au fost definiți cu un nou mod de acces – `protected`. Acest mod de acces este strâns legat de moștenire. Membrii definiti cu mod `protected` pot fi accesati de oriunde din interiorul clasei în care au fost definiți, sau din clasele derivate. Avem nevoie de un astfel de mod pentru cîmpurile din Persoana, pentru a le putea accesa din `Student::afisare()`.

Practic, drepturile pentru modul `protected` se situează între modurile `private` și `public`. Mai jos prezentăm un sumar cu modurile de acces:

Mod de acces	public	protected	private
Membri ai aceleiași clase	da	da	da
Membri ai claselor derivate	da	da	nu
Non-membri	Da	nu	nu

Aici non-membri semnifică orice funcție din afara clasei sau a claselor derivate, cum ar fi funcția `main()`, orice funcție globală sau metodele altiei clase.

2. Modul de moștenire. Observăm declarația clasei `Student`:

`class Student: public Persoana`

Cuvântul cheie `public` semnifică modul de acces maxim pe care îl vor avea membrii moșteniți din clasa de bază. Specificând modul de moștenire `public`, toți membrii moșteniți își vor păstra modul de acces inițial.

Dacă moștenim clasa de bază în modul `private`, toți membrii moșteniți din Persoana își vor păstra modul de acces pentru clasa `Student`, dar vor deveni private pentru restul programului. De exemplu, vom putea accesa metoda `setValoriPersoana()` din `setValoriStudent()`, și programul de mai sus se va compila. Dar nu vom putea accesa `student.setValoriPersoana()` din funcția `main()`. Se poate folosi și modul de moștenire `protected`.

În aproape toate cazurile se folosește moștenirea publică. Celelalte moduri de moștenire nu sunt recomandate.

3. Ascunderea metodei `afisare()`. Metoda `afisare()` a fost declarată atât în clasa Persoana, cât și în `Student` cu același nume și aceeași listă de parametri. Este permisă o astfel de declarație. În acest caz, metoda afişare din clasa derivată **ascunde** metoda cu același prototip din clasa de bază. În apelul

`student.afisare();`

se va apela metoda afişare din clasa `Student`. De fapt clasa `Student` va avea 2 metode cu același nume și aceeași parametri – `Persoana::afisare()` și `Student::afisare()`. Dar a 2-a metodă o ascunde pe prima. Totuși, metoda `Persoana::afisare()` nu este dispărută definitiv, ea poate fi accesată de exemplu de alte metode ale clasei `Persoana`.

2. Membrii moșteniți din clasa de bază

Clasa derivată moștenește următorii membri ai clasei de bază:

- Câmpurile
- Metodele
- Operatorii supraîncărcați, mai puțin operatorul =.

Nu sunt moșteniți din clasa de bază:

- Constructorii
- Destructorul
- Membrii operator=()
- Prietenii

3. Crearea și ștergerea obiectelor instantiate dintr-o clasă derivată

Deși constructorul și destructorul clasei de bază nu sunt moșteniți, aceștia au un rol și pentru clasa derivată.

Atunci când o nouă instanță a clasei deriveate este creată, compilatorul apelează mai întâi constructorul clasei de bază fără parametri, și pe urmă constructorul clasei deriveate.

Atunci când instanța unei clase deriveate este ștersă, compilatorul va apela mai întâi destructorul clasei deriveate, și mai apoi destructorul clasei de bază.

Alternativ, putem să specificăm ce constructor al clasei de bază trebuie apelat pentru fiecare constructor al clasei deriveate. Folosind următoarea sintaxă pentru a declara constructorul:

```
nume_clasa_derivata(parametri constructor)
    : nume_clasa_de_baza (parametri constructor clasa de
baza) {...}
```

De exemplu:

```
// constructori și clase deriveate
#include<iostream>
using namespace std;
#include<conio.h>

class A {
public:
    A() {
        cout << "A: fara parametri\n";
    }
    A(int a) {
        cout << "A: parametru int\n";
    }
};

class B : public A {
public:
```

```

        B (int a) {
            cout << "B: parametru int\n\n";
        }
    };

class C : public A {
public:
    C(int a) : A(a){
        cout << "C: parametru int\n\n";
    }
};

int main () {
    B b(0);
    C c(0);
    _getch();
    return 0;
}

```

Ieșire:

```

A: fara parametri
B: parametru int

A: parametru int
C: parametru int

```

Remarcăm că atunci când se crează un obiect de tip B, se apelează constructorul fără parametri al clasei A. Iar atunci când se instanțiază C, se apelează constructorul cu parametru al lui A. Diferența se datorează declarației constructorilor claselor B și C:

```

B(int a) /*nu s-a specificat constructorul bazei*/
           /* se apeleaza constructorul implicit*/
C(int a) : A(a) /* apeleaza constructorul bazei
specificat*/

```

În exemplele de până acum clasa derivată moștenea o singură clasă de bază, ceea ce se numește **moștenire simplă**. În C++ există posibilitatea ca o clasă să moștenească mai multe clase de bază. Această proprietate se numește **moștenire multiplă**.

4. Ierarhii de clase

Dacă considerăm o mulțime de obiecte diferite dar înrudite, și dacă din ele grupăm toate obiectele care au numai proprietățile comune pentru toată mulțimea, acestea vor fi instanțieri ale unei clase pe care am numit-o clasă de bază. Dacă grupăm în continuare obiecte care au proprietățile clasei de bază la care se adaugă proprietăți proprii grupului, obținem una sau mai multe clase derivate. În felul acesta se ajunge la noțiunea de ierarhie a claselor.

Clasa de bază a ierarhiei are întotdeauna nivelul 0. Clasele derivate din ea au nivel 1. Clasele derivate din clasele de nivel 1 au nivel 2, etc. Clasa de nivel 1 din care se derivează o clasă de nivel 2 este clasă de bază pentru aceasta. Deci noțiunea de clasă de bază – clasă derivată are caracter recursiv.

Exemplu:

```
class X {  
    //...  
};  
class Y :public X{  
    //...  
};  
class Z : public Y{  
    //...  
};
```

În exemplul prezentat avem o ierarhie de 3 clase.

5. Proiectul Persoana

Mai jos este dat codul sursă a unui program ce conține 2 clase – Persoana și Data. Exercițiile din acest laborator vor presupune completarea acestui program cu facilități noi. Pentru început, să creăm un proiect pe baza acestor fișiere:

Data.h

```
#ifndef _Data_h_  
#define _Data_h_  
  
class Data {  
  
private:  
    int an,luna,zi;  
  
public:  
    Data() {}  
    Data(int an, int luna, int zi);  
    int getAn();  
    int getLuna();  
    int getZi();  
  
    /*returneaza 1 daca this > data2, 0 daca this <= data2*/  
    int maiMare(Data data2);  
};  
  
#endif
```

Data.cpp

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

Data::Data(int an, int luna, int zi) {
    this->an = an;
    this->luna = luna;
    this->zi = zi;
}

int Data::getAn() {
    return an;
}

int Data::getLuna() {
    return luna;
}

int Data::getZi() {
    return zi;
}

/*returneaza 1 daca data1 > (este mai recenta decat)
data2, 0 in caz contrar*/
int Data::maiMare(Data data2) {
    if (an > data2.an) {
        return 1;
    } else if (an < data2.an) {
        return 0;
    } else {
        if (luna > data2.luna) {
            return 1;
        } else if (luna < data2.luna) {
            return 0;
        } else {
            if (zi > data2.zi) {
                return 1;
            } else if (zi < data2.zi) {
                return 0;
            } else {
                return 0;
            }
        }
    }
}
```

Persoana.h

```
#include "Data.h"

#ifndef _Persoana_h_
#define _Persoana_h_
#pragma warning(disable : 4996)

class Persoana {
private:
    char *nume, *prenume;
    Data *dataNastere;

protected:
    void afisarePersoana();

public:
    Persoana(char *nume, char *prenume, Data
    *dataNastere);
    ~Persoana();
    char *getNume();
    char *getPrenume();
    Data *getDataNastere();
    void afisare();
};

#endif
```

Persoana.cpp

```
#include<iostream>
#include<string.h>
#include "Data.h"
#include "Persoana.h"
using namespace std;

Persoana::Persoana(char *nume, char *prenume, Data
*pdataNastere) {
    this->nume = new char[strlen(nume) + 1];
    this->prenume = new char[strlen(prenume) + 1];
    strcpy(this->nume, nume);
    strcpy(this->prenume, prenume);
    this->dataNastere =
        new Data(*pdataNastere);
}

Persoana::~Persoana() {
    delete[] nume;
    delete[] prenume;
    delete dataNastere;
}

char *Persoana::getNume() {
    return nume;
}

char *Persoana::getPrenume() {
```

```

        return prenume;
    }

Data *Persoana::getDataNastere() {
    return dataNastere;
}

void Persoana::afisarePersoana() {
    cout << nume << " " << prenume
    << ", data nastere: "
    << dataNastere->getAn() << "."
    << dataNastere->getLuna() << "."
    << dataNastere->getZi();
}

void Persoana::afisare() {
    afisarePersoana();
    cout << endl;
}

```

DemoMain.cpp

```

#include "Data.h"
#include "Persoana.h"

int main() {

    Data data(2000, 3, 20);
    Persoana radu("Radu", "Stefan", &data);

    radu.afisare();
    return 0;
}

```

6. Exerciții

1. Creați o clasă Student, derivată din Persoana. Clasa va avea următorii membri noi:

- câmpul privat grupa de tip întreg. Grupa va lua valori între 1001 și 1999
- afisare() – va afișa studentul la consolă.

Scriți o funcție main care să creeze un vector de 5-10 studenți, din 3 grupe diferite, și să-i afișeze sortați după grupe.

2. Scriți o clasă Angajat, derivată din Persoana. Clasa angajat va avea următorii membri:

- câmpul privat dataAngajare de tip Data
- câmpul privat salariu de tip int – salariu lunar.
- Constructor care să initializeze toate câmpurile pe baza parametrilor
- Destructor
- Metode getDataAngajare(), getSalariu(), care să returneze valoarea câmpurilor.
- int getVarstaAngajare() – va returna vîrstă la care s-a făcut angajarea, în ani

împliniți.

- void afisarePerioadaMuncita(Data &dataCurenta) – va afișa perioada muncită în firmă, până la data curentă. Sirul va avea formatul "ani.luni.zile". Exemplu: "2.9.20". Vom considera pentru simplitate că toate lunile au câte 30 de zile.

- afisare() – va afișa informațiile despre angajat la consolă. Doar informațiile stocate în câmpurile clasei, nu și cele calculate. Este indicat să se folosească metoda afisarePersoana().

Scrieți o funcție main care să creeze o listă de 3-4 angajați, și să afișeze vârsta angajării, perioada muncită și detaliile despre angajat, pentru toți angajații, folosind metodele definite.

POO - C++ - Laborator 8

Cuprins

1.	Ordinea de apelare a constructorilor și a destructorilor	1
2.	Mostenirea multiplă	3
3.	Exerciții	4

1. Ordinea de apelare a constructorilor și a destructorilor

- ▶ *Ordinea de apelare a constructorilor pentru un obiect dintr-o clasă derivată:*
 - ▶ *Constructorul clasei de bază*
 - ▶ *Constructorul clasei deriveate*
- ▶ *Ordinea de apelare a destructorilor pentru un obiect dintr-o clasă derivată:*
 - ▶ *Destructorul clasei deriveate*
 - ▶ *Destructorul clasei de bază*

Prezentăm un exemplu complet de moștenire, cu clasele completeate cu constructori și destructori. În cazul destructorilor, aceștia vor afisa un simplu mesaj, deoarece nu există dealocări de memorie necesare:

```
Persoana.h
class PersoanaAC
{
protected:
    string m_sCnp;
    string m_sNume;
    string m_sAdresa;
public:
    PersoanaAC();
    PersoanaAC(string cnp, string nume, string adresa);
    ~PersoanaAC();
    void afisareProfil();
    void schimbareAdresa(string adresaNoua);
};

Student.h
class StudentAC : public PersoanaAC
{
    int m_iAnStudiu;
    int m_iNotaP2;
public:
    StudentAC();
    StudentAC(string cnp, string nume, string adresa, int
anStudiu, int notaP2);
    ~StudentAC();
    void afisareProfil();
    void inscriereAnStudiu(int anStudiuNou);
};
```

Persoana.cpp

```
PersoanaAC::PersoanaAC()
{
    cout<<"constr. fara arg. PersoanaAC"<<endl;
    m_sCnp = string(13, '0');
    m_sNume = "";
    m_sAdresa = "";
}
PersoanaAC::PersoanaAC(string cnp, string nume, string adresa)
{
    cout<<"constr. cu arg. PersoanaAC"<<endl;
    m_sCnp = cnp;
    m_sNume = nume;
    m_sAdresa = adresa;
}
PersoanaAC::~PersoanaAC()
{
    cout<<"destructor PersoanaAC"<<endl;
}
```

Student.cpp

```
StudentAC::StudentAC()
{
    cout<<"constr. fara arg. StudentAC"<<endl;
    m_iAnStudiu = 0;
    m_iNotaP2 = 0;
}
StudentAC::StudentAC(string cnp, string nume, string adresa,
int anStudiu, int notaP2) :
    PersoanaAC(cnp, nume, adresa), m_iAnStudiu(anStudiu),
    m_iNotaP2(notaP2)
{
    cout<<"constr. cu arg. StudentAC"<<endl;
}
StudentAC::~StudentAC()
{
    cout<<"destructor StudentAC"<<endl;
}
```

Test.cpp

```
int main ()
{
    PersoanaAC p1("1234567890123" , "Ana", "Iasi");
    p1.afisareProfil();
    StudentAC s2;
    s2.afisareProfil();
    StudentAC s1("1234567890122", "Ion", "Vaslui", 2, 10);
    s1.schimbareAdresa("Bucuresti");
    s1.inscriereAnStudiu(3);
    s1.afisareProfil();
    return 0;
}
```

La rulare, acest exemplu va furniza la ieșire:

```
constr. cu arg. PersoanaAC
Nume: Ana CNP: 1234567890123 Adresa: Iasi
constr. fara arg. PersoanaAC
constr. fara arg. StudentAC
Nume: CNP: 00000000000000 Adresa:
An studiu: 0 Nota P2: 0
constr. cu arg. PersoanaAC
constr. cu arg. StudentAC
Nume: Ion CNP: 1234567890122 Adresa: Bucuresti
An studiu: 3 Nota P2: 10
destructor StudentAC
destructor PersoanaAC
destructor StudentAC
destructor PersoanaAC
destructor PersoanaAC
```

Se observă apelarea constructorului fără argumente al clasei `PersoanaAC` pentru obiectul `s2` de tipul `StudentAC`. Acest constructor este apelat deoarece pentru un obiect al unei clase derivate, constructorul apelat implicit pentru clasa de bază este constructorul fără listă de argumente (dacă există constructori declarați în clasă).

Prin apeluri de forma

```
StudentAC::StudentAC(...):
    PersoanaAC(cnp, nume, adresa), ...
```

se poate controla ce constructor din clasa de bază va fi folosit (se forțează compilatorul pentru a apela un anumit constructor).

2. Mostenirea multiplă

Mostenirea multiplă permite unei clase derivate să moștenescă mai mult de un singur părinte. Considerăm exemplul clasei `Profesor` care poate extinde atât clasa `Persoana`, cât și clasa `Angajat`:

```
Persoana.h
class Persoana
{
private:
    string m_sNume;
    int m_iVarsta;

public:
    Persoana(string nume, int varsta)
        : m_sNume (nume), m_iVarsta(varsta)
    {}
    int getVarsta()
    {
        return m_iVarsta;
```

```
    }  
};
```

Angajat.h

```
class Angajat  
{  
private:  
    double m_dSalariu;  
  
public:  
    Angajat(double salariu)  
        : m_dSalariu(salariu)  
    {}  
    double getSalariu()  
    {  
        return m_dSalariu;  
    }  
};
```

Profesor.h

```
class Profesor: public Persoana, public Angajat  
{  
private:  
    int m_iGradDidactic;  
  
public:  
    Profesor(string nume, int varsta, double salariu, int  
gradDidactic)  
        : Persoana(nume, varsta),  
        Angajat(salariu),  
        m_iGradDidactic(gradDidactic)  
    {  
    }  
};
```

Test.cpp

```
int main()  
{  
    Profesor p1("Ion", 23, 100.08, 1);  
    return 0;  
}
```

In exemplul de mai sus, obiectul p1 de tipul Profesor va avea variabila membră proprie clasei Profesor (m_iGradDidactic), dar va avea și variabilele membre din cele două clase de bază pe care le moștenește: m_sNume și M_iVarsta din clasa Persoana, respectiv m_dSalariu din clasa Angajat.

3. Exerciții

- Implementați cele două exemple furnizate în cadrul laboratorului.

- Verificați la adresa <http://www.cplusplus.com/reference/string/string/string/> ce alte tipuri de constructori pot fi folosiți la inițializarea variabilelor de tip string. Pentru unul din exemplele implementate, utilizați cel puțin alți doi constructori din lista de pe site.
- Extindeți primul exemplu cu implementarea următoarelor:
 - O funcție afisareProfil ce nu aparține nici unei clase și afișează datele corespunzătoare unui Student.
 - O funcție membră a clasei StudentAC ce compară notele a doi studenți (între nota studentului pentru care se apelează funcția și nota studentului primit ca parametru) și returnează un pointer la studentul cu nota cea mai mare.
 - O nouă clasă StudentMaster derivată din StudentAC cu următorii membri:
 - m_sNumeDizertatie de tip string ce va conține numele lucrării de dizertație
 - Pentru clasa StudentMaster să se implementeze constructorii necesari și un destructor și să se testeze ordinea de apelare a acestora.
 - Sa se creeze în main() un vector de obiecte de tip StudentMaster și să se determine studentul cu nota cea mai mare (m_inotap2).

POO - C++ - Laborator 9

Cuprins

1.	Pointeri la clasa de bază.....	1
2.	Metode virtuale. Polimorfismul.....	4
1.	Destructor virtual.....	6
2.	Clase abstracte.....	7
3.	Exerciții.....	11

1. Pointeri la clasa de bază

O proprietate importantă a moștenirii este faptul că pointerii către o clasă de bază pot primi adresa unui obiect de tip clasă derivată.

Să ilustrăm această proprietate într-un exemplu. Avem clasa de bază `Carte` cu câmpul `titlu` și funcția `afisare()`. Din ea este derivată clasa `Culegere`, care este o carte ce conține mai multe lucrări. `Culegere` are 2 câmpuri în plus – `nrLucrari` și `lucrari` – numărul și titlurile lucrărilor. și mai are o altă funcție `afisare()` – care afișează toate datele din culegere.

Carti.h

```
#ifndef _Carti_
#define _Carti_
#pragma warning(disable : 4996)

class Carte {
protected:
    char *titlu;
public:
    Carte(char *titlu);
    ~Carte();
    void afisare();
};

class Culegere : public Carte {
protected:
    int nrLucrari;
    char **lucrari;
public:
    Culegere(char *titlu, int nrLucrari, char
**lucrari);
    ~Culegere();
    void afisare();
};
```

```
#endif
```

Carti.cpp

```
#include<iostream>
#include<string.h>
#include "Carti.h"
using namespace std;

Carte::Carte(char *titlu) {
    this->titlu = new char[strlen(titlu)+1];
    strcpy(this->titlu, titlu);
}

Carte::~Carte() {
    delete[] titlu;
}

void Carte::afisare() {
    cout << titlu << endl;
}

Culegere::Culegere(char *titlu, int nrLucrari, char **lucrari): Carte(titlu) {
    this->nrLucrari = nrLucrari;
    this->lucrari = new char*[nrLucrari];
    for(int i=0; i<nrLucrari; i++) {
        char *lucrare =
            new char[strlen(lucrari[i])+1];
        strcpy(lucrare, lucrari[i]);
        this->lucrari[i] = lucrare;
    }
}

Culegere::~Culegere() {
    for(int i=0; i<nrLucrari; i++) {
        delete[] lucrari[i];
    }
    delete[] lucrari;
}

void Culegere::afisare() {
    cout << titlu << ". Lucrari:" << endl;
    for(int i=0; i<nrLucrari; i++) {
        cout << "    " << lucrari[i] << endl;
    }
}
```

CartiMain.cpp

```
#include<iostream>
#include<conio.h>
#include "Carti.h"
```

```

using namespace std;

int main() {
    Carte *carte = new Carte("Moara cu Noroc");
    char *poeme[] = {"Luceafarul", "Memento Mori"};
    Culegere *culeg = new Culegere("Poeme Eminescu", 2,
poeme);
    Carte *cculeg = culeg;

    cout << "carte->afisare(): ";
    carte->afisare();
    cout << "cculeg->afisare(): ";
    cculeg->afisare();
    cout << "culeg->afisare(): ";
    culeg->afisare();

    delete carte;
    delete culeg;
    _getch();
    return 0;
}

```

Iesire:

```

carte->afisare(): Moara cu Noroc
cculeg->afisare(): Poeme Eminescu
culeg->afisare(): Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori

```

Observăm atribuirea din funcția main():

```
Carte *cculeg = culeg;
```

Atribuirea unui pointer de tip clasă derivată `Culegere*` către un pointer de tip clasă de bază `Carte*` este perfect validă.

Prin intermediul pointerului la clasa de bază `Carte` putem accesa orice membru din `Carte`, indiferent dacă obiectul referit este de tip `Carte` sau `Culegere`. De exemplu putem să accesăm funcția `afisare()`.

Și într-adevar, observăm că în rezultatul apelului

```
cculeg->afisare();
```

este executată funcția `Carte::afisare()`. Ca să putem accesa funcția `Culegere::afisare()`, în acest exemplu, suntem nevoiți să utilizăm pointerul la clasa derivată:

```
culeg->afisare();
```

Funcțiile `Carte::afisare()` și `Culegere::afisare()` nu au nici o legătură între ele în acest program.

2. Metode virtuale. Polimorfismul.

Metodă (funcție) virtuală este o metodă definită în clasa de bază, și care poate fi redefinită în clasele derivate. Se declară cu ajutorul cuvântului cheie `virtual`. Atunci când apelăm o metodă virtuală prin intermediul unui pointer la obiect, se va apela întotdeauna metoda din tipul obiectului, indiferent de tipul pointerului.

Se spune că metodele virtuale pot fi **redefined** (overloaded) în clasele derivate. (A se face diferența dintre *redefineire* și *supraîncarcare*, două facilități diferite.)

Să modificăm exemplul de mai sus, și să declarăm funcția `afisare()` din clasa `Carte` cu cuvântul cheie `virtual` în față:

Carti.h

```
...
class Carte {
protected:
    char *titlu;
public:
    Carte(char *titlu);
    ~Carte();
    virtual void afisare();
};

...
```

Restul programului rămâne neschimbat. Executăm programul. Rezultatul devine:

```
carte->afisare(): Moara cu Noroc
cculeg->afisare(): Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori
culeg->afisare(): Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori
```

De data aceasta funcțiile `Carte::afisare()` și `Culegere::afisare()` sunt ambele virtuale, și sunt legate. A 2-a funcție o redifineste pe prima.

Întrucât pointerii `culeg` și `cculeg` referă același obiect, atât pentru expresia

```
cculeg->afisare();
```

cât și pentru

```
culeg->afisare();
```

se va apela funcția `Culegere::afisare()` – funcția din tipul obiectului. Chiar dacă cei 2 pointeri sunt de tipuri diferite.

O funcție definită virtuală în clasa de bază, va rămâne virtuală în tot arborele ierarhic al acelei clase, indiferent către nivelele de derivare. Funcția redefinită trebuie să aibă aceiași parametri și același tip de return ca și funcția originală din clasa de bază.

Datorită funcției afisare() virtuale, putem rescrie funcția main() astfel încât să folosim doar pointeri de tip Carte*:

CartiMain.cpp

```
#include<iostream>
#include<conio.h>
#include"Carti.h"
using namespace std;

int main() {
    char *poeme[] = {"Luceafarul", "Memento Mori"};
    Carte *carte = new Carte("Moara cu Noroc");
    Carte *cculeg = new Culegere("Poeme Eminescu", 2,
poeme);

    cout << "carte->afisare(): ";
    carte->afisare();
    cout << "cculeg->afisare(): ";
    cculeg->afisare();

    delete carte;
    delete cculeg;
    _getch();
    return 0;
}
```

Iesire:

```
carte->afisare(): Moara cu Noroc
cculeg->afisare(): Poeme Eminescu. Lucrari:
    Luceafarul
    Memento Mori
```

Putem generaliza programul și mai mult și să utilizăm un vector de pointeri de tip Carte*, al căruia elemente să fie pointeri la diferite tipuri de obiecte. Să modificăm funcția main() în felul următor:

CartiMain.cpp

```
#include<iostream>
#include<conio.h>
#include"Carti.h"
using namespace std;

int main() {
    char *poeme[] = {"Luceafarul", "Memento Mori"};
    Carte *carti[3];
    carti[0] = new Carte("Moara cu Noroc");
    carti[1] = new Culegere("Poeme Eminescu", 2,
poeme);
    carti[2] = new Carte("Amintiri din copilarie");

    for(int i=0; i<3; i++) {
        cout << i << ". ";
        carti[i]->afisare();
        delete carti[i];
    }
}
```

```
    _getch();
    return 0;
}
```

Ieșire:

- 0. Moara cu Noroc
- 1. Poeme Eminescu. Lucrari:
 - Luceafarul
 - Memento Mori
- 2. Amintiri din copilarie

Observăm că singurul loc din program care știe tipul obiectelor sunt liniile de cod care realizează instanțierea:

```
carti[0] = new Carte("Moara cu Noroc");
carti[1] = new Culegere("Poeme Eminescu", 2, poeme);
carti[2] = new Carte("Amintiri din copilarie");
```

În restul programului, o singură instrucțiune știe să afișeze fiecare carte în mod corect, corespunzător tipului său:

```
carti[i]->afisare();
```

Polimorfismul reprezintă capacitatea de a apela metode diferite prin intermediul unei singure expresii de genul

```
pb->f()
```

, unde `pb` este pointer la o clasă de bază, iar `f()` este o funcție virtuală redefinită în clasele derivate. Funcția apelată este decisă în timpul execuției programului, în funcție de tipul obiectului referit de `pb`.

Un astfel de apel de funcție, pentru care nu se cunoaște funcția concretă apelată în momentul compilării programului, se numește **apel polimorfic**.

Polomorfismul reprezintă cel de-al 3-lea principiu al POO, după **Încapsulare (abstractizare)** și **Moștenire**.

Principiul polimorfismului ridică POO la adevărata sa valoare, îndeosebi în proiectele mari. Putem să scriem o bibliotecă de clase în care să manipulăm obiectele prin intermediul pointerilor la clasa de bază, fără să cunoaștem tipul concret al obiectelor. Iar în alte proiecte, să folosim biblioteca în combinație cu mai multe clase derivate.

1. Destructor virtual

Ultimul program dat ca exemplu afișează rezultatul așteptat, dar conține un defect. Pe linia:

```
delete carti[i];
```

operatorul `delete` va apela destructorul. Dar pentru că pointerul este de tip `Carte*`, destructorul apelat va fi doar `Carte::~Carte()`. Iar câmpul - pointer `lucrari`, din obiectele de tip `Culegere` va rămâne dezalocat. Ca să corectăm acest defect, vom declara și destructorul clasei `Carte` virtual:

```

...
class Carte {
protected:
    char *titlu;
public:
    Carte(char *titlu);
    virtual ~Carte();
    virtual void afisare();
};
...

```

De data aceasta, la apelarea operatorului `delete` pentru pointer la clasa de bază, se va apela tot lanțul de destructori de la tipul obiectului până la cea mai de bază clasă, indiferent de tipul pointerului.

Pentru a avea garanția că obiectele sunt întotdeauna dezalocate corect, există următoarea regulă:

În orice ierarhie de clase C++, clasa cea mai de bază trebuie să conțină un destructor virtual. Chiar dacă în clasa de bază nu avem ce dezaloca, vom defini un destructor virtual vid (fără instrucțiuni).

2. Clase abstracte

Există cazuri când o funcție virtuală nu are sens să fie implementată într-o clasă de bază, dar are sens în clasele derivate. De exemplu putem avea clasa de bază `Figura` cu funcția virtuală `arie()` (convenim că orice figură în plan are o aria), și clasele derivate `Dreptunghi` și `Cerc`. Cunoaștem formula ariei pentru dreptunghi și cerc, dar nu putem defini aria pentru o figură în general.

În acest caz, vom declara funcția din clasa de bază **virtuală pură**.

Funcție virtuală pură este o funcție virtuală care nu are corp. Se declară adăugând simbolurile `"=0;"` la sfârșitul declarației funcției.

De exemplu, declarația funcției virtuale pure `arie()` va arăta în felul următor:

```

class Figura {
public:
    virtual float arie()=0;
    ...
};

```

Clasă abstractă este o clasă care conține cel puțin o funcție virtuală pură. Clasele abstracte au restricția că nu pot fi instanțiate, adică nu putem crea obiecte pe baza lor. Ele sunt create special pentru a fi derivate. Însă putem să declarăm pointeri pe clase abstracte, care vor referi obiecte de tip clase derivate concrete.

Clasele derivate în schimb trebuie să implementeze funcțiile virtuale pure moștenite, pentru a deveni clase concrete.

Codul de mai jos conține declarația și implementarea claselor `Figura`, `Dreptunghi` și `Cerc`. Clasele derivate implementează metodele virtuale pure moștenite de la clasa de bază.

figuri.h

```
#ifndef _figuri_
#define _figuri_
#pragma warning(disable : 4996)

class Figura {
public:
    virtual ~Figura() {}
    virtual float arie()=0;
    virtual void afisare()=0;

};

class Dreptunghi : public Figura {
private:
    int x1,y1,x2,y2;
public:
    Dreptunghi(int x1,int y1,int x2,int y2);
    float arie();
    void afisare();
};

class Cerc : public Figura {
private:
    int x,y,r;
public:
    Cerc(int x, int y, int r);
    float arie();
    void afisare();
};

#endif
```

figuri.cpp

```
#include<iostream>
#include"Figuri.h"
using namespace std;

Dreptunghi::Dreptunghi(int x1, int y1, int x2, int y2) {
    this->x1 = x1;
    this->y1 = y1;
    this->x2 = x2;
    this->y2 = y2;
}

float Dreptunghi::arie() {
    return (float)(x2 - x1)*(y2 - y1);
}

void Dreptunghi::afisare() {
    cout << "Dreptunghi cu coordonatele (" 
<<x1<<, "<<y1
                <<") - ("<<x2<<, "<<y2<<"), si aria " <<
arie()<<endl;
}
```

```

Cerc::Cerc(int x, int y, int r) {
    this->x = x;
    this->y = y;
    this->r = r;
}

float Cerc::arie() {
    const float PI = 3.14F;
    return PI * r * r;
}

void Cerc::afisare() {
    cout << "Cerc cu coordonatele (" 
        <<x<<", "<<y
        <<"), raza " << r <<" si aria "
        << arie()<<endl;
}

```

figuriMain.cpp

```

#include<conio.h>
#include"Figuri.h"

int main() {
    Figura *dr = new Dreptunghi(1,2,4,4);
    Figura *cerc = new Cerc(1,1,3);
    //urmatoarea linie ar genera eroare:
    //Figura *fig = new Figura();
    dr->afisare();
    cerc->afisare();
    delete dr;
    delete cerc;
    _getch();
    return 0;
}

```

Dacă am încerca să creăm un obiect de tip `Figura` în funcția `main()`, am obține o eroare de compilare, deoarece clasa `Figura` este abstractă și nu poate fi instantiată.

În continuare vom folosi aceste 3 clase într-un exemplu care demonstrează mai elovent avantajele polimorfismului.

Vom adăuga la program o funcție globală care va determina figura cu aria maximă dintr-un vector de pointeri la figuri:

figuri.h

```

...
Figura *figCuArieMax(Figura **figuri, int n);

#endif

```

figuri.cpp

```

//...
Figura *figCuArieMax(Figura **figuri, int n) {

```

```

float max = 0;
Figura *figMax = NULL;
for(int i=0; i<n; i++) {
    float arie = figur[i]->arie();
    if (arie > max) {
        max = arie;
        figMax = figur[i];
    }
}
return figMax;
}

```

În funcția `main()` sunt instanțiate câteva figuri. Apoi este determinată și afișată figura cu arie maximă.

figuriMain.cpp

```

#include<iostream>
#include<conio.h>
#include"Figuri.h"
using namespace std;

int main() {
    const int n = 3;
    Figura *figuri[n];
    figur[0] = new Dreptunghi(0,0,2,5);
    figur[1] = new Dreptunghi(0,0,2,2);
    figur[2] = new Cerc(0,0,3);
    Figura *figMax = figCuArieMax(figuri, n);

    cout << "      Dintre figurile:"<<endl;
    for(int i=0; i<3; i++) {
        cout << i << ". ";
        figur[i]->afisare();
    }
    cout << endl
        <<"      aria maxima o are:"<<endl;
    figMax->afisare();
    for(int i=0; i<n; i++) {
        delete figur[i];
    }
    _getch();
    return 0;
}

```

Ieșire

```

Dintre figurile:
0. Dreptunghi cu coordonatele (0,0)-(2,5), si aria 10
1. Dreptunghi cu coordonatele (0,0)-(2,2), si aria 4
2. Cerc cu coordonatele (0,0), raza 3 si aria 28.26

aria maxima o are:
Cerc cu coordonatele (0,0), raza 3 si aria 28.26

```

Se observă că funcția `figCuArieMax()` poate să determine figura cu aria maximă chiar și dintr-o listă care conține atât dreptunghiuri cât și cercuri. Acest lucru a fost posibil datorită apelului polimorfic a funcției `arie()` în funcția `figCuArieMax()`:

```
float arie = figur[i]->arie();
```

Putem chiar să extindem programul și să adăugam noi tipuri de figuri, iar funcția `figCuArieMax()` va să automat să determine aria maximă și pentru ele.

3. Exerciții

Continuați dezvoltarea ultimului exemplu din acest laborator. Clasa `Figura` și derivatele sale. Efectuați următoarele înbunătățiri:

1. Adăugați clasa `Triunghi` derivată din `Figura`.

Triunghiul va fi definit de cele 3 vârfuri, fiecare având coordonatele x și y. (Câți parametri va avea constructorul clasei `Triunghi`?)

Aria triunghiului se poate calcula după formula lui Heron: $s = \sqrt{p*(p-a)*(p-b)*(p-c)}$, unde p este semiperimetru: $p = (a + b + c) / 2$.

2. Adăugați la clasa `Figura` funcția virtuală pură `perimetru()`.

Afișați figura cu perimetru maxim.

3. Scrieți o funcție care sortează un vector de pointeri la figuri crescător după aria. Afișați figurile sortate.

POO - C++ - Laborator 10

Cuprins

1. Template-uri de funcții	1
2. Template-uri de clase	4
3. Exerciții	6

1. Template-uri de funcții

Template-urile de funcții se utilizează în situațiile în care operațiile realizate nu depind în mod necesar de tipul de dată al parametrilor implicați. Spre exemplu, metoda de calcul a maximului dintre două numere este aceeași pentru valori întregi sau reale. În mod normal, ar trebui să scriem câte o funcție pentru fiecare tip de dată implicat (int, float sau double):

```
int maxim(int a, int b)
{
    return (a < b) ? b : a;
}

float maxim(float a, float b)
{
    return (a < b) ? b : a;
}

double maxim(double a, double b)
{
    return (a < b) ? b : a;
}
```

Observăm faptul că toate trei funcțiile fac același lucru, fiind suficientă doar existența unei modalități de comparare a celor doi parametri. Template-urile permit evitarea implementării acelorași operații pentru tipuri de dată multiple. Astfel, vom crea un singur template de funcție, căruia ii vom transmite tipul de dată ca parametru, după modelul urmator:

```
template <typename T>
T maxim(T a, T b)
{
    return (a < b) ? b : a;
}
```

Tipul de dată T este încă nespecificat. Vom specifica acest tip la apelul funcției:

```
void main()
{
    int x = 1, y = 2;
    float a = 2.3, b = 1.5;

    int maxInt = maxim<int>(x, y);
    float maxFloat = maxim<float>(a, b);
}
```

Tipul de dată al unui template poate fi oricare, inclusiv o clasă creata de utilizator:

```
class Masina
{
    char *culoare;
    int an;

public:
    Masina(char *c, int a)
    {
        int len = strlen(c);
        culoare = new char[len + 1];
        strcpy_s(culoare, len + 1, c);
        an = a;
    }

    Masina(const Masina &m)
    {
        int len = strlen(m.culoare);
        culoare = new char[len + 1];
        strcpy_s(culoare, len + 1, m.culoare);
        an = m.an;
    }

    void afiseaza()
    {
        cout << culoare << " " << an << endl;
    }
};
```

```
void main()
{
    Masina m1("alb", 1999), m2("negru", 2000);
    Masina m3 = maxim<Masina>(m1, m2); //eroare! cum comparăm două mașini?
    m3.afiseaza();
}
```

În acest caz, pentru ca funcția maxim() să lucreze corect, trebuie specificată o modalitate de comparare a două obiecte din acea clasă, de exemplu prin supraîncărcarea operatorului de comparare < . Presupunem că dorim sa determinăm masina cu cel mai recent (mai mare) an de fabricație:

```
class Masina
{
    char *culoare;
    int an;

public:
    Masina(char *c, int a)
    {
        int len = strlen(c);
        culoare = new char[len + 1];
        strcpy_s(culoare, len + 1, c);
        an = a;
    }

    Masina(const Masina &m)
    {
        int len = strlen(m.culoare);
        culoare = new char[len + 1];
        strcpy_s(culoare, len + 1, m.culoare);
        an = m.an;
    }

    void afiseaza()
    {
        cout << culoare << " " << an << endl;
    }

    bool operator<(const Masina &m)
    {
        return (an < m.an) ? true : false;
    }
};
```

```
void main()
{
    Masina m1("alb", 1999), m2("negru", 2000);

    //in acest caz se compara m1.an si m2.an
    Masina m3 = maxim<Masina>(m1, m2);
    m3.afiseaza();
}
```

Același lucru e valabil și în cazul altor operatori (+, *, =, ==, etc.). Ori de câte ori se dorește utilizarea lor în cadrul unei funcții, trebuie ca rolul lor să fie clar stabilit atunci când se lucrează cu tipuri de date definite

de programator (clase). Acest lucru este cu atât mai ușor de trecut cu vederea în cazul template-urilor, unde tipurile de date utilizate nu sunt specificate, fiind astfel mai dificil de anticipat dacă e nevoie ca o anumită operație să fie definită explicit.

Template-urile pot apărea, de asemenea, și ca membri ai unei clase (template-uri de metode):

```
class A
{
    int val;

public:
    A(int v):val(v) {}

    template <typename T>
    T Produs(T x)
    {
        return x * val;
    }
};
```

```
void main()
{
    A a(2);
    float b = 3;

    cout << a.Produs<float>(b) << endl;
}
```

2. Template-uri de clase

Clasele pot fi, de asemenea, specificate prin intermediul template-urilor. Un template de clasă are unul sau mai multe tipuri, date ca parametri. Acești parametri pot fi folosiți în interiorul clasei, urmând să se specifice la crearea unui obiect din acea clasă.

Considerăm ca exemplu o clasă care definește un punct în plan, cu cele două coordonate ale sale, x, y. La definirea clasei, nu interesează deocamdată dacă acele coordonate sunt întregi sau reale. Acest lucru se va specifica doar în momentul în care vom crea obiecte de tip Punct:

```
template <typename T>
class Punct
{
    T x, y;
public:
    Punct(T xcoord, T ycoord) : x(xcoord), y(ycoord) {}
    void afiseaza()
    {
        cout << x << ", " << y << endl;
    }
};
```

```
};
```

```
void main()
{
    Punct<int> punctI(2, 3);
    Punct<float> punctF(2.3f, 3.4f);
    Punct<double> punctD(4.5, 5.6);

    punctI.afiseaza();
    punctF.afiseaza();
    punctD.afiseaza();
}
```

Un caz frecvent de utilizare a template-urilor de clasă îl constituie încapsularea și gestiunea unei structuri de date (ex. un vector). Ca și în cazul anterior, nu interesează tipul de date efectiv al elementelor din acea structură de date. Dorim să scriem o singură clasă care, teoretic, să funcționeze cu date de orice tip. Acel tip va fi specificat doar la crearea obiectelor din acea clasă:

```
template <typename T>
class MyVector
{
    T* elem;
    int nrElem;
public:
    MyVector(int n)
    {
        nrElem = n;
        elem = new T[n];
    }
    ~MyVector()
    {
        if(elem)
            delete[] elem;
    }

    //supraincarcarea operatorului de indexare:
    T& operator[](int index)
    {
        return elem[index];
    }
};
```

```
void main()
{
    int n = 10;
    MyVector<int> vectorI(n); //un vector de numere întregi
    MyVector<float> vectorF(n); //un vector de numere reale

    for(int i = 0; i < n; i++)
```

```

    {
        /* accesam elementele din membrul privat elem
        prin intermediul operatorului [] supraincarcat in clasa */
        vectorI[i] = i + 1;
        vectorF[i] = 1.5 * i;
    }
}

```

Template-urile se utilizează în cardul unui stil de programare numit "programare generică" (*generic programming*), unde se dorește separarea algoritmilor și metodelor de tipul și caracteristicile datelor cărora le vor fi aplicate. De exemplu, un algoritm de sortare se aplică datelor de orice tip, fiind suficient să existe metode de comparare și interschimbare a oricărora două elemente de acel tip.

3. Exerciții

1. Scrieți un template de funcție care să sorteze elementele unui vector și să returneze vectorul astfel sortat. Funcția primește ca parametri un pointer la un tip oarecare și un număr întreg (numărul de elemente ale vectorului). Se poate utiliza orice metodă de sortare (eg. *bubble sort*). Aplicați funcția pentru vectori de diferite tipuri (int, float, etc).

2. Scrieți un template de clasă Multime, care să conțină un pointer la un tip oarecare (vectorul cu elementele mulțimii) și un număr întreg (numărul de elemente din mulțime).

- Definiți constructori de inițializare și copiere și destructorul clasei. Testați-i pentru mulțimi de numere întregi și reale;
- Supraîncărcați operatorul ! (semnul exclamării) pentru a obține cel mai mare element din mulțime;
- Supraîncărcați operatorul + pentru a aduna două mulțimi element cu element;
- Scrieți un template de funcție Aduna, care primește doi parametri de un tip oarecare și returnează rezultatul adunării lor. Utilizați template-ul pentru a aduna două multimi cu elemente de tip double.

POO - C++ - Laborator 11

Cuprins

1. Biblioteca standard std C++
2. Funcții standard de intrare/ieșire
3. Funcții standard pentru manipularea șirurilor de caractere
4. Iteratorul
5. Clasa Vector
6. Metoda sort din biblioteca <algorithm>
7. Probleme propuse

1. Biblioteca standard C++

Funcțiile bibliotecii standard din C++ pot fi împărțite în două categorii:

Biblioteca **funcțiilor standard** conține funcții generice de sine stătătoare care nu pot fi încadrate în nici o clasă. Este moștenită din C. Biblioteca de funcții standard este împărțită în următoarele categorii:

- Funcții de intrare/ ieșire (I/O);
- Funcții pentru șiruri de caractere și pentru manipularea acestora;
- Funcții matematice;
- Funcții pentru managementul timpului, a datei și a localizării;
- Funcții pentru alocare dinamică;
- Funcții auxiliare;
- Funcții support pentru diferite seturi de caractere.

Clasele orientate obiect sunt reprezentate de o colecție de clase și de funcțiile asociate acestora. Biblioteca standard de clase orientate obiect din C++ definește o gamă variată de clase asigurând suportul pentru numeroase operații de bază, inclusive I/O, șiruri sau procesări numerice. Biblioteca cuprinde următoarele clase:

- Clase standard C++ de intrare/ieșire;
- Clasa String;
- Clase numerice;
- Algoritmi STL;
- Iteratori STL, etc.

1. Funcții standard de intrare/ieșire

Fișierul Header	Funcții și descriere
<iostream>	Acest fișier definește funcțiile cin , cout , cerr și clog corespunzătoare intrării standard, ieșirii standard, erorii standard fără memorie-tampon și erorii standard cu memorie-tampon.
<iomanip>	Acest fișier declară servicii utile pentru desfășurarea operațiilor de intrare/ieșire cu format prin intermediul manipulatorilor parametrizează cum ar fi setw și setprecision .
<fstream>	Acest fișier declară sevicii folosite pentru procesarea controlată a fișierelor.

```
#include <iostream>

using namespace std;

int main()
{
    //cout
    char str[] = "Hello C++";
    cout << "Valoarea sirului este : " << str << endl;

    //cin
    char name[50];
    cout << "Introduceti numele: ";
    cin >> name;
    cout << "Numele introdus este: " << name << endl;

    //cerr
    char stre1[] = "Citire nereusita....";
    cerr << "Mesaj eroare : " << stre1 << endl;

    //clog
    char stre2[] = "Citire nereusita....";
    clog << "Mesaj eroare : " << stre2 << endl;

    return 0;
}
```

Cele 3 streamuri de ieșire se folosesc, de obicei, pentru:

- std::cout – Regular output (console output)
- std::cerr – Error output (console error)
- std::clog – Log output (console log)

În cele mai multe cazuri aceste streamuri sunt direcționate spre ieșirea standard (consola), dar ele pot fi re-direcționate în funcție de specificațiile programului. De exemplu, cout poate fi direcționat pentru a scrie într-un fișier iar cerr pentru a scrie în alt fișier.

Aceste redirecționări se fac cu funcția din libraria standard `std::ios::rdbuf()`. Pentru un exemplu de folosire a acestei funcții, consultați pagina <http://www.cplusplus.com/reference/ios/ios/rdbuf/>

Atât `cout` cât și `cerr` și `clog` sunt obiecte de tipul `ostream` (output stream). Această clasă permite setarea unor parametri în funcție de care se va face scrierea propriuzisă în bufferul de ieșire. Pentru o listă completă a acestor membri, accesăti <http://www.cplusplus.com/reference/ostream/ostream/>

2. Funcții standard pentru manipularea șirurilor de caractere

String-urile sunt obiecte reprezentate sub forma unor secvențe de caractere. Clasa `string` standard oferă suport pentru aceste obiecte prin implementarea unor metode specifice de prelucrare a stringurilor formate din caractere reprezentate pe un singur octet. C++ oferă două tipuri de reprezentări pentru șirurile de caractere:

- Reprezentarea tip C unui șir de caractere;

Indexul	0	1	2	3	4	5
Variabila	H	e	l	l	o	\0
Adresa	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    char greeting1[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
    char greeting2[] = "Hello";

    cout << "Greeting message1: ";
    cout << greeting1 << endl;

    cout << "Greeting message2: ";
    cout << greeting2 << endl;

    return 0;
}
```

C++ oferă o gamă variată de funcții pentru manipularea șirurilor care se termină cu '\0':

Funcția	Descriere
<code>strcpy_s(s1, dim, s2);</code>	Copie șirul s2 în s1.

strcat_s(s1, dim, s2);	Concatenează sirurile s1 și s2, prin adăugarea lui s2 la sfârșitul lui s1.
strlen(s1);	Returnează lungimea lui s1.
strcmp(s1, s2);	Returnează 0 dacă sirurile s1 și s2 sunt identice, o valoare negativă dacă s1<s2 și o valoare pozitivă dacă s1>s2.
strchr(s1, ch);	Returnează un pointer la prima apariție a caracterului ch în sirul s1.
strstr(s1, s2);	Returnează un pointer la prima apariție a sirului s2 în sirul s1.

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

int main()
{
    char greeting1[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
    char greeting2[] = "Hello";
    cout << "Greeting message1: " << greeting1 << endl;
    cout << "Greeting message2: " << greeting2 << endl;

    cout << "\nLungimea sirului " << greeting1 << " este " << strlen(greeting1) << endl;

    char str[200];
    strcpy_s(str, 200, "Acesta ");
    strcat_s(str, 200, "siruri ");
    strcat_s(str, 200, "sunt ");
    strcat_s(str, 200, "concatenate.");
    cout << str << endl;

    cout << "\n Sirul " << greeting1 << " comparat cu " << greeting2 << " prin
strcmp returneaza " << strcmp(greeting1, greeting2) << endl;

    char str1[] = "Acesta este un sir de caractere de test";
    char * pch;
    cout << "\nSe cauta caracterul s in sirul \" " << str1 << " \" " << endl;
    pch = strchr(str1, 's');
    while (pch != NULL)
    {

```

```

cout << "gasit la " << pch-str1+1<< endl;
pch = strchr(pch + 1, 's');
}

pch = strstr(str1, "sir");
strcpy_s(pch, 10, "fragment");
cout << "\n" << str1 << endl;

return 0;
}

```

- Clasa String din biblioteca <string>

Principalele metode implementate în cadrul acestei clase sunt:

- constructori: constructor fără parametri, constructor de copiere, constructori cu parametri (pe bază de subșiruri, pe bază de c-string-uri - secvențe de caractere terminate printr-un null, pe bază de buffer, constructor de umplere/ocupare, pe bază de gamă de valori, pe bază de listă de initializare, constructor de transfer);

```

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

int main()
{
    string s0("Sirul initial");

    // constructorii clasei String
    //constructor fara argumente
    string s1;

    //constructor de copiere
    string s2(s0);

    //constructori cu parametri
    // din s0 sunt preluate primele 3 caractere incepand cu pozitia 6
    string s3(s0, 6, 3);
    // din sirul "Secventa de caractere" sunt preluate primele 6
    string s4("Secventa de caractere", 6);
    // este preluat sirul dat ca parametru
    string s5("O alta secventa de caractere");
    // este multiplicat caracterul x de 10 ori
    string s6a(10, 'x');
    // este multiplicat caracterul * de 10 ori
    string s6b(10, 42);           // 42 = cod ASCII('*')
    // este preluat subsirul din s0 dintre pozitiile 0 si 5
    string s7(s0.begin(), s0.begin() + 5);

```

```

cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: " << s3;
cout << "\ns4: " << s4 << "\ns5: " << s5 << "\ns6a: " << s6a;
cout << "\ns6b: " << s6b << "\ns7: " << s7 << '\n';
return 0;
}

```

- destructor;
- operatorul de asignare.
- iteratori;

Metoda **begin()** returnează un iterator care pointează spre începutul şirului de caractere. Dacă şirul de caractere este definit de tip constant, atunci metoda returnează un **const_iterator**; altfel returnează un **iterator**. Ambele tipuri de iterator sunt de tip **random access iterator**.

Metoda **end()** returnează un iterator care pointează după terminatorul de şir şi este recomandat ca acesta să nu fie dereferențiat. Metoda este folosită adeseori împreună cu **begin()** pentru a specifica un anumit interval în cadrul şirului de caractere. Dacă obiectul este un şir vid, atunci funcția returnează același lucru ca și **begin()**.

```

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

int main()
{
    string str("String de test");
    for (string::iterator it = str.begin(); it != str.end(); ++it)
        cout << *it;
    cout << '\n';

    return 0;
}

```

- funcții pentru determinarea dimensiunii unui şir;

Funcțiile **size()** și **length()** sunt sinonime și returnează dimensiunea unui şir în octeți, corespunzând conținutului stringului, nefiind neapărat egală cu capacitatea sa maximă de stocare.

Funcția **resize()** este folosită pentru a redimensiona un string la lungimea de n caractere. Dacă valoarea n este mai mică decât dimensiunea actuală a şirului, aceasta va fi scurată, şirul redimensionat va cuprinde doar primele n caractere, restul conținutului fiind eliminat. Dacă n este mai mare decât dimensiunea şirului curent, atunci pozițiile suplimentare din şirul redimensionat vor putea fi completate cu un caracter sau cu caracterul null, în funcție de lista de parametri.

Funcția **clear()** șterge conținutul unui string, acesta devenind vid (lungime=0).

Funcția **empty()** verifică dacă un string este vid(lungimea sa este 0). Returnează true dacă stringul este vid. Funcția nu alterează conținutul stringului.

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

int main()
{
    string str("String de test");
    cout << "Dimensiunea stringului str este de " << str.size() << " bytes.\n";
    cout << "Dimensiunea stringului str este de " << str.length() << " bytes.\n";

    string str1("Imi place sa scriu cod in C");
    cout << str1 << '\n';
    unsigned sz = str1.size();
    str1.resize(sz + 2, '+');
    cout << str1 << '\n';
    str1.resize(14);
    cout << str1 << '\n';

    char c;
    string str2;
    cout << "Introduceti textul pe linii. Introduceti caracterul (.) pentru a finaliza:\n";
    do {
        c = cin.get();
        str2 += c;
        if (c == '\n')
        {
            cout << str2;
            str2.clear();
        }
    } while (c != '.');

    string content;
    string line;
    cout << "Introduceti textul. Introduceti o linie goala pentru a finaliza:\n";
    do {
        getline(cin, line);
        content += line + '\n';
    } while (!line.empty());
    cout << "Textul introdus este:\n" << content;

    return 0;
}
```

- funcții pentru accesarea elementelor unui șir;

Operatorul **[pos]** returnează referința la caracterul de pe poziția pos din cadrul stringului. Dacă pos este egală cu dimensiunea stringului și stringul este definit de tip constant, funcția returnează o referință la caracterul null '\0'. Stringul este indexat de la 0.

Metoda **at(pos)** returnează referință la caracterul de pe poziția pos din cadrul stringului. Funcția verifică automat dacă pos este o poziție validă (pos<dimensiunea stringului) și generează o excepție out of range în caz contrar.

Funcția **back()** este folosită pentru a accesa ultimul caracter dintr-un string. Funcția **front()** este folosită pentru a accesa primul caracter dintr-un string. Aceste funcții nu trebuie apelate pentru stringuri vide.

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

int main()
{
    string str("String de test");
    for (int i = 0; i<str.length(); ++i)
    {
        cout << str[i];
    }

    for (unsigned i = 0; i<str.length(); ++i)
    {
        cout << str.at(i);
    }

    str.back() = '!';
    cout << str << '\n';

    str.front() = 'T';
    cout << str << '\n';

    return 0;
}
```

- funcții supraîncărcate.

Operatorul +

Operatorii relaționali

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str1 = "Hello";
    string str2 = "World";
    string str3;
    int len;
```

```

// copie str1 in str3
str3 = str1;
cout << "str3 : " << str3 << endl;

// concateneaza str1 si str2 si depune rezultatul in str3
str3 = str1 + str2;
cout << "str1 + str2 : " << str3 << endl;

// returneaza lungimea sirului concatenat
len = str3.size();
cout << "str3.size() : " << len << endl;

return 0;
}

```

3. Iteratorul

Iteratorul este un obiect care reținând adresa unui anumit element dintr-un set de elemente cum ar putea fi de exemplu un array are capacitatea de a parurge elementele setului folosind un set de operatori cuprinzând minim un operator de incrementare `++` și un operator de derefențiere `*`. Pointerul este și el o formă de iterator, dar nu toți iteratorii îndeplinesc aceleași funcționalități ca și pointerii.

```

#include <iostream>
#include <string>

using namespace std;

int main()
{

    cout << "Introduceti primul sir: ";
    string sir1;
    cin >> sir1;

    string::iterator si;
    for (si = sir1.begin(); si != sir1.end(); si++)
        cout << *si << endl;

    cout << "Introduceti al doilea sir: ";
    string sir2;
    cin >> sir2;

    sir1 += sir2;
    cout << sir1 << endl;
    string c("aaa");
    size_t found;

    found = sir1.find(c);
}

```

```

    if (found != string::npos)
        cout << "first " << c << " found at: " << int(found) << endl;
    else
        cout << "sirul nu a fost gasit" << endl;

    return 0;
}

```

5. Clasa Vector

Vectorii din biblioteca `<vector>` sunt structuri de date secvențiale ce utilizează spații de memorie continue pentru elementele lor. Elementele unui vector pot fi accesate utilizând anumite deplasamente (vezi aritmética pointerilor). Vectorii își pot schimba în mod dinamic dimensiunea. Ocupă mai multă memorie comparativ cu tablourile uni-dimensionale alocate static pentru a putea administra într-un mod eficient stocarea elementelor.

```

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

int main()
{
    vector<int> myVector;

    myVector.push_back(5);
    myVector.push_back(6);
    myVector.push_back(7);

    for (int j = 0; j < myVector.size(); j++)
        cout << myVector[j] << " ";
    cout << endl;

    //afisarea unui vector folosind iteratori
    vector<int>::iterator i;
    for (i = myVector.begin(); i != myVector.end(); i++)
        cout << *i << " ";

    return 0;
}

```

4. Metoda sort din biblioteca `<algorithm>`

Headerul `<algorithm>` definește o colecție de funcții special create pentru a fi aplicate pe șiruri de elemente. Un șir este definit ca o secvență oarecare de obiecte ce pot fi accesate prin intermediul iteratorilor sau a pointerilor, cum sunt de exemplu array-urile și vectorii. Algoritmii

operează în mod direct prin intermediul iteratorilor asupra valorilor și nu afectează în nici un fel structura containerului (dimensiunea sau spațiul alocat acestuia).

```
template <class RandomAccessIterator>
void sort (RandomAccessIterator first, RandomAccessIterator last);
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare
comp);
```

Metodele **sort(first, last)** și **sort(first, last, comp)** sortează elementele din subșirul specificat prin doi iteratori (first, last) în ordine crescătoare. Elementele sunt comparate folosind operatorul `<` pentru primul exemplu și funcția `comp` pentru cel de-al doilea. Parametrii **first** și **last** sunt iteratori de acces aleator la pozițiile inițială și finală în secvența ce urmează să fie sortată. Subșirul pe care se va aplica sortarea va fi delimitat de cei doi iteratori. Iteratorii trebuie să pointeze către un tip de date pentru care interschimbarea de elemente să fie definite. Parametrul **comp** este o funcție binară care primește ca argumente două elemente din intervalul specificat și returnează o valoare convertibilă la tipul `bool`. Valoarea returnată indică ordinea celor două argumente în sirul sortat.

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

bool functie(int i, int j)
{
    return (i<j);
}

struct clasa{
    bool operator() (int i, int j)
    {
        return (i<j);
    }
} obiect;

int main()
{
    int vi[] = { 32,71,12,45,26,80,53,33 };
    vector<int> v(vi, vi + 8);

    cout << "vectorul v initial:";
    for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        cout << ' ' << *it;
    cout << '\n';

    sort(v.begin(), v.begin() + 4);
    sort(v.begin() + 4, v.end(), functie);
    sort(v.begin(), v.end(), obiect);

    cout << "vectorul v dupa sortare:";
    for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
```

```

        cout << ' ' << *it;
        cout << '\n';

    return 0;
}

```

5. Probleme propuse

6.1.

- a) Scrieți o funcție care primește ca parametru un string și returnează numărul de litere mari (uppercase). Parcurgeți sirul de caractere utilizând indecsăi.
- b) Scrieți o funcție care primește ca parametru un string și returnează numărul de cifre. Parcurgeți sirul de caractere utilizând iteratori.

6.2. Fie următoarea clasă:

```

class StudentAC
{
    string nume
    int nota;

public:
    StudentAC();
    StudentAC(string nume, int nota);
    void afisare();
    void modificareNota(int nouaNota);
};

```

- a) Completați clasa cu definițiile metodelor deja declarate.
- b) Scrieți și testați o funcție care primește ca parametru un vector< > de studenți și afișează informațiile fiecărui student din vector. Parcurgeți vectorul folosind iteratori.
- c) Scrieți și testați o funcție care primește ca parametru un pointer la StudentAC care reprezintă un vector clasic, alocat dinamic. Funcția returnează un vector< > din std de studenți, cu același conținut ca și vectorul primit ca parametru.
- d) Supraîncărcați operatorul de comparare din clasa StudentAC și creați un vector< > de studenți pe care ulterior să îl sortați în ordinea ascendentă a notelor, utilizând funcția sort din std.

7. Bibliografie

<http://www.cplusplus.com/reference/cstring/>

<http://www.cplusplus.com/reference/string/string/>

<http://www.cplusplus.com/reference/vector/vector/>

<http://www.cplusplus.com/reference/iterator/iterator/>

<http://www.cplusplus.com/reference/algorithm/>

POO - C++ - Laborator 12

Cuprins

1.	Proiectul meniu consolă.....	1
2.	Biblioteca de clase “meniu.h”	1
3.	Clasa ElementMeniu	1
4.	Clasa Operatie.....	3
5.	Clasa Meniu.....	3
6.	Clasele operații.....	6
7.	Funcția main()	6
8.	Clasa Lista.....	8
9.	Meniul listei.....	9
10.	Exerciții.....	10

1. Proiectul meniu consolă

În cadrul acestui laborator este prezentat un program care realizează un meniu arborescent la consolă, oarecum similar cu meniul principal al aplicațiilor windows. Partea centrală a programului este o bibliotecă de clase pe baza căreia poate fi creat un meniu propriu. Biblioteca demonstrează utilizarea noțiunilor de POO învățate până în prezent. La sfârșit, sunt propuse câteva exerciții, care presupun realizarea unor meniuri proprii.

În continuare vom crea un proiect pe baza codului din arhiva atașată.

Să executăm programul obținut și să studiem funcționarea lui.

2. Biblioteca de clase “meniu.h”

Header-ul “meniu.h” declară o ierarhie de 3 clase. Clasa de bază este ElementMeniu, care reprezintă orice element de meniu – atât meniul principal, un submeniu sau o operație finală. Toate elementele meniu concrete sunt clase derivate direct sau indirect din ElementMeniu.

Din ElementMeniu sunt derivate direct clasele Meniu și Operatie. Meniu reprezintă meniul principal sau un submeniu. Iar Operatie – o operație finală.

3. Clasa ElementMeniu

...

```

class ElementMeniu {
private:
    char *nume;
    ElementMeniu *parinte;

protected:
    ElementMeniu(char *nume);
    virtual ~ElementMeniu();
    void afisareIncompletaTitlu();
    void afisareTitlu();

public:
    char *getNume();
    virtual void executa() = 0;
    friend class Meniu;
};

...

```

Clasa conține membrii comuni pentru toate elementele.

Câmpurile clasei:

- nume – numele elementului, afișat atunci când se ajunge la acel element.
- parinte – elementul părinte. null dacă este meniul principal.

Constructorul permite crearea unui ElementMeniu pe baza numelui.

Câteva metode:

- afisareIncompletaTitlu() – afișează titlul tuturor părinților, de la rădăcina până la elementul curent, separate prin săgeată "->":

```

void ElementMeniu::afisareIncompletaTitlu() {
    if (parinte != NULL) {
        parinte->afisareIncompletaTitlu();
        cout << " -> " << nume;
    } else {
        cout << nume;
    }
}

```

- afisareTitlu() – afișează titlul aşa cum apare pe prima linie de ecran, când ne aflăm în elementul curent. Implementarea este bazată pe afisareIncompletaTitlu() :

```

void ElementMeniu::afisareTitlu() {
    afisareIncompletaTitlu();
    cout << ":" << endl << endl;
}

```

Acste 2 funcții au modul acces `protected` pentru că sunt apelate din funcția `executa()` a claselor deriveate. Apelarea lor din exteriorul ierarhiei nu este necesară.

- `executa()` – este o metodă virtuală pură. În clasele deriveate conține logica ce este executată atunci când utilizatorul selectează meniul curent.

4. Clasa Operatie

```
class Operatie : public ElementMeniu {  
protected:  
    Operatie(char *nume);  
  
    /*operatia specifica acestui element  
     virtual void execOperatie() = 0; */  
  
public:  
    /*intreaga logica a elementului - afisare  
     titlu + operatie */  
    void executa();  
};
```

Constructorul clasei preia argumentul nume și îl transmite constructorului clasei de bază:

```
Operatie::Operatie(char *nume)  
: ElementMeniu(nume) {}
```

Funcția `executa()` realizează partea comună tuturor operațiilor – șterge ecranul, afișează poziția curentă în meniu, care apare pe prima linie a ecranului, și apelează `execOperatie()` pentru a-i da posibilitatea clasei derivate să-și realizeze logica:

```
void Operatie::executa() {  
    clrscr();  
    this->afisareTitlu();  
    this->execOperatie();  
}
```

- `execOperatie()` – funcția virtuală pură va conține în clasele derivate logica specifică operației.

Rostul de a avea 2 funcții – `executa()` și `execOperatie()` în loc să avem doar `executa()` a fost de a **reutiliza codul**. Logica comună tuturor operațiilor este plasată în funcția `executa()`, iar partea diferită – în funcția `execOperatie()` a claselor derivate.

5. Clasa Meniu

```
class Meniu : public ElementMeniu {  
private:  
    static const int nrMaxElemente = 9;  
    int nrElemente;  
    ElementMeniu **elemente;  
  
    void afisare();  
    int citireComanda();  
  
public:  
    Meniu(char *nume);  
  
    //va dealoca fii sai, eventual recursiv.  
    ~Meniu();
```

```

    void adaugaElement(ElementMeniu *element);
    void executa();
};


```

Reamintim, clasa `Meniu` reprezintă un meniu principal sau un submeniu. Clasa conține o listă de referințe către alte elemente de tip `ElementMeniu`.

Câmpuri:

- `nrMaxElemente` – câmp constant, numărul maxim de elemente a meniului. Întrucât spre elemente se navighează cu tastele 1-9, numărul lor maxim este 9.
- `nrElemente` – numărul de elemente
- `elemente` – vector de pointeri către elementele meniului. Deoarece tipul pointerilor este `ElementMeniu`, elementele pot fi de orice tip derivat din `ElementMeniu`, atât submeniuri cât și operații.

Construtor:

```

Meniu::Meniu(char *nume) : ElementMeniu(nume) {
    this->elemente = new ElementMeniu*[nrMaxElemente];
    this->nrElemente = 0;
}

```

Parametrul `nume` este transmis constructorului clasei de bază. În corpul constructorului se initializează câmpurile clasei.

Destructor:

```

Meniu::~Meniu() {
    for (int i=0; i<nrElemente; i++) {
        delete elemente[i];
    }
    delete[] elemente;
}

```

Până în prezent în clasele pe care le-am studiat, în destructor s-au dealocat doar acele zone de memorie care au fost alocate în cadrul clasei, de obicei în constructor. Însă în clasa `Meniu` ne este mai simplu să dealocăm și elementele meniu referite, chiar dacă acestea au fost alocate în altă parte. Aceasta pentru că elementele meniu nu au nici o utilitate în afara meniului principal, și trebuie dealocate odată cu el.

Destructorul clasei `Meniu` își va dealoca elementele, eventual recursiv, și memoria dinamică alocată.

Unele funcții:

- `citireComanda()` – execută o buclă infinită în care se citește o tastă. Dacă tastă este validă se ieșe din funcție și se returnează codul corespunzător acțiunii selectate, în caz contrar bucla se repetă.

```

/*returneaza indicele elementului activat, sau -1 pentru
iesire*/
int Meniu::citireComanda() {
    while (1) {
        char ch;
        cout << "Introduceti comanda:";
        ch = getch();
    }
}

```

```

        cout << endl;
        if (ch > '0' &&
            (ch - '0') <= this->nrElemente) {
            //element meniu
            int comanda = ch - '1';
            /*pt '1' va fi elementul 0*/
            return comanda;
        } else if (ch == '0' || ch == 0x1B) {
            /*0 sau ESC - iesire din meniu
            return -1;*/
        } else {
            //tasta invalida
            cout << "Tasta invalida: " << ch
            << " Tastele valide sunt '0' - ''"
            << nrElemente << endl << endl;
        }
    }
}

```

- `adaugaElement(ElementMeniu *element)` – adaugă un element. În element, părintele este setat să fie obiectul curent – `this`.

```

void Meniu::adaugaElement(ElementMeniu *element) {
    nrElemente++;
    elemente[nrElemente - 1] = element;
    element->parinte = this;
}

```

Pentru a putea executa instrucțiunea:

```
element->parinte = this;
```

a fost nevoie de a declara clasa `Meniu` prietenă a clasei `ElementMeniu`. Vedeti ultima linie a declaratiei clasei `ElementMeniu` în `meniu.h`. A fost necesar pentru a putea accesa câmpul `element->parinte` din clasa `Meniu`.

- `executa()` – conține o buclă infinită în care se afișează meniul, se citește o comandă și se execută elementul selectat. Din funcție se ieșe atunci când comanda citită este -1 – ieșirea din meniu.

```

void Meniu::executa() {
    for (;;) {
        int comanda;

        clrscr();
        afisareTitlu();
        afisare();
        comanda = citireComanda();
        if (comanda >=0 &&
            comanda < nrElemente) {
            elemente[comanda]->executa();
        } else {
            //probabil iesirea - -1
            return;
        }
    }
}

```

```
}
```

6. Clasele operații

O operație din meniu este o clasă derivată din `Operatie`. Pentru a le separa de clasele din biblioteca, operațiile au fost definite în fișiere separate – `operatiiSimple.h` și `operatiiSimple.cpp`. De exemplu, clasa `OperatieAdunare`:

`operatiiSimple.h`

```
...
class OperatieAdunare : public Operatie {
public:
    OperatieAdunare(char *nume);
    void execOperatie();
};

...
```

`operatiiSimple.cpp`

```
...
OperatieAdunare::OperatieAdunare(char *nume)
: Operatie(nume) {}

void OperatieAdunare::execOperatie() {
    int a, b;
    cout << "Introduceti 2 numere:";
    cin >> a >> b;
    cout << "suma = " << a + b << endl;
    pauza();
}
...
```

Tot ce trebuie să facem într-o operație simplă este să definim constructorul, și să implementăm funcția virtuală `execOperatie()`. La sfârșitul lui `execOperatie()` s-a apelat `pauza()` pentru a da posibilitatea utilizatorului să vadă rezultatul, înainte să se întoarcă în meniu.

7. Funcția main()

În `main()` este construit arborele meniului, apoi este executat meniul principal.

`meniuMain.cpp`

```
#include<iostream>
#include "globale.h"
#include "meniu.h"
#include "operatiiSimple.h"
using namespace std;

int main() {
```

```

Meniu *meniu =
    new Meniu("Meniu Principal");

Meniu *submeniuCalculator =
    new Meniu("Calculator");
meniu->adaugaElement(submeniuCalculator);
submeniuCalculator->adaugaElement(
    new OperatieAdunare("+'"));
submeniuCalculator->adaugaElement(
    new OperatieScadere("'-"));

meniu->adaugaElement(
    new Meniu("Meniu vid"));
meniu->adaugaElement(
    new ElementDespre("Despre program"));

meniu->executa();

/* intreg arborele de elemente va fi dealocat
recursiv*/
delete meniu;

cout << endl << endl
    << "Sfarsit." << endl;
pauza();
return 0;
}

```

Pe prima linie este instanțiat meniu principal:

```
Meniu *meniu = new Meniu("Meniu Principal");
```

În continuare la meniu sunt adăugate elementele:

```

Meniu *submeniuCalculator = new Meniu("Calculator");
meniu->adaugaElement(submeniuCalculator);
submeniuCalculator->adaugaElement(
    new OperatieAdunare("+'"));
submeniuCalculator->adaugaElement(
    new OperatieScadere("'-"));

meniu->adaugaElement(new Meniu("Meniu vid"));
meniu->adaugaElement(
    new ElementDespre("Despre program"));

```

Ca să adăugăm un element trebuie să-l instanțiem și să-l adăugăm la părintele său, apelând funcția `adaugaElement()` din părinte:

```
submeniuCalculator->adaugaElement(new
OperatieAdunare("+'"));
```

Pornirea meniului principal se face apelând:

```
meniu->executa();
```

Atât timp cât programul se va afla în meniu, ne vom afla în această funcție.

La sfârșit, meniul principal este dealocat, iar destructorul său dealocă recursiv tot arborele de elemente:

```
delete meniu;
```

8. Clasa Lista

Mai jos este prezentată clasa `Lista`, care reprezintă o listă de numere. Clasa urmează să fie utilizată într-un meniu. Suportă operația de adăugare a unui număr, prin intermediu operatorului `+=`, ștergere a unui număr după index, și afișarea listei. Fiecare număr are un index, începând de la 0, la fel ca un vector. Codul listei este deja adăugat la proiectul meniuri.

lista.h

```
#ifndef _lista_
#define _lista_

/* reprezinta o lista de numere. Fiecare numar are un
index, incepand de la 0, la fel ca intr-un vector.*/
class Lista {
private:
    int *elem;
    int n, dim;
public:
    Lista(int dim);
    ~Lista();
    void operator+=(int num);
    void sterge(int index);

    // afiseaza numerele si indecsii lor
    void afisare();
};

#endif
```

lista.cpp

```
#include<iostream>
#include "lista.h"
using namespace std;

Lista::Lista(int dim) {
    this->dim = dim;
    elem = new int[dim];
}

Lista::~Lista() {
    delete[] elem;
}

void Lista::operator+=(int num) {
    if (n == dim) {
        cout << "Lista plina" << endl;
    } else {
        elem[n] = num;
    }
}
```

```

        n++;
    }

void Lista::sterge(int index) {
    for(int i=index; i<n-1; i++) {
        elem[i] = elem[i+1];
    }
    n--;
}

void Lista::afisare() {
    for(int i=0; i<n; i++) {
        cout << i << ". " << elem[i] << endl;
    }
}

```

9. Meniul listei

Ne propunem sa creăm un meniu care să gestioneze o listă de numere reprezentată de `Lista`. Avem nevoie de minim 2 operații – una care să adauge un numar la listă, și alta care să afișeze lista. De data aceasta clasele operații sunt mai complexe. Ele trebuie să aibă acces la obiectul listă. Iar lista trebuie să fie aceeași pentru ambele operații. Singurul mod elegant în care putem realiza acest lucru este să avem în clasele operații un câmp de tip pointer la `Lista`. și ambele operații să refere prin intermediul pointerului aceeași listă.

`Lista` nu poate fi instanțiată în nici una din clasele operații. Vom instanția lista cu un nivel mai sus – în funcția `main()`. și o vom transmite operațiilor prin intermediul unui parametru la constructor.

Mai jos este prezentat codul celor 2 operații:

lista.h

```

...
class OpAdaugaInLista : public Operatie {
private:
    Lista *lista;
public:
    OpAdaugaInLista(char *nume, Lista *lista);
    void execOperatie();
};

class OpAfisareLista : public Operatie {
private:
    Lista *lista;
public:
    OpAfisareLista(char *nume, Lista *lista);
    void execOperatie();
};
...

```

lista.cpp

```

...
OpAdaugaInLista::OpAdaugaInLista(char *nume, Lista

```

```

*lista)
: Operatie(nume) {
    this->lista = lista;
}

void OpAdaugaInLista::execOperatie() {
    int num;
    cout << "num=";
    cin >> num;
    (*lista)+=num;
    pauza();
}

OpAfisareLista::OpAfisareLista(char *nume, Lista *lista):
Operatie(nume) {
    this->lista = lista;
}

void OpAfisareLista::execOperatie() {
    lista->afisare();
    pauza();
}

```

Adăugați cele 2 clase în program. Completăți funcția `main()` cu codul necesar pentru a crea un submenu al listei cu cele 2 operații. Observați că ambele clase operații conțin un câmp pointer la listă. Obiectul `Lista` trebuie creat și dealocat în `main()`.

Rulați programul rezultat.

10. Exerciții

1. Implementați o operație pentru ștergerea unui element din listă după index.
2. Adăugați la listă operatorul de indexare – `[]`. Operatorul va servi pentru accesarea unui element după index. Implementați în meniu operația de afișare a unui element după index.
3. Implementați un meniu pentru 2 liste. Redefiniți în clasa `listă` operatorul `+=`, astfel încât să accepte un operand dreapta de tip `Lista&`. Instrucțiunea `list1+=list2` va adăuga toate elementele din `list2` în `list1`. Implementați o operație meniu care să testeze acest operator.
 - `afisare()` – afișare multime.
4. Implementați un meniu care să testeze clasa `Multime` de la exercițiul 1 din lab. 3.