

LABORATORUL 1

NOTIUNI RECAPITULATIVE DE C

1. SCOPUL SEDINTEI DE LABORATOR

Se dorește revederea unor elemente, aparținând standardului C, studiate în cadrul disciplinelor anterioare (*PCLP I*, *PCLP II*), cum ar fi: secvențe escape, directive de preprocesare și funcții cu număr variabil de argumente.

Ținând cont de faptul că o formă slabă de polimorfism, notiunea ce va fi studiată în continuare, este deja prezentă în cadrul limbajului C, s-a considerat utilă reanaliza notiunii de funcție cu număr variabil de parametri.

În final, s-a dorit introducerea unui alt mod de a întreține proiectele, utilitarul MAKE, modalitate promovată și de către alte platforme de dezvoltare, în cazul aplicațiilor scrise în C/C++ și formate din mai multe fișiere sursă.

1.1. OBIECTIVE MINIME, NECESARE PENTRU NOTA 5

- Asimilarea notiunii de secvență escape, notiune aparținând standardului C și utilizarea secvențelor escape comune (`\n`, `\t`, `\a`), în programe simple, realizate în limbajul C;
- Utilizarea corectă a constantelor simbolice și a macrodefinițiilor
- Realizarea temei numerotate cu 1, de la sfârșitul lucrării de laborator

1.2. OBIECTIVE MINIME, NECESARE PENTRU NOTA 10

- Asimilarea notiunii de secvență Escape, notiune aparținând standardului C și utilizarea tuturor secvențelor escape în programe realizate în limbajul C;
- Revizuirea notiunilor privind directivele de preprocesare și utilizarea corectă a acestora;



- Revizuirea notiunii de functie cu numar variabil de parametric si definirea mai multor protocoale privind modul de transmitere a parametrilor catre acestea;
- Intretinerea proiectelor folosind utilitarul MAKE;
- Realizarea tuturor temelor, de la sfarsitul lucrarii de laborator

1.3. COMPETENTE MINIME, DOBANDITE PENTRU NOTA 5

- Stapanirea notiunii de secventa escape, notiune apartinand standardului;
- Utilizarea corecta a constantelor simbolice si a macrodefinitiiilor;
- Crearea de aplicatii simple in limbajul C, utilizand Borland C++ 3.1 cat si Code::Blocs.

1.4. COMPETENTE MINIME, DOBANDITE PENTRU NOTA 10

- Operarea cu majoritatea elementelor specifice limbajului C: secvente escape, directive de preprocesare, functii cu numar variabil de parametri, etc;
- Crearea de aplicatii complexe in limbajul C, utilizand Borland C++ 3.1 cat si Code::Blocs.
- Intretinerea proiectelor utilizand aplicatia make; creare de fisiere makefile cu diferite tinte, pentru intretinerea proiectului cat si pentru operatii adiacente.

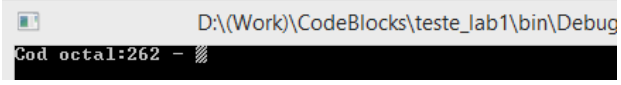
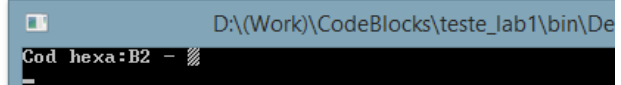
2. BAZELE TEORETICE

Notiuni recapitulative de C precum adăugirile aduse standardului ANSI C, poate mai puțin cunoscute.

2.1. SECVENȚE ESCAPE

SECVENTA	NUME	INTERPRETARE SAU ACTIUNE
\a	Alertă	Emite semnal sonor



\b	Backspace	Mută cursorul înapoi cu un spațiu
\f	Form feed	Mută cursorul la începutul paginii următoare
\n	New line	Mută cursorul la începutul liniei
\r	Carriage return	Mută cursorul la începutul rândului curent
\t	Tab orizontal	Tab orizontal
\v	Tab vertical	Tab vertical
\\, \', \", \?	Caractere speciale	Tipărește caracterul de după \
\<octal> \262	Constanta în baza 8	Depinde de imprimantă sau terminal <pre>printf("Cod octal:262 - \262\n");</pre> 
\x<hexa> \xb2	Constanta în baza 16	Depinde de imprimantă sau terminal <pre>printf("Cod hexa:B2 - \xB2\n");</pre> 

2.2. DIRECTIVE DE PREPROCESARE

Sunt acele secvențe de program ce sunt prefixate de simbolul #

2.1.1. MACRODEFINIȚII



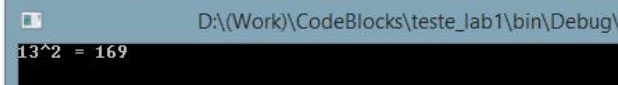
```
#define square( x ) ( ( x ) * ( x ) )
```

```
int main()
```

```
{
```

```
    printf("13^2 = %d", square(13) );
```

```
}
```



2.1.2. OPERATORUL DE CONCATENARE A SIMBOLURILOR (*TOKEN-PASTING*)

Aadaugă un simbol la altul creând un al treilea simbol disponibil.

```
#define TYPE1      0x4d4d  
#define TYPE2      0x4949  
#define TYPE( n )  TYPE##n
```

Prin utilizarea `TYPE(1)` ne referim la `TYPE1`, iar prin `TYPE(2)`, ne referim la constanta `TYPE2`.

2.1.3. OPERATORUL DE CONVERSIE LA UN ȘIR DE CARACTERE (*STRING-SIZING OPERATOR*)

Acest operator transformă în șir de caractere orice parametru precedat de caracterul `#` scriind parametrul între ghilimele.

```
#define Trace(x) printf("#x" = %d\n", x)
```

Utilizarea se face sub forma:

```
Trace( flag );
```

Efectul fiind:

```
flag = 3
```

de exemplu, unde `flag` ar fi numele unei variabile, având valoarea 3 în momentul apelării macrocomenzii. Am folosit `printf(..)` deoarece ne referim la standardul ANSI C. Convențiile rămân valabile și pentru standardul C++.

2.1.4. DIRECTIVELE CONDIȚIONALE DEJA CUNOSCUTE

Directivele conditionale sunt prezentate mai jos:

<code>#if</code>
<code>#ifdef</code>
<code>#ifndef</code>

<code>#else</code>
<code>#elif</code>
<code>#endif</code>

Semnificația lor este ușor de depistat, în continuare dând un exemplu de utilizare:

```
#if CPU_TYPE == 80286
    #include < mode_16.h >
#elif CPU_TYPE == 80386
    #include < mode_32.h >
#else
    #error Procesor de tip necunoscut
#endif
```

Directiva `#error` are ca efect tipărirea mesajului în faza de compilare (mai curând precompilare), în fereastra de erori (se generează o eroare cu textul directivă specificat după `#error`, până la sfârșitul liniei).

Exemplu

```
//-----fisierul header.h -----
#ifndef _HEADER.H_
#define _HEADER.H_
#include <math.h>

    //exemplu definire constanta
    #define NMAX 10

    //define HEXA
    #define OCTAL
    #define DEBUG

    //exemplu macrodefinitie
    #define radical(x) sqrt(x)

    //exemplu operator de conversie la sir de caractere
    #if defined( HEXA )
        #ifdef DEBUG
            #define afisval(x) printf("#x"=%x Adresa=%p",x,&x)
        #else
            #define afisval(x) printf("#x"=%x",x)
        #endif
    #elif defined(OCTAL)
        #ifdef DEBUG
            #define afisval(x) printf("#x"=%o Adresa=%p",x,&x)
        #else
            #define afisval(x) printf("#x"=%o",x)
```



```
        #endif
    #else
        #ifdef DEBUG
            #define afisval(x) printf(#x"=%d Adresa=%p",x,&x)
        #else
            #define afisval(x) printf(#x"=%d",x)
        #endif
    #endif
#endif
```

```
//-----fisierul main.c -----
#include "header.h"
#include <stdio.h>
void main(void)
{
    int val=20;
    unsigned space;
    printf( "NMAX=%d\n", NMAX );
    printf( "radical(20)=%f\n", radical(20) );
    afisval( val );

    //Determinarea spatiului de memorie neutilizat
    #if defined( NEARPOINTERS )
        space = farcoreleft();
    #elif defined(FARPOINTERS)
        space = coreleft();
    #endif

    //Daca modul de lucru este DEBUG se afiseaza
    //spatiul de memorie neutilizat
    #ifdef DEBUG
        printf("\nTotal space %d\n", space);
    #endif
}
```

2.3. FUNCȚII CU NUMĂR VARIABIL DE ARGUMENTE

Macrouri pentru funcții cu număr variabil de argumente :

```
void va_start( va_list ap, lastfix );
type va_arg( va_list ap, type );
```



```
void va_end( va_list ap );
```

Aceste macrouri dau posibilitatea utilizatorului de a accesa argumentele unei funcții ce nu are un număr constant (fix) de argumente după cum umează:

- *va_start()* setează variabila *ap* de tip *va_list* la primul argument ce este transmis funcției.
- *va_args()* transformă argumentul curent din lista la tipul *type* și îl întoarce, după care trece la următorul argument din listă.
- *va_end()* ajută funcția să efectueze un *return* normal (să golească stiva înainte de întoarcere).

Parametrul *lastfix* al funcției *va_start()* este de fapt numele ultimului parametru fix transmis funcției.

Exemplu

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

//Sorteaza numerele dintr-o lista de parametrii terminata cu 0
void Sort(char * msg, ...)
{
    va_list ap;
    va_start(ap, msg);
    int arg;
    int v[100];
    int i, j, temp, n = 0;
    printf("\n Message = %s", msg);
    while( (arg = va_arg(ap, int)) !=0 )
    {
        v[n] = arg; n++;
    }
    va_end( ap );

    //Sortarea propriu zisa
    for(i = 0; i < n-1; i++)
    for(j = i+1; j < n; j++)
        if( v[i] > v[j] )
        {
            temp = v[i];
```



```
        v[i] = v[j];
        v[j] = temp;
    }
    for(i = 0; i < n; i++)
        printf("\n v[%d] = %d", i, v[i]);
}

void main( void )
{
    //Doua exemple de apel
    Sort("Numerele sortate: ", 3, 2, 9, 3, 1, 0 );
    Sort("Numerele sortate: ", 11, 2, -3, 4, 21, 44, 12, 0);
}
```

2.4. REALIZAREA PROIECTELOR FOLOSIND UTILITARUL MAKE

Dezvoltarea unor programe de dimensiuni mai mari, într-un limbaj de programare, presupune un stil de lucru diferit de cazul programelor de dimensiuni mici. La programele simple este suficientă crearea unui fișier ce conține codul și apoi, printr-o singură comandă la nivelul procesorului de comenzi sau în cadrul unui mediu integrat, se va obține fișierul executabil.

La programele mai mari lucrurile nu sunt atât de simple, astfel dacă am avea un singur fișier programul ar fi foarte greu de înțeles și în plus trecerea la programul executabil ar necesita un timp mare, întrucât modificarea celui mai mic detaliu ar impune recompilare întregului cod. În plus, proiectele mari sunt întotdeauna rezultatul muncii unei întregi echipe, ceea ce face imperios necesară împărțirea într-un număr mare de fișiere (numite module), codul-sursă și de a compila independent aceste fișiere. Această facilitate se numește compilare separată.

Deși distincte și compilabile separat, diferitele module aparținând aceluiași proiect, de multe ori nu sunt total independente între ele. Numim aceste relații între diferitele module, dependente. Procesul de modularizare are ca scop tocmai detectarea și minimizarea acestor dependențe între module. Aceste dependențe au un impact evident asupra procesului de compilare și anume: atunci când a fost modificat un modul, recompilarea sa implică și recompilarea tuturor modulelor ce depind de el.

2.4.1. UTILITARUL DE ÎNTREȚINERE MAKE, FISIERE "MAKEFILE"



Programul *make.exe* este un utilitar pentru intretinerea proiectelor, iar pentru a-l putea utiliza trebuie sa scrieti un fisier numit *Makefile* (sau *makefile.mak*) care descrie relatiile de dependenta intre diferitele fisiere din care se compune programul, si specifica regulile de actualizare pentru fiecare fisier in parte. In mod normal, intr-un program fisierul executabil este actualizat (recompilat) pe baza fisierelor-obiect, care la randul lor sunt obtinute prin compilarea fisierelor sursa, in care sunt incluse, prin directive de tipul `"#include"`, fisiere header, cu declaratii de variabile si functii.

Programul MAKE utilizeaza fisierul "*makefile*" ca baza de date si pe baza timpilor ultimei modificari a fisierelor din "*makefile*" decide care sunt fisierele ce trebuiesc actualizate. Pentru fiecare din aceste fisiere, declanseaza comenzile precizate in "*makefile*".

Acest utilitar se lanseaza specificându-se si fișierul "*makefile*", un fișier text ce va fi creat după o sintaxă riguroasă. Dacă se lansează MAKE fără *makefile* se consideră că există un fișier "*makefile*" în directorul curent, fișier ce va fi interpretat de către utilitarul MAKE.

Principiul de analiză a utilitarului MAKE este că încearcă să realizeze primul obiectiv "target" stabilit în fișier, iar pentru aceasta va analiza și va reface fișierele dependente "dependent", fișiere ce nu au aceeași dată cu data țintei. Linia ce urmează după dependență descrie modul în care se obține fișierul țintă din cele subordonate.

2.4.2. STRUCTURA UNUI „MAKEFILE”

Un „*makefile*” se poate compune din urmatoarele elemente:

- *Reguli* . O regula defineste *cand* si *cum* trebuie refacute unul sau mai multe fisiere denumite *obiectivele* regulii. O regula enumera fisierele de care depind obiectivele regulii, numite *dependente* si defineste *comenzile* ce trebuie executate pentru crearea sau actualizarea obiectivelor.
- *Declaratii de Variabile (macrocomenzi in MAKE)*. Sunt linii care atribuie unui nume (variabila) un sir de caractere, nume ce va putea fi folosit in continuare pentru a subsitui respectivul text.
- *Directivele* sunt comenzi pentru programul make prin care i se comunica acestui sa face ceva special in timp ce citeste fisierul Makefile, de exemplu sa includa un alt Makefile



sau sa decida bazat pe valoarea unei variabile daca sa ignore sau nu o anumita parte a fisierului Makefile. Nu tratam acest aspect aici.

- *Comentarii.* O linie care incepe cu caracterul diez (#) este o linie de comentariu, continutul ei fiind ignorat.

Observatie

Intr-un Makefile o linie tine pana la intalnirea caracterului CR (, \n'). Totusi o linie poate fi „continuata” si dupa caracterul CR (, \n') prin utilizarea caracterului backslash (\) ca in exemplul de mai sus.

Exemplu

```
Form.obj: form.cpp form.h  
        bcc -c form.cpp
```

In cazul prezentat mai sus obiectivul este numele fisierului “Form.obj”, iar fisierele din care este construit obiectivul sunt specificate pe acelasi rand cu numele obiectivului imediat dupa simbolul “:”. Pe linia urmatoare este definita regula dupa care este construita (reconstruita) tinta. Aceasta regula se aplica doar atunci cand cel putin unul din fisierele constitutive ale tintei a fost modificat. In acest caz este aplicata regula specificata pentru a reactualiza tinta din fisierele dependente.

Observatie

Fiecare comanda trebuie plasata pe o alta linie in fisierul Makefile. Este obligatoriu ca fiecare linie ce contine o comanda sa inceapa cu un *tab*. De obicei o comanda apare intr-o regula cu dependente si serveste la crearea unui fisier-obiectiv daca vreunul din fisierele dependenta sunt modificate.

Cel mai mare avantaj al folosirii utilitarului MAKE este că se evită orice compilare care nu este necesară. MAKE assemblează un fișier țintă doar dacă cel puțin unul din fișierele dependente au dată diferită față de data fișierului țintă.

În plus MAKE tratează tinta ca pe numele unui scop care trebuie atins – ținta nu trebuie să fie neaparat un fișier.



Exemplu

```
clean:
    erase form.obj
    erase form.exe
```

Ținta nu depinde de nimic în acest caz. Instrucțiunile de dependență care urmează cer programului MAKE să apeleze comenzile de ștergere pentru cele două fișiere.

2.4.3. MACROCOMENZI ÎN MAKE

Definirea unei macrocomenzi se face după sintaxa

```
<nume_macrocomanda> = <sir>
BCCFLAGS = -c -ml -v -Od
```

Utilizarea macrocomenzii se face sub forma

```
$( <nume_macrocomanda> )
$(BCCFLAGS)
```

Exemplu

```
BCC = bcc $(BCCFLAGS)
form.obj: form.cpp form.h
    $(BCC) form.cpp
```

2.4.4. MACROCOMENZI PREDEFINITE

- \$*** Numele (cu cale completă) fișierului țintă, fără extensie
- \$@** Calea completă până la fișierul țintă
- **** Toate numele ce apar în lista de dependențe
- \$<** Numele unui fișier subordonat care este expirat (din punct de vedere al datei) față de fișierul țintă.



\$? Toate fişierele subordonate expirate faţă de ţintă

MAKE__ Numărul de versiune pentru MAKE

MAKE Numele fişierului executabil MAKE

2.4.5. REGULI DE DEDUCERE

Utilitarul MAKE permite definirea de “reguli de deducere” (inference rules) – reguli ce stabilesc modul cum un fişier cu o anumită terminaţie poate fi obţinut (generat) dintr-un fişier cu o altă terminaţie. Pentru fişierele cu terminaţiile cunoscute sunt definite reguli interne de deducere.

Exemplu

```
#Obtinerea unui fisier .obj dintr-un fisier .cpp
.cpp .obj:
    bcc -c $<
```

2.4.6. EXECUTIA UNUI MAKEFILE

Intrucat un Makefile contine mai multe reguli, intrebarea care se ridica este: care dintre reguli se va executa la apelul lui make fara nici un parametru? In acest caz se va executa intotdeauna *prima regula intalnita in Makefile*. Din acest motiv la compilarea unui proiect se obisnuieste declararea prima data a regulii prin care se obtine executabilul. Daca se doreste executarea unei anumite reguli se apeleaza make sub forma: make *numele_regulii* ;

Observaţie

În cazul mediului de dezvoltare Borland C++ 3.1, există un utilitar capabil să convertească fişiere proiect (*.prj) în fişiere respectând structura descrisă pentru “*makefiles*”. Acest utilitar este prj2mak.exe, iar sintaxa de apel este dată mai jos:

```
Prj2mak.exe <fisier.prj> <fisier.mak>
```

2.4.7. RULAREA COMPILATORULUI DE C++, BCC



Sintaxa generală de lansare a compilatorului bcc din linia de comandă este:

```
bcc [-optiuni] [fisier1] [fisier2]..
```

2.4.8. RULAREA LINK-EDITORULUI TLINK

Sintaxa generală de lansare a link-editorului tlink din linia de comandă este:

```
tlink.exe <opt> <fis_ob>, <fis_exe>, <fis_map>,<fis_lib>, <fis_def>
```

<opt>:

Optiunile listate prin tastarea comenzii *tlink* la tastatura

<fis_ob>:

Ordinea fisierelor *obj* este critica. Primul va trebui sa fie *c0x.obj* (unde 'x' corespunde la: *s,m,c,l,h* fiecare pentru cate un model de memorie). In continuare este dispusa lista urmatoarelor fisiere obiect.

<fis_exe>:

Aceasta intrare este optionala. Daca este specificat un nume, atunci iesirea va trebui sa aiba denumirea respectiva.

<fis_map>:

Aceasta intrare este optionala. Daca este specificat un nume, atunci fisierul *MAP* de iesirea va trebui sa aiba denumirea respectiva.

<fis_lib>:

Ordinea fisierelor *lib* este critica. Orice librerie creata de catre utilizator va fi introdusa la inceputul listei. Daca programul utilizeaza rutine *BGI*, libraria grafica va trebui sa fie urmatoarea (*graphics.lib*). Urmatoarele in ordine sunt *emu.lib* (pentru emulare) sau *fp87.lib* (daca doresti in mod special sa utilizezi coprocesorul). Libraria *mathx* va veni urmatoarea, unde 'x' corespunde initialei modelului de memorie utilizat. In final *cx.lib* va fi ultima in lista, unde 'x' corespunde initialei modelului de memorie utilizat.

De exemplu, daca utilizezi modelul de memorie *large* si rutine *GBI*, linia de comanda pentru *tlink* va fi:

```
tlink /v c0l.obj myobj.obj,,,mylib.lib graphics.lib emu.lib  
mathl.lib cl.lib
```

sau



```
tlink /v c0l myobj,,mylib graphics emu mathl cl
```

Se va avea în vedere să nu se depășească lungimea liniei de comandă de 127 caractere.

Exemplu

```
# Definitii de constante
CC = bcc
TLINK = tlink
LIBPATH = F:\BORLANDC\LIB
INCLUDEPATH = F:\BORLANDC\INCLUDE

# Reguli implicite
.cpp.obj:
    $(CC) -c -I$(INCLUDEPATH)
{$< }

# Lista macrourilor
EXE_dependencies = main.obj
triunghi.obj

# Reguli de deductie explicite
triunghi.exe: $( EXE_dependencies )
    $(TLINK) /x/c/L$(LIBPATH)
@&&|
```

```
c0s.obj+
main.obj+
triunghi.obj
triunghi

# no map file
graphics.lib+
emu.lib+
maths.lib+
cs.lib
|
# Dependintele fisierelor
individuale
main.obj: main.cpp triunghi.h
triunghi.obj: triunghi.cpp
triunghi.h

del:
    del *.bak
```

3. TEMA

1. Creati o aplicatie ce va implementa un meniu, scrisa in limbajul C, de tip proiect, aplicatie ce va realiza urmatoarele operatii:
 - optiune ce va genera in difuzor un sunet, ori de cate ori se va tasta ENTER, iar la apasarea tastei ESC va parasi optiunea;
 - optiune ce va afisa pe ecran tabelul cu toate secventele ESCAPE studiate.
 - Aplicatia va devini si utiliza cel putin o macrodefinitie
2. Realizați un proiect folosind utilitarul MAKE, avand mai multe tinte:
 - a) Intretinerea unui program (proiect) realizat in limbajul C,



- b) Curatarea directorului curent de fisierele temporare,
 - c) Arhivarea si salvarea continutului intregului proiect
3. Realizati o functie cu numar variabil de parametri, functie ce va afisa minimul si maximul dintr-o lista de valori intregi, terminata cu 0.

LABORATORUL 2

ELEMENTE INTRODUCTIVE DE C++

1. SCOPUL SEDINTEI DE LABORATOR

În cadrul acestui laborator studentul are pentru prima dată contactul cu noțiunea de clasă și obiect, elemente esențiale pentru conceptul de Programare Orientată pe Obiecte (POO).

Tot aici sunt prezentate metodele de acces la datele sau metodele unei clase precum și modul de a declara și defini o funcție membră a unei clase. În același timp va fi prezentat noul operator al limbajului C++, operatorul de rezoluție `::`.

La început se face o scurtă paralelă între noțiune de structură, întâlnită pentru prima dată în limbajul C, și noțiunea de clasă, aparținând standardului C++.

În partea finală a laboratorului este prezentată noțiunea de funcție inline cât și cea de funcție cu parametri având valori implicite.

Din punct de vedere practic este propusă clasa `Data`, clasă ce dorește să definească conceptul de dată calendaristică. Printre membri, clasa definește și metoda `int Data::NrZile(Data dc)`. În acest caz va trebui calculat corect numărul de zile dintre două date calendaristice, ținând



seama de diferenta de zile in fiecare luna cat si de *anul bisect*¹. Problema propusa spre rezolvare este marcata prin "TEMA PE PARCURS - #1"

In finalul lucrarii este propusa o tema noua, pe langa "Tema pe parcurs #1", definirea conceptului de numar complex, ce va trebui finalizata, deoarece analiza problemei cat si rezolvarea acesteia este deja prezentata in cadrul laboratorului.

1.1. OBIECTIVE MINIME, NECESARE PENTRU NOTA 5

- Asimilarea notiunii de clasa, notiune ce apartine si standardului C++, impreuna cu notiunea de obiect/instanta;
- Definirea si utilizarea corecta a datelor si functiilor membre unei clase, utilizarea corecta a operatorului de rezolutie `::`;
- Verificarea clasei Data (data calendaristica), prezentata in cadrul lucrarii si incercarea de corectare a metodei de calcul a numarului de zile dintre doua date calendaristice;
- Analiza materialului furnizat corespunzator temei propuse, de la sfarsitul lucrarii de laborator.

1.2. OBIECTIVE MINIME, NECESARE PENTRU NOTA 10

- Intelegerea notiunii de clasa, notiune ce apartine si standardului C++, impreuna cu notiunea de obiect/instanta;
- Verificarea conceptului de functie inline in paralel cu apelul functiilor ce implementeaza transferul parametrilor prin valoare (mecanismul de stiva)²;

¹ Anul bisect - http://ro.wikipedia.org/wiki/An_bisect

² Cursul 9 PCLP I, pagina 12-13 - http://apollo.eed.usv.ro/~remus/arhive/PCLP_I/cursuri/2009_Curs_9.pdf



- Analiza notiunii de functie cu parametri avand valori implicite si compararea acestui ultim concept cu notiunea de macrodefinitie, din limbajul C;
- Definirea si utilizarea corecta a datelor si functiilor membre unei clase, utilizarea corecta a operatorului de rezolutie ::;
- Verificarea clasei Data (data calendaristica), prezentata in cadrul lucrarii si corectarea metodei de calcul a numarului de zile dintre doua date calendaristice;
- Definirea de scenarii de testare pentru captarea majoritatilor bg-urilor, in utilizarea clasei *Data*;
- Analiza si finalizarea materialului furnizat, corespunzator temei propuse, de la sfarsitul lucrarii de laborator.

1.3. COMPETENTE MINIME, DOBANDITE PENTRU NOTA 5

- intelegerea notiunii de clasa si obiect/instanta;
- analiza si utilizarea corecta a unor clase simple deja definite, in limbajul C++;

1.4. COMPETENTE MINIME, DOBANDITE PENTRU NOTA 10

- intelegerea notiunii de clasa si obiect/instanta;
- intelegerea conceptului de functie inline cat se de functie cu parametri avand valori implicite;
- analiza si utilizarea corecta a unor clase simple deja definite, in limbajul C++, modificarea/upgrade-ul acestor clase;
- definirea de scenarii de testare pentru a "prinde" eventualele bug-uri ce pot aparea in utilizarea claselor definite.



2. BAZELE TEORETICE

2.1. LIMBAJUL C++. CLASE

Noțiunea de clasă introdusă de C++ se apropie de noțiunea de structură dar apar unele diferențe:

- structura are ca membrii doar date în viziune C ;
- clasa conține date și funcții.

Variabilele de tip clasă sunt denumite instanțe (obiecte - ce sunt noțiuni fizice (nu au memorie alocată). Variabilele de tip clasă există în memorie.

2.2. PARALELĂ ÎNTRE STRUCTURĂ ȘI CLASĂ

	STRUCTURI	CLASE
Cuvânt cheie	<code>struct</code>	<code>class</code>
Modul de declarare	<code>struct A {...};</code>	<code>class {...};</code>
Definirea unei variabile membre	<code>struct A a, s[10];</code> <code>A a, s[10];</code>	<code>class B b, s[20];</code> <code>B b, s[20];</code>

Observatie

În cadrul limbajului C++, în momentul declarației de variabile de tip clasă sau structură, nu mai este valabilă apariția cuvintelor *class* respectiv *struct*.



2.3. CLASE SI OBIECTE

Sintaxa generală pentru definirea unei clase:

```
class <ID_clasa> [ : [specificator_acces] <ID_clasa_baza> [...] ]  
{  
    <lista_membrii>  
    - <date>  
    - <metode>  
} [<lista_obiecte>] ;
```

În declarația de mai sus avem următoarele semnificații :

1. ID_clasa - un identificator (numele clasei similar cu structurile)
2. ID_clasa_baza - numele clasei / claselor de bază din care derivă clasa ID_clasa
3. specificator_acces - indică tipul moștenirii ce poate fi împărțită în:
 - moștenire publică
 - moștenire privată
 - moștenire protected
4. lista_membrii - lista tuturor membrilor clasei cu observația că pe lângă date membre întâlnim și funcții membre
5. lista_obiecte – lista variabilelor aparținând clasei respective (a obiectelor)

Exemplu

```
class Finala: public Baza1, private Baza2  
{  
    int x; //data membra de tip private  
public:
```



```
float y; //data membra de tip public
int GetData( void ); //functie membra de tip public
void PutData( int x ); //functie membra de tip public
private:
void Temp( void ); //functie membra de tip private
protected:
char nume[ 20 ]; //data membra de tip protected
} obl, sir_ob[ 10 ], *p_ob; //obiecte/instante/variab. de tip Finala
```

2.4. OPERATORII DE ACCES

Operatorii de acces la membrii clasei sunt aceeași doi operatori de la structuri :

CLASĂ	STRUCTURĂ
<i>obiect.membru</i>	<i>variabila.membru</i>
<i>pointer_ob->membru</i>	<i>pointer → membru</i>

Exemplu

```
Finala obl;
t1 = obl.x; // greșit pentru că încearcă accesarea unui membru privat
t2 = obl.y; // corect
obl.PutData( t1 );
```

Observatie

Datele private nu sunt accesibile din exteriorul clasei, în schimb funcțiile *public* ale unei clase pot returna valori *private*.

2.5. OPERATORUL DE REZOLUȚIE ::

Operatorul de rezoluție apare utilizat în limbajul C++ în două situații:

1. La accesarea unor variabile mascate de existența altor variabile locale



Exemplu

```
int x=2; //variabila externa
//...
{   int x=3;
    cout << x << " , " << ::x; // afiseaza 3, 2
}
//...
cout << x; //afiseaza 2
```

2. In cazul stabilirii apartenenței unei funcții la o anumită clasă:

- cazul definirii funcțiilor membre ale unei clase
- cazul apelului unei funcții dintr-o clasă de bază mascată de o funcție în clasa derivată

Aplicatie practica

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

class Data
{
    int _an, _luna, _zi; // sunt date membre
public:
    void SetData( int an, int luna, int zi); //seteaza datele interne
    void Print( void ); // functie ce printeaza
    int NrZile( Data dc); // calculeaza diferenta de zile între data
    // curenta si data stocata la adresa resp.
};

char *luni[] = { " ", "IANUARIE", "FEBRUARIE", "MARTIE", "APRILIE",
                 "MAI", "IUNIE", "IULIE", "AUGUST", "SEPTEMBRIE",
                 "OCTOMBRIE", "NOIEMBRIE", "DECEMBRIE" };

void Data::SetData( int an, int luna, int zi)
{
    _an = an;
    _luna = luna;
    _zi = zi;
}

void Data::Print( void )
{
    printf("\n%d %12s %3d", _an, luni[_luna ], _zi);
}
```



```
int Data::NrZile( Data dc) // se considera ca fiecare luna are 30 zile
{
    int nr=0;
    nr += dc._zi >= _zi?dc._zi - _zi:(dc._luna--, 30 + dc._zi - _zi);
    nr += 30 * ( dc._luna >= _luna ? dc._luna - _luna :
        ( dc._an--, 12 + dc._luna - _luna) );
    nr += 365 * (dc._an - _an); //nr++;
    return nr;
}
void main ( void )
{
    clrscr();
    Data dn, dc; //obiecte apartinand clasei Data
    dn.SetData(2008, 9, 1);
    dc.SetData(2008, 10, 12);
    dn.Print();
    dc.Print();
    printf("\n Nr zile trecute = %d", dn.NrZile( dc ) );
}
```

O funcție cu un anumit nume poate fi definită ca fiind o funcție externă (în manieră C) sau fiind membră a unei clase, ca în acest material.

TEMA PE PARCURS - #1

Dupa cum se observa, clasa Data, implementeaza in functia NrZile(...), un algoritm de calcul ce porneste de la urmatoarele doua premise:

- toate lunile au 30 zile;
- toti anii au 365 zile.

Evident, ca un astfel de rationament este gresit. S-a apelat la acesta simplificare pentru a ne xa mai mult pe tehnica de calcul a zilelor, urmand ca imbunatatirea metodei sa se realizeze in cadrul celor doua ore de laborator.



Pentru a face diferenta intre lunile cu 30, respectiv 31 zile, se recomanda urmatoarea tehnica de programare – metoda *LookUp Table*³, astfel secventa:

```
nr += 30 * ( dc._luna >= _luna ? dc._luna - _luna : ( dc._an--, 12 +  
dc._luna - _luna) );
```

se va inlocui prin:

```
nr += LUT[ dc._luna ) * ( dc._luna >= _luna ? dc._luna - _luna :  
( dc._an--, 12 + dc._luna - _luna) );
```

unde LUT este un vector ce contine numarul de zile din fiecare luna.

```
int LUT[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Dupa corectia numarului de zile din luna se va aborda problema anilor bisecti, problema ce se propaga in doua zone in cadrul aplicatiei:

- A. problema lunii februarie, ca si numar de zile – corectarea valorii 28, corespunzator lunii februarie, din tabela LUT[], cu valoarea 29 in cazul anului bisect;
- B. problema numarului de zile intre doi ani – care in acest caz se rezuma la a aduna 365 sau 366 zile, functie de anul in discutie.

```
nr += 365 * (dc._an - _an); //nr++;
```

2.6. FUNCȚII INLINE

Această tehnică poate fi utilizată atât pentru funcții externe ce nu aparțin unei clase cât și pentru funcții membre ce aparțin unei clase.

³ *LookUp Table* – tabela cu valori precalculate si accesaibile prin index -

<http://tibasicdev.wikidot.com/lookuptables>



Pentru ca o funcție să o declarăm în inline în general se folosește cuvântul cheie inline *dispus* fie în fața declarației fie înaintea funcției. Funcțiile *inline* se aseamănă cu macrodefinițiile cunoscute în limbajul C dar sunt tratate de compilator din punct de vedere a argumentelor ca niște funcții.

Funcțiile *inline* respectă toate condițiile funcției membre a unei clase și pot fi moștenite. Orice funcție membră a unei clase definită în cadrul unei clase implicit este considerată a fi *inline* (stabilirea corpului funcției în clasă). Poate exista situația ca funcția să fie declarată în interiorul clasei și definită în exteriorul clasei, caz în care se va dispune specificatorul *inline* în fața funcției (în momentul declarării sau în momentul definirii).

Exemplu

```
class Test
{
    int x;
public:
    void FunctieNormala( void );
    void FunctieInlineImpl( void )
    {
        x=0;
        //...
    }
    inline void FunctieInlineExp( void );
};

void Test :: FunctieInlineExp( void )
{
    x=2;
    //...
}

void Test :: FunctieNormala( void )
{
    x=3;
    //...
}
```

2.7. FUNCTII CU ARGUMENTE AVAND VALORI IMPLICITE

Este cazul funcțiilor ce au în lista argumentelor o parte din aceste argumente inițializate cu niște valori denumite valori implicite. Aceste valori de inițializare intervin în apelul funcției



atunci când apelăm funcția cu mai puține valori (argumente). Ultimele valori netransmise înseamnă folosirea de către argumentele respective a valorilor implicite.

Sintaxa generală după care se declară:

```
<tip_returnat> <nume_functie>( <tip_1> <arg_1> [ = <exp_1> ] ... );
```

Exemplu

```
double max( double x1, double x2, double x3 = 0 );
```

Această funcție poate fi apelată în două moduri:

```
double a, b, c;  
a = max( b, c );           // x3 = 0  
a = max( b, c, c + b );    // x3 = ( c + b ) nu mai ia valoare implicita
```

Ordinea de asociere a argumentelor cu parametrii transmiși se face de la stânga la dreapta.

3. TEMĂ

2.1. Rezolvarea problemei propuse in cadrul „TEMA PE PARCURS - #1 ”

2.2. Realizati un program care va implementa notiunea de numar complex, in format algebric si va realiza operatiile de baza asociate unui numar complex:

- adunare (a doua numere complexe cat si adunarea unui complex cu un scalar),
- scaderea,
- inmultirea,
- impartirea,
- modulul unui numar complex,
- argumentul numarului complex.

LABORATORUL 3

STREAMURI. OPERATII DE INTRARE / IESIRE IN C++

1 SCOPUL SEDINTEI DE LABORATOR

In cadrul acestui laborator este prezentata notiune de stream, din perspectiva limbajului C++, streamuri ce sunt asimilate cu consolele standard de intrare / iesire.

In acelasi timp sunt definite obiectele standard de interfata *cin*, *cout*, *cerr*, *clog* cat si principalele operatii cu aceste obiecte.

Tot in cadrul acestei lucrari de laborator sunt prezentati si exemplificati manipulatorii asociati acestor obiecte de interfata.

In ultima parte a lucrarii sunt prezentate clasele specifice de lucru cu fisiere precum si interfata asociata acestor clase.

Din punct de vedere practic sunt propuse doua aplicatii, prima pentru testarea modului de formatare a datelor pe consola standard de iesire (*o alternativa la printf*), iar a doua aplicatie va implementa operatii simple pe fisiere, utilizand clasele standard: *ifstream*, respectiv *ofstream*.

In finalul lucrarii este propusa o tema, definirea clasei *clsFileOperation*, clasa ce implementeaza un set de operatii cu fisiere.



Pentru clasa in discutie, `clsFileOperation`, studentul dispune de prototip, urmand sa implementeze metodele clasei, sa construiasca functia `main()` in care va utiliza clasa si sa defina un set de testare prin care clasa sa poata fi validata si introdusa in lucru .

1.1. OBIECTIVE MINIME, NECESARE PENTRU NOTA 5

- Asimilarea notiunii stream, notiune ce apartine si standardului C++, impreuna cu operatiile standard;
- Utilizarea corecta a operatiilor standard cu consola de intrare si iesire (`cout`, `cin`);
- Utilizarea minimala a operatiilor fisiere in C++, utilizarea minimala a claselor `ifstream` si `ofstream`;
- Analiza problemelor propuse si finalizate in cadrul materialului.

1.2. OBIECTIVE MINIME, NECESARE PENTRU NOTA 10

- Asimilarea notiunii stream impreuna cu intreaga arhitectura prezentata, notiune ce apartine si standardului C++, impreuna cu operatiile standard;
- Utilizarea corecta a operatiilor standard cu consola de intrare si iesire (`cout`, `cin`);
- Utilizarea minimala a operatiilor fisiere in C++, utilizarea minimala a claselor `ifstream` si `ofstream`;
- Analiza problemelor propuse si finalizate in cadrul materialului. Rezolvarea temei propuse in dreptul fiecarei probleme propuse;
- Analiza si finalizarea materialului furnizat, corespunzator temei propuse, de la sfarsitul lucrarii de laborator.

1.3. COMPETENTE MINIME, DOBANDITE PENTRU NOTA 5

- intelegerea notiunii de stream in C++;



- operatii simple cu obiectele cin, cout, respectiv cu fisiere in C++;

1.4. COMPETENTE MINIME, DOBANDITE PENTRU NOTA 10

- intelegerea notiunii de stream in C++, realizarea de operatii complexe cu streamurile de iesire, respectiv intrare;
- operatii complexe cu fisiere in C++;
- definirea de clase specializate pentru lucru cu fisiere, in C++.
- definirea de scenarii de testare pentru a "prinde" eventualele bug-uri ce pot aparea in utilizarea claselor definite.

2 BAZELE TEORETICE

2.1 STREAM-URI. ARHITECTURA DE CLASE

Stream-urile au in principal rolul de a abstractiza operatiile de intrare- iesire. Ele ofera metode de scriere si citire a datelor independente de dispozitivul I/O si chiar independente de platforma. Stream-urile incapsuleaza (ascund) problemele specifice dispozitivului cu care se lucreaza, sub libraria standard iostream.

Alt avantaj al folosirii stream-urilor se datoreaza implementarii libreriei iostream, care utilizeaza un sistem de buffer-e. Se stie ca in general operatiile de intrare/iesire cu dispozitivele periferice sunt relativ mari consumatoare de timp, astfel incat aplicatiile sunt uneori nevoite sa astepte terminarea acestor operatiuni.

Informatiile trimise catre un stream nu sunt scrise imediat in dispozitivul in cauza, ci sunt transferate intr-o zona de memorie tampon, din care sunt descarcate catre dispozitiv abia in momentul umplerii acestei zone de memorie.

In C++ stream-urile au fost implementate utilizand clase, dupa cum urmeaza:

- clasa *streambuf* gestioneaza buffer-ele.



- clasa *ios* este clasa de baza pentru clasele de stream-uri de intrare si de iesire. Clasa *ios* are ca data mebra o referinta (pointer) la un obiect de tip *streambuf*.
- clasele *istream* si *ostream* sunt derivate din *ios*.
- clasa *iostream* este derivata din *istream* si *ostream* si ofera metode pentru lucrul cu terminalul.
- clasa *fstream* ofera metode pentru operatii cu fisiere.

Toata acesta arhitectura de clase este prezentata mai jos. Intelegerea completa a acestei arhitecturi va fi posibila doar dupa studiul mostenirii. In acest moment ea va fi prezentata doar cu scop informativ.

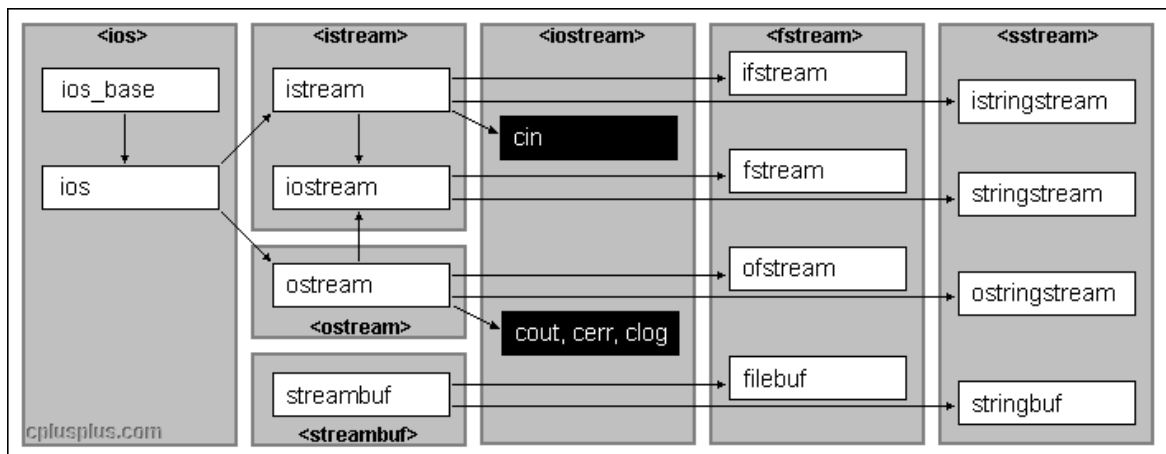


Figura 1. Arhitectura de clase pentru libraria standard de clase de intrare/iesire¹

2.2 OBIECTE STANDARD

Cand un program C++ care include *iostream.h* / *iostream* este lansat in executie, sunt create si initializate automat patru obiecte:

¹ IO stream Library - <http://www.cplusplus.com/reference/iostream/>



- cin gestioneaza intrarea de la intrarea standard (tastatura).
- cout gestioneaza iesirea catre iesirea standard (ecranul).
- cerr gestioneaza iesirea catre dispozitivul standard de eroare (ecranul), neutilizand buffer-e.
- clog gestioneaza iesirea catre dispozitivul standard de eroare (ecranul), utilizand buffer-e.

Toate aceste aspecte pot fi urmarite si in figura 1, de mai sus.

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

int main( void )
{
    cerr << "cerr"<<endl; //INSERARE IN CERR
    clog << "clog"<<endl; //INSERARE IN CLOG
    cout << "cout"<<endl; //INSERARE IN COUT

    return 0;
}
```

Vizualizare cod sursa C++ in Linux utilizand comanda *vi lab3.cc*.

```
#include <iostream>
using namespace std;
int main( void )
{
    cerr << "cerr"<<endl; //INSERARE IN CERR
    clog << "clog"<<endl; //INSERARE IN CLOG
    cout << "cout"<<endl; //INSERARE IN COUT
    return 0;
}
```

Compilare cod C++, in Linux, utilizand compilatorul g++, din familia gcc. Va scoate la iesire executabilul *lab3*.



```
[remus@apollo poo]$ g++ lab3.cc -o lab3  
[remus@apollo poo]$
```

Vizualizare continut folder, in Linux, utilizand comanda `ls -al`. Se observa prezenta unui nou fisier `lab3`, cu drept de executie.

```
[remus@apollo poo]$ ls -al  
total 36  
drwxrwxr-x  3 remus remus  4096 2016-10-19 13:35 .  
drwxr-xr-- 15 remus remus  4096 2016-10-18 09:31 ..  
-rwxrwxr-x  1 remus remus  6611 2016-10-19 13:35 lab3  
-rw-rw-r--  1 remus remus   200 2016-10-19 13:21 lab3.cc
```

Lansare in executie a fisierului executabil cu separarea erorilor, redirectarea iesirii standard de eroare (`2>`).

```
[remus@apollo poo]$ ./lab3 2>err  
cout  
[remus@apollo poo]$
```

Vizualizare continut folder in Linux utilizand comanda `ls -al`. Se observa prezenta unui nou fisier `err`, creat de comanda anterioara.

```
[remus@apollo poo]$ ls -al  
total 40  
drwxrwxr-x  3 remus remus  4096 2016-10-19 13:38 .  
drwxr-xr-- 15 remus remus  4096 2016-10-18 09:31 ..  
-rw-rw-r--  1 remus remus    10 2016-10-19 13:38 err  
-rwxrwxr-x  1 remus remus  6611 2016-10-19 13:35 lab3  
-rw-rw-r--  1 remus remus   200 2016-10-19 13:21 lab3.cc
```

Vizualizare continut fisier `err`.

```
[remus@apollo poo]$ cat err  
cerr  
clog
```




2.3 REDIRECTARI

Dispozitivele standard de intrare, iesire si eroare pot fi redirectate catre alte dispozitive. Erorile sunt de obicei redirectate catre fisiere, iar intrarea si iesirea pot fi conduse ("piped") catre fisiere utilizand comenzi ale sistemului de operare (utilizarea iesirii unui program ca intrare pentru altul).

Sintaxa pentru operatii de iesire, cout:

```
cout << <expresie1> [ << <expresie2> [...] ];
```

respectiv pentru intrare, cin:

```
cin >> <variabila1> [ >> <variabila2> [...] ];
```

De fapt, *cin* si *cout* sunt obiecte definite global, si au supraincarcat² operatorul >> respectiv <<, de mai multe ori, pentru fiecare tip fundamental, derivat din cele fundamentale, sau pointeri la tipurile fundamentale si derivate din acestea (int, char *, unsigned int, long double, etc.):

```
istream &operator >> ( TipParametru & );
```

Exemplu

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

int main( void )
{
    int intNumber;
    cout << "Numar int = "; //INSERARE IN FLUX
    cin >> intNumber;       //EXTRAGERE DIN FLUX
    //INSERARE IN FLUX
    cout << "\nAi introdus= " << intNumber << endl;
    return 0;
}
```

² Supraincarcarea operatorilor in C++ - Curs 6 POO - <http://www.cplusplus.com/doc/tutorial/classes2/>



```
Numar int = 32
Ai introdus= 32
Process returned 0 (0x0)   execution time : 4.656 s
Press any key to continue.
```

Acest scurt program citește de la intrarea standard o valoare întreagă, pe care o trimite apoi către ieșirea standard. Se observă posibilitatea de a utiliza simbolurile '\n', '\t', s.a.m.d (ca la printf, scanf, etc.). Utilizarea simbolului endl va forța golirea zonei tampon, adică trimiterea datelor imediat către ieșire.

Atât operatorul >> cât și << returnează o referință către un obiect al clasei istream. Deoarece cin respectiv cout este și el un obiect istream, valoarea returnată de o operație de citire/scriere din/in stream poate fi utilizată ca intrare/ieșire pentru următoarea operație de același fel, efectul de înșiruire a operatorului << (respectiv >>) observat în exemplele date.

2.4 OBIECTUL CIN. FUNCȚII MEMBRE PUBLICE

FUNCȚIA CIN.GET()

Funcția membră `get()` poate fi utilizată pentru a obține un singur caracter din intrare, apelând-o fără nici un parametru, caz în care returnează valoarea utilizată, sau ca referință la un caracter.

FUNCȚIA CIN.GET() FĂRĂ PARAMETRI

În această formă, funcția întoarce valoarea caracterului găsit. De remarcat este faptul că, spre deosebire de operatorul >>, nu poate fi utilizată pentru a citi mai multe intrări, deoarece valoarea returnată este de tip întreg, nu un obiect *istream*. Mai jos, un exemplu de utilizare:

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

int main( void )
{
    char c;
    while((c = cin.get()) != '\r')
        cout << "c = " << c << endl;
```



```
}  
    return 0;  
}
```

CITIREA SIRURILOR DE CARACTERE UTILIZAND *CIN.GET()*

Operatorul >> nu poate fi utilizat pentru a citi corect siruri de caractere de la intrare deoarece spatiile sunt interpretate ca separator intre diverse valori de intrare. In astfel de cazuri trebuie folosita functia `get()`. Sintaxa de utilizare a functiei `get` in acest caz este urmatoarea:

```
cin.get(char *PtrLaSir, int LngMax, char CharDeSfarsit);
```

Primul parametru este un pointer la zona de memorie in care va fi depus sirul de caractere. Al doilea parametru reprezinta numarul maxim de caractere ce poate fi citit plus unu. Cel de-al treilea parametru este caracterul de incheiere a citirii, care este optional (implicit considerat '\n' – valoarea implicita).

In cazul in care caracterul de incheiere este intalnit inainte de a fi citit numarul maxim de caractere, acest caracter nu va fi extras din stream. Exista o functie similara functiei `get()`, cu aceeasi sintaxa, numita `getline()`. Functionarea sa este identica cu `get()`, cu exceptia faptului ca acel ultim caracter mentionat mai sus este si el extras din stream.

FUNCTIA *CIN.IGNORE()*

Aceasta functie se utilizeaza pentru a trece peste un numar de caractere pana la intalnirea unui anumit caracter. Sintaxa sa este:

```
cin.ignore(int NumarMaximDeCaractere, char Sfarsit);
```

Primul parametru reprezinta numarul maxim de caractere ce vor fi ignorate iar al doilea parametru caracterul care trebuie gasit.

FUNCTIA *CIN.PEEK()*

Aceasta functie returneaza urmatorul caracter din stream, fara insa a-l extrage.

FUNCTIA *CIN.PUTBACK()*

Aceasta functie insereaza in stream un caracter.



2.5 OBIECTUL *COUT*. FUNCTII MEMBRE PUBLICE

FUNCTIA *COUT.FLUSH()*

Functia `cout.flush()` determina trimiterea catre iesire a tuturor informatiilor aflate in zona de memorie tampon. Aceasta functie poate fi apelata si in forma `cout << flush`.

FUNCTIA *COUT.PUT()*

Functia `cout.put()` scrie un caracter catre iesire. Sintaxa sa este urmatoarea:

```
cout.put(char Caracter);
```

Deoarece aceasta functie returneaza o referinta de tip `ostream`, pot fi utilizate apeluri succesive ale acesteia, ca in exemplul de mai jos:

```
#include <iostream>

using namespace std;

int main()
{
    cout.put('H').put('i').put('!').put('\n');
    return 0;
}
```

FUNCTIA *COUT.WRITE()*

Aceasta functie are acelasi rol ca si operatorul `<<`, cu exceptia faptului ca se poate specifica numarul maxim de caractere ce se doresc scrise. Sintaxa functiei `cout.write()` este:

```
cout.write(char *SirDeCaractere, int CaractereDeScriis );
```

2.6 OBIECTUL *COUT*. FORMATAREA IESIRII

FUNCTIA *COUT.WIDTH()*

Aceasta functie permite modificarea dimensiunii valorii trimise spre iesire, care implicit este considerata exact marimea campului in cauza. Ea modifica dimensiunea numai pentru urmatoarea operatie de iesire. Sintaxa este:

```
cout.width(int Dimensiune );
```



OPTIUNI DE FORMATARE A IESIRII

Pentru formatarea iesirii sunt definite doua functii membre ale *cout*, si anume:

FUNCTIA COUT.SETF()

Aceasta functie activeaza o optiune de formatare a iesirii, primita ca parametru:

```
cout.setf( ios::Optiune );
```

Unde Optiune poate fi:

- ios::showpos
- ios::left, ios::right, ios::internal
- ios::dec, ios::oct, ios::hex
- ios::showbase

precum si altii prezentati in Anexa2

ANEXA 1.

Manipulatorii bibliotecii *iostream*

Manipulator	Exemplu	Efect
Dec	cout<<dec<<x cin>>dec>>x	reprezentare in baza 10
Hex	cout<<hex<<x cin>>hex>>x	reprezentare in baza 16
Oct	cout<<oct<<x cin>>oct>>x	reprezentare in baza 8
Ws	cin>>ws	elimina spatiile libere din flux
Endl	cout<<endl	'\n' si descarca bufferul de iesire
Ends	cout<<ends	insereaza caracter NULL in flux
Flush	cout<<flush	Descarca bufferul <i>ostream</i>
Resetiosflags(cout<<resetiosflags(ios::dec)	reinitializeaza bitii de



long)	cin>>resetiosflags(ios::hex)	formatare specificati de catre argument
Setbase(int)	cout<<setbase(10) cin>>setbase(8)	stabileste baza de conversie
Setfill(int)	cout<<setfill('.') cin>>setfill(' ')	stabileste caracterul de umplere
Setiosflags(int)	cout<<setiosflags(ios::dec) cin>>setiosflags(ios::hex)	stabileste bitii de formatare specificati de catre argument
Setprecision(int)	cout<<setprecizion(6) cin>>setprecizion(12)	stabileste precizia la numarul specificat de zecimale
setw(int)	cout<<setw(6) cin>>setw(12)	stabileste lungimea unui camp, la numarul specificat

ANEXA 2.

Indicatorii de formatare din *iostream*

Nume indicator	Efectul produs
ios::skipws	Elimina spatiile goale din intrare
ios::left	Aliniaza iesirea la stanga in interiorul latimii
ios::right	Aliniaza iesirea la dreapta in interiorul latimii
ios::scientific	Foloseste notatia stiintifica pt. numere reale
ios::fixed	Foloseste notatia zecimala pt. numere reale
ios::dec	Foloseste notatia zecimala pt. numere intregi
ios::hex	Foloseste notatia hexazecimala pt. numere intregi
ios::oct	Foloseste notatia octala pt. numere intregi
ios::uppercase	Foloseste litere mari pentru iesire
ios::showbase	Indica baza sistemului de numeratie
ios::showpoint	Include punctul zecimal pentru numere in virgula flotanta
ios::showpos	Indica semnul + la numerele pozitive
ios::unitbuf	Goleste toate fluxurile dupa introducerea caracterelor in flux

Pentru utilizarea manipulatorilor in cadrul unui program scris in limbajul C++ se va include fisierul `<iomanip.h>/iomanip`.

APLICATIE PRACTICA #1

In continuare vom exemplifica utilizarea functiilor pentru formatarea iesirii.

```
#include <iostream>
```



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

int main()
{
    int number = 783;
    cout << "numar zecima = | " << setw(10) << number << " | " << endl;
    cout.setf(ios::showbase);
    cout << "numar in hexa = | " << setw(10) << hex << number << " | " << endl;
    cout.setf(ios::left);
    cout << "numar in octal, aliniat la stanga = | ";
    cout << setfill('.') << setw(10) << oct << number << " | " << endl;
}
```

```
numar zecina = |          783!
numar in hexa = |        0x30f!
numar in octal, aliniat la stanga = !01417.....!
Process returned 0 (0x0)   execution time : 0.109 s
Press any key to continue.
```

Dupa modelul aplicatiei propuse mai sus se va dezvolta o noua aplicatie ce va „*tabela*” functia *sinus* pe un interval *a, b*, specificat de la consola.

2.7 OPERATII DE INTRARE/IESIRE CU FISIERE

Lucrul cu fisiere se face prin intermediul clasei *ifstream* pentru citire respectiv *ofstream* pentru scriere. Pentru a le utiliza, aplicatiile trebuie sa includa *fstream.h*. Clasele *ofstream* si *ifstream* sunt derivate din clasa *iostream*, ca urmare toti operatorii si toate functiile descrise mai sus sunt mostenite si de aceasta clasa.

Sintaxa pentru constructorii acestor doua clase este:

```
ofstream v1(char *nFis, int mod=ios::mod);
ifstream v2(char *nFis, int mod=ios::in, int buf = filebuf::openprot );
```

Acesti constructori au rolul de a deschide fisierul specificat ca parametru. Cel de-al doilea parametru al constructorului *ofstream* este optional si specifica modul de deschidere a fisierului:

- `ios::append` - adauga la sfarsitul fisierului;



- `ios::atend` - pozitioneaza pointer-ul la sfarsitul fisierului, insa informatiile pot fi scrise oriunde in cadrul fisierului;
- `ios::truncate` - este modul de deschide implicit: vechiul continut al fisierului este pierdut;
- `ios::nocreate` - daca fisierul nu exista, atunci operatia esueaza;
- `ios::noreplace` - daca fisierul deja exista, atunci operatia esueaza.

Pot fi utilizate prescurtarile: `app` pentru `append`, `ate` pentru `atend`, si `trunc` pentru `truncate`. Pentru a inchide aceste fisiere trebuie apelata functia membra `close()`. Rezultatul operatiilor de intrare/iesire poate fi testat prin intermediul a patru functii membre:

- `eof()` verifica daca s-a ajuns la sfarsitul fisierului;
- `bad()` verifica daca s-a executat o operatie invalida;
- `fail()` verifica daca ultima operatie a esuat;
- `good()` verifica daca toate cele trei rezultate precedente sunt false.

POZITIONAREA IN CADRUL UNI FISIER SE FACE PRIN INTERMEDIUL FUNCTIEI:

- `seekg(int deplas, int ref)` unde `ref` reprezinta referinta fata de care se face deplasarea si anume: `ios::beg`, `ios::end`, `ios::cur`, acesta fiind si un argument avand valoarea implicita `ios::cur`.

Aflarea pozitiei curente in cadrul unui fisier se poate face prin intermediul functiei:

- `tellg()`, `tellp()` ce va returna pozitia curenta pentru un flux de intrare si respectiv pentru un flux de iesire.

APLICATIE PRACTICA #2



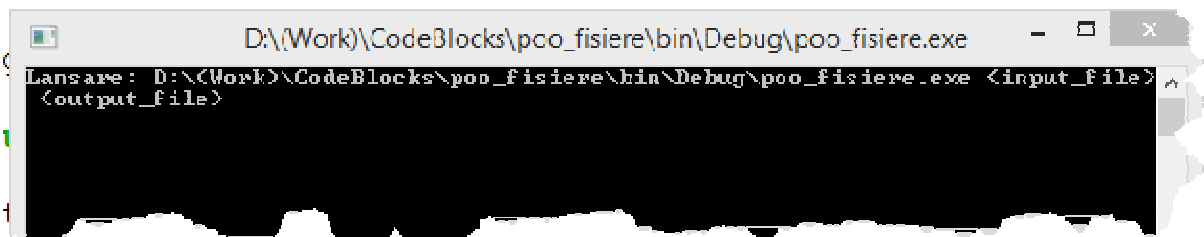
Este prezentat un program ce va deschide 2 fisiere ale caror nume au fost furnizate in linia de comanda si va realiza o copiere la nivel de octet intre cele 2 fisiere.

```
#include <iostream> // clasele standard de I/O si cin, cout, ...
#include <fstream> // clasele de lucru cu fisiere ifstream, ofstream
#include <iomanip> // lista tuturor manipulatorilor
#include <conio.h>

using namespace std;

int main( int argc, char *argv[ ] )
{
    if( argc < 3 ) // nu au fost furnizate 2 nume de fisiere
    {
        cout<<"Lansare: "<<argv[0]<<" <input_file> <output_file>"
            <<endl;
        getch();
        return 1;
    }
    ifstream iFile( argv[1], ios::binary ); //Obiect de tipul ifstream
    if( iFile.fail() ) // stabileste daca a esuat deschiderea
    {
        cout<<endl<<"Eroare la deschiderea lui "<<argv[1]<<endl;
        return 2;
    }
    ofstream oFile(argv[2],ios::binary|ios::trunc ); //Obiect ofstream
    if( oFile.fail() ) // stabileste daca a esuat deschiderea
    {
        cout<<endl<<"Eroare la deschiderea lui "<<argv[2]<<endl;
        return 3;
    }
    while( !iFile.eof() )
        oFile<<(char)iFile.get();
    iFile.close();
    oFile.close();
    cout<<"Copiere realizata cu succes!"<<endl;
    getch();
    return 0;
}
```

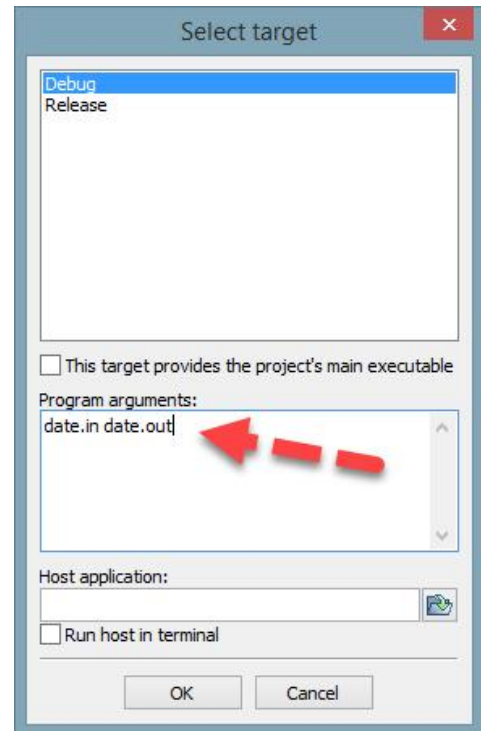
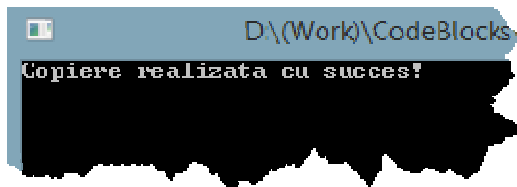
Lansarea in executie fara parametri in linia de comanda se face astfel:





Pentru lansarea in executie, cu parametri in linia de comanda, vom seta numele celor doua fisiere in meniul Code::Blocks, *Project* -> *Set program's arguments*, ca in figura din dreapta.

Efectul rularii, cu parametri in linia de comanda, se vede in figura de mai jos.



2.7.1 TEMA PE PARCURS

Dupa modelul aplicatiei propuse mai sus se va modifica aplicatie astfel incat in fisierul de iesire continutul sa fie convertit in hexazecimal, fiecare caracter va fi afisat ca si valoare ASCII, in format hexazecimal.

3 TEMĂ

Sa se scrie un program care va implementa toate metodele clasei *clsFileOperation*. Pe baza acestei clase sa se realizeze o aplicatie care va citi dintr-un fisier un vector de siruri si va realiza sortarea acestui vector.

```
#define false    0
#define true     1
#define NMAX_CHAR 200

typedef int boolean;

class clsFileOperation
{
private:
    int      m_lLengthFile;           // Specifica lungimea fisierului
    fstream * m_pFilePointer;        // Pointer-ul catre fisier
    long     m_lPositionInFile;      // Specifica pozitia in fisier
```



```
public:
    char    m_sFileName[NMAX_CHAR]; // Specifica numele fisierului
    boolean Open (int nMode);      // Deschiderea fisierului.
    boolean Close();               // Inchiderea fisierului
    boolean Create();              // Crearea fisierului
    boolean DeleteContains();      // Stergerea continutului fisierului
    long getLength();              // Functie care calculeaza lungimea
    // fisierului, seteaza m_lLengthFile si
    // intoarce lungimea fisierului
    char * getLine();             // Aceasta functie citeste o linie din fisier
    // si intoarce aceasta linie
    char getChar();               // Functie citeste un caracter din fisier
    //si intoarce acest caracter
    boolean putChar(char c);       // Functie insereaza un caracter la
    // sfirsitul fisierului. Functia intoarce true
    // la succes si false in caz contrar
    boolean putLine(char *sLine); // Functie insereaza o linie la
    // sfirsitul fisierului.
    // Functia intoarce true daca operatia s-a
    // terminat cu success si false in caz contrar
    void setPos(int lPos);         // Functie seteaza pozitia in fisier,
    // si apoi seteaza m_lPositionInFile la valoarea lPos
    long getPos();                // Functie care preia pozitia curenta din fisier
    // seteaza variabila m_lPositionInFile si
    // intoarce aceasta pozitie
    void Print();                 // Functie utilizata pentru afisarea fisierului
};
```

LABORATORUL 4

1. SCOP

Laboratorul prezinta notiunea de functie constructor si functie destructor a unei clase, proprietatile acestor metode precum si modul in care aceste functii sunt apelate.

In partea a doua a laboratorului se analizeaza alocatorul de memorie dinamica specific limbajului C++, urmata de o paralela intre cele doua tehnici, cea specifica C-ului si cea specifica limbajului C++.

In partea finala a laboratorului sunt prezentate notiunile de obiecte statice, automate si dinamice, urmand sa se incheie cu o tema propusa spre rezolvare.

2. BAZELE TEORETICE

2.1. CONSTRUCTORII SI DESTRUCTORII CLASEI

Utilizarea funcțiilor de inițializare a obiectelor de tipul `init()` sau `set_data()` din exemplele precedente nu este elegantă și putand produce erori de interpretare. Deoarece nu este specificat faptul că un obiect trebuie inițializat, un programator poate uita să facă acest lucru, sau (adesea cu rezultate dezastruoase) să facă acest lucru de 2 ori.

O solutie mai bună este aceea de a permite programatorului să declare o funcție cu scopul explicit de a inițializa obiecte. Deoarece această funcție construiește obiecte de un tip dat, ea se numește constructor.

2.2. CONSTRUCTORI

Utilitatea constructorilor este evidentă cel puțin sub două aspecte:

1. constructorul asigură inițializarea corectă a tuturor variabilelor membru ale unui obiect;
2. constructorul oferă o garanție că inițializarea unui obiect se va realiza exact o dată.

Următoarele sublinieri sunt necesare:

1. un constructor poartă numele clasei căreia îi aparține

```
class Date
{
    //....
    Date( int, int, int );
};
```

2. constructorii nu pot returna valori. În plus, prin convenție, nici la declararea și nici la definirea lor nu poate fi specificat tipul "void" ca tip returnat.
3. adresa constructorilor nu este accesibilă programatorului. (&X : X() - nu se poate).
4. constructorii sunt apelați implicit ori de câte ori este nevoie.
5. În cazul în care o clasă nu are nici un constructor declarat de programator, compilatorul va genera *constructorul implicit*. Acesta va fi public, fără nici un parametru, și va avea o listă vidă de instrucțiuni.

Constructor implicit se numește și constructorul fără listă de argumente declarat de programator X(). Dacă constructorul are argumente, ele pot fi furnizate:

```
date today = date( 12, 12, 1996 );
date xday(1,12,1996); //forma prescurtata
date day;           //eronat, lipseste initializarea
```

De multe ori este util să existe mai multe moduri de inițializare a obiectelor unei clase. Acest lucru se poate realiza furnizând diferiți constructori. Atâta timp cât constructorii diferă suficient

în tipurile argumentelor lor, compilatorul le poate selecta corect, unul pentru fiecare utilizare.

```
class date
{
    int month,day,year;
public:
    date(int,int,int);    //zi luna an
    date();               //data curenta - constructor implicit
    date(char *);        //sir de caractere
};

date xday( 12,12,1996 );
date yday( "10 Oct. 1996" );
date zday;               //initializare implicita
```

Un obiect al unei clase poate fi inițializat și prin atribuirea unui alt obiect al aceleiasi clase:

```
date d = xday;
```

2.3. DESTRUCTORI

Pentru ștergerea obiectelor inițializate cu constructori, se utilizează funcții specifice, denumite destructori. Numele destructorului pentru clasa X este ~X(). De exemplu, pentru clasa lista simplu înlănțuită, o soluție ar fi urmatorul cod, cu dezavantajele de rigoare:

```
class lista
{
    int cheie;
    lista* next;

public:
    lista( int& );           // Constructor
    ~lista() { delete next; } // Functia destructor INLINE
    void insert( lista* );
    void print();
};

lista::lista( int &k )      // Constructor
{
    cheie = k;
    next = NULL;
}
```

2.4. ALOCAREA DINAMICA A MEMORIEI. OPERATORUL NEW

Alocarea dinamica de memori, in C++, este realizata folosind cei doi operatori, new si delete. Sintaxa generala de utilizare a operatorilor este:

```
<pointer> = new <tip> [(<parametric reali>) ];
```

```
<pointer> = new <tip> [ [<dim>] ];
```

```
delete pointer;
```

```
delete []pointer;
```

Exemplu:

```
class Test
{
    <date membre>
    <metode>
};

int *p = new int; //se alocă un întreg
int *ps = new int[20]; //se alocă 20 de întregi
Test *pa = new Test; //se alocă dinamic memorie pentru un obiect
Test *p1 = new Test("Test"); //se alocă memorie pentru un obiect
Test *p2 = new Test[10]; //se alocă dinamic memorie pentru 10 obiecte
Test *p3 = new Test( *p1 ); //se alocă memorie pentru 1 obiect.

//... Se va apela constructorul de copiere
delete p; //se dealoca memoria unui int
delete ps; //se dealoza zona celor 20 de caractere
delete p1; //se dealoca memoria ocupata de un obiect
delete []p2; //se apeleaza destructorul de atâtea ori cât specifică lungimea sirului
delete p3; //se apeleaza destructorul o singura data, dar se dealoca corect
```

În alocarea dinamică, cea mai uzuală eroare este generată de imposibilitatea alocării memoriei. Pe lângă soluția banală, dar extrem de incomodă, de testare a valorii adresei returnate de către operatorul new, limbajul C++ oferă și posibilitatea invocării, în caz de eroare, a unei funcții definite de programator.

Rolul acestei funcții este de a obține memorie, fie de la sistemul de operare, fie prin eliberarea unor zone deja ocupate. Mai exact, atunci când operatorul *new* nu poate alocă spațiul solicitat, el invocă funcția a cărei adresă este dată de variabila globală *_new_handler*, de tipul *std::new_handler* și apoi încearcă din nou să aloce memorie. Tipul *std::new_handler* este obținut astfel:

```
typedef void (*new_handler)();
```

Valoarea pointerului este implicită 0 (NULL).

Valoarea NULL a pointerului `_new_handler` marchează lipsa funcției de tratare a erorii și în această situație, operatorul `new` va returna 0 ori de câte ori nu poate alocă memorie. Programul poate modifica valoarea acestui pointer, prin intermediul funcției de bibliotecă

```
pFunctie = std::set_new_handler( mem_warn );
```

Toate declarațiile necesare pentru utilizarea pointerului `_new_handler` se găsesc în fișierul header `<new>`.

```
#include <iostream>      // std::cout
#include <cstdlib>        // std::exit
#include <new>            // std::set_new_handler
#include <conio.h>
//using namespace std;

// pointer la o functie void fara parametri, in biblioteca <new> - typedef
void (*new_handler)();
std::new_handler pFunctie;

// functia handler de evenimente
void mem_warn( void )
{
    std::cout << "...Esuare alocare!\n";
    // seteaza vechea adresa de evenimente...
    std::set_new_handler( pFunctie );
    getch();
    std::exit (1);
}

int main( void )
{
    std::cout << "START alocare memorie\n";

    const long int dim_alocare = 100*1024*1024; //100MB

    // seteaza un nou handler de evenimente si primeste adresa existenta
    pFunctie = std::set_new_handler( mem_warn );

    while (true)
    {
        std::cout << "\nReincercare de alocare a [100 MB]... ";
        new char[dim_alocare];
        std::cout << "OK!";
    }
    //delete[] p;

    // seteaza vechea adresa de evenimente...
    std::set_new_handler( pFunctie );
```


PROGRAMARE ORIENTATA PE OBIECTE. 2016

Indrumar de Laborator

```
std::cout << "FINAL alocare memorie\n";

getch();
return 0;
}
```

In urma executie programului, comentand structura while(true), in Code::Blocks, vom gasi urmatorul raspuns al programului, din care intelegem ca alocarea au avut loc cu succes.

```
std::set_new_handler( mem_warn );

//while (true)
{
    std::cout << "\n";
    new char[dim_al
    std::cout << "0
}
```



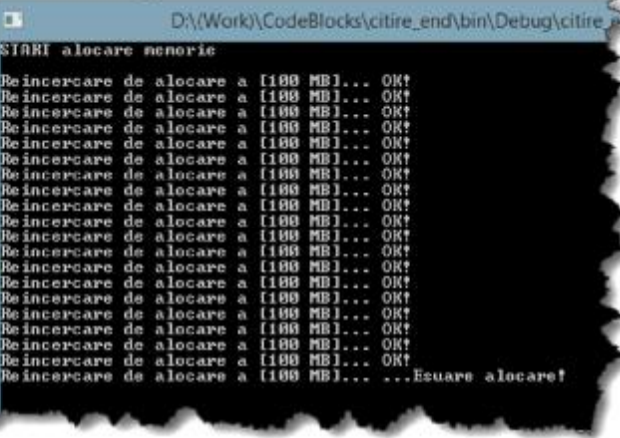
The screenshot shows the console output of the program. It starts with "START alocare memorie", followed by "Reincercare de alocare a [100 MB]... OK!FINAL alocare memorie". The output is displayed in a black window with white text.

Eliberand comentariul anterior vom observa un alt effect al programului, in momentul rularii.

```
while (true)
{
    std::cout
    new char[d
    std::cout

    ///delete[] p;
    std::cout << "

    getch();
    return 0;
}
```



The screenshot shows the console output of the program in a loop. It starts with "START alocare memorie", followed by multiple "Reincercare de alocare a [100 MB]... OK!" messages. The output is displayed in a black window with white text.

EXPLICATI!!!!

2.5. OBIECT STATIC, AUTOMATIC ȘI DINAMIC

In procesul de instantiere a obiectelor din clase, acestea pot fi create ca:

1. obiect automatic: este creat de fiecare dată când se întâlnește declarația lui la execuția programului și este distrus de fiecare dată când se iese din blocul în care el a apărut.
2. obiect static: este creat o singură dată, la pornirea programului și se distruge odată cu terminarea programului. Cazul variabilelor externe, în cazul nostru al obiectelor externe.
3. obiect dinamic: este creat folosind operatorul *new* și este distrus cu operatorul *delete*.

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

class lista_d
{
    int *tab;          // adresa tabloului
    int nrcrt;         // nr. curent de elemente din lista
    int prim;          // indexul primului element din lista
    int nmax;          // dimensiune lista
public:
    lista_d(int);
    lista_d();
    ~lista_d();
};

lista_d::lista_d( int n )
{
    nmax=n;
    tab = new int[n];
    nrcrt = prim=0;
    cout << "\nConstructor 1 --> " << nmax << "elemente";
}

lista_d::lista_d()
{
    nmax=10;
    tab = new int[10];
    nrcrt = prim = 10;
    cout << "\nConstructor 2 --> 10 elemente";
}

lista_d::~~lista_d()
{
    cout << "\nDestructor --> " << nmax << "elemente";
    delete tab;
}

void funct(int);

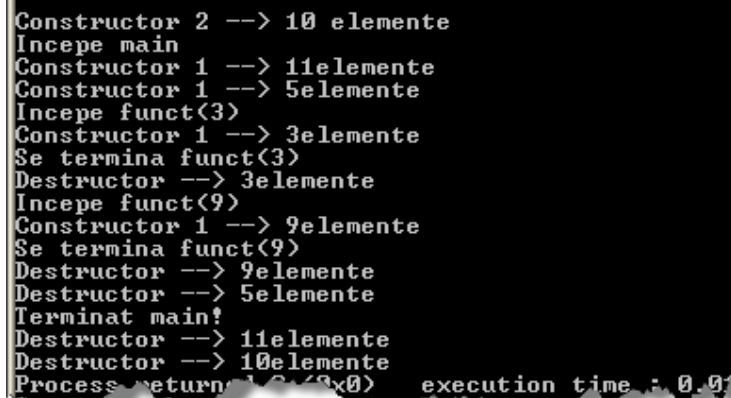
lista_d ls;           //obiect global (static) - creare

int main( void )
{
    cout << "\nIncepe main";
    lista_d la( 11 );  //obiect automatic - creare
    lista_d *lp;       //obiect dinamic
    lp = new lista_d( 5 ); //creare obiect dinamic
```

```
    funct( 3 );
    funct( 9 );
    delete lp;
    cout << "\nTerminat main!";
    return 0;
}

void funct( int i)
{
    cout << "\nIncepe funct(" << i << ")";
    lista_d lf( i );           //creare obiect automatic
    cout << "\nSe termina funct(" << i << ")";
}
```

În acest caz programul afișează:



```
Constructor 2 --> 10 elemente
Incepe main
Constructor 1 --> 11elemente
Constructor 1 --> 5elemente
Incepe funct(3)
Constructor 1 --> 3elemente
Se termina funct(3)
Destructor --> 3elemente
Incepe funct(9)
Constructor 1 --> 9elemente
Se termina funct(9)
Destructor --> 9elemente
Destructor --> 5elemente
Terminat main!
Destructor --> 11elemente
Destructor --> 10elemente
Process return 0 (0x0) execution time : 0.0
```

Comparând textul programului cu mesajele afișate se poate studia crearea și eliminarea diverselor categorii de variabile. Pentru obiectul static ls aplicarea constructorului 1 are loc înainte de funcția main(), iar eliminarea obiectului se face după revenirea din main.

Obiectul automatic la este creat cu al doilea constructor și eliminat după încheierea funcției main(). Același fenomen se observă la obiectele automate create în funcția de test funct(). Se observă că eliminarea obiectelor se face în ordine inversă creării lor.

Operatorul *new* apelează automat constructorul clasei din care face parte obiectul și îi transferă valorile de inițializare. Operatorul *delete* apelează destructorul clasei și șterge obiectul.

3. TEMĂ. PROBLEMA PROPUSA

Se consideră declarația clasei String, ce va implementa lucrul cu șiruri de caractere, prin alocare dinamică de memorie. Datele membre ale clasei sunt s, un pointer la sir de caractere, l în întreg ce va indica lungimea sirului de caractere și p un întreg ce va indica poziția curentă în cadrul șirului de caractere.

```
#ifndef _STRING
#define _STRING

class String
{
//D A T E    M E M B R E (private)
    char *s;           // pointer catre sir de caractere
    int l;              // lungimea curenta a sirului
    int p;              // pozitia curenta in cadrul sirului

public:
    //M E T O D E (publice)
    ~String();          //Destructorul clasei
    String( void );     //Constructorul implicit al clasei
    String( const char * ); //Constructorul clasei cu argument sir
    String( const int ); //Constructorul clasei.
                          //Stabileste lungimea sirului
    String( const String & ); //Constructorul de copiere
                          //Constructorul ce concateneaza doua siruri
    String( const char*, const char* );
    String( const String &, const String & ); //Constructorul ce
                          // concateneaza doua String-uri
    void Print( void ); // Afiseaza continutul obiectului
    char Getc( int i=0 ); // Returneaza caracterul de pe pozitia curenta
    void Next( int i = 1 ); // Schimba pozitia curenta cu i pozitii
    void Putc( char c, int i = 0 ); // Pune un caracter in
                          // pozitia curenta si muta pozitia curenta cu i
    String& AddString( const char * ); // Adauga un nou sir de
                          // caractere la obiectul curent
    int StrLen( void ); // returneaza lungimea sirului
    int StrPos( void ); // returneaza pozitia curenta
    void SetStrPos( int pos ); // seteaza pozitia curenta ca fiind pos
};
#endif
```

Se vor implementa toate metodele clasei String respectând cerințele formulate, astfel încât secvența următoare de program să fie validă.

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

```
void main( void )
{
    char cc;

    // Apelurile implicite de C O N S T R U C T O R I de catre compilator
    String a;           // un sir de dimensiune 0
    String b("Acesta este un "); // un sir de dimensiune 16 cu
                                // continutul din argument
    String c(20);        // un sir de dimensiune 20 octeti
    String d(b);         // un obiect similar cu b
    String e=b;          // un obiect similar cu b
    String f("TEST", " in C++"); // un obiect cu continutul celor
                                // doua siruri concatenate
    String g(b, f);       // un obiect cu continutul celor doua obiecte
                                // concatenat

    clrscr();
    // A P E L U R I de FUNCTII MEMBRE (M E T O D E)
    g.Print();           //Efect: Acesta este un TEST in C++
    cout<<g.Getc(1)<<endl;
    cout<<g.Getc(1)<<endl;
    g.Putc('M', 3);
    g.Putc('M', -3);
    g.Print();
    a.AddString("Test pe o platforma DOS");
    a.Print();
    a.SetStrPos(10);
    cout<<"Lungimea " <<a.StrLen()<<endl;
    cout<<"Pozitia " <<a.StrPos()<<endl;
    cout<<"Caracterul " <<a.Getc()<<endl;
    // Apelurile implicite de D E S T R U C T O R I de catre compilator
}
```

Rezultatul acestui program va scoate pe dispozitivul de iesire urmatoarele mesaje:

```
Acesta este un TEST in C++
A
C
AcMstM este un TEST in C++
Test pe o platforma DOS
Lungimea 23
Pozitia 10
Caracterul p
```

Laboratorul 5

1 Scop

În prima parte a laboratorului este prezentată noțiunea de referință, urmând să se facă o introducere a noțiunii de static din perspectiva limbajului C++, cât și particularitățile pe care le aduce această noțiune. Tot în cadrul laboratorului este pusă în discuție situația sirurilor de obiecte și cum este rezolvat cazul funcțiilor constructor.

În final este prezentat cazul obiectelor (claselor) ce conțin ca și date membre alte obiecte (clase).

2 Bazele teoretice

2.1 Referințe

Tipul referință poate fi definit ca un alias acordat unei variabile. Declarația unui tip de dată referință are sintaxa următoare:

```
<tip> &<nume_ref> = <nume_var>; // declarare referință  
<tip> <nume_var> [ = <expr> ]; // & simbol pentru referință
```

Referința poate fi o singură dată inițializată cu nume de variabilă și aceasta se face în momentul declarării.

```
int x=0;  
int &y= x;  
x=2;  
y=3;  
cout <<x <<y; //afișează 3 3
```

2.2 Parametri și valori returnate de tip referință

```
void f1( int a ); //param transmis prin valoare  
void f2( int *a ); //parametru de tip pointer  
void f3( int &a ); //parametru de tip referință
```



declarații	apel	afișare
<pre>void f1(int a) { a = 2; cout<<a; }</pre>	<pre>f1(x); cout<<x;</pre>	2 3
<pre>void f2(int *a) { *a = 2; cout<<a; }</pre>	<pre>f2(&x); cout<<x;</pre>	2 2
<pre>void f3(int &a) { a = 2; cout<<a; }</pre>	<pre>f3(x); cout<<x;</pre>	2 2

După cum se vede în exemplul prezentat mai sus, argumentul de tip referință a produs același efect ca și un argument de tip pointer, în schimb la folosirea sa seamănă ca utilizare cu argumentele transmise prin valoare.

Pericolele pot apărea în utilizarea referinței atunci când argumentul de tip referință este:

- modificat folosim specificatorul const
- este o expresie
- este de alt tip decat tipul referintei

Sunt identificate surse de erori frecvente în utilizarea referințelor ca parametri de funcții:

- transmiterea unei constante către un parametru de tipul referinței
- transmiterea unei variabile de tip diferit față de tipul referinței
- transmiterea unei expresii către o referință

```
const int x = 10;  
f3( x );  
f3( 3.2 );  
f3( y + 2 );
```

2.3 Funcții ce returnează referințe

Funcțiile ce returnează referințe sunt acele funcții ce pot fi utilizate și în stânga unei atribuirii (Left values = Lvalues). Sintaxa generală pentru declararea unei funcții ce returnează o referință este :

```
<tip_ref> & <nume_f>(<lista_arg>);
```

Exemplu

```
char &f(int i );
```

În acest caz funcția nu va mai returna o valoare (nu va mai fi folosit mecanismul de returnare a valorii prin stivă) ci va face un link spre o variabilă deja existentă și care nu este locală funcției.



Observatie

O funcție ce returnează referințe nu poate returna o variabilă locală declarată în interiorul funcției. Aceasta ar fi semnalizată cu eroare de compilator. Singurele variabile sau entități externe globale sau zone alocate dinamic.

```
#include <iostream.h>
char s[ 100 ];
char &f( int );
void main( void )
{
    for( int i=0; i < n; i++ )
        f( i ) = 'A'; // ~s[i]='A';
}

char &f( int ind )
{
    return s[ ind ];
}
```

În cadrul acestui exemplu funcția `f(i)` nu returnează o valoare și apelul ei este înlocuit cu legătura către entitatea returnată adică spre unul din elementele șirului `s`. Apelul ei va face un link către unul din elementele șirului `s`. Atribuirea `f(i) = 'A'` se rezumă la o expresie aproximativă.

2.4 Membri statici ai unei clase

Pentru datele nestatice ale unei clase există copii distincte în fiecare obiect. Datele statice există într-o singură copie, comună tuturor obiectelor. Crearea, inițializarea și accesul la această copie sunt total independente de obiectele clasei. Funcțiile membre statice efectuează operații care nu sunt asociate obiectelor individuale, ci întregii clase. Din acest motiv, la apelare nu este necesară indicarea unui obiect. Funcțiile membre au acces direct la datele și funcțiile statice, ca și în cazul celorlalți membri. Un membru static poate fi referit de funcțiile ne-membre prin două metode:

- indicând numele clasei și folosind operatorul de rezoluție, chiar dacă nu există obiecte ale clasei.
- specificând un obiect al clasei și folosind operatorii de selecție (la fel cu membri nestatici).

Declarația din cadrul clasei este fără definire și trebuie să existe o definiție unică în exterior. Funcțiile membre statice nu primesc implicit adresa unui obiect. Din acest motiv, în definiția lor nu se poate folosi cuvântul `this`, deci membrii nestatici pot fi referiți doar specificând numele unui obiect. În ceea ce privește membrii statici, accesul este direct.

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

class stat
{
    int i;
```




```
static int contor;  
public:  
    stat() {i=0;}  
    void inc() { i++; cout << "\ni=" << i;}  
    void inc_contor(){contor++;cout << "\ncontor" << contor;}  
    static void fct(stat *);  
};  
int stat::contor = 0;           //initializare variabila statica  
  
void stat::fct(stat *c)  
{  
    I = i + 10;                 //eroare -> nu se stie obiectul  
    c->i = c->i + 10;           //corect  
    contor++;                  //corect, contor e variabila statica  
    cout << "\nIn fct : " << c->i << "," << contor;  
}  
void main()  
{ clrscr();  
  stat ob1,ob2,ob3;  
  ob1.inc(); // ob1.i=1  
  ob2.inc(); // ob2.i=1  
  ob3.inc(); // ob3.i=1  
  
  stat::fct(&ob1);             //corect, ob1.i=11 si contor=1  
  ob2.fct(&ob1);               // corect, ob1.i=21 si contor=2  
  ob3.fct(&ob2);               // corect, ob2.i=11 si contor=3  
  ob1.inc_contor();            //contor=4  
  ob2.inc_contor();            //contor=5  
  ob3.inc_contor();            //contor=6  
}
```

2.5 Tablouri de obiecte

Tablourile pot avea elemente de orice tip, inclusiv un tip clasă, deci se pot declara tablouri de obiecte. Este necesară următoarea precizare: pentru a declara un tablou de obiecte ale unei clase, acea clasă trebuie să aibă un constructor implicit, adică un constructor care să poată fi apelat fără o listă de argumente.

```
class tabel  
{  
    char *tab;  
    int size;  
public:  
    tabel(int size=15);  
    ~tabel();  
    afisare();  
};  
  
tabel::tabel(int s)
```

```
{  
    if( s < 0 )  
        cout << "eroare...";  
    tab = new char[ size = s];  
}  
tabel::~tabel()  
{ delete tab;  
}  
void main()  
{ tabel tvector[5];  
}
```

La crearea tabloului tvector[5] va fi apelat de 5 ori constructorul clasei tabel. În cazul în care clasa dispune atât de un constructor implicit cât și de un constructor cu toate valorile



implicit, compilatorul va detecta eroare, afișând un mesaj de "Ambiguitate".

```
void f( )
{
    tabel *t1=new tabel;           //constructor -> un tabel;
    tabel *t2=new tabel[3];        //constructor -> 3 tabele;
    delete t1;                     // destructor -> un tabel;
    delete t2;                     // !! apar probleme: 3 tabele
}
```

Operatorul new alocă spațiu pentru cele 3 obiecte dinamice și apelează constructorul cu parametru implicit al clasei.

Destructorul pentru vectorul de 3 elemente nu este apelat decât o singură dată în exemplu de mai sus; pentru ca distrugerea să se realizeze corect este necesar ca programatorul să furnizeze dimensiunea tabloului: delete [3] t2;

Această problemă apare doar în cazul vectorilor alocați cu new, în celelalte cazuri destructorul se apelează automat pentru fiecare element al vectorului.

Pentru a opera asupra unui element al tabloului se poate scrie:

```
t2[i].afisare(); sau tvector[j].afisare();
```

2.6 Liste de initializare

```
Segment :: Segment( Punct &a, Punct &b ): Org( a ),varf( b )
{
    // corpul vid
}

Segment :: Segment ( int x1, int y1, int x2, int y2 ):
    Org(x1,y1), varf(x1,y2)
{
}
```

În cazul exemplului prezentat mai sus se deosebesc două tehnici utilizate pentru inițializarea obiectelor membre ale unei clase:

- în primul caz are loc o inițializare prin intermediul constructorului implicit a obiectului în cauză, apoi este creat un obiect temporar ce este inițializat în modul dorit de noi și în faza a treia are loc atribuirea între obiectul nostru și respectivul obiect temporar. Aceasta este o procedură neelegantă și deosebit de greoaie.
- al doilea caz constă în utilizarea listei de inițializare. Această listă de inițializare face parte din corpul constructorului (și nu numai) și ne este permis ca în interiorul ei să putem realiza inițializarea în maniera dorită. Astfel am putut inițializa din start obiectul origine cu valorile x1, y1 și celălalt obiect a fost inițializat tot prin intermediul aceluiași constructor transmițându-i parametrii x2, y2.

Dacă în acest sistem avem două alternative una mai elegantă și una puțin elegantă în cazul în care în interiorul unei clase declarăm o variabilă de tip cost sau o referință singura șansă de a putea inițializa aceste două entități o avem doar utilizând listele de inițializare.

```

class T
{
    const int x;
    int &r;
    int v;
public:
    T( void );
};
//O variantă incorectă de
//inițializare pentru x și r
T :: T(void)

```

```

{
    v=0;
    x=10;
    v=x;
}

//Varianta corectă de inițializare
T :: T(void):
x(10),r(v);
{
    v=0;
}

```

2.7 Clase cu membri obiecte

```

class punct
{
    int x,y; //abscisa si coordonata
public:
    punct( int a = 0,int b = 0 ) { x = a; y = b }; //constructor
};
class vector3
{
    punct origine, varf; //clasa incuibarita
public:
    vector3( int, int );
};
vector3::vector3( int x1, int y1 ) : origine( x1, y1 );
// origine - initializat prin apel explicit - 3.2
// varf - initializat prin apel implicit - 3.1, x=0,y=0

```

Observație

Întotdeauna apelul constructorilor obiectelor încuibărite se realizează înainte de executarea constructorilor obiectului "gază". Nu se poate însă preciza cu exactitate ordinea de apelare a constructorilor obiectelor încuibărite.

Din acest motiv nu se recomandă scrierea unor secvențe de genul:

```

vector2::vector2( int x1, int y1, int x2, int y2)
    : origine( x1++, y1++ ), varf( x1 + x2, y1 - y2 )
{
    //....
};

```

În ceea ce privește destructorii claselor încuibărite se impun două observații pe care le vom discuta pe baza exemplului următor:

```

class baza
{
    class incuib1 ob1;
    class incuib2 ob2;
public:
    ~baza();
} ob0;

```

**Observatii**

- Apelul destructorilor obiectelor ob1 și ob2 se va face după execuția destructorului ~baza().
- În cazul în care în loc de obiecte încuibărite se folosesc membri de tipul pointer la obiect (încuibărire indirectă), va fi necesară atât alocarea dinamică, cât și eliberarea explicită a memoriei necesare pentru obiectele la care vor pointa membrii în cauză.

De obicei aceste instrucțiuni au loc în cadrul constructorului, respectiv destructorului clasei "cuib".

```
//----- Fisierul punct.h -----  
#ifndef _PUNCT.H_  
#define _PUNCT.H_  
class Punct  
{  
private:  
    int x,y; //abscisa si coordonata  
public:  
    Punct(); //constructorul implicit al clasei  
    Punct(int _x,int _y); //constructor cu argumente  
    ~Punct(); //destructorul clasei  
    int GetX(void) { return x; } //intoarce valoarea membrului x  
    int GetY(void) { return y; } //intoarce valoarea membrului y  
};  
#endif
```

```
//----- Fisierul punct.cpp -----  
#include "punct.h"  
#include <iostream.h>  
Punct::Punct()  
{  
    cout<<"\nApel constructor implicit clasa Punct x=0 y=0";  
    x=y=0;  
}  
Punct::Punct(int _x,int _y)  
{  
    cout<<"\nApel constructor clas Punct x="<<_x<<" y="<<_y;  
    x=_x;  
    y=_y;  
}  
Punct::~~Punct()  
{  
    cout<<"\nApel destructor clasa Punct pentru punctul de coordonate ("  
        <<x<<" "<<y<<")";  
}
```

```
//----- Fisierul segment.h -----  
#ifndef _SEGMENT.H_  
#define _SEGMENT.H_  
#include "punct.h"  
class Segment  
{  
private:  
    Punct p1,p2; //membri obiecte apartinand clasei punct
```



```
public:
    Segment(); //constructorul implicit al clasei segment
    Segment(int _x1, int _y1, int x2, int y2); //constructor cu argumente
    ~Segment(); //destructorul clasei
    double Lungime(void); //intoarce lungimea segmentului de dreapta
};
#endif
```

```
//----- Fisierul segment.cpp -----
#include "segment.h"
#include <iostream.h>
#include <math.h>
Segment::Segment()
{
    cout<<"\nApel constructor implicit clasa Segment";
}
Segment::Segment(int _x1, int _y1, int _x2, int _y2): p1(_x1, _y1), p2(_x2, _y2)
{
    cout<<"\nApel constructor clasa Segment: P1("<<_x1<<","<<_y1<<") P2("
        <<_x2<<","<<_y2<<");
}
Segment::~~Segment()
{
    cout<<"\nApel destructor clasa Segment";
}
double Segment::Lungime(void)
{
    return sqrt(pow(abs(p1.GetX()-p2.GetX()),2)+pow(abs(p1.GetY()-
        p2.GetY()),2));
}
```

```
//----- Fisierul main.cpp -----
#include "segment.h"
#include <iostream.h>
void main(void)
{
    Segment s1;
    Segment s2(1,5,8,6);
    cout<<"\n\nLungimea segmentului="<<s2.Lungime()<<endl;
}
```

3 Temă

Să se construiască clasa Matrice, în care liniile matricei sunt obiecte de tip Sir. În cadrul acestei ultime clase vom avea elementele de tip double, iar pentru clasa Sir se vor dezvolta metode de operare cu siruri:

1. adunarea a doua siruri,
2. adunarea unei valori la fiecare element a sirului,
3. suma produselor elementelor a două siruri,
4. etc



Clasa Matrice va conține operațiile elementare pe matrici (adunare, scadere, înmulțire) iar pentru realizarea acestor operații se vor folosi metodele deja dezvoltate în cadrul clasei Sir.

Clasele Sir și respectiv Matrice vor conține date alocate dinamic iar pentru o bună gestiune a memoriei se va utiliza variabile statice ce vor fi incrementate la fiecare alocare de memorie și respectiv decrementate la fiecare dealocare de memorie.

La sfârșitul programului se va face o analiză pentru verificarea eventualelor scăpări în gestiunea memoriei.

Laboratorul 6

1 Scop

Materialul prezentat in cadrul laboratorului este orientat pe tehnica supraincararii operatorilor, in cadrul limbajului C++. Se incepe cu prezentarea operatorilor binari, supradefinirea facandu-se prin functii membre claselor cat si prin functii prietene.

2 Bazele teoretice

2.1 Dilema `<iostream>` sau `<iostream.h>`

In ultimul timp gasim tot mai des in mediile moderne de programare, in programele C++, includerea bibliotecii `iostream` astfel: `#include <iostream>`. Evident, ca nepunem intrebarea: este corect `#include <iostream>` sau `#include <iostream.h>`?

Avem posibilitatea de a utiliza ambele variante, intr-un program scris in C++, doar ca utilizarea „bibliotecii” `iostream.h` (`fstream.h`) este inechita si nu este in concordanta cu standardul curent pentru limbajul C++. Varianta initiala (`#include <iostream.h>`), a fost dezvoltata la Bell Labs¹ de catre Bjarne Stroustrup² si a fost folosita pe compilatorul original CFront³. Acesta prima varianta a librarii a fost descrisa in prima editiei a cartii lui Stroustrup „*The C++ Programming Language*”, librerie ce rezida in fisierele `iostream.h`, `fstream.h`, etc. Din acel moment orice alt compilator creat a copiat continutul acestor fisiere.

Dupa ce standardizarea limbajului C++ s-a realizat, au fost produse modificari, atat in nucleul acestui limbaj cat si in sistemul de librarii. Pentru a se putea folosi ambele sisteme de librarii, s-a ajuns la conventia ca, atunci cand dorim sa utilizam librariile standardizate sa realizam includerea prin secventa `#include <iostream>`, iar cand dorim sa folosim vechiul standard sa realizam includerea in maniera `#include <iostream.h>`.

2.1.1 Tehnici de migrare

Deoarece exista foarte mult cod scris in vechiul stip C++ s-a ajuns la concluzia ca ar fi util sa se definesca un set de reguli prin care putem porta codul respecti, astfel:

- In noul standard, libraria `iostream` este iclusa in sapatul de nume `std`. Asta inseamna ca la utilizarea acestei librarii vom folosi „`using namespace std`” sau vom prefixa fiecare entitate din acesta lirie cu `std::` (`std::cout << x << std::endl;`).

¹ <http://www.alcatel-lucent.com/wps/portal/BellLabs>

² http://ro.wikipedia.org/wiki/Bjarne_Stroustrup

³ <http://en.wikipedia.org/wiki/Cfront>



În continuare este prezentate metode prin care trecem de la versiunea originală a unui program C++ la tehnica modernă de accesare a bibliotecii iostream.

A. Programul original „hello.cpp”

```
#include <iostream.h>

int main(int, char **)
{
    cout << "Hello World!" << endl;
    return 0;
}
```

B. Tehnica 1. Prefixarea elementelor de bibliotecă cu std::

```
#include <iostream>

int main(int, char **)
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

C. Tehnica 2. Utilizarea directivei using namespace

```
#include <iostream>

int main(int, char **)
{
    using namespace std;
    cout << "Hello World!" << endl;
    return 0;
}
```

sau

```
#include <iostream>

int main(int, char **)
{
    using std::cout;
    using std::endl;
    cout << "Hello World!" << endl;
    return 0;
}
```

2.2 Supraincercarea operatorilor

Clasele furnizează o facilități de reprezentare a obiectelor neprimitive, împreună cu un set de operații care pot fi efectuate cu astfel de obiecte. Se permite programatorului să furnizeze o notăție mai convențională și mai convenabilă pentru a manipula obiectele unei clase. Acest lucru se realizează prin supraincercarea operatorilor.

Trebuie evidențiat faptul că operatorii sunt deja supradefiniți, fie pentru a putea opera asupra mai multor tipuri de bază (de exemplu operatorii aritmetici admit ca operand orice tip numeric), fie pentru



a efectua mai multe operații matematice (de exemplu $*$ este asociat înmulțirii dar și referirii datelor de tip *pointer*), operația adecvată fiind selectată de compilator în funcție de tipul operanzilor.

Procedeul de supradefinire constă în definirea unei funcții (*membra sau prietena clasei*), cu numele **operator <simbol>(...)**, unde:

- **operator** este un cuvântul cheie dedicat, specific limbajului C++,
- **<simbol>** este simbolul oricărui operator C++ ($+$ $-$ $*$ $/$ $%$ $!$ \sim $<<$ $>>$, etc),
- mai puțin următorii (*punct*) $.$ $*$ $::$ $?:$).

2.3 Reguli utilizate în supradefinire

Supradefinirea operatorilor este supusă în C++ unor restricții:

1. Se pot supradefini numai operatori existenți, deci simbolul asociat funcției operator trebuie să fie definit deja ca operator pentru tipurile standard.
2. Următorii patru operatori nu pot fi supradefiniți: $.$ $*$ $::$ $?:$
3. Nu se pot modifica următoarele caracteristici ale operatorilor (*definite pentru tipurile standard*):
 - a. pluralitatea (nu se poate defini un operator unar ca operator binar și invers).
 - b. precedența și asociativitatea
4. Funcția operator trebuie să aibă cel puțin un parametru de tipul clasei a căreia îi este asociat operatorul supradefinit. Această restricție implică faptul că supradefinirea este posibilă numai pentru tipurile de clasă definite de programator, pentru tipurile standard operatorii își păstrează definiția.
5. Pentru operatorii $\{ =, [], (), -> \}$, funcția operator trebuie să fie membră nestatică a clasei.
6. O funcție operator care acceptă un tip de bază ca primul operand nu poate să fie o funcție membru.

Exemplu

- Considerăm cazul sumării unei variabile complexe **cc** la întregul **2**: **2 + cc**;
- Expresia **cc + 2** poate, printr-o funcție membru, să fie interpretată de tipul **cc.operator+(2)**; Operația **2 + cc** nu poate fi realizată, deoarece nu există o clasă **int**, pentru care să se definească operatorul **+**, operator ce poate fi interpretat de tipul: **2.operator+(cc)**;
- Compilatorul nu poate implementa comutativitatea în mod automat, între operanzi, astfel încât să substituie secvența **2 + aa** cu **aa + 2**;
- Problema se rezolvă ușor folosind funcții prietene (**operator+(2, cc)**).

2.4 Operatorii aritmetici

În cazul supraîncărcării operatorilor, prin funcții membre sau prietene, operatorii aritmetici ($+$ $-$ $*$ $/$ $%$) sunt cei mai sugestivi și ușor de implementat. Sintaxa prin care supradefinim unul din acești 5 operatori, prin funcții membre sau prietene clasei *Complex*, este următoarea:

```
class Complex
{
    // ...
public:
    // supradefinire prin funcție membru
    Complex& operator+( Complex& );
};
```



```
// supradefinire prin functie membra  
friend Complex& operator+( float, Complex& );  
};
```

2.5 Operatorii de incrementare/decrementare

Operatorii de incrementare și decrementare sunt operatori unari în număr de patru: `++/--`, *pre/post*, *incrementare/decrementare*. Datorită simbolurilor comune atașate, operatorilor de incrementare pe de o parte și operatorilor de decrementare pe de altă parte, pentru a ajuta compilatorul să discearnă care formă (prefixată sau postfixată) a operatorilor este supraincercata, se va specifica un argument suplimentar, de tipul `int`, în semnătura operatorului.

Asupra tipului de retur al operatorilor de incrementare și decrementare nu există nici un fel de restricții speciale dar având în vedere semnificația consacrată operatorilor se recomandă ca tipul de retur să fie tipul pentru care au fost redefiniți sau o referință la acesta.

```
class Complex  
{ // ...  
public:  
    // supradefinirea operatorului pre-incrementare  
    void operator++( void );  
    // supradefinirea operatorului post-incrementare  
    void operator++( int );  
};
```

2.6 Operatorul de indexare

Operatorul de indexare este un operator binar, care poate fi redefinit pentru un tip utilizator, cu restricția să apară definit ca metodă a tipului respectiv. Definițiile globale sau de tip static sunt interzise. Primul operand este instanța pentru al cărei tip a fost redefinit operatorul, iar al doilea operand este „*indexul*”, dar poate fi de orice tip dorim.

Ca exemplu de utilizare a acestor operatori prezentăm o clasă `String`, având ca și data membra un sir de caractere. Operatorul `[]` are rolul, în acest caz, de a accesa un element al sirului în modul „*read/write*”. Acest lucru este posibil deoarece, acest operator, returnează o referință către un element al sirului.

```
#include <iostream>  
using namespace std;  
  
class String  
{  
    char s[80];  
public:  
    char& operator[]( unsigned int idx );  
    friend ostream& operator<<( ostream &c, String &str )  
    { c << str.s; }  
};  
char& String::operator[]( unsigned int idx )  
{  
    return idx >= 0 && idx < 80 ? s[ idx ] : s[ 0 ];  
}  
  
int main( void )
```



```
{
    String str;
    str[0] = 'P'; str[1] = 'O'; str[2] = 'O'; str[3] = '\\0';
    cout << str << endl;
    return 0;
}
```

2.7 Operatorii de inserare/extragere din flux

În acest caz vorbim despre operatorii `<<` și `>>`, operatori ce vor lucra în asociere cu un obiect din clasele `ostream` (`cout`) sau `istream` (`cin`). Operatorii vor avea întotdeauna obiectul `cout/cin` ca prim operand, iar obiectul din clasa noastră ca și operand secund. Acești operatori pot fi supraincarcati doar ca și funcții globale, prietene clasei, astfel:

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

class Complex
{
    double re, im;
public:
    friend ostream& operator<< ( ostream &c, Complex &str );
    friend istream& operator>> ( istream &c, Complex &str );
};

ostream& operator<< ( ostream &c, Complex &str )
{
    c << "Nr: " << str.re << " +i " << str.im;
    return c;
}

istream& operator>> ( istream &c, Complex &str )
{
    cout << "Introdu nr. complex (re, im ): " << endl;
    c >> str.re; c >> str.im;
    return c;
}

int main( void )
{
    Complex A;
    cin >> A;
    cout << A << endl;
    return 0;
}
```

2.8 Operatorii de asignare

În cadrul operatorilor de asignare intră operatorii de asignare asociată (`+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `>>=`, `<<=`) și operatorul de atribuire (`=`). Operatorii de asignare sunt operatori binari. Asupra tipului de retur al operatorilor de asignare nu există restricții speciale dar este indicat ca acesta să fie un tip referință la tipul pentru care sunt redefiniți pentru a permite înlanțuirea succesivă a operatorilor. Se returnează referință deoarece nu returnăm un obiect automatic, ci obiectul curent, respectiv pe cel primit ca argument în funcția operator.



```
Test a, b;
a = b; // a.operator=( b );
// ...
Test& Test::operator=( Test& b )
{
    if( this == &b ) //evita autocopiarea
        return *this;
    // prelucrari necesare la copiere
    return *this; // obiectul din stangaatribuirii
    //return b;    // obiectul din dreapta atribuirii
};
```

În lipsa redefinirii operatorului de atribuire pentru un tip utilizator va fi generat unul implicit de către compilator (deci operatorul de atribuire este gata supraincărcat) cu semnătura

```
Test& Test::operator=( const Test& )
```

astfel încât la atribuirea obiectelor de tipul Test să se realizeze o copie de tip membru la membru, unde CClasa este un tip oarecare.

Atât operatorii de asignare asociată cât și operatorul de atribuire pot fi redefiniți ca metode virtuale. Este recomandat ca redefinirea operatorilor de asignare asociată pentru un tip să păstreze semnificația operatorilor asociați asignării, în caz că aceștia sunt redefiniți pentru tipul respectiv.

Deși este permisă definirea operatorilor de asignare asociată ca funcții de tip friend, situațiile care cer o astfel de tratare sunt foarte rare. Secvența de program de mai jos, de exemplu, este cam deplasată și nu are sens decât prin atribuirea unor funcționalități diferite de cele încetățenite operatorului +=:

```
int el;
CSet set;
/* . . .
 * reuniunea unui element cu o multime și
 * atribuirea rezultatului la un element ?
 */
el += set;
```

3 Temă

Pentru clasele Matrice/Vector, din tema din laboratorul anterior, supraincarcati urmatorii operatori:

- operatorul de indexare [], pentru accesul la un elemnt al Vectorului sau al Matricii;
- operatorii +, -, * pentru a realiza operatiile de adunare, scadere si inmultire intre doi/doua Vectori/Matrici;
- operatorii de inserare si extragere din flux (<<, >>), avand semnificatia standard;
- operatorul de atribuire, daca simtiti nevoia.

Laboratorul 7

1 SCOP

Acest laborator finalizează seria redefinirii operatorilor punând în discuție operatorii speciali ai limbajului C++.

Tot aici se face o prezentare a principiului de clasă iterator cât și implementarea acestuia prin intermediul operatorului apel de funcție.

În final sunt prezentați operatorii new, delete cât și operatorul de selecție a membrilor.

2 BAZELE TEORETICE

2.1 Operatorul apel de funcție

Operatorul apel de funcție este un operator binar care poate fi redefinit pentru orice tip utilizator doar ca metodă nestatică. Primul operand este instanța de apel iar al doilea operand e ceva mai deosebit, fiind de fapt o listă de argumente care poate fi vidă, cu un singur argument sau cu mai multe argumente. Asemănător operatorului de indexare, nici operatorul apel de funcție nu este definit pentru tipurile fundamentale. Operatorului apel de funcție ca tip de retur nu se impun nici un fel de restricții speciale.

Numele operatorului este înșelător dar definirea operatorului pentru un tip nu are nici o influență asupra celorlalte metode sau funcții definite în cadrul programului. Se numește astfel datorită simbolului operatorului care este același cu cel folosit în cazul apelurilor de funcții dar modul de aplicare a operatorului diferă față de un apel de funcție.

```
<tip_retur> operator()( <lista_argumente> );
```

2.2 Operatorul de selecție a membrilor



Operatorul de selecție indirectă a membrilor nu este supraîncărcat pentru tipurile fundamentale ale limbajului C++ dar este permisă supraîncărcarea lui pentru orice tip utilizator. Este un operator unar și poate fi redefinit doar ca metodă iar tipul de retur al operatorului este restricționat la *CClasa**, unde *CClasa* este tip utilizator oarecare, pentru care este supradefinit operatorul. Operatorul este folosit la implementarea a ceea ce în literatura de specialitate se numește *smart pointers* sau clase de tip *handle* a căror comportare este asemănătoare tipurilor indicator. Motivul principal al redefinirilor operatorului de selecție indirectă a membrilor pentru un tip utilizator este posibilitatea specificării unor sarcini colaterale de îndeplinit odată cu utilizarea operatorului în conjuncție cu instanțele clasei care-l redefineste.

```
<pointer_la_clasa> operator-> ( );
```

Pentru o clasă *CClasa*, pentru care am redefinit *operatorul ->*, putem avea expresiile:

```
class CClasa;          // Declarație incompletă
CClasa a, *p;
p = new CClasa;
p->Functie(); // Normal
a->Functie(); // Atentie!!!! Operatorul -> în asociere cu un obiect
a.Functie();  // Normal
delete p;
```

2.3 Redefinirea operatorilor new și delete

Operatorii de gestiune a memoriei dinamice sunt gata supraîncărcați pentru toate tipurile limbajului C++ și pot fi redefiniți pentru orice tip utilizator. Versiunile gata supraîncărcate ale operatorilor new și delete sunt funcții de tip global și vor fi utilizate în conjuncție cu toate tipurile fundamentale, tipurile derivate sau tipurile utilizator care nu au redefinit operatorii respectivi.

Dacă o clasă redefineste operatorii de gestiune a memoriei dinamice atunci alocarea și eliberarea memoriei în vederea instanțierii și distrugerii obiectelor se face pe baza noilor definiții ale operatorilor. Acest comportament poate fi evitat explicit - prin utilizarea operatorului de rezoluție :: - sau va fi *automat evitat* în cazul *instanțierii unui tablou de obiecte*. La alocarea și eliberarea memoriei în vederea instanțierii unui tablou de elemente se vor apela versiunile globale ale operatorilor new și delete indiferent de existența unor redefiniri ale operatorilor pentru tipul elementelor tabloului.

Operatorii de gestiune a memoriei dinamice sunt redefiniți pentru un tip ca metode statice chiar și în lipsa menționării cuvântului cheie static în semnătura metodei. Aceasta deoarece operatorii acționează asupra tipului și nu a instanțelor tipului: aplicarea operatorului new are loc înainte instanțierii dinamice prin intermediul constructorului, deci când instanța încă nu există, iar aplicarea operatorului delete are loc după distrugerea dinamică a instanțelor prin intermediul destructorului, deci când instanța nu mai există.

Limbajul C++ permite și supraîncărcarea versiunilor globale ale operatorilor de gestiune a memoriei dinamice. Prin supraîncărcarea versiunii globale a operatorului new se poate controla alocarea memoriei în vederea instanțierii dinamice a tipurilor fundamentale, a



tipurilor derivate, a tipurilor utilizator care nu au redefinit operatorul precum și a instanțierii tablourilor de elemente indiferent de categoria tipului elementelor tabloului. Același efect are și supraîncărcarea versiunii globale a operatorului `delete` cu diferențele de rigoare. În continuare prin expresiile `::new` și `::delete` vom înțelege operatorii globali, indiferent dacă sunt gata definiți de compilator sau supraîncărcați de utilizator iar prin expresiile `CClasa::new` respectiv `CClasa::delete` vom înțelege operatorii redefiniți de utilizator pentru tipul `CClasa`.

```
void* operator new( size_t lungime );  
void operator delete ( void* adresa [, size_t lungime] );
```

În acest caz *lungime* reprezintă dimensiunea zonei de memorie ce urmează a fi alocată și este calculată, respectiv transmisă ca parametru, automat de către compilator. Parametrul *adresa*, reprezintă adresa zonei de memorie ce urmează a fi dealocată. Pentru operatorul `delete` al doilea parametru este opțional și acesta fiind plasat automat de către compilator, dacă utilizatorul definește operatorul în acest mod. Cel puțin compilatorul Borland C++ 3.1, nu acceptă o redefinire a operatorului `delete` în acest mod.

3 STUDI DE CAZ. Clasa FileStr

Ca exemplu prezentăm o clasă `FileStr`, clasă ce implementează noțiunea de sir de caractere, prin intermediul unui fișier. Astfel toate elementele sirului sunt pastrate în fișier, iar operarea asupra unui element din `FileStr`, va conduce la operare asupra fișierului pe o anumită poziție.

Interesant pentru această clasă este următoarea secvență:

```
FileStr f( "test.dat" ); // unde "test.dat" este fisierul asociat  
char x ;  
f[ 10 ] = 'A'; // ca efect se va pune in fisier pe pozitia 10  
caracterul 'A'  
x = f[ 10 ]; // va fi extras din fisier de pe pozitia 10 un  
caracter si  
// va fi atribuit lui x
```

Se va observa că operatorul de indexare returnează obiectul curent (nu înainte de a poziționa pointerul de fișier pe poziția specificată). Din acest moment evenimentele se desfășoară diferit pentru cele două atribuiri prezentate mai sus. Astfel pentru `f[10] = 'A'` se va apela operatorul de atribuire de către obiectul returnat de operatorul de indexare, având ca argument caracterul 'A', caracter ce va fi depus pe poziția specificată. În celălalt caz, se va apela implicit operatorul *cast* (*const char*), de conversie la caracter, efectul fiind returnarea caracterului din fișier, de pe poziția stabilită de operatorul de indexare.

Fisierul FILESTR.H

```
#ifndef FILESTR_H  
#define FILESTR_H  
  
#include <iostream>
```



```
#include <fstream>
//#define _OP_GLOBALI_ // sunt validati operatorii globali redefiniti
//#define _OP_MEMBRI_ // sunt validati operatorii membri redefiniti

using namespace std;

class FileStr
{
private:
    fstream fp; // obiect de tipul fstream (fisier intrare/iesire)
    char nume[30]; // numele fisierului asociat FileString-ului
public:
    FileStr( const char* name ); // constructor de conversie
    ~FileStr(); // destructor
    FileStr& operator[] ( int loc ); // operatorul de indexare.
    // Atentie la ce returneaza
    void operator=( char c ); // atribuirea unui caracter
    void operator=( const char* str ); // atribuirea unui sir de caractere
    operator const char(); // operatorul cast, de conversie la
    caracter
    FileStr* operator->( void ); // operatorul de accesare a membrilor
    int operator()( int i, int j, char *sir );
#ifdef _OP_MEMBRI_
    void* operator new( size_t );
    void operator delete( void* );
#endif // _OP_MEMBRI_
    void Print( void );
};
#endif // FILESTR_H
```

Fisierul FILESTR.CPP

```
// Fisier: filestr.cpp - Trateaza un fisier ca pe un sir
#include <iostream>
#include <new>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include "FileStr.h"
using namespace std;

void f_tratare( );
// Constructorul clasei
FileStr::FileStr( const char* name)
{
    fp.open(name, ios::in | ios::out ); // deschide fisierul
    if( fp.fail() ) // daca esueaza deschiderea...
    {
        cerr<<"Eroare deschider fisier <"<<name<<">"<<endl;
        getch();
        exit(0);
    }

    strcpy( nume, name);
}
// Destructorul clasei
```




```
FileStr::~FileStr()
{
    fp.close();
}
// operator[] pozitioneaza fisierul si creaza o instanta a clasei File

FileStr& FileStr::operator[]( int loc)
{
    fp.seekp( loc ); // pozitionare in fisier, functie de indexul primit
    return *this; // este returnat obiectul curent
}
//-----
// operator = ( char )
// pentru atribuirii de forma:
// f[n] = c, unde f[n] este un fstream si c este un char
void FileStr::operator=(char c)
{
    if( !fp.eof() ) //este pus caracterul in fisier
        fp.put( c );
}
//-----
// operator = ( char * )
// pentru atribuirii de forma:
// f[n] = "string", unde f[n] este un obiect File
void FileStr::operator=(const char* str)
{
    if( !fp.eof() ) //este pus un intreg sir in fisier
        fp.write( str, strlen( str ) );
}
//-----
// operator const char ( )
// pentru atribuirii de forma:
// c = f[n], unde f[n] este un File si c este un char
// Acesta citeste un caracter din fisier si returneaza valoarea sa
FileStr::operator const char()
{
    if( !fp.eof() )
        return fp.get( ); // se varetorna caracterul citit
    return EOF;
}

void FileStr::Print( void )
{
    long pos = fp.tellp();
    char c;
    fp.seekp( 0, ios::beg );
    if( fp.fail() )
    {
        fp.close();
        fp.open( nume, ios::in | ios::out );
        if( fp.fail() )
            return;
    }
    if( fp.fail() )
        return;
    while( !fp.eof() )
    {
        fp.get(c); //fstream
```



```
        cout<<c;
    }
    fp.seekp( pos );
}
// SMART POINTERS, apel de genul: <obiect> -> <membru>
FileStr* FileStr::operator->( void )
{
    // pot fi facute diferite teste ( Ex. integritate a datelor... )
    cout<<"\nAtentie la modul de apel al Metodei <obiect>->Print();"<<endl;
    return this;
}
// Va cauta un subsir, in fisier, intre pozitiile i, j
int FileStr::operator()(int i, int j, char *sir)
{
    long pos = fp.tellp(), k;
    int flag;
    char c;
    if( fp.fail() )
    {
        fp.close();
        fp.open( nume, ios::in | ios::out );
        if( fp.fail() )
            return -1;
    }
    for( ; i<j; i++ )
    {
        fp.seekp( i, ios::beg );
        k = i;
        flag = 0;
        while( k - i < strlen( sir ) )
        {
            if( k >= j )
                return -1;
            c = (char)fp.get();
            if( c != sir[k-i] )
            {
                flag = 1;
                break;
            } // end if
            k++;
        } // end while
        if( flag == 0 )
            break;
    } // end for
    fp.seekp( pos, ios::beg);
    return flag == 0 ? i : -1;
}

#ifdef _OP_MEMBRI_
// operatorul new mebru al clasei
void* FileStr::operator new( size_t lng )
{
    cout<<"\n__Operatorul NEW al Clase File__";
    void * p = ::new char[lng];
    cout<<"\nNEW_File: alocat "<<lng<<" octeti";
    if( p == NULL )
        f_tratare();
    cout<<" - adresa: "<< p;
```



```
        return p;
    }
    // operatorul delete mebru al clasei
    void FileStr::operator delete( void* adr )
    {
        cout<<"\n___Operatorul DELETE al Clasei File___";
        ::delete adr;
        cout<<"\nDELETE_File: dealocat la adresa - "<<adr;
    }
#endif // _OP_MEMBRI_
```

Fisierul MAIN.CPP

```
// Un program de test pentru clasa File
// Inainte de a rula va trebui sa creati fisierele "test.dat"
// cu urmatoarea linie: Testare, clasa File-0123456789
// si "test.txt" cu urmatoarea linie: Hello, un fisier de test
#include <iostream>
#include <conio.h>
#include <stdlib.h>
#include <new>
#include "filestr.h"

using namespace std;

// functie ce se apeleaza la eroare de alocare memorie ( _new_handler )
void f_tratare( )
{
    cout<<"Eroare alocare memorie!"<<endl;
    exit( 0 );
}

#ifdef _OP_GLOBALI_
// supradefinit operatorul new global
void* operator new ( size_t lng )
{
    cout<<"\n___Operatorul NEW global___";
    void * p = malloc( lng );
    cout<<"\nNEW_Global: alocat "<<lng<<" octeti";
    if( p == NULL )
        f_tratare();
    cout<<" - adresa: "<< p;
    return p;
}
// supradefinit operatorul delete global
void operator delete( void * adr )//, size_t lng)
{
    cout<<"\n___Operatorul DELETE global___";
    free( adr );
    cout<<"\nDELETE_Global: dealocat la adresa - "<<adr;
}
#endif // _OP_GLOBALI_

int main( void )
{
    //void (*pf)( );//declar pointerul pf
```



```
/// pointer la o functie void fara parametri, in biblioteca <new> -  
/// typedef void (*new_handler)();  
std::new_handler pf;  
  
///stabilesc noul handler pentru functia de tratare err. alocare  
pf = std::set_new_handler( f_tratare );  
system( "cls" );  
  
char *p;  
p = new char[1000]; // incercare de alocare. Atentie la mesaje  
delete p; // dealocare memorie. Atentie la mesaje  
cout<<"I. Apasa o tasta!"<<endl;  
getch();  
FileStr f( "test.dat" );  
int i;  
char c;  
system( "cls" );  
cout << "Primii 30 octeti sunt:" << endl;  
for(i=0; i<30; i++)  
{  
    c = f[i];  
    cout << c;  
}  
cout << endl;  
cout<<"II. Apasa o tasta!"<<endl;  
getch();  
// Schimba primii 7 octeti cu '_' (linie de subliniere)  
for(i=0; i<7; i++)  
    f[i] = '_'; // se scrie in fisier caracterul _. !!!!!  
// Afiseaza primii 14 octeti din nou  
cout << "Acum primii 30 octeti sunt:" << endl;  
for(i=0; i<30; i++)  
{  
    c = f[i]; // citeste cate un caracter din fisier. !!!!!  
    cout << c;  
}  
cout << endl;  
cout<<"III. Apasa o tasta!"<<endl;  
getch();  
// Inmagazineaza un sir intr-un fisier  
f[0] = "Testare:"; // operatorul de atribuire redefinit. !!!!!  
cout << "Dupa inserarea sirului: primii 30 octeti sunt:" << endl;  
for(i=0; i<30; i++)  
{  
    c = f[i];  
    cout << c;  
}  
cout << endl;  
f->Print(); // Operatorul -> redefinit. !!!!!  
cout << endl;  
f.Print(); // Accesul normal la membrul clasei. !!!!!  
cout<<"IV. Apasa o tasta!"<<endl;  
getch();  
cout << endl;  
char sir[] = "clasa";  
cout<< "Sirul <"<< sir << "> incepe de pe pozitia: "<<f(8, 20, sir);  
cout << endl;  
FileStr *pF;
```



```
pF = new FileStr("test.txt");
cout << "\nAfiseaza un nou obiect <test.txt>"<<endl;
pF->Print();
delete pF;

///restabilesc vechiul handler pentru functia de tratare err. alocare
std::set_new_handler( pf );
cout<<"\nSFARSIT program"<<endl;
cout<<"V. Apasa o tasta!"<<endl;
getch();
return 0;
}
```

OUTPUT

```
__Operatorul NEW global__
NEW_Global: alocat 1000 octeti - adresa: 0x922ae0
__Operatorul DELETE global__
DELETE_Global: dealocat la adresa - 0x922ae0
```

```
Primii 30 octeti sunt:
Testare: clasa File-0123456789
Acum primii 30 octeti sunt:
__ : clasa File-0123456789
Dupa inserarea sirului: primii 30 octeti sunt:
Testare: clasa File-0123456789

Atentie la modul de apel al Metodei <obiect>->Print();
Testare: clasa File-01234567899

__Operatorul DELETE global__
DELETE_Global: dealocat la adresa - 0x922b20
__Operatorul NEW global__
NEW_Global: alocat 512 octeti - adresa: 0x922b20

__Operatorul DELETE global__
DELETE_Global: dealocat la adresa - 0x922b20
__Operatorul NEW global__
NEW_Global: alocat 512 octeti - adresa: 0x922b20Sirul <clasa> incepe de pe p
ia: -1

__Operatorul NEW al Clase File__
__Operatorul NEW global__
NEW_Global: alocat 288 octeti - adresa: 0x922d28
NEW_File: alocat 288 octeti - adresa: 0x922d28
Process returned 0 (0x0) execution time : 63.554 s
Press any key to continue.
```

4 TEMĂ

Pentru clasa *FileStr* prezentată în acest laborator să se atingă următoarele obiective:



- Verificarea scurgerilor de memorie, prin utilizarea unei liste în care vor fi inserate toate adresele alocate și respectiv de unde vor fi eliminate acele adrese, pentru care s-a realizat o dealocare.
- Stabilirea modului de lucru DEBUG și respectiv RELEASE, clasa (chiar programul) având comportament diferit. Astfel în modul DEBUG vor fi urmărite *scurgerile de memorie*, raportând la sfârșitul programului anomaliile întâlnite. În varianta RELEASE, programul se va comporta normal fără verificări suplimentare, ale *scurgerilor de memorie*.

Laboratorul 8

1 SCOP

Laboratorul atinge pentru prima data notiunea de mostenire si face o prezentare a acestui mecanism ce sta la baza principiilor de Programare Orientata pe Obiecte (POO).

In cadrul laboratorului se dezvoltă o ierarhie de clase ce pornesc de la clasa de baza *Lista*, implementata prin noduri simplu inlantuite.

2 BAZELE TEORETICE

2.1 Principiul Mostenirii

Conceptul de mostenire este o notiune foarte naturala si pe care o intalnim in viata de zi cu zi. In C++ intalnim notiunea de derivare, care este in fapt o abstractizare a notiunii de mostenire. O clasa care adauga proprietati noi la o clasa deja existenta vom spune ca este derivata din clasa originala. Clasa originala poarta denumirea de clasa de baza.

Clasa derivata mosteneste toate datele si functiile membre ale clasei de baza; ea poate adauga noi date la cele existente si poate suprascrie sau adauga functii membre. Clasa de baza nu este afectata in nici un fel in urma acestui proces de derivare si ca urmare nu trebuie recompilata. Declaratia si codul obiect sunt suficiente pentru crearea clasei derivate, ceea ce permite reutilizarea si adaptarea usoara a codului deja existent, chiar daca fisierul sursa nu este disponibil. Astfel, nu este necesar ca programatorul unei clase derivate sa cunoasca modul de implementare a functiilor membre din componenta clasei de baza.

O notiune noua legata de derivare este cea de supraincarcare sau suprascriere a functiilor membre. Aceasta se refera, in mod evident, la redefinirea unor functii a clasei de baza in clasa derivata. De notat este faptul ca functiile originale din clasa parinte sunt in continuare accesibile in clasa derivata, deci caracteristicile clasei de baza nu sunt pierdute.

Dintr-o clasa de baza pot fi derivate mai multe clase si fiecare clasa derivata poate servi mai departe ca baza pentru alte clase derivate. Se poate astfel realiza o ierarhie de clase, care sa modeleze adecvat sisteme complexe. Pornind de la clase simple si generale, fiecare nivel al ierarhiei acumuleaza caracteristicile claselor "parinte" si le adauga un anumit grad de



specializare. Mai mult decat atat, in C++ este posibil ca o clasa sa mosteneasca simultan proprietatile mai multor clase, procedura numita mostenire multipla.

Sintaxa simplificata a derivarii este:

```
class NumeClDerivata : [ [virtual][specif_acces] NumeClBazal[ , ...] ]
{
    /*
     * Declaratii date membre proprii
     */
    /*
     * Declaratii metode proprii
     */
} [ lista_declaratii_obiecte ];
```

2.2 Exemplu de Mostenire

In cadrul exemplului urmator este aratat modul in care derivam clasa *FigGeom* pentru a obtine un nou tip de data, mai specializata, tipul *Triunghi*. Noua clasa mosteneste toate metodele vechii clase, dar pentru acele metode ce nu satisfac noile cerinte, clasa introduce variante supraincarcate. Acest lucru se poate observa in cazul metodelor *Arie()* si *Perimetru()*, in clasa *Triunghi*.

```
#include <iostream>

using namespace std;
/*
 * Clasa FIGURA GEOMETRICA - generica
 */
class FigGeom
{
protected:
    char nume[ 80 ];
public:
    FigGeom( char *_nume = "{ nedefinita }" );
    double Arie( void );
    double Perimetru( void );
    char* TipFigura( void );
};

FigGeom::FigGeom( char *_nume )
{
    strcpy( nume, _nume );
}
double FigGeom::Arie( void )
{
    return 0.0;
}
double FigGeom::Perimetru( void )
```




```
{
    return 0.0;
}
char* FigGeom::TipFigura( void )
{
    return nume;
}
/*
 * Clasa TRIUNGHI - derivata din FigGeom
 */
class Triunghi : public FigGeom
{
public:
    Triunghi( char *_nume = "{ TRIUNGHI }" );
    double Arie( void );
    double Perimetru( void );
    //char* TipFigura( void ); // NU ESTE NECESARA
};

Triunghi::Triunghi( char *_nume ): FigGeom( _nume )
{
    // Constructorul pt. CB
    /* fara alte operatii ... deocamdata! */
}

double Triunghi::Arie( void )
{
    return 1.0; /* DE ABORDAT CORECT */
}

double Triunghi::Perimetru( void )
{
    return 2.0; /* DE ABORDAT CORECT */
}

/*
 * Functia MAIN
 */
int main()
{
    cout << "[ START ] Studiul mostenirii" << endl;
    FigGeom fgl;
    cout << "\nArie        figura " << fgl.TipFigura() << ": "
         << fgl.Arie() << endl;
    cout << "Perimetru figura " << fgl.TipFigura() << ": "
         << fgl.Perimetru() << endl;

    Triunghi tr1;
    cout << "\nArie        figura " << tr1.TipFigura() << ": "
         << tr1.Arie() << endl;
    cout << "Perimetru figura " << tr1.TipFigura() << ": "
         << tr1.Perimetru() << endl;
    cout << "\n[ END ]" << endl;

    /*
    Triunghi tr2( "TRIUNGHI Echilateral", 2, 2, 2 );
    */
}
```



```
cout<<"\nArie      figura " << tr2.TipFigura() << ": "
    << tr2.Arie() << endl;
cout<<"\Perimetru figura " << tr2.TipFigura() << ": "
    << tr2.Perimetru() << endl;
cout << "\n[ END ]" << endl;
*/
return 0;
}
```

OUTPUT

```
[ START ] Studiul mostenirii
Arie      figura < nedefinita >: 0
Perimetru figura < nedefinita >: 0

Arie      figura < TRIUNGHI >: 1
Perimetru figura < TRIUNGHI >: 2

[ END ]

Procesul a returnat valoarea 0, executiunea s-a terminat.
```

3 STUDI DE CAZ. Clasa Stiva

În continuare vom deriva din clasa Lista o clasa specializată, Stiva, ce va utiliza o parte din interfața pusă la dispoziție de către clasa Lista, o parte o va redefini și respectiv va mai adăuga un set de funcții specifice ei.

```
/*
 * ----- Fisierul LISTA.H -----
 */
#ifndef _LISTA.H_
#define _LISTA.H_
*/
#include <conio.h>
#include <iostream>
#include <fstream>

using namespace std;

struct Element
{
    int Val;
    Element *Urm;
};

class Lista // va fi o clasa de baza pentru alte clase derivate
{
protected: // este folosit ca zona de date privata la mostenire...
    Element *Inceput;
public: // zona de interfata
```



```
// Constructori, destructor.
Lista( void );
~Lista( );
// Operare la inceputul listei
void InsertBegin( int x );
void DeleteBegin( void );
int ReturnBegin( void );
// Operare la sfarsitul listei
void InsertEnd( int x );
void DeleteEnd( void );
int ReturnEnd( void );
// alte operatii pe lista
void InsertAfter( int ElemAfter, int x );
int IfExist( int Elem );
// Testare lista
int IsEmpty( void );
int IsFull( void ); // atentie la implementare. O alta idee...
// Afisare / Initializare lista. Aceste functii sunt utilizate si
// pentru a realiza persistenta datelor din lista.
friend ostream& operator<<( ostream& c, Lista& l);
friend istream& operator>>( istream& c, Lista& l); // Atentie!
};
/*
 * #endif // _LISTA.H_

----- Fisierul LISTA.CPP -----
*/

Lista::Lista( void )
{
    cout<<"\nConstructor Lista ..."<<endl;
    Inceput = NULL;
}

Lista::~~Lista( )
{
    Element *t = Inceput;
    while ( t )
    {
        t = t->Urm;
        delete Inceput;
        Inceput = t;
    }
    cout<<"\nDestructor Lista..."<<endl;
}

void Lista::InsertBegin( int x )
{
    Element *t;
    t = new Element;
    t -> Val = x;
    t -> Urm = Inceput;
```



```
Inceput = t;
}

void Lista::DeleteBegin( void )
{
    if ( Inceput == NULL )
        return;
    Element *t = Inceput -> Urm;
    delete Inceput;
    Inceput = t;
}

int Lista::ReturnBegin( void )
{
    if (Inceput == NULL )
        return -1;
    return Inceput -> Val;
}

void Lista::InsertEnd( int x )
{
    if ( IsEmpty( ) ) // pt. o lista goala avem o inserare la inceput
    {
        InsertBegin( x );
        return;
    }
    Element *t, *tmp = Inceput;
    t = new Element;
    t -> Val = x;
    t -> Urm = NULL;
    while ( tmp->Urm != NULL ) // cautarea sfarsitului de lista
        tmp = tmp->Urm;
    tmp->Urm = t; // legarea de catre ultimul element
}

void Lista::DeleteEnd( void )
{
    Element *tmp = Inceput;
    if ( IsEmpty( ) ) // pt. o lista goala nu vom face nimic
        return;
    if ( Inceput->Urm == NULL ) // daca am un singur nod
    {
        delete Inceput;
        Inceput = NULL;
    }
    else // pentru cel putin doua elemente
    {
        while ( tmp->Urm->Urm != NULL ) // penultimului element
            tmp = tmp->Urm;
        delete tmp->Urm; // stergere ultim element.
        tmp->Urm = NULL; // indicam sfarsitul listei
    }
}
```



```
int Lista::ReturnEnd( void )
{
    Element *tmp = Inceput;
    if ( IsEmpty( ) ) // pt. o lista goala returnam o valoare EXPLICIT
        return -1;
    while ( tmp->Urm != NULL ) // cautarea ultimului element
        tmp = tmp->Urm;
    return tmp->Val; // valoarea ultimului element
}

void Lista::InsertAfter( int ElemAfter, int x )
{
    Element * tmp = Inceput;
    int poz = IfExist( ElemAfter );
    if ( poz == 0 ) // daca nu exista elementul
        return;
    while ( poz - 1 )
        tmp = tmp->Urm, poz--;
    Element *t;
    t = new Element;
    t -> Val = x;
    t -> Urm = tmp->Urm;
    tmp->Urm = t;
}

int Lista::IfExist( int Elem )
{
    int n = 0;
    Element *tmp = Inceput;
    while ( tmp != NULL ) // cautarea elementului
    {
        n++;
        if ( tmp->Val == Elem )
            return n;
        tmp = tmp->Urm;
    }
    return 0;
}

int Lista::IsEmpty( void )
{
    return Inceput == NULL ? ~0 : 0;
}

int Lista::IsFull( void ) // atentie la implementare. O alta idee...
{
    Element *t = new Element; // se incearca alocarea unui element
    if ( t )
    {
        delete t; // se dealoca elementul alocat si ...
    }
}
```



```
        return 0; // vom returna 0 ( Lista nu este plina ).
    }
    return ~0;    // altfel, lista este plina ...
}
// Afisare / Initializare lista. Aceste functii sunt utilizate si
// pentru a realiza persistenta datelor din lista.
ostream& operator<<( ostream& c, Lista& l)
{
    Element *t = l.Inceput;

    if ( &c == &cout )
        c<<"Continutul: { "; // este transmis explicit  catre ecran
    while ( t )
    {
        c << t -> Val << " "; // este transmis catre " c ", care
        t = t -> Urm;          // poate fi si display-ul
    }
    if ( &c == &cout )
        c<< "\\b }" << endl; // ... ca si acesta!
    return c;
}

istream& operator>>( istream& c, Lista& l) // cum va fi utilizat!
{
    int t;

    while ( c.good( ) )
    {
        c >> t ;
        if ( c.fail() )
            break;
        l.InsertEnd( t );
    }
    return c;
}

/*
 * ----- Fisierul STIVA.H -----
*/
#ifdef _STIVA.H_
#include "lista.h"
*/
class Stiva: public Lista
{
public:
    Stiva(void);
    ~Stiva();
    void push(int val);
    void pop( void );
    int top( void );
    friend ostream& operator<<( ostream& c, Stiva& s);
    friend istream& operator>>( istream& c, Stiva& s);
};
```



```
/*#endif // _STIVA.H_

* ----- Fisierul STIVA.CPP -----
#include<conio.h>
#include<iostream.h>
#include"stiva.h"
*/
Stiva::Stiva(void): // apelul constructorului pt. clasa mostenita
    Lista()
{
    cout<<"\nConstructor Stiva..."<<endl;
}
Stiva::~Stiva()
{
    cout<<"\nDestructor Stiva..."<<endl;
}
void Stiva::push(int val)
{
    InsertBegin( val );
}
void Stiva::pop( void )
{
    DeleteBegin( );
}
int Stiva::top( void )
{
    return ReturnBegin();
}
ostream& operator<<( ostream& c, Stiva& s)
{
    c << (Lista &) s; // apel explicit al operat. din C.B.
    return c;         // conversia pt. a nu se apela recursiv
}
istream& operator>>( istream& c, Stiva& s)
{
    /* nu putem adopta aceeasi idee deoarece nu ar fi respectat
    * principiul de stiva. De aceea vom supradefini operatorul
    * special pentru obiecte de tip Stiva.
    */
    int t;

    while ( c.good( ) )
    {
        c >> t ;
        if ( c.fail() )
            break;
        s.push( t ); // in lista era: l.InsertEnd( t )
    }
    return c;
}
/*
* ----- Fisierul MAIN.CPP -----
```



```
#include<conio.h>
#include<iostream.h>
#include<fstream.h>
#include"lista.h"
#include"stiva.h"
    */
int main ( void )
{
    system( "cls" );
    // TEST      L I S T A
    Lista l;
    l.InsertBegin( 1 );
    l.InsertBegin( 2 );
    l.InsertBegin( 4 );
    l.InsertAfter( 4, 3 );
    cout << l;
    cout<<"Lista este salvata in 'Lista.txt': "<<endl;
    ofstream fout("lista.txt");
    fout << l; // Atentie. Referinta la ostream contine o referinta
    // la ofstream. Are loc salvarea listei in fisier!
    cout<<"Varful: "<<l.ReturnBegin()<<endl;
    l.DeleteBegin();
    cout<<l;
    l.DeleteBegin();
    l.DeleteBegin();
    l.DeleteBegin();
    cout<<"O lista vida: "<<l;
    fout.close(); // este inchis fisierul "lista.txt"
    ifstream fin( "lista.txt" ); // si este asociat obiectului f2
    fin>>l;
    cout<<"O lista initializata din fisierul 'lista.txt': "<<l;
    // TEST      S T I V A
    Stiva s;
    s.push( 101 );
    s.push( 102 );
    s.push( 103 );
    cout<<"Continutul stivei: "<<s;
    cout<<"Stiva este salvata in 'stiva.txt': "<<endl;
    fout.close(); // este inchis fisierul "lista.txt"
    fout.open("stiva.txt");// este asociat fisierul "stiva.txt"
    fout << s; // Atentie. Referinta la ostream contine o referinta
    // la ofstream. Are loc salvarea stivei in fisier!
    fout.close(); // este inchis fisierul "stiva.txt"
    fin.close(); // este inchis fisierul "lista.txt"
    fin.open("stiva.txt");// este asociat fisierul "stiva.txt"
    fin >> s; // sunt adaugate si datele din fisier
    cout<<"Continutul stivei: " << s;
    return 0;
} //end main
```


*OUTPUT*

```
Constructor Lista ...
Continutul: < 4 3 2 1 >
Lista este salvata in 'Lista.txt':
Varful: 4
Continutul: < 3 2 1 >
0 lista vida: Continutul: < >
0 lista initializata din fisierul 'lista.txt': Continutul: < 4 3 2 1 >

Constructor Lista ...

Constructor Stiva...
Continutul stivei: Continutul: < 103 102 101 >
Stiva este salvata in 'stiva.txt':
Continutul stivei: Continutul: < 103 102 101 >

Destructor Stiva...

Destructor Lista...

Destructor Lista...
```

CONCLUZII

Un lucru interesant ce apare in cadrul acestui program este utilizarea operatorilor de inserare in flux respectiv de extragere din flux a datelor, atât pentru afișarea pe ecran, respectiv citirea de la tastatură, cât și pentru salvarea, respectiv inițializarea cu date din / in fișier.

```
ofstream fout( "lista.txt" );
fout << l;
```

O atenție deosebită se va acorda și modului de apel a operatorului de inserare în flux pentru clasa *Lista*, în momentul redefinirii acestui operator pentru clasa *Stiva*.

```
c << ( Lista &) s;
```

Se observă că are loc o deplasare a adresei de început către instanța moștenită din clasa *Lista*, care din întâmplare este chiar la începutul instanței din clasa *Stiva*. Nu același lucru se întâmplă dacă conversia era făcută către un obiect și nu către o referință din clasa *Lista*.

```
c << ( Lista ) s;
```



4 TEMĂ

1. Pentru cazul clasei Triunghi, prezentata la inceputul sedintei de laborator, abordati corect calculul Ariei si al Perimetrului.
2. Studiul programului și lămurirea aspectelor prezentate în ultima parte a laboratorului;
3. Explicați comportamentul diferit al programului atunci când coversia, în cadrul operatorului de inserare în flux, se face către (Lista) și nu către (Lista&);
4. Utilizând clasa Lista ca și clasa de bază, obțineți următoarele două TDA-uri:
 - a. Coadă,
 - b. Multime.

Laboratorul 9

1 SCOP

Odata cu acest laborator se face o prezentare a celui mai interesant comportament introdus de care clase in cadrul erarhiilor obtinute prin mostenire.

Se incepe cu o trecere in revista a principalelor variante de polimorfism intalnite inca din cadrul limbajului C (functii cu numar variabil de parametri) si pana la forma cea mai avansata de polimorfism si anume polimorfismul de mostenire.

2 BAZELE TEORETICE

2.1 Polimorfismul

Polimorfismul se manifestă în limbajul C++ sub trei forme diferite: polimorfismul parametric, polimorfismul ad-hoc și polimorfismul de moștenire.

2.1.1 Polimorfismul parametric

Acesta se referă la existența funcțiilor cu număr variabil de parametri, iar un exemplu clasic de polimorfism parametric este dat de funcțiile de ieșire formatată a informației: printf, fprintf și sprintf.

Fiecare dintre aceste funcții este o entitate polimorfă parametric, în sensul posibilității apelului funcției cu un număr oarecare de argumente, de tipuri diferite.

În limbajul C++, pentru a suporta funcțiile polimorfe parametric s-a introdus un nou tip de argument, argumentul oricărui tip (. . .) - ellipsis care menționat întotdeauna ca ultim argument al unei funcții va valida apelurile de funcție cu un număr oarecare de argumente. Semnătura funcției printf în limbajul C++ are forma int printf(. . .) sau int printf(const char*, . . .). Forma a doua a semnăturii este mai sigură deoarece impune tipul primului argument de apel ca fiind



const char*. O semnătură de genul `int printf()` are o cu totul altă interpretare în limbajul C++ față de limbajul C, fiind echivalentă cu `int printf(void)`.

2.1.2 Polimorfismul ad-hoc (supraîncărcarea numelor de funcții)

Polimorfismul ad-hoc sau supraîncărcarea funcțiilor (function overloading) este tot un polimorfism al funcțiilor, introdus de limbajul C++. O funcție este supraîncărcată dacă suportă mai multe implementări distincte (versiuni) în cadrul aceluiași scop. Declarațiile de mai jos introduc două versiuni ale funcției `sqrt`, una pentru tipuri de date întregi și una pentru tipuri de date în virgulă flotantă:

```
double fRadical( double );  
long   fRadical( long );
```

Discriminarea funcțiilor se va face exclusiv în funcție de tipul parametrilor de apel, tipul de retur neavând nici o influență asupra alegerii unei versiuni anume a funcției.

Uneori în încercarea de a aplica una dintre versiunile funcțiilor supraîncărcate vor fi semnalate erori de ambiguitate. Dacă tipurile argumentelor de apel nu corespund exact cu tipurile parametrilor formali ai unei versiuni oarecare de funcție supraîncărcată, compilatorul va încerca aplicarea regulilor de conversie implicită pentru argumentele în cauză. Cazul în care nu poate fi determinată în mod unic o versiune de apel a funcției supraîncărcate (adică nici o versiune sau mai mult de una) este o eroare.

2.1.3 Polimorfismul de moștenire

Polimorfismul de moștenire, spre deosebire de celelalte forme de polimorfism, este un polimorfism al obiectelor. La baza polimorfismului obiectelor stă relația de tip IS_A existentă între subclase și superclase, obținută prin moștenire publică. În virtutea acestei relații este permisă manipularea instanțelor claselor derivate în contexte care cer instanțe ale claselor de bază:

```
class CBaza  
{  
public :  
    void Metoda( );  
    /* ... */  
};  
  
class CDerivata : public CBaza  
{  
public :  
    void Metoda( );  
    /* ... */  
};
```

```
};  
/* ...  
*/  
void f( CBaza b )  
{  
    b.Metoda( );  
}  
void g( CBaza& rB )  
{  
    rB.Metoda( );  
}  
void h( CBaza* pB )
```



```
{  
    pB->Metoda( );  
}  
/* ...  
*/
```

```
int main( void )  
{  
    CDerivata d;  
    f( d );  
    g( d );  
    h( &d );  
}
```

2.2 Funcțiile virtuale și legarea dinamică

Folosirea cuvîntului cheie `virtual` înaintea semnăturii unei funcții va desemna funcția respectivă ca fiind o funcție virtuală. Cu ajutorul funcțiilor virtuale se asigură apelul metodelor corespunzătoare tipului dinamic al obiectelor, chiar dacă acestea sunt uzate în contexte care cer tipuri diferite. Prin virtualizarea metodei `Metoda` din cadrul claselor `CBaza` și `CDerivata`:

```
class CBaza  
{  
public :  
    virtual void Metoda( );  
};  
  
class CDerivata : public CBaza  
{  
public :  
    /* nu mai este necesar specificarea "virtual" */  
    void Metoda( );  
};
```

apelurile `rB.Metoda()` și `pB->Metoda()` din funcțiile `g` și `h` vor determina execuția uneia din metodele `CBaza::Metoda` sau `CDerivata::Metoda` în funcție de tipul parametrului de apel al funcțiilor `g` și `h`, `CBaza` sau `CDerivata`.

Orice clasă care definește cel puțin o metodă virtuală este o clasă polimorfă. În exemplul anterior, atât clasa `CBaza` cât și clasa `CDerivata` sunt clase polimorfe.

Mecanismul funcțiilor virtuale este aplicat doar în conjuncție cu tipurile referință (`rB.Metoda()`) și tipurile indicator (`pB->Metoda()`) astfel încît apelul `b.Metoda()` din cadrul funcției `f` va determina, ca și pînă acum, execuția metodei `CBaza::Metoda` indiferent de tipul parametrului de apel al funcției `f`.

2.3 Destructorii virtuali

Destructorii ca orice altă funcție de tip metodă pot fi virtuali. Deoarece fiecare clasă promovează un nume unic asociat destructorului lanțul destructorilor virtuali nu va conține semnături identice de funcții, fiind de fapt singura excepție de la constrîngerile care apar la



redefinirea funcțiilor virtuale. Prin definirea destructorilor virtuali se asigură apelul corect al acestora în momentul distrugerii obiectelor, indiferent de tipul sub care sunt manipulate obiectele. Specificațiile limbajului C++ nu admit definirea constructorilor virtuali.

2.4 Funcții virtuale pure și clase abstracte

Prin metodele virtuale pure (metoda `CPrimGrafica::Traseaza`) înțelegem o metodă ce nu a fost implementată (nu a fost definită) în cadrul clasei respective și va fi desemnată ca atare prin sufixul `= 0`, prezent imediat după semnătura metodei:

```
class CPrimGrafica
{
public :
    virtual void Traseaza( ) = 0;
    /* ... */
};
```

În continuare, indiferent de context, prin metode pure vom înțelege metode virtuale pure. O clasă care declară cel puțin o metodă pură se numește clasă abstractă. Dacă o clasă derivată dintr-o clasă abstractă nu redefineste toate metodele pure definite de clasa de bază ca fiind metodele virtuale normale atunci este tot o clasă abstractă iar metodele pure moștenite din clasa de bază își vor păstra caracterul.

3 STUDI DE CAZ. Clase Figura, Punct, Segment, Triunghi

În continuare este prezentat exemplul unei arhitecturi de clase ce au la bază clasa `Figura`. Aceasta clasă este una abstractă, deoarece definește trei metode pur virtuale (`Perimetrul`, `Aria`, `Volumul`).

SOLUTIE

```
/* ----- Fisierul F I G U R A . H -----
 * #ifndef _FIGURA.H_
 * #define _FIGURA.H_
 */
#include <iostream>
#include <math.h>
using namespace std;

struct Coordonate
{
    int X;
    int Y;
};

class Figura // va fi o clasa de baza pentru alte clase derivate
```



```
{
protected: // Nu avem date private pentru aceasta clasa

public: // Declaratii cu definitiile corespunzatoare INLINE
    Figura( void )
    {
        cout<<"\nConstructorul implicit al clasei FIGURA!"<<endl;
    }
    Figura( Figura& )
    {
        cout<<"\nConstructorul de copiere al clasei FIGURA!"<<endl;
    }
    virtual ~Figura( )
    {
        cout<<"\nDestructorul clasei FIGURA!"<<endl;
    }
public: // zona de interfata
    virtual double Perimetrul( void ) = 0;
    virtual double Aria( void ) = 0;
    virtual double Volumul( void ) = 0;
};
/* #endif // _FIGURA.H_
*
* ----- Fisierul P U N C T . H -----
* #ifndef _PUNCT.H_
* #define _PUNCT.H_
* #include "Figura.h"
*/
class Punct: public Figura
{
protected: // Datele private pentru aceasta clasa
    Coordonate P;

public: // Declaratii cu definitiile corespunzatoare INLINE
    Punct( void )
    {
        cout<<"\nConstructorul implicit al clasei PUNCT!"<<endl;
        P.X = 0;
        P.Y = 0;
    }
    Punct( int _X, int _Y )
    {
        cout<<"\nConstructorul ... clasei PUNCT!"<<endl;
        P.X = _X;
        P.Y = _Y;
    }
    Punct( Punct& x)
    {
        cout<<"\nConstructorul de copiere al clasei PUNCT!"<<endl;
        P = x.P;
    }
    ~Punct( )
```



```
{
    cout<<"\nDestructorul clasei PUNCT!"<<endl;
}
double Perimetrul( void )
{
    return 0;
}
double Aria( void )
{
    return 0;
}
double Volumul( void )
{
    return 0;
}
};
/*
 * #endif // _PUNCT.H_
 *
 * ----- Fisierul S E G M E N T . H -----
 * #ifndef _SEGMENT.H_
 * #define _SEGMENT.H_
 * #include "Figura.h"
 * #include <math.h>
 */
class Segment: public Figura
{
protected: // Datele private pentru aceasta clasa
    Coordonate P1, P2;

public: // Declaratii cu definitiile corespunzatoare INLINE
    Segment( void )
    {
        cout<<"\nConstructorul implicit al clasei SEGMENT!"<<endl;
        P1.X = 0;
        P1.Y = 0;
        P2 = P1;
    }
    Segment( int _X1, int _Y1, int _X2, int _Y2 )
    {
        cout<<"\nun constructor oarecare al clasei SEGMENT!"<<endl;
        P1.X = _X1;
        P1.Y = _Y1;
        P2.X = _X2;
        P2.Y = _Y2;
    }
    Segment( Segment& x)
    {
        cout<<"\nConstructorul de copiere al clasei SEGMENT!"<<endl;
        P1 = x.P1;
        P2 = x.P2;
    }
    ~Segment( )
}
```




```
{
    cout<<"\nDestructorul clasei SEGMENT!"<<endl;
}
double Perimetrul( void )
{
    return sqrt( pow( P1.X - P2.X, 2 ) + pow( P1.Y - P2.Y, 2 ) );
}
double Aria( void )
{
    return 0;
}
double Volumul( void )
{
    return 0;
}
};
/*#endif // _SEGMENT.H_
*
* ----- Fisierul T R I U N G H I . H -----
* #ifndef _TRIUNGHI.H_
* #define _TRIUNGHI.H_
* #include <math.h>
* #include "Figura.h"
*/
class Triunghi: public Figura
{
protected: // Datele private pentru aceasta clasa
    Coordonate P1, P2, P3;

public: // Declaratii cu definitiile corespunzatoare INLINE
    Triunghi( void )
    {
        cout<<"\nConstructorul implicit al clasei TRIUNGHI!"<<endl;
        P1.X = 0;
        P1.Y = 0;
        P2 = P3 = P1;
    }
    Triunghi( int _X1, int _Y1, int _X2, int _Y2 , int _X3, int _Y3 )
    {
        cout<<"\nun constructor oarecare al clasei TRIUNGHI!"<<endl;
        P1.X = _X1;
        P2.X = _X2;
        P3.X = _X3;
        P1.Y = _Y1;
        P2.Y = _Y2;
        P3.Y = _Y3;
    }
    Triunghi( Triunghi& x)
    {
        cout<<"\nConstructorul de copiere al clasei TRIUNGHI!"<<endl;
        P1 = x.P1;
        P2 = x.P2;
```



```
        P3 = x.P3;
    }
    ~Triunghi( )
    {
        cout<<"\nDestructorul clasei TRIUNGHI!"<<endl;
    }
    double Perimetrul( void )
    {
        return sqrt( pow( P1.X - P2.X, 2 ) + pow( P1.Y - P2.Y, 2 ) ) +
               sqrt( pow( P2.X - P3.X, 2 ) + pow( P2.Y - P3.Y, 2 ) ) +
               sqrt( pow( P3.X - P1.X, 2 ) + pow( P3.Y - P1.Y, 2 ) );
    }
    double Aria( void )
    {
        double p = Perimetrul() / 2;
        double a = sqrt( pow( P1.X-P2.X, 2 ) + pow( P1.Y-P2.Y, 2 ) ),
               b = sqrt(pow(P2.X-P3.X, 2) + pow( P2.Y-P3.Y, 2)),
               c = sqrt(pow(P3.X-P1.X, 2) + pow(P3.Y-P1.Y, 2));
        return sqrt( p * (p - a) * (p - b) * (p - c) );
    }
    double Volumul( void )
    {
        return 0;
    }
};
/*#endif // _TRIUNGHI.H_
*
* ----- Fisierul M A I N . H -----
* #include<conio.h>
* #include<iostream.h>
* #include<fstream.h>
* #include"Punct.h"
* #include"Segment.h"
* #include"Triunghi.h"
*/
int main ( void )
{
    cout<<"[START]"<<endl;
    Punct p(1, 2);
    Segment s(0, 0, 1, 1);
    Triunghi t( 0, 0, 1, 1, 2, 0 );
    /* Cazul utilizarii pointerilor */
    cout<<"\n[POINTERI]-----"<<endl;
    Figura *f;
    f = &p;
    cout<<"Perimetrul este: "<<f->Perimetrul()<<endl;
    cout<<"Aria este: "<<f->Aria()<<endl;
    f = &s;
    cout<<"Perimetrul este: "<<f->Perimetrul()<<endl;
    cout<<"Aria este: "<<f->Aria()<<endl;
    f = &t;
    cout<<"Perimetrul este: "<<f->Perimetrul()<<endl;
    cout<<"Aria este: "<<f->Aria()<<endl;
```



```
/* Cazul utilizarii referintelor */
cout<<"\n[REFERINTE]-----"<<endl;
Figura &f1 = p;
cout<<"Perimetrul este: "<<f1.Perimetrul()<<endl;
cout<<"Aria este: "<<f1.Aria()<<endl;
Figura &f2 = s;
cout<<"Perimetrul este: "<<f2.Perimetrul()<<endl;
cout<<"Aria este: "<<f2.Aria()<<endl;
Figura &f3 = t;
cout<<"Perimetrul este: "<<f3.Perimetrul()<<endl;
cout<<"Aria este: "<<f3.Aria()<<endl;
f3 = s;
cout<<"Perimetrul este: "<<f3.Perimetrul()<<endl;
cout<<"Aria este: "<<f3.Aria()<<endl;
cout<<"[END]\n"<<endl;

return 0;
}/* END. main( void ) */
```

OUTPUT

```
[START]
Constructorul implicit al clasei FIGURA!
Constructorul ... clasei PUNCT!
Constructorul implicit al clasei FIGURA!
un constructor oarecare al clasei SEGMENT!
Constructorul implicit al clasei FIGURA!
un constructor oarecare al clasei TRIUNGHI!

[POINTERI]-----
Perimetrul este: 0
Aria este: 0
Perimetrul este: 1.41421
Aria este: 0
Perimetrul este: 4.82843
Aria este: 1

[REFERINTE]-----
Perimetrul este: 0
Aria este: 0
Perimetrul este: 1.41421
Aria este: 0
Perimetrul este: 4.82843
Aria este: 1
Perimetrul este: 4.82843
Aria este: 1
[END]

Destructorul clasei TRIUNGHI!
Destructorul clasei FIGURA!
Destructorul clasei SEGMENT!
Destructorul clasei FIGURA!
Destructorul clasei PUNCT!
Destructorul clasei FIGURA!
```



CONCLUZII

- Metodele pur virtuale sunt redefinite în cadrul claselor specializate (*Punct*, *Segment* și respectiv *Triunghi*) pentru a putea obține un comportament polimorf la nivelul acestor clase.
- Acest comportament polimorf este pus în evidență în cadrul funcției `main()`, prin manipularea pointerilor și al referințelor la clasa de bază *Figură*..

4 TEME

1. Studiul laboratorului și al programului prezentat;
2. Verificarea comportamentului polimorfic, al celor trei clase (figuri geometrice) în cazul celor trei funcții, ca fiind virtuale, respectiv în cazul în care funcțiile nu sunt virtuale.
3. Explicați de ce după atribuirea: `f3 = s,` programul va afișa aceleași informații (*referitoare la arie și perimetru*), neținând seama de ultima atribuire. Referința se comportă ca și cum ar fi un *Triunghi* și nu un *Segment*.
4. Creați un tip abstract de dată (TDA), de tipul *Lista*, *Stiva*, *Coadă*, *Multime*, ce manipulează date de tip heterogene (*Puncte*, *Segmente*, *Triunghiuri*) și nu de tip omogen după cum le-am cunoscut până acum.

PROBELMA 2. SOLUTIE

Formulare

Explicați de ce după atribuirea `f3 = s`, programul va afișa aceleași informații (referitoare la arie și perimetru), neținând seama de ultima atribuire. Se constata ca "referința" se comportă ca și cum ar fi un "Triunghi" și nu un "Segment".

Explicatie

Variabila f3 este o referinta si a fost initializata cu un obiect de tip Triunghi, in atribuirea urmatoare din functia main.

```
/* ... */
Triunghi t( 0, 0, 1, 1, 2, 0 );
/*
 *
 */
Figura &f3 = t;
cout<<"Perimetrul este: "<<f3.Perimetrul()<<endl;
cout<<"Aria este: "<<f3.Aria()<<endl;
/* ... */
```

In momentul in care facem o atribuire cu acesta referinta, ca in exemplul de mai jos, are loc o copiere "bit cu bit" doar la nivelul "sub-obiectelor" din calsa de baza, clasa comun ambelor obiecte (f3 si s).

```
/* ... */
f3 = s;
cout<<"Perimetrul este: "<<f3.Perimetrul()<<endl;
cout<<"Aria este: "<<f3.Aria()<<endl;
cout<<"[END]\n"<<endl;
/* ... */
```

In continuare, referinta f3 indica spre un "Triunghi", doar ca in acest moment continutul "sub-obiectului", din clasa de baza (*Figura*), din cadrul lui f3 a fost schimbat cu continutul, aceluiasi 'sub-obiect', din obiectul s.

Exemplificare

Pentru a intelege acest fenomen ne vom folosi aplicatia, "raspuns" la problema 3, iar modificand functia main, dupa cum se vede mai jos, vom obtine un raspuns conform explicatiei date mai sus.

main(...)

```
int main( void )
{
```



```
cout << "[START] Teste\n" << endl;

cout << "\n(Dreptunghi) ...." << endl;
Obiect *p1 = new Dreptunghi( "dr_I", 2, 4 );
p1->Print();
cout<<"Aria: "<<p1->Aria()<<endl;

cout << "\n(Triunghi) ...." << endl;
Obiect *p2 = new Triunghi( "tr_I", 1, 1, 1 );
p2->Print();
cout<<"Aria: "<<p2->Aria()<<endl;

Obiect &f2 = *p1;
cout<<"\nREFERINTA_1: "<<f2.Aria()<<endl;
f2.Print();

f2 = *p2;
cout<<"\nREFERINTA_2: "<<f2.Aria()<<endl;
f2.Print();
delete p1;
delete p2;

/* ... */
}
```

OUTPUT

```
[START] Teste

(Dreptunghi) ....
dr_I< 2, 4 > Aria: 8

(Triunghi) ....
tr_I< 1, 1, 1 > Aria: 0.1875

REFERINTA_1: 8
dr_I< 2, 4 >
REFERINTA_2: 8
tr_I< 2, 4 >
return 0 (0x0) exe
```

Explicatie

Atentie la ce se afiseaza in ultimele 4 mesaje, corespunzatoare utilizarii fererintei f2!

Inainte de atribuirea `f2 = s`, in `f2` aveam, in variabila nume - „dr_I”, iar dupa atribuirea avem, in aceeasi variabila, „tr_I”. In rest referinta indica spre un „Triunghi”.

Concluzie:

A avut loc copierea, in cadrul atribuirii, doar la nivelul „sub-obiectului” din clasa de baza, pentru cele doua entitati intrate in relatie, `f2` si `s`.

Laboratorul 12

1. Scop

Este prezentat mecanismul ce stau la baza parametrizării notiunii de clasă în C++. Se va introduce notiunile de funcție parametrizată (template) cât și de clasă parametrizată (template).

Mecanismul pune la dispoziție metode prin intermediul cărora putem defini clase cât și funcții identice ca și concepție diferind între ele doar prin intermediul unor tipuri de date.

2. Bazele teoretice

2.1. Mecanismul TEMPLATE

O entitate este o entitate template dacă ea descrie un șablon pentru un număr nespecificat de entități de cod concrete înrudite între ele. Așadar vom avea atât funcții template cât și clase template. Semnalarea unei funcții sau clase template se face prin cuvântul cheie template prezent imediat înaintea entității respective. Imediat după cuvântul cheie template urmează lista argumentelor template, o listă de tipuri specificate în mod generic și care urmează a fi substituite prin tipuri concrete la instanțierea entității template.

2.2. Funcții template

Modul în care compilatorul își alege funcția de aplicat în momentul întâlnirii unui apel de funcție e dat de parcurgerea secvențială a următorilor pași:

1. Dacă există o funcție cu aceeași semnătură generează cod de apel. Altfel,
2. Dacă există o funcție template cu același nume generează cod pentru funcția de tip template (dacă e primul apel de acest gen) și cod de apel a funcției. Altfel,
3. Încearcă aplicarea conversiilor implicite de tip pentru funcțiile cu același nume. Dacă găsește o singură funcție generează cod pentru conversiile necesare și apelul funcției. Altfel,
4. Semnalează eroare.

Specificarea tipurilor în lista argumentelor template odată cu declararea sau definirea unei funcții template respectă constrângerile prezentate în continuare. Tipurile din lista argumentelor template pot fi doar tipuri generice de forma `<class T1, class T2, . . . , class Tn>`, unde dacă T_i este numele unui tip utilizator atunci nu există nici o legătură între tipul utilizator și tipul generic cu același nume din lista argumentelor template. Așadar nu se acceptă tipuri fundamentale, tipuri utilizator sau constante. Prezența unui tip generic în lista argumentelor template ale unei funcții determină necesitatea apariției tipului respectiv în cadrul listei parametrilor formali ai funcției. În fine, orice situație conflictuală detectată de compilator în momentul generării de cod va fi semnalată ca atare. Aceste observații sunt surprinse de codul prezentat în continuare:

```
class CClasa { };
```



```
// eroare, lista argumentelor template specifică tipuri invalide
template< char > void FunctieTemplatel( char );
template< class CClasa, 10 > void FunctieTemplate2( CClasa )
    // eroare, tipurile "T" și "T1" nu apar în lista
    // parametrilor formali a declarațiilor de funcții template
template< class T > void FunctieTemplate3( );
template< class T1, class T2 > T1& FunctieTemplate4( );
template< class T > void FunctieTemplate5( T, T );
    // eroare, tipul generic "T" nu este instanțiat univoc
FunctieTemplate5(10, 5.5);
```

2.3. Clase template

Asemănător funcțiilor template se pot defini și clase template având aceleași caracteristici funcționale, furnizarea unor șabloane pentru un set de clase înrudite. Majoritatea compilatoarelor pun la dispoziția utilizatorului o bibliotecă de clase template, așa numitele clase container care abstractizează principalele tipuri de date abstracte. O clasă template poate fi reprezentată simplificat în forma:

```
template < lista_argumentelor_template > declarație_clasă
```

unde *declarație_clasă* este declarația unui tip utilizator care uzează tipurile generice specificate în *lista_argumentelor_template*. Instanțierea unei clase template, în vederea generării de cod pentru clasele de tip template, comportă menționarea tipurilor concrete și a numelui clasei template:

```
nume_clasă < lista_argumente_concrete > nume_obiect;
```

Ca o facilitate în plus față de funcțiile template este posibilă specificarea în lista de argumente a unei clase template și a altor tipuri decât cele definite de utilizator, adică a tipurilor fundamentale. Acest amănunt este exploatat uneori la inițializarea unor tipuri fundamentale din cadrul clasei altfel decât prin intermediul parametrilor constructorului:

```
template< int dim, class T>
class CClasaTemplate
{
    public :
        CClasaTemplate( const T& t ) : _t( t ), _dim( dim )    { }
        // . . .
    private :
        T _t;
        int _dim;
};
    // . . . instantierea clasei
CClasaTemplate< 10, void* > ob;
```

2.4. Template și friend

Funcțiile friend ale unei clase template nefiind funcții membru nu *moștenesc* argumentele template. Ele își vor păstra statutul de funcții normale.

2.5. Template și static



Fiecare clasă de tip template generată dintr-o clasă template are propriile sale instanțieri ale membrilor statici. La definirea membrilor statici pot fi specificate tipurile concrete de instanțiere și/sau tipurile generice. Pentru o clasă de tip template dacă se specifică tipuri concrete la definirea membrilor statici atunci se vor folosi definițiile respective. Dacă nu există astfel de definiții se va încerca generarea de cod din definițiile generice ale membrilor statici (dacă există). Dacă din anumite motive nu poate fi generat cod se va semnaliza eroare.

```
template < class T>
class Test
{
    private:
        static int el;
        // . . .
};
template <class T>
int Test<T>:el = 10;    // definirea membrului static sub forma unui
sablon
```

2.6. Template și moștenirea

În procesul de integrare a claselor template în cadrul taxonomiilor de clase distingem următoarele situații distincte:

1. O clasă template derivată dintr-o clasă normală. Situația apare când se dorește dotarea unei clase cu facilități de parametrizare.
2. O clasă normală derivată dintr-o clasă de tip template. Cea mai frecvent întâlnită situație, apare odată cu specializarea parametrică a claselor frunză dintr-o ierarhie template.
3. O clasă template derivată dintr-o altă clasă template. Este situația cea mai banală și este echivalentul derivării claselor normale.
4. O clasă template derivată dintr-o clasă de tip template. Cea mai puțin întâlnită, este o generalizare a situației de la punctul 1.

2.7. Template și polimorfismul

La generarea de cod template nu este permisă aplicarea polimorfismului de moștenire. Deși situații similare codului de mai jos nu sunt acceptate de către standardul ANSI C++.

2.8. Implementarea listelor în C++

2.8.1. Modele de abstractizare a listelor

Pentru prezentarea principalelor modele de abstractizare a listelor vom lua ca exemplu lista simplu înălțuită. Extinderea exemplelor în vederea abstractizării unor alte tipuri de liste este trivială și constă în adăugarea unor noi membri sau modificarea membrilor existenți. Prin clasa *CInfo* vom desemna tipul elementelor listei în discuție iar operațiile fundamentale de manipulare a listelor vom presupune că sunt cele clasice și nu vom insista asupra lor decât la modul general.

2.8.2. Modelul A (recursiv simplu)

Primul model de abstractizare este cel care modelează cel mai fidel definirea recursivă a listelor:



O *listă* este formată dintr-un cap de listă, adică un element și o coadă de listă, adică o *listă*.

Declarația clasei *CLista* este prezentată în continuare:

```
// abstractizarea listei conform definiției
class CLista
{
    public :
        CLista(const CInfo& info) : _info(info), _pCoadăListei(NULL){ }
        ~CLista( );
        // operații fundamentale ...
    private :
        // capul listei
        CInfo _info;
        // coada listei
        CLista* _pCoadăListei
};
```

Acest model este cel mai puțin utilizat datorită operațiilor de manipulare a listei care din cauza organizării listei sunt implementate într-o manieră recursivă. Tot din cauza recursivității operațiilor fundamentale, modelul nu se pretează pentru toate tipurile de liste cum ar fi listele dublu înlanțuite sau alte tipuri de liste. Lista este memorată prin intermediul capului listei care poate fi o entitate de tip global, în orice caz externă declarației listei.

2.8.3. Modelul B (recursiv client – server)

Al doilea model de abstractizare a listelor separă noțiunile *listă* și *nod de listă* prin declararea a două tipuri diferite între care există relații de tip **client - server**. Interfața este definită la nivelul clasei care abstractizează lista ca un întreg iar caracterul recursiv al listei este descris la nivelul clasei care abstractizează nodul listei.

Deși lista este descrisă recursiv - de altfel toate modelele adoptă o astfel de strategie ca fiind cea mai naturală - spre deosebire de modelul prezentat anterior, implementarea operațiilor de manipulare a listei **nu** sunt recursive ci iterative. și în acest caz lista este memorată printr-o entitate externă declarațiilor clasei nodului de listă și a listei.

Acest model cunoaște două variante deoarece există două moduri diferite de abordare a relației și implicit a serviciilor existente între clasa care abstractizează nodul listei și cea care abstractizează lista ca atare.

2.8.4. Modelul B1

Primul presupune o relație obișnuită de tipul **client-server** consumată prin intermediul interfeței clasei nodului de listă, *CNodLista*, implementată ca și clasă de sine stătătoare. Situația este ilustrată în continuare:

```
// descrierea clasei nodului de listă
class CNodLista
{
    public :
        CNodLista( const CInfo&, CNodLista* );
        // observați prezența tipurilor referință
        // ca tip de retur al metodelor de interfață
        CInfo& Info( ) const;
        CNodLista*& Urm( ) const;

    private:
```



```
CInfo      _info;
CNodeLista* _pUrm;
};

// abstractizarea listei ca întreg
class CLista
{
public:
    CLista( ) : _pCap( NULL ){ }
    ~CLista( )
        // alte operații fundamentale . . .
private:
    CNodeLista* _pCap;
};
```

Principalul inconvenient introdus de această variantă e posibilitatea exploatării clasei *CNodeLista* de către oricare altă clasă **client**. Ori clasa *CNodeLista* este doar o clasă de implementare introdusă în scopul descrierii caracterului recursiv al unei liste deci singurul beneficiar al serviciilor oferite de aceasta ar fi clasa *CLista*.

2.8.5. Modelul B2

A doua variantă a acestui model elimină inconvenientul semnalat anterior dar recurge la utilizarea declarațiilor de tip **friend** tocmai pentru a asigura unicitatea tipului de **client** al nodului de listă:

```
// descrierea clasei nodului de listă
class CNodeLista
{
private:
    // constructorul nu promovează o interfață
    // ci este definit doar pentru a ușura inițializarea
    CNodeLista( CInfo&, CNodeLista* );

    // implementare
    CInfo      _info;
    CNodeLista* _pUrm;
    // unicul tip de client
    friend class CLista;
};
// declarația clasei CLista se păstrează
```

Acest model, într-o variantă sau cealaltă, este cel mai des utilizat în practică.

2.8.6. Modelul B3

Al treilea și ultimul model se aseamănă întrucâtva cu primul model prin abstractizarea listei cu ajutorul unei singure clase și cu al doilea model prin implementarea operațiilor fundamentale într-o formă iterativă. Lista este memorată de data aceasta prin intermediul unei date membru statice:

```
class CLista
{
public:
    CLista(CInfo& info, CLista* pUrm) : _info(info), _pUrm(pUrm) { }
    ~CLista( )
        // operații fundamentale
private:
    static CLista* _pCap;
```



```
CInfo      _info;
CLista*    _pUrm;
static CLista* pCap;
};
CLista* CLista::pCap = NULL;
```

2.8.7. Liste generale

Cuvîntul cheie **void** este folosit la declararea funcțiilor pentru specificarea absenței parametrilor formali sau a tipului de retur și ca tip derivat sub forma *void**. Tipul *void** este considerat supertip al oricărui tip de forma *TIP**, unde *TIP* este un tip oarecare, iar pentru a fi acceptată o secvență de genul:

```
int i;      void* p = &i;
```

va uza de trăsăturile de polimorfism ale limbajului. Acesta este de altfel singura facilitare predefinită a polimorfismului de moștenire pe care limbajul C++ o moștenește din C. Deși acceptată în ANSI C, conversia inversă din *void** în *TIP** este interzisă în C++ în spiritul respectării regulilor de transformare între supertipuri și subtipuri. De fapt conversia poate fi realizată în mod explicit:

```
void Functie( void* p )
{
    int* pInt = ( int* )p;
}
```

dar acesta nu e modul cel mai favorabil de a trata lucrurile. Datorită proprietăților sale de supertip, tipul *void** este deseori utilizat ca tip generic în colecțiile eterogene de obiecte. Declarațiile claselor care implementează o coadă de așteptare (structură de date de tip **LIFO - Last In Last Out**) cu ajutorul unei liste dublu înlănțuite formată din elemente de tipul *void** sunt prezentate în continuare :

```
// abstractizarea nodului de coadă
class CNodGeneral
{
private :
    CNodGeneral( CNodGeneral* pUrm, CNodGeneral* pAnt, void*
pInfo )
        :_pUrm( pUrm ), _pAnt( pAnt ), _pInfo( pInfo ) { }

    NodGeneral* _pUrm;
    CNodGeneral* _pAnt;
    void* _pInfo;
    // clientul nodului de coadă
    friend class CCoadGeneral;
};

// abstractizarea cozii
class CCoadGeneral
{
public:
    CCoadGeneral( ): _pCap( NULL ), _pCoad( NULL ) { }
    ~CCoadGeneral( );
    // operații fundamentale
    void InCoad( void* pElem );
    void* DinCoad( );
```



```
        // forțează și testează coada vidă
void CoadăVida( );
int ECoadăVida( ) const;

private :
    CNodGeneral* _pCap;
    CNodGeneral* _pCoadă;
};
```

O astfel de listă o vom numi **listă generală** datorită caracterului general al informației conținute. În forma de mai sus, clasa *CCoadăGeneral* poate stoca orice adresă de obiect dar apar complicații la extragerea adreselor din coada de așteptare. Dacă presupunem că toate adresele din coadă sunt adresele unor obiecte de același tip (*CClasă* în cazul de față), efortul de extragere se rezumă la o conversie explicită:

```
CCoadăGeneral coada;
CClasă* pOb = ( CClasă* )coada.DinCoadă( );
```

dar se poate întâmpla ca conversia aplicată să nu fie și cea corectă. Pentru colecții omogene de obiecte este de preferat utilizarea mecanismului **template** așa cum se va vedea în continuare.

Și mai gravă e situația unei cozi de așteptare eterogene, cu adresele unor obiecte de tipuri diferite între care nu există relații de tipul IS_A. Atunci e nevoie de conversie selectivă în funcție de tipul obiectului extras. Apare așadar inevitabil necesitatea memorării în coadă a unor variabile auxiliare de selecție a conversiei pentru fiecare obiect în parte.

Soluția în astfel de cazuri este integrarea tipurilor într-o ierarhie și crearea unei colecții cu elemente de tipul *CBaza**, unde *CBaza* este superclasa ierarhiei.

2.8.8. Liste generice

Listele generale oferă utilizatorului posibilitatea folosirii acestora în mod eronat datorită inexistenței unui mecanism de verificare a tipurilor. În plus listele generale sunt liste de pointeri, pot fi stocate doar adresele unor obiecte și oferă o formă slabă de polimorfism. Vom schița implementarea cozii de așteptare din paragraful anterior folosind facilitățile introduse de mecanismul **template**.

```
// abstractizarea nodului de coadă
template < class T >
class CNodGeneric
{
public :
    CNodGeneric( CNodGeneric<T>* pUrm, CNodGeneric<T>* pAnt, T& t )
        : _pUrm( pUrm ), _pAnt( pAnt ), _t( t ) { }
    CNodGeneric*& Ant( );
    CNodGeneric*& Urm( );
    T& Info( );

private:
    CNodGeneric* _pUrm;
    CNodGeneric* _pAnt;
    T _t;
};

// abstractizarea cozii
template < class T >
class CCoadăGeneric
{
}
```



```
public:
    CCoadăGeneric( ) : _pCap( NULL ), _pCoadă( NULL ) { }
    ~CCoadăGeneric( );
    // operații fundamentale
    void InCoadă( const T& );
    T DinCoadă( );
    // forțează și testează coada vidă
    void CoadăVida( );
    int ECoadăVida( ) const;

private :
    CNodGeneric< T >* _pCap;
    CNodGeneric< T >* _pCoadă;
};
```

Această listă datorită caracterului generic al tipului folosit pentru memorarea informației o vom numi **listă generică**.

Ea beneficiază de avantajul verificării tipurilor și de generalitatea informației stocate care pot fi pointeri, obiecte sau chiar alte cozi de așteptare.

Singurul dezavantaj major este generarea de cod funcțional identic pentru fiecare instanțiere a tipului generic T cu un tip concret oarecare.

2.8.9. Liste generale cu interfață generică

Uneori în practică se folosește un compromis între cele două posibilități de implementare a listelor, rezultând o a treia variantă care încearcă să cumuleze avantajele ambelor alternative. Aceste liste le vom numi **liste generale cu interfață generică**.

Aceeași coadă de așteptare, implementată ca listă generală cu interfață generică va arăta astfel:

```
// abstractizarea nodului general
class CNodGeneral
{ /* declarația clasei din paragraful Liste generale se păstrează */
};

// abstractizarea cozii generale
class CCoadăGeneral
{ /* declarația clasei din paragraful Liste generale se păstrează
   cu mențiunea că secțiunea publică a clasei este
   transformată în secțiune de tip protected */
};

// clasă generică de interfață
template< class T >
class CCoadă : private CCoadăGeneral
{
public:
    // operații fundamentale
    void InCoadă( T* pT )
    {
        CCoadăGeneral::InCoadă( pT );
    }
    T* DinCoadă( )
    {
        return ( T* )CCoadăGeneral::DinCoadă( );
    }

    // o încercare de redefinire a specificatorilor de
    // accesare a moștenirii ar fi eșuat din cauza
```



```
        // secțiunii de tip protected a clasei de bază
void CCoadaVida( )
{
    CCoadaGeneral::CCoadaVida( );
}
int ECoadaVida( )
{
    return CCoadaGeneral::ECoadaVida( );
}
};
```

Remarcăm schimbarea dreptului de acces la metodele clasei *CCoadaGeneral* din **public** în **protected**, ca urmare clasa *CCoadaGeneral* este o clasă resursă a clasei **template** *CCoada*< *T* >. De aici și moștenirea de tip **private**.

Clasa *CCoada*< *T* > joacă rolul de clasă interfață între clasa *CCoadaGeneral* și apelurile utilizator, prin ea se realizează verificarea tipurilor. Codul generat de către clasa *CCoada* în urma instanțierii sale este practic nul și indiferent de numărul de instanțieri ale tipului generic *T* cu tipuri concrete diferite codul obiect **este unic**. O astfel de *îmbrăcare* a claselor poate fi aplicată în toate situațiile în care clasele *generale* sunt deja definite și nu se dorește abandonarea sau rescrierea lor.

Există și alte modalități de integrare a claselor **template** în cadrul unei taxonomii de clase. De exemplu efectul din secvența anterioară de program putea fi obținut și într-o manieră de genul:

```
template < class T >
class CCoada : private CCoadaGeneric< void* >
{ /* . . . */ };
```

în care modificările cerute de noua situație sunt neglijabile.

3. Temă

- Studiul programului și lămurirea aspectelor prezentate în ultima parte a laboratorului;
- Implementarea TDA-ului Lista simplu înlantuită de tip generică.

Laboratorul 13

1. Scop

Acest capitol al îndrumarului dezbate mecanismul tratării excepțiilor, mecanism introdus în cadrul standardului C++.

Se porneste de la necesitatea introducerii mecanismului și sunt prezentate metodele prin care acest mecanism poate prelua sarcina tratării excepțiilor aparute în programe.

Se face o diferențiere între excepție și eroare.

2. Bazele teoretice

2.1. Tratarea Excepțiilor

Tratarea unei erori constă în detectarea erorii și anunțarea componentei software însărcinată cu rezolvarea acestei situații. Mecanismul excepțiilor înlătură necesitatea de a verifica de fiecare dată starea unui apel de funcție și separă codul de tratare a erorilor de codul “normal”; indiferent de numărul de apeluri ale unei funcții, eventualele erori sunt tratate o singură dată, într-un singur loc.

Mecanismul de tratare a excepțiilor este acea componentă software însărcinată cu transferul controlului execuției programului de la punctul în care este generată o excepție la punctul în care excepția este gestionată. O excepție este o instanță fie a unui tip predefinit, fie a unui tip definit de utilizator;

Pașii care trebuie în general urmați în vederea tratării excepțiilor în cadrul programelor C++ sunt:

1. se identifică acele zone din program în care se efectuează o operație despre care se cunoaște că ar putea genera o excepție și se marchează în cadrul unui bloc de tip *try*. În cadrul acestui bloc, se testează condiția de apariție a excepției și în caz pozitiv se semnalează apariția excepției prin intermediul cuvântului cheie *throw*. Trebuie precizat faptul că excepția nu este generată în mod obligatoriu de către programator, existând posibilitatea ca excepția să fie generată de către o funcție a mediului de dezvoltare, cuvântului cheie *throw* aflându-se, astfel, în definiția respectivei funcții.
2. se realizează blocuri de tip *catch* pentru a capta excepțiile atunci când acestea sunt întâlnite.

Sintaxa *try-catch* este următoarea:

```
try
{
    // cod care poate genera excepții
    throw tipExcepție; //generarea unei excepții
}
catch(tip_excepție &id1)
```




```
{
    //tratarea excepției de tipul tip_except1
}
catch(tip_except2 &id2)
{
    //tratarea excepției de tipul tip_except2
}
...
catch(tip_exceptn &idn)
{
    //tratarea excepției de tipul tip_exceptn
}
catch(...)
{
    //tratarea celorlalte excepții, diferite de except1, except2, ... ,
    exceptn
}
```

După blocul *try* urmează unul sau mai multe blocuri *catch*, în cadrul cărora sunt tratate excepțiile. Dacă în blocul *try* este generată o excepție, mecanismul de tratare a excepției “pleacă în căutarea” primului bloc *catch* care are argumentul de un tip ce se potrivește cu tipul excepției. Dacă un astfel de bloc este găsit, sunt executate toate instrucțiunile din blocul respectiv și excepția se consideră a fi tratată (prinsă). Apoi, programul continuă cu instrucțiunea imediat următoare blocului *try*. Dacă nu se poate găsi un bloc *catch* pentru tratarea excepției, este apelata rutina predefinită, care încheie execuția programului în curs.

tipExcepție nu este altceva decât instanțierea unei clase vide (care determină tipul excepției), putând fi declarat ca:

```
class tipExcepție {};
```

Următorul program, în care se încearcă împărțirea la 0, va genera o excepție, execuția lui încheindu-se odată cu generarea excepției. Execuția programului în mediul de dezvoltare Visual C++ va avea ca efect și afișarea unui mesaj (Fig. 1) cu informații legate de excepția apărută.

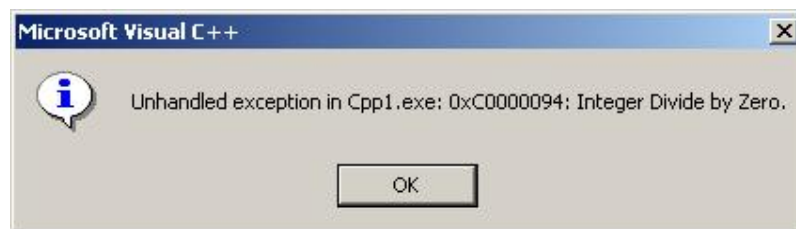


Fig. 1

```
#include <iostream.h>
void main(void)
{
    int a,b;
    a=3;
    b=0;
    cout<<"a/b="<<a/b<<endl;//cod ce genereaza o exceptie
}
```

Programul de mai jos, va genera și el o excepție, dar execuția lui va continua cu instrucțiunea imediat următoare blocului *try*, deoarece excepția a fost tratată.



```
#include <iostream.h>
#include <conio.h>
int divide(int x, int y)
{
    return x / y; //code ce genereaza o exceptie
}
void main(void)
{
    int x, y,z;
    //tratarea exceptiilor
    try
    {
        //secventa de cod c ear putea genera o eroare
        x = 5;
        y = 0;
        z= divide(x, y);
    }
    //interceptarea tuturor exceptiilor
    catch (...)
    {
        cout << "Impartire la zero\n";
    }
    cout<< "executia programului continua ... "<<endl;
    getch();
}
```

În acest ultim exemplu, excepția a fost lansată de către mediul de dezvoltare Visual C++. În exemplul următor, aceeași excepție este lansată de către programator, prin intermediul cuvântului cheie *throw*. Se poate observa că excepția a fost lansată în funcția *divide*, dar a fost tratată în funcția apelantă, *main*.

```
#include <iostream.h>
#include <conio.h>
int divide(int x, int y)
{
    if (y == 0)
        throw int(); //lansarea exceptiei de tipul int
    return x / y;
}
void main(void)
{
    int x, y,z;
    try
    {
        x = 5;
        y = 0;
        z = divide(x, y);
    }
    //tratarea exceptiilor de tipul int
    catch (int)
    {
        cout << "Impartire la zero"<<endl;
    }
    //tratarea celorlalte exceptii
    catch(...)
}
```



```
{  
    cout << "Exceptie necunoscuta"<<endl;  
}  
getch();  
}
```

În următorul exemplu, se prezintă modalitatea de tratare a excepțiilor în cadrul unui program ce utilizează clase. Se prezintă cazul unei clase Punct ce conține metode care pot genera excepții. Ne referim aici la excepții lansate de către programator. În acest scop, au fost definite și două clase vide, corespunzătoare excepțiilor care pot să apară în program.

```
#include <iostream.h>  
#include <conio.h>  
#define MAX_X      80  
#define MAX_Y      25  
class Punct  
{  
private:  
    int x,y;  
public:  
    ///definirea claselor corespunzatoare exceptiilor  
    class CoordZero {};  
    class CoordInAfaraEcranului {};  
  
    Punct(int _x, int _y)  
    {  
        x=_x;  
        y=_y;  
    }  
    int GetX()  
    {  
        return x;  
    }  
    int GetY()  
    {  
        return y;  
    }  
    void SetX(int _x)  
    {  
        if(_x>0)  
            if(_x<=MAX_X)  
                x=_x;  
            else  
                throw CoordInAfaraEcranului(); //lansarea exceptiei  
        else  
            throw CoordZero();//lansarea exceptiei  
    }  
    void SetY(int _y)  
    {  
        if(_y>0)  
            if(_y<=MAX_Y)  
                y=_y;  
            else  
                throw CoordInAfaraEcranului();//lansarea exceptiei  
        else  
            throw CoordZero();//lansarea exceptiei  
    }  
}
```



```
};  
void main()  
{  
    Punct p1(1, 1);  
    try  
    {  
        p1.SetX(5);  
        cout << "p1.x setat la valoarea " << p1.GetX() << "." << endl;  
        p1.SetX(100);  
    }  
    catch(Punct::CoordZero)  
    {  
        cout<<"Coordonata este <= 0 !\n";  
    }  
    catch(Punct::CoordInAfaraEcranului)  
    {  
        cout<<"Coordonata este in afara ecranului!\n";  
    }  
    catch(...)  
    {  
        cout<<"Exceptie necunoscuta!\n";  
    }  
  
    cout<<endl;  
    getch();  
}
```

Dacă la începutul funcției *main* se adăugă următoarea secvență de cod:

```
Punct p2(2, 50);  
p2.SetY(-5);  
cout << "p2.x setat la valoarea " << p2.GetY() << "." << endl;  
p2.SetY(100);
```

atunci execuția programului s-ar încheia fără a se trece la *Punct p1(1, 1)*; deoarece a apărut o excepție care nu a fost tratată.

Datorită faptului că excepția este instanțierea unei clase, prin derivare pot fi realizate adevărate ierarhii de tratare a excepțiilor. Trebuie avută însă în vedere posibilitatea de apariție a unor excepții chiar în cadrul codului de tratare a unei excepții, situații care trebuie evitate.

O astfel de situație este prezentată în exemplul de mai jos. Pentru reprezentarea excepțiilor au fost definite două clase: clasa *CExceptie* care este o clasă generală și clasa *CExceptieAlocare* care este derivată în mod public din *CExceptie*. Clasa vector este o clasă pentru lucrul cu vectori de numere întregi. În metoda *Aloca* a acestei clase, se generează o excepție de tipul *CExceptieAlocare*, în cazul în care numărul de elemente este mai mare decât 100. În bucla *catch* de tratare a excepției este apelată metoda *SeteazaExceptie* a clasei *CExceptieAlocare*. Această metodă va apela metoda *AlocaMesaj* a clasei de bază, unde va avea loc o alocare de memorie dinamică. Dacă parametrul cu care a fost apelată metoda are o valoare foarte mare, se va genera o excepție chiar în codul care ar trebui să trateze excepția. Caz în care, execuția programului se va încheia. În cazul de față, această situație poate fi evitată prin includerea celor două apeluri *SeteazaExceptie* și *Afiseaza*, într-o construcție *try-catch*.



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

//Clasa pentru exceptii generale
class CExceptie
{
protected:
    char* mesaj;
public:
    CExceptie() { mesaj = NULL; }
    ~CExceptie() {if(mesaj) delete mesaj;}
    void AlocaMesaj(int n);
    void CopieMesaj(char *_mesaj);
    void Afiseaza() {cout<<"Exceptie: "<<mesaj;}
};

void CExceptie::AlocaMesaj(int n)
{
    //cod ce poate lansa o exceptie
    mesaj=new char[n];
}

void CExceptie::CopieMesaj(char *_mesaj)
{
    strcpy(mesaj,_mesaj);
}

class CExceptieAlocare:public CExceptie
{
public:
    CExceptieAlocare();
    ~CExceptieAlocare() {};
    void SeteazaExceptie();
};

CExceptieAlocare::CExceptieAlocare()
:CExceptie()
{
}

void CExceptieAlocare::SeteazaExceptie()
{
    //Acest apel va duce la lansarea unei exceptii
    AlocaMesaj(1000000000);
    CopieMesaj("eroare alocare!");
}

class vector
{
private:
    int *v;
public:
    vector() {v=NULL;}
    ~vector() {if(v) delete v;}
    void Aloca(int n);
};

void vector::Aloca(int n)
{
    try
```



```
{
    if(n>100)
        throw CExceptieAlocare();//lansarea exceptiei
    else
        v=new int[n];
}
catch(CExceptieAlocare &e)
{
    e.SeteazaExceptie();
    e.Afiseaza();
}
}

void main(void)
{
    vector objv;
    objv.Aloca(150);
    cout<<"\nSfirsit!"<<endl;
    getch();
}
```

Utilizarea exceptiilor standard

```
/** Lansare exceptii standard C++ de tipul:
**      bad_array_new_length, invalid_argument, length_error, bad_typeid
**/
#include <iostream>
#include <conio.h>
#include <bitset>
#include <vector>
#include <typeinfo>
#include <exception>
#include <stdexcept>

using namespace std;

class Polymorphic
{
    virtual void member() {}
};

int main ()
{
    unsigned char c;

    cout<<"START. Mecanism tratare exceptii"<<endl;
    try
    {
        //Poate lansa exceptie la o dimensiune mare (bad_array_new_length)
        int* myarray= new int[ 100 ]; //1000000000

        // Constructorul bitset lanseaza o exceptie de tipul invalid_argument
        // daca se initializeaza cu un string continand caractere diferite de
        // 0 sau 1
        bitset<5> mybitset1 (string("10010"));
        //bitset<5> mybitset2 (string("012345"));

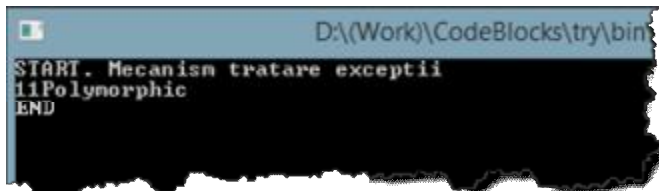
        // Clasa vector lanseaza o exceptie length_error daca > max_size
    }
```



```
std::vector<int> myvector;  
//myvector.resize( myvector.max_size()+1 );  
  
//Poate lansa exceptie la o initializare cu NULL  
Polymorphic * pb = new Polymorphic;  
//Polymorphic * pb = NULL;  
cout<<typeid( *pb ).name()<<endl; // Lanseaza exceptie bad_typeid  
}  
catch (const invalid_argument& ex)  
{  
    cerr << "\tErr.1. Invalid argument: " << ex.what() << endl;  
}  
catch (const length_error& ex)  
{  
    cerr << "\tErr.2. Length error: " << ex.what() << endl;  
}  
catch (exception& ex)  
{  
    cout << "\tErr.3. Standard exception: " << ex.what() << endl;  
}  
  
cout<<"END"<<endl;  
getch();  
return 0;  
}
```

Daca utilizam secventa de alocare a 100 octeti nu vom avea exceptie generata:

```
int* myarray= new int[ 100 ];
```



Daca utilizam secventa de alocare a 1000000000 octeti vom avea o exceptie:

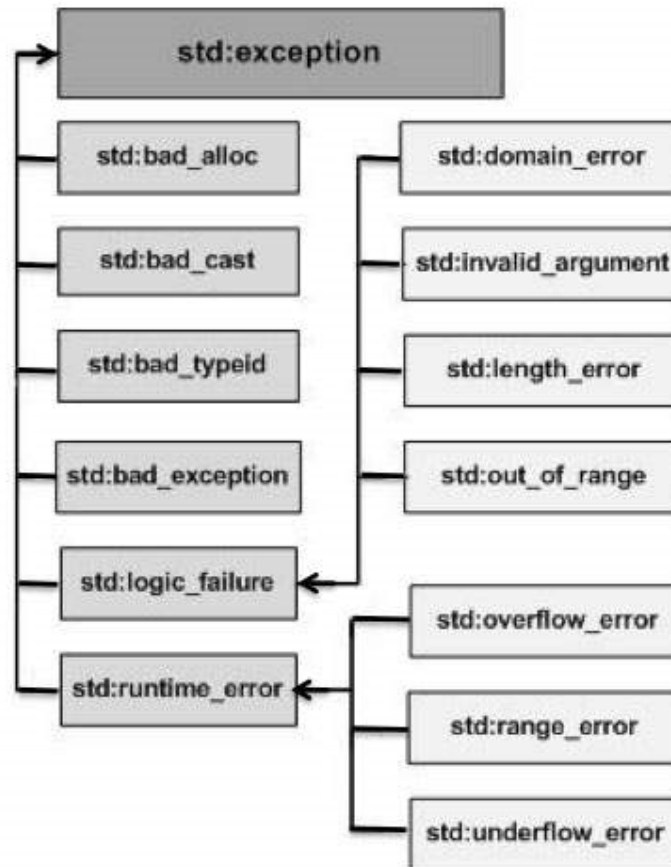
```
int* myarray= new int[ 1000000000 ];
```



In mod asemanator putem experimenta si pentru celelalte categorii de exceptii, comentand sau nu zonele de cod corespunzatoare.



C++ furnizeaza un set de exceptii standard, definite in `<exception>` si `<stdexcept>`. Arhitectura acestor exceptii poate fi observata in continuare:



Descrierea sumara a fiecarui tip de exceptie este data mai jos:

Exceptia	Descrierea
std::exception	O exceptie si o clasa parinte pentru restul exceptiilor standard C++.
std::bad_alloc	Exceptie lansata de catre operatorul new .
std::bad_cast	Exceptie lansata de catre dynamic_cast .
std::bad_exception	Pentru exceptiile neprevazute in programe C++
std::bad_typeid	Exceptie lansata de catre typeid .



std::logic_error	Exceptie lansata ce poate fi interpretata la parcurgerea codului (logica programului).
std::domain_error	Exceptie lansata cand este utilizat incorect un domeniu matematic.
std::invalid_argument	Exceptie lansata in cazul unui argument invalid.
std::length_error	Exceptie lansata cand este creat un std::string prea mare.
std::out_of_range	Exceptie lansata de catre o metoda de tipul std::vector and std::bitset<>::operator[]().
std::runtime_error	Exceptie lansata in la run time.
std::overflow_error	Exceptie lansata la o depasire (overflow) matematica.
std::range_error	Exceptie lansata in momentul in care se stocheaza o valoare ce dapaseste domeniul.
std::underflow_error	Exceptie lansata cand apare o depasire (underflow) matematica.

3. Temă

Studiul programului și lămurirea aspectelor prezentate în ultima parte a laboratorului;