



CLUJ-NAPOCA  
2001

LIMBAJELE  
C ȘI C++  
PENTRU  
ÎNCEPĂTORI  
**VOLUMUL II**

LIMBAJUL  
C  
+

DIACONU ROXANA

Reeditare

Autor:

# LIVIU NEGRESCU

A confruntat cu originalul  
Irina Mitrov

Editura Albastră

Director editură  
Smaranda Derveșteanu

Coordonator serie  
Codruța Poenaru

Coperta  
Liviu Derveșteanu

Tiraj: 1000 exemplare

Tipărit  
EDITURA ALBASTRĂ  
comanda 143 / 2001



## CUPRINS

<b>20. LIMBAJUL C++ CA EXTENSIE A LIMBAJULUI C . . . . .</b>	<b>7</b>
20.1. Comentariu . . . . .	7
20.2. Definiția unei funcții . . . . .	8
20.2.1. Antetul și prototipul unei funcții . . . . .	8
Exerciții . . . . .	11
20.2.2. Corpul unei funcții . . . . .	13
20.3. Tipuri predefinite în C și C++ . . . . .	14
20.4. Constante caracter . . . . .	15
Exerciții . . . . .	15
20.5. Operatori . . . . .	16
20.5.1. Operatorul de rezoluție . . . . .	16
20.5.2. Operatorul adresă (&) . . . . .	17
20.5.3. Operatorul de alocare dinamică a memoriei (new) . . . . .	18
20.5.4. Operatorul de dezalocare a memoriei (delete) . . . . .	20
20.6. Apel prin referință (call by reference) . . . . .	21
Exerciții . . . . .	22
20.7. Funcții care returnează date de tip referință . . . . .	29
Exerciții . . . . .	31
20.8. Modificatori . . . . .	33
20.8.1. Modificatorul const . . . . .	33
20.8.2. Modificatorii near, far și huge . . . . .	36
20.9. Cuvântul cheie void . . . . .	36
20.10. Structuri . . . . .	38
20.11. Reuniune . . . . .	39
20.12. Tipul enumerare . . . . .	40
20.13. Supraîncărcarea funcțiilor . . . . .	41
Exerciții . . . . .	43
20.14. Funcții inline . . . . .	54
Exerciții . . . . .	57
<b>21. PROGRAMAREA PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>60</b>
Exerciții . . . . .	71
<b>22. CLASE . . . . .</b>	<b>75</b>
22.1. Definiția claselor . . . . .	86
22.2. Obiecte . . . . .	88
22.3. Domeniul unui nume . . . . .	91

22.4.	Vizibilitate . . . . .	94	28.	<b>PROGRAMAREA ORIENTATĂ SPRE OBIECTE . . . . .</b>	324																																																																																																																																																																									
22.5.	Durata de viață a datelor . . . . .	94	29.	<b>CLASE DERIVATE ȘI CLASE DE BAZĂ . . . . .</b>	335																																																																																																																																																																									
22.6.	Alocarea și dezalocarea obiectelor . . . . .	95	22.7.	Inițializare . . . . .	96	29.1.	Relația dintre constructorii și destrutorii claselor de bază și ai clasei derivate . . . . .	337	22.8.	Constructor . . . . .	97	22.9.	Destructor . . . . .	105	29.2.	Exerciții . . . . .	342	22.10.	Exerciții . . . . .	106		Funcție prieten (Friend function) . . . . .	137	29.3.	Conversii . . . . .	356		Exerciții . . . . .	140		<b>Redefinirea datelor membru ale unei clase de bază într-o clasă derivată . . . . .</b>	357	<b>23.</b>	<b>SUPRAÎNCĂRCAREA OPERATORILOR . . . . .</b>	<b>143</b>		Supraincărcarea funcțiilor membru ale unei clase de bază într-o clasă derivată . . . . .	358	23.1.	Exerciții . . . . .	149		Exerciții . . . . .	361		Supraincărcarea operatorilor new și delete . . . . .	177	23.2.	Exerciții . . . . .	185		Clase virtuale . . . . .	394		Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>
22.7.	Inițializare . . . . .	96	29.1.	Relația dintre constructorii și destrutorii claselor de bază și ai clasei derivate . . . . .	337																																																																																																																																																																									
22.8.	Constructor . . . . .	97	22.9.	Destructor . . . . .	105	29.2.	Exerciții . . . . .	342	22.10.	Exerciții . . . . .	106		Funcție prieten (Friend function) . . . . .	137	29.3.	Conversii . . . . .	356		Exerciții . . . . .	140		<b>Redefinirea datelor membru ale unei clase de bază într-o clasă derivată . . . . .</b>	357	<b>23.</b>	<b>SUPRAÎNCĂRCAREA OPERATORILOR . . . . .</b>	<b>143</b>		Supraincărcarea funcțiilor membru ale unei clase de bază într-o clasă derivată . . . . .	358	23.1.	Exerciții . . . . .	149		Exerciții . . . . .	361		Supraincărcarea operatorilor new și delete . . . . .	177	23.2.	Exerciții . . . . .	185		Clase virtuale . . . . .	394		Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>									
22.9.	Destructor . . . . .	105	29.2.	Exerciții . . . . .	342																																																																																																																																																																									
22.10.	Exerciții . . . . .	106		Funcție prieten (Friend function) . . . . .	137	29.3.	Conversii . . . . .	356		Exerciții . . . . .	140		<b>Redefinirea datelor membru ale unei clase de bază într-o clasă derivată . . . . .</b>	357	<b>23.</b>	<b>SUPRAÎNCĂRCAREA OPERATORILOR . . . . .</b>	<b>143</b>		Supraincărcarea funcțiilor membru ale unei clase de bază într-o clasă derivată . . . . .	358	23.1.	Exerciții . . . . .	149		Exerciții . . . . .	361		Supraincărcarea operatorilor new și delete . . . . .	177	23.2.	Exerciții . . . . .	185		Clase virtuale . . . . .	394		Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																		
	Funcție prieten (Friend function) . . . . .	137	29.3.	Conversii . . . . .	356																																																																																																																																																																									
	Exerciții . . . . .	140		<b>Redefinirea datelor membru ale unei clase de bază într-o clasă derivată . . . . .</b>	357	<b>23.</b>	<b>SUPRAÎNCĂRCAREA OPERATORILOR . . . . .</b>	<b>143</b>		Supraincărcarea funcțiilor membru ale unei clase de bază într-o clasă derivată . . . . .	358	23.1.	Exerciții . . . . .	149		Exerciții . . . . .	361		Supraincărcarea operatorilor new și delete . . . . .	177	23.2.	Exerciții . . . . .	185		Clase virtuale . . . . .	394		Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																											
	<b>Redefinirea datelor membru ale unei clase de bază într-o clasă derivată . . . . .</b>	357																																																																																																																																																																												
<b>23.</b>	<b>SUPRAÎNCĂRCAREA OPERATORILOR . . . . .</b>	<b>143</b>		Supraincărcarea funcțiilor membru ale unei clase de bază într-o clasă derivată . . . . .	358	23.1.	Exerciții . . . . .	149		Exerciții . . . . .	361		Supraincărcarea operatorilor new și delete . . . . .	177	23.2.	Exerciții . . . . .	185		Clase virtuale . . . . .	394		Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																	
	Supraincărcarea funcțiilor membru ale unei clase de bază într-o clasă derivată . . . . .	358																																																																																																																																																																												
23.1.	Exerciții . . . . .	149		Exerciții . . . . .	361		Supraincărcarea operatorilor new și delete . . . . .	177	23.2.	Exerciții . . . . .	185		Clase virtuale . . . . .	394		Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																							
	Exerciții . . . . .	361																																																																																																																																																																												
	Supraincărcarea operatorilor new și delete . . . . .	177	23.2.	Exerciții . . . . .	185		Clase virtuale . . . . .	394		Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																													
23.2.	Exerciții . . . . .	185		Clase virtuale . . . . .	394		Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																
	Clase virtuale . . . . .	394																																																																																																																																																																												
	Supraincărcarea operatorului = . . . . .	193		Exerciții . . . . .	397	23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																						
	Exerciții . . . . .	397																																																																																																																																																																												
23.3.	Exerciții . . . . .	198	<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>		Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																												
<b>30.</b>	<b>FUNCTII VIRTUALE . . . . .</b>	<b>405</b>																																																																																																																																																																												
	Supraincărcarea operatorului [] (operatorul de indexare) . . . . .	205		Exerciții . . . . .	416		Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																		
	Exerciții . . . . .	416																																																																																																																																																																												
	Exerciții . . . . .	207	30.1.	Clase abstrakte . . . . .	441	23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																								
30.1.	Clase abstrakte . . . . .	441																																																																																																																																																																												
23.4.	Supraincărcarea operatorului () (operatorul de apel funcție) . . . . .	218		Exerciții . . . . .	449		Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																														
	Exerciții . . . . .	449																																																																																																																																																																												
	Exerciții . . . . .	225	<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>	23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																				
<b>31.</b>	<b>INTRĂRI/IESIRI . . . . .</b>	<b>459</b>																																																																																																																																																																												
23.5.	Supraincărcarea operatorului -> . . . . .	232		Intrări/Ieșiri standard . . . . .	460	<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																										
	Intrări/Ieșiri standard . . . . .	460																																																																																																																																																																												
<b>24.</b>	<b>CONVERSII . . . . .</b>	<b>235</b>	31.1.	Ieșire standard . . . . .	461	24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																
31.1.	Ieșire standard . . . . .	461																																																																																																																																																																												
24.1.	Conversia datelor de tipuri predefinite în date de tip abstract . . . . .	240		Exerciții . . . . .	467		Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																						
	Exerciții . . . . .	467																																																																																																																																																																												
	Exerciții . . . . .	246	31.1.1.	Manipulatori . . . . .	468	24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																												
31.1.1.	Manipulatori . . . . .	468																																																																																																																																																																												
24.2.	Conversia dintr-un tip abstract într-un tip predefinit . . . . .	265		Exerciții . . . . .	469		Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																		
	Exerciții . . . . .	469																																																																																																																																																																												
	Exerciții . . . . .	267	31.1.1.2.	Ieșire neformatată . . . . .	470	24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																								
31.1.1.2.	Ieșire neformatată . . . . .	470																																																																																																																																																																												
24.3.	Conversia dintr-un tip abstract într-un alt tip abstract . . . . .	278		31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471		Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																														
	31.1.1.3. Supraincărcarea operatorului << pentru ieșiri de obiecte . . . . .	471																																																																																																																																																																												
	Exerciții . . . . .	282	31.1.2.	Intrare standard . . . . .	472	<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																																				
31.1.2.	Intrare standard . . . . .	472																																																																																																																																																																												
<b>25.</b>	<b>MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU . . . . .</b>	<b>296</b>		Exerciții . . . . .	476		31.1.2.1. Intrări neformatate . . . . .	477	<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																																										
	Exerciții . . . . .	476																																																																																																																																																																												
	31.1.2.1. Intrări neformatate . . . . .	477																																																																																																																																																																												
<b>26.</b>	<b>OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (*. ȘI -&gt;) . . . . .</b>	<b>298</b>		31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480		Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																																																			
	31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte . . . . .	480																																																																																																																																																																												
	Exerciții . . . . .	301		Exerciții . . . . .	481	<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																																																									
	Exerciții . . . . .	481																																																																																																																																																																												
<b>27.</b>	<b>CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR . . . . .</b>	<b>305</b>	31.2.	Formatarea în memorie . . . . .	497		31.3.	Prelucrarea fișierelor . . . . .	499		Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																																																															
31.2.	Formatarea în memorie . . . . .	497																																																																																																																																																																												
	31.3.	Prelucrarea fișierelor . . . . .	499																																																																																																																																																																											
	Exerciții . . . . .	505		<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																																																																									
	<b>BIBLIOGRAFIE . . . . .</b>	<b>518</b>																																																																																																																																																																												

## 20. LIMBAJUL C++ CA EXTENSIE A LIMBAJULUI C

În [14] B. Stroustrup arată că limbajul C++ este proiectat în aşa fel încit:

- să fie un C mai bun;
- să permită stilul de programare prin abstractizarea datelor;
- să permită stilul de programare orientat spre obiecte.

În aceeași carte, Stroustrup afiră că C++ este un “C mai bun” deoarece furnizează un suport mai bun pentru stilurile de programare suportate de limbajul C.

Amintim că limbajul C permite utilizatorului să folosească stilurile de programare incetătenite sub denumirile de “*programare procedurală*” (vezi 4.14) și “*programare modulară*” (vezi capitolul 7). Ambele stiluri sunt suportate mai bine de limbajul C++ pe baza introducerii unor extensii la facilitățile existente în limbajul C.

Limbajul C devine un *subset* al limbajului C++ (adesea se obișnuiește să se spună că C++ este un *superset* al limbajului C). Aceasta înseamnă că un program scris în C este în același timp și un program scris în C++.

În linii mari, acest lucru este adevărat. Compatibilitatea nu este asigurată 100% dar cazurile de incompatibilitate sunt neesențiale și pot fi ușor eliminate la programe concrete.

În capitolul de față se trec în revistă extensiile mai importante ale construcțiilor din C introduse în limbajul C++. Cu alte cuvinte, în acest capitol se discută aspectele care conduce la afirmația lui B. Stroustrup că limbajul C++ este un *C mai bun*.

Celelalte facilități oferite de limbajul C++ (programarea prin abstractizarea datelor și orientată spre obiecte) vor fi abordate în alte capitole.

În încheiere, plastic vorbind, putem afirma că limbajul C++ este un “C incrementat” și înțelegem prin aceasta că pe de o parte este un “C mai bun”, fiind o extensie a limbajului C, iar pe de altă parte permite stiluri de programare impracticabile în C:

- programarea prin abstractizarea datelor și
- programarea orientată spre obiecte.

### 20.1. Comentariu

Comentariile sunt explicații pentru programatori. Ele pot fi introduse în orice punct în care este legală apariția unui caracter alb.

În limbajul C, și deci și în C++, comentariile se includ între /\* (inceputul

comentariului) și \*/ (sfîrșitul comentariului). Comentariile pot fi scrise pe mai multe rînduri.

În afara de aceasta posibilitate, în limbajul C++ există și varianta de a scrie comentarii pe un singur rînd. Un astfel de comentariu începe cu secvența:

//

și se termină la sfîrșitul rîndului.

Comentariile de acest fel însoțesc, de obicei, cîte o instrucție sau o componentă a unei instrucții care necesită explicații pentru programatorii.

Comentariile din antetul unei funcții se scriu adesea pe mai multe linii și din această cauză ele se inserează mai simplu folosind convenția din limbajul C.

**Exemplu:**

```
...  
if(b && zi < 29) // an bisect -> b=1; altfel b=0  
    zi++;  
...
```

În compunerea unui comentariu se pot utiliza orice caractere ale codului ASCII.

Comentariile sunt eliminate din programe la compilare.

## 20.2. Definiția unei funcții

Definiția unei funcții în limbajul C++ se compune din *antet* urmat de *corpu*l ei.

O funcție poate fi apelată dacă este precedată de definiția sau de prototipul ei.

### 20.2.1. Antetul și prototipul unei funcții

O funcție în mod implicit este globală. Ea poate fi localizată într-un fișier sursă folosind cuvîntul cheie *static* la începutul antetului ei. În acest caz, se spune că funcția este *statică*.

O funcție poate să returneze (întoarcă) sau nu o valoare la revenirea din ea.

Tipul valorii returnate de funcție se indică în antetul funcției. În cazul în care funcția nu returnează (întoarce) o valoare, în locul tipului din antetul ei se folosește cuvîntul cheie *void*.

Antetul unei funcții în limbajele C și C++ are formatul:

*tip nume(lista declarațiilor parametrilor formali)*

sau

**static** *tip nume (lista declarațiilor parametrilor formali)*

Declarațiile de parametri sunt separate prin virgulă. Dacă funcția nu are parametri, atunci lista din parantezele rotunde este vida.

Spre deosebire de limbajul C, în limbajul C++ se pot face inițializări ale parametrilor formali.

**Exemplu:**

1. Boolean v\_calend (int zi,int luna,int an = 1994)

În acest exemplu, parametrul formal *an* este inițializat cu valoarea 1994.

2. double aria(double r,double a=0,double b=360,  
double pi=3.14159265358979)

Parametrii formali inițializați se numesc *parametrii impliciti*.

La apelul funcțiilor, unui parametru implicit poate să-i corespundă sau nu un parametru efectiv. În cazul în care unui parametru implicit nu-i corespunde, la apel un parametru efectiv, parametrul formal respectiv ia ca valoare valoarea sa inițială. Aceasta se mai numește *valoare implicită* a parametrului respectiv.

Dacă la un apel, unui parametru implicit ii corespunde un parametru efectiv, atunci parametrului formal respectiv i se atribuie în mod obișnuit valoarea parametrului efectiv care ii corespunde. Cu alte cuvinte, în acest caz se neglijă valoarea implicită a parametrului formal.

De exemplu, apelul funcției *v\_calend* având antetul de mai sus:

i = v\_calend(28, 2);

implică lansarea funcției *v\_calend* pentru următoarele valori ale parametrilor formali:

zi = 28, luna = 2, an = 1994.

Pentru *an* s-a utilizat valoarea implicită. În schimb apelul:

i = v\_calend(31, 3, 1995);

lansează funcția *v\_calend* cu valorile:

zi = 31, luna = 3, an = 1995.

În acest caz, valoarea implicită a fost neglijată.

La utilizarea parametrilor impliciti există o limitare în legătură cu poziția acestora și anume, ei trebuie să ocupe ultimele pozitii în lista declarărilor din antetul funcției, neputind fi amestecati cu parametri care nu sunt inițializați.

În limbajele C și C++ se pot defini funcții cu un număr variabil de parametri. În acest caz, lista declarărilor parametrilor formali conține numai declarăriile parametrilor care sunt prezente la orice apel al funcției. Aceștia sunt așa numiți *parametri fixi* ai funcției spre deosebire de ceilalți care se numesc *parametri variabili*. Parametrii fixi preced pe cei variabili. Prezența parametrilor variabili se indică, în antetul funcției, prin trei puncte care se scriu după ultimul parametru fix al funcției.

## Exemplu:

Fie antetul:

```
void vf(int n,double x, ...)
```

Din antet rezultă că funcția are doi parametri fieși: *n* și *x*, precum și alți parametri al căror număr și tip nu este în prealabil precizat și diferă de la apel la apel.

Funcțiile cu un număr variabil de parametri se definesc folosind niște macrouri speciale care permit accesul la parametri variabili. Aceste macrouri sunt definite în fișierul *stdarg.h*.

De obicei, funcțiile cu un număr variabil de parametri sunt funcții de bibliotecă. Exemple de astfel de funcții din biblioteca sistemului sunt funcțiile *printf* și *scanf*. Acestea au fiecare un singur parametru fix și anume parametrul care definește formatele de conversie și eventualele texte constante care intervin în operațiile de intrare/iesire inițiate prin apelul lor.

Antetul unei funcții poate fi mai general decât cel indicat mai sus. Astfel, în cazul limbajului Turbo C++, antetul unei funcții are următoarea structură generală:

*specificator\_de\_clasă tip modificatori nume (lista declarărilor parametrilor formali)*

Specificatorul de clasă, dacă este prezent, este cuvintul *'cheie static* sau *extern*. Cuvintul cheie *static* se utilizează pentru a localiza funcția în fișierul în care este definită. Cuvintul cheie *extern* indică faptul că funcția este globală. În mod implicit o funcție se consideră că este globală și de aceea specificatorul *extern*, de obicei, nu se utilizează.

Tipul din antet definește tipul datei returnate de funcție. Dacă funcția returnează un alt tip decât cel indicat în antetul ei, atunci valoarea respectivă este convertită automat spre tipul indicat în antet, înainte de a reveni în punctul de după apel.

Dacă tipul nu este prezent în antetul funcției, atunci se consideră că funcția returnează o valoare de tip *int*. Cu toate acestea se recomandă ca tipul să fie prezent totdeauna. S-a constatat că omisiunea tipului din antetul funcțiilor conduce la erori, deoarece acesta se omite și cînd trebuie să fie prezent (adică atunci cînd este diferit de *int*) și o astfel de eroare nu poate fi depistată de compilator.

Modificatorii care pot apărea între tip și nume sunt specifici diferitelor implementări ale limbajului C++. În cazul limbajului Turbo C++ cei mai importanți modificatori sunt:

**cdecl, pascal, near, far și huge**

Primii doi stabilesc convenții de transfer a parametrilor și de apel a funcțiilor. Ei se folosesc cînd se utilizează module scrise în mai multe limbiage de programare.

Modificatorul *cdecl* activează convenția din C pentru apelul funcției respective. În mod analog, modificatorul *pascal* activează convenția din Pascal pentru apelul funcției respective. Convenția de apel implicită, în lipsa acestor modificatori, este cea definită prin opțiunile de compilare.

Modificatorii *near*, *far* și *huge* influențează apelurile funcțiilor. În mod implicit ei se definesc prin modelul de memorie curent și anume modelele *tiny*, *small* și *compact* implică modificatorul *near*, modelele *medium* și *large* implică modificatorul *far*, iar modelul *huge* implică modificatorul *huge*.

Un exemplu de utilizare a modificatorului *far* îl oferă funcțiile de gestiune grafică a ecranului (vezi capitolul 19). Acestea toate au în antet modificatorul *far*, deoarece implică apeluri și pointeri de tip *far*.

Prototipul unei funcții poate fi obținut scriind punct și virgulă după o construcție care, fie coincide cu antetul funcției respective, fie se obține din antet eliminind numele parametrilor formali.

Astfel, prototipul funcției *v\_calend* al cărui antet se indică în exemplul 1 de mai sus, se poate scrie în următoarele forme:

```
Boolean v_calend(int zi,int luna,int an=1994);
```

sau

```
Boolean v_calend(int,int,int=1994); *
```

Prima formă clarifică sensul parametrilor și de aceea ea este sugestivă în comparație cu cea de două.

Pentru exemplul 2, se pot utiliza următoarele prototipuri:

```
double aria(double r,double a = 0,double b = 360,  
           double pi = 3.14159265358979);
```

sau

```
double aria(double,double = 0,double = 360,  
           double = 3.14159265358979);
```

## Exerciții:

- 20.1 Să se scrie o funcție care citește un întreg scris în baza de numerație *b* ( $2 \leq b \leq 10$ ), îl convertește spre tipul *int* și-l memorează în zona spre care poinează parametrul *n* al funcției. Baza implicită a întregului este zece. Funcția returnează valoarea zero dacă se intilnește sfîrșitul de fișier și unu în caz contrar.

## FUNCȚIA BXX1

```
int bint(int *n,int b=10)  
/* - citește un întreg scris în baza b din intervalul [2,10];  
   - convertește numarul citit în binar de tip int;  
   - paștează numarul convertit în zona spre care poinează n;  
   - returnează 0 la sfîrșitul de fișier; altfel returnează 1;  
*/  
{  
    long s;
```

```

int c,sign;
char er[] = "se reia citirea caracterelor intregului\n";
for(;;){
    printf("tastati caracterele intregului = ");
    sign = 1;
    while((c = getchar()) == ' ' || c == '\t' || c == '\n')
        ; // salt peste caractere albe
    if(c == '-'){ // numar negativ
        sign = -1;
        c = getchar(); // salt peste minus
    }
    else
        if(c == '+') c = getchar(); // salt peste plus
    if(c == EOF) return 0;
    if(!(c >='0' && c <='0'+b)){
        // nu s-a citit o cifra din intervalul [0,b)
        printf("se cer cifre din intervalul [0,%d]\n",b);
        printf(er);
        fflush(stdin); // se videaza buferul de la intrarea standard
        continue;
    }
    for(s=0;c >='0' && c <='0'+b;){
        s = s*b+c-'0';
        if(s > 32767){
            printf("valoarea absoluta a numarului\
                    citit depaseste 32767\n");
            printf(er);
            fflush(stdin);
            break;
        }
        c = getchar();
    }
    if(s > 32767) // se reia citirea intregului
        continue;
    *n = sign*s;
    return 1;
}

```

- 20.2 Să se scrie un program care citește o succesiune de numere separate prin caractere albe și afișează fiecare număr pe căte o linie. Numerele citite sunt scrise în sistemul de numerație cu baza  $b$  ( $2 \leq b \leq 10$ ). După ultimul număr se tastează sfîrșitul de fișier. Numerele se afișeză în sistemul de numerație cu baza 10. Baza  $b$  este argumentul programului din linia de comandă.

### PROGRAMUL BXX2

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "BXX1.CPP"

```

```

main(int argc,char *argv[])
/* - citeste o succesiune de numere scrise in baza b si le afiseaza in sistemul de numerație cu
   baza 10;
   - dupa ultimul numar se tasteaza sfirsitul de fisier;
   - 2 <= b <= 10.
*/
{
    int b,lng,n;

    if(argc != 2){
        printf("baza de numerație nu este argument in \
               linia de comanda\n");
        exit(1);
    }
    lng = strlen(argv[1]);
    if(lng < 1 || lng > 2){
        printf("argument in linia de comanda eronat: %s\n",
               argv[1]);
        exit(1);
    }
    if(argv[1][0] < '0' || argv[1][0] > '9'){
        printf("argumentul din linia de comanda nu\
               este un intreg: %s\n", argv[1]);
        exit(1);
    }
    if(lng == 2 && argv[1][1] != '0'){
        printf("baza de numerație eronata: %s\n", argv[1]);
        exit(1);
    }
    b = atoi(argv[1]); // conversia bazei in binar
    if(b < 2 || b > 10){
        printf("baza de numerație eronata: %s\n", argv[1]);
        exit(1);
    }
    for(;;){
        if(b == 10) lng = bint(&n);
        else lng = bint(&n,b);
        if(lng) printf("%d\n",n);
        else break;
    }
}

```

### 20.2.2. Corpul unei funcții

*Corpul unei funcții* este o instrucțiune compusă. În limbajul C, o instrucțiune compusă este o succesiune de *instrucțiuni* care poate fi precedată de o succesiune de declarații, succesiuni care sunt incluse între acolade. În particular, o instrucțiune compusă se poate reduce la o pereche de acolade. Acest lucru este adevărat și în limbajul C++. În plus, în limbajul C++ nu este necesar ca intr-o instrucțiune compusă, declarațiile să preceadă instrucțiunile.

### Exemplu:

```
for(int i=1;i < 10;i++){ ... }
```

Variabila *i* se declară în antetul ciclului *for*. Această declarație este valabilă în continuare pînă la sfîrșitul blocului (instrucțunii compuse) care conține instrucționarea *for* respectivă.

Inserarea declaratiilor în orice punct al corpului unei funcții permite să declarăm variabilele locale atunci cînd avem nevoie de ele. În felul acesta se elimină situațiile în care se declară variabile locale la începutul corpului funcției care apoi rămîn neutilizate.

## 20.3. Tipuri predefinite în C și C++

Cuvînt cheie	Dimensiune în biți	Interval
char	8	[-128,127]
signed char	8	[-128,127]
unsigned char	8	[0,255]
enum	16	[-32768,32767]
short	16	[-32768,32767]
int	16	[-32768,32767]
unsigned	16	[0,65535]
long	32	[-2147483648,2147483647]
unsigned long	32	[0,4294967295]
float	32	[3.4*10**(-38),3.4*10**38]
double	64	[1.7*10**(-308),1.7*10**308]
long double	80	[3.4*10**(-4932),1.1*10**4932]
tip *	16 sau 32	
tip near *	16	
tip far *	32	
tip huge *	32	

În cazul datelor flotante (*float*, *double* și *long double*) se indică intervalele pentru valorile absolute.

Pointerii pot să spre o dată sau spre o funcție. Tipul pointer *tip \**, spre o dată, se reprezintă pe 16 biți pentru modelele de memorie *tiny*, *small* și *medium*. Pentru restul modelelor, datele de tip pointer se reprezintă pe 32 de biți.

Datele de tip *float* asigură o precizie de 6-7 cifre, cele de tip *double* o precizie de 15 cifre, iar cele de tip *long double* o precizie de maximum 19 cifre.

## 20.4. Constante caracter

În limbajul C, o constantă caracter are tipul *int*. Ea se reprezintă printr-un caracter imprimabil inclus între caractere apostrof sau printr-o secvență *escape* (vezi 1.6.3.).

În limbajul C++ o astfel de constantă caracter are tipul *char*.

În afară de aceste constante, în limbajul C++ se admit și constante caracter care au tipul *int*. Ele se reprezintă pe doi octeți.

### Exemple:

- 'a'                    - Constantă caracter păstrată în C++ pe un octet.
- 'ab'                  - Constantă caracter păstrată pe doi octeți.
- 'abc'                - Octetul mai puțin semnificativ păstrează codul ASCII al lui *a*, iar cel mai semnificativ, codul ASCII al lui *b*.
- 'abc'                - *sizeof ('ab')* are valoarea 2; are tipul int.
- 'abc'                - Eroare.
- 'abc'                - Constantă caracter nu poate avea decît cel mult 2 caractere.

### Exerciții:

20.3 Să se scrie un program care afișează dimensiunile în octeți ale constantelor caracter 'a' și 'ab', precum și valorile octetilor pe care se reprezintă acestea.

### PROGRAMUL BXX3

```
#include <stdio.h>

main()
/* - afiseaza dimensiunile in octeti ale constantelor 'a' si 'ab';
   - afiseaza valorile octetilor pe care se reprezinta cele doua constante caracter.
*/
{
    printf("dim lui 'a' = %d\n", sizeof('a'));
    printf("dim lui 'ab' = %d\n", sizeof('ab'));
    printf("codul lui 'a' = %d\n", 'a');
    printf("octetul mai putin semnificativ a lui 'ab' = %d\n",
           'ab' & 0377);
    printf("octetul mai semnificativ a lui 'ab' = %d\n",
           ('ab' >> 8) & 0377);
```

## 20.5. Operatori

Operatorii utilizati in limbajul C pot fi utilizati si in limbajul C++ si au aceleasi prioritate si asociativitate.

Tabela cu prioritatile operatorilor din limbajul C este data in paragraful 3.2.16.

In paragraful de fata extindem tabela respectiva cu operatori noi, specifici limbajului C++.

### 20.5.1. Operatorul de rezolutie

Acest operator permite accesul la o data globala redefinita ca locala. El se noteaza prin ::

Fie variabila i definita ca globala:

```
int i;
```

si functia f in care variabila i se redeclară ca si locală:

```
void f(...)  
{  
    ...  
    char i;  
    ...  
}
```

In aceasta situatie, o expresie de forma:

```
i=10;
```

scrisa in corpul functiei f, atribuie valoarea 10 variabilei locale i declarata in corpul functiei respective.

Pentru a utiliza variabila globala i, in corpul functiei f, este necesar ca i sa fie precedata de operatorul de rezolutie. Cu alte cuvinte, expresia:

```
*{ ::i = 10;
```

va atribui valoarea 10, variabilei globale i.

Operatorul de rezolutie este de prioritate maxima, adica are aceeasi prioritate cu operatorii:

```
( ) [] . si ->
```

Deci, in tabela operatorilor amintita mai sus, vom completa prima linie cu operatorul de rezolutie (::).

### 20.5.2. Operatorul adresă (&)

In limbajul C operatorul adresă se utilizează pentru a defini adresa unei variabile. Astfel, construcția:

*tip&*

definește adresa de inceput a zonei de memorie alocată pentru *nume*.

In limbajul C++ acest operator are si o alta utilizare si anume aceea de a introduce tipul referinta.

Daca tipul pointer se introduce prin constructia:

*tip \**

tipul referinta se introduce prin:

*tip &*

Tipul referinta creeaza sinonime pentru nume deja definite. De asemenea, datele de acest tip pot fi folosite pentru a realiza apelul prin referinta (call by reference).

Avind in vedere aceasta utilizare noua a operatorului &, se obisnuiese ca acesta sa mai fie numit si *operator de referinta*.

Operatorul unar \* utilizat la declararea pointerilor are in expresii un efect invers celui de referinta si de aceea el se mai numeste operator de *derefereinta*.

Construcția *tip &* se utilizează in declarații de forma:

*tip &nume;*

Menționăm că poziția operatorului & in declarația de mai sus este variabilă, adică declarația de mai sus poate fi scrisă ca mai jos:

*tip & nume;*

sau

*tip& nume;*

Ultima formă este cea mai utilizată.

Numele declarat in acest fel se spune că este o referinta.

O variabila referinta se poate initializa la definirea sau declararea ei, cu numele unei alte variabile.

**Exemplu:**

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

Variabila j este sinonimă cu variabila i. Ele reprezintă același intreg. De exemplu, dacă atribuim lui i o valoare:

```
i = 1234;
```

atunci și *j* are (referă) aceeași valoare.

Variabila *j* de mai sus, este un alt nume cu ajutorul căruia avem acces la întregul păstrat în zona de memorie alocată lui *i*.

De aceea, expresiile:

*i*\*7+3

și

*j*\*7+3

au aceeași valoare.

De asemenea, atribuirea:

*j*=12345

păstrează întregul 12345 în zona de memorie alocată pentru *i*.

Declarația:

int & *j*=*i*;

trebuie interpretată ca fiind procedeul prin care *j* se definește a fi un nume sinonim pentru *i*. Acest mod de interpretare a declarației de mai sus diferă de sensul obișnuit al inițializărilor care se fac prin declarații.

Variabilele referință se utilizează frecvent pentru a avea acces la zonele de memorie alocate dinamic în memoria heap.

### 20.5.3. Operatorul de alocare dinamică a memoriei (new)

În limbajul C se pot aloca zone de memorie în memoria heap folosind funcții de bibliotecă. Astfel de funcții sunt, de exemplu, funcțiile *malloc* și *calloc* (vezi 8.8).

În afara acestor funcții, limbajul C++ permite alocări în zona heap prin intermediul operatorului *new*. Aceasta este unar și are aceeași prioritate ca și ceilalți operatori unari.

Operatorul *new* are ca valoare adresa de început a zonei de memorie alocată în memoria heap sau zero (pointerul nul) în cazul în care nu se poate face alocarea.

Operandul operatorului *new* în cea mai simplă formă, este numele unui tip (predefinit sau definit de utilizator).

Exemple:

1. int \*pint;  
pint = new int;

Prin intermediul acestei expresii, se alocă în memoria heap, o zonă de memorie în care se pot păstra date de tip int.

Adresa de început a zonei alocate se atribuie pointerului *pint*. Expresia:

\*pint = 100;

păstrează întregul 100 în zona respectivă.

Aceeași alocare se obține, în limbajul C, folosind funcția *malloc* ca mai jos:

int\* pint = (int \*)malloc(sizeof(int));

2.

int& i = \*new int;

Prin intermediul acestei declarații (definiții) se alocă în memoria heap o zonă de memorie în care se pot păstra date de tip int. Numele *i* permite referirea la întregul păstrat în zona respectivă. Expresia de atribuire:

i = 100

păstrează întregul 100 în zona respectivă.

Zonele de memorie alocate cu ajutorul operatorului *new* pot fi inițializate.

În acest scop se utilizează o expresie de forma:

new tip[expresie]

unde:

tip

- Este numele unui tip.

expresie

- Este o expresie a cărei valoare inițializează zona de memorie alocată prin operatorul *new*.

Exemple:

1. double \*pdouble;  
pdouble = new double(3.14159265);

Această instrucție realizează următoarele:

- alocă în memoria heap o zonă de memorie în care se păstrează valoarea 3.14159265 în flotantă dublă precizie;
- adresa de început a acestei zone de memorie se atribuie variabilei *pdouble*.

2.

double& pi = \*new double(3.14159265);

Prin această declarație se rezervă, în memoria heap, o zonă de memorie în care se păstrează valoarea 3.14159265 în flotantă dublă precizie. Data respectivă se poate referi cu ajutorul numelui *pi*. De exemplu, *pi* poate fi utilizat în mod obișnuit în expresii de forma:

pi\*r\*r  
sin(pi/2)  
x\*180/pi  
etc.

O altă utilizare importantă este aceea de alocare a unei zone de memorie, în memoria heap, pentru tablouri.

În acest scop, utilizăm o expresie de forma:

new tip[exp]

unde:

exp

- Este o expresie de tip intreg.

Prin această construcție se rezervă, în memoria heap, o zonă de memorie de  $\text{exp}^*\text{sizeof}(tip)$  octeți.

Valoarea expresiei de mai sus, este adresa de început a zonei de memorie rezervată prin operatorul *new*.

Exemplu:

```
double *tab;  
int m,n;  
...  
tab=new double[m*n];
```

Această instrucție rezervă în memoria *heap* o zonă de  $m*n*\text{sizeof(double)}$  octeți. Adresa de început a acestei zone de memorie se atribuie pointerului *tab*.

Elementele unei astfel de zone de memorie nu pot fi inițializate decit numai prin secvențe de program corespunzătoare. De exemplu, pentru a anula cele  $m*n$  elemente de tip *double* rezervate ca mai sus, putem folosi instrucția *for* de mai jos:

```
for(int i=0; i < m*n;i++) tab[i]=0.0;
```

#### 20.5.4. Operatorul de dezalocare a memoriei (delete)

O zonă de memorie alocată prin operatorul *new* se eliberează prin operatorul *delete*.

Dacă *p* este un pointer spre *tip*:

```
tip *p;
```

și

```
p = new tip;
```

atunci zona din memoria *heap* alocată cu ajutorul lui *new* se eliberează folosind construcția:

```
delete p;
```

De asemenea, operatorul *delete* se utilizează pentru adezaloca tablourile alocate prin *new*.

Fie alocarea:

```
tip *p = new tip[expresie];
```

Această zonă se eliberează folosind o construcție de forma:

```
delete [expresie] p;
```

\* } Menționăm că expresia care definește numărul elementelor tabloului nu este totdeauna necesară la dezalocare.

Mai tîrziu, vom reveni asupra operatorilor *new* și *delete* și vom indica situația cind este necesară prezența dimensiunii tabloului la dezalocarea lui.

La utilizarea operatorului *delete* este important ca pointerul la care se aplică să aibă o valoare obținută ca rezultat al aplicării prealabile a operatorului *new*.

Dacă operatorul *delete* se aplică la un pointer nenufără căruia valoare nu a fost obținută ca rezultat al aplicării operatorului *new*, atunci efectul aplicării lui *delete* este imprevizibil.

## 20.6. Apel prin referință (call by reference)

În limbajul C apelul se face prin valoare (call by value). Acesta devine prin referință cind parametrul efectiv este un nume de tablou (vezi 4.15.). De asemenea, apelul prin referință se poate realiza cu ajutorul pointerilor (vezi 8.2). În capitolul 8 sunt date o serie de exerciții care utilizează pointerii pentru a realiza apelul prin referință (ex. 8.1, 8.2, 8.3, 8.6, 8.7).

În limbajul C++ se pot utiliza toate facilitățile amintite mai sus cu privire la apelul funcțiilor. În plus, s-a introdus apelul prin referință. Aceasta se realizează prin intermediul parametrilor de tip referință.

Exemplul devenit deja clasic pentru explicarea apelului prin referință este cel al funcției de permutare a valorilor a două variabile.

Fie funcția *perm* definită ca mai jos:

```
void perm(int x, int y)  
{  
    int t;  
  
    t = x; x = y; y = t;  
}
```

Această funcție nu are nici un efect asupra parametrilor efectivi de la apelurile ei.

Intr-adevăr, fie apelul:

```
int a,b;  
...  
perm(a,b);  
...
```

Apelul fiind prin valoare, se atribuie parametrilor formali *x* și *y* valorile lui *a* și respectiv *b*. Funcția permute valorile parametrilor formali *x* și *y* dar această permutare nu are nici un efect asupra parametrilor efectivi *a* și *b*.

Pentru ca funcția să permute valorile parametrilor efectivi este necesar să se realizeze apelul ambilor parametri prin referință.

În limbajul C (și deci și în limbajul C++) este posibil să realizăm acest lucru folosind pointerii, ca mai jos:

```
void pperm( int *x, int *y)  
{  
    int t;
```

```
t = *x; *x = *y; *y = t;
}
```

În acest caz, funcția *pperm* se apelează astfel:

```
int a,b;
...
pperm(&a,&b);
```

Prin acest apel se atribuie pointerilor *x* și *y* adresele parametrilor *a* și respectiv *b*. Funcția *pperm* nu permute valorile pointerilor *x* și *y* ci valorile intregilor spre care pointează cei doi pointeri, adică chiar valorile lui *a* și *b*.

În limbajul C++ se poate defini funcția de permutare cu parametri formali de tip referință:

```
void rperm(int& x,int& y)
{
    int t;
    t = x; x = y; y = t;
}
```

Funcția *rperm* se apelează astfel:

```
int a,b;
...
rperm(a,b);
...
```

În acest caz *x* și *y* sunt sinonime cu variabilele *a* și respectiv *b*. Aceasta înseamnă că *x* și *y* accesează aceleași date din memorie ca și *a* și *b*. De aceea, permutarea valorilor referite de *x* și *y* înseamnă permutarea valorilor referite de *a* și *b*.

Comparind funcțiile *perm* și *rperm* observăm că ele se apelează la fel și singura diferență dintre ele constă în modul de declarare al parametrilor formali. În cazul funcției *perm* parametrii formali sunt date de tip *int*, iar în cazul funcției *rperm* aceștia sunt referințe la date de tip *int*.

#### Exerciții:

20.4 Să se scrie o funcție care afișează caracterele unui tablou și citește un întreg de tip *int*. Funcția are doi parametri:

*text*:

- Tablou unidimensional de tip caracter și care are aceeași utilizare ca în exercițiul 6.3.

*x*:

- Parametru de tip referință la intregi de tip *int*. Numărul citit se păstrează în zona de memorie alocată parametrului efectiv corespunzător lui *x*.

Funcția returnează valoarea zero la întîlnirea sfîrșitului de fișier și unu în caz contrar.

Această funcție este analogă cu funcțiile definite în exercițiile 6.3 și 8.2.

În cazul funcției din exercițiul 6.3, numărul citit se atribuie variabilei globale *v\_int*. Funcția definită în exercițiul 8.2., păstrează numărul citit în zona de memorie spre care pointează parametrul formal *x* de tip pointer.

Cu alte cuvinte, aceste funcții sunt 3 variante pentru a citi un întreg de tip *int* după ce, în prealabil, ele afișează un sir de caractere. Funcția definită în exercițiul 6.3. este varianta care utilizează o variabilă globală, cea din exercițiul 8.2. este varianta cu *pointer*, iar funcția definită mai jos este varianta cu parametru de tip referință (apel prin referință).

#### FUNCȚIA BXX4

```
int rcit_int(char text[],int& x)
/* - afișaza sirul de caractere spre care pointează text;
   - citește un întreg și-l păstrează în zona de memorie alocată parametrului efectiv
     corespunzător lui x;
   - returnează:
     0 - la întîlnirea sfîrșitului de fisier;
     1 - altfel.
*/
{
    char t[255];
    for(;;){
        printf(text);
        if(gets(t)==0) return 0;
        if(sscanf(t,"%d",&x)==1) return 1;
    }
}
```

#### Observație:

Funcția de față se apelează prin apeluri de forma:

```
int n;
...
if(rcit_int("n=",n)){ ... }
else // s-a tastat EOF
...
```

Parametrul *x* este sinonim cu *n* și de aceea la apelul funcției *sscanf*, expresia *&x* înseamnă *&n*.

20.5 Să se scrie o funcție care citește un întreg de tip *int* care aparține unui interval dat.

Această funcție este analogă cu funcția definită în exercițiul 8.3. Ea folosește un parametru de tip referință în locul parametrului de tip pointer *oint* al funcției din exercițiul amintit mai sus.

#### FUNCȚIA BXX5

```
int rcit_int_lim(char text[],int inf,int sup,int &rint)
```

```

/* - afiseaza sirul de caractere spire care pointeaza text;
   - citeste un intreg de tip int ce apartine intervalului [inf,sup] si-l pastreaza in zona de
     memorie alocata parametrului efectiv corespunzator lui rint;
   - returneaza:
     0 - la intilnirea sfirsitului de fisier;
     1 - altfel.
*/
{
    for(;;){
        if(rcit_int(text,rint)==0)    return 0; //s-a intilnit EOF
        if(rint >= inf&&rint <= sup) return 1;
        printf("intregul tastat nu apartine intervalului:");
        printf("[%d,%d]\n",inf,sup);
        printf("se reia citirea\n");
    }
}

```

- 20.6 Sa se scrie o funcție care citește o dată calendaristică compusă din *zi*, *luna* și *an*. Funcția validează data calendaristică respectivă.

Această funcție este analogă cu funcția definită în exercițiul 8.4. Deosebirea dintre ele constă în aceea că parametrii funcției din exercițiul 8.4. sunt de tip pointer, iar cei din exercițiul de față sunt de tip referință.

### FUNCȚIA BXX6

```

int rcit_data_calend(int& rzi,int& rluna,int& ran)
/* - citeste data calendaristica, o valideaza si o pastreaza in zonele de memorie alocate
parametrilor efectivi de la apel;
   - returneaza:
     0 - la intilnirea sfirsitului de fisier;
     1 - altfel.
*/
{
    static char ziua[] = "ziua:";
    static char luna[] = "luna:";
    static char an[] = "an:";
    static char er[] = "s-a tastat EOF";

    for(;;){ //se citeste ziua,luna si anul
        if(rcit_int_lim(ziua,1,31,rzi)==0){
            printf("%s\n",er);
            return 0;
        }
        if(rcit_int_lim(luna,1,12,rluna)==0){
            printf("%s\n",er);
            return 0;
        }
        if(rcit_int_lim(an,1600,4900,ran)==0){
            printf("%s\n:",er);
            return 0;
        }
    }
}

```

```

// validare data calendaristica
if(v_calend(rzi,rluna,ran)) return 1;
printf("data calendaristica este eronata\n");
printf("se reia citirea datei calendaristice\n");
}
}

```

- 20.7 Să se modifice funcția din exercițiul 8.1, înlocuind parametrii de tip pointer cu parametrii de tip referință.

### FUNCȚIA BXX7

```

void rluna_si_ziua(int zz,int an,int & zi,int& luna)
/* determina luna si ziua din luna;
zz - este ziua din an;
an - este anul calendaristic;
zi - referinta la zona in care se pastreaza ziua din luna;
luna - referinta la zona in care se pastreaza luna.
*/
{
    int bisect = an %4==0&&an%100||an%400==0;
    int i;
    extern int nrzile[];

    for(i=1; zz>nrzile[i]+(i==2&&bisect); i++)
        zz -= (nrzile[i]+(i==2&&bisect));
    zi = zz;
    luna = i;
}

```

- 20.8 Să se scrie un program care citește o dată calendaristică și afișează data calendaristică pentru ziua următoare.

Acest program este analog cu cele definite în exercițiile 6.9. și 8.5.

Programul de față folosește funcții care utilizează parametri de tip referință pentru interfață în locul variabilelor globale sau al parametrilor de tip pointer.

### PROGRAMUL BXX8

```

#include <stdio.h>
#include <stdlib.h>
#include "bxx4.cpp" // functia rcit_int
#include "bxx5.cpp" // functia rcit_int_lim
#include "bvi5.cpp" // functia v_calend
#include "bvi6.cpp" // functia zi-din-an
#include "bxx6.cpp" // functia rcit_data_calend
#include "bxx7.cpp" // functia rluna_si_ziua

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

main()
/* citeste o data calendaristica, o valideaza si in caz ca este corecta,
afișeaza data calendaristica a zilei urmatoare */

```

```

{
    int zz, l1, aa;

    // citeste si valideaza data calendaristica
    if(rcit_data_calend(zz, l1, aa)==0) exit(1);
    if(zz==31&&l1==12){
        /* 31 decembrie; ziua urmatoare este 1 ianuarie din anul urmator */
        zz=1; // 1 ianuarie
        l1=1;
        aa++; // anul urmator
    }
    else // se determina ziua urmatoare
        rluna_si_ziua(zi_din_an(zz, l1, aa)+1, aa, zz, l1);

    // afiseaza data calendaristica a zilei urmatoare
    printf("ziua:%d\tluna:%d\tan:%d\n", zz, l1, aa);
}

```

20.9 Să se scrie o funcție care citește:

- valoarea variabilei  $m$  de tip *int*;
- valoarea variabilei  $n$  de tip *int*;
- $m \times n$  numere care reprezintă elementele unei matrice de ordinul  $m \times n$ .

Funcția de față este analogă cu funcția definită în exercițiul 8.6. Ea folosește parametri de tip referință în locul parametrilor de tip pointer.

### FUNCȚIA BXX9

```

int rdcitmat(double dmat[], int max, int& nrlin, int& nrcol)
/* - citeste pe m - numar de linii;
   n - numar de coloane;
   m*n - numar de tip double pe care le pastreaza in matrica dmat prin liniarizare;
   - returneaza valoarea m*n;
   - valoarea lui m se pastreaza in zona referentiata de nrlin;
   - valoarea lui n se pastreaza in zona referentiata de nrcol.
*/
{
    int i;
    char t[255];
    char er[] = "s-a tastat EOF\n";

    do{ // se citesc valorile lui m si n
        // citeste pe m
        if(rcit_int_lim("numarul de linii=", 1, max, nrlin)==0){
            printf(er);
            exit(1);
        }

        // citeste pe n
        if(rcit_int_lim("numarul de coloane=", 1, max, nrcol)==0){
            printf(er);
            exit(1);
        }
    }

```

```

        i=nrlin*nrcol;
        if(i <= max) break;
        printf("produsul m*n = %d depaseste pe max=%d\n", i, max);
        printf("se reiau citirile lui m si n\n");
    }while (1);

    /* se citesc cele m*n elemente ale matricii tastate pe linii */
    if(ndcit(i, dmat) != i){
        printf("nu s-au tastat %d elemente\n", i);
        exit(1);
    }
    return i;
}

```

20.10 Să se scrie un program care citește elementele de tip *double* ale două matrice  $a$  și  $b$ , calculează produsul lor și-l afișează.

Acest program este analog cu cel din exercițiul 8.7.

În cazul programului de față se folosesc funcții cu parametri de tip referință în locul funcțiilor cu parametri de tip pointer.

### PROGRAMUL BXX10

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "bxx4.cpp" // rcit_int
#include "bxx5.cpp" // reit_int_lim
#include "biv37.cpp" // ndcit
#include "bxx9.cpp" // rdcitmat

#define MAX 900

main() /* citeste elementele a doua matrice, calculeaza si
           afiseaza matrica produs, cite 4 elemente pe o linie */
{
    int i, j, k;
    int m, n, p, s, r, q;
    int mn, np;

    double a[MAX], b[MAX], c[MAX];

    // citeste matrica a
    mn=rdcitmat(a, MAX, m, n);

    // citeste matrica b
    np=rdcitmat(b, MAX, p, s);
    if(n!=p){
        printf("numarul coloanelor matricei a=%d\n", n);
        printf("diferă de numarul liniilor ");
        printf("matricei b=%d\n", p);
        exit(1);
    }
}

```

```

// se calculeaza produsul c=a*b
for (i=0; i<m; i++) {
    q=i*n;
    for (j=0; j<s; j++) {
        r=i*s+j;
        c[r]=0.0;
        for (k=0; k<n; k++)
            c[r] += a[q+k]*b[k*s+j];
    }
}

// afiseaza elementele matricei produs
printf("\n\n\t\t matricea produs\n");
k=1;
for (i=0; i<m; i++) {
    p = i*s;
    for (j=0; j<s; j++) {
        printf("c[%d,%d]=%8g ", i, j, c[p+j]);
        if (j%4==3) {
            // afiseaza 4 elemente pe un rind
            printf("\n");
            k++; // numara liniile
        }
        if (k==23) {
            printf("actionati o tasta pentru a continua\n");
            getch();
            k=1;
        }
    }
    printf("\n");
    k++;
}

```

- 20.11 Sa se scrie o funcție care păstrează un sir de caractere într-o zonă de memorie alocată în memoria *heap* cu ajutorul operatorului *new*. Funcția returnează adresa de început a zonei în care se păstrează sirul de caractere sau zero (pointerul nul), în cazul în care nu se poate rezerva zona respectivă în memoria *heap*.

Aceasta funcție este analoga cu funcția din exercițiul 8.15. Aceasta din urmă utilizează funcția *malloc* pentru a aloca zona de memorie *heap*.

### FUNCȚIA BXX11

```

char *nmemsir(char *s)
/* păstreaza in memoria heap sirul de caractere spre care pointeaza s */
{
    char *p;
    if ((p=new char[strlen(s)+1]) !=0) {
        /* p are ca valoare adresa de început a zonei de memorie rezervata in
           memoria heap si care are strlen(s)+1 octeti */
        strcpy(p,s); // copierea sirului in memoria heap
    }
}

```

```

        return p;
    }else return 0; // nu s-a putut rezerva zona in memoria heap
}

```

- 20.12 Să se scrie un program care citește o succesiune de cuvinte și-l afișează pe cel mai mare.

Acest program este analog cu cel din exercițiul 8.16. În cazul de față, se utilizează operatorul *delete* pentru a elibera o zonă de memorie din zona *heap* spre deosebire de programul din exercițiul 8.16, care utilizează în acest scop funcția *free*.

### PROGRAMUL BXX12

```

#include <stdio.h>
#include <string.h>
#include "bxx11.cpp" // nmemsir

#define MAX 100

main()
/* citeste o succesiune de cuvinte si-l afiseaza pe cel mai mare */
{
    char cuvcrt[MAX+1];
    char *cuvmax=0; // pointeaza spre cuvintul maxim

    while(scanf("%100s",cuv crt)!=EOF)
        if(cuvmax==0) // prima citire
            cuvmax=nmemsir(cuv crt);
        else
            if(strcmp(cuv crt,cuvmax)>0) {
                /* cuvintul citit este mai mare decat cel din memoria
                   heap si spre care pointeaza cuvmax */
                delete cuvmax; // se elibereaza zona din memoria heap
                // se păstreaza cuvintul citit curent in memoria heap
                cuvmax=nmemsir(cuv crt);
            }
    printf("cel mai mare cuvint este\n");
    printf("%s\n",cuvmax);
}

```

### 20.7. Funcții care returnează date de tip referință

În limbajul C++, o funcție poate returna o referință la o dată de un tip oarecare. În acest caz, *tip* din antetul unei funcții are forma:

*tip&*

**Exemplu:**

```

typedef struct {
    double x;
}

```

```

    double y;
} COMPLEX;
COMPLEX z;
...
double& re = retreal(z);
double& im = retimag(z);

```

Funcțiile *retreal* și *retimag* returnează referință la partea reală și respectiv cea imaginară a numărului complex *z*. Ele se definesc ca mai jos:

```

double& retreal(COMPLEX& c)
{
    return c.x;
}
double& retimag(COMPLEX& c)
{
    return c.y;
}

```

O funcție care returnează o referință poate fi apelată atât în dreapta semnului egal *cit și în stînga*. Un astfel de apel este un operand atât *lvalue* *cit și rvalue* (vezi 8.7.).

De exemplu, funcțiile *retreal* și *retimag* pot fi utilizate pentru a inițializa numărul complex *z* din exemplul de mai sus, scriind:

```

retreal(z) = 1;
retimag(z) = -3;

```

Prima instrucțiune are același efect cu atribuirea:

```

z.x = 1;

```

iar cea de a doua, cu atribuirea:

```

z.y = -3;

```

Evident, atribuirile de forma *z.x = ...* și *z.y = ...* sint operații simple care nu se justifică a fi înlocuite prin atribuiri în care se apelează funcțiile *retreal* și *retimag*.

Astfel de apeluri se utilizează în cazul în care funcțiile apelate reprezintă procese de calcul mai complexe.

Apelurile funcțiilor care returnează referință, fiind operanzi de tip *lvalue*, lor li se pot aplica operatorii de incrementare și decrementare. Astfel, instrucțiunile:

```

++retreal(z);

```

și

```

retreal(z)++;

```

sunt corecte și ele incrementează partea reală a lui *z*.

În principiu, apelul unei funcții care returnează o referință trebuie considerat ca și cum ar reprezenta un nume.

De exemplu, *retreal(z)* este echivalent cu numele calificat *z.x*.

Fie tipul:

```

typedef struct {
    COMPLEX z;
    double modul;
    double arg;
} NRC;

```

și declarația:

```

NRC nrc;

```

Pentru a ne referi la componentele *x* și *y* ale datei *nrc* se utilizează numele calificat:

*nrc.z.x* pentru partea reală

și

*nrc.z.y* pentru partea imaginară.

De exemplu:

```

nrc.z.x = 3;
nrc.z.y = 4;

```

definesc părțile reală și respectiv imaginară a componentei *z* a datei structurate *nrc*.

Același lucru se realizează folosind următoarele atribuiri:

```

nrc.retreal(nrc.z) = 3;
nrc.retimag(nrc.z) = 4;

```

sau dacă considerăm declarația:

```

COMPLEX& rc = nrc.z;

```

atunci aceleași atribuiri pot fi scrise astfel:

```

nrc.retreal(rc) = 3;
nrc.retimag(rc) = 4;

```

### Exerciții:

20.13 Să se scrie o funcție care calculează modulul unui număr complex și returnează o referință la el.

Această funcție este analogă cu cea definită în exercițiul 10.1.

### FUNCȚIA BXX13

```

double& rmodul(COMPLEX& z)
/* calculeaza modulul numarului complex si returneaza o referinta la el */
{
    static double c;
    return c=sqrt(z.x*z.x+z.y*z.y);
}

```

20.14 Să se scrie o funcție care calculează argumentul unui număr complex și

returnează o referință la el.

Această funcție este analoga cu cea definită în exercițiul 10.2.

## FUNCȚIA BXX14

```
double& rarg(COMPLEX& z)
/* calculeaza argumentul numarului complex si returneaza o referinta la el */
{
    static double a;

    if(z.x == 0 && z.y == 0) return a=0.0;
    if(z.y == 0)
        if(z.x >0) return a=0.0;
        else //y=0 si x<0
            return a=PI;
    if(z.x == 0)
        if(z.y > 0) return a=PI/2;
        else return a=(3*PI)/2;

    //x!=0 si y!=0
    a=atan(z.y/z.x);
    if(z.x < 0) //x<0 si y!=0
        return a=a+PI;
    else
        if(z.y < 0) //x>0 si y<0
            return a=2*PI+a;
    //x>0 si y>0
    return a;
}
```

### Observație:

Variabila *a* a fost declarată statică din cauză că funcția returnează o referință la ea (se utilizează instrucția *return a;*).

Dacă variabila *a* ar fi fost automatică, ea s-ar fi alocat pe stivă. În acest caz o referință la ea nu are sens, deoarece la revenirea din funcție se face curățirea stivei și deci variabila *a* este dezalocată.

20.15 Să se scrie un program care citește numere complexe și le afișează împreună cu modulul și argumentul fiecaruia.

## PROGRAMUL BXX15

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

#define PI 3.14159265358979

typedef struct {
    double x;
    double y;
}COMPLEX;
```

```
#include "bxx13.cpp" // rmodul
#include "bxx14.cpp" // targ

main()
/* citeste numere complexe si le afiseaza impreuna cu modulul si argumentul fiecaruia */
{
    COMPLEX complex;

    while(scanf("%lf %lf",&complex.x,&complex.y)==2){
        printf("a+ib=%g+i*(%g)\n",complex.x,complex.y);
        printf("modul=%g\targ=%g\n",rmodul(complex),
               rarg(complex));
    }
}
```

## 20.8. Modificatori

Modificatorii sunt cuvinte cheie care se utilizează în declarații sau definiții de variabile și funcții.

Cei mai folosiți modificatori utilizati la definirea funcțiilor sunt:

**cdecl, pascal, near, far și huge.**

Sensul acestor modificatori este amintit în paragraful 20.2.

În acest paragraf amintim modificatorii utilizati mai frecvent în legătură cu variabilele.

Aceștia sunt:

**const, near, far și huge.**

### 20.8.1. Modificatorul const

În principiu, modificatorul *const* se utilizează în definiții sau declarații pentru a defini date constante.

Cuvintul cheie *const* trebuie considerat că modifica tipul unei date restrințind modul de utilizare al datei respective.

De obicei, cuvintul *const* precede, în declarație, tipul datei.

#### Exemple:

1. const int versiune = 3;
2. const double pi = 3.14159265358979;
3. const int xci[] = {'a','b','c','d','e','f'};

Datele declarate (definite) în acest fel nu pot fi modificate direct, deci atribuirile de forma:

```
versiune = 4;
pi = 3.14159;
xci[0] = 'A';
```

sint eronate.

Într-o declarație (definiție) în care se utilizează modificatorul *const*, se poate omite tipul. În acest caz se subînțelege tipul *int*. Astfel, constanta declarată ca mai jos:

```
const an = 1995;
```

are tipul *int*.

Modificatorul *const* are utilizări importante mai ales în legătură cu pointerii.

O construcție de forma:

```
* tip *nume = ... ;
```

definește un pointer spre o *dată constantă*.

De exemplu, fie declarația:

```
const char *sir = "abc";
```

Prin această declarație *sir* devine un pointer spre un sir constant. Deci atribuirile de forma:

```
*sir = 'A';  
*(sir+1) = 'B' etc.
```

sunt eronate.

O construcție de forma:

```
tip *const nume = ... ;
```

definește un *pointer constant* spre o *dată care nu este constantă*.

De exemplu, fie declarațiile:

```
char *const psir = "abc";  
char *s;  
const char *sir = "sir";
```

Atunci, o atribuire de forma:

```
sir = s
```

este corectă, dar:

```
psir = s
```

nu este acceptată deoarece *psir* este un pointer constant. În schimb, atribuirile:

```
*psir = 'A'  
*(psir+1) = 'B'  
*(psir+2) = 'C'
```

sunt corecte.

Este posibil să se definească un pointer constant spre o *dată constantă*. În acest scop se utilizează o declarație (definiție) de forma:

```
const tip *const nume = ... ;
```

De exemplu, fie declarația:

```
const char *const atribut = "IBM PC";
```

În acest caz, *atribut* este un pointer constant spre un sir constant de caractere.

Amintim că modificatorul *const* se utilizează frecvent la declararea parametrilor formali de tip pointer pentru a interzice funcțiilor respective modificarea datelor spre care pointează parametrii respectivi.

Astfel de parametri se întâlnesc, de exemplu, la funcțiile de prelucrare a sirurilor de caractere:

```
unsigned strlen(const char *s);  
char *strcpy(char *dest, const char *sursa);  
etc.
```

Funcția *strlen* nu poate modifica data spre care pointează *s*. În mod analog, funcția *strcpy* nu poate modifica data spre care pointează *sursa*.

Modificatorul *const* poate fi utilizat și pentru a proteja o *dată returnată* de o funcție.

De exemplu, funcția de mai jos returnază un pointer spre un sir constant, care nu poate fi modificat în mod accidental (direct).

```
const char *denlun(int i)  
/* returneaza un pointer la denumirea lunii a i-a */  
{  
    static const char *tdl[] =  
        {"luna ilegală", "ianuarie", ..., "decembrie"};  
    return(i < 1 || i > 12) ? tdl[0] : tdl[i];  
}
```

Protecția datelor obținută cu ajutorul modificatorilor *const* nu este totală.

Într-adevăr, dacă vom considera declarațiile de mai jos:

```
const char *sir = "abc";  
char *s;
```

atunci o atribuire de forma:

```
s = sir
```

este eronată deoarece *sir* și *s* sunt pointeri de tipuri diferite:

*sir* este un pointer spre o *dată constantă* de tip caracter, iar *s* este un pointer spre o *dată de tip caracter*.

În schimb, o atribuire de forma:

```
s = (char *)sir
```

este corectă. În continuare *s* pointează spre aceeași *dată* ca și *sir*.

Cum *s* nu a fost declarat cu ajutorul modificatorului *const*, rezultă că atribuirile de mai jos sunt legale:

```
*s = 'A';  
*(s+1) = 'B';
```

```
*(s+2) = 'C';
```

În felul acesta, data "abc", protejată cu ajutorul modificadorului *const*, a putut fi modificată cu ajutorul pointerului *s*. O modificare de acest fel se spune că este o *modificare indirectă*.

Modificadorul *const* nu permite *modificarea directă* a datei, adică modificări de forma:

```
*sir = ...  
*(sir+1) = ...  
*(sir+2) = ...
```

În felul acesta, modificadorul *const* protejează datele împotriva unor modificări directe care pot surveni în urma unor erori.

## 20.8.2. Modificatorii *near*, *far* și *huge*

Acești modificatori sunt specifici limbajului TURBO C++. Ei dă posibilitatea utilizatorului de a defini explicit dimensiunea pointerelor: 16 sau 32 biți.

În cazul limbajului C, modelul memoriei (tiny, small, ...) determină formatul intern al pointerelor.

În C++ se pot folosi declarații de forma:

```
tip modificador *nume;
```

unde:

*modificador* - Este unul din cuvintele cheie: *near*, *far* sau *huge*.

Modificadorul *near* specifică faptul că pointerul se reprezintă pe 16 biți. Ceilalți modificatori forțează reprezentarea pointerelor pe 32 de biți.

Utilizarea acestor modificatori poate conduce, uneori, la economie de memorie și de timp calculator evitând lucru cu pointerii pe 32 de biți cind acesta nu este necesar.

Diferența dintre modificatorii *far* și *huge* se manifestă la calculele cu operanzi de tip pointer. Operațiile cu pointeri de tip *far* pot conduce uneori la rezultate eronate spre deosebire de cele realizate asupra pointerilor de tip *huge*, operații care se efectuează totdeauna corect, dar sunt mai costisitoare în timp.

Acești modificatori nu sunt necesari pentru programe de dimensiuni relativ mici și care nu utilizează volume mari de date.

Programele din volumul de față se incadrează în această categorie și de aceea nu vom utiliza acești modificatori în exercițiile care urmează.

O excepție de la această regulă o constituie funcțiile standard pentru gestiunea grafică a ecranului care implică modificadorul *far*.

## 20.9. Cuvântul cheie void

Cuvântul cheie **void** are diferite utilizări în funcție de context. O primă

utilizare a lui *void* este în legătură cu funcțiile care nu returnează o valoare la revenirea din ele:

```
void nume(...)
```

Tot în antetul unei funcții se poate utiliza cuvântul *void* între parantezele care urmează după numele funcției. În acest caz, *void* indică faptul că funcția nu are parametri.

Utilizarea lui *void* pentru a indica absența parametrilor nu este obligatorie. În schimb, este obligatoriu ca *void* să apară înaintea numelui funcției cind aceasta nu returnează o valoare. Aceleași reguli se aplică și la prototipul funcțiilor.

**Exemple:**

1. `void f(void);`

Funcția *f* nu are parametri și nu returnează o valoare.

2. `void g();`

Funcția *g* nu are parametri și nu returnează o valoare.

O altă utilizare a cuvântului cheie *void* este aceea de a defini un tip pointer universal:

```
void *
```

Un pointer spre *void* nu are tipul precizat.

Un pointer spre un tip precizat se poate atribui direct unui pointer spre *void*. Fie declarațiile:

```
tip *p;  
void *pv;
```

unde:

*tip* - Este diferit de *void*.

În acest caz, o atribuire de forma:

```
pv = p;
```

este corectă.

În schimb, atribuirea inversă, adică:

```
p = pv;
```

este eronată.

Astfel de atribuiriri se pot realiza numai folosind conversii explicite, ca mai jos:

```
p = (tip *)pv.
```

Nu se pot face operații cu pointeri spre *void*. Deci, expresiile de forma:

```
pv++
```

```
pv--
```

```
pv+n
```

sint eronate.

Pentru a putea face operații cu astfel de pointeri este necesar să se folosească conversii explicite:

```
(int *)pv +n
(double *)pv-n
etc.
```

În general, se vor evita astfel de operații deoarece ele pot conduce la erori.

## 20.10. Structuri

În limbajul C++ se pot defini tipuri noi folosind declarația *struct* ca în limbajul C (vezi capitolul 10).

De asemenea, se pot atribui nume la tipuri predefinite sau definite de utilizator folosind construcția *typedef*.

Pentru tipurile definite cu ajutorul lui *struct* putem defini date ca mai jos:

```
struct nume {
    ...
};

struct nume nume1, nume2, ...;
```

În limbajul C++ nu mai este necesară utilizarea lui *struct* la declararea structurilor *nume1*, *nume2*, ..., de tipul *nume*:

Deci, vom scrie mai simplu:

```
nume nume1, nume2, ...;
```

unde:

*nume* - Este tipul utilizator definit printr-o construcție *struct* ca mai sus.

În felul acesta, în limbajul C++, nu mai este necesară denumirea tipurilor structurate (definite cu ajutorul lui *struct*) prin intermediul construcției *typedef*.

În limbajul C++ o structură poate fi transferată direct prin parametri. Astfel, dacă avem declarația:

```
struct nume {
    ...
};
```

atunci datele de tip *nume* pot fi transferate prin parametri în 3 moduri:

- direct: tip f1(nume par) { ... }
- pointer spre structură: tip f2(nume \*par) { ... }
- referință la structură: tip f3(nume& par) { ... }

De asemenea, o funcție poate returna o structură, un pointer spre structură sau o referință la o structură:

- nume g1( ... ) { ... } - returnează o dată de tip nume;
- nume \*g2( ... ) { ... } - returnează un pointer spre o dată de tip nume;
- nume& g3( ... ) { ... } - returnează o referință la o dată de tip nume.

Structurile de același tip se pot atribui direct una alteia.

Fie:

```
nume a1, a2;
```

Atunci atribuirea:

```
a1 = a2
```

este legală.

## 20.11. Reuniune

În limbajul C++, față de limbajul C, reuniunile pot fi inițializate. Inițializarea se poate realiza numai pentru prima componentă a ei.

**Exemplu:**

```
union {
    double pi;
    int i;
    char c;
} u = {3.14159265358979};
```

Cuvântul *union* poate fi omis la declararea datelor de tip reuniune.

Astfel, fie declarația:

```
union alternative {
    int i;
    float f;
    double d;
    char c;
};
```

În continuare se pot declara, defini, date de tip *alternative* ca mai jos:

```
alternative zona1, zona2;
```

O altă facilitate existentă în C++ este posibilitatea de a utiliza reuniuni *anonyme* ca și componente într-o structură. Aceasta permite accesul mai simplu la componentele structurii.

**Exemplu:**

```
struct {
    int tip;
    union {
```

```

    int i;
    float f;
    double d;
};

}a;

```

Reuniunea, componentă a structurii *a*, este anonimă. La componentele ei ne referim ca mai jos:

```

a.i = 1234;
...
a.f = 3.14159;
...
a.d = 3.14159265;

```

## 20.12. Tipul enumerare

În limbajul C++, tipul enumerare se definește ca în limbajul C. Cu toate acestea există unele diferențe.

Diferența esențială constă în aceea că în limbajul C++ se fac teste asupra valorilor care se atribuie unei date de tip enumerare. De aceea, la o dată de tip enumerare se pot atribui numai enumeratoare (componentele) ei.

### Exemplu:

Fie declarațiile:

```

enum sapt {luni=1,marti,miercuri,joii,vineri,simbata,duminica};
enum sapt ziua;

```

Expresiile de mai jos sunt legale:

```

ziua = vineri;
if(ziua >= luni && ziua <= vineri)
// zi lucratoare
else
    // zi de odihnă

```

În C++, o expresie de forma:

```
ziua = 2
```

este eronată, deși conform declarației de mai sus, *marti* are valoarea 2.

Enumeratoare sint de tip int.

Ca și în cazul structurilor, cuvintul *enum* poate fi omis la declararea datelor de tip enumerare.

Astfel, *ziua* se poate declara mai simplu scriind:

```
sapt ziua;
```

## 20.13. Suprîncărcarea funcților

În limbajul C, funcțiile utilizate într-un program au nume *distințe*.

În limbajul C++ există posibilitatea ca funcții diferite să aibă un *același* nume.

De obicei, funcțiile "înrudite" pot fi denumite printr-un același nume.

Cazurile cele mai frecvente de funcții cu același nume sunt funcțiile care realizează același proces de calcul dar asupra unor date de tipuri diferite. De exemplu, în limbajul C există funcții, în biblioteca sistemului, cu nume distințe pentru calculul valorii absolute:

- |             |   |
|-------------|---|
| <i>abs</i>  | - Pentru calculul valorii absolute a unei date de tip <i>int</i> .    |
| <i>labs</i> | - Pentru calculul valorii absolute a unei date de tip <i>long</i> .   |
| <i>fabs</i> | - Pentru calculul valorii absolute a unei date de tip <i>double</i> . |

ACESTE FUNCȚII AU URMĂTOARELE PROTOTIPI:

```

int abs(int);
long labs(long);
double fabs(double);

```

Primele două prototipuri se află în fișierul *stdlib.h*, iar cel de al treilea în fișierul *math.h*.

În limbajul C++, aceste funcții pot fi numite cu un același nume.

În acest caz, ele se disting prin tipul parametrului. Astfel, în limbajul C++ este posibil să definim aceste 3 funcții ca mai jos:

1. int abs(int n) { return n < 0 ? -n : n; }
2. long abs(long n) { return n < 0L ? -n : n; }
3. double abs(double n) { return n < 0.0 ? -n : n; }

La un apel de forma:

*x = abs(expresie);*

compilatorul C++ apelează una din cele 3 funcții *abs* definite mai sus, îninind seama de tipul parametrului efectiv, adică de *expresie* dintre paranteze. Astfel, dacă *expresie* are tipul *int*, atunci se apelează funcția definită la punctul 1, dacă *expresie* are tipul *long*, atunci se apelează funcția definită la punctul 2, iar dacă *expresie* are tipul *double*, atunci se apelează funcția de la punctul 3. În felul acesta, numele *abs* reprezintă 3 funcții pe care compilatorul le poate distinge, la apel, după tipul parametrului efectiv.

Se obișnuiește să se spună că cele 3 funcții suprîncarcă numele *abs*.

De asemenea, se spune că numele *abs* este *suprîncărcat* (*overload*).

\* În general, un nume se spune că este *suprîncărcat* dacă el este un nume comun pentru mai multe funcții.

Despre funcțile cu același nume vom spune că sunt suprîncărcate (*functions overloading*).

La ora actuală există incetătenit în limba română și alți termeni pentru supraîncărcare, ca de exemplu, *suprapunere*, *redefinire* sau *supradefinire*.

În această carte adoptăm termenul de supraîncărcare.

O condiție esențială pentru supraîncărcarea funcțiilor este ca ele să aibă prototipuri *diferite*. În caz contrar, compilatorul nu poate face distincție, între ele, la apelul lor.

În principiu, numărul și tipul parametrilor efectivi furnizează un criteriu de selecție al funcțiilor supraîncărcate.

Având în vedere acest fapt, rezultă că la apelul funcțiilor supraîncărcate, de obicei, nu se mai aplică regula simplă de conversii a tipurilor parametrilor efectivi spre tipurile parametrilor formali corespunzători, regulă utilizată în limbajul C.

În acest caz, se pot aplica conversii dacă nu există coincidență între tipurile parametrilor efectivi și cei formali cu care se corespund.

În limbajul C++ se încearcă selectarea acelei funcții pentru care există coincidență între numărul și tipul parametrilor efectivi (de la apel) și al parametrilor formali ai funcției supraîncărcate. În cazul în care nu se poate stabili o astfel de coincidență, se fac și în acest caz conversii ale parametrilor efectivi. Aceste conversii pot să nu fie unice și de aceea sunt necesare reguli suplimentare.

De exemplu, reluind apelul de mai sus:

*x = abs(expresie);*

unde *expresie* are tipul *float*, se observă că nu se realizează coincidență cu tipul parametrului formal al nici uneia din funcțiile care supraîncarcă numele *abs*. De aceea, în acest caz, se va face o conversie a valorii expresiei de la apel. Sunt posibile 3 conversii:

```
float -> int
float -> long
float -> double
```

În principiu, se aleg conversii care să nu conducă la pierderea de informație. De aceea, în cazul de față, sunt excluse primele două conversii.

La un apel, selecția funcției se realizează în mai mulți pași și în ordinea de mai jos:

1. Se caută o corespondență exactă și se apeleză funcția respectivă dacă ea există.
2. Dacă nu se poate determina o funcție cu coincidență exactă, atunci se caută una pe baza efectuării de conversii predefinite (pentru tipuri predefinite) care nu conduc la pierderi de informație, adică se fac conversii fără trunchieri.
3. Dacă nu se poate determina o funcție conform punctelor precedente, atunci se încearcă selectarea unei funcții folosind conversii pentru tipuri predefinite care pot să conducă la trunchieri (de exemplu, date de tip *double* se convertesc în date de tip *float*, *long* sau *int*).
4. Dacă în pașii precedenți nu se poate determina funcția de apelat, atunci se

caută o funcție admitind conversii nu numai pentru tipuri predefinite ci și pentru cele definite de utilizator.

În cazul în care nu se poate selecta o funcție unică parcurgind în ordine pașii de mai sus, compilatorul C++ semnalează eroare.

Mai târziu, se va reveni asupra problemei apelurilor funcțiilor în legătură cu conversia datelor de tip utilizator.

#### Exerciții:

- 20.16 Să se supraîncarce numele *modul* cu funcții de calcul a valorii absolute pentru numere intregi și neîntregi, iar pentru numere complexe se va calcula modulul.

#### FIȘIERUL BXX16

```
int modul(int i) // returneaza valoarea absolută a lui i
{
    return i < 0 ? -i : i;
}

long modul(long n) // returneaza valoarea absolută a lui n
{
    return n < 0 ? -n : n;
}

double modul(double d) // returneaza valoarea absolută a lui d
{
    return d < 0 ? -d : d;
}

typedef struct {
    double x;
    double y;
} COMPLEX;

double modul(COMPLEX z) // returneaza modulul lui z
{
    return sqrt(z.x * z.x + z.y * z.y);
}

long double modul(long double ld) // returneaza valoarea absolută a lui ld
{
    return ld < 0 ? -ld : ld;
}
```

- 20.17 Să se defincească funcții supraîncărcate cu numele *afiseaza*, care să afișeze numere și siruri de caractere.

Funcțiile au unul sau mai mulți parametri. Primul parametru definește numărul de afișat. Parametrul al doilea, dacă este prezent, definește dimensiunea minimă a cimpului în care se afișează cadrată în dreapta și

eventual este precedată de spații.

Parametrul al treilea, dacă este prezent, indică numărul de zecimale în cazul numerelor neintregi sau numarul de caractere în cazul în care se afișează siruri de caractere.

## FIŞIERUL BXX17

```
char *format(const char *s1,int cimp);
char *format(const char *s1,const char *s2);

void afiseaza(int i,int cimp=5)
/* afiseaza valoarea lui i în cimpul de dimensiune minima egala cu cimp */
{
    char *sir,*sir1;

    sir1 = format("%",cimp);
    sir = format(sir1,"d");
    printf(sir,i);
    delete sir;
    delete sir1;
}

void afiseaza (long l,int cimp = 10)
/* afiseaza valoarea lui l în cimpul de dimensiune minima egala cu cimp */
{
    char *sir,*sir1;

    sir1 = format("%",cimp);
    sir = format(sir1,"ld");
    printf(sir,l);
    delete sir;
    delete sir1;
}

void afiseaza(double d,int cimp = 15,int precizie = 6)
/* afiseaza valoarea lui d în cimpul de dimensiune minima
   egala cu cimp și cu precizia definită de precizie */
{
    char *sir,*sir1,*sir2,*sir3;

    sir1 = format("%",cimp);
    sir2 = format(sir1,".");
    sir3 = format(sir2,precizie);
    sir = format(sir3,"g");
    printf(sir,d);
    delete sir;
    delete sir1;
    delete sir2;
    delete sir3;
}

void afiseaza(long double lf,int cimp = 15,int precizie = 6)
/* afiseaza valoarea lui lf în cimpul de dimensiune minima
```

egala cu cimp și cu precizia definită de precizie \*/

```
{ 
    char *sir,*sir1,*sir2,*sir3;

    sir1 = format("%",cimp);
    sir2 = format(sir1,".");
    sir3 = format(sir2,precizie);
    sir = format(sir3,"Lg");
    printf(sir,lf);
    delete sir;
    delete sir1;
    delete sir2;
    delete sir3;
}
```

```
void afiseaza(COMPLEX z,int cimp = 15,int precizie = 6)
/* afiseaza numarul complex z în cimpul cimp și cu precizia definită de precizie */
{
    char *sir;
    char *sir1,*sir2;
```

```
sir = format("%",cimp);
sir1 = format(sir,".");
delete sir;
sir = format(sir1,precizie);
delete sir1;
sir2 = format(sir,"g ");
delete sir;
sir1 = format("Partea reala = ",sir2);
sir = format(sir1,"Partea imaginara = ");
delete sir1;
sir1 = format(sir,sir2);
printf(sir1,z.x,z.y);
delete sir;
delete sir1;
delete sir2;
}
```

```
void afiseaza(const char *s,int cimp = 70,int precizie = 70)
/* afiseaza sirul de caractere spre care pointează s într-un cimp de
```

lungime minima egala cu valoarea lui cimp și cu precizia egala cu precizie \*/

```
{
    char *sir,*sir1,*sir2,*sir3;

    sir1 = format("%",cimp);
    sir2 = format(sir1,".");
    sir3 = format(sir2,precizie);
    sir = format(sir3,"s");
    printf(sir,s);
    delete sir;
    delete sir1;
    delete sir2;
    delete sir3;
}
```

```

char *format(const char *s,int n)
/*- construieste un sir prin concatenarea caracterelor spre care
   pointeaza s cu cele rezultate prin conversia lui n in decimal;
   - sirul rezultat se pustreaza in memoria heap.*/
{
    char zona[10];

    itoa(n,zona,10);
    return format(s,zona);
}

char *format(const char *s1,const char *s2)
/*construieste un sir prin concatenarea sirului spre care pointeaza s2 la
   sfirsitul sirului spre care pointeaza s1 */
{
    char *mem;

    mem = new char[strlen(s1)+strlen(s2)+1];
    strcpy(mem,s1);
    strcat(mem,s2);
    return mem;
}

```

20.18 Să se scrie un program care afișează modulele numerelor citite de la intrarea standard.

### PROGRAMUL BXX18

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <math.h>
#include "BXX16.CPP" // functiile supraincarcate modul
#include "BXX17.CPP" // functiile supraincarcate afiseaza

main()
/*afiseaza modulele numerelor citite de la intrarea standard*/
{
    int i;
    char c;
    char cont[] = "\n Actionati o tasta pentru a continua\n";
    char intreg_int[] = "\nint=";
    char t[255];

    for(;;) {
        for(;;) {
            afiseaza(intreg_int,strlen(intreg_int));
            if(gets(t)==0) exit(1);
            if(sscanf(t,"%d",&i)==1) break;
            afiseaza("nu s-a tastat un intreg",10);
            afiseaza("se reia citirea intregilor\n");
        }
        afiseaza("\n\n\t tip int\n",10);
    }
}

```

```

afiseaza(modul(i));
afiseaza(cont,50);
getch();
afiseaza("Alt int? Se raspunde cu D sau N ");
if((c = getch()) != 'D' && c != 'd') break;
} // sfirsit for pentru int

char intreg_long[] = "\nlong";
long l;

for(;;) {
    for(;;) {
        afiseaza(intreg_long,10);
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%ld",&l) == 1) break;
        afiseaza("nu s-a tastat un intreg\n",10);
        afiseaza("se reia citirea intregilor\n");
    }
    afiseaza("\n\n\t tip long\n",10);
    afiseaza(modul(l));
    afiseaza(cont,50);
    getch();
    afiseaza("Alt long ? Se raspunde cu D sau N ");
    if((c = getch()) != 'D' && c != 'd') break;
} // sfirsit for pentru long

char flotant_simplu[] = "\nflotant simpla precizie=";
char rel[] = "se reia citirea numarului\n";
float f;
char er[] = "nu s-a tastat un numar\n";

for(;;) {
    for(;;) {
        afiseaza(flotant_simplu,10);
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%f",&f) == 1) break;
        afiseaza(er,10);
        afiseaza(rel,10);
    }
    afiseaza("\n\n\t tip float \n",10);
    afiseaza(modul(f));
    afiseaza(cont,50);
    getch();
    afiseaza("Alt float ? Se raspunde cu D sau N ");
    if((c=getch()) != 'D' && c != 'd') break;
} // sfirsit for pentru float

char flotant_dublu[] = "\nflotant dubla precizie=";
double d;

for(;;) {
    for(;;) {
        afiseaza(flotant_dublu,10);
        if(gets(t) == 0) exit(1);
    }
}

```

```

        if(sscanf(t,"%lf",&d)==1) break;
        afiseaza(er,10);
        afiseaza(rel,10);
    }
    afiseaza("\n\n\t tip double\n\n",10);
    afiseaza(modul(d),25,16);
    afiseaza(cont,50);
    getch();
    afiseaza("Alt double ? Se raspunde cu D sau N ");
    if((c=getch()) != 'D' && c != 'd') break;
} //sfîrșit for pentru double

char flotant_lung[] = "\nfloatant lung=";
long double ld;

for(); {
    for(); {
        afiseaza(flotant_lung,10);
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%Lf",&ld) == 1) break;
        afiseaza(er,10);
        afiseaza(rel,10);
    }
    afiseaza("\n\n\t tip long double\n\n",10);
    afiseaza(modul(ld),30,20);
    afiseaza(cont,50);
    getch();
    afiseaza("Alt long double? Se raspunde cu D sau N ");
    if((c=getch()) != 'D' && c != 'd') break;
} //sfîrșit for pentru long double

char n_complex[] = "\ncomplex Partea reala=";
COMPLEX complex;

for(); {
    for(); {
        afiseaza(n_complex,10);
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%lf",&complex.x) != 1) {
            afiseaza(er,10);
            afiseaza(rel,10);
            continue;
        }
        afiseaza("\nPartea imaginara=",10);
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%lf",&complex.y) == 1) break;
        afiseaza(er,10);
        afiseaza(rel,10);
    }
    afiseaza("\n\n\t tip complex\n\n",10);
    afiseaza(modul(complex),25,16);
    afiseaza(cont,50);
    getch();
    afiseaza("Alt complex ? Se raspunde cu D sau N ");
}

```

```

        if((c = getch()) != 'D' && c != 'd') break;
    } // sfîrșit for pentru complex
}

```

### Observații:

1. Funcțiile supraîncărcate utilizate în acest program sunt:

- *afiseaza*;
- *format*;
- *modul*.

Ele se selectează prin coincidență între tipurile parametrilor efectivi și formalii care se corespund prin poziție, exceptând apelul funcției *modul* pentru tipul *float* cind se face o conversie spre *double*.

2. Un apel de forma:

```
afiseaza(sir);
```

unde *sir* este un pointer spre un sir de caractere, afișează sirul respectiv într-un cimp de minimum 70 de caractere. Aceasta deoarece parametrul implicit *cimp* al funcției *afiseaza* are valoarea 70. În cazul în care sirul care se afișează are mai puține caractere, el se afișează cadrat în dreapta în cimpul respectiv.

Dacă sirul conține mai mult de 70 de caractere, atunci numai primele 70 se vor afișa, deoarece parametrul implicit *precizie* are valoarea 70.

Un apel de forma:

```
afiseaza(&sir,10);
```

atribuie parametrului *cimp* valoarea 10. În acest caz, sirurile de caractere care conțin mai puțin de 10 caractere se afișează într-un cimp de 10 caractere și cadrat în dreapta.

Sirurile cu peste 10 caractere, dar care nu depășesc 70, se afișează pe atâtea caractere cîte intră în compunerea lor.

Parametrul implicit *precizie*, nefiind definit în apelul de mai sus, sirurile cu peste 70 de caractere se afișează trunchiat și anume numai primele 70 de caractere vor fi afișate.

- 20.19 Să se scrie funcții care supraîncarcă numele *putere* și care ridică la putere numere de diferite tipuri.

### FISIERUL BXX19

```

int putere(int a,int b)
// ridica pe a la puterea b
{
    int c;
    int i;

    if(a == 0) return 0;
    if(b < 0) return 0;
    for(i = 1,c = 1;i <= b;i++)

```

```

    c *= a;
    return c;
}

long putere(long a,int b)
// ridică pe a la puterea b
{
    long c;
    int i;

    if(a == 0) return 0;
    if(b < 0) return 0;
    for(i = 1,c = 1;i <= b;i++) c *= a;
    return c;
}

double putere(double a,int 'b)
// ridică pe a la puterea b
{
    double c;
    int i,j;

    if(a == 0) return 0.0;
    j = b < 0 ? -b : b; // j=abs(b)
    for(i = 1,c = 1.0; i <= j;i++) c *= a;
    return b < 0 ? 1.0/c : c;
}

double putere(double a,double b)
// ridică pe a la puterea b
{
    return pow(a,b);
}

```

20.20 Să se scrie un program care citește siruri de perechi de numere de forma  $(x,y)$  și afișează, pentru fiecare pereche, valoarea lui  $x$  la puterea  $y$ . Numerele  $x$  și  $y$  sunt de diferite tipuri.

## PROGRAMUL BXX20

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "BXX19.CPP"

main()
/* citeste perechi de numere de forma (a,b) si afiseaza pe a la b */
{
    char t[255];
    char baza[] = "baza\n";
    char exp[] = "exponent\n";
    char erint[] = "nu s-a tastat un intreg\n";
    int i,j;

```

```

    int c;

    for(;;) {
        for(;;) {
            printf(baza);
            printf("int=");
            if(gets(t) == 0) exit(1);
            if(sscanf(t,"%d",&i) == 1) break;
            printf(erint);
        }
        for(;;) {
            printf(exp);
            printf("int=");
            if(gets(t) == 0) exit(1);
            if(sscanf(t,"%d",&j) == 1) break;
            printf(erint);
        }
        int k = putere(i,j);
        printf("i = %d\t j = %d\t i**j = %d\n",i,j,k);
        printf("Alt int**int ? Se raspunde cu D sau N \n");
        if((c = getch()) != 'D' && c != 'd') break;
    } // Sfirsit int**int

    long u;

    for(;;) {
        for(;;) {
            printf(baza);
            printf("long=");
            if(gets(t) == 0) exit(1);
            if(sscanf(t,"%ld",&u) == 1) break;
            printf("nu s-a tastat un intreg de tip long\n");
        }
        for(;;) {
            printf(exp);
            printf("int=");
            if(gets(t) == 0) exit(1);
            if(sscanf(t,"%d",&j) == 1) break;
            printf(erint);
        }
        long m = putere(u,j);
        printf("u = %ld\t j = %d\t u**j = %ld\n",u,j,m);
        printf("Alt long**int ? Se raspunde cu D sau N \n");
        if((c = getch()) != 'D' && c != 'd') break;
    } // Sfirsit long**int

    float f;
    char ern[] = "Nu s-a tastat un numar\n";

    for(;;) {
        for(;;) {
            printf(baza);
            printf("float=");
            if(gets(t) == 0) exit(1);

```

```

        if(sscanf(t,"%f",&f) == 1) break;
        printf(ern);
    }
    for(;r) {
        printf(exp);
        printf("int=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%d",&j) == 1) break;
        printf(erint);
    }
    float q = putere(f,j);
    printf("f = %g\t j = %d\t f**j = %g\n",f,j,q);
    printf("Alt float**int ? Se raspunde cu D sau N \n");
    if((c = getch()) != 'D' && c != 'd') break;
} // Sfisit float**int

double d;

for(); {
    for(); {
        printf(baza);
        printf("double=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%lf",&d) == 1) break;
        printf(ern);
    }
    for(); {
        printf(exp);
        printf("int=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%d",&j) == 1) break;
        printf(erint);
    }
    double e = putere(d,j);
    printf("d = %g\t j = %d\t d**j = %g\n",d,j,e);
    printf("Alt double**int ? Se raspunde cu D sau N \n");
    if((c = getch()) != 'D' && c != 'd') break;
} // Sfisit double**int
double h;

for(); {
    for(); {
        printf(baza);
        printf("double=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%lf",&d) == 1) break;
        printf(ern);
    }
    for(); {
        printf(exp);
        printf("double=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%lf",&h) == 1) break;
        printf(ern);
    }
}

```

```

    }
    double g = putere(d,h);
    printf("d = %g\t h = %g\t d**h = %g\n",d,h,g);
    printf("Alt double**double ? Se raspunde cu D sau N \n");
    if((c = getch()) != 'D' && c != 'd') break;
} // Sfisit double**double

float fexp;
long double ld;

for(); {
    for(); {
        printf(baza);
        printf("long double=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%Lf",&ld) == 1) break;
        printf(ern);
    }
    for(); {
        printf(exp);
        printf("float=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%f",&fexp) == 1) break;
        printf(ern);
    }
    double p = putere(ld,fexp);
    printf("ld = %Lf\t fexp = %f\t \
           ld**fexp = %g\n",ld,fexp,p);
    printf("Alt long double**float ? Se raspunde \
           cu D sau N \n");
    if((c = getch()) != 'D' && c != 'd') break;
} // Sfisit long double**float

float q,r;

for(); {
    for(); {
        printf(baza);
        printf("float=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%f",&q) == 1) break;
        printf(ern);
    }
    for(); {
        printf(exp);
        printf("float=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%f",&r) == 1) break;
        printf(ern);
    }
    double s = putere(q,r);
    printf("q = %g\t r = %g\t q**r = %g\n",q,r,s);
    printf("Alt float**float ? Se raspunde cu D sau N \n");
    if((c = getch()) != 'D' && c != 'd') break;
}

```

```
) // Sfîrșit float**float  
}
```

### Observații:

- În acest program se utilizează funcția *putere* supraîncărcată cu următoarele tipuri predefinite:

Rezultat returnat	baza	exponentul
1. int	int	int
2. long	long	int
3. double	double	int
4. double	double	double

- Funcția *putere* se apelează cu parametri efectivi de diferite tipuri predefinite, după cum urmează:

- a. *putere(int,int)*
- b. *putere(long,int)*
- c. *putere(float,int)*
- d. *putere(double,int)*
- e. *putere(double,double)*
- f. *putere(long double,float)*
- g. *putere(float,float)*

Funcțiile *putere* supraîncărcate se selectează astfel:

La punctul Se selectează funcția de la punctul

- a 1: selectare prin coincidență;
- b 2: selectare prin coincidență;
- c 3: selectare prin conversie; baza de tip *float* se convertește spre tipul *double*;
- d 3: selectare prin coincidență;
- e 4: selectare prin coincidență;
- f 4: selectare prin conversie; baza de tip *long double* se convertește spre *double* și exponentul de tip *float* spre *double*;
- g 4: selectare prin conversia ambilor parametri din *float* spre *double*.

## 20.14. Funcții inline

În paragraful 15.1. s-a arătat posibilitatea de definire a macrouilor în limbajul C. Această facilitate este prezentă și în limbajul C++, dar utilizarea este diminuată prin prezența funcțiilor *inline*.

Amintim că un macro se definește printr-o construcție de forma:

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

Parametrii *p1*, *p2*, ..., *pn*, numiți parametri formali, apar în textul de substituție *text*.

Un macro poate fi apelat prin construcții de forma:

*nume(e1,e2, ... ,en)*

unde:

*e1,e2, ... ,en* - Sunt parametrii efectivi ai apelului.

Apelul macroului se înlocuiește cu *text* după ce, în prealabil, parametrii formalii *pi*, *i* = 1,2,...,n s-au înlocuit, în textul *text*, cu parametrii efectivi corespunzători *ei*, *i* = 1,2,...,n. Aceasta înlocuire a apelului macroului se numește *expandare*.

Expandarea macrouilor diferă de modul în care se realizează apelul funcțiilor obișnuite.

La apelul unei funcții obișnuite nu se substituie apelul funcției prin corpul ei, ci se realizează un salt la zona de memorie în care se păstrează corpul funcției respective.

La terminarea execuției funcției se revine în punctul imediat următor apelului. Un apel de această formă, numit și apel cu revenire, implică diferite operații suplimentare.

În cazul în care funcția este foarte simplă (2- 3 instrucțiuni), operațiile implicate de apel și revenire pot fi mai costisitoare decât cele implicate de funcția însăși. De aceea, în astfel de situații, ar fi util ca apelul funcțiilor să se realizeze la fel ca apelul de macro, adică prin expandare (inlocuirea apelului prin corpul funcției).

Acest lucru este posibil dacă antetul funcției este precedat de cuvintul cheie *inline*. O astfel de funcție se numește funcție *inline*.

Exemplu:

Pentru calculul maximului dintre două numere se poate defini macroul de mai jos:

```
#define MAX(x,y) (x) < (y) ? (y) : (x)
```

Fie secvența pentru apelul macroului MAX:

```
int a,b,c;  
c = MAX(a+b,a-b);
```

La procesare, se substituie apelul macroului prin textul său și se obține:

```
c = (a+b) < (a-b) ? (a-b) : (a+b);
```

Calculul maximului se poate realiza folosind funcția *inline* de mai jos:

```
inline int max(int x,int y) { return x < y ? y : x; }
```

Instrucțiunea de atribuire de mai sus se scrie:

```
c = max(a+b,a-b);
```

În acest caz, apelul funcției *max* se schimbă la compilare prin corpul ei. Se obișnuiește să se spună că apelul funcției se realizează prin expandarea ei. În felul acesta se elimină operațiile specifice de la apelul și revenirea din funcțiile obișnuite.

Funcțiile *inline* au unele avantaje față de macrouri. Astfel, la expandarea unui macro nu se ține seama de tipul parametrilor. De aceea, nu se realizează nici un control asupra tipurilor parametrilor efectivi sau conversii ale valorilor acestora spre tipurile parametrilor formali.

La apelul unui macro, parametrii formali se substituie în textul din corpul macroului prin parametrii efectivi corespunzători.

Această substituție se realizează înlocuind un sir de caractere (numele parametrului formal) printr-un alt sir de caractere care definește parametrul efectiv corespunzător.

De aici rezultă și un alt dezavantaj al macrourilor și anume acela al *efectelor secundare*, care în cazul macrourilor sunt mult mai greu de pus în evidență decât în cazul funcțiilor. Efectele secundare sau *colaterale* apar la calculul expresiilor care conțin operanți ale căror valori se schimbă la evaluarea lor. Astfel de efecte se întâlnesc mai ales la utilizarea, în expresii, a operatorilor de incrementare și decrementare.

De exemplu, atribuirea:

```
c = MAX(a++, b++);
```

se expandează astfel:

```
c = (a++) < (b++) ? (b++) : (a++);
```

Efectul acestei instrucțiuni este diferit de cel al instrucțiunii obținute apelând funcția *inline max* în locul macroului *MAX*:

```
c = max(a++, b++);
```

Acum apelul atribuie lui *c* maximul dintre *a* și *b*, funcția *max* nefiind influențată de incrementarea valorilor lui *a* și *b*.

Valoarea lui *c* în urma apelului macroului *MAX* este influențată de efectele secundare rezultate din incrementarea valorilor variabilelor *a* și *b*.

Mentionăm că efectele secundare pot fi prezente și în cazul funcțiilor, dar în acest caz ele pot fi controlate mai ușor.

De exemplu, apelul funcției *max* în instrucțiunea de mai jos:

```
c = max(b+a++, 2*a++);
```

atribuie lui *c* o valoare care poate să fie diferită pentru implementări diferite ale limbajului C++. Nu există o regulă pentru ordinea de evaluare a parametrilor efectivi și din această cauză nu putem ști dacă evaluarea expresiei:

```
2*a++
```

se realizează înainte sau după evaluarea expresiei *b+a++*.

De aceea, se recomandă evitarea apelurilor de acest tip prin introducerea unor atribuiri prealabile pentru parametri efectivi cu efecte secundare:

```
e1 = b+a++;
e2 = 2*a++;
```

```
c = max(e1, e2);
```

Având în vedere avantajele funcțiilor *inline* față de macrouri, rezultă interesul scăzut în utilizarea macrourilor în limbajul C++.

Construcția *#define* în limbajul C++ se utilizează frecvent în legătură cu compilarea condiționată, precum și în cazul funcțiilor de bibliotecă utilizate atât în programe C cât și în programe C++.

Funcțiile *inline* se vor utiliza totuși cu precauție, deoarece ele sunt avantajoase numai cind sunt simple. O funcție *inline* ce conține mai mult de 3 instrucțiuni poate deveni ineficientă deoarece expandarea ei în program în locul apelurilor sale poate conduce la o creștere substanțială a dimensiunii programului.

Funcțiile *inline* au anumite dezavantaje față de funcțiile obișnuite. Astfel, aceste funcții nu pot fi externe și deci ele pot fi definite și utilizate într-un singur modul al programului.

De asemenea, corpul unei funcții *inline* nu poate conține secvențe ciclice (secvențe introduse prin instrucțiunile *while*, *for* și *do-while*).

Atributul *inline* este neglijat de către compilator în cazul în care funcția nu poate fi tratată ca atare.

De exemplu, dacă o funcție cu atributul *inline* conține o instrucțiune ciclică, atunci atributul *inline* va fi neglijat și funcția respectivă va fi tratată în mod obișnuit. Compilatorul emite un mesaj de avertismen (*warnings*) corespunzător.

### Exerciții:

20.21 Să se scrie un program care citește un sir de perechi de numere întregi și afișează, pentru fiecare pereche, maximul dintre ele.

### PROGRAMUL BXX21

```
#include <stdio.h>

inline int max(int x, int y) /* returneaza maximul dintre x si y */
{
    return x < y ? y : x;
}

main()
/* citește un sir de perechi de intregi și afișează, pentru fiecare pereche, maximul dintre ele */
{
    int a,b;

    while(scanf("%d %d", &a,&b) == 2 )
        printf("a=%d\tb=%d\tmax(a,b)=%d\n", a,b, max (a,b));
}
```

20.22 Să se scrie un program care citește un sir de perechi de întregi și afișează, pentru fiecare pereche, maximul dintre valorile lor absolute.

## PROGRAMUL BXX22

```
#include <stdio.h>
inline int abs(int x) /* returneaza valoarea absoluta a lui x */
{
    return x< 0 ? -x: x;
}

inline int maxabs(int x, int y)
/* returneaza maximul valorilor absolute ale lui x si y */
{
    int z= abs(x);
    int u= abs(y);

    return z < u ? u : z;
}

main() /* citeste un sir de perechi de intregi si afiseaza, pentru fiecare pereche,
         maximul dintre valorile lor absolute */
{
    int a,b;

    while (scanf("%d %d",&a,&b)== 2 )
        printf("a=%d\ tb=%d\ tmaxabs(a,b)=%d\n",a,b,maxabs(a,b));
}
```

- 20.23 Să se scrie un program care citește un sir de perechi de intregi și afișează, pentru fiecare pereche, maximul dintre valorile lor absolute, apoi realizează același lucru pentru perechi de numere.

Pentru a termina sirul de perechi de numere intregi se va tasta un caracter oricare care nu intră în compunerea unui intreg (de exemplu o literă).

## PROGRAMUL BXX23

```
#include <stdio.h>

inline int abs(int x) /* returneaza valoarea absoluta a lui x */
{
    return x< 0 ? -x: x;
}

inline double abs(double x) /* returneaza valoarea absoluta a lui x */
{
    return x < 0.0 ? -x : x ;
}

inline int maxabs(int x, int y)
/* returneaza maximul valorilor absolute ale lui x si y */
{
    int z = abs(x); int u = abs(y);

    return z < u ? u : z;
}
```

```
inline double maxabs(double x, double y)
/* returneaza maximul valorilor absolute ale lui x si y */
{
    double z = abs(x);
    double u = abs(y);

    return z < u ? u : z ;
}

main()
/* - citeste un sir de perechi de intregi si afiseaza, pentru fiecare pereche,
   maximul dintre valorile lor absolute;
   - citeste un sir de perechi de numere si afiseaza, pentru fiecare pereche, maximul dintre
   valorile lor absolute. */
{
    int a,b;

    printf("sirul de perechi de numere intregi\n");
    while (scanf("%d %d",&a,&b)== 2 )
        printf("a=%d\ tb=%d\ tmaxabs(a,b)=%d\n",a,b,maxabs(a,b));

    double c,d;

    fflush(stdin); // videaza zona de intrare
    printf("sirul de perechi de numere\n");
    while(scanf("%lf %lf", &c,&d) == 2 )
        printf("c=%f\ td=%f\ tmaxabs(c,d)=%f\n",c,d,maxabs(c,d));
}
```

## 21. PROGRAMAREA PRIN ABSTRACTIZAREA DATELOR

Am văzut că limbajul C, și deci și limbajul C++, suportă stilurile de programare *procedurală* și programare *modulară*.

Programarea procedurală (vezi paragraful 4.14.) are la bază utilizarea procedurilor a căror echivalent în limbajele C și C++ sunt funcțiile.

Programarea procedurală este cel mai vechi stil de programare.

Mai tîrziu, pe măsură ce complexitatea programelor a crescut, a apărut ideea de a descompune problemele în subprobleme mai simple care la rîndul lor pot fi descompuse în altele mai simple și aşa mai departe. În felul acesta se ajunge la o descompunere arborescentă a problemei date în subprobleme mai simple. Programarea subproblemelor devine o problemă mai simplă și fiecare subproblemă are o anumită independentă față de celelalte subprobleme. De asemenea, interfața ei cu celelalte subprobleme este limitată și bine precizată prin procesul de descompunere a problemei inițiale. De obicei, programarea unei subprobleme, componentă a descompunerii arborescente a problemei inițiale, conduce la realizarea unui număr relativ mic de funcții.

Acstea funcții pot prelucra în comun anumite date. Unele dintre ele sunt independente de funcțiile realizate pentru alte subprobleme componente ale descompunerii arborescente. Altele realizează chiar interfață cu subproblemele învecinate.

Despre funcțiile obținute în urma programării unei subprobleme se obisnuiește să se spună că sunt *înrudite*. De obicei, aceste funcții, împreună cu datele pe care le prelucreză, se păstrează într-un fișier și se compilează *independent*.

O colecție de funcții înrudite, împreună cu datele pe care le prelucrază în comun formează un *modul*. În felul acesta, problema inițială se realizează într-un program alcătuit din module.

Programarea modulară are la bază elaborarea programelor pe module.

O parte din datele utilizate în comun de funcțiile modulului, sau chiar toate datele modulului, nu sunt necesare și în alte module. Aceste date pot fi *protectate* sau cum se mai spune, *ascunse în modul*.

Limbajul C și deci și C++, permite ascunderea datelor în modul folosind date care au clasa de memorie *static* (vezi paragraful 5.2.).

Mai mult decît atât, pot fi declarate și funcțiile ca statice și atunci ele vor fi ascunse în modul (nu pot fi apelate din afara modulului). Ascunderea funcțiilor în modul se face pentru acele funcții care nu se utilizează la realizarea interfeței modulului cu celelalte module.

Ascunderea datelor și funcțiilor în module permite protejarea datelor și preîmpină utilizarea eronată a funcțiilor.

În capitolul 7 se dă un exemplu de modul prin care se implementează o stivă pentru numere întregi.

În capitolul respectiv, stiva este implementată printr-un *tablou* cu ajutorul declarației:

```
static int stack[MAX];
```

unde *MAX* este o constantă simbolică definită în prealabil și care stabilește dimensiunea maximă a stivei. Numerele de tip *int* puse pe stivă sunt păstrate ca elemente ale tabloului *stack* și ele ocupă zona:

```
stack[0], stack[1], ... , stack[next-1]
```

Elementul *stack[0]* se află la baza stivei, iar *stack[next-1]* se află în virful ei.

Variabila *next* definește atât poziția virfului stivei (elementul *stack[next-1]*), cit și elementul liber al tabloului de cel mai mic indice.

Înîțial *next* are valoarea zero, deoarece stiva fiind vidă, *stack[0]* este elementul liber de cel mai mic indice. Variabila *next* se declară astfel:

```
static int next = 0;
```

În felul acesta, cele două date (*stack* și *next*) utilizate pentru a implementa o stivă pentru gestionarea numerelor de tip *int* sunt ascunse în modul, avind clasa de memorie *static*.

Gestiunea lor se face prin funcții globale accesibile din orice modul al programului. Astfel, în exercițiul 7.1. s-au definit funcțiile:

- *push* - Pune în virful stivei un număr de tip *int*.
- *pop* - Scoate numărul de tip *int* din virful stivei.
- *top* - Permite acces la numărul de tip *int* aflat în virful stivei.
- *clear* - Videază stiva.
- *empty* - Returnează valoarea 1 dacă stiva este vidă și zero în caz contrar.
- *full* - Returnează valoarea 1 dacă stiva este plină și zero în caz contrar.

Datele *stack* și *next* fiind ascunse în modul, utilizatorul nu are acces direct la ele din alte module. De aceea, ele nu pot fi deteriorate accidental. Utilizatorul are acces numai la elementul din virful stivei. Acesta poate fi scos din stivă (funcția *pop*) sau poate fi folosit lăsindu-l pe stivă (funcția *top*). De asemenea, utilizatorul poate pune un număr pe stivă numai în poziția următoare celei corespunzătoare elementului din virful ei (funcția *push*).

Menționăm că stiva implementată în acest fel nu este chiar o stivă *veritabilă*, deoarece stiva, prin natura ei, este o structură dinamică. În cazul de față, este rezervată o zonă fixă pentru stivă deoarece tablourile din limbajul C nu sunt dinamice.

În capitolul 11 se dă o implementare pentru stive cu ajutorul listelor simplu

înlănțuite. Acestea sint structuri de date dinamice ceea ce permite ca și stivele implementate cu ajutorul lor să fie de natură dinamică.

În acest caz, un element al stivei este o dată de un tip definit de utilizator și anume:

```
typedef struct tnod {  
    declaratii  
    struct tnod *urm;  
} TNOD;
```

În același capitol, pentru gestiunea listelor simplu înlănțuite s-au folosit variabilele globale *prim* și *ultim* care se definesc ca fiind pointeri spre tipul *TNOD*.

La implementarea stivelor printr-o listă simplu înlănțuită, operațiile *push* și *pop* se definesc cu ajutorul funcțiilor *iniprim* și *spn* (vezi paragraful 11.1.3.1. și respectiv 11.1.4.1.).

Vidarea stivei (operația *clear*) se realizează folosind funcția *sterglis*t definită în 11.1.5.

În mod analog, se pot defini simplu și alte funcții pentru a realiza accesul la elementul din virful stivei (corespondentul pentru *top*) sau a stabili starea stivei (vidă sau nu-corespondentul pentru *empty*) etc.

Funcțiile de gestiune a stivei pot fi rescrise în aşa fel incit să nu fie nevoie decit de variabila *prim*.

Se poate realiza și în acest caz un modul cu ajutorul căruia să se ascundă datele de implementare ale stivei. În acest scop nu avem decit să declarăm variabila *prim* ca statică:

```
static TNOD *prim;
```

Înțial, variabila *prim* are valoarea zero (pointerul nul). Aceasta deoarece inițial stiva este vidă.

Stivele implementate ca mai sus nu sint convenabile dacă într-un program este nevoie să se folosească, în același timp, mai multe stive pentru date de același tip.

Într-un program se pot defini și utiliza oricite date pentru tipuri predefinite. De exemplu, putem defini date de tip *int* prin declarații de forma:

```
int nume_1,nume_2, ... ,nume_n;
```

Cu astfel de date se pot realiza diferite operații predefinite, cum sint cele patru operații, operații de comparație, operații logice etc.

În cazul tipurilor definite de utilizator se definește modul de reprezentare al datelor, dar nu și operațiile care se realizează asupra datelor respective. De aici rezultă simplitatea în prelucrarea datelor de tipuri predefinite în comparație cu tratarea datelor ale căror tipuri sint definite de utilizator.

Pentru a putea utiliza simplu, într-un program, mai multe stive pentru date de un același tip, ar fi necesar să putem defini tipul *stivă*, înțelegind prin aceasta că se definesc atât reprezentarea datelor de tip *stivă* cit și operațiile admise asupra

datelor de tip *stivă*.

Tipul utilizator folosit în limbajul C nu stabilește nici o legătură între reprezentarea datelor de un anumit tip și operațiile executabile asupra datelor de tipul respectiv.

De exemplu, în paragraful 10.3., se definește tipul utilizator *COMPLEX* ca mai jos:

```
typedef struct {  
    double real;  
    double imag;  
} COMPLEX;
```

Prin această declarație se definește reprezentarea datelor de tip *COMPLEX* și anume, o astfel de dată este o pereche ordonată de numere, fiecare de tip *double*.

Se pot declara date de tip *COMPLEX* ca și date de tip *int*, *long* sau orice alt tip predefinit. De exemplu, folosind declarația:

```
COMPLEX z;
```

înseamnă că *z* este o dată care se compune din două numere de tip *double*. Această declarație este analogă cu declarația:

```
int i;
```

prin care se stabilește că *i* este o dată de tip *int*. Cum tipul *int* are predefinite și operațiile asupra datelor de acest tip, rezultă că declarația de mai sus, precizează atât reprezentarea lui *i* (reprezentarea în binar prin complement față de 2 pe 16 biți) cit și operațiile posibile asupra lui *i*. De exemplu, expresiile de mai jos sint corecte:

```
i+10  
i/123  
i++  
etc.
```

În cazul datei *z*, este precizată numai reprezentarea lui *z* și accesul la componentele sale:

<i>z.real</i>	- Acces la prima componentă a lui <i>z</i> .
<i>z.imag</i>	- Acces la a doua componentă a lui <i>z</i> .

La definirea tipului *COMPLEX* nu se precizează nimic asupra operațiilor admisibile pentru datele de acest tip.

Utilizatorul poate defini astfel de operații prin intermediul funcțiilor. De exemplu, pentru adunarea numerelor complexe se poate defini funcția:

```
void adcomplex(COMPLEX *a, COMPLEX *b, COMPLEX *c)  
{  
    c -> real = a -> real + b -> real;  
    c -> imag = a -> imag + b -> imag;  
}
```

În felul acesta, dacă *u,v,w* sint declarate ca mai jos:

```
COMPLEX u, v, w;
```

atunci apelul:

```
adcomplex(&u, &v, &w);
```

realizează suma dintre numerele complexe *u* și *v*, iar *w* devine egal cu suma respectivă, ceea ce în notație matematică se scrie astfel:

```
w = u+v
```

În mod analog, prin intermediul funcțiilor, se pot realiza și alte operații cu numere complexe (vezi exercițiile 10.1, 10.2, 10.12, 10.13, 10.14, 10.15, 10.16, 10.17).

Prin realizarea acestor funcții nu se precizează că ele definiște operațiile asupra datelor de tip *COMPLEX* și că alte operații sunt interzise.

Un prim pas, în încercarea de a aprobia modul de tratare al tipurilor utilizator de cel al tipurilor predefinite, a fost acela de a preciza, la definirea tipului utilizator, nu numai reprezentarea datelor tipului respectiv, ci și funcțiile care definiște operații cu datele de tipul respectiv.

Acest lucru s-a obținut simplu prin enumerarea prototipurilor funcțiilor respective împreună cu definirea reprezentării datelor.

De exemplu, pentru tipul *complex* putem enumera funcțiile care realizează operațiile cu datele de tip *complex*, ca mai jos:

```
struct complex {           // reprezentarea tipului
    double real;
    double imag;

    // funcțiile prin care se realizează operațiile asupra numerelor complexe
    double modul();        // modulul numărului complex
    double arg();          // argumentul numărului complex
    void neg();            // real = -real și imag = -imag

    // real = a.real+b.real; imag = a.imag+b.imag
    void adcomplex(complex a, complex b);

    // real = a.real-b.real; imag = a.imag-b.imag
    void sccomplex(complex a, complex b);

}; // sfârșit struct complex
```

\* Funcțiile enumerate ca mai sus, în declarația *complex*, se numesc funcții *membru*.

Prin aceasta nu se realizează facilități deosebite deoarece enumerarea funcțiilor membru indică, pentru programator, doar faptul că funcțiile respective sunt strins legate de datele componente ale structurii respective.

Programatorul poate defini funcții suplimentare care să realizeze și alte

operații asupra datelor structurii.

De exemplu, funcția de prototip:

```
int pcficomplex(complex *a);
```

care citește partea reală și cea imaginată a numărului *complex*, spre care pointează *a*, poate fi definită și utilizată pentru numere de tip *complex*, deși ea nu a fost enumerată în declarația *struct* de mai sus.

Datele de tip *complex* se definesc în mod obișnuit:

```
complex z;
```

Aceasta înseamnă că *z* este o dată care are două componente:

*real*

și

*imag*

de tip *double*.

Funcțiile membru care sunt foarte simple pot fi definite în cadrul declarației *struct* înlocuind prototipul prin *antetul* și *corpul* funcției respective.

De exemplu, prototipul:

```
double modul();
```

din definiția tipului *complex* de mai sus, poate fi înlocuit cu definiția funcției *modul*:

```
double modul() { return sqrt(real*real+imag*imag); }
```

Funcțiile membru, definite în interiorul declarațiilor *struct*, se realizează în mod implicit prin expandare *inline* (vezi 20.14). De aceea, ele nu vor fi definite în interiorul declarației *struct* decât dacă sunt foarte simple.

Funcțiile membru, definite în afara declarației *struct*, se pot expanda *inline* dacă se declară explicit acest lucru. Deci, funcția *modul* de mai sus, poate fi definită *inline* în afara declarației *struct* dacă se declară explicit ca fiind o funcție *inline*.

Funcția membru *arg* este destul de complexă (are peste 3 instrucțiuni) și de aceea ea nu va fi expandată *inline*. Deci, pentru ea se indică prototipul în declarația tipului *complex* și se definește în afara declarației respective ca o funcție obișnuită.

O funcție membru care se definește în afara declarației tipului a cărui membru este, se califică cu numele tipului prin intermediul operatorului de rezoluție`(::)`. Astfel, numele funcției din antet se înlocuiește cu:

*nume\_tip::nume\_funcție\_membru*

De exemplu, funcția membru *arg* a tipului *complex*, definit mai sus, are antetul:

```
double complex::arg() { ... }
```

Această convenție se aplică și pentru funcțiile *inline* care sunt definite în afara definiției tipului pentru care ele sunt funcții membru. De exemplu, funcția *modul* se poate defini ca funcție *inline* în afara definiției tipului *complex* astfel:

```
inline complex::modul() { return sqrt(real*real+imag*imag); }
```

Accesul la componente date ale unui tip, numite și *date membru* ale tipului respectiv, se realizează în mod obișnuit folosind operatorii punct (.) și săgeată (->).

Fie, de exemplu, declarațiile:

```
complex z;
complex *p;
```

La datele membru ale lui *z* ne referim prin nume calificate:

```
z.real
și
z.imag.
```

În mod analog, la datele membru ale numărului complex spre care pointează *p* ne referim prin:

```
p->real
și
p->imag.
```

În continuare, ne interesează apelul funcțiilor membru. O funcție membru se poate apela numai pentru o dată de tipul pentru care funcția respectivă este membru.

La apelul unei astfel de funcții se folosesc aceeași operatori ca în cazul datelor membru, adică punctul și săgeata.

Astfel, pentru calculul modulului numărului *z*, declarat ca mai sus, se va utiliza apelul:

```
z.modul()
```

În mod analog, pentru a calcula modulul numărului complex spre care pointează *p* se va utiliza apelul:

```
p->modul()
```

Din aceste apeluri se vede că în cazul funcțiilor membru nu este nevoie de un parametru pentru a avea acces la data curentă prelucrată de funcția apelată.

În cazul funcțiilor obișnuite, funcția are acces la data respectivă dacă aceasta se transferă printr-un parametru sau este o dată globală.

De exemplu, funcția *dmodul* pentru calculul modulului unui număr de tip *COMPLEX* a fost definită în exercițiul 10.1, ca mai jos:

```
double dmodul(COMPLEX *z)
{
    return sqrt(z->x*z->x+z->y*z->y);
}
```

unde tipul *COMPLEX* se definește astfel:

```
typedef struct {
    double x;
    double y;
} COMPLEX;
```

Tipul *COMPLEX* definit în acest fel, nu are funcții membru.

Dacă se folosește declarația:

```
COMPLEX a;
```

atunci pentru calculul modulului numărului complex *a* se va utiliza apelul:

```
dmodul(&a)
```

Parametrul formal *z* al funcției *dmodul* este un pointer spre data de tip *COMPLEX* pentru care se calculează modulul. El se utilizează în corpul funcției pentru a avea acces la componentele parametrului de la apel.

În cazul apelului unei funcții membru există în mod implicit un astfel de parametru. Numele lui este *this*.

Ei are declarația implicită:

```
tip *const this;
```

unde:

*tip* - Este tipul pentru care funcția apelată este o funcție membru.

Valoarea lui *this* se definește la fiecare apel al funcției membru astfel:

- dacă se folosește apelul:

```
nume_data.nume_functie(...)
```

atunci *this* are ca valoare adresa datei *nume\_data*;

- dacă se folosește apelul:

```
pointer->nume_functie(...)
```

atunci *this* are aceeași valoare ca și *pointer*.

Pointerul *this* se aplică în mod implicit la componente datei.

Astfel, dacă considerăm apelul:

```
z.modul()
```

atunci *this* are valoarea implicită adresa lui *z*, adică &*z*.

În cazul funcției *modul*, *this* se utilizează în mod implicit. De aceea expresia:

```
real*real+imag*imag
```

în mod implicit se aplică la componente datei spre care pointează *this*, adică expresia respectivă se interpretează astfel:

```
this->real*this->real+this->imag*this->imag
```

Se observă că utilizarea implicită a pointerului *this* simplifică mult definiția funcțiilor membru.

Pointerul *this* poate fi utilizat în mod explicit, de către programator, în funcțiile membru, dacă este nevoie să se facă referire la data spre care pointează *this*.

Funcțiile membru pot fi enumerate și pentru tipurile utilizator introduse prin *union*. Acestea au aceeași facilități ca și în cazul tipurilor definite cu *struct*.

Din cele de mai sus, rezultă că tipurile de date prezintă două aspecte: unul legat de *reprezentarea datelor*, ceea ce înseamnă că tipul este o mulțime de date care au aceeași reprezentare, iar celălalt aspect este legat de *operațiile* care se definesc asupra mulțimii respective.

Tipurile de date definite în acest fel se numesc *tipuri abstracte de date*.

Tipurile abstracte de date au ca și componente atât date cât și funcții. De aceea, tipurile abstracte pot fi considerate ca o generalizare a tipului utilizator la definiția căruia se au în vedere numai *componentele dată*.

Un tip este considerat de B. Stroustrup ca o reprezentare concretă a unui concept. De exemplu, tipul *float* din limbajele C sau C++ definește o submulțime a mulțimii numerelor *rationale* asupra cărora sunt definite diferite operații ca: adunare, scădere, înmulțire etc.

De obicei, orice problemă utilizează atât concepte cărora le corespund tipuri predefinite în limbajele de programare cât și concepte cărora nu le corespund astfel de tipuri. Un exemplu simplu este *conceptul de stivă* pentru care, de obicei, nu există un tip predefinit în limbajele de programare.

Pentru astfel de concepte, utilizatorul urmează să definească tipuri noi. Acest lucru se poate realiza numai în parte în limbajul C cu ajutorul tipurilor definite de utilizator, deoarece nu se precizează operațiile cu astfel de tipuri, ci numai *reprezentarea lor*.

Așa cum s-a arătat mai sus, tipul definit de utilizator poate fi extins în limbajul C++ așa încit odată cu definiția reprezentării datelor unui tip să se enumere, sau chiar definească, funcțiile cu ajutorul cărora se realizează operațiile cu datele respective. Cu toate acestea, tipurile definite în acest fel nu reprezintă suficient de bine conceptul pentru care s-au definit.

Principalul neajuns rezultă din faptul că tipurile introduse în acest fel nu definesc o legătură strinsă între datele membru și funcțiile membru. Astfel, datele unui astfel de tip pot fi accesate și prelucrate și de alte funcții decât cele precizate la definiția tipului prin *struct* sau *union*. Aceasta înseamnă că la compilare nu se pot stabili controale în legătură cu utilizarea datelor de tipul respectiv în diferite funcții. De aceea, utilizarea eronată a datelor, de astfel de tipuri, de obicei nu poate fi pusă în evidență prin compilare. Se obișnuiește să se spună că datele de acest fel *nu sunt protejate*.

Amintim că programarea modulară permite o anumită protecție a datelor prin ascunderea lor în modul.

Acastă idee a condus la noțiunea de *clasă* în limbajul C++. De data aceasta

se încearcă ascunderea datelor la definiția tipului abstract. Astfel, datele membru (o parte sau în totalitate) pot fi protejate interzicind accesul la ele prin funcții care nu sunt funcții membru ale tipului respectiv.

De asemenea, o parte din funcțiile membru pot fi și ele protejate, neputind fi apelate decât numai de către funcții membru ale tipului respectiv.

Protecțiile de acest fel se realizează cu ajutorul claselor și ele se definesc folosind *modificator de protecție*. Aceștia sunt trei: *private*, *protected* și *public*. \*

Modificatorul *public* aplicat la o dată sau funcție membru face ca, componenta respectivă să nu fie protejată și ea poate fi utilizată fără restricții în program. Primii doi modificatori (*private* și *protected*) asigură protecția datei sau funcției membru. Efectul fiecărui, din acești modificatori, va fi explicat mai târziu. În momentul de față amintim că modificatorii se scriu urmăți de două puncte (

- *private*:
- *public*:
- *protected*:

O *clăsă* se definește printr-o declarație de clăsă. Aceasta are același format ca și declarația *struct*, schimbând cuvintul *struct* cu *class*. Într-o astfel de definiție, modificatorul *private* este implicit. Tipul *complex*, definit mai sus prin *struct*, îl definim printr-o clăsă astfel:

```
class complex {  
    double real;  
    double imag;  
public:  
    double modul()  
    {  
        return sqrt(real*real+imag*imag);  
    }  
    double arg();  
    void adcomplex(complex a,complex b);  
    void sccomplex(complex a,complex b);  
};
```

Datele membru *real* și *imag* sunt protejate prin modificatorul implicit *private*. Aceasta înseamnă că la ele nu au acces decât funcțiile membru ale clasei, adică funcțiile *modul*, *arg*, *adcomplex* și *sccomplex*.

De exemplu, nu este posibil să atribuim valori componentelor *real* și *imag*, să afișăm valorile lor etc.

În acest caz se realizează protecția datelor respective. Deci, dacă vom defini o dată de tipul *complex*:

```
complex z;  
atribuirile:  
z.real = 1;  
z.imag = -1;
```

sunt eronate și vor fi semnalate la compilare.

Functiile membru ale clasei *complex* sunt toate publice (un modificador de protecție este valabil din punctul aparitiei lui pînă la sfîrșitul definiției clasei sau pînă în punctul aparitiei unui alt modificador de protecție).

Aceasta înseamnă că ele se pot folosi în tot programul pentru date de tip complex.

Tipurile definite prin *struct* și *union* au toate componentele *publice*. De aceea, despre *struct*, se obișnuiește să se spună că este o *clasa* cu *toate* componentele *publice*.

Clasele asigură protecția datelor și a funcțiilor membru la un nivel superior față de module. Ele permit definirea simplă de tipuri noi, care se numesc *tipuri abstracte de date*. Tipurile introduse cu ajutorul claselor se apropie de cele predefinite și de aceea se pot defini și utiliza simplu date de tipuri abstracte în comparație cu facilitățile oferite prin module.

Așa cum s-a arătat mai sus, modulele nu oferă o soluție simplă cînd utilizăm în program, în același timp, mai multe date de tipul celor ascunse într-un modul.

Clasa oferă o soluție simplă pentru astfel de situații aşa cum se va vedea în continuare și aceasta se realizează pe baza apropierii modului de utilizare al tipurilor abstracte de a celor predefinite.

B. Stroustrup afirmă că, în principiu, clasele permit să se introducă tipuri noi de date pentru care accesul este limitat printr-un set de funcții bine precizat.

Programarea bazată pe utilizarea tipurilor abstracte de date se numește *programare prin abstractizarea datelor*. Ea este superioară programării modulare asigurînd protecția datelor la un nivel calitativ superior care rezultă din facilitățile existente în definirea și utilizarea tipurilor abstracte de date.

Întrucît tipurile abstracte de date se definesc în limbajul C++ prin intermediul claselor, rezultă că la baza programării prin abstractizarea datelor se află noțiunea de clasă.

În principiu, programatorul trebuie să stabilească clar conceptele de care are nevoie pentru rezolvarea problemei concrete și apoi conceptele respective se realizează prin intermediul claselor.

La stabilirea și implementarea conceptelor necesare rezolvării unei probleme se realizează un proces de abstractizare care privește reprezentarea datelor și prelucrarea acestora. Aceasta justifică termenul de *programare prin abstractizarea datelor*.

Ca orice proces de abstractizare, și cel legat de abstractizarea datelor presupune un proces de generalizare care se realizează prin neglijarea aspectelor particulare.

Concepțele realizate prin clase au o parte *protejată* și una *neprotejată*. Partea protejată exprimă partea specifică, proprie conceptului exprimat prin clasa respectivă și ea are rolul de a asigura *implementarea* clasei (conceptului). Partea neprotejată, numită și *publică*, reprezintă *interfața* clasei (conceptului) cu celelalte clase (concepțe) ale programului. Implementarea clasei este neglijată în

relația (utilizarea) clasei cu celelalte clase din program. În felul acesta, implementarea unei clase poate să schimbe fară a schimba utilizarea ei în restul programului.

### Exerciții:

21.1 Se consideră tipul *complex*, definit ca mai jos, cu ajutorul lui *struct*. Se cere să se definească funcțiile membru ale tipului respectiv. Funcția *arg* se definește la fel ca în exercițiul 10.2.

### FIșIERUL BXXI1

```
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif

struct complex {
    double real;
    double imag;

    double modul() // returneaza modulul numarului complex
    {
        return sqrt(real*real + imag*imag);
    }

    double arg(); // returneaza argumentul numarului complex

    void adcomplex(complex *a,complex *b) // calculeaza a+b
    {
        real = a -> real + b -> real;
        imag = a -> imag + b -> imag;
    }

    void sccomplex(complex *a,complex *b) // calculeaza a-b
    {
        real = a -> real - b -> real;
        imag = a -> imag - b -> imag;
    }

    void mulcomplex(complex *a,complex *b) // calculeaza a*b
    {
        real = a -> real * b -> real - a -> imag * b -> imag;
        imag = a -> real * b -> imag + a -> imag * b -> real;
    }

    int divcomplex(complex *a,complex *b); // calculeaza a/b
};

#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif
```

```

double complex :: arg() // returnaza argumentul numarului complex
{
    double a;

    if(real == 0 && imag == 0) return 0.0;
    if(imag == 0)
        if(real > 0) return 0.0;
        else // imag=0 si real<0
            return PI;
    if(real == 0)
        if(imag > 0)
            return PI/2;
        else // real=0 si imag<0
            return (3*PI)/2;

    // real!=0 si imag!=0
    a = atan(imag/real);
    if(real < 0) // real<0 si y!=0
        return a+PI;
    else // real>0
        if(imag < 0) // real>0 si imag<0
            return 2*PI + a;

    // real>0 si imag>0
    return a;
} // sfarsit arg

int complex :: divcomplex (complex *a,complex *b)
// calculeaza a/b; returnaza 0 la impartirea cu zero
{
    double d;
    d = b -> real * b -> real + b -> imag * b -> imag;

    if(d == 0.0) return 0;
    real = ( a -> real * b -> real + a -> imag * b -> imag)/d;
    imag = ( a -> imag * b -> real - a -> real * b -> imag)/d;
    return 1;
}

```

21.2 Să se scrie un program care citește trei numere complexe  $a, b, c$  care sunt coeficienții ecuației:

$$a*x^2+b*x+c=0$$

rezolvă și afișează rădăcinile ecuației respective.

## PROGRAMUL BXXI2

```

#include <stdio.h>
#include <stdlib.h>
#include "BXXI1.CPP"
#include "BX18.CPP" // citeste un numar de tip double

```

```

int citcomplex(complex *a)
/* - citeste partea reală și cea imaginată a unui număr complex;
   - returnează:
     0 - la sfârșit de fisier;
     1 - altfel.
*/
{
    if(pcit_double("partea reală: ", &a -> real) == 0) return 0;
    if(pcit_double("partea imaginată: ", &a -> imag) == 0)
        return 0;
    return 1;
}

main()
/* - citeste pe a,b,c - numere complexe; rezolvă și
   afișează rădăcinile ecuației:
    $a*x^2+b*x+c=0$  */
{
    complex a,b,c,x1,x2,bp,temp,aa,temp1;
    char er[] = "s-a tastat EOF\n";
    double r,argument;
    complex patru = {4.0,0.0};

    // se citesc coeficienții
    if(citcomplex(&a) == 0){
        printf(er);
        exit(1);
    }
    if(citcomplex(&b) == 0){
        printf(er);
        exit(1);
    }
    if(citcomplex(&c) == 0){
        printf(er);
        exit(1);
    }
    if(a.real == 0 && a.imag == 0 && b.real == 0 &&
       b.imag == 0 && c.real == 0 && c.imag == 0) {
        // a=b=c=0
        printf("ecuație nedeterminată\n");
        exit(0);
    }
    if(a.real == 0 && a.imag == 0 && b.real == 0 && b.imag == 0 )
    {
        printf("ecuația nu are soluție\n");
        exit(1);
    }
    if(a.real == 0 && a.imag == 0){
        printf("ecuație de gradul întâi\n");
        x1.real = -c.real; x1.imag = -c.imag;
        x2.divcomplex(&x1,&b);
        printf("x=%g+i*(%g)\n",x2.real,x2.imag);
        exit(0);
    }
}

```

```

// temp = b*b - 4*a*c
bp.mulcomplex(&b, &b);
temp.mulcomplex(&patru, &a);
temp1.mulcomplex(&temp, &c);
temp.sccomplex(&bp, &temp1);

/* temp = real + i * imag
   r = modul(temp);
   argument = arg(temp)
*/
r = temp.modul();
argument = temp.arg();

/* temp = r*(cos(argument)+i*sin(argument))
   sqrt(temp) = sqrt(r)*(cos(argument/2)+i*sin(argument/2)) */
r = sqrt(r);
argument = argument/2;

// temp = sqrt(b*b-4*a*c)
temp.real = r*cos(argument);
temp.imag = r*sin(argument);

// aa = a + a
aa.adcomplex(&a, &a);

// x1 = (-b+temp)/aa
bp.real = -b.real;
bp.imag = -b.imag; // bp = -b

// temp1 = -b+temp
temp1.adcomplex(&bp, &temp);
x1.divcomplex(&temp1, &aa);

// x2 = (-b-temp)/aa
temp1.sccomplex(&bp, &temp);
x2.divcomplex(&temp1, &aa);
printf("x1 = %g + i(%g)\n", x1.real, x1.imag);
printf("x2 = %g+i(%g)\n", x2.real, x2.imag);
}

```

## 22. CLASE

La programarea problemelor complexe intervin concepte noi care nu pot fi exprimate simplu prin tipuri predefinite de date.

De obicei, orice limbaj de programare pune la dispoziția programatorului un număr de tipuri predefinite, care însă, în mod frecvent, nu corespund tuturor conceptelor necesare programului. Astfel de concepte se implementează în limbajul C++ prin intermediul claselor.

\* O clasă definește un tip abstract de date.

\* Prin tip abstract de date înțelegem o mulțime de date care au o aceeași reprezentare și pentru care este definit setul de operații care se pot executa asupra elementelor mulțimii respective. Ca exemple de concepte care nu corespund unor tipuri predefinite din limbajul C++ amintim:

- sir de caractere;
- număr complex;
- listă;
- arbore;
- etc.

Pentru fiecare din aceste concepte se poate defini un tip abstract de date prin intermediul claselor.

Din definiția tipului abstract de date rezultă că acesta are 2 părți, o parte care definește reprezentarea datelor tipului respectiv și o parte care definește operațiile asupra datelor respective.

Partea care definește reprezentarea datelor este formată din componente care sunt date de diferite tipuri. Aceste componente se numesc date membru.

Partea care definește operațiile asupra datelor tipului respectiv conține funcții numite funcții membru.

Prima încercare de a face legătura dintre reprezentarea datelor unui tip și operațiile asupra datelor respective a condus la extinderea construcției *struct*, așa cum s-a văzut în capitolul precedent. Astfel, construcția *struct* permite definirea reprezentării datelor tipului care se introduce de utilizator (*date membru*), precum și enumerarea funcțiilor care definesc operații cu datele respective (*funcții membru*).

Acest fapt nu este încă suficient pentru implementarea tipurilor abstrakte de date deoarece nu se asigură protejarea datelor membru. Ele pot fi accesate direct și de către alte funcții decit funcțiile membru.

Lipsa unei protecții a datelor, face ca tipurile de date introduse prin construcția *struct* să nu poată fi supuse unor controale cu privire la operațiile care se execută asupra lor.

De aceea, pasul următor în implementarea tipurilor abstrakte de date este acela de a introduce protejarea datelor și funcțiilor membru. Acest lucru a condus la noțiunea de *clasa*.

Protejarea datelor și funcțiilor membru se realizează utilizând *modificatorii de protecție*:

- **private**;
- **protected**;
- **public**.

Acești modificatori sunt urmări de două puncte. Modificatorii *private* și *protected* protejează elementele (date și funcții membru) aflate în domeniul lor de acțiune.

Domeniul de acțiune al unui modificator de protecție tine din punctul în care este scris modificatorul respectiv și pînă la sfîrșitul definiției care îl conține sau pînă la un alt modificator de protecție.

Membrii din domeniul de acțiune al modificatorului *public* nu sunt protejați și ei pot fi folosiți fără restricții în tot programul unde ei sunt "vizibili".

În mod implicit, membrii unei clase sunt protejați ca și în cazul în care s-ar afla în domeniul de acțiune a lui *private*. Despre un astfel de membru vom spune că este *privat*. El poate fi utilizat de către o funcție membru dar nu și într-o funcție arbitrară. Vom spune că accesul la un astfel de membru nu se face direct, ci *indirect* prin intermediul funcțiilor membru. Nu același lucru este valabil pentru tipurile definite cu ajutorul construcției *struct*.

În acest caz, membrii *dată* sau *funcție* sunt în mod implicit publici, deci la ei putem face acces direct. Aceasta explică de ce în programul din exercițiul 21.2. se pot folosi direct (în afara funcțiilor membru) expresii de forma:

*a.real*

și

*a.imag*

unde:

*a* - Este o variabilă de tip *complex*.

Intr-adevăr, construcția *struct* nu protejează datele membru *real* și *imag*, deci ele pot fi folosite în mod obișnuit, la fel ca în limbajul C. De aceea, compilatorul nu poate realiza nici un control asupra operațiilor realizate cu datele de tip *complex*, definite cu ajutorul construcției *struct*, ca în exercițiul 21.1. Cu totul alta este situația dacă definim tipul *complex* cu ajutorul unei clase.

Menționăm că pentru a defini o clasă se utilizează același format ca și în cazul construcției *struct*.

La definirea tipului *complex* cu ajutorul clasei vom proteja datele membru, adică *real* și *imag*.

Formatul clasei pentru tipul *complex* va fi:

```
class complex {
    // date membru private
    double real;
    double imag;
```

// funcții membru

...  
};

În continuare, putem defini date de tip complex în mod obișnuit. De exemplu:  
*complex a*;

defineste pe *a* de tip complex. În acest caz, *a* este o dată care are două componente: *real* și *imag*, ambele de tip *double* dar ele sunt private și deci nu putem avea acces direct la ele. De aceea, instrucțiunile de mai jos sunt interzise în afara funcțiilor membru:

```
a.real = 1;
a.imag = 2;
printf("a=%g+i*(%g)\n",a.real,a.imag);
```

Mai mult decât atât, datele de acest tip nici nu pot fi initializate în mod obișnuit. Deci, o declarație de forma:

```
complex a = {1,-1};
```

este interzisă.

Deoarece la datele membru *real* și *imag* este posibil accesul prin intermediul funcțiilor membru, rezultă că este nevoie de funcții membru care să realizeze numai operațiile obișnuite cu numere complexe (modul, argument, adunare, scădere, inmulțire și împărțire), ci și alte operații cum sunt:

- inițializare;
- atribuire;
- afișare.

Este necesar ca toate aceste funcții membru să fie *publice*. Rezultă că o primă variantă pentru definirea tipului *complex* ar putea fi clasa de mai jos:

```
class complex {
    // date membru private

    double real;
    double imag;

    public:      // funcții membru publice
        void atribuire(double x = 0,double y = 0) // real+i*imag = x+i*y
        {
            real = x; imag = y;
        }

        double retreal() // returneaza partea reala a numărului complex
        {
            return real;
        }

        double retimag() // returneaza partea imaginara a numărului complex
        {
```

```

        return imag;
    }

void afiscomplex(char *format)
// afisaza numarul complex conform formatului definit de pointerul format
{
    printf(format,real,imag);
}

void adcomplex(complex *x,complex *y) // calculeaza x+y
{
    real = x -> real + y -> real;
    imag = x -> imag + y -> imag;
}

void negcomplex(complex *x) // calculeaza -x
{
    real = -x -> real;
    imag = -x -> imag;
}

void sccomplex(complex *x,complex *y) // calculaza x-y
{
    real = x -> real-y -> real;
    imag = x -> imag - y -> imag;
}

void mulcomplex(complex *x,complex *y) // calculaza x*y
{
    real = x -> real*y -> real-x -> imag*y -> imag;
    imag = x -> real*y -> imag+x -> imag*y -> real;
}

int divcomplex(complex *x,complex *y);
/* - calculeaza x/y;
   - returneaza:
     0 - la impartirea cu zero;
     1 - altfel.
*/
}; // sfirsitul definitiei clasei complex

int complex :: divcomplex(complex *a,complex *b)
/* - calculeaza a/b;
   - returneaza:
     0 - la impartirea cu zero;
     1 - altfel.
*/
{
    double d;

    d = b -> real*b -> real+b -> imag*b -> imag;
    if(d == 0.0) return 0;
    real = (a -> real*b -> real+a -> imag*b -> imag)/d;
    imag = (a -> imag*b -> real-a -> real*b -> imag)/d;
}

```

```

    return 1;
}

```

Funcțiile membru sint funcții *inline*, exceptind funcția *divcomplex*, care, conținind mai multe instrucțiuni decit celelalte, a fost definită în afara definiției clasei *complex* și deci ea nu este o funcție *inline*.

În antetul funcției *divcomplex* s-a utilizat numele:

complex :: divcomplex

construit cu ajutorul operatorului de rezoluție. Această regulă se utilizează pentru toate funcțiile membru care nu se definesc în interiorul definiției clasei pentru care ele sunt funcții membru.

O clasă definește un tip care a fost numit *tip abstract*.

Partea privată a unei clase definește modul de *implementare* al datelor de tipul abstract definit de clasa respectivă. Partea publică definește *interfața* acestor date cu restul programului. Această interfață conține, de obicei, funcții care au fost numite *funcții membru*. Funcțiile membru se mai numesc și *metode*.

În general, partea publică poate conține și date membru, dar atunci datele respective nu mai sunt protejate, lucru care pe cit posibil trebuie evitat. Așa cum s-a subliniat mai sus, lipsa unei protecții pentru datele membru nu permite compilatorului să controleze operațiile realizate cu datele respective.

În cazul clasei *complex* de mai sus, programatorul nu are acces *direct* la componentele *real* și *imag* ale datelor de tip *complex*. Toate operațiile care se pot realiza cu datele de acest tip se fac numai prin intermediul funcțiilor membru care au fost declarate ca publice.

Datele ale căror tipuri se definesc într-o clasă se numesc *obiecte*. Cu alte cuvinte, un obiect în C++ este o dată de un tip abstract.

Obiectele, la fel ca și datele predefinite, se declară într-o declarație.

În forma cea mai simplă, o declarație de obiect are formatul:

nume\_clasă listă\_de\_nume\_de\_objekte; —> *declarare obiect*

unde prin *listă\_de\_nume\_de\_objekte* înțelegem un nume sau mai multe separate prin virgulă. De asemenea, *nume\_clasă* este un nume.

#### Exemplu:

Dacă se consideră definiția clasei *complex* de mai sus, atunci:

complex z,x1,x2;

declară pe *z*, *x1* și *x2* ca obiecte, adică date de tip *complex*.

Despre obiectele unei clase se obișnuiește să se spună că sunt *instanțieri* ale clasei respective. Deci *z*, *x1* și *x2*, sunt instanțieri ale clasei *complex*.

Pentru a propria tipurile abstracte de cele predefinite, se impune ca obiectele să poată fi *initializate*.

Initializarea unui obiect este o problemă mult mai complexă decit cea a datelor obișnuite (tipuri predefinite sau definite de utilizator cu ajutorul

construcțiilor *struct* sau *union*). De aceea, initializarea unui obiect este posibilă folosind în acest scop o funcție membru specială. Aceasta poartă denumirea de constructor.

Constructorul unei clase are totdeauna *același nume cu numele clasei*. !!!

Dacă o clasă are constructor, atunci acesta se apelează automat la instantierea unui obiect al clasei respective.

#### Exemplu:

Pentru clasa *complex* definită mai sus, putem defini un constructor cu ajutorul funcției membru:

```
complex(double a=0,double b=0) —  
// constructorul clasei complex  
{  
    real = a;  
    imag = b;  
}
```

Accastă funcție membru se presupune că se definește în interiorul definiției clasei *complex*. Deci, este o funcție inline. Pentru a o defini în afara definiției clasei *complex* se schimbă antetul funcției astfel:

```
inline complex :: complex(double a,double b)
```

Pentru a initializa un obiect, în declarația obiectului, după numele lui, se includ în paranteze rotunde valorile efective ale parametrilor constructorului.

#### Exemplu:

```
complex z(1,2);
```

Se instantiază obiectul *z*, de tip *complex*, ale cărui componente au valorile inițiale:

```
z.real = 1  
și  
z.imag = 2
```

În cazul în care există o singură valoarea de initializare, atunci se poate utiliza semnul de atribuire (=). De exemplu, declarația:

```
complex z = 123;  
este identică cu declarația:
```

```
complex z(123);
```

Ele inițializează obiectul *z* astfel:

```
z.real = 123;  
z.imag = 0 - valoarea implicită.
```

La o declarație de forma:

complex z;

adică fără initializare, constructorul este apelat în mod automat și se instantiază obiectul *z* cu ambele componente egale cu zero (valorile implicate ale parametrilor constructorului).

Obiectele sunt *distruse* de către programator sau în mod automat.

Distrugerea obiectelor se poate realiza folosind o funcție membru specială numită destructor. Un destructor are un nume aparte și anume:

~nume

unde:

nume - Este numele clasei. !

De asemenea, destructorul nu are parametri. El se apelează de programator cind nu mai este nevoie de un obiect. Acest lucru este important atunci cind obiectul a fost creat în memoria *heap*.

Destructorul se apelează automat cind obiectul își incetează existența (vezi clasele de memorie).

Funcțiile membru se apelează calificând numele lor cu numele obiectului pentru care se realizează apelul:

nume\_obiect.nume\_functie\_membru(lista parametrilor efectivi) !!!

De exemplu, dacă *z* este o instanțiere a clasei *complex* (*complex z;*), atunci pentru a afișa partea reală și cea imaginată a obiectului *z*, se va apela funcția membru *afiscomplex* ca mai jos:

```
z.afiscomplex("Partea reală = %g\tPartea imaginată = %g\n");
```

În mod analog, dacă obiectele *z*, *z1* și *z2* sunt instanțieri ale clasei *complex*, atunci pentru a realiza atribuirea:

```
z = z1+z2;
```

se va apela funcția membru *adcomplex* astfel:

```
z.adcomplex(&z1,&z2);
```

În cazul în care se utilizează un pointer spre un obiect:

nume\_clasa \*po;

funcțiile membru se apeleză printr-o construcție de forma:

po -> nume\_functie\_membru(lista parametrilor efectivi) !!!

În corpul unei funcții membru, ne putem referi la obiectul pentru care a fost apelată funcția membru folosind pointerul implicit this. Acest pointer se utilizează în corpul funcțiilor membru în mod implicit la datele membru ale obiectului pentru care este apelată funcția membru respectivă. Cu toate acestea, programatorul poate utiliza explicit pointerul *this* pentru a se referi, în corpul unei funcții membru, la obiectul pentru care s-a apelat funcția respectivă.

În continuare, prin obiect curent vom înțelege obiectul pentru care s-a apelat o funcție membru, adică obiectul spre care pointează pointerul *this*.

Apelul unei funcții membru referitor la un obiect al clasei pentru care funcția respectivă este funcție membru, simplifică modul de exprimare al operațiilor care se realizează asupra obiectelor. De asemenea, acest mod de apel poate fi controlat de compilator. Orice apel neautorizat va fi semnalat și eliminat de compilator. În felul acesta, lucrul cu obiectele unei clase devine similar cu cel existent pentru datele de tipuri predefinite:

- obiectele se pot inițializa;
- asupra lor se pot face numai operații în prealabil definite prin funcții membru și care sunt controlabile prin compilator.

Protecția datelor și funcțiilor membru ale unei clase care nu sunt *public* constituie un mijloc de "ascundere" a elementelor membru respective. Această "ascundere", numită și incapsulare se realizează la un nivel calitativ superior față de ascunderea realizată cu ajutorul modulelor.

De exemplu, prin intermediul clasei *complex*, definită mai sus, se pot instanța oricite obiecte (numere) de tip *complex* care pot fi și inițializate, iar operațiile cu ele se realizează simplu cu ajutorul funcțiilor membru și aplicarea lor este controlată de compilator.

Mai jos, introducem tipul abstract *stiva* pentru a pune mai bine în evidență diferența dintre facilitățile obținute pe bază utilizării claselor față de cele oferite de programarea modulară.

Amintim că în capitolul 7, a fost definit un modul pentru implementarea unei stive cu ajutorul unui tablou de tip *int*.

Mai jos, vom implementa tipul *stiva* tot cu ajutorul unui tablou de tip *int* care are cel mult 100 de elemente.

Modulul definit în capitolul 7 implementează o *singură* stivă. În cazul de față se pot instanța oricite obiecte de tip *stivă*.

```
enum Boolean {false,true};
class stiva {
    int stack[100];
    int istack;

    public:
        stiva() // constructor
        {
            istack=0; // la inceput stiva este vida
        }

        void push(int n); // punе pe n pe stiva

        int pop(); // scoate elementul din virful stivei
                    // se returneaza elementul din virful stivei

        int top(); // returneaza elementul din virful stivei
                    // elementul ramine pe stiva
```

```
void clear() // videaza stiva
{
    istack=0;
}

Boolean empty() // returneaza true daca stiva este vida si false altfel
{
    return istack == 0 ? true : false;
}

Boolean full() // returneaza true daca stiva este plina false altfel
{
    return istack == 100 ? true : false;
}
```

;

În continuare, se pot instanția obiecte de tip *stiva*:

```
stiva stival, stiva2;
```

La instanțierea obiectelor *stival* și *stiva2* se apelează în mod automat constructorul clasei. Aceasta realizează atribuirea:

```
istack = 0;
```

În felul acesta, stivele *stival* și *stiva2* sunt vide la instanțiere.

Datele membru *stack* și *istack* sunt "incapsulate" (ascunse) în clasa *stiva*, fiind private. Ele nu pot fi accesate direct ca și în cazul modulului definit în capitolul 7.

Stivele se pot gestiona simplu cu ajutorul funcțiilor membru. Astfel, pentru a pune un element pe stiva *stival*, utilizăm apelul funcției *push*:

```
stival.push(expresie)
```

În mod analog, punem un element pe stiva *stiva2*, apelând aceeași funcție *push* ca mai jos:

```
stiva2.push(expresie)
```

În felul acesta, funcțiile membru gestioneză simplu oricite stive de tip *stiva*. Acest lucru nu este posibil prin intermediul modulului descris în capitolul 7. Evident, se poate construi un modul care să gestioneze mai multe stive, dar acest lucru nu se realizează atât de simplu și elegant ca mai sus, prin utilizarea claselor. Acest fapt explică afirmația lui B. Stroustrup: dacă într-un program este suficient un *singur* exemplar al unei date de un anumit tip, atunci este suficient un modul pentru lucrul cu data respectivă. Altfel, se definește o clasă pentru tipul datei respective și astfel se vor putea instanța oricite exemplare.

În aceasta constă esența saltului realizat prin trecerea de la programarea modulară la stilul programării prin abstractizarea datelor.

În rezumat, programarea prin abstractizarea datelor are la bază utilizarea tipurilor abstracție de date. În acest scop, se pornește cu stabilirea conceptelor

necesare la realizarea unui program. O parte din aceste concepte se realizează cu ajutorul tipurilor predefinite, existente în limbajul de programare utilizat (în cazul de față limbajul C++). Celelalte concepte se realizează sub formă de tipuri abstracte de date care se definesc în limbajul C++ cu ajutorul claselor.

Un tip abstract de date prezintă două aspecte: unul legat de implementarea tipului și celălalt legat de utilizarea lui, care definește interfața tipului respectiv cu restul programului.

Implementarea tipului este partea lui protejată, iar interfața constituie partea publică a tipului, utilizabilă în tot programul.

Despre partea protejată se spune că conține informația encapsulată în clasa care definește tipul respectiv.

La definirea unei clase se precizează:

- reprezentarea datelor tipului abstract;
- funcțiile care descriu operații cu datele tipului abstract.

Reprezentarea datelor se definește prin componente date numite date membru.

Funcțiile care definesc operații cu datele tipului abstract, se numesc funcții membru.

De obicei, partea encapsulată în clasă conține date membru, dar ea poate conține și funcții membru.

Elementele publice sunt funcții membru. Ele pot fi și date membru, dar atunci acestea nu mai sunt protejate, lucru care este bine să fie evitat.

Un tip abstract de date trebuie astfel definit, încât la utilizarea lui să se facă abstracție de implementare. Mai mult decât atât, utilizarea datelor de un tip abstract nu trebuie să fie influențată de implementare. Aceasta înseamnă că implementarea tipului abstract poate fi oricând schimbată, fără a schimba și utilizarea, în program, a datelor tipului respectiv.

De exemplu, la implementarea tipului *stiva* s-a folosit un tablou de tip *int* de 100 de elemente și variabila *istack* de tip *int*.

Schimbând implementarea prin înlocuirea tabloului cu o listă simplu înlanțuită (vezi capitolul 11), interfața formată din funcțiile membru:

*push, pop, top, clear, empty și full*

se utilizează în același mod. Evident, funcțiile se modifică, dar ele sunt apelate la fel și au același efect. Dacă acest lucru nu este realizabil, înseamnă că tipul abstract nu a fost bine definit, deoarece la utilizarea lui nu se poate face abstracție de implementare.

Datele de un tip abstract sunt numite obiecte ale tipului respectiv.

Obiectele se declară printr-o declarație asemănătoare cu declarația datelor pentru tipuri predefinite. Ele pot fi inițializate la declararea lor.

Obiectele se creează, de obicei, printr-un constructor, care este o funcție membru specială, al cărui nume coincide cu numele clasei. Constructorul unei clase se apelează automat la construirea unui obiect al clasei respective. Despre un obiect al unei clase se spune că este o instanțiere a clasei respective.

Un obiect poate fi distrus printr-o funcție specială numită *destructor*. Numele *destructorului* este numele clasei precedat de caracterul *"~"*.

Funcțiile membru pot fi definite în interiorul definiției clasei dacă sunt foarte simple. Aceasta deoarece, în acest caz, ele se definesc în mod implicit ca funcții inline.

O funcție membru definită în afara definiției clasei are numele din antet construit cu ajutorul operatorului de rezoluție:

*nume\_clasa :: nume\_functie\_membru*

La apelul unei funcții membru se califică numele funcției cu numele obiectului pentru care se apelează funcția.

Facilitățile obținute cu ajutorul claselor apropie modul de tratare și utilizare al obiectelor de cel al datelor de tipuri predefinite.

O diferență între utilizarea obiectelor și a datelor de tipuri predefinite apare la scrierea expresiilor. Așa de exemplu, dacă se consideră datele declarate ca mai jos:

```
int i,j,k;  
double a,b,c;
```

Atunci se pot utiliza expresii de forma:

```
k = i+j;  
c = a+b;
```

Care sunt sugestive față de exprimarea adunării datelor de tip *complex* prin apelul funcției membru *adcomplex*:

```
complex z,u,v;  
...  
v.adcomplex(&z,&u);
```

De aceea, ar fi bine ca expresii de forma:

```
v = z+u;
```

să fie acceptate de compilator și pentru obiecte. Această idee ne conduce la noțiunea de *supraîncărcare a operatorilor*. Într-adevăr, se poate spune că operatorul + este supraîncărat pentru tipuri predefinite.

Ei se poate aplica la operanzi de tipuri numerice predefinite (*int, long, float, double, unsigned și long double*). De aceea, pentru a putea scrie adunarea obiectelor complexe *z* și *u* sub forma expresiei:

*z+u*

Ca în cazul tipurilor numerice predefinite, este suficient ca operatorul + să poată fi supraîncărat cu operanzi de tip *complex*.

Supraîncărcarea operatorilor este o facilitate care simplifică mult exprimarea operațiilor asupra obiectelor. Pe baza ei se pot scrie expresii cu obiecte la fel ca și cu date de tipuri predefinite.

Supraincărcarea operatorilor se realizează prin construcții asemănătoare funcțiilor dar care au un antet special în care este prezent cuvintul *operator*.

În sfîrșit amintim că, supraincărcarea operatorilor pentru obiecte este completată și cu diferite conversii care se aplică la operații cu obiectele respective. În felul acesta se ajunge să utilizăm obiectele la fel de simplu ca și datele de tipuri predefinibile.

În încheiere putem afirma că, clasele sunt un instrument pentru a *crea* tipuri noi care pot fi utilizate tot așa de bine ca și tipurile predefinibile. Ideal, tipurile noi (tipurile abstractive) nu trebuie să difere, în utilizare, de tipurile predefinibile, ci numai în modul în care sunt create.

## 22.1. Definiția claselor

O clasă definește un tip abstract de date. Ea are o definiție al cărei format simplificat este indicat mai jos:

\* **class nume {lista\_elementelor\_membru};**

Ulterior o să vedem și un alt format mai complex, pentru clase.

În limbajul C++, formatul de mai sus poate fi folosit și pentru a defini tipuri noi cu ajutorul cuvintelor cheie *struct* și *union*. Deci, în formatul de mai sus, se poate înlocui cuvintul *class* prin *struct* sau *union*. De fapt, în cele ce urmează, vom considera că și formatele în care cuvintul cheie *class* se înlocuiește prin *struct* sau *union* definesc tot clase. Diferența constă în aceea că, în cazul utilizării cuvintului *class*, elementele membru în mod implicit sunt protejate prin protecția oferită de modifierul de protecție *private*, iar în cazul lui *struct* și *union*, în mod implicit elementele membru sunt neprotejate (publice).

În cazul utilizării cuvintului *struct* se poate modifica protecția cu ajutorul modifierilor de protecție. În schimb, în cazul utilizării cuvintului *union*, elementele membru pot fi numai publice. De aceea, se obișnuiește să se spună că *struct* și *union* definesc clase cu elemente membru publice.

În cele ce urmează vom înțelege prin clasă de tip *struct* o clasă definită cu ajutorul lui *struct*, iar prin clasă de tip *union*, o clasă definită cu ajutorul lui *union*.

Numele aflat după *class*, *struct* sau *union* este numele tipului introdus prin definiția respectivă. El se numește și *numele clasei* și în continuare poate fi utilizat pentru a declara (instanta) date de tipul respectiv, date numite și obiecte de tipul respectiv.

Lista elementelor membru poate conține:

- declarații de date;
- definiții de funcții;
- prototipuri de funcții;
- modifieri de protecție.

Datele declarate într-o definiție de clasă se numesc *date membru*.

Funcțiile definite sau pentru care este prezent numai prototipul, în definiția clasei, se numesc *funcții membru*.

Amintim că funcțiile definite în definiția unei clase trebuie să fie simple, deoarece ele se definesc în mod implicit ca funcții *inline*. Pentru funcțiile mai complexe se indică în definiția clasei numai prototipurile funcțiilor respective și ele se definesc ulterior. În acest caz, antetul funcției conține un nume format cu ajutorul operatorului de rezoluție:

**nume\_clasă :: nume\_funcție\_membru**

De obicei, o clasă are unul sau mai mulți constructori precum și un destrutor. Aceștia sunt funcții membru de nume speciale, și anume, numele constructorilor coincide cu numele clasei (dacă sunt mai mulți constructori, atunci aceștia sunt funcții supraincărcate), iar numele destrutorului este numele clasei precedat de caracterul ~ (totdeauna există cel mult un destritor).

Un alt caz particular de funcții membru sunt cele care definesc operatori și conversii pentru obiectele clasei respective.

Ca modifieri de protecție am văzut că se pot folosi:

- **private**:
- **protected**:
- și
- **public**:

Primii doi asigură o protecție a datelor sau funcțiilor membru din domeniul de acțiune al lor, iar ultimul se utilizează pentru elemente membru care dorim să nu fie protejate.

Domeniul unui modifier de protecție începe din punctul în care este scris și pînă la sfîrșitul definiției clasei care îl conține sau pînă la întîlnirea unui alt modifier de protecție.

De obicei, datele membru sunt protejate, dar aceasta nu înseamnă că ele nu pot fi publice.

De asemenea, funcțiile membru, numite și metode, de obicei sunt publice, dar ele pot fi și protejate.

Clasele pot fi definite incomplet în cazul în care este nevoie să ne referim la ele. O astfel de definiție incompletă are formatul:

- **class nume;**
- **struct nume;**
- sau
- **union nume;**

Datele membru se declară în mod obișnuit. Dacă este prezentă o clasă de memorie, atunci aceasta poate fi numai clasa de memorie *static*.

O dată membru a unei clase nu poate fi de tipul definit prin clasa respectivă.

Ea poate fi numai un pointer spre tipul respectiv sau o referință la tipul respectiv.

Fie definiția:

```

class nume {
    ...
    nume *ptr; // corect
    ...
    nume& ref; // corect
    ...
    nume obiect; // incorrect
};

```

Data membru *obiect* de tip *nume* nu este admisă. În schimb, se pot declara date de tipuri introduse prin alte clase.

Domeniul de existență al numărului unei clase este din punctul definirii ei pînă la sfîrșitul blocului (instrucțiunii compuse care o conține).

De obicei, definiția unei clase se scrie la începutul fișierului sursă în care este utilizată și în afara oricărui bloc. Se obișnuiește adesea să se construiască un fișier de tip *h* care să conțină definiții de clase. Fișierul respectiv se include la începutul fiecărui fișier care utilizează definițiile claselor respective.

## 22.2. Obiecte

Un obiect este o dată de un tip definit printr-o clasă. Se obișnuiește să se spună că este o instantiere a clasei respective.

Declarația de obiecte este asemănătoare cu cea pentru datele de tipuri predefinite.

În cea mai simplă formă, un obiect se declară folosind formatul:

*nume\_clasa nume\_object;*

De obicei, clasele au constructori care se apeleză automat la intilnirea declaratiei de instantiere a unui obiect al clasei respective.

Datele membru se alocă distinct la fiecare instanțiere a clasei. Deci, datele membru există în atîtea exemplare cîte obiecte au fost instanțiate. O excepție o constituie datele membru care au clasa de memorie *static*. O dată membru de clasa de memorie *static* (numită *dată membru statică*) este o parte comună pentru toate instantierile clasei și există într-un singur exemplar.

Funcțiile membru sunt într-un singur exemplar oricîte instanțieri ar exista. O funcție membru se apeleză totdeauna în strînsă dependență cu un obiect care este o instanțiere a clasei pentru care funcția respectivă este funcție membru.

Legătura dintre funcții membru și obiectul pentru care se face apelul se realizează folosind operatorul punct sau săgeată.

**Exemplu:**

Considerăm tipul *complex* definit la începutul capitolului. Fie declarațiile:

complex z;

```

complex *pz;
char *format = "%g\t%g\n";

```

Atunci:

*z.afiscomplex(format);*

afisează componentele numărului complex *z* (în cazul de față 0 0 deoarece constructorul clasei are parametri impliciti zero).

Același efect se obține folosind secvența:

```

pz = &z
pz -> afiscomplex(format);

```

Programatorul poate utiliza în mod explicit pointerul *this*, în corpul unei funcții membru. Acesta are ca valoare adresa obiectului curent, adică a obiectului pentru care s-a facut apelul funcției membru.

O excepție de la aceste reguli o reprezintă funcțiile membru care au clasa de memorie *static*. Acestea se numesc funcții membru statice. O astfel de funcție poate fi apelată în două moduri:

- Ca orice funcție membru, folosind operatorul punct sau săgeată.
- Independent de un obiect al clasei pentru care este funcție membru. În acest caz, apelul se realizează folosind operatorul de rezoluție, adică sub forma:

*nume\_clasa :: nume\_funcție\_membru\_statica*

În acest caz, pointerul *this* nu mai poate fi utilizat.

**Exemplu:**

Fie definiția de clasă:

```

class dc {
    int zi,luna,an;
    static int zz,ll,aa; → comune pt. V instanțiere
public:
    dc(int z=1,int l=1,int a=1995)
    {
        zi = z; luna = l; an = a;
    }
    static Boolean v_calend(dc *d);
    ...
};

```

Datele membru *static*: *zz*, *ll*, *aa* sunt comune pentru toate instanțierile, spre deosebire de celelalte date care se alocă distinct la fiecare instanțiere. Acestea din urmă, se initializează cu ajutorul constructorului.

Funcția *v\_calend* verifică data calendaristică definită de datele membru *zi*, *luna* și *an*, care sunt componente ale obiectului spre care pointează *d*, precum și cea definită de datele membru statice *zz*, *ll* și *aa*. Ea poate fi definită ca mai jos:

```

Boolean dc :: v_calend(dc *d)
/* verifică data calendaristică
   - zi, luna, an, alc obiectului spre care pointează d;
   - zz, ll, aa;
   - returnează True dacă datele sunt corecte și False altfel.
*/
{
    static tnrz[] = {0,31,28,31,30,31,30,31,31,31,30,31,30,31};
    int z,l,a;

    if(d->an < 1600 || d->an > 4900) return False;
    if(d->luna < 1 || d->luna > 12) return False;
    if(d->zi < 1 || d->zi > tnrz[d->luna]+
       (d->luna==2 && (d->an%4==0 && d->an%100 ||
       d->an%400 == 0))) return False;
    z = dc::zz;
    l = dc::ll;
    a = dc::aa;

    if(a < 1600 || a > 4900) return False;
    if(l < 1 || l > 12) return False;

    if(z<1 || z>tnrz[l]+(l==2 && (a%4==0 && a%100 || a%400==0)))
       return False;
    return True;
}

Fie instanțierea:
dc data_calend(29,2);

```

Pentru a valida instanțierea `data_calend` se va apela funcția `v_calend` astfel:

```

if(dc::v_calend(&data_calend) == False){
    // data calendaristica cronata
}
else{
    // data calendaristica corecta
}

```

În acest caz, funcția `v_calend` a fost apelată folosind operatorul de rezoluție. Se verifică data calendaristică definită de datele membru `zi`, `luna` și `an` a instanțierii `data_calend`. Constructorul acestei clase a inițializat obiectul `data_calend` cu valorile:

```

zi = 29
luna = 2
an = 1995

```

Datele membru statice(`zz, ll, aa`) există în afara instanțierilor clasei `dc`. Deși aceste date sunt private, referirea la ele se poate face în funcția `v_calend` care este o funcție membru a clasei `dc`. Totuși, referirea la ele nu se poate face direct,

deoarece în acest caz pointerul `this` nu este definit, funcția `v_calend` nefiind apelată pentru un obiect. De aceea, referirea la datele statice `zz`, `ll`, și `aa` s-a facut folosind numele clasei și operatorul de rezoluție:

$$\left. \begin{array}{l} \text{dc :: zz, dc :: ll} \\ \text{și} \\ \text{dc :: aa} \end{array} \right\} ! \text{ pt. date membru static }$$

## 22.3. Domeniul unui nume

Unui *nume* îi corespunde un *domeniu*. Acesta se definește prin declarația lui sau este *corpul unei funcții* dacă el este numele unei *eticheți*.

Prin *bloc* înțelegem o instrucțiune compusă. În limbajul C o declarație poate fi în interiorul unui bloc sau în afara blocurilor. În plus, în limbajul C++ o declarație poate fi și în interiorul unei clase (în interiorul declarației de clasă).

O declarație, care este în afara blocurilor sau a claselor și care nu este o declarație de date externe, se spune că este o *definiție*.

Pentru un nume vom distinge 3 tipuri de domenii, în funcție de poziția declarației (definiției) sale. Aceste tipuri sunt: *local*, *fisier* și *clasă*.

Un nume declarat într-un bloc are un *domeniu de tip local*. Aceasta începe în punctul în care este declarat și ține până la sfârșitul blocului respectiv. Un astfel de nume poate fi utilizat în domeniul lui, inclusiv în blocurile incluse în domeniul respectiv, dacă el nu este redeclarat în aceste blocuri.

### Exemplu:

Fie instrucțiunile compuse imbricate de mai jos:

```

{
    ...
    int i; // începe domeniul de tip local a lui i
    ...
    // nu începe alt bloc

    i=10; // atribuire corecta
    ...

    {
        int j; // nu există declarație pentru j
        j=i+2; // expresie corecta deoarece i se află în
        // domeniul lui și nu a fost redeclarat
        ...
    }

    long n;
    long i; // redeclararea lui i
    // în acest bloc nu se poate utiliza variabila
}

```

```

// i declarata la inceput de tip int
...
n = i+20; // se utilizeaza i de tip long
...
} // in acest punct se termina domeniul lui i declarat de tip long i
... // nu exista redeclararea lui i
// este valabila prima declaratie a lui i
i=100; // se atribuie 100 variabilei i de tip int
...
} // in acest punct se termina domeniul lui i declarat int i

```

Un nume declarat în afara oricărui bloc sau **declarație (definīție) de clasă** are un **domeniu de tip fișier**. Acest domeniu începe în punctul în care numele este definit și ține pînă la sfîrșitul fișierului care conține definiția respectivă. El poate fi utilizat în domeniul respectiv fără nici o restricție dacă nu este redefinit în blocurile incluse în domeniul său.

Dacă un nume care are un domeniu de tip fișier este redefinit într-un bloc inclus în domeniul său, atunci el poate fi folosit în acel bloc, dacă este precedat de operatorul de rezoluție.

#### Exemplu:

```

int i = 100; // definīția lui i nu este inclusă în nici un bloc sau clasa;
// are domeniu de tip fișier care începe în acest punct
...
int j;
... // i nu este redeclarat pînă în acest punct
j=i+123; // se utilizează i definit mai sus
...
long i; // redeclarare a lui i
// începe un domeniu de tip local pentru i
j=i-123; // se utilizează i declarat prin long i
...
j=j+ :: i // se utilizează i declarat prin int
...
}
...
// sfîrșit fișier: se termină domeniul lui i declarat prin int i

```

O clasă are o declarație (definīție) care definește un **tip abstract de date**. **Numele acestui tip** este considerat totodată ca fiind **numele unei clase**.

Numele unei clase are un domeniu care se stabilește la fel ca și domeniul oricărei variabile.

Un nume de clasă poate fi redeclarat ca orice nume. În acest caz, putem folosi numele respectiv într-un domeniu inclus în care este redefinit folosind construcția:

```
class :: nume_clasa
```

în locul numelui.

#### Exemplu:

```

class a {
...
};

... // instrucție compusă aflată în domeniul numelui a
double a; // redeclararea lui a
a = 3.1415; // atribuire corectă
class::a x; // x este o instanțiere a lui a
...
}

```

Această regulă se utilizează și în cazul construcțiilor **struct**, **union** și **enum**. De fapt, construcțiile **struct** și **union** se consideră că definesc clase ale căror elemente membru sunt toate publice.

Un nume al unui element membru al unei clase care nu este public are un **domeniu de tip clasă**. Aceasta înseamnă că el poate fi folosit numai în corpul funcțiilor membru ale clasei respective.

Elementele membru statice vor fi prefixate de numele clasei urmat de } \* operatorul de rezoluție (vezi paragraful precedent).

O clasă poate conține ca și date membru obiecte care sunt instantieri ale altor clase, dar nu ale clasei respective.

#### Exemplu:

```

class clasa {
...
clasa1 obiect1; // obiect1 este instantierea clasei clasa1
clasa obiect; // eroare; o clasa nu poate conține ca obiecte
// membru instantieri ale ei
...
};

```

În schimb, se pot defini date membru ale unei clase care să fie pointeri sau referințe spre obiectele clasei respective.

#### Exemplu:

```

class clasa {
...
clasa *pobiect; // pointer spre un obiect de tip clasa
clasa& robiect; // referință la un obiect de tip clasa
...
};

```

Clasele pot fi declarate incomplet.

O astfel de declarație are formatul:

```
class nume;
```

Astfel de declarații se pot folosi și în cazul construcțiilor **struct** și **union**.

### Exemplu:

```
class a;  
struct s;  
union u;  
class b {  
    ...  
    a *pa;  
    s *ps;  
    u *pu;  
    ...  
};
```

## 22.4. Vizibilitate

Un nume este *vizibil* în domeniul său dacă nu este redefinit în blocuri incluse în domeniul respectiv. Un nume redefinit în blocuri din domeniul său, devine temporar ascuns. Un nume cu domeniul de tip fișier poate fi făcut vizibil în domeniul în care este redefinit, folosind operatorul de rezoluție iar dacă numele respectiv este numele unei clase, atunci el va fi precedat de cuvântul cheie corespunzător: *class*, *struct* sau *union*.

Domeniul de vizibilitate al unui nume este acea parte a domeniului său în care el poate fi utilizat.

De obicei, domeniul de vizibilitate al unui nume coincide cu domeniul său.

## 22.5. Durata de viață a datelor

Durata de existență a datelor este legată de clasa de memorie a acestora. Prin *durata de viață a datelor* se înțelege perioada în care ele sunt alocate în memorie.

Există 3 feluri de durată: *statică*, *locală* și *dinamică*.

### \*Durata statică

- Înseamnă că data respectivă este alocată în memorie pe perioada execuției programului: de la lansare și până la terminarea execuției programului.

Datele care au un domeniu de tip fișier sunt date cu durată statică.

Datele care au un domeniu de tip fișier sunt date globale sau locale fișierului dacă ele sunt declarate cu ajutorul cuvintului cheie *static*. În ambele cazuri, ele au o durată statică. De asemenea, datele care au un domeniu de tip local au o durată statică, dacă sunt declarate cu ajutorul cuvintului cheie *static*.

În concluzie, datele globale, precum și cele declarate cu ajutorul cuvintului *static* au o durată statică.

### \*Durata locală

- Este durata datelor *automatice*. Acestea sunt date cu domeniu de tip local și care sunt alocate, la execuție, pe stivă sau în registri. Ele nu conțin, în declarația lor, cuvintul

*static*. Alocarea pe stivă se face cînd controlul programului ajunge la locul în care sunt declarate. Cînd controlul programului ieșe din blocul în care sunt declarate, datele respective se elimină de pe stivă.

### \* Durata dinamică

- Este durata datelor alocate în memoria *heap*. Acestea se alocă și se eliberează la execuție prin funcții sau operatori corespunzători. Ea se realizează de către programator. În acest scop, în limbajele C și C++, se pot utiliza funcțiile *malloc* și *free*.

De obicei, în limbajul C++ se utilizează operatorii *new* și *delete* (vezi capitolul 20).

## 22.6. Alocarea și dezalocarea obiectelor

*Alocarea* obiectelor se face în funcție de durată de viață a obiectelor. Obiectele de durată statică și locală se alocă automat.

Obiectele dinamice se alocă de către programator. În acest scop, de obicei, se utilizează operatorul *new*.

În mod analog, *dezalocarea* obiectelor se realizează automat dacă ele au durată statică sau locală și de către programator dacă sunt dinamice. Obiectele alocate cu ajutorul operatorului *new* se dezalocă cu ajutorul operatorului *delete*.

Alocarea obiectelor se mai numește și *crearea sau construirea* obiectelor.

Dezalocarea obiectelor se mai numește și *distrugerea* obiectelor.

Alocarea obiectelor statice care sunt globale se realizează înainte de execuția funcției *main* a programului. Ele se dezalocă la terminarea programului ca parte a procedurii de ieșire din funcția *main*.

Obiectele locale se alocă în momentul în care domeniul lor devine activ, adică atunci cînd controlul programului ajunge la instanțierea lor. Ele se dezalocă în momentul în care controlul programului ieșe din domeniul lor.

Alocarea obiectelor este o operație mai complexă decit alocarea datelor de tip predefinit sau definit de utilizator. Alocarea datelor de tip predefinit sau definit de utilizator se poate face împreună cu inițializarea datelor respective. În cazul obiectelor, inițializarea datelor membru este o problemă mai complexă din cauza protecției datelor respective. De aceea, se pune problema ca inițializarea obiectelor să fie o operație care se realizează la alocarea lor, prin funcții membru speciale. Aceste funcții membru, care realizează alocarea și inițializarea obiectelor, se consideră că ele construiesc obiectul care se instanțiază. De aceea, ele se numesc *constructori*.

Un constructor este o funcție membru al unei clase care se apelează la fiecare instanțiere. El are același nume ca și numele clasei.

Dezalocarea unui obiect este și el un proces complex care se realizează cu o funcție membru specială numită *destructor*. Destructorul unei clase se apelează

la distrugerea obiectului. El se apelează automat sau uncori explicit de către programator. Apelul explicit al destructorului se face pentru a distruge obiectele dinamice. Pentru celelalte obiecte, destructorul se apelează automat la înecarea existenței lor:

- la ieșirea prin funcția *exit* pentru obiectele globale;
- la ieșirea din domeniul unui obiect de durată locală.

## 22.7. Inițializare

Datele pot fi inițializate prin declarațiile (definițiile) lor.

În general, datele de durată statică (globale sau care au clasa de memorie *static*, adică declarația lor începe prin cuvântul cheie *static*) (vezi cap. 5 și 6) *ninițializate*, au valoarea inițială egală cu zero. Celelalte categorii de date dacă nu sunt inițializate, au o valoare inițială *nedefinită*.

Datele de tipuri predefinite și cele definite de utilizator se inițializează folosind formatele din limbajul C, care au fost precizate în capitolul 6 și paragraful 10.1.

În limbajul C++ se pot inițializa și datele de tip utilizator definite cu ajutorul construcției *union*. Aceasta este posibil numai pentru prima componentă a reuniunii.

Elementele unui tablou, unei structuri, unei reuniuni sau un enumerator, se pot inițializa prin *expresii constante*, adică expresii care conțin operanzi ce pot fi evaluati la compilare, la întlnirea lor. În aceste expresii se poate folosi operatorul *sizeof*.

Datele care nu sunt tablouri, structuri, reuniuni sau de tip enumerare, se pot inițializa prin expresii care nu este necesar să fie expresii constante. În acest caz, operanții expresiilor care se utilizează la inițializare, trebuie să poată fi evaluati în momentul în care controlul programului ajunge la ei.

**Exemplu:**

```
int f(int n)
{
    int i = n+10;
    /* parametrul n are precizată valoarea cînd controlul programului ajunge la evaluarea
       expresiei n+10 și anume valoarea lui este egală cu a parametrului efectiv de la apelul lui f
    */
    int j = k-2 // eroare; k nu este definit la întlnirea expresiei k-2
    int k=3;
}
```

Un loc aparte îl ocupă inițializarea obiectelor. Așa cum s-a spus mai sus, obiectele se inițializează cu ajutorul constructorilor care se apelează în mod automat la instanțierea lor.

Datele membru ale obiectelor statice (*obiecte de durată statică*) care nu sunt

inițializate au valoarea inițială zero.

Datele membru ale celorlalte obiecte (*obiecte de durată locală sau dinamică*) care nu sunt inițializate, au o valoare inițială nedefinită.

Datele membru statice se inițializează în afara constructorilor. Aceasta, deoarece o dată membru statică este o zonă comună care nu se multiplică la fiecare instanțiere a clasei respective.

Inițializarea unei astfel de date se realizează la fel ca o dată globală obișnuită, adică printr-o definiție a datei respective în care este prezentă și valoarea de inițializare, definiție care se scrie în afara corpului oricărei funcții.

**Exemplu:**

```
class dc {
    int zi,luna,an;
    static int zz,ll,aa;
public:
    ...
};
```

Datele membru statice *zz*, *ll*, *aa* se vor inițializa astfel:

```
{ int dc :: zz = 1;
int dc :: ll = 1;
int dc :: aa = 1600;
```

Se observă prezența numelui clasei și a operatorului de rezoluție pentru a specifica faptul că *zz*, *ll* și *aa* sunt date membru ale clasei *dc*. În absența numelui clasei și a operatorului de rezoluție, datele *zz*, *ll*, *aa* devin date globale inițializate cu valorile respective.

Atribuirile de forma:

```
dc :: zz = 1; dc :: ll = 1; dc :: a = 1600;
```

sunt posibile, dar numai dacă sunt scrise în corpul unei funcții membru (datele respective sunt protejate). Astfel de atribuirile, de obicei, nu sunt considerate a fi inițializări. Ele se realizează numai dacă funcția membru care le conține este apelată. Dacă datele membru statice nu se inițializează, ele nu trebuie definite în modul indicat mai sus. Ele sunt alocate în mod automat și au valoarea inițială egală cu zero.

## 22.8. Constructor

Datele de tip predefinit sau definit de utilizator se alocă în mod automat, în conformitate cu declarația sau definiția acestora. Odată cu alocarea datelor se pot face și inițializări.

În cazul obiectelor, acestea se alocă la instantierea lor. De asemenea, obiectele pot fi inițializate la instanțiere. În acest scop, utilizatorul poate defini constructori, care sunt funcții membru de același nume cu numele clasei. Se pot

defini mai mulți constructori pentru o clasă. În acest caz, ei sunt funcții supraincarcate și deci ei difera prin numărul și/sau tipurile parametrilor.

Valorile de initializare se transferă constructorului și ele joacă același rol ca parametrii efectivi de la apelurile funcțiilor obișnuite. Ele formează o listă care se include între paranteze rotunde și sunt prezente în declarația (definiția) obiectelor. În felul acesta, o declarație sau definiție de obiect poate avea formatul:

nume\_clasa nume\_object(lista);

unde:

lista - Este formată dintr-o expresie sau mai multe, separate prin virgulă.

\* Lista, împreună cu parantezele care o includ, sunt absente dacă obiectul nu se initializează sau dacă există un constructor cu toți parametri impliciti.

#### Exemplu:

Se consideră clasa *complex* definită ca mai jos:

```
class complex {
    double real;
    double imag;
public:
    complex(double x=0,double,y=0)

    /* constructor pentru numere complexe; implicit se instantiază
       numarul complex cu ambele parti egale cu zero */
    {
        real = x; imag = y;
    }
    ...
};
```

Exemple de instanțieri ale clasei *complex*:

```
complex z;           // se instantiază numarul complex
                     // initializat implicit: z = 0+0*i
complex r(3);       // se instantiază numarul complex
                     // r = 3+0*i
complex i(0,1);     // se instantiază numarul complex
                     // i = 0+1*i
complex c(1.5,-1.5); // se instantiază numarul complex
                     // c = 1,5-1,5*i
```

La initializare, se utilizează regulile de la apelurile funcțiilor supraincarcate dacă există mai mulți constructori. Dacă există un singur constructor, atunci se aplică regula de la apelurile funcțiilor din limbajul C, adică parametrii efectivi (în cazul de față expresiile prin care se face initializarea) se convertește spre tipurile parametrilor formali corespunzători ai constructorului.

Această reglă se aplică și în cazul clasei *complex*, de mai sus, care are un singur constructor. Mai jos, dăm un exemplu de clasă cu mai mulți constructori:

#### Exemplu:

```
class dc {
    int zi,luna,an;
public:
    dc() // constructor fără parametri
    {
        zi = 1;
        luna = 1;
        an = 1600;
    }

    dc(int z,int l,int a=1995)
    /* constructor cu trei parametri de tip int */
    {
        zi = z;
        luna = l;
        an = a;
    }

    dc(int z,char *den1, int a);
    /* constructor pentru initializare cu denumirea lunii calendaristice */

    ...
};
```

Exemple de instanțieri ale clasei *dc*:

```
dc d1; // se apelează constructorul fără parametri
        // d1: zi=1, luna=1, an=1600
dc d2(15,9,1995); // se apelează constructorul cu toți parametri de tip int
                    // d2: zi=15, luna=9, an=1995
dc d3(15,9); // se apelează același constructor ca la d2 și se obține același rezultat
dc d4(15,"septembrie",1997);
/* se apelează constructorul cu parametrul char *den1 */
dc *pd = new dc(15,9);
/* se apelează constructorul cu și în cazul obiectelor d2 și d3;
   obiectul se aloca în memoria heap, datele membru se initializează ca la obiectele d2 și d3;
   pd are ca valoare adresa de început a obiectului alocat în memoria heap */
```

În acest exemplu s-a definit un constructor fără parametri. Un astfel de constructor se numește *constructor implicit*.

În cazul în care există un constructor implicit nu se mai poate defini, pentru clasa respectivă, un constructor cu toți parametrii impliciti. Într-adevăr, un astfel de constructor conduce la ambiguitate la instanțierea obiectelor. De exemplu, dacă alături de constructorul *dc()* al clasei *dc*, am defini constructorul:

```
dc(int z=1,int l=1,int a=1600)
```

atunci instanțierea:

```
dc d;
```

este ambiguă, deoarece se pot apela ambii constructori.

Prezența constructorilor nu este obligatorie. Se pot defini clase și fără

constructori. În acest caz, compilatorul C++ generează în mod automat un constructor fără parametri, adică un constructor implicit. Acesta are rol numai de alocare a obiectelor clasei respective. Constructorii definiti de programator sunt necesari numai în cazul în care se dorește inițializarea obiectelor la instanțierea lor.

In cazul în care o clasă are cel puțin un constructor și nici unul nu este constructorul implicit, atunci nu se pot instanția obiecte neinițializate. Aceasta, deoarece compilatorul nu creează constructorul implicit pentru clasele care au cel puțin un constructor.

#### Exemplu:

```
class complex {  
    double real;  
    double imag;  
public:  
    complex(double x,double y)  
    {  
        real = x; imag = y;  
    }  
};
```

În acest caz, se pot instanția numai obiecte cu ambele părți inițializate:

```
complex z(1,2);  
complex z1; // eroare: nu există constructor implicit
```

Declarația (definiția) obiectelor pentru care lista de inițializare se reduce la un singur parametru, poate fi scrisă într-un format care să nu difere de cel utilizat la inițializarea variabilelor simple. Astfel:

nume\_clasa nume\_object = expresie;

realizează instanțierea obiectului nume\_object al clasei nume\_clasa, instanțiere la care se apelează un constructor pentru care primul parametru are ca valoare, valoarea lui expresie. Alți parametri, sau nu există la constructorul apelat sau sunt parametri implicați.

#### Exemplu:

```
class complex {  
    double real;  
    double imag;  
public:  
    complex(double x=0,double y=0)  
    {  
        real = x; imag = y;  
    }  
};
```

Exemple de instanțieri:

```
complex z;           // z = 0+0*i  
complex z1(1);      // z1 = 1+0*i ] |  
complex z2 = 1;      // z2 = 1+0*i ] *  
complex z3(1,2);    // z3 = 1+2*i
```

În cazul în care dorim să instanțiem obiecte atât inițializate, cit și neinițializate, putem folosi un constructor implicit vid, care se va apela la instanțierea obiectelor neinițializate.

#### Exemplu:

```
class dc {  
    int zi,luna,an;  
public:  
    dc() // constructor implicit vid  
    { // se utilizează pentru instanțierea obiectelor neinițializate  
    }  
    dc(int z,int l,int a)  
    /* constructor utilizat pentru inițializarea obiectelor */  
    {  
        zi = z; luna = l; an = a;  
    }  
    ...  
};
```

Se pot utiliza instanțieri de forma:

```
dc d; // se apelăza constructorul implicit  
dc d1(1,2,1997); // se instanțiază obiectul d1 inițializat  
// astfel: zi=1, luna=2, an=1997
```

Parametrii unui constructor pot fi de orice tip, cu excepția tipului definit de clasa pentru care este funcție membru. Deci, dacă clasa este numele unei clase, atunci nu se poate defini un constructor de astfel:

clasa(clasa p)

În schimb, constructorul unei clase poate avea ca parametri pointeri sau referințe la obiectele clasei respective.

Deci:

```
clasa(clasa *p)  
și  
clasa(clasa& p) ] *
```

sunt antetele corecte de constructori.

Constructorul:

```
| clasa(const clasa& p) |
```

este un constructor special care permite copierea obiectelor. El se numește constructor de copiere. Constructorul de copiere poate avea și alți parametri care însă trebuie să fie implicați.

Constructorul de copiere se apelează într-o instanțiere de felul celei de mai jos:

```
* clasa c(...); // instantiere cu initializare obisnuită
  clasa c1 = c; // se apelează constructorul de copiere;
  // c1 este o copie a lui c
```

Dacă programatorul nu definește un constructor de copiere, atunci compilatorul generează un constructor de copiere implicit, dacă este nevoie de el.

### Exemplu:

Mai jos, definim un constructor de copiere pentru numerele complexe.

```
class complex {
    double real;
    double imag;
public:
    complex(double x=0,double y=0)
    {
        real = x;
        imag = y;
    }
    complex(const complex& c) // constructor de copiere
    {
        real = c.real;
        imag = c.imag;
    }
    ...
};
```

Exemple de instantieri:

```
complex z(1,2); // z = 1+2*i
complex z1 = z; // se apelează constructorul de copiere
// z1 = 1+2*i
* complex z2(z); // se apelează constructorul de copiere
// z2 = 1+2*i
```

Datele membru ale unei clase pot fi date arbitrară dar nu obiecte ale clasei respective. În particular, datele membru pot fi obiecte ale unei alte clase.

Un exemplu simplu este cel oferit de primitivele care se utilizează la definirea figurilor pe ecranul setat în mod grafic.

Considerăm clasa *Punct* care definește un punct pe ecran. Acesta are o poziție definită prin coordonatele sale (coloana și linia în care se afișează punctul respectiv). Definim clasa *Punct* ca mai jos:

```
class Punct {
    int x; // coloana
    int y; // linia
public:
    Punct(int col=0,int linia = 0)
    // constructor: punctul implicit este cel de coordonate (0,0)
```

```
{
    x = col; y = linia;
}
...
```

Un dreptunghi se poate trasa dacă se definesc două virfuri diagonale opuse.

De obicei, se consideră virful din stînga sus al dreptunghiului și cel din dreapta jos. Un virf al dreptunghiului este un obiect al clasei *Punct*. De aceea, putem defini clasa *Dreptunghi* cu ajutorul a două obiecte ale clasei *Punct*:

```
class Dreptunghi {
    Punct st_sus;
    Punct dr_jos;
}
...
```

Constructorul clasei *Dreptunghi* trebuie să transfere valori pentru parametrii constructorului clasei *Punct*.

Acest lucru se realizează modificind antetul constructorului, ca mai jos.

Fie clasa *cl* definită astfel:

```
class cl {
    cl1 c1;
    cl2 c2;
    ...
    cln cn;
    ...
};
```

unde:

*c1, c2, ..., cn*

- Sunt nume de clase, în prealabil definite, care nu neapărat sunt toate distincte. Ele, au fiecare, constructori pentru inițializarea obiectelor.

Constructorul *cl* transferă valorile de inițializare pentru obiectele membru *c1, c2, ..., cn* prin antetul său de format:

```
* cl(...); c1(...),c2(...),...,cn(...)
```

În acest antet vor lipsi obiectele pentru care nu se transferă date de inițializare.

Reluind exemplul de mai sus, completăm definiția clasei *Dreptunghi* cu doi constructori:

```
class Dreptunghi {
    Punct st_sus;
    Punct dr_jos;
public:
    Dreptunghi(int dr,int j): dr_jos(dr,j)
    // se instantiază dreptunghiul cu virfurile:
    //     stînga sus: (0,0) -implicit
```

```

// dreapta jos: (dr,j)
{
}
Dreptunghi(int st,int sus,int dr,int jos):
    st_sus(st,sus),dr_jos(dr,jos)
// se instantiaza dreptunghiul cu virfurile:
//      stinga sus: (st,sus)
//      dreapta jos: (dr,jos)
{
}
}

;

```

Metoda de inițializare a obiectelor, care sunt date membru ale unei clase, poate fi utilizată și pentru date membru care nu sunt obiecte.

Aștept, un constructor de forma:

```

complex(double x,double y)
{
    real = x; imag = y;
}

```

Poate fi scris și sub forma:

```

complex(double x,double y):real(x),imag(y)
{
}

```

În încheierea acestui paragraf, enumerăm caracteristicile de bază ale constructorilor.

1) Constructorii nu returnează nici o valoare la revenirea din ei. Mai mult decit atât, antetul lor constituie o excepție, deoarece nu este admis cuvintul cheie void, cuvint care trebuie să fie prezent în antetul funcțiilor care nu returnează o valoare.

2) În cazul în care o clasă are obiecte membru care au constructori, aceștia se apelează înainte de a se apela constructorul clasei respective.

3) Obiectele claselor care au cel puțin un constructor nu pot fi componente ale unei reuniuni.

4) Spre deosebire de funcțiile obișnuite, adresa constructorului nu se poate determina.

5) De obicei, constructorii sunt funcții membru publice, dar nu se pot apela explicit la fel ca celelalte funcții membru.

Un constructor poate fi apelat explicit pentru situații de felul celui de mai jos:

complex z = complex(1,2); *(Fiecare din părți, de către)*

În acest caz a fost apelat constructorul clasei complex pentru a crea un obiect anonim care este inițializat cu valorile 1 pentru partea reală și 2 pentru partea imaginară. Apoi, obiectul respectiv este copiat în obiectul z.

## 22.9. Destructor

Destructorii sunt funcții care pot fi considerați ca actionează în sens invers față de constructori. Ei au multe caracteristici în comun cu constructorii, dar între ei există și diferențe.

Numele unui destructor este numele clasei precedat de caracterul "~". Un destructor nu are parametri și el este unic pentru o clasă. Dacă programatorul nu a definit un destructor, atunci compilatorul generează un destructor pentru clasa respectivă.

Antetul destructorilor nu conține cuvântul void, deși ei nu returnează valori. De aceea, destructorii au antetul:

~nume\_clasa()

în interiorul definiției clasei nume\_clasa.

În afară definiției clasei, antetul destructorului va fi:

nume\_clasa :: ~nume\_clasa()

Că și în cazul constructorilor, adresa destructorului nu poate fi determinată.

Obiectele unei clase care au un destructor și/sau cel puțin un constructor nu se pot utiliza ca și componente ale unei reuniuni.

Dacă la instantierea unui obiect s-au apelat mai mulți constructori, atunci la distrugerea lui, destructorii se vor apela în ordine inversă.

Pentru un obiect global, destructorul se apelează ca parte a procedurii exit de la terminarea execuției funcției main. De aceea, un astfel de destructor nu trebuie să apeleze funcția exit deoarece se intră într-un ciclu infinit.

Pentru un obiect local, destructorul se apelează cind controlul programului ieșe din domeniul lui (se ieșe din blocul în care este declarat). \*

Obiectele dinamice nu pot fi distruse automat. Distrugerea se realizează de către programator deoarece numai el știe cind un astfel de obiect nu mai este necesar.

Destructorul poate fi apelat de programator atât direct, cât și indirect prin intermediul operatorului delete.

Noi utilizăm operatorul delete pentru a distruge obiecte create dinamic cu ajutorul operatorului new (obiecte dinamice).

**Exemple:**

1. Fie tipul complex definit în exemplele precedente.

```

complex *pz;
...
pz = new complex(1,2);
// în memoria heap se construiește un obiect de tip complex cu partea reală egală
// cu 1 și partea imaginară egală cu 2; pz pointează spre obiectul respectiv
...

```

```
    delete pz; // se distrug obiectul creat mai sus prin new
```

## 2. Se definește clasa *string* ca mai jos:

```
class string {
    char *sir;
    int lung;
public:
    string(char *);
    ~string();
    ...
};

string :: string(char *s) // definitia constructorului
{
    lung = strlen(s) + 1;
    sir = new char[lung];
    strcpy(sir,s);
}

string :: ~string() // definitia destructorului
{
    delete sir;
}
```

Exemple de instanțieri:

```
string sir_de_caractere("Acesta este un sir de caractere");
// se creeaza obiectul sir_de_caractere in memoria heap si se initializeaza cu textul :
// "Acesta este un sir de caractere"
...
delete sir_de_caractere; // se distrug obiectul apelindu-se indirect destructorul
```

Apelul direct al destructorului se poate face numai dacă numele lui este precedat de numele clasei care este urmat de operatorul de rezoluție:

```
...
string s("exemplu");
...
s.string :: ~string(); // apel direct al destructorului
```

## Exerciții:

22.1 Să se definească tipul abstract de date *complex* care să aibă funcții membru pentru modul, argument, pentru accesul la partea reală și imaginără a obiectelor de tip complex, pentru afișarea obiectelor complexe, precum și constructori pentru inițializare și copiere.

## FIŞIERUL BXXIII1.H

```
class complex {
    // date membru protejate(private)
    double real; // partea reala
    double imag; // partea imaginara

public: // functii membru neprotejate
    complex(double x=0,double y=0);
```

```
// constructor folosit la initializare
complex(const complex&);

// constructor de copiere

double modul(); // modulul numarului complex
double arg(); // argumentul numarului complex
double retreal(); // returneaza partea reala
double retimag(); // returneaza partea imaginara
void afiscomplex(); // afiseaza numarul complex
}; // sfirsit definitie clasa complex
```

## FIŞIERUL BXXIII1

```
#ifndef __BXXIII1_H
#include "BXXIII1.H"
#define __BXXIII1_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif

// constructor pentru initializarea obiectelor
inline complex :: complex(double x,double y)
{
    real = x;
    imag = y;
}

// constructor de copiere
inline complex :: complex(const complex& z)
{
    real = z.real; imag = z.imag;
}

inline double complex :: modul() // calculeaza modulul numarului complex
{
    return sqrt(real*real+imag*imag);
}

double complex :: arg() // calculeaza argumentul numarului complex
{
    double a;

    if(real==0 && imag==0) return 0.0;
    if(imag==0)
        if(real > 0) return 0.0;
        else return PI;
    if(real==0)
        if(imag > 0) return PI/2;
```

```

        else return (3*PI)/2;

// real != 0 si imag != 0
a = atan(imag/real);
if(real < 0) // real < 0 si y != 0
    return a+PI;
else // real > 0
    if(imag < 0) // real > 0 si imag < 0
        return 2*PI+a;

// real > 0 si imag > 0
return a;
} // sfarsit arg

inline double complex :: retreal()
// returneaza partea reala a numarului complex
{
    return real;
}

inline double complex :: retimag()
// returneaza partea imaginara a numarului complex
{
    return imag;
}

inline void complex :: afiscomplex()
// afiseaza numarul complex
{
    printf ("%g+i* (%g)\n", real, imag);
}

```

22.2 Sa se scrie un program care realizeaza următoarele:

- citește perechi de numere care reprezintă, fiecare, partea reală și respectiv partea imaginără a unui număr complex;
- afișează:
  - numărul complex citit;
  - rădăcina patrată din fiecare număr complex citit;
  - suma numerelor complexe citite.

## PROGRAMUL BXXII2

```

#include <stdio.h>
#include <conio.h>
#include "BXXIII1.CPP"

main()
/* - citeste perechi de numere care reprezinta partea reala si respectiv cea imaginara a unui
numar complex;
- afiseaza:
- numarul complex citit;
- radacina patrata din numarul complex citit;

```

```

- suma numerelor complexe citite.
*/
{
    double x;
    double y;
    double sx = 0;
    double sy = 0;

    do {
        printf("partea reala =");
        if(scanf("%lf",&x) != 1) break; // nu mai sunt numere
        printf("partea imaginara =");
        if(scanf("%lf",&y) != 1){ // eroare
            printf("partea imaginara eronata\n");
            printf("se reia citirea numarului\n");
            fflush(stdin); // videaza zona tampon de la intrarea standard
            continue; // se reia ciclul de citire
        }
        // se insumeaza partea reala si cea imaginara a numerelor complexe citite
        sx += x; sy += y;

        complex z(x,y); // se construieste numarul complex avind partile citite;
        // se apeleaza constructorul care realizeaza initializarea obiectului
        // z: z=x+i*y
        z.afiscomplex(); // afiseaza numarul citit
        double m = z.modul(); // calculeaza modulul lui z
        double a=z.arg(); // calculeaza argumentul
        // numarului complex
        double m1 = sqrt(m); // radacina patrata din modulul z
        double a1 = a/2; // semicircularul numarului z

        // rz1 = sqrt(z)
        printf("modulul = %g\targumentul = %g\n",m,a);
        double preal = m1*cos(a1);
        double pimag = m1*sin(a1);
        complex rz1 = complex(preal,pimag);

        // rz2 = sqrt(z);
        complex rz2 = complex(-preal,-pimag);
        printf("radacina patrata\n");
        printf("sqrtz1 = ");
        rz1.afiscomplex();
        printf("sqrtz2 = ");
        rz2.afiscomplex();
    } while(1);
    complex sz = complex(sx,sy); // sz = suma
    printf("suma numerelor complexe citite = ");
    sz.afiscomplex(); // afiseaza suma numerelor complexe citite
}

```

22.3 Să se extindă tipul abstract complex, definit în exercițiul 22.1, astfel încât să se poată realiza următoarele operații asupra obiectelor complexe:

- adunare;

- scădere;
- negativare;
- înmulțire;
- împărțire;
- citirea de la intrarea standard a componentelor unui obiect complex.

## FIŞIERUL BXXII3.H

```
enum Boolean {false,true};
class complex {
    // date membru protejate (private)
    double real;
    double imag;
public: // functii membru neprotejate
    complex(double x=0,double y=0); // constructor
    complex(const complex&); // constructor de copiere
    double modul(); // modulul numarului complex
    double arg(); // argumentul numarului complex
    double retreal(); // returneaza partea reala
    double retimag(); // returneaza partea imaginara
    void afiscomplex(); // afiseaza numarul complex

    // functii membru noi

    Boolean citcomplex(); // citeste componentele numarului complex;
    // returneaza false la EOF
    void adcomplex(complex *z1,complex *z2);
    // calculeaza z = z1+z2
    void sccomplex(complex *z1,complex *z2);
    // calculeaza z = z1-z2
    void negcomplex(complex *z1);
    // calculeaza z = -z1
    void mulcomplex(complex *z1,complex *z2);
    // calculeaza z = z1*z2
    Boolean divcomplex(complex *z1,complex *z2);
    // calculeaza z = z1/z2; returneaza false la impartirea cu zero
};
```

Fișierul de mai jos, conține definițiile funcțiilor membru noi.  
Definițiile funcțiilor membru vechi se preiau din fișierul BXXII1.CPP.

## FIŞIERUL BXXII3

```
#ifndef __BXXII3_H
#include "BXXII3.H"
#define __BXXII3_H
#endif
#define __BXXII1_H
#ifndef __BX18_CPP
#include "BX18.CPP"
#define __BX18_CPP
#endif
```

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif

// se defineste BXXII1_H pentru a nu mai include fisierul
// BXXII1.H prin includerea fisierului BXXII1.CPP

#include "BXXII1.CPP" // se preiau definitiile functiilor membru vechi

// definitiile functiilor membru noi

Boolean complex :: citcomplex()
/* - citeste componentele numarului complex;
   - returneaza:
       false la sfarsitul fisierului;
       true altfel.
*/
{
    double preal;
    double pimag;

    if(pcit_double("Partea reala:",&preal) == 0)
        return false; //EOF
    if(pcit_double("Partea imaginara: ",&pimag) == 0)
        return false; //EOF
    real = preal;
    imag = pimag;
    return true;
}

inline void complex :: adcomplex(complex *z1, complex *z2)
// calculeaza z = z1+z2
{
    real = z1 -> real+z2 -> real;
    imag = z1 -> imag+z2 -> imag;
}

inline void complex :: sccomplex(complex *z1, complex *z2)
// calculeaza z = z1-z2
{
    real = z1 -> real - z2 -> real;
    imag = z1 -> imag - z2 -> imag;
}

inline void complex :: negcomplex(complex *z1)
// calculeaza z = -z1
{
    real = -z1 -> real;
    imag = -z1 -> imag;
}
```

```

inline void complex :: mulcomplex(complex *z1, complex *z2)
// calculeaza z=z1*z2
{
    real = z1 -> real*z2 -> real - z1 -> imag*z2 -> imag;
    imag = z1 -> real*z2 -> imag + z1 -> imag*z2 -> real;
}

Boolean complex :: divcomplex(complex *z1, complex *z2)
// calculeaza z=z1/z2
{
    double d = z2 -> real *z2 -> real + z2 -> imag*z2 -> imag;
    if(d == 0) //divizor nul
        return false;
    real = (z1 -> real*z2 -> real + z1 -> imag*z2 -> imag)/d;
    imag = (z1 -> imag*z2 -> real - z1 -> real*z2 -> imag)/d;
    return true;
}

```

22.4 Sa se scrie un program care citește numerele complexe  $a, b, c$ , rezolvă și afișează radacinile ecuației de gradul 2:

$$a*x^2 + b*x + c = 0$$

#### PROGRAMUL BXXII4

```

#include <stdlib.h>
#ifndef __STDLIB_H
#include <stdio.h>
#define __STDLIB_H
#endif
#include "BXXII3.CPP"

main()
/* - citește: a,b,c
   - rezolvă ecuația: a*x^2+b*x+c=0
   - afișează soluțiile ecuației
*/
{
    complex a,b,c;
    char er[] = "s-a tastat EOF\n";
    if(a.citcomplex() == false){
        printf(er);
        exit(1);
    }
    if(b.citcomplex() == false){
        printf(er);
        exit(1);
    }
    if(c.citcomplex() == false){
        printf(er);
        exit(1);
    }
    if(a.retreal() == 0 && a.retimag() == 0 &&
       b.retreal() == 0 && b.retimag() == 0 &&

```

```

       c.retreal() == 0 && c.retimag() == 0) {
        printf("ecuatie nedeterminata\n");
        exit(0);
    }
    if(a.retreal() == 0 && a.retimag() == 0 &&
       b.retreal() == 0 && b.retimag() == 0) {
        printf("ecuatie nu are solutie\n");
        exit(1);
    }
    if(a.retreal() == 0 && a.retimag() == 0){
        printf ("ecuatie de gradul 1\n");
        complex x,z;
        z.divcomplex(&c,&b); // z=c/b
        x.negcomplex(&z); // x=-z
        printf("x=");
        x.afiscomplex();
        exit(0);
    } // obiectele x si z se distrug in acest punct
    //ecuatie de gradul 2

    complex bp;
    bp.mulcomplex(&b,&b); // bp=b*b

    complex patru(4,0);
    complex patrua;
    patrua.mulcomplex(&a,&patru); // 4*a

    complex patruac;
    patruac.mulcomplex(&patrua,&c); // 4*a*c

    complex deltap;
    deltap.sccomplex(&bp,&patruac); // b*b-4*a*c
    double r = deltap.modul();
    r = sqrt(r);
    double argument = deltap.arg();
    argument = argument/2;

    complex delta(r*cos(argument),r*sin(argument));
    // delta=sqrt(b*b-4*a*c)

    complex mb;
    mb.negcomplex(&b); // mb=-b

    complex doia;
    doia.adcomplex(&a,&a); // doia=a+a

    complex x;
    x.adcomplex(&mb,&delta); // x=-b+delta

    complex x1;
    x1.divcomplex(&x,&doia); // x1=(-b+delta)/(2*a)
    printf("x1 = ");
    x1.afiscomplex(); // afiseaza radacina x1

```

```

complex x2;
    x.sccomplex(&mb,&delta); //x = -b-delta
    x2.divcomplex(&x,&doia); //x2 = (-b-delta)/(2*a)
    printf("x2 = ");
    x2.afiscomplex(); //afiseaza radacina x2
}

```

22.5 Să se definească tipul abstract *dc* pentru implementarea datei calendaristice. În acest scop se definește clasa *dc* care are următoarele componente de tip *int*:

- *zi, luna, an*;
- *minzz, minll, minaa* - pentru data calendaristică considerată ca dată validă minimă;
- *maxzz, maxll, maxaa* - pentru data calendaristică considerată ca dată validă maximă;
- *tnez* - este un tablou ale cărui elemente definesc numărul de zile ale lunilor calendaristice (luna februarie se consideră că are 28 de zile).

Componentele *minzz, minll, minaa, maxzz, maxll, maxaa* și *tnez* nu se multiplică la fiecare instanțiere a clasei *dc*, ele fiind utilizate în comun de către funcțiile membru. De aceea, ele se definesc ca date membru statice.

În mod implicit, data calendaristică minimă se consideră 1 ianuarie 1600, iar cea maximă se consideră 31 decembrie 4900.

Clasa *dc* are 3 funcții pentru verificarea corectitudinii datelor calendaristice. Una este o funcție membru obișnuită care verifică data calendaristică a obiectului curent, iar celelalte două verifică corectitudinea datelor calendaristice definite de datele membru statice *minzz, minll, minaa, maxzz, maxll, și maxaa*. Aceste două funcții sunt funcții membru statice. Aceste 3 funcții apelează o funcție membru statică *v\_calend* care validează o dată calendaristică definită prin parametrii ei. Faptul că o dată calendaristică este dintr-un an bisect sau nu, se determină folosind trei funcții, una pentru obiectele de tip *dc* și două pentru cele două date membru statice. Aceste funcții folosesc în comun o funcție *bisect* care stabilește dacă data calendaristică definită de parametrii ei este o dată dintr-un an bisect sau nu.

Funcția *bisect* este o funcție obișnuită. Ea se definește în același fișier cu funcțiile membru ale clasei.

Clasa are un constructor implicit care se utilizează la instanțierea obiectelor fară inițializarea datei calendaristice nestatice (*zi, luna, și an*).

Amintim că datele statice se inițializează independent, în afara constructorilor. Constructorii clasei *dc* vor verifica corectitudinea datelor calendaristice statice.

În caz de eroare, se afișează un mesaj corespunzător și se forțează datele implicate.

Pentru instanțierea obiectelor inițializate se utilizează un constructor cu trei parametri pentru cele 3 date membru care nu sunt statice (*zi, luna și an*).

Constructorul utilizat în acest scop, verifică atât datele statice, cit și cele care

se inițializează prin constructorul respectiv.

În caz de eroare, se afișează un mesaj corespunzător și se instanțiază un obiect cu data calendaristică minimă.

Alte funcții membru:

- constructor de copiere;
- *retzi*: returnează ziua din data obiectului curent;
- *retluna*: returnează luna din data obiectului curent;
- *retan*: returnează anul din data obiectului curent;
- *afisdata*: afișează data calendaristică a obiectului curent;
- *datamin*: returnează data minimă;
- *datamax*: returnează data maximă;
- *modifmin*: modifică data minimă;
- *modifmax*: modifică data maximă;
- *citdata*: citește o dată calendaristică;
- *adzi*: adună un număr de zile la data obiectului curent;
- *disdata*: determină diferența, în zile, dintre două date calendaristice;
- *zi\_din\_an*: returnează ziua din an pentru data obiectului curent;
- *ziua\_si\_luna*: determină luna și ziua din lună din parametrii an și ziua din an;
- *nr\_zile\_luna*: returnează numărul de zile din luna calendaristică a obiectului curent;
- *verif\_min\_max*: verifică datele minimă și maximă; dacă o dată este eronată, se dă un mesaj de eroare și se forțează data implicită: 1 ianuarie 1600 pentru data minimă și 31 decembrie 4900 pentru data maximă.

## FIŞIERUL BXXII5.H

```

#ifndef __Boolean
#define __Boolean
enum Boolean {false,true};
#endif

class dc {
// date membru protejate
    int zi,luna,an;
    static int minzz,minll,minaa;
    static int maxzz,maxll,maxaa;
    static int tnez[13];
    static char *tdenluna[13];
    static char *ermin;
    static char *ermax;
    static char *erdic;
    static void afiser(char *sir);
}

```

```

// afiseaza textul spre care pointeaza sir
static void verif_min_max();

// verifica datele minima si maxima; la eroare se da mesaj
// si se forteaza data implicita corespunzatoare

public:
    Boolean valid_dc();
    /* verifica corectitudinea datei calendaristice a obiectului curent;
     * la eroare returneaza false */

    static Boolean valid_dc_min();
    /* verifica corectitudinea datei minime; la eroare returneaza false */

    static Boolean valid_dc_max();
    /* verifica corectitudinea datei maxime; la eroare returneaza false */

    Boolean bisect_dc();
    /* returneaza:
        true - daca data obiectului curent este dintr-un an bisect;
        false - altfel.
    */

    static Boolean bisect_dc_min();
    /* returneaza:
        true - daca data minima este dintr-un an bisect;
        false - altfel
    */

    static Boolean bisect_dc_max();
    /* returneaza:
        true - daca data maxima este dintr-un an bisect;
        false - altfel.
    */

    static Boolean v_calend(int z,int l,int a);
    /* returneaza:
        true - daca data definita de parametrii z, l si a este valida;
        false - altfel.
    */

    dc(); // constructor implicit pentru obiecte neinitializate

    dc(int z,int l,int a);
    // constructor pentru initializarea obiectelor

    dc(const dc&); // constructor pentru copiere

    int retzi(); // returneaza ziua din data obiectului curent

    int retrluna(); // returneaza luna din data obiectului curent

    int retan(); // returneaza anul din data obiectului curent

```

```

void afisdata(); // afiseaza data obiectului curent

dc *datamin(); // returneaza un pointer spre data minima

dc *datamax(); // returneaza un pointer spre data maxima

void modifmin(); /* schimba data calendaristica minima cu
                    data obiectului curent */

void modifmax(); /* schimba data calendaristica maxima cu
                    data obiectului curent */

int citdata(); /* citeste o data calendaristica;
                    returneaza 0 la sfarsit de fisier si 1 altfel */

Boolean adzi(long z); /* aduna, la data obiectului curent,
                        numarul de zile dat de parametrul z */

long difdata(dc *d);
/* returneaza diferența, in numar de zile, dintre data
   calendaristica a obiectului curent si cea spre care pointeaza d */

char *denluna(); /* returneaza un pointer spre denumirea
                     lunii datei obiectului curent */

int zi_din_an(); /* returneaza ziua din an pentru data obiectului curent */

Boolean ziua_si_luna(int z,int a);
/* determina data calendaristica a obiectului curent din
   z - ziua din an
   si din
   a - anul calendaristic */

int nr_zile_luna(); /* returneaza numarul de zile din
                     luna calendaristica a datei obiectului curent */

}; // sfarsit definitia clasei dc

```

Definițiile funcțiilor membru și inițializarea datelor statice se dau intr-un fișier de extensie .CPP

## FIŞIERUL BXII5

```

#ifndef __STUDIO_H
#include <stdio.h>
#define __STUDIO_H
#endif
#ifndef __Boolean
enum Boolean {false,true};
#define __Boolean
#endif
#ifndef __DC_H
#include "BXII5.H" // definitia clasei dc
#define __DC_H

```

```

#endif

// initializarea datelor statice

int dc::minzz = 1; // implicit, data calendaristica
int dc::minll = 1; // minima este

int dc::mina = 1600; // 1 ianuarie 1600
int dc::maxzz = 31; // implicit, data calendaristica
int dc::maxll = 12; // maxima este
int dc::maxaa = 4900; // 31 decembrie 4900

// initializarea tabloului cu numarul zilelor din lunile calendaristice
int dc::tnrz[13] = {0,31,28,31,30,31,30,31,31,30,31,30,31};

// initializarea textelor de eroare
char *dc::erdc = "data calendaristica eronata\n";
char *dc::ermin = "data minima eronata\n";
char *dc::ermax = "data maxima eronata\n";

// initializarea tabloului cu denumirile lunilor calendaristice
char *dc::tdenluna[13] = {
    "luna ilegală",
    "ianuarie",
    "februarie",
    "martie",
    "aprilie",
    "mai",
    "iunie",
    "iulie",
    "august",
    "septembrie",
    "octombrie",
    "noiembrie",
    "decembrie"
};

// functie obisnuita folosita de functiile membru

inline int bisect(int a)
/* returneaza:
   1 daca a defineste un an bisect;
   0 altfel.
*/
{
    return a%4==0 && a%100 || a%400 == 0;
}

// functii membru

Boolean dc::v_calend(int z,int l,int a)
/* returneaza:
   true daca data calendaristica definita de parametrii z,l si a este valida;

```

```

false in caz contrar. */
{
    if(a < 1600 || a > 4900) return false;
    if(a < dc::mina || a > dc::maxaa) return false;
    if(l < 1 || l > 12) return false;
    if(a==dc::mina && l<dc::minll || a==dc::maxaa && l>dc::maxll)
        return false;
    if(z < 1 || z > tnrz[l]+(l==2 && bisect(a))) return false;
    if(a==dc::mina && l==dc::minll && z < dc::minzz ||
       a==dc::maxaa && l==dc::maxll && z > dc::maxzz)
        return false;
    return true;
}

inline void dc::afiser(char *sir)
// afiseaza textul spre care pointaza sir
{
    printf("%s\n",sir);
}

inline Boolean dc::valid_dc()
/* - verifica corectitudinea datei calendaristice a obiectului curent;
   - la eroare returneaza false. */
{
    return v_calend(zi,luna,an);
}

inline Boolean dc::valid_dc_min()
/* - verifica corectitudinea datei minime;
   - la eroare returneaza false. */
{
    return v_calend(dc::minzz,dc::minll,dc::mina);
}

inline Boolean dc::valid_dc_max()
/* - verifica corectitudinea datei maxime;
   - la eroare returneaza false. */
{
    return v_calend(dc::maxzz,dc::maxll,dc::maxaa);
}

Boolean dc::bisect_dc()
/* returneaza true daca anul calendaristic al obiectului curent este bisect */
{
    if(bisect(an)) return true;
    else return false;
}

Boolean dc::bisect_dc_min()
/* returneaza true daca anul minim este bisect */
{
    if(bisect(dc::mina)) return true;
    else return false;
}

```

```

Boolean dc::bisect_dc_max()
/* returneaza true daca anul maxim este bisect */
{
    if(bisect(dc::maxaa)) return true;
    else return false;
}

void dc::verif_min_max()
/* - verifică datele minima și maxima;
   - daca o data este eronata se da un mesaj de eroare si se forțează data implicită corespunzătoare.
*/
{
    if(dc::valid_dc_min() == false){
        dc::afiser(dc::ermin); // se forțează data minima implicită
        dc::minzz = 1;
        dc::minll = 1;
        dc::mina = 1600;
    }
    if(dc::valid_dc_max() == false){
        dc::afiser(dc::ermax); // se forțează data maxima implicită
        dc::maxzz = 31;
        dc::maxll = 12;
        dc::maxaa = 4900;
    }
}

dc::dc(int z,int l,int a)
/* - constructor pentru initializarea obiectelor;
   - se verifică data minima, maxima și cea care se instantiază. */
{
    dc::verif_min_max();

    // initializarea obiectului care se instantiază

    zi = z;
    luna = l;
    an = a;
    if(valid_dc()==false){
        afiser(dc::erdc);
        zi = dc::minzz;
        luna = dc::minll;
        an = dc::mina;
    }
}

inline dc::dc() /* constructor pentru obiecte neinitializate */
{
    dc::verif_min_max();
}

dc::dc(const dc &d) /* constructor de copiere */
{
    zi = d.zi;
    luna = d.luna;
}

```

```

an = d.an;
if(valid_dc() == false){
    afiser(dc::erdc);
    zi = dc::minzz;
    luna = dc::minll;
    an = dc::mina;
}

inline int dc::retzi() // returnează ziua obiectului curent
{
    return zi;
}

inline int dc::retluna() // returnează luna obiectului curent
{
    return luna;
}

inline int dc::retan() // returnează anul obiectului curent
{
    return an;
}

inline void dc::afisdata() // afisează data obiectului curent
{
    printf("zi: %d\\tluna: %d\\tan: %d\\n", zi,luna,an);
}

dc *dc::datamin() // returnează un pointer spre data minima
{
    dc *d_min = new dc(dc::minzz,dc::minll,dc::mina);
    return d_min;
}

dc *dc::datamax() // returnează un pointer spre data maxima
{
    dc *d_max = new dc(dc::maxzz,dc::maxll,dc::maxaa);
    return d_max;
}

void dc::modifmin() // schimba data calendaristica minima cu data obiectului curent
{
    dc::minzz = zi;
    dc::minll = luna;
    dc::mina = an;
    dc::verif_min_max();
}

void dc::modifmax() // schimba data calendaristica maxima cu data obiectului curent
{
    dc::maxzz = zi;
    dc::maxll = luna;
    dc::maxaa = an;
}

```

```

    dc::verif_min_max();
}

int dc::citdata()
// citeste o data calendaristica;
// returneaza:
//   0 la sfarsit de fisier;
//   1 altfel.
{
    int i,c;

    for(;;){
        for(;;){
            printf("ziua: ");
            if((c=scanf("%d",&i)) == 1 && i > 0 && i <= 31)
                break;
            if(c == EOF) return 0;
            printf("ziua eronata\n");
            fflush(stdin);
        }
        zi=i;
        for(;;){
            printf("luna: ");
            if((c=scanf("%d",&i)) == 1 && i > 0 && i <= 12)
                break;
            if(c == EOF) return 0;
            printf("luna eronata\n");
            fflush(stdin);
        }
        luna=i;
        for(;;){
            printf("anul: ");
            if((c=scanf("%d",&i)) == 1 && i >= 1600 && i <= 4900)
                break;
            if(c == EOF) return 0;
            printf("anul eronat\n");
            fflush(stdin);
        }
        an=i;
        if(valid_dc() == true) break;
        printf("data calendaristica: %d /%d/%d\n"
               "eronata\n", zi,luna,an);
    }
    return 1;
}

int dc::zi_din_an()
// returneaza ziua din an pentru data obiectului curent
{
    int b = bisect(an);
    int z=zi;

    for(int i=1;i < luna;i++) z +=dc::tnrz[i]+(i==2&&b);
    return z;
}

```

```

    }

Boolean dc::ziua_si_luna(int z,int a)
// determina data calendaristica din:
//   z - ziua din an
// si din
//   a - anul
{
    int b = bisect(a);
    if(z > 365+b){ // ziua din an eronata
        printf("ziua = %d eronata\n",z);
        return false;
    }
    if(a < 1600 || a > 4900) // anul eronat
    {
        printf("anul = %d eronat\n",a);
        return false;
    }
    int i = 1;
    do{
        int j = dc::tnrz[i]+(i==2 && b);
        if(z <= j) break;
        z -= j;
        i++;
    }while(1);
    zi = z; luna = i; an = a;
    return valid_dc();
}
inline char *dc::denluna()
// returneaza un pointer spre denumirea lunii obiectului curent
{
    return luna<1 || luna>12 ? dc::tdenluna[0] :
                           dc::tdenluna[luna];
}

Boolean dc:: adzi(long z)
/* aduna la data obiectului curent numarul de zile dat de parametrul z */
{
    long totzile = zi_din_an();
    totzile += z; /* numarul total de zile care se considera dupa
                    31 decembrie din anul precedent celui curent */

    int ziledinan = 365 + bisect(an);
    if(totzile >= 0)
        while(totzile >= ziledinan) { /* se socotesc anii */
            totzile -= ziledinan;
            an++;
            if( an <= 4900) ziledinan = 365+bisect(an);
        }
    else{
        an--;
        ziledinan = 365 +bisect(an);
        while(-totzile >= ziledinan){ /* se socotesc anii */
            totzile += ziledinan;
        }
    }
}

```

```

        an--;
        if(an >= 1600) ziledinan = 365+bisect(an);
    }
    totzile += ziledinan; /* determină numarul de zile din anul determinat */
}
if(totzile == 0){
    zi = 31; luna = 12; an--;
    return valid_dc();
}
return ziua_si_luna(totzile,an);
}

long dc::difdata(dc *d)
/* returneaza diferenta, in numar de zile, dintre data
   calendaristica curenta si cea spre care pointeaza d */
{
    long zidinan1 = zi_din_an(); /* numarul de zile din anul curent */
    long zidinan2 = d -> zi_din_an(); /* numarul de zile din anul definit de
                                         pointerul d */
    long difzi = zidinan1 - zidinan2;
    int an1,an2;

    if(an < d -> an) {
        an1 = an;
        an2 = d -> an;
    }
    else{
        an1 = d -> an;
        an2 = an;
    }
    long difzian = 0;
    while(an1 < an2){
        int b = bisect(an1);
        difzian += 365+b;
        an1++;
    }
    if(an < d -> an) difzian = -difzian;
    return difzi+ difzian;
}

```

#### Observații:

- Funcțiile membru statice, de obicei, se apelează calificind numele lor cu numele clasei urmat de operatorul de rezoluție. Mai sus a fost apelată funcția membru statică *v\_calend* fără a respecta acest lucru. Aceasta este posibil dacă nu există definită în același program încă o altă funcție cu același nume și listă a parametrilor formali.  
Pentru a evita astfel de situații se recomandă să se respecte regula cu privire la calificarea numelor funcțiilor membru statice prin numele clasei urmat de operatorul de rezoluție.
- Funcțiile membru care nu sunt statice se apelează numai în legătură cu un obiect al clasei pentru care ele sunt funcții membru. Apelurile unei astfel de

funcții au forma:

*obiect.nume\_functie( ... )*

sau

*pobiect -> nume\_functie( ... )*

unde:

*pobiect* - este un pointer spre tipul implementat prin clasa pentru care *nume\_functie* este funcție membru.

Mai sus, s-au apelat funcții membru nestatic ale clasei *dc* în corpul altor funcții membru nestatic ale aceleiași clase fără a respecta regula de mai sus. De exemplu, în corpul funcției membru *citdata* se apelează funcția membru *valid\_dc* astfel:

```
if(valid_dc() == true) break;
```

Acesta este corect deoarece în mod implicit apelul de mai sus este realizat folosind pointerul *this*:

```
if(this -> valid_dc() == true) break;
```

- 22.6 Să se scrie un program care citește, de la intrarea standard, o succesiune de date calendaristice și afișează fiecare dată calendaristică împreună cu cea a zilei următoare.

#### PROGRAMUL BXXII6

```

#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXXII5.CPP"

main()
/* citeste o succesiune de date calendaristice si afiseaza datele citite
   impreuna cu data zilei urmatoare */
{
    for(;;) {
        dc data_citita;
        if(data_citita.citdata() == 0) break; // s-a intilnit EOF
        data_citita.afisdata(); // afiseaza data citita
        dc data_urm = data_citita; // copiere
        data_urm.adzi(1L); // determină data zilei urmatoare
        data_urm.afisdata(); // afiseaza data zilei urmatoare
    }
}

```

- 22.7 Să se scrie un program care realizează următoarele:

- citește date calendaristice care sunt situate între două date limită;
- afișează fiecare dată citită împreună cu diferența, în număr de zile, dintre

data citită și datele limită.

Cele două date limită se definesc prin argumente în linia de comandă a programului. Data minimă este prima dată calendaristică din linia de comandă.

## PROGRAMUL BXXII7

```
#ifndef __STUDIO_H
#include <stdio.h>
#define __STUDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "bxxii5.cpp"

main(int argc,char *argv[])
/* - citește date calendaristice situate între două date limite;
 - afiseaza datele citite împreună cu diferența, în număr de
 zile, dintre datele limite și data citită */
{
    if(argc != 7){
        printf("numar argumente = %d eronat\n",argc);
        exit(1);
    }
    int tabarg[6];

    for(int i=1;i < 7;i++){
        char *p = argv[i];
        for(int j=0;*p;j++,p++)
            if(*p < '0' || *p > '9'){
                printf("argument eronat:%c\t%d\n",*p,*p);
                exit(1);
            }
        if(j > 4){
            printf("argument eronat: %s\n",argv[i]);
            exit(1);
        }
        tabarg[i-1] = atoi(argv[i]); // conversia argumentului
    }

    dc liminf(tabarg[0],tabarg[1],tabarg[2]);
    dc limsup(tabarg[3],tabarg[4],tabarg[5]);

    liminf.afisdata(); // afiseaza limita inferioara
    limsup.afisdata(); // afiseaza limita superioara
    liminf.modifmin(); // defineste data statică minima
    limsup.modifmax(); // defineste data statică superioara
```

```
// citirea datelor calendaristice
for(;){
    dc data_crt;
    if(data_crt.citdata() == 0) break; // s-a întîlnit EOF
    data_crt.afisdata(); // afiseaza data citită
    long difi,difs;
    difi=liminf.difdata(&data_crt);
    difs = limsup.difdata(&data_crt);
    printf("diferenta in numar de zile = %ld\n",dif,i,difs);
    dc *pinf;
    pinf = pinf -> datamin();
    dc *psup ;
    psup = psup -> datamax();
    printf("limita inferioara\n");
    pinf -> afisdata();
    printf("limita superioara\n");
    psup -> afisdata();
    delete pinf;
    delete psup;
}
}
```

22.8 Să se definească tipul abstract *punct* care să se utilizeze la instanțierea punctelor din plan prin coordonate rectangulare.

## FIŞIERUL BXXII8.H

```
class punct {
    double x;
    double y;
public:
    punct(double abs=0,double ord=0);
    punct(const punct&);
    int citpunct(); /* citește coordonatele punctului;
    returneaza:
        0 la EOF;
        1 altfel. */
    void afispunct(); /* afiseaza coordonatele punctului */
    void xtrans(double dx); /* translatie pe directia abscisei */
    void ytrans(double dy); /* translatie pe directia ordonatei */
    double retx(); // returneaza abscisa
    double rety(); // returneaza ordonata
};
```

Funcțiile membru se definesc separat într-un fișier de tip CPP.

## FIŞIERUL BXXII8

```
#ifndef __PUNCT_H
#include "BXXII8.H"
#define __PUNCT_H
#endif

inline punct::punct(double abs,double ord)
/* - constructor pentru instantierea obiectelor de tip punct;
```

```

    - implicit se instantiază origină axelor.*/
{
    x = abs; y = ord;
}

inline punct::punct(const punct& p)
// constructor de copiere
{
    x = p.x; y = p.y;
}

int punct::citpunct()
/* - citeste coordonatele punctului:
   - returneaza:
     0 - la EOF;
     1 - altfel.
*/
{
    int c;

    for(;;){
        printf("Abscisa=");
        if((c=scanf("%lf",&x))==1) break;
        if(c==EOF) return 0;
        printf("nu s-a tastat un numar\n");
        fflush(stdin);
    }
    for(;;){
        printf("Ordonata=");
        if((c=scanf("%lf",&y))==1) return 1;
        if(c==EOF) return 0;
        printf("nu s-a tastat un numar\n");
        fflush(stdin);
    }
}

inline void punct::afispunct()
// afiseaza coordonatele punctului
{
    printf("x=%g\ty=%g\n",x,y);
}

inline void punct::xtrans(double dx)
// translatic în direcția axei x
{
    x += dx;
}

inline void punct::ytrans(double dy)
// translatic în direcția axei y
{
    y += dy;
}

```

```

inline double punct::retx() // returneaza abscisa
{
    return x;
}

inline double punct::retty() // returneaza ordonata
{
    return y;
}

22.9 Să se scrie un program care generează obiecte de tip punct ale căror
coordonate se generează aleator. Programul afișează punctele care aparțin
ecranului, iar în final se indică:
- numărul m al punctelor afișate;
- numărul n al punctelor generate;
- raportul m/n.

PROGRAMUL BXXII9

#ifndef __STDLIB_H
#include <stdio.h>
#define __STDLIB_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXII8.CPP"

main()
/* - genereaza puncte cu coordonate aleatoare;
   - afiseaza punctele care aparțin ecranului (80 coloane a 25 linii);
   - afiseaza numarul m al punctelor afisate;
   - afiseaza numarul n al tuturor punctelor generate;
   - raportul m/n.
*/
{
    int m=0,n=0;

    for(;;){
        double xaleat = rand()%100;
        double yaleat = rand()%100;
        punct pct(xaleat,yaleat);
        n++;
        if(xaleat > 0 && xaleat <= 80 &&
           yaleat > 0 && yaleat <= 25){ // afiseaza punctul
            m++;
            pct.afispunct();
            if(m%22 == 0){

```

```

        printf("Actionati o tasta pentru a continua\n");
        printf("Actionati zero pentru a termina\n");
        if(getch() == '0') break;
    }
}

printf("numarul punctelor afisate = %d\n",m);
printf("numarul punctelor generate = %d\n",n);
printf("raportul m/n = %g\n", (double) m/n);
}

```

## 22.10 Să se definească tipul abstract *vector*.

Un vector este definit de două puncte. Unul este originea vectorului, iar celălalt este virful lui.

### FIŞIERUL BXXII10.H

```

#ifndef __PUNCT_H
#include "BXXII8.CPP"
#define __PUNCT_H
#endif

class vector {
    punct origine;
    punct virf;
public: //constructori
    vector(double xo,double yo,double xv,double yv);
    vector(double xv,double yv);
    vector(const vector& );
    double modul(); //modulul vectorului

    double arg(); //argumentul vectorului (unghiul facut cu axa Ox)

    void x_trans(double dx);
    //translatie in directia axei Ox

    void y_trans(double dy);
    //translatie in directia axei Oy

    punct retorig(); //returneaza originea vectorului
    punct retrvirf(); //returneaza virful vectorului

    double prodscalar(vector *v);
    //returneaza produsul scalar dintre vectorul curent si cel spre care pointeaza v
};

```

### Observație:

Modulul și argumentul vectorilor se calculează asemănător cu modulul și argumentul numerelor complexe. În acest caz se face diferență dintre coor-

donatele extremităților vectorului.

Funcțiile membru ale clasei *vector* se definesc în fișierul de mai jos, de

extensie CPP.

### FIŞIERUL BXXII10

```

#ifndef __VECTOR_H
#include "BXXII10.H"
#define __VECTOR_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif

inline vector::vector(double xv, double yv):virf(xv,yv)
// originea are coordonatle egale cu zero (valori implice)
{
}

inline vector::vector(double xo,double yo,double xv,double yv):
    origine(xo,yo),virf(xv,yv)
// constructor pentru initializarea originii si virfului unui vector
{
}

vector::vector(const vector& v): origine(v.origine),virf(v.virf)
// constructor de copiere
{
}

double vector::modul() // returneaza modulul unui vector
{
    double a = virf.retx() - origine.retx();
    double b = virf.rety() - origine.rety();
    return sqrt(a*a+b*b);
}

double vector::arg() // returneaza argumentul unui vector
{
    double a = virf.retx() - origine.retx();
    double b = virf.rety() - origine.rety();
    if(a == 0 &&b == 0) return 0.0;
    if(b == 0)
        if(a > 0) return 0.0;
        else return PI;
    if(a == 0)
        if(b > 0) return PI/2;
        else return(3*PI)/2;

    //a != 0 si b != 0
    double c = atan(b/a);
    if(a < 0) return PI+c; //a<0 si b!=0
}
```

```

    if(b < 0) return 2*PI+c; //a>0 si b<0
    //a>0 si b>0
    return c;
}

inline void vector::x_trans(double dx) //translatie in directia axei Ox
{
    origine.xtrans(dx);
    virf.xtrans(dx);
}

inline void vector::y_trans(double dy) //translatie in directia axei Oy
{
    origine.ytrans(dy);
    virf.ytrans(dy);
}

inline punct vector::retorig() //returneaza originea vectorului
{
    return origine;
}

inline punct vector::retvirf() //returneaza virful vectorului
{
    return virf;
}

double vector::prodscalar(vector *v)
/* returneaza produsul scalar dintre vectorul curent si cel spre care pointeaza v */
{
    double ax = v->virf.retx() - v->origine.retx();
    double ay = v->virf.rety() - v->origine.rety();
    double bx = virf.retx() - origine.retx();
    double by = virf.rety() - origine.rety();
    return ax*bx+ay*by;
}

```

- 22.11 Să se scrie un program care citește coordonatele a doi vectori din plan și afișează modulul și argumentul fiecarui vector, precum și produsul lor scalar.

### PROGRAMUL BXXII11

```

#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXII10.CPP"

```

```

main()
/*
- citeste coordonatele a doi vectori;
- calculeaza si afiseaza:
  - modulul si argumentul fiecarui vector;
  - produsul scalar al celor doi vectori.
*/
{
    punct ov1; // origine
    punct vv1; // virf
    punct ov2; // origine
    punct vv2; // virf

    // citeste primul vector
    ov1.citpunct();
    vv1.citpunct();

    // citeste al doilea vector
    ov2.citpunct();
    vv2.citpunct();

    // construieste vectorii
    vector v1(ov1.retx(), ov1.rety(), vv1.retx(), vv1.rety());
    vector v2(ov2.retx(), ov2.rety(), vv2.retx(), vv2.rety());

    // afiseaza modulul si argumentul lui v1 si v2
    printf("Vectorul v1\n");
    printf("modul = %g\t argument = %g\n", v1.modul(), v1.arg());
    printf("Vectorul v2\n");
    printf("modul = %g\t argument = %g\n", v2.modul(), v2.arg());

    // afiseaza produsul scalar
    printf("(v1,v2) = %g\n", v1.prodscalar(&v2));
}

```

- 22.12 Să se definișcă tipul abstract *sir* pentru instanțierea sirurilor de caractere. Acest tip este definit în cele mai multe lucrări relative la limbajul C++.

Interesul manifestat pentru acest tip rezultă din faptul că sirurile de caractere sunt utilizate frecvent în aproape toate programele. Mai jos, prezentăm o implementare a tipului *sir* analogă cu cea indicată în lucrarea [7].

Ca date membru alegem un pointer spre zona de memorie în care se păstrează caracterele sirului și lungimea acestuia în număr de caractere (fără caracterul *nul* de la sfîrșitul sirului).

### FISIERUL BXXII12.H

```

class sir {
    char *psir;
    int lung;
public:
    sir(char *s); /* constructor pentru initializarea obiectului cu pointerul spre sirul
                    de caractere; acesta se păstrează în memoria heap */
}

```

```

sir(int nrcar=70); /* constructor care rezerva zona de memorie in
                     memoria heap si pastreaza in ea sirul vid */
sir(const sir&); //constructor de copiere
~sir();           //destructor
int retlung();   //returneaza lungimea sirului
void afsir();    //afiseaza sirul de caractere
int citsir();    //citeste un sir de caractere

Boolean atribsir(sir *s);
/* - transfera sirul spre care pointeaza s in zona rezervata pentru sirul obiectului curent;
   - daca nu exista zona suficienta, se truncheaza sirul spre care pointeaza s si se
     returneaza valoarea false;
   - altfel se returneaza true. */
};

Mai jos, se definesc functiile membru ale clasei sir.

```

## FIŞIERUL BXXII12

```

#ifndef __Boolean
#define __Boolean
enum Boolean {false,true};
#endif
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __SIR_H
#include "BXXII12.H"
#define __SIR_H
#endif

sir::sir(char *s)
/* constructor utilizat la initializarea obiectului cu pointerul spre copia
   in memoria heap a sirului spre care pointeaza s */
{
    lung = strlen(s); // lungimea sirului
    psir = new char[lung+1]; // rezerva zona in memoria heap
    strcpy(psir,s); // transfera sirul in memoria heap
}

sir::sir(int dim)
/* - constructor care rezerva zona pentru siruri de dimensiunea dim;
   - pastreaza sirul vid in zona respectiva. */
{
    lung = dim;           // determina lungimea
    psir = new char[lung+1]; // rezerva zona
    *psir = '\0';          // pastreaza sirul vid
}

```

```

sir::sir(const sir& s) // constructor de copiere
{
    lung = s.lung;           // lungimea sirului
    psir = new char[lung+1]; // rezerva zona
    strcpy(psir,s.psir);    // copiază sirul
}

inline sir::~sir()
/* destructor: elibereaza zona din memoria heap ocupata de sir */
{
    delete psir;
}

inline int sir::retlung()
// returneaza lungimea sirului: numarul caracterelor sirului fara caracterul nul
{
    return lung;
}

inline void sir::afsisr()
// afiseaza sirul spre care pointeaza psir
{
    printf(psir);
    printf("\n");
}

int sir::citsir()
/* - citeste un sir de la intrarea standard si-l pastreaza in zona heap rezervata pentru obiectul curent;
   - returneaza:
     0 - la sfarsit de fisier
     -1 - la trunchierea sirului citit
     1 - altfel. */
{
    char temp[255];

    if(gets(temp)==0) return 0; // s-a intalnit sfarsitul de fisier
    strncpy(psir,temp,lung);
    *(psir+lung)='\0';
    if(strlen(temp) > lung) return -1; // trunchiere
    else return 1;
}

Boolean sir::atribuisir(sir *s)
/* - transfera sirul spre care pointeaza s in zona rezervata pentru sirul obiectului curent;
   - daca nu exista zona suficienta, se truncheaza sirul spre care pointeaza s si se returneaza
     valoarea false; altfel se returneaza true */
{
    strncpy(psir,s->psir,lung);
    if(strlen(s->psir) > lung) return false;
    return true;
}

```

22.13 Să se scrie un program care realizează următoarele operații asupra obiectelor

- de tip *sir*:
- initializare;
  - citire de siruri de caractere de la intrarea standard;
  - copiere de obiecte de tip *sir*;
  - atribuirile obiectelor de tip *sir*;
  - afişarea sirurilor de caractere din compunerea obiectelor de tip *sir*.

## PROGRAMUL BXXII13

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXII12.CPP"

main() /* operatii cu obiecte de tip sir */
{
    // instantieri de obiecte de tip sir

    sir sir1("Limbajul C++ este un C mai bun");
    sir sir2("Limbajul C++ suporta stilul de programare:\n"
            "- prin abstractizarea datelor;\n"
            "- orientata spre obiecte");
    sir sir3; // instantiere fara initializare; sir3 contine sirul vid
    sir sir4 = sir1; // instantiere folosind constructorul de copiere

    // afisarea obiectelor instantiatate mai sus
    printf("sir1\n");
    sir1.afsir();
    printf("sir2\n");
    sir2.afsir();
    printf("sir3\n");
    sir3.afsir();
    printf("sir4\n");
    sir4.afsir();

    // citiri de siruri de la intrarea standard
    while(sir3.citsir()) // citirea sirurilor
        sir3.afsir(); // afisarea sirurilor citite

    // atribuirile de siruri
    if(sir3.atrabsir(&sir1) == false)
        printf("trunchiere la atribuire\n");
    sir3.afsir();
    if(sir3.atrabsir(&sir2) == false)
        printf("trunchiere la atribuire\n");
    sir3.afsir();
}
```

## 22.10. Funcție prieten (Friend function)

La inceputul capitolului, s-a afirmat că o proprietate de bază a tipurilor abstrakte este protecția datelor membru ale tipului respectiv. Elementele protejate constituie astfel numita implementare a tipului abstract.

Tipurile abstrakte se definesc cu ajutorul claselor. Se obișnuiește să se spună că datele protejate ale unui tip abstract sunt incapsulate în clasa care definește tipul respectiv.

Protecția datelor se realizează prin aceea că la ele au acces numai funcțiile membru ale tipului (clasei). De asemenea, dacă o funcție membru este protejată, atunci ea poate fi apelată numai prin intermediu unei funcții membru a tipului (clasei).

Acest mod de lucru, deși asigură o protecție bună a elementelor membru protejate ale clasei (elemente protejate prin *private* sau *protected*), uneori este considerat prea rigid. Astfel, deși există funcții descrise în C care pot fi utilizate pentru a prelucra instanțe ale unei clase, ele nu se pot utiliza simplu deoarece nu sunt funcții membru ale clasei respective și deci nu au acces la datele membru. Mai mult decit atât, o funcție membru se apelează totdeauna în dependență cu un obiect care este numit *obiectul curent* al apelului. În acest fel, o funcție membru se apelează prin una din următoarele formate:

*nume\_obiect.nume\_functie\_membru(...)*

sau

*pointer\_nume\_clasa -> nume\_functie\_membru()*

În cazul funcțiilor obișnuite nu se admit astfel de apeluri, toate datele prelucrate de funcție se transferă prin parametri sau sunt globale.

De aceea, o funcție obișnuită poate fi utilizată pentru a prelucra obiectele unei clase dacă ea se modifică în astfel încât să devină funcție membru.

S-a făcut un compromis pentru a admite accesul la elementele protejate și pentru anumite funcții care nu sunt membru. Aceste funcții au fost numite *funcții prieten* pentru clasa respectivă. Ele trebuie să fie precizate ca atare în definiția clasei. În acest scop, prototipurile lor sunt prezente în definiția clasei și sunt precedate de cuvintul cheie *friend*.

### Exemplu:

Fie tipul definit de utilizator:

```
struct complex {
    double real;
    double imag;
};
```

Funcția *modul*, pentru calculul modulului unei date de tip *complex* poate fi definită, ca mai jos, ca o funcție obișnuită:

```

double modul(complex *z)
/* returneaza modulul numarului complex spre care pointeaza z */
{
    return sqrt(z->real*z->real + z->imag*z->imag);
}

```

Dacă se consideră declarațiile:

```

complex z1 = {1,1};
double d;

```

atunci instrucția de atribuire:

```
d = modul(&z1);
```

atribuie lui *d* modulul numărului complex *z1* = 1+i.

Același lucru se poate realiza implementând tipul abstract *complex* ca în exercițiul 22.1.:

```

class complex {
    double real;
    double imag;
public:
    complex(double x=0,double y=0)
    {
        real = x; imag = y;
    }
    double modul()
    {
        return sqrt(real*real+imag*imag);
    }
    ...
};

Declararea pentru z1 se scrie:
```

```
complex z1(1,1);
```

iar instrucția de atribuire devine:

```
d = z1.modul();
```

În acest caz, funcția *modul* nu are parametru deoarece ea se apelează pentru obiectul *z1*. În corpul funcției *modul* este definit pointerul implicit *this* și acesta are ca valoare chiar adresa lui *z1*.

Expresia:

```
real*real+imag*imag
```

din corpul funcției membru *modul*, trebuie considerată ca și cind ar fi scrisă sub forma:

```
this->real*this->real+this->imag*this->imag
```

Funcția *modul*, definită pentru tipul utilizator *complex* poate înlocui funcția membru *modul* a clasei *complex* dacă ea se declară ca funcție prieten a clasei

*complex*, ca mai jos:

```

class complex {
    double real;
    double imag;
public:
    complex(double x=0,double y=0)
    {
        real = x; imag = y;
    }
    friend double modul(complex *z);
    ...
};

```

Funcția *modul*, prieten a clasei *complex*, are aceeași definiție ca și funcția *modul* definită pentru tipul *complex* introdus prin construcția *struct*. De asemenea, ea se apelează ca orice funcție obișnuită:

În exemplul de față, dacă se consideră instanțierea:

```
complex z(1,1);
```

atunci funcția prieten *modul* se apelează în mod obișnuit, adică prin:

```
d = modul(&z);
```

Spre deosebire de funcțiile membru, în cazul funcțiilor prieten nu este definit pointerul implicit *this*. Acest lucru conduce la faptul că o funcție prieten are un parametru în plus față de o funcție membru care are același efect.

O funcție prieten poate fi o funcție obișnuită (ca în cazul funcției *modul* din exemplul de mai sus) sau o funcție membru a unei alte clase.

**Exemplu:**

```

class clasa1 {
    ...
    tip functie_membru(...);
    ...
};

class clasa2 {
    ...
    friend tip clasa1::functie_membru(...);
    ...
};

```

Funcția *functie\_membru* este o funcție membru a clasei *clasa1*. Ea este o funcție prieten a clasei *clasa2*.

În cazul în care se dorește ca toate funcțiile membru ale clasei *clasa1* să fie funcții prieten ale clasei *clasa2*, se poate proceda ca mai jos, în loc de a indica individual fiecare funcție din clasa *clasa1* că este funcție prieten:

```

class clasa1; //definitia prescurtată de clasa
class clasa2 {
    ...
};

```

```

friend clasal;
};

}

```

În acest caz, se spune că *clasa1* este o clasă prieten a clasei *clasa2*.

Proprietatea de clasă prieten nu este tranzitivă. Astfel, dacă *clasa1* este o clasă prieten pentru *clasa2* și *clasa2* este o clasă prieten pentru *clasa3*, aceasta nu implică faptul că *clasa1* este clasă prieten pentru *clasa3*.

Modificatorii de protecție nu au nici o influență asupra unei funcții prieten. De aceea, specificarea faptului că o funcție este prieten pentru o clasă, poate fi scrisă în orice punct din interiorul definiției clasei respective.

Funcția prieten nu este protejată și deci poate fi utilizată fără nici o restricție ca orice funcție obișnuită.

### Exerciții:

22.14 Să se scrie o funcție care ridică un număr complex la o putere întreagă pozitivă.

Un număr complex se poate ridica la o putere întreagă pozitivă folosind formula lui Moivre. Aceasta se exprimă prin relația:

$$(r(\cos(a) + i\sin(a)))^{**n} = r^{**n}(\cos(n*a) + i\sin(n*a))$$

unde:

*r* - Este modulul numărului complex

*a* - Este argumentul acestuia.

Tipul *complex* se implementează folosind *clasa* definită în exercițiul 22.3.

### FUNCȚIA BXXII14

```

void cputere(complex& z,int n)
/* ridică la puterea n numărul complex la care z este referință */
{
    double r;
    double a;

    r = z.modul(); /* - se calculează modulul numărului complex la care z este referință;
                      - se apelă funcția membru modul a clasei complex */
    a = z.arg(); /* - se calculează argumentul numărului complex la care z este referință;
                      - se apelă funcția membru arg a clasei complex */
    double rlan = pow(r,(double)n);
    double na = n*a;
    z.real = rlan*cos(na);
    z.imag = rlan*sin(na);
}

```

### Observație:

Funcția *cputere* nu este o funcție membru a clasei *complex*. Ea are acces la componentele *real* și *imag* ale obiectului de tip *complex* referit de *z*, numai dacă este o funcție prieten a clasei *complex*.

22.15 Să se modifice definiția clasei *complex* din fișierul BXXII3.H inserind funcția *cputere* ca funcție prieten.

### FIȘIERUL BXXII15.H

```

enum Boolean {false,true};

class complex { // date membru protejate (private)
    double real;
    double imag;
public: // funcții membru neprotejate
    complex(double x = 0,double y = 0);
    complex(const complex&);

    double modul(); double arg();
    double retreal(); double retimag();
    void afiscomplex();
    Boolean citcomplex();
    void adcomplex(complex *z1,complex *z2);
    void sccomplex(complex *z1,complex *z2);
    void negcomplex(complex *z1);
    void mulcomplex(complex *z1,complex *z2);

    Boolean divcomplex(complex *z1,complex *z2);

    // funcție prieten
    friend void cputere(complex& z,int n);
    // calculează  $z^{**n}$ , pentru n natural
};

```

22.16 Să se scrie un program care ridică numărul complex  $1+i$  la puterea *n*. Numărul *n* este un întreg citit de la intrarea standard.

### PROGRAMUL BXXII16

```

#ifndef __PI
#define PI 3.141592653589793238
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif

// pentru a nu include fișierul BXXII3.H se definește __BXXII3_H
#define __BXXII3_H
// se include definiția clasei complex din fișierul BXXII15.H
#include "BXXII15.H"

// se includ definițiile funcțiilor membru
#include "BXXII3.CPP"

```

```

// se include definitia functiei prieten cputere
#include "BXXIII14.CPP"
#include <stdlib.h>

main()
/* citeste intregul n, calculeaza si afiseaza (1+i)**n */
{
    int n,c;
    complex z(1,1); // z = 1+i
    complex zlan; // zlan = 0+0*i

    // citeste pe n
    for(;;){
        printf("exponent=");
        if((c = scanf("%d",&n)) == 1) break;
        printf("nu s-a tastat un intreg\n");
        if(c == EOF){
            printf("s-a tastat EOF\n");
            exit(1);
        }
        fflush(stdin);
    }

    complex complex_unu(1,0);
    int m;

    m = n < 0 ? -n : n; // calculeaza abs(n)
    if(m == 0){           // (1+i)**0 = 1+0.i
        complex_unu.afiscomplex();
        exit(0);
    }
    if(n == 1){           // (1+i)**1 = 1+i
        z.afiscomplex();
        exit(0);
    }
    if( n == -1){         // (1+i)**(-1) = 1/(1+i)
        zlan.divcomplex(&complex_unu,&z);
        zlan.afiscomplex();
        exit(0);
    }

    // m>1
    cputere(z,m); // z=z**m
    if(n > 1){
        z.afiscomplex();
        exit(0);
    }

    // putere negativa: z**n = 1/z**abs(n)
    zlan.divcomplex(&complex_unu,&z);
    zlan.afiscomplex();
}

```

## 23. SUPRAÎNCĂRCAREA OPERATORILOR

Tipurile abstracte de date se definesc in limbajul C++ cu ajutorul claselor. Ar fi ideal ca tipurile abstracte să se comporte ca și cele predefinite. Așa cum s-a arătat in capitolul precedent, există o serie de asemănări între tipurile abstracte și cele predefinite. Așa de exemplu, datele de tip abstract (obiectele) se declară la fel ca și cele de tip predefinit. De asemenea, ele pot fi inițializate la declarare.

Operațiile care se pot executa asupra obiectelor sunt bine precizate ca și în cazul tipurilor predefinite. În cazul tipurilor abstracte, operațiile se definesc cu ajutorul *funcțiilor membru și prieten*.

### Exemplu:

Pentru clasa *complex* se pot defini funcții membru și prieten pentru a realiza cele 4 operații aritmetice asupra obiectelor de tip *complex*.

În definiția de clasă, de mai jos, se definește o funcție membru pentru adunarea obiectelor de tip *complex* și o funcție prieten pentru scăderea obiectelor de tip *complex*.

```

class complex {
    double real;
    double imag;
public:
    complex(double x=0,double y=0)
        // constructor
    {
        real = x;
        imag = y;
    }

    // functie membru pentru adunarea obiectelor de tip complex
    complex adcomplex(complex& z2);

    // functie prieten pentru scaderea obiectelor de tip complex
    friend complex sccomplex(complex& z1,complex& z2);
}

```

În continuare, se pot defini funcțiile *adcomplex* și *sccomplex* ca mai jos.

```

complex complex::adcomplex(complex& z)
/* returneaza suma dintre obiectul complex curent si cel referit de z */
{
    complex ztemp;

    ztemp.real = real + z.real;
    ztemp.imag = imag + z.imag;
}

```

```

    return ztemp;
}

complex sccomplex(complex& z1,complex& z2)
/* returneaza diferența z1-z2 */
{
    complex ztemp;

    ztemp.real = z1.real -z2.real;
    ztemp.imag = z1.imag -z2.imag;
    return ztemp;
}

```

Dacă *a*, *b*, *c* sunt 3 obiecte de tip *complex* instanțiate cu ajutorul declarației:

```
complex a,b,c;
```

atunci instrucțiunea:

(1) *c* = *a*.adcomplex(*b*);

atribuie lui *c*, de tip *complex*, rezultatul adunării obiectelor complexe *a* și *b*.

Menționăm că apelul:

(2) *a*.adcomplex(*b*)

returnează un obiect de tip *complex* la fel cum o funcție de antet:

*tip nume\_f(...)*

unde *tip* este un tip predefinit (diferit de cuvintul *void*) returnează o valoare de tip *tip*.

Obiectul *complex* returnat prin apelul (2) (adică suma *a+b*) se atribuie obiectului *complex c* prin instrucțiunea (1).

În general, dacă *ob1* și *ob2* sunt instanțieri ale clasei *nume\_clasa*:

*nume\_clasa ob1,ob2;*

atunci o expresie de forma:

\* (3) *ob1* = *ob2*

este admisă de compilator și ea atribuie lui *ob1* valoarea obiectului *ob2*. Această atribuire se realizează printr-o copiere bit cu bit a valorilor componentelor lui *ob2* în zonele de memorie alocate componentelor lui *ob1*. O astfel de atribuire nu este totdeauna corectă.

În cazul nostru, ea este suficientă pentru atribuirea obiectelor de tip *complex*.

În mod analog, instrucțiunea:

(4) *c* = *sccomplex(a,b);*

atribuie lui *c* obiectul *complex* rezultat prin scăderea obiectului *complex b* din obiectul *complex a*.

În felul acesta se pot introduce și alte operații asupra obiectelor de tip *complex* folosind funcții membru sau prieten.

Mecanismul de protecție a datelor private asigură ca să nu se poată realiza alte operații asupra obiectelor de tip *complex* decât cele precizate la definiția clasei *complex*.

Se observă o analogie între tipurile predefinite *int*, *long*, *float*, *double* și tipul abstract *complex*.

Cu toate acestea, tipul *complex* definit ca mai sus, nu permite utilizarea operatorilor obișnuiți pentru a exprima operații cu obiecte *complex* ca în cazul operațiilor cu numere de tip *int*, *long*, *float* sau *double*.

Astfel, dacă *i*, *j*, *k* sunt declarate prin declarația:

int *i,j,k;*

atunci se pot scrie expresii de forma:

(5) *k* = *i+j;*

sau

(6) *k* = *i-j;*

Expresii de acest fel sunt valabile și pentru alte tipuri predefinite. De exemplu, dacă *x*, *y* și *z* sunt date de tip *double*:

double *x,y,z;*

atunci se pot utiliza expresii similare:

(7) *z* = *x+y;*

sau

(8) *z* = *x-y;*

Evident, expresiile de forma (5), (6), (7) și (8) sunt sugestive și ar fi de dorit ca ele să poată fi extinse și pentru obiecte de tip *complex*. Cu alte cuvinte, am dori ca în locul instrucțiunii (1) să putem folosi o instrucțiune de forma:

(9) *c* = *a+b;*

iar în locul lui (4), instrucțiunea:

(10) *c* = *a-b;*

Acest lucru este posibil prin mecanismul de supraincărcare a operatorilor.

Operatorii + și - trebuie supraincarcați pentru a realiza operații corespunzătoare cu obiecte de tip *complex*.

De altfel, ei sunt supraincarcați pentru tipurile predefinite *int*, *long*, *float*, *double*, *unsigned*, *unsigned long* și *long double*, deoarece se admit expresii de forma:

*p* = *q+r;*

sau

*p* = *q-r;*

pentru operanzi de tipurile predefinite, enumerate mai sus. Supraincărcările operatorilor + și - pentru operanzi de tipuri predefinite sunt predefinite. De aceea, pentru a putea utiliza acești operatori și cu operanzi de tipuri abstractive, este nevoie ca la definirea tipurilor respective să se supraincarce operatorii în mod corespunzător. Ca rezultat al supraincărcării operatorilor se ajunge ca operatorii respectivi să poată fi utilizati în expresii obișnuite cu operanzi obiecte de tipuri abstractive.

Limbajul C++ permite supraincărcarea numai a operatorilor existenți în limbaj. Dintre aceștia nu pot fi supraincărați operatorii:

`.. :: ? și : *`

De asemenea, prin supraincărcarea operatorilor nu se poate schimba n-aritatea, prioritatea sau asociativitatea operatorilor, acestea fiind elemente predefinite pentru tipuri predefinite și deci ele se vor menține și pentru tipuri abstractive.

Prin n-aritate înțelegem că operatorul este *binar* sau *unar*.

Supraincărcarea operatorilor se realizează cu ajutorul unor funcții membru sau prieten speciale. Specificul lor constă în numele acestora. El se compune din cuvintul cheie *operator* și unul sau mai multe caractere care definesc operatorul care se supraincarcă. Între cuvintul cheie *operator* și caracterele care definesc operatorul care se supraincarcă se află cel puțin un spațiu alb.

În cazul tipului *complex*, vom folosi pentru supraincărcarea operatorului + funcția definită mai jos:

```
complex complex::operator + (complex& z)
/* - supraincarca operatorul + pentru obiecte de tip complex;
   - functia returneaza un obiect complex care reprezinta suma dintre
     obiectul complex curent si cel referit de z.
*/
{
    complex temp;
    temp.real = real + z.real;
    temp.imag = imag + z.imag;
    return temp;
}
```

În acest caz, numele funcției este:

*operator +*

În rest, funcția care supraincarcă operatorul + este identică cu funcția membru *adcomplex* definită mai sus.

Efectul schimbării numelui *adcomplex* prin *operator +* este acela al supraincărcării operatorului + pentru a admite ca operanzi obiecte de tip *complex*. Deci, o expresie de forma:

`a+b`

este legală și ea are ca rezultat obiectul *complex* rezultat prin adunarea obiectelor *complex* *a* și *b*. Cu alte cuvinte, această expresie poate fi utilizată în locul expresiei:

`a.adcomplex(b)`

al cărui sens este mai puțin evident. În felul acesta, instrucțiunea (1) de mai sus, se transcrie sub forma:

`c = a+b;`

care este la fel de simplă și clară ca și instrucțiunile (5) și (7) utilizate pentru operanzi de tipurile predefinite *int* și respectiv *double*.

În mod analog, operatorul - poate fi supraincărat schimbând numele funcției prieten *sccomplex* cu *operator -* ca mai jos:

```
complex operator - (complex& z1,complex & z2)
/* - supraincarca operatorul - pentru obiecte de tip complex;
   - functia returneaza un obiect complex care reprezinta diferența dintre obiectele z1 si z2 */
{
    complex ztemp;
    ztemp.real = z1.real - z2.real;
    ztemp.imag = z1.imag - z2.imag;
    return ztemp;
}
```

În continuare, se pot utiliza expresii de forma:

`a-b`

în locul apelului funcției *sccomplex*:

`sccomplex(a,b)`

Din cele de mai sus se observă că operatorii pot fi supraincărați prin funcții membru sau prin funcții prieten. O diferență între cele două tipuri de funcții constă în numărul de parametri. Astfel, funcțiile membru care supraincarcă operatorii unari nu au parametri, iar cele care supraincarcă operatori binari au un singur parametru.

În cazul funcțiilor prieten, numărul parametrilor este egal cu n-aritatea operatorului care se supraincarcă.

Motivul pentru a utiliza o funcție membru sau prieten pentru a supraincarca un operator va rezulta dintr-un paragraf ulterior.

La supraincărcarea operatorilor pentru obiecte de tip abstract nu se poate face diferență între formele prefixate și postfixate.

Anumiți operatori prezintă unele particularități la supraincărcarea lor și de aceea ei vor fi tratați în paragrafe separate.

În principiu, funcțiile membru care supraincarcă un operator nu sunt statice.

O excepție de la această regulă o constituie supraincărcarea operatorilor unari *new* și *delete*. Aceștia se pot supraincarca numai prin funcții membru statice.

Alți operatori care prezintă particularități la supraincărcare sunt parantezele, operatorul  $\rightarrow$  și operatorul de atribuire ( $=$ ).

Mai sus, am considerat supraincărcarea operatorilor binari pentru cazul cînd ambii operanzi sunt de tipul abstract *complex*.

Se pune problema de a utiliza operatorii respectivi și cu operanzi diferiți, adică în expresii de forma:

$a+k$ ,  $k+a$ ,  $a-k$ ,  $k-a$

unde:

$k$  - Este o variabilă de tip *int*, iar  $a$  un obiect de tip *complex*.

Există două soluții pentru a putea utiliza expresii de această formă. O primă soluție este supraincărcarea operatorului  $+$  nu numai pentru operanzi de tip *complex*, ci și pentru operanzi de tipuri diferite.

A doua soluție constă în conversia operandului care nu este de tip *complex* într-un obiect de tip *complex* înainte de a aplica operatorul *plus* supraincărcat pentru operanzi de tip *complex*.

Acest lucru se poate realiza destul de simplu, dar nu totdeauna este eficient.

Pentru moment ne referim la prima soluție, iar soluția a doua va fi tratată într-un capitol nou cu privire la conversii.

Pentru a legaliza expresii de forma:

$a+k$

unde  $a$  este un obiect de tip *complex*, iar  $k$  o variabilă de tip *int*, putem adăuga o nouă funcție *operator membru* sau prieten de felul celor de mai jos.

```
complex complex::operator + (int k)
/* returneaza obiectul de tip complex rezultat prin adaugarea la obiectul complex curent
   a valorii lui k */
{
    complex ztemp;
    ztemp.real = real + k;
    ztemp.imag = imag;
    return ztemp;
}
```

În locul acestei funcții membru putem defini funcția prieten, de mai jos, care are același efect.

```
complex operator + (complex& z1, int k)
/* returneaza obiectul de tip complex rezultat prin adăugarea la z1 a valorii lui k */
{
    complex ztemp;
    ztemp.real = z1.real + k;
    ztemp.imag = z1.imag;
    return ztemp;
}
```

Adăugind una din aceste funcții, se pot utiliza instrucțiuni de forma:

```
complex a,b;
int k;
...
b = a+k;
b = a+123;
```

Să observăm că funcțiile de mai sus implică faptul că primul operand al operatorului  $+$  să fie de tip *complex*. Aceasta înseamnă că expresiile de forma:

$k+a$  |  
123+a .

sunt ilegale. Pentru a accepta astfel de expresii este necesar ca operatorul  $+$  să fie supraincărat în mod corespunzător, adică pentru expresii în care primul operand este de tip *int*, iar cel de al doilea de tip *complex*.

O astfel de supraincărcare se poate realiza dar numai printr-o funcție prieten deoarece în cazul funcțiilor membru, primul parametru este totdeauna obiectul curent și deci acesta nu poate fi de tip *int*.

De aceea, pentru a admite expresii în care primul operand este de tip *int*, iar cel de al doilea este un obiect de tip *complex*, vom utiliza funcția prieten de mai jos.

```
complex operator + (int k,complex& z)
/* returneaza obiectul complex k+z */
{
    complex ztemp;
    ztemp.real = z.real+k;
    ztemp.imag = z.imag;
    return ztemp;
}
```

Din acest exemplu se vede că supraincărcarea operatorilor nu permite moștenirea proprietății de comutativitate a acestora. Astfel, deși operatorul  $+$  este comutativ pentru operanzi de tipuri predefinite, supraincărcarea lui pentru expresii de forma:

$a+k$

unde:

$a$  - Este obiect de tip *complex*.

$k$  - Este o variabilă de tip *int*, nu este valabilă și pentru expresii de forma:

$k+a$

### Exerciții:

23.1 Să se definească tipul abstract *complex* supraincărcind operatori pentru operații cu numere complexe după cum urmează:

operatori	tipul operandului stîng	tipul operandului drept	tipul rezultatului
<i>Adunare</i>			
+	complex	complex	complex
+	complex	double	complex
+	double	complex	complex
<i>Scădere</i>			
-	complex	complex	complex
-	complex	double	complex
-	double	complex	complex
<i>Inmulțire</i>			
*	complex	complex	complex
*	complex	double	complex
*	double	complex	complex
<i>Împărțire</i>			
/	complex	complex	complex
/	complex	double	complex
/	double	complex	complex
<i>Ridicare la putere</i>			
<sup>^</sup>	complex	int	complex
<i>Test de egalitate</i>			
==	complex	complex	int
==	complex	double	int
==	double	complex	int
<i>Negativare</i>			
-	complex		complex
<i>Incrementare</i>			
++	complex		complex
<i>Decrementare</i>			
--	complex		complex
<i>Rădăcina patrată</i>			
!	complex		complex

De asemenea, se definesc funcții membru pentru:

- modulul unui număr complex;
- argumentul unui număr complex;
- citirea unui număr complex;

- afișarea unui număr complex;
- incrementarea părții imaginare a unui număr complex;
- decrementarea părții imaginare a unui număr complex;
- returnarea părții reale a unui număr complex;
- returnarea părții imaginare a unui număr complex.

Ridicarea la putere (operatorul <sup>^</sup>) se realizează folosind formula lui Moivre.  
Dacă  $z = a+b*i$ , atunci se formează forma trigonometrică a lui  $z$ :

$$z = r*(\cos(\alpha)+i*\sin(\alpha))$$

Pentru  $n$  întreg și pozitiv, avem:

$$z^{**n} = r^{**n}(\cos(n*\alpha)+i*\sin(n*\alpha))$$

Unghiul  $n*\alpha$  se poate reduce la primul cerc dacă se pune sub forma:  
 $n*\alpha = 2*k*\pi + \beta$

unde:

$k$  - Este partea întreagă a împărțirii  $(n*\alpha)/(2*\pi)$ .

$$\pi = 3.14159265358979.$$

Dacă  $n < 0$ , atunci:

$$z^{**n} = 1/z^{**abs(n)}$$

Întrucit operatorul  $**$  nu există în limbajul C, pentru ridicarea la putere s-a ales operatorul <sup>^</sup>. Prioritatea acestuia este mai mică decât a operatorilor binari aritmici. De aceea, în expresii mai complicate se vor utiliza paranteze rotunde pentru a impune ordinea dorită a operațiilor.

De exemplu, ordinea operațiilor în evaluarea expresiei:

$$a*x^{10}$$

unde  $a$  și  $x$  sunt numere *complex*, este următoarea:

- se înmulțește  $a$  cu  $x$ ;
- rezultatul înmulțirii se ridică la puterea 10.

Pentru a realiza întâi ridicarea la putere a lui  $x$  și apoi înmulțirea cu  $a$ , vom utiliza expresia:

$$a*(x^{10})$$

Testul de egalitate este util mai ales la stabilirea dacă un număr *complex* este nul sau nu. O expresie de formă:

$$a == b$$

unde  $a$  și  $b$  sunt ambele numere complexe sau unul este complex și celălalt de tip numeric (*int*, *long*, *unsigned*, *double* etc.) are ca rezultat una din valorile 0 sau 1. Rezultatul expresiei este 0 dacă cei doi operanzi sunt diferenți și 1 altfel.

Operatorul de incrementare (++), incrementă partea reală a numărului complex la care se aplică.

În mod analog, operatorul de decrementare (--), decrementă partea reală a numărului complex la care se aplică.

Rădăcina pătrată se calculează folosind forma trigonometrică a numărului complex.

Astfel, dacă:

$$z = a+ib = r(\cos(\alpha) + i\sin(\alpha))$$

atunci rădăcina pătrată din z se determină ca mai jos:

$$z_1 = \sqrt{r}(\cos(\alpha/2) + i\sin(\alpha/2))$$

și

$$\begin{aligned} z_2 &= \sqrt{r}(\cos(\alpha/2 + \pi) + i\sin(\alpha/2 + \pi)) = \\ &= -\sqrt{r}(\cos(\alpha/2) + i\sin(\alpha/2)) = -z_1 \end{aligned}$$

### FIŞIERUL BXIII1.H

```
class complex {
    double real;
    double imag;

    static int citdouble(char *s, double& d);
    /* - afiseaza s;
       - citeste un numar si-l atribuie datei referite de d;
       - returnaza:
         0 - la sfarsit de fisier;
         1 - altfel. */
public:           // constructor
    complex(double x=0, double y=0);
    // modulul numarului complex

    double modul();
    // argumentul numarului complex

    double arg();
    // citeste un numar complex

    int citcomplex (char *s);
    /* - afiseaza s;
       - citeste componentele numarului complex;
       - returnaza:
         0 - la sfarsit de fisier;
         1 - altfel. */

    // afiseaza un numar complex

    void afiscomplex(char *f);
    /*afiseaza componentele numarului complex conform formatului f*/
}
```

```
// incrementarea partea imaginara
void ipi();

// decrementarea partea imaginara
void dpi();

// obiectul curent se notcaza cu z
// obiectul rezultat se notcaza cu r
// operatori unari

// negativare
complex operator - () ; // r=-z

// incrementarea partea reala
complex operator ++ () ;

// decrementarea partea reala
complex operator -- () ;

// radacina patrata
complex operator !(); // r=sqrt(z)

// operatori binari
// adunare

// r=z+z2
complex operator + (complex& z2);

// r=z+d
complex operator + (double d);

// r=d+z2
friend complex operator + (double d, complex& z2);

// scadere
// r=z-z2
complex operator - (complex& z2);

// r=z-d
complex operator - (double d);

// r=d-z2
friend complex operator - (double d, complex& z2);

// inmultire
// r=z*z2
complex operator * (complex& z2);

// r=z*d
complex operator * (double d);

// r=d*z2
friend complex operator * (double d, complex& z2);
```

```

// impartire
// r = z/z2
complex operator / (complex& z2);

// r = z/d
complex operator / (double d);

// r = d/z2
friend complex operator / (double d, complex& z2);

// ridicare la putere
// r = z**i
complex operator ^ (int i);

// test de egalitate

/* a == b; se returneaza 0 sau 1 dupa cum relatia este falsa sau adevarata */

// z==b
int operator == (complex& b);

// z==d
int operator == (double d);

// d==b
friend int operator == (double d,complex& b);

// returnaza partea reala a lui z
double retreal();

// returnaza partea imaginara a lui z
double retimag();
};


```

Funcțiile care intervin în definiția clasei *complex* se definesc într-un fișier separat, de extensie *CPP*.

## FISIERUL BXXIII1

```

#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H

```

```

#endif
#ifndef __BXXIII1_H
#include "BXXIII1.H"
#define __BXXIII1_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif

inline complex::complex(double x,double y)
//constructor utilizat la initializare;
// implicit x=y=0
{
    real = x;
    imag = y;
}

inline double complex::modul()
// returneaza modulul obiectului curent
{
    return sqrt(real*real + imag*imag);
}

double complex::arg()
// returneaza argumentul obiectului curent
{
    if(real == 0 && imag == 0) return 0.0; //z==0
    if(imag == 0) //z=parte reala
        if(real > 0) return 0.0; //z=parte reala > 0
        else return PI; //z=parte reala < 0
    if(real == 0) //z=parte imaginara
        if(imag > 0) return PI/2; //z=parte imaginara > 0
        else return -(3*PI)/2; //z=parte imaginara < 0
    // z = real+i*imag; real != 0 si imag != 0

    double a = atan(imag/real);
    if(real < 0) return PI+a; // real < 0 si imag != 0
    if(imag < 0) return 2*PI+a; // real > 0 si imag < 0
    return a; // real > 0 si imag > 0
}

int complex::citdouble(char *s,double& d)
/* - afiseaza s;
   - citeste un numar si-l atribuie datei referite de d;
   - returneaza:
       0 - la sfarsit de fisier;
       1 - altfel.
*/
{
    char t[255];

```

```

for();{
    if(strlen(s)) // sirul s nu este vid
        printf(s);
    if(gets(t) == 0) return 0; // s-a intilnit EOF
    if(sscanf(t,"%lf",&d) == 1) break;
    printf("nu s-a tastat un numar\n");
    printf("se reia citirea\n");
}
return 1;
}

int complex::citcomplex(char *s)
/* - afiseaza s daca nu este vid; daca s este vid se afiseaza texte standard:
   Partea reala si Partea imaginara
   - citeste componentele obiectului complex curent;
   - returneaza:
     0 - la sfarsit de fisier;
     1 - altfel.
*/
{
    if(*s){ // sirul s nu este vid;
    // se citesc componente obiectului curent
        for();{
            printf(s);
            int i;
            double d;

            if((i = scanf("%lf",&d)) == EOF) return 0;
            if(i != 1){
                printf("nu s-a tastat un numar\n");
                printf("pentru partea reala\n");
                printf("se reia citirea de la inceput\n");
                fflush(stdin); // videaza zona tampon
                continue;
            }
            else real = d;
            if((i = scanf("%lf",&d)) == EOF) return 0;
            if(i != 1){
                printf("nu s-a tastat un numar\n");
                printf("pentru partea imaginara\n");
                printf("se reia citirea de la inceput\n");
                fflush(stdin); // videaza zona tampon
            }
            else{
                imag = d;
                fflush(stdin);
                return 1;
            }
        } // sfarsit for
    } // sfarsit if(s)

    // s pointeaza spre sirul vid

    if(complex::citdouble("Partea reala = ",real) == 0)

```

```

        return 0;
    if(complex::citdouble("Partea imaginara = ",imag) == 0)
        return 0;
    return 1;
}

void complex::afiscomplex(char *f)
/* - afiseaza componentele obiectului curent cu formatul f;
   - daca f reprezinta sirul vid, componentele se afiseaza in format standard */
{
    if(*f) // formatul de afisare nu este vid
        printf(f,real,imag);
    else // format standard
        printf("%g+i*%g\n",real,imag);
}

inline void complex::ipi()
/* incrementarea partea imaginara a obiectului complex curent */
{
    imag++;
}

inline void complex::dpi()
/* decrementarea partea imaginara a obiectului complex curent */
{
    imag--;
}

complex complex::operator -()
/* returneaza negativul obiectului complex curent: r = -z
{
    complex r;
    r.real = -real;
    r.imag = -imag;
    return r;
}

inline complex complex::operator ++()
/* incrementarea partea reala a obiectului complex curent */
{
    real++;
    return *this;
}

inline complex complex::operator --()
/* decrementarea partea reala a obiectului complex curent */
{
    real--;
    return *this;
}

complex complex::operator !()
/* calculeaza radacina patrata din obiectul complex curent */

```

```

{
complex r; //implicit real=imag=0
double d;

if((d=modul()) == 0) return r; //obiect complex nul

double alfa;
alfa = arg(); //argumentul obiectului complex
d = sqrt(d); //radacina patrata din modul
alfa = alfa/2;
r.real = d*cos(alfa);
r.imag = d*sin(alfa);
return r;
}

complex complex::operator + (complex& z2) //r=z+z2
{
complex r;

r.real = real + z2.real;
r.imag = imag + z2.imag;
return r;
}

complex complex::operator + (double d) //r=z+d
{
complex r;

r.real = real + d;
r.imag = imag;
return r;
}

complex operator + (double d,complex& z2) //r=d+z2
{
complex r;

r.real = d + z2.real;
r.imag = z2.imag;
return r;
}

complex complex::operator -(complex& z2) //r=z-z2
{
complex r;

r.real = real - z2.real;
r.imag = imag - z2.imag;
return r;
}

complex complex::operator - (double d) //r=z-d
{
complex r;

```

```

r.real = real-d;
r.imag = imag;
return r;
}

complex operator - (double d,complex& z2) //r=d-z2
{
complex r;

r.real = d - z2.real;
r.imag = -z2.imag;
return r;
}

complex complex::operator * (complex& z2) //r=z*z2
{
complex r;

r.real = real*z2.real - imag*z2.imag;
r.imag = real*z2.imag + imag*z2.real;
return r;
}

complex complex::operator * (double d) //r=z*d
{
complex r;

r.real = real*d;
r.imag = imag*d;
return r;
}

complex operator * (double d,complex& z2) //r=d*z2
{
complex r;

r.real = d * z2.real;
r.imag = d * z2.imag;
return r;
}

complex complex::operator / (complex& z2) //r=z/z2;r=0 daca z2=0
{
complex r;
double d = z2.real*z2.real + z2.imag*z2.imag;

if(d == 0) //z2=0
    return r;
r.real = (real*z2.real+imag*z2.imag)/d;
r.imag = (imag*z2.real - real*z2.imag)/d;
return r;
}

```

```

complex complex::operator / (double d) // r=z/d; r=0 daca d=0
{
    complex r;
    if(d == 0) return r;
    r.real = real/d;
    r.imag = imag/d;
    return r;
}

complex operator / (double d,complex& z2) // r=d/z2; r=0 daca z2=0
{
    complex r;
    double dm = z2.real*z2.real + z2.imag*z2.imag;

    if(dm == 0) //z2=0
        return r;
    r.real = z2.real*d/dm;
    r.imag = -z2.imag*d/dm;
    return r;
}

complex complex::operator ^ (int i) //r=z**i; r=0 daca z=0
{
    complex r; // real=imag=0
    double d = modul(); // modulul obiectului curent
    if(d == 0) return r; // obiectul curent este nul; ridicat la puterea i
                        // rezulta obiectul nul

    int n = i < 0 ? -i : i; //n=abs(i)

    if(n == 0) //z**0=1
        return complex(1,0);
    if(i == 1) //z**1=z
        return *this;
    if(i == -1) //z**(-1)=1/z
        r = *this;
    if(n > 1){
        d = pow(d,(double)n); // d=d**n
        double a = arg(); // a=argumentul obiectului complex curent
        a *= n; // a argument*n

        // reducere la primul cerc

        double doipi=2*PI;
        long x = a/doipi;
        a -= x*doipi;
        r.real = d*cos(a);
        r.imag = d*sin(a);
    }
    if(i < 0) //x**i=1/z**n
        r = 1.0/r; // operatorul / este supraincarcat pentru double/complex
    return r;
}

```

```

inline int complex::operator == (complex& z2)
/* returnaza:
   1 daca z=z2;
   0 altfel.
*/
{
    return real == z2.real && imag == z2.imag;
}

inline int complex::operator == (double d)
/* returnaza:
   1 daca z=d;
   0 altfel.
*/
{
    return real == d && imag == 0;
}

inline int operator == (double d,complex& z2)
/* returnaza:
   1 daca d=z2;
   0 altfel.
*/
{
    return d == z2.real && z2.imag == 0;
}

inline double complex::retreal()
/* returnaza partea reala a obiectului complex curent */
{
    return real;
}

inline double complex::retimag()
/* returnaza partea imaginara a obiectului complex curent */
{
    return imag;
}

```

23.2 Să se scrie un program care rezolvă ecuații de gradul întâi cu coeficienți complecsi:

$$ax+b=0 \quad (a \text{ și } b \text{ sunt numere complexe})$$

Programul citește numerele complexe  $a$  și  $b$  și afișează valoarea lui  $x$  dacă  $a \neq 0$ . Se pot rezolva mai multe ecuații de această formă. Pentru a întrerupe execuția programului se tastează sfîrșitul de fișier.

## PROGRAMUL BXXIII2

```

#include "BXXIII1.CPP"

main()
/* - rezolvă ecuații de gradul 1 cu coeficienți complecsi de forma:

```

```

ax + b = 0;
- programul se termina la tastarea sfîrșitului de fișier.

*/
{
    complex a,b,x;

    for(;;){
        if(a.citcomplex("a= ") == 0) exit(0);
        if(b.citcomplex("b= ") == 0) exit(1);
        if(a == 0.0 && b == 0.0){
            printf("ecuație nedeterminată\n");
            continue;
        }
        if(a == 0.0){
            printf("ecuația nu are soluție\n");
            continue;
        }
        x = -b/a;
        x.afiscomplex("x = %g+i*(%g)\n");
    } // sfîrșit for
}

```

#### Observații:

1. Instrucțiunea:  
 $x = -b/a;$   
 conține doi operatori suprareîncărați:  
 – operatorul unar –  
 și  
 – operatorul binar /  
 Operatorul unar este prioritar celui binar.
2. La lansarea programului se afișează:  
 $a=$   
 apoi, se așteaptă tastarea componentelor numărului complex  $a$ .  
 Dacă nu se tastează EOF, după tastarea componentelor numărului  $a$ , se va afișa:  
 $b=$   
 apoi, se așteaptă tastarea componentelor numărului complex  $b$ .
- 23.3 Să se scrie un program care citește numere complexe și le afișează cu partea reală mărită cu 1.

#### PROGRAMUL BXXIII3

```

#include "BXXIII1.CPP"

main()
/* - citește numere complexe și le afișează cu partea reală marită cu 1;
 - programul se termină la întâlnirea sfîrșitului de fișier.
*/
{

```

```

    complex a;

    for(;;){
        if(a.citcomplex("") == 0) exit(0);
        a++; // incrementea partea reală a lui a
        a.afiscomplex(" ");
    } // sfîrșit for
}

```

#### Observații:

1. Funcția *citcomplex* este apelată cu șirul vid. În acest caz se afișează textul:  
 Partea reală =  
 și după tastarea numărului respectiv se afișează textul:  
 Partea imaginara =
2. Partea reală a numărului complex se incrementeaază apelindu-se operatorul unar ++ supraîncărat pentru numere complexe. În program s-a folosit forma postfixată. Se poate utiliza la fel de bine forma prefixată:  
 $+a;$   
 obținându-se același efect.

Menționăm că nu se face nici o diferență între formele postfixate și prefixate ale operatorilor de incrementare și decrementare supraîncărați pentru tipuri abstrakte.

- 23.4 Să se scrie un program care rezolvă ecuații binome cu coeficienți complecsi.

Prin ecuație binomă înțelegem o ecuație de forma:

$$a*x^{**n}+b=0$$

unde:

- $n$  – Este un număr natural pozitiv.  
 $a$  și  $b$  – Sunt numere complexe.

Pentru  $n > 1$ , soluția ecuației este rădăcina de ordinul  $n$  din numărul complex  $-b/a$ .

Rădăcina de ordinul  $n$  dintr-un număr complex se determină din forma trigonometrică a acestuia.

Fie:

$$z=x+i*y \text{ un număr complex}$$

și

$$z=r*(\cos(\alpha)+i*\sin(\alpha)) \text{ forma lui trigonometrică.}$$

Atunci rădăcinile de ordinul  $n$  din  $z$  se calculează ca mai jos:

$$zk = r^{**}(1/n)*(\cos((\alpha+2*k*\pi)/n)+i*\sin((\alpha+2*k*\pi)/n)), \text{ pentru } k=0,1,2,\dots,n-1.$$

Expresia  $r^{**}(1/n)$  are ca valoare rădăcina de ordinul  $n$  din numarul rațional  $r$ . Valoarea ei se determină cu ajutorul funcției *pow*.

#### PROGRAMUL BXXIII4

```
#include "BXXIII1.CPP"

main()
/* - rezolva ecuatii binome(a*x**n + b = 0);
 - programul se termina la tastarea sfîrșitului de fisier. */
{
    complex a,b,x;
    int n;

    for(;;){
        if(a.citcomplex("a= ") == 0) exit(0);
        if(b.citcomplex("b= ") == 0) exit(1);
        if(a == 0.0 && b == 0.0){
            printf("ecuatie nedeterminata\n");
            continue;
        }
        if(a == 0.0){
            printf("ecuatie nu are solutie\n");
            exit(1);
        }

        // a!=0

        int i;

        for(;;){
            printf("gradul ecuatiei = ");
            if((i=scanf("%d",&n)) == EOF){
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(i == 1 && n > 0) break;
            printf("nu s-a tastat un intreg pozitiv\n");
            fflush(stdin);
        }

        // n>0

        x = -b/a;
        if(n > 1 && !(x == 0)) {
            //ecuatie de grad mai mare ca 1
            double r = x.modul();
            double teta = x.arg();
            r = pow(r,1.0/n);
            for(int k = 0; k < n; k++){
                double alfa = (teta + 2*PI*k)/n;
                complex xk(r*cos(alfa),r*sin(alfa));
                printf("x%d=%d,%d",k);
                xk.afiscomplex("");
            }
        }
    }
}
```

```
    }
    else // ecuatie de gradul intii sau solutie nula
        x.afiscomplex("x=%g+i(%g)\n");
    } // sfîrșit for
}
```

23.5 Să se scrie un program care rezolvă ecuații de gradul 2 cu coeficienți complecsi.

#### PROGRAMUL BXXIII5

```
#include "BXXIII1.CPP"

main()
/* rezolva ecuatii de gradul 2 cu coefficientii complecsi:
   a*x*x + b*x + c = 0
*/
{
    complex a,b,c,x;

    for(;;){
        if(a.citcomplex("a= ") == 0) exit(0);
        if(b.citcomplex("b= ") == 0) exit(1);
        if(c.citcomplex("c= ") == 0) exit(1);
        if(a == 0.0 && b == 0.0 && c == 0.0){
            printf("ecuatie nedeterminata\n");
            continue;
        }
        if(a == 0.0 &&b == 0.0){
            printf("ecuatie nu are solutie\n");
            continue;
        }
        if(a == 0.0){
            printf("ecuatie de gradul 1\n");
            x = -c/b;
            x.afiscomplex("x = %g+i*(%g)\n");
            continue;
        }

        // a != 0: ecuatie de gradul 2

        complex doia = 2.0*a;
        complex delta = b*b-4.0*a*c;
        delta =! delta; // radacina patrata din delta
        x = (-b+delta)/doia;
        x.afiscomplex("x1 = %g+i*(%g)\n");
        x = (-b-delta)/doia;
        x.afiscomplex("x2 = %g+i*(%g)\n");
    } // sfîrșit for
}
```

23.6 Să se scrie un program care citește numere complexe și afișează puterea a 5-a a fiecărui număr citit, adunată cu 1.

Programul folosește operatorul  $\wedge$  care a fost supraincărat cu funcția putere pentru numere complexe. Dacă  $z$  este numărul complex citit, atunci programul afișează numărul complex definit de expresia:

$z^*z^*z^*z+1$

care, cu ajutorul operatorului  $\wedge$  se scrie:

$(z^5)+1$

### PROGRAMUL BXXIII6

```
#include "BXXIII1.CPP"

main()
/* - citeste numere complexe;
 - afiseaza puterea a 5-a a fiecarui numar citit adunata cu 1
*/
{
    complex z,v;

    for(;;){
        if(z.citcomplex("") == 0) exit(0);
        v = (z^5) +1;
        z.afiscomplex("z = %g+i*(%g)");
        v.afiscomplex(" z**5+1 = %g+i*(%g)\n");
    }
}
```

#### Observație:

În absența parantezelor, expresia:

$z^5+1$

ridică pe  $z$  la puterea a șasea, deoarece operatorul  $+$  este mai prioritar decit  $\wedge$ .

23.7 Numim *vector* n-dimensional un sistem ordonat de  $n$  numere reale. Se cere să se implementeze conceptul de *vector bidimensional*. În acest scop, vom presupune că vectorul bidimensional este un sistem ordonat de 2 numere de tip *double*.

Asupra vectorilor vom defini următoarele operații:

- suma a doi vectori;
- diferența a doi vectori;
- produsul dintre un vector și un număr;
- negativarea unui vector;
- lungimea unui vector;
- produsul scalar a doi vectori;
- citirea și afișarea componentelor vectorului;
- returnarea componentelor unui vector.

### FIŞIERUL BXXIII7.H

```
class vb {
    double c1,c2;

public:
    // constructor
    vb(double v1=0, double v2=0);

    // suma a doi vectori: r=vb+vb2
    vb operator + (vb& vb2);

    // diferența a doi vectori: r=vb-vb2
    vb operator - (vb& vb2);

    // produsul dintre un vector și un număr: r=vb*a
    vb operator * (double a);

    // produsul dintre un număr și un vector: r=a*vb2
    friend vb operator * (double a, vb& vb2);

    // negativarea unui vector: r = -vb
    vb operator -();

    // lungimea vectorului curent
    double lungime();

    // produsul scalar a 2 vectori r=(vb,vb2)
    double operator *(vb& vb2);

/* - citește componentele unui vector, după afișarea unui text;
 - returnează:
    0 - la sfîrșit de fisier;
    1 - altfel.
*/
    int citvb(char *text);

/* afișează componentele unui vector cu un format standard: (c1,c2) */
    void afisvb();

/* afișează componentele unui vector cu un format dat */
    void afisvb(char *format);

    // returnează prima componentă a vectorului
    double retc1();
    // returnează a doua componentă a vectorului
    double retc2();
};

Funcțiile membru și operatorii se definesc în fișierul de mai jos, de extensie cpp.
```

## FISIERUL BXXIII7

```

#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __BXXIII7_H
#include "BXXIII7.H"
#define __BXXIII7_H
#endif

inline vb::vb(double x,double y) // constructor
{
    c1 = x; c2 = y;
}

vb vb::operator + (vb& vb2) // suma a doi vectori r = vb+vb2
{
    vb vtemp;

    vtemp.c1 = c1 + vb2.c1;
    vtemp.c2 = c2 + vb2.c2;
    return vtemp;
}

vb vb::operator - (vb& vb2) // diferența a doi vectori r=vb-vb2
{
    vb vtemp;

    vtemp.c1 = c1 - vb2.c1;
    vtemp.c2 = c2 - vb2.c2;
    return vtemp;
}

vb vb::operator * (double a) // produsul dintre un vector si un numar
{
    vb vtemp;

    vtemp.c1 = a*c1;
    vtemp.c2 = a*c2;
    return vtemp;
}

inline vb operator *(double a,vb& vb2) //produsul dintre un numar si un vector
{
    return vb2*a;
}

```

```

/* operatorul * este supraincarcat pentru produsul dintre un vector si
   un numar prin functia precedenta */
}

vb vb::operator - () // negativarea unui vector r = -vb
{
    vb vtemp;

    vtemp.c1 = -c1;
    vtemp.c2 = -c2;
    return vtemp;
}

inline double vb::lungime() // lungimea vectorului curent
{
    return sqrt(c1*c1+c2*c2);
}

inline double vb::operator*(vb& vb2) // produsul scalar a doi vectori
{
    return c1*vb2.c1+c2*vb2.c2;
}

int vb::citvb(char *text)
/* - citeste componentele unui vector dupa afisarea unui text definit de parametru;
   - returneaza:
      0 - la intilnirea sfarsitului de fisier;
      1 - altfel.
*/
{
    char t[255];
    double d;

    for(;;) {
        if(strlen(text)) printf(text);
        printf("\n prima componenta a vectorului = ");
        if(gets(t) == 0) return 0; // s-a intilnit EOF
        if(sscanf(t,"%lf",&d) != 1) {
            printf("nu s-a tastat un numar\n");
            printf("se reia citirea\n");
            continue;
        }
        c1 = d;
        printf("componenta a doua a vectorului = ");
        if(gets(t) == 0) return 0; // s-a intilnit EOF
        if(sscanf(t,"%lf",&d) == 1){
            c2 = d;
            return 1;
        }
        printf("nu s-a tastat un numar\n");
        printf("se reia citirea\n");
    }
}

```

```

inline void vb::afisvb()
/* afiseaza componentele vectorului curent folosind un format standard */
{
    printf("vb = (%g,%g)\n",c1,c2);
}

void vb::afisvb(char *format)
/* afiseaza componentele vectorului curent folosind formatul definit de parametru */
{
    if(format && strlen(format)) printf(format,c1,c2);
    else
        /* afisarea cu format standard deoarece parametrul format este si nul, fie
         pointeaza spre sirul vid */
        afisvb(); // afiseaza cu format standard
}

inline double vb::retc1() // returneaza prima componenta a vectorului
{
    return c1;
}

inline double vb::retc2() // returneaza componenta a doua a vectorului
{
    return c2;
}

```

23.8 Să se scrie un program care citește 2 perechi de numere ce reprezintă componentele vectorilor  $a$  și  $b$ , apoi calculează și afișează:

- lungimea fiecărui vector;
- suma:  $a+b$ ;
- diferența:  $a-b$ ;
- suma:  $3*a+7*b$ ;
- suma:  $-a-b$ ;
- produsul scalar al celor doi vectori:  $(a,b)$ .

### PROGRAMUL BXXIII8

```

#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIII7.CPP"

main()
/* - citește componentele vectorilor a și b;
   - calculează și afișează:
     - lungimea vectorilor a și b;
     - suma: a+b;
     - diferența: a-b;
     - suma: 3*a+7*b;
     - suma: -a-b;
     - produsul scalar: (a,b). */

```

```

vb a,b;
char er[] = "s-a intilnit EOF\n";
if(a.citvb("vectorul a\n") == 0){
    printf(er);
    exit(1);
}
if(b.citvb("vectorul b\n") == 0){
    printf(er);
    exit(1);
}

// afiseaza lungimile vectorilor
printf("lungime(a) = %g\n", a.lungime());
printf("lungime(b) = %g\n", b.lungime());

// afiseaza valorile celorlalte expresii
vb c;
c=a+b;
c.afisvb("a+b = %g\t %g\n");
c=a-b;
c.afisvb("a-b = %g\t %g\n");
c=3*a+7*b;
c.afisvb("3*a+7*b = %g\t %g\n");
c=-a-b;
c.afisvb("-a-b = %g\t %g\n");

// afiseaza produsul scalar
printf("(a,b) = %g\n", a*b);
}

```

23.9 Să se definească tipul abstract care implementează conceptul de matrice de ordinul 2.

La definirea unei matrice pătratice de ordinul doi se poate proceda în mai multe moduri:

- tablou bidimensional de ordinul  $2 \times 2$ ;
- tablou unidimensional de două elemente, fiecare element fiind un obiect de tip vector bidimensional;
- două obiecte de tip vector bidimensional.

În exercițiul de față, matricea se implementează printr-un sistem ordonat de două obiecte de tip vector bidimensional. Primul obiect definește prima linie a matricei, iar cel de al doilea obiect definește linia a doua a matricei.

Asupra matricelor pătratice de ordinul doi se definesc următoarele operații:

- suma a două matrice;
- diferența a două matrice;
- produsul a două matrice;
- inversa unei matrice;
- negativarea unei matrice;

- produsul unei matrice cu un număr;
- calculul determinantului unei matrice;
- produsul dintre o matrice și un vector bidimensional.

Alte funcții membru:

- citirea elementelor unei matrice;
- afișarea elementelor unei matrice;
- returnarea liniilor unei matrice.

### FIȘIERUL BXXIII9.H

```
#ifndef __BXXIII7_H
#include "BXXIII7.H" // clasa vb definește vectorii bidimensionali
#define __BXXIII7_H
#endif

class mpdoi {
    vb linial; vb linia2;
public:
    //constructor
    mpdoi(double c11=0,double c12=0,double c21=0,
          double c22=0);

    // funcții membru
    /* - citeste elementele unei matrice;
       - returneaza:
         0 -la intînlarea sfîrșitului de fisier;
         1 - altfel.
    */
    int citmpdoi(char *text);

    /* afiseaza elementele matricei curente folosind un format standard */
    void afismpdoi();

    /* afiseaza elementele matricei curente folosind formatul format1
       pentru linia unu si format2 pentru linia doi */
    void afismpdoi(char *format1, char *format2);

    // returneaza linia intială a matricei
    vb retlin1();

    // returneaza linia a doua a matricei
    vb retlin2();

    // operatori
    // suma a doua matrice
    mpdoi operator + (mpdoi& mpdoi2);

    // diferența a doua matrice
    mpdoi operator - (mpdoi& mpdoi2);

    // produsul a doua matrice
    mpdoi operator * (mpdoi& mpdoi2);
}
```

```
// inversa unei matrice
mpdoi operator !();

// negativarea unei matrice
mpdoi operator -();

// produsul unei matrice cu un numar
mpdoi operator * (double a);

// produsul unei matrice cu un vector
vb operator * (vb& vb2);

// calculul determinantului unei matrice
double operator ~();
};
```

Funcțiile membru și supraincărcarea opereatorilor se definesc într-un fișier separat, de extensie *cpp*.

Amintim că inversa matricei de ordinul 2:

$$\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix}$$

este matricea:

$$\begin{matrix} a_{22}/d & -a_{12}/d \\ -a_{21}/d & a_{11}/d \end{matrix}$$

unde prin *d* s-a notat determinantul matricei inițiale.

În comentariile funcțiilor definite mai jos vom folosi următoarele notații:

<i>r</i>	- matricea rezultat;
<i>mpdoi</i>	- matricea curentă.

### FIȘIERUL BXXIII9

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __BXXIII7_CPP
#include "BXXIII7.CPP"
#define __BXXIII7_CPP
#endif
#ifndef __BXXIII9_H
#include "BXXIII9.H"
#define __BXXIII9_H
#endif

// constructor
inline mpdoi::mpdoi(double c11,double c12,double c21,
                     double c22):linial(c11,c12),linia2(c21,c22)
{
}
```

```

// functii membru

int mpdoi::citmpdoi(char *text)
/* - afiseaza textul spre care pointeaza parametrul text;
 - citeste elementele matricii curente;
 - returneaza:
   0 - la intalnirea sfarsitului de fisier;
   1 - altfel. */
{
    if(text && strlen(text)) printf(text);
    if(linia1.citvb("\n prima linie\n") == 0) return 0;
    if(linia2.citvb("\n linia a doua\n") == 0) return 0;
}

void mpdoi::afismpdoi()
/* afiseaza matricea curenta folosind un format standard */
{
    linia1.afisvb();
    linia2.afisvb();
}

void mpdoi::afismpdoi(char *format1,char *format2)
/* afiseaza matricea curenta folosind formatele:
   format1 - pentru prima linie;
   format2 - pentru linia a doua
*/
{
    linia1.afisvb(format1);
    linia2.afisvb(format2);
}

inline vb mpdoi::retlin1()
// returneaza prima linie a matricii curente
{
    return linia1;
}

inline vb mpdoi::retlin2()
// returneaza linia a doua a matricii curente
{
    return linia2;
}
// operatori

mpdoi mpdoi::operator + (mpdoi& mpdoi2) // r = mpdoi+mpdoi2
{
    mpdoi rtemp;
    rtemp.linial = linial+mpdoi2.linial;
    rtemp.linia2 = linia2+mpdoi2.linia2;
    return rtemp;
}

```

```

mpdoi mpdoi::operator - (mpdoi& mpdoi2) // r = mpdoi-mpdoi2
{
    mpdoi rtemp;
    rtemp.linial = linial-mpdoi2.linial;
    rtemp.linia2 = linia2-mpdoi2.linia2;
    return rtemp;
}

mpdoi mpdoi::operator * (double a) // r = mpdoi*a
{
    mpdoi rtemp;
    rtemp.linial = linial*a;
    rtemp.linia2 = linia2*a;
    return rtemp;
}

vb mpdoi::operator * (vb& vb2) // r = mpdoi*vb2
{
    vb rtemp(linia1*vb2,linia2*vb2);
    return rtemp;
}

mpdoi mpdoi::operator * (mpdoi& mpdoi2) // r = mpdoi*mpdoi2
{
    // coloana intia a matricii mpdoi
    vb col1(mpdoi2.linial.retc1(),mpdoi2.linia2.retc1());
    // coloana a doua a matricii mpdoi2
    vb col2(mpdoi2.linial.retc2(),mpdoi2.linia2.retc2());
    // (r11,r21)= mpdoi * col1
    vb rvb1;
    rvb1 = *this*col1;
    // (r12,r22)
    vb rvb2;
    rvb2 = *this*col2;
    // matricea produs
    mpdoi rtemp(rvb1.retc1(),rvb2.retc1(),
                rvb1.retc2(),rvb2.retc2());
    return rtemp;
}

inline double mpdoi::operator~() // calculeaza determinantul matricii curente
{
    return linial.retc1()*linia2.retc2() -
           linial.retc2()*linia2.retc1();
}

mpdoi mpdoi::operator -() // r = -mpdoi
{

```

```

mpdoi rtemp;

rtemp.linial = -linial;
rtemp.linia2 = -linia2;
return rtemp;
}

mpdoi mpdoi::operator }()
// returneaza inversa matricei curente
{
    double d = ~*this; // calculeaza determinantul matricei curente
    if(d != 0){
        mpdoi rtemp(linia2.retc2()/d, -linial.retc2()/d,
                    -linia2.retc1()/d, linial.retc1()/d);
        return rtemp;
    }
    printf("matricea are determinantul nul\n");
    mpdoi rtemp;
    return rtemp; // matricea nula
}

```

23.10 Să se scrie un program care rezolvă un sistem de două ecuații liniare cu două necunoscute.

Fie sistemul:

$$(1) \ ax = b$$

unde:

- $a$  - Este o matrice pătratică de ordinul 2.
- $b$  - Este o matrice coloană de ordinul  $2 \times 1$ .

Matricea  $b$  corespunde conceptului de vector bidimensional, implementat cu ajutorul clasei  $vb$ , definită în exercițiul 23.7.

Soluția sistemului (1) există și este unică dacă determinantul matricei  $a$  este diferit de zero. Dacă notăm cu  $inv_a$  inversa matricei  $a$ , atunci soluția sistemului (1) se poate exprima astfel:

$$(2) \ x = inv_a * b$$

Programul din acest exercițiu citește elementele matricelor  $a$  și  $b$ , calculează soluția  $x$  cu ajutorul relației (2) și afișează soluția respectivă. Apoi, se determină vectorul reziduu  $r$  cu ajutorul relației:

$$(3) \ r = ax - b$$

unde:

- $x$  - Este soluția determinată mai sus.

În final, se afișează vectorul reziduu.

## PROGRAMUL BXXIII10

```

#ifndef __STDLIB_H
#include <stdlib.h>

```

```

#define __STDLIB_H
#endif
#include "BXXIII9.CPP"

main()
/* - citesc elementele matricelor a și b;
   - calculeaza si afiseaza solutia sistemului:
     ax = b
   - calculeaza si afiseaza vectorul reziduu:
     r = ax - b
*/
{
    mpdoi a;
    vb b,x,r;
    char er[] = "S-a tastat EOF\n";

    // citeste matricea a
    if(a.citmpdoi("matricea a\n") == 0){
        printf(er);
        exit(1);
    }
    if(~a == 0){ // determinantul nul
        printf("determinantul sistemului este nul\n");
        exit(1);
    }

    // citeste vectorul b
    if(b.citvb("termenul liber\n") == 0){
        printf(er);
        exit(1);
    }

    // calculeaza solutia sistemului
    x = !a *b; // x = inva *b

    // afiseaza solutia sistemului
    x.afisvb("x1 = %.15g\t x2 = %.15g\n");

    // calculeaza vectorul reziduu
    r = a*x-b;

    // afiseaza vectorul reziduu
    printf("vectorul reziduu\n");
    r.afisvb("linial = %.16g\t linia2 = %.16g\n");
}

```

## 23.1. Supraîncărcarea operatorilor new și delete

În capitolul 20 s-a arătat că operatorii *new* și *delete* pot fi utilizati pentru alocarea dinamică a datelor în memoria *heap*. Operatorul *new* alocă, la execuție, o dată în memoria *heap*. Operatorul *delete* dezalocă (distrugе) o dată care a fost

alocată, în prealabil, prin operatorul *new*. Acești operatori pot fi utilizati atât pentru a gestiona date de tipuri predefinite cât și date de tip abstract, adică obiecte.

Operatorul *new* are ca operand numele tipului datei pentru care se aloca memorie:

**new nume\_tip**

Printr-o expresie de această formă se aloca în memoria *heap* o zonă de memorie necesară pentru a păstra o dată de tip *nume\_tip*. Rezultatul aplicării operatorului *new* este adresa de început a zonei de memorie alocate. Această adresă este un pointer spre tipul *nume\_tip*.

**Exemple:**

1. `int *p = new int;`

Se aloca o zonă de memorie necesară pentru a păstra o dată de tip *int*. Adresa de început a acestei zone reprezintă un pointer spre *int* și se atribuie lui *p*.

2. `double *q = new double;`

Se aloca o zonă de memorie necesară pentru a păstra o dată de tip *double*. Adresa de început a acestei zone reprezintă un pointer spre *double* și se atribuie lui *q*.

3. `class vb {`  
    `double c1,c2;`  
    `public:`  
        `vb()`  
        `{`  
            `c1 = 0;`  
            `c2 = 0;`  
        `}`  
        `...`  
    `}`  
`vb *pvb = new vb;`

Se aloca zonă de memorie pentru a păstra un obiect de tip *vb*. Adresa de început a acestei zone reprezintă un pointer spre *vb* și se atribuie lui *pvb*.

\* La alocare, se apelează automat constructorul implicit.

Operatorul *new* se poate aplica pentru tipurile abstracte el având o supraincărcare globală. Această supraincărcare globală este numită *standard* sau *predefinită*.

În mod analog, operatorul *delete* are o supraincărcare globală și el poate fi utilizat pentru a dezaloca (distrunge) datele pentru care s-a alocat memorie cu ajutorul lui *new*. Astfel, pentru a distrunge datele alocate în exemplele de mai sus, nu avem decât să utilizăm expresiile:

- `delete p;` pentru data alocată în exemplul 1;
- `delete q;` pentru data alocată în exemplul 2;
- `delete pvb;` pentru obiectul alocat în exemplul 3.

Amintim că la distrugerea obiectelor se apelează destructorul clasei (cel definit de utilizator sau cel implicit dacă nu există un astfel de destructor).

Operatorul *new* se poate utiliza nu numai pentru a aloca memorie pentru o dată, ci și pentru a inițializa data respectivă.

**Exemple:**

1. `int *p = new int(10);`

În zona alocată în memoria *heap* se păstrează valoarea 10 de tip *int*, iar *p* pointează spre zona respectivă.

2. `class vb {`

`double c1,c2;`

`public:`

`vb()`

`{`

`c1 = 0;`

`c2 = 0;`

`}`

`vb(double v1,double v2)`

`{`

`c1 = v1;`

`c2 = v2;`

`}`

`...`

`};`

`vb *pvb = new vb(1,2);`

În zona alocată în memoria *heap* se păstrează un obiect care are componentele:

*c1 = 1* și *c2 = 2*

În acest caz, se apelează în mod automat constructorul care are parametri și care realizează inițializarea componentelor obiectului instanțiat. Pointerul *pvb* pointează spre obiectul instanțiat în acest fel.

Operatorii *new* și *delete* permit alocarea și respectiv dezalocarea de tablouri în memoria *heap*. În acest caz, se utilizează construcții de felul celor de mai jos.

**Exemple:**

1. `int *p = new int[10];`

Se aloca o zonă de memorie capabilă pentru a memora 10 date de tip *int* ( $10 * \text{sizeof}(\text{int})$ ).

Adresa de început a acestei zone se atribuie lui *p*. La elementele respective ne putem referi prin expresii de forma:

*p[0], p[1], ..., p[9]*

sau

*\*p, \*(p+1), ..., \*(p+9)*.

Zona respectivă se eliberează aplicind operatorul *delete* în mod obișnuit:

delete p; ? dilit tip

2. char \*q = new char[100];  
Se rezervă o zonă de memorie de 100 de octeți și adresa de inceput a zonei respective se atribuie lui *q*.  
Dezalocarea se realizează printr-o instrucție de forma:  

```
delete q;
```
3. class vb {
 double c1,c2;
public:
 vb()
 {c1 = 0; c2 = 0; }

 vb(double x,double y)
 {c1 = x; c2 = y; }

 ...
};

vb \*ptvb;
ptvb = new vb[100];

Se rezervă zonă de memorie pentru 100 de obiecte de tip *vb*. În acest scop se utilizează supraincărcarea standard a operatorului *new*. În acest caz, se apelează constructorul implicit de 100 de ori.

Adresa de inceput a zonei rezervate pentru cele 100 de elemente se atribuie pointerului *ptvb*.

Zona de memorie alocată în acest fel se eliberează printr-o instrucție de forma:

```
delete ptvb;
```

La obiectele alocate mai sus ne putem referi prin construcții de forma:

\*ptvb, \*(ptvb+1), ..., \*(ptvb+99). ←

La alocarea tablourilor cu ajutorul operatorului *new* nu se pot face inițializări. De exemplu, dacă se dorește inițializarea elementelor tabloului de mai sus în așa fel încit elementul:

\*(ptvb+i)

să aibă valoarea inițială (*i,i*), se poate apela o funcție membru sau prieten a clasei, definită ca mai jos.

```
void vb::initab(int n)
{
    vb *t;
    t = this;

    for(i = 0; i < n;i++,t++) *t = vb(i,i);
}
```

Pentru a inițializa tabloul de mai sus, vom apela funcția membru astfel:

```
ptvb -> initab(100);
```

Amintim că în instrucțiunea:

*\*t = vb(i,i);*

expresia din partea dreaptă definește un obiect *anonim* (fără nume). El se initializează prin constructorul clasei *vb*, apoi obiectul respectiv se copiază în zona spre care pointeză *t*. Obiectul *anonim* este distrus automat.

În rezumat, operatorul *new* poate fi apelat pentru a rezerva memorie atât pentru date de tipuri predefinite, cât și pentru tipuri abstracte. Datele alocate în acest fel pot fi inițializate.

Să pot aloca tablouri, dar în acest caz nu se mai pot face inițializări. În cazul obiectelor trebuie să precizăm următoarele:

1. Nu este obligatoriu să existe constructor definit de programator.
2. Dacă există un constructor fără parametri definit de utilizator, atunci el se aplică automat la alocări prin expresii de forma:

**new nume\_clasa**

3. Dacă se fac alocări cu inițializare:

**new nume\_clasa(par1,par2,...,parN)**

atunci clasa trebuie să aibă un constructor care să poată realiza inițializarea respectivă.

4. Dacă se fac alocări de forma:

**new nume\_clasa[nr\_elemente]**

atunci se utilizează supraincărcarea standard (predefinită) a operatorului *new*. Dacă clasa are un constructor implicit definit de programator, acesta se apelează la o astfel de alocare de *nr\_elemente* ori.

Obiectele alocate cu ajutorul lui *new* se distrug cu ajutorul operatorului *delete*. În acest scop se utilizează o instrucție de forma:

```
delete p;
```

unde:

*p* - Este un pointer spre obiectul care se distrugă.

Dacă clasa are un destructor definit de programator, atunci instrucția de mai sus implică apelul destructorului respectiv.

Obiectele tablou, alocate printr-o expresie de forma:

**p = new nume\_clasa[nr\_elemente]**

se pot distrugă prin una din instrucțiunile:

**delete p;**

sau

**delete [nr\_elemente]p;**

Prima instrucție apelează în mod automat destructorul clasei (o singură dată, spre deosebire de cea de a doua, care distrug obiectul apelind de

nr\_elemente ori destructorul clasei.

Utilizatorul poate supraincărca acești operatori în cazul în care supraincărcarea standard nu este suficientă.

Supraincărcarea operatorilor *new* și *delete* prezintă unele diferențe față de supraincărcarea celorlalți operatori.

O primă diferență constă în aceea că, supraincărcarea operatorilor *new* și *delete* se realizează cu ajutorul *funcțiilor membru statică*, spre deosebire de ceilalți operatori care se supraincărcă prin funcții membru care nu sunt statice sau prin funcții prieten.

Funcțiile care supraincărca operatorii *new* și *delete* sunt assimilate, în mod implicit, ca funcții membru statice, așa că nu mai este nevoie să specificăm explicit acest lucru.

O altă particularitate a supraincărcării acestor operatori este antetul funcțiilor prin care se realizează supraincărcarea.

Pentru operatorul *new* se utilizează următorul prototip:

```
void *operator new(size_t lung);
```

Funcția returnează un pointer universal a cărui valoare este adresa de început a zonei de memorie rezervată în memoria *heap*.

Tipul *size\_t* este definit în fișierele de bibliotecă *stddef.h* și *stdlib.h* astfel:

```
typedef unsigned size_t;
```

Tipul *size\_t* trebuie să fie prezent în antetul funcțiilor pentru supraincărcarea operatorului *new* (el deși este un alt nume pentru *unsigned*, nu poate fi înlocuit prin *unsigned*).

Antetul funcției pentru supraincărcarea operatorului *new* este de forma:

```
void *nume_clasa::operator new(size_t lung)
```

Parametrul *lung* are ca valoare dimensiunea în octeți a zonei de memorie care se rezervă în memoria *heap*.

O altă particularitate a supraincărcării operatorului *new* este faptul că la aplicarea lui nu se indică nici o valoare pentru parametrul formal *lung*. Aceasta, deoarece compilatorul calculează, în mod automat, dimensiunea obiectului pentru care se rezerva zonă de memorie și această dimensiune se atribuie parametrului *lung*.

În felul acesta, operatorul *new* supraincărat se utilizează în mod obișnuit în expresii de forma:

```
new nume_clasa
```

unde:

*nume\_clasa* - Este numele clasei pentru care este supraincărat.

Aplicarea operatorului *new* supraincărat de utilizator apelează în mod automat constructorul corespunzător clasei, dacă există constructori definiți pentru clasa respectivă sau pe cel implicit definit de compilator dacă clasa nu are

constructori. De aceea, la aplicarea operatorului *new*, pot să fie prezenti parametri pentru constructorii clasei.

În general, operatorul *new* se aplică la un operand folosind urmatorul format:

```
new nume_clasa(par1,par2,...,parN)
```

unde:

*par1,par2,...,parN* - Sunt expresii și reprezintă parametri pentru apelul unui constructor al clasei:  
*nume\_clasa*.

Supraincărcarea predefinită (globală) a operatorului *new* poate fi apelată și în cazul în care acesta este supraincărat de utilizator, folosind operatorul de rezoluție. Astfel, o expresie de forma:

```
::new nume_clasa
```

aplică operatorul *new* predefinit, care, așa cum s-a afirmat mai sus, are o supraincărcare standard, globală.

Pentru operatorul *delete*, de asemenea există particularități specifice. Prototipul funcției pentru supraincărcarea operatorului *delete* se poate specifica astfel:

```
void operator delete(void *p);
```

Ca și în cazul operatorului *new*, funcția pentru supraincărcarea operatorului este în mod implicit o funcție membru *statică*.

Antetul funcției este de forma:

```
void nume_clasa::operator delete (void *p)
```

Operatorul are ca operand un pointer spre obiectul care se dezalocă. Valoarea acestui pointer trebuie să fie cea definită prin aplicarea operatorului *new* la construirea obiectului.

**Exemplu:**

```
nume_clasa *p = new nume_clasa;  
/* - se aloca zona de memorie pentru un obiect de tip nume_clasa;  
   - adresa de început a zonei de memorie alocată se atribuie lui p */  
...  
delete p; /* se eliberează zona de memorie alocată prin new */
```

La aplicarea operatorului *delete* se apelează în mod automat destructorul clasei dacă există un destructor definit de utilizator. În caz contrar, se apelează destructorul implicit definit de compilator.

Ca și în cazul operatorului *new*, dacă operatorul *delete* a fost supraincărat pentru tipul abstract *nume\_clasa*, atunci operatorul *delete* predefinit (global) poate fi aplicat precedindu-l de operatorul de rezoluție:

```
::delete nume_clasa;
```

Operatorii *new* și *delete* supraincărați de utilizator nu se folosesc la alocarea

și dezalocarea tablourilor de obiecte.

În cazul alocării și distrugerii tablourilor de obiecte se utilizează operatorii *new* și *delete* standard (globali).

Operatorii *new* și *delete* definiti de utilizator au un avantaj față de cei standard și anume ei sunt mai economici din punct de vedere al timpului de execuție și al memoriei consumate pentru gestiunea obiectelor dinamice.

În corpul funcțiilor pentru definirea supraincărării operatorilor *new* și *delete* se pot utiliza versiunile standard pentru acești operatori.

În încheiere amintim că, funcția pentru supraincărarea operatorului *delete* are un al doilea parametru facultativ, de tip *size\_t*, care este gestionat automat de compilator. El nu este obligatoriu să apară în antetul funcției pentru supraincărarea operatorului *delete*. Antetul general al funcției pentru supraincărarea operatorului *delete* este următorul:

```
void nume_clasa::operator delete(void *p, size_t lung)
```

#### Exemplu:

Fie clasa de mai jos, pentru care s-au supraincărat operatorii *new* și *delete*:

```
class nume_clasa {  
public:  
    nume_clasa();  
    void *operator new(size_t);  
    void operator delete(void *);  
    ...  
    ~nume_clasa();  
};
```

Considerăm următoarele declarații:

1. char \*pcar = new char;
2. char \*tc = new char[100];
3. double \*td = new double[100];
4. nume\_clasa \*pnume = new nume\_clasa;
5. nume\_clasa \*tnume = new nume\_clasa[100];

În cazul primei declarații se aplică operatorul *new* standard.

Se rezervă zonă pentru un caracter (un octet) și adresa ei se atribuie lui *pcar*. Aceasta zonă se eliberează aplicind operatorul *delete* standard:

```
delete pcar;
```

La declarația a doua se aplică operatorul *new* standard pentru a aloca o zonă de 100 de octeți. Adresa de început a acestei zone se atribuie lui *tc*. Zona respectivă se eliberează aplicind operatorul *delete* standard:

```
delete tc;
```

Declarația a treia este analogă cu a doua, deosebirea constă în aceea că, în acest caz, se alocă 100\*sizcof(double) octeți în loc de 100 de octeți.

La declarația a patra se alocă zonă pentru un obiect de tip *nume\_clasa*.

La alocare se apelează constructorul implicit al clasei, definit de utilizator.

Alocarea se face apelindu-se operatorul *new* supraincărat de utilizator.

Adresa de început a zonei de memorie alocată în acest fel se atribuie lui *pnume*.

Zona respectivă se eliberează aplicind operatorul *delete* supraincărat definit de utilizator:

```
delete pnume;
```

Această instrucțiune apelează destructorul clasei.

La ultima declarație se alocă zonă pentru 100 de elemente de tip *nume\_clasa*.

Se apelează operatorul *new* standard.

Adresa de început a zonei rezervate se atribuie pointerului *tnume*.

La alocare se apelează constructorul implicit de o sută de ori.

Zona respectivă se eliberează aplicind operatorul *delete* standard:

```
delete[100] tnume;
```

sau

```
delete tnume;
```

În primul caz, se apelează destructorul de o sută de ori. În cel de al doilea caz, destructorul se apelează o singură dată.

#### Exerciții:

23.11 Să se definească tipul abstract *tnod* care are următoarele date membru:

- *cuvint*: pointer spre un sir de caractere;
- *frecv*: intreg utilizat ca numărator de cuvinte;
- *urm*: pointer spre *tnod*;
- *ind*: indicator pentru citirea sfîrșitului de fișier.

Clasa care implementează tipul abstract *tnod* are funcții membru care realizează următoarele:

- constructor;
- destructor;
- afișează datele membru *cuvint* și *frecv*;
- returnează valoarea lui *ind*;
- supraincarcă operatorii *new* și *delete*;
- operatorul pentru incrementarea numărătorului *frecv*.

Clasa *slist* este clasă prieten pentru clasa *tnod*.

## FIŞIERUL BXXIII11.H

```
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __Boolean
#define __Boolean

enum Boolean {false,true};

#endif

class tnod {
    char *cuvint;
    int frecv;
    tnod *urm;
    static Boolean ind;
    friend class slist;
public:
    tnod();
    ~tnod();
    void afistnod();
    void *operator new(size_t);
    void operator delete(void *);
    void operator++();
    static Boolean retind();
};
```

Funcțiile membru ale clasei *tnod* se definesc în fișierul de extensie *CPP*, de mai jos.

Indicatorul *ind* are valoarea *false* atât timp cât nu s-a întâlnit sfîrșitul de fișier.

## FIŞIERUL BXXIII11

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __BXXIII11_H
#include "BXXIII11.H"
#define __BXXIII11_H
#endif

// Initializarea variabilei ind

Boolean tnod::ind = false;

tnod::tnod()
/* - citește cuvintul curent dacă ind = false;
```

```
- păstrează cuvintul citit în memoria heap;
- realizează initializările:
    frecv = 1;
    urm = 0.
*/
{
    char t[255];
    int sf;
    cuvint = 0;

    if(tnod::ind == false){
        while((sf = scanf("%s", t)) != EOF){
            char *p = t;
            int c;

            // saltează caractere care nu sunt litere
            while((c = *p) && (c < 'A' || c > 'Z' &&
                c < 'a' || c > 'z'))
                p++;
            if(c == 0) // nu sunt litere
                continue;

            // păstrează începutul cuvintului
            char *q = p;

            // cauta sfîrșitul cuvintului
            while((c = *p) && (c >= 'A' && c <= 'Z' ||
                c >= 'a' && c <= 'z'))
                p++;
            *p = '\0'; // caracterul NUL la sfîrșitul cuvintului

            // rezerva zona pentru cuvint în memoria heap
            // se apelează operatorul new standard

            printf("tnod::tnod %lu\n", coreleft());
            if((cuvint = new char[strlen(q)+1]) == 0){
                printf("memorie insuficientă\n");
                exit(1);
            }
            printf("tnod::tnod %lu\n", coreleft());

            // se transferă cuvintul în zona rezervată
            strcpy(cuvint, q);

            // initializare
            frecv = 1;
            urm = 0; // pointerul nul
            break;
        } // sfîrșit while

        if(sf == EOF) // s-a întâlnit EOF
            tnod::ind = true;
    }
} // sfîrșit constructor
```

```

inline tnod::~tnod()
{
    // se apeleaza operatorul delete standard
    printf("\ndestructor tnod %lu\n", coreleft());
    delete cuvint;
    printf("\ndestructor tnod %lu\n", coreleft());
}

inline void tnod::afistnod() // afiseaza cuvintul si frecventa
{
    printf("cuvintul=%s\t are frecventa=%d\n", cuvint, freqv);
}

void *tnod::operator new(size_t lung) // suprareincarcarea operatorului new
{
    tnod *p;

    // se aplica operatorul new standard
    printf(" operator new %lu\n", coreleft());
    p=(tnod *)new char[lung];
    printf(" operator new %lu\n", coreleft());
    return p;
}

void tnod::operator delete(void *p) // suprareincarcarea operatorului delete
{
    // se aplica operatorul delete standard
    printf(" operator delete %lu\n", coreleft());
    ::delete p;
    printf(" operator delete %lu\n", coreleft());
}

void tnod::operator++() // incrementeaza freqv
{
    freqv++;
}

Boolean tnod::retind() // returnaza valoarea ind
{
    return tnod::ind;
}

```

#### Observație:

Funcția *coreleft* returnează dimensiunea, în octeți, a memoriei libere în momentul apelului. Ea a fost apelată în mai multe funcții pentru a putea urmări momentele în care se alocă și dezalocă obiectele care sunt instanțieri ale clasei *tnod*.

23.12 Să se definiște tipul abstract *slist* pentru implementarea listei simplu înlățuite.

Listele simplu înlățuite au fost implementate în limbajul C în capitolul 11.

Pentru gestiunea listelor simplu înlățuite s-au folosit două variabile *prim* și *ultim* care sunt pointeri spre primul și respectiv ultimul nod al listei.

ACESTE VALORI SE VOR UTILIZA ȘI ÎN CAZUL DE FAȚĂ CU ACELAȘI SCOP. ELE SUNT DATE MEMBRU PROTEJATE ALE TIPULUI ABSTRACT *slist*.

Avantajul implementării tipului abstract *slist* este acela că, se pot defini simultan și simplu orice obiecte de tip listă.

Funcțiile membru ale tipului abstract *slist* sunt:

- constructor implicit;
- destructor;
- adăugarea unui nod după ultimul nod al listei;
- inserarea unui nod înaintea primului nod al listei;
- stergerea primului nod al listei;
- stergerea ultimului nod al listei;
- căutarea în listă a unui nod;
- afișarea datelor aflate în nodurile listei.

Nodurile listei sunt obiecte de tip *tnod*, tip definit în exercițiul 23.11.

#### FIŞIERUL BXXIII12.H

```

#ifndef __BXXIII11_H
#include "BXXIII11.CPP"
#define __BXXIII11_H
#endif

class slist {
    tnod *prim;
    tnod *ultim;
public:
    slist(); // constructor
    ~slist(); // destructor
    tnod *adauga(tnod *); // adauga un nod după cel spre care pointează ultim
    tnod *insereaza(tnod *p); // inserează un nod înaintea celui spre care
                                // pointează prim
    void sprim(); // sterge primul nod din lista
    void sultim(); // sterge ultimul nod al listei

    tnod *cauta(tnod *p);
    /* - cauta nodul din lista pentru care p->cuvint pointeaza
       spre un sir identic cu cel spre care pointeaza cuvint din nod;
       - returneaza pointerul spre nodul respectiv sau zero daca nu exista un astfel de nod */

    void afislist(); // afiseaza datele din nodurile liste
};

Funcțiile membru ale clasei slist se definesc în fișierul de extensie CPP, de mai jos.

```

#### FIŞIERUL BXXIII12

```

#ifndef __BXXIII12_H
#include "BXXIII12.H"

```

```

#define __BXXIIII12_H
#endif

inline slist::slist() // constructor
{
    prim = ultim = 0;
}

slist::~slist() // destructor; distruge toate nodurile listei
{
    tnod *p,*q;

    for(p=prim;p;p=q){
        q=p->urm; // pointer spre nodul urmator
        delete p; // operatorul delete supraincarcat
    }
    prim = ultim = 0;
}

tnod *slist::adauga(tnod *p) // adauga un nod dupa ultimul nod al listei
{
    if(prim == 0) // lista vida
        prim = p;
    else ultim->urm = p;
    ultim = p;
    return p;
}

tnod *slist::insereaza(tnod *p)
// insereaza un nod inaintea celui spre care pointeaza prim
{
    if(prim==0) // lista vida
        ultim = p;
    else p->urm = prim;
    prim = p;
    return p;
}

void slist::sprim() // sterge primul nod din lista
{
    if(prim){
        tnod *p = prim->urm;
        delete prim; // se aplica operatorul delete supraincarcat
        prim = p;
        if(prim == 0) ultim = 0; // lista a devenit vida
    }else
        printf("lista vida\n");
}

void slist::sultim() // sterge ultimul nod al listei
{
    if(prim){
        tnod *p,*pl;

```

```

        // p - pointeaza spre nodul curent
        p = prim;

        // pl - pointeaza spre nodul precedent
        pl = 0;
        while(p != ultim) { // parurge lista
            pl = p;
            p = p->urm;
        }
        // p - pointeaza spre ultimul nod;
        // pl - pointeaza spre nodul precedent
        if(pl == 0){ // lista are un singur nod si devine vida
            delete p; // se aplica operatorul supraincarcat
            prim = ultim = 0;
            return;
        }

        // nodul spre care pointeaza pl devine ultimul nod al listei
        pl->urm = 0;
        ultim = pl;

        // se sterge nodul spre care pointeaza p
        delete p; // se aplica operatorul supraincarcat
    }else
        printf("lista vida\n");
}

tnod *slist::cauta(tnod *p)
/* - cauta nodul pentru care cuvant pointeaza spre un sir identic cu cel spre care
   pointeaza p->cuvint;
   - returneaza pointerul spre nodul respectiv sau 0 daca nu exista un astfel de nod. */
{
    tnod *q;

    for(q=prim;q;q=q->urm)
        if(strcmp( p->cuvint, q->cuvint ) == 0) return q;
    return 0;
}

void slist::afislist()
/* - afiseaza pentru fiecare nod:
   - cuvint
   - si
   - freev.
*/
{
    int nrlin = 1;

    for(tnod *p = prim; p; p = p->urm){
        p->afistnod();
        nrlin++;
        if(nrlin%23==0){
            printf("Actionati o tasta pentru a continua\n");
        }
    }
}

```

```

        getch();
    }
}

```

23.13 Să se scrie un program care citește un text și afișează frecvența de apariție a fiecărui cuvint din text. Nu se face distincție între literele mari și mici.

Această problemă a fost rezolvată în mai multe moduri în capitolele precedente.

În particular, în capitolul 11 se dă o rezolvare folosind listele simplu înlanțuite.

Programul de față utilizează aceeași metodă.

### PROGRAMUL BXIII13

```

#include <alloc.h>
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXIII12.CPP"

main()
/* citeste un text si afiseaza frecventa cuvintelor citite */
{
    slist lista;
    tnod *nod,*p;

    printf("%lu\n",coreleft());
    while(tnod::retind() == false)
    // nu s-a ajuns la sfîrșitul textului
    {
        nod = new tnod; // se aplică operatorul new supraincarcat
        // cauta nodul în lista
        if(tnod::retind()==false)
            if(p = lista.cauta(nod)){
                // cuvintul există deja în lista
                (*p)++; // incrementează frecv
                delete nod; // se aplică operatorul delete supraincarcat
            }else
                lista.adauga(nod);
                /* nodul spre care pointează nod conține un cuvint negasit în lista;
                   se adaugă la lista */
        else delete nod;
        printf("Pentru a continua actionati o tasta\n");
        getch();
    } // sfîrșit while

    // listaza nodurile listei
    lista.afislist();
}

```

### 23.2. Supraîncărcarea operatorului =

În limbajul C++ se pot face atribuiri de obiecte care sunt instanțieri ale aceluiși clasei.

**Exemplu:**

Fie instanțierile:

```

nume_clasa obiect1,obiect2;
...

```

O atribuire de forma:

```

obiect2 = obiect1;

```

este acceptată de compilator.

La atribuirea de obiecte, în mod implicit, se atribuie datele membru ale obiectului din dreapta operatorului = la datele membru corespunzătoare ale obiectului din stînga aceluiași operator. Astfel de atribuiri se pot utiliza și în declarații.

**Exemplu:**

```

class complex {
    double real;
    double imag;
public:
    complex(double x=0,double y=0)
    {
        real = x;
        imag = y;
    }
    ...
};

complex z(1,2); // z = 1+2i
complex z1 = z; // z1 = 1+2i
complex z2;
...
z2 = z1; // z2 = 1+2i

```

Atât în cazul declarației lui *z1*, cât și în cazul instrucțiunii de atribuire pentru *z2*, se copiază componente date ale obiectului *complex* din dreapta caracterului "=" în zona de memorie alocată componentelor date ale obiectului aflat în stînga aceluiași caracter.

În cazul declarației lui *z1*, caracterul "=" se utilizează pentru a realiza o inițializare, iar în cazul instrucțiunii:

```

z2 = z1;

```

același caracter se utilizează pentru a realiza o atribuire.

Ambele operații sint definite in mod implicit și se realizează așa cum s-a indicat mai sus, adică prin simpla copiere a datelor membru.

B. Stroustrup numește "bitwise copy" (copiere la nivel de biți) copierile de acest fel.

În cazul obiectelor clasei *complex*, copierea din cadrul operației de inițializare este identică cu cea utilizată în cadrul operației de atribuire.

În general, cele două operații nu sint identice. De asemenea, copierea de tip "bitwise copy" nu este totdeauna suficientă. Acest fapt rezultă imediat dacă reluăm exemplul pentru implementarea tipului abstract *string* (vezi 22.9) sau *sir* (vezi exercițiul 22.12.).

Considerăm clasa *sir*, definită ca mai jos:

```
class sir {
    char *psir;
    int lung;

public:
    sir(char *s);
    sir(int nrcar = 70);
    int retlung(); // returneaza lungimea sirului
    void afsir(); // afiseaza sirul definit de psir
    int citsir(); // citeste un sir de caractere
    ~sir() (delete psir;}
};
```

Funcțiile membru de mai sus, sint definite in exercițiul 22.12.

Fie declarația:

```
sir sir1("C++ este un C mai bun");
```

La instanțierea obiectului *sir1* se apeleză constructorul:

```
sir(char *s)
```

Acesta rezervă zonă de memorie în memoria *heap* pentru textul:

C++ este un C mai bun

și atribuie adresa de început a acestei zone pointerului:

```
sir1.psir
```

De asemenea, se atribuie datei membru:

```
sir1.lung
```

valoarea 21 (numărul de caractere din compunerea textului de inițializare a obiectului *sir1*, fără a considera și caracterul NUL).

Să considerăm declarația pentru instanțierea obiectului *sir2* cu inițializarea acestuia folosind obiectul *sir1*:

```
(1) sir sir2 = sir1;
```

Utilizatorul folosește o astfel de declarație pentru a obține același efect ca și cind ar utiliza declarația:

```
(2) sir sir2("C++ este un C mai bun");
```

În realitate, cele două declarații nu realizează același lucru.

În cazul declarației (1), la instanțierea lui *sir2* se apeleză constructorul:

```
sir(int nrcar = 70)
```

care rezervă o zonă de memorie de 70 de octeți în care se păstrează sirul vid.

Adresa de început a acestei zone se atribuie lui *sir2.psir*, apoi se copiază bit cu bit (bitwise copy) valorile componentelor obiectului *sir1* (*sir1.psir* și *sir1.lung*) în zonele de memorie alocate componentelor corespunzătoare ale obiectului *sir2*. În felul acesta, *sir2.psir* are aceeași valoare ca și *sir1.psir*. Zona de memorie de 70 de octeți alocată în memoria *heap* este pierdută și nu se mai poate face acces la ea. De asemenea, textul de inițializare este păstrat într-un singur exemplar, situație care, de obicei, va conduce la erori în prelucrarea ulterioară a obiectelor *sir1* și *sir2*.

De asemenea, la distrugerea obiectelor *sir1* și *sir2* ar urma ca textul de inițializare, păstrat într-un singur exemplar, să fie eliberat din memoria *heap* de două ori. De aceea, copierea implicită bit cu bit a obiectelor nu este o soluție corectă pentru obiectele clasei *sir*.

Instanțierea lui *sir2* cu ajutorul declarației (2) nu are nici o legătură cu instanțierea lui *sir1* și se rezervă o zonă de memorie în memoria *heap* distinctă față de cea rezervată pentru același text la instanțierea lui *sir1*. De aceea,

```
sir1.psir
```

și

```
sir2.psir
```

au valori diferite.

Același lucru este valabil și în cazul atribuirilor. Fie de exemplu instanțierile:

```
sir sir1("C++ este un C mai bun");
sir sir2;
```

Atribuirea:

```
sir2 = sir1;
```

realizează o copiere bit cu bit, deci *sir1.psir* și *sir2.psir* devin egale și pointează spre aceeași zonă de memorie din memoria *heap* care a fost rezervată la instanțierea lui *sir1*. Ca și în cazul declarației (2), zona de memorie din memoria *heap* alocată la instanțierea lui *sir2* este pierdută în urma atribuirii, deoarece ea devine inaccesibilă.

Din cele de mai sus, rezultă că pentru obiectele care sunt instanțieri ale clasei *sir* nu este suficientă copierea implicită bit cu bit. În astfel de cazuri este nevoie ca utilizatorul să definiască copieri specifice, care să țină seama de natura

componentelor obiectelor.

În general, operația de inițializare prin copiere diferează de cea de atribuire, deoarece inițializarea este însoțită de alocare. De aceea, sunt necesare două tipuri de copieri, una care să se realizeze la inițializare prin copiere, adică la declarații de forma:

```
(3) nume_clasa obiect2 = obiect1;
```

și una pentru atribuirii de obiecte:

```
(4) obiect2 = obiect1;
```

Inițializarea prin copiere se realizează cu ajutorul *constructorului de copiere* care a fost definit în paragraful 22.8.

Un astfel de constructor are un prototip de forma:

```
nume_clasa(const nume_clasa & obiect);
```

Menționăm că, un astfel de constructor se apelează automat la instanțierea unui obiect printr-o declarație de forma (3).

Copierea bit cu bit se realizează numai dacă clasa nu are un astfel de constructor.

Clasa *sir*, definită în exercițiul 22.12., are un constructor de copiere definit ca mai jos:

```
sir::sir(const sir& s)
{
    lung = s.lung; // se copiază lungimea sirului

    // se aloca zona pentru sir
    psir = new char[lung+1];

    // se copiază sirul în zona rezervată
    strcpy(psir, s.psir);
}
```

Acest constructor se va apela automat la întâlnirea declarației (1) și rezultatul lui constă în aceea că declarația (1) are același efect ca și declarația (2).

Constructorul de copiere nu se apelează la atribuiri.

Pentru a soluționa în mod corect atribuirile de obiecte de felul celor de mai sus, este necesar să suprainscrăcăm în mod corespunzător operatorul de atribuire (=).

În principiu, constructorii de copiere și suprainscrăcarea operatorului de atribuire sunt necesari pentru clasele care au ca date membri pointeri spre zone alocate în memoria *heap* (alocate dinamic).

Un alt caz în care este nevoie de copierea obiectelor este acela cind se apelează funcții care au ca parametri obiecte sau care returnează obiecte.

În astfel de cazuri se alocă obiecte temporare pe stivă și de aceea intervin operații similare cu cele de la inițializare.

Pentru ca aceste transferuri de obiecte să se facă corect pentru obiecte cu

componente pointeri spre date din memoria *heap*, este necesar să avem constructor de copiere în clasa respectivă.

Acesta se va apela automat atât la transferul prin parametri a obiectelor, cit și în cazul în care funcția returnează un obiect.

În absența constructorului de copiere, transferul prin parametri a obiectelor precum și returnarea de obiecte, se realizează aplicându-se copierea implicită, adică copierea bit cu bit.

Un exemplu în care se utilizează copierea bit cu bit, la returnarea de obiecte, este clasa *complex*, implementată în exercițiul 22.1.

În legătură cu suprainscrăcarea operatorului de atribuire este necesar să amintim următoarele:

- suprainscrăcarea operatorului de atribuire se poate realiza numai printr-o funcție membru care nu este statică;
- obiectele operatorului se pot transfera prin valoare sau referință;
- obiectul din stînga operatorului de atribuire, dacă are componente pointeri spre zone alocate în memoria *heap*, trebuie eliberate înainte de a se aloca zone noi în vederea copierii elementelor corespunzătoare ale obiectului din dreapta operatorului respectiv; se obișnuiește să se spună că obiectul din stînga operatorului de atribuire trebuie "curățat" înainte de a se atribui valorile obiectului din dreapta operatorului de atribuire.

De obicei, funcția pentru suprainscrăcarea operatorului de atribuire are un parametru de tip referință și returnează o referință la obiectul curent. Aceasta deoarece transferul prin referință este mai eficient decât transferul obiectelor.

Avind în vedere acest fapt, se recomandă următorul antet pentru suprainscrăcarea operatorului de atribuire:

```
nume_clasa& nume_clasa::operator = (const nume_clasa & operand_drept)
```

Obiectul curent este operandul stîng al atribuirii.

O astfel de funcție se apelează automat pentru realizarea atribuirilor de forma:

```
obiect2 = obiect1;
```

unde:

*obiect1* și *obiect2* - Sunt instanțieri ale clasei *nume\_clasa*.

Atribuirile de forma:

```
obiect1 = obiect1;
```

nu au nici un efect. De aceea, atribuirile de acest fel se vor elimina. În acest scop, în corpul funcției pentru suprainscrăcarea operatorului de atribuire se testează dacă operatorul curent este diferit de cel referit prin parametru și numai în acest caz se execută corpul funcției.

Folosind antetul de mai sus, un astfel de test se poate realiza astfel:

```
if(this != &operand_drept) {
    // operanzi diferiti
```

```

} else // operanzi identici
    return *this;

```

Operatorul de atribuire are și o altă formă care se utilizează pentru a face prescurtări. Este vorba de formatul:

*op=*

unde:

*op* - Este un operator binar aritmetic sau logic pe biți.

Aceste variante pot fi și ele supraincărate în mod corespunzător. Menționăm că, dacă operatorii *op* și *=* sunt supraincărați, nu degurge că este supraincărat și operatorul *op=*. Acesta trebuie supraincărat separat în mod corespunzător.

Operatorul *op=* se supraincarcă folosind funcții membru nestatică care au un prototip similar cu funcțiile pentru supraincărcarea operatorului de atribuire.

#### Exerciții:

23.14 Să se implementeze tipul abstract *sir* pentru instantierea sirurilor de caractere.

Tipul *sir* a fost definit în exercițiul 22.12.

În exercițiul de față, se schimbă funcția membru *atribuiră* a clasei *sir* cu funcția care supraincarcă operatorul de atribuire.

În plus, se supraincarcă operatorii *+* și *+=* pentru concatenarea sirurilor de caractere.

Expresia:

*sir1* = *sir2+sir3*

este legală pentru obiectele clasei *sir* și ea atribuie lui *sir1*, sirul obținut prin concatenarea lui *sir3* la sfîrșitul lui *sir2*.

Expresia:

*sir1* += *sir2*

este legală pentru obiectele clasei *sir* și ea are ca efect concatenarea lui *sir2* la sfîrșitul lui *sir1*.

#### FIŞIERUL BXXIII14.H

```

class sir {
    char *psir;
    int lung;
public:
    sir(char *s); /* constructor pentru initializarea obiectului cu pointerul spre
                    copia în memoria heap a sirului spre care pointează s */
    sir(int nrcar=70); /* constructor care rezerva zona de memorie în memoria
                        heap și pastreaza în ea sirul vid */
    sir( const sir& ); /* constructor de copiere */

```

```

~sir(); // destructor
int retlung(); // returneaza lungimea sirului
void afsir(); // afiseaza sirul
int citsir(); // citeste un sir de caractere
sir& operator = (const sir&); // supraincarca =
sir operator+(sir&); /* returneaza un obiect de tip sir obtinut prin
concatenarea parametrului la sfirsitul
obiectului curent */
sir& operator += (sir&); /* concateneaza parametrul la sfirsitul
obiectului curent */
};

Fișierul de mai jos este de extensie .cpp și el conține definițiile funcțiilor membru.
```

#### FIŞIERUL BXXIII14

```

#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H

#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __SIR_H
#include "BXXIII14.H"
#define __SIR_H
#endif

sir::sir(char *s)
/* constructor utilizat la initializarea obiectului cu pointerul spre copia
   în memoria heap a sirului spre care pointează s */
{
    lung = strlen(s); // lungimea sirului
    psir=new char[lung+1]; // rezerva zona în memoria heap
    strcpy(psir,s); // transfera sirul în memoria heap
}

sir::sir(int dim)
/* - constructor utilizat pentru a rezerva în memoria heap o zona de dim+1 octetii;
   - pastreaza sirul vid în zona respectiva. */
{
    lung = dim; // pastreaza lungimea
    psir = new char[lung+1]; // rezerva zona
    *psir = '\0';
}

sir::sir(const sir& s) // constructor de copiere
{
    lung = s.lung; // pastreaza lungimea sirului
    psir = new char[lung+1]; // rezerva zona
    strcpy(psir,s.psir); // copiază sirul
}

```

```

}

inline sir::~sir() // destructor
{
    delete psir;
}

inline int sir::retlung()
/* returneaza numarul de caractere din compunerea sirului fara caracterul NUL */
{
    return lung;
}

inline void sir::afsir() // afisaza sirul spre care pointeaza psir
{
    printf(psir);
    printf("\n");
}

int sir::citsir()
/* - citeste un sir de la intrarea standard si-l pastreaza in zona heap rezervata pentru obiectul curent;
   - returneaza:
       0 - la sfarsit de fisier;
       -1 - la trunchiere;
       1 - altfel.
*/
{
    char temp[255];

    if(gets(temp) == 0) return 0; // s-a intilnit EOF
    strncpy(psir,temp,lung); /* se copiaza sirul citit */
    *(psir+lung) = '\0';
    if(strlen(temp) > lung) return -1; // trunchiere
    else return 1;
}

sir& sir::operator = (const sir & operand_drept)
// supraincarca operatorul =
{
    if(this != &operand_drept){
        // operanzi diferiti
        delete psir; // curata obiectul curent
        lung = operand_drept.lung; // pastreaza lungimea
        psir = new char[lung+1]; // rezerva zona
        // transfera sirul in zona rezervata
        strncpy(psir,operand_drept.psir,lung);
        *(psir+lung) = '\0';
    }
    return *this;
}

sir sir::operator+(sir& sir2)
/* returneaza un obiect care este rezultatul concatenarii obiectului

```

```

referit de sir2 la obiectul curent */
{
    sir rtemp;

    delete rtemp.psir; // curata obiectul rtemp
    rtemp.lung = lung+sir2.lung; /* determina lungimea sirului rezultat */

    // rezerva memorie pentru sirul rezultat
    rtemp.psir = new char[rtemp.lung+1];
    strcpy(rtemp.psir,psir); // transfera sirul curent

    /* concatenaza sirul referit de sir2 la sfarsitul sirului spre care pointeaza rtemp.psir */
    strcat(rtemp.psir,sir2.psir);
    return rtemp;
}

sir& sir::operator += (sir& operand_drept)
/* concatenarea obiectului operand_drept la obiectul curent */
{
    /* pentru concatenare se utilizeaza operatorul + supraincarcat,
       iar pentru atribuire operatorul = supraincarcat */

    *this = *this+operand_drept;
    return *this;
}

```

23.15 Să se scrie un program care citește întregi din intervalul [1,99] și afișează exprimările lor prin cuvinte.

Programul construiește exprimarea în cuvinte a numerelor din intervalul respectiv prin concatenarea componentelor sale.

#### Exemplu:

```

11 unsprezece
12 doisprezece
...
15 cincisprezece
16 saisprezece
17 saptesprezece
...
20 douazeci
21 douazeci si unu
22 douazeci si doi
...
29 douazeci si noua
30 trezeci
...
37 treizeci si sapte
...

```

40	patruzeci
...	
45	patruzeci si cinci
...	
60	saizeci
...	
66	saizeci si sase
...	
70	saptezeci
...	
76	saptezeci si sase
...	
99	nouazeci si noua

Se observă că numerele peste 10, dar mai mici decât 20 se termină în sufixul *sprezece*. De asemenea, dacă considerăm numerele de la 1 la 9, observăm că exprimarea în cuvinte a acestora se utilizează drept componente pentru exprimarea în cuvinte a numerelor mai mari. Există cîteva excepții, ca de exemplu:

11, 16, 20-29 etc.

Tinind seama de aceste observații, vom proceda ca mai jos.

Se definește tabloul cu denumirile unităților:

```
char *unitate[] = {
    "",
    "unu",
    "doi",
    ...
    "noua"
};
```

Pentru un număr  $n$  din intervalul [1,9] exprimarea este dată de elementul *unitate*[ $n$ ].

Pentru un număr  $n$  din intervalul [12,19] diferit de 16, exprimarea se obține concatenând *unitate*[ $n/10$ ] cu *sprezece*.

Pentru numerele  $n$  mai mari decât 20, exprimările în cuvinte se termină prin:

si *unitate*[ $r$ ],

unde  $r = n \% 10$  și  $n$  nu este multiplu de 10.

Începutul exprimării în cuvinte se definește tinind seama de cifra zecilor:

douazeci	pentru 20-29;
treizeci	pentru 30-39;
patruzeci	pentru 40-49;
cincizeci	pentru 50-59;
saizeci	pentru 60-69;
saptezeci	pentru 70-79;

optzeci	pentru 80-89;
nouazeci	pentru 90-99.

Aceste cuvinte au sufixul *zeci*, iar prefixul este un element al tabloului *unitate*, exceptind intervalele 20-29 și 60-69. Pentru primul interval prefixul este *doua*, iar pentru al doilea interval prefixul este *sai*.

Rezultă următorul procedeu:

- dacă  $n/10=2$ , atunci prefixul *doua* se concatenează cu *zeci*;
- dacă  $n/10=6$ , atunci prefixul *sai* se concatenează cu *zeci*;
- altfel *unitate*[ $n/10$ ] se concatenează cu *zeci*.

### PROGRAMUL BXIII15

```
#include <stdlib.h>
#include "BXIII14.CPP" // clasa sir
#include "BVIII2.CPP" // pcit_int
#include "BVIII3.CPP" // pcit_int_lim

main()
/* - citeste intregi zecimali din intervalul [1,99];
   - afiseaza exprimarile lor in cuvinte
*/
{
    int n;
    char *unit [] = {"", "unu", "doi", "trei", "patru", "cinci",
                     "sase", "sapte", "opt", "noua"};
    sir zece("zece");
    sir spre("spre");
    sir un("un");
    sir sai("sai");
    sir doua("doua");
    sir zeci("zeci ");
    sir si("si ");
    sir rez;
    int q,r;

    for(;;) {
        if(pcit_int_lim("n = ",1,99,&n) == 0) exit(0);
        q = n/10;
        r = n%10;
        switch(q){
            case 0: {      // [1,9]
                sir rez0(unit [n]);
                rez = rez0;
                break;
            }
            case 1: {      // [10,19]
                if(n == 10) rez = zece;
                else
                    if(n == 11) rez = un+spre+zeca;
                    else
                        if(n == 16) rez = sai+spre+zeca;
                        else
                            if(n == 17) rez = doua+spre+zeca;
                            else
                                if(n == 18) rez = doua+si+zeca;
                                else
                                    if(n == 19) rez = doua+sai+zeca;
                                    else
                                        if(n == 20) rez = doua+zece;
                                        else
                                            if(n == 21) rez = doua+uno;
                                            else
                                                if(n == 22) rez = doua+doi;
                                                else
                                                    if(n == 23) rez = doua+trei;
                                                    else
                                                        if(n == 24) rez = doua+patru;
                                                        else
                                                            if(n == 25) rez = doua+cinci;
                                                            else
                                                                if(n == 26) rez = doua+sase;
                                                                else
                                                                    if(n == 27) rez = doua+sapte;
                                                                    else
                                                                        if(n == 28) rez = doua+opt;
                                                                        else
                                                                            if(n == 29) rez = doua+noua;
            }
        }
    }
}
```

```

        sir rez0(unit[r]);
        rez = rez0+spre+zece;
    }
    break;
}
case 2: { // [20-29]
    rez = doua+zeci;
    if(n == 20) break;
    rez += si;
    sir rez0(unit[r]);
    rez += rez0;
    break;
}
case 6: { // [60-69]
    rez = sai+zeci;
    if(n == 60) break;
    rez += si;
    sir rez0(unit[r]);
    rez += rez0;
    break;
}
default: { // [30-39], [40-49], [50-59], [70-79], [80-89], [90-99]
    sir rez0(unit[q]);
    rez = rez0+zeci;
    if(r == 0) break;
    rez += si;
    sir rez1(unit[r]);
    rez += rez1;
}
}
} // sfârșit switch

// afisare n și exprimarea prin cuvinte
printf("\n n = %d\t",n);
rez.afsir();
}

```

#### Observație:

Corpul funcției poate fi scris mai compact folosind instrucțiuni *if* în locul instrucțiunii *switch*:

```

if(q == 0){ // [1,9]
    sir rez0(unit[n]);
    rez = rez0;
}
else
    if (q == 1) { // [10,19]
        if(n == 10) rez = zece;
        else {
            if(n==11) rez = un;
            else
                if(n==16) rez = sai;
                else {

```

```

        sir rez0 (unit[r]);
        rez = rez0;
    }
    rez += spre + zece;
}
} // sfârșit [10,19]
else{
    if(q==2) // [20,29]
        rez = doua;
    else
        if(q==6) // [60,69]
            rez = sai;
        else{
            sir rez0(unit[q]);
            rez=rez0;
        }
    rez += zeci;
    if(r){
        sir rez0(unit[r])
        rez += si+rez0;
    }
}

```

### 23.3. Supraîncărcarea operatorului [] (operatorul de indexare)

Operatorul [] predefinit se utilizează pentru a face acces la elementele unui tablou.

De exemplu, dacă *tab* este un tablou unidimensional de un tip predefinit, atunci o expresie de forma:

(1) *tab[exp]*

permite acces la elementul tabloului *tab* de indice *exp*, *exp* fiind o expresie care furnizează o valoare de tip întreg.

Construcția (1) a fost numită *variabilă cu indici*. Ea este o expresie formată din doi operanzi:

*tab* și *exp*

la care se aplică operatorul de indexare ([]). De obicei, expresia (1) are și o interpretare unitară și anume se consideră ca formind un *operand*. Această interpretare este posibilă datorită faptului că parantezele pătrate sunt operatori de prioritate maximă.

Interpretarea expresiei (1) ca operand se justifică datorită faptului că ea are utilizări similare cu numele unei variabile simple: ambele permit acces la o dată de un tip predefinit.

Construcția de mai sus este valabilă și în cazuri mai generale, cind primul operand este chiar o expresie de pointeri, care are ca valoare un pointer.

### Exemplu:

```
int *p;  
int i, j;
```

Expresiile de mai jos sint legale:

```
p[1]  
p[i]  
p[i+2]  
p[i-1]  
(p+2)[i*3+j]  
(p-10)[i*3-2*j]
```

Utilizatorul poate supraincărca operatorul [] pentru a da sens construcțiilor de felul celor de mai sus și în cazul în care operanții sunt obiecte.

Expresiile de formă (1) sunt expresii *lvalue* (se pot utiliza ca parte stîngă într-o atribuire). De aceea, la supraincărarea operatorului [] pentru tipuri abstracțe, ca și la supraincărarea operatorului =, se va utiliza o funcție membru nstatică, funcție care să returneze o referință la elementul selectat prin funcția respectivă.

În aceste condiții, funcția pentru supraincărarea operatorului [] are antetul general:

*tip& nume\_clasa::operator[](tip\_indexe i)*

Odată supraincarcat operatorul [] ca mai sus, sint legale expresiile de forma:

(2) *obiect[expresie]*

unde:

*obiect* - Este o instațiere a clasei *nume-clasa*.

*expresie* - Este o expresie care are tipul *tip\_indexe* sau un tip convertibil spre acesta.

Expresia (2) are ca rezultat o referință la elementul definit prin funcția care supraincarcă operatorul []. Ea este o *lvalue* și deci se poate utiliza în ambele părți ale unei expresii de atribuire. De asemenea, expresia (2) se poate interpreta ca fiind un operand, neglijind faptul că ea se compune din doi operanzi la care se aplică operatorul de indexare.

Operandul de această formă reprezintă un apel al funcției membru operator []. Astfel, operandul (2) de mai sus, se evaluează prin apelul:

(3) *obiect.operator[](expresie)*

unde:

*operator []* - Reprezintă numele funcției membru care supraincarcă operatorul [] și ea este apelată pentru obiectul *obiect* și indicele *expresie*.

Interesul pentru formatul (2) față de apelul exprimat prin (3) rezultă din faptul că reprezintă o exprimare clară și care este incetătenită pentru tipurile predefinite de date.

### Exerciții:

23.16 Să se definiște tipul abstract *nzl* care gestionează numărul zilelor din lunile calendaristice.

Obiectele clasei *nzl* se pot utiliza în expresii de forma:

*obiect[n]*

sau

*obiect[den\_luna]*

unde:

*n* - Este numărul lunii calendaristice.

*den\_luna* - Este un pointer spre denumirea unei luni calendaristice.

Acste expresii au ca valoare numărul zilelor corespunzătoare unei luni calendaristice. Acest număr de zile, este în mod implicit, numărul zilelor dintr-o lună calendaristică dar el poate fi mai mic, exprimind numărul zilelor lucrătoare sau lucrate de o persoană în luna respectivă etc.

### FISIERUL BXXIII16.H

```
class nzl {  
    int tnz[13]; /* numarul curent de zile din lunile calendaristice */  
    static int tnrz[13]; /* tabloul cu numarul de zile din lunile calendaristice */  
    static char *tdenluna[13];  
    /* tablou de pointere spre denumirile lunilor calendaristice */  
public:  
    int& operator[] (char *dl);  
    /* - returneaza o referinta la elementul tnz[nl];  
     * nl este numarul lunii spre a carui denumire pointeaza dl */  
    int& operator[](int nl); /* returneaza o referinta la elementul tnz[nl] */  
    nzl(int an); /* elementele tnz se initializeaza cu cele ale lui tnrz */  
    static char *retdenl(int n); /* returneaza denumirea lunii a n-a */  
};
```

Mai jos, se inițializează datele membru statice și se definesc funcțiile membru.

### FISIERUL BXXIII16

```
#ifndef __NZL_H  
#include "BXXIII16.H"  
#define __NZL_H  
#endif  
  
int nzl::tnrz[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};  
  
char *nzl::tdenluna[] = {  
    "",  
    "ianuarie",  
    "februarie",  
    "martie",  
    "aprilie",  
    "mai",  
    "iunie",  
    "iulie",  
    "august",  
    "septembrie",  
    "octombrie",  
    "noiembrie",  
    "decembrie"};
```

```

"mai",
"iunie",
"iulie",
"august",
"septembrie",
"octombrie",
"noiembrie",
"decembrie"
};

int& nzl::operator[](char *dl)
/* - returneaza o referinta la elementul tnz[nl];
   - nl este numarul lunii spre a carei denumire pointeaza dl;
   - returneaza referinta spre zero daca nu exista o luna cu denumirea spre care pointeaza dl */
{
    for(int nl=1;nl < 13;nl++)
        if(strcmp(nzl::tdenluna[nl],dl) == 0) return tnz[nl];
    return tnrz[0]; // returneaza referinta la zero
}

int & nzl::operator[](int nl)
/* returneaza o referinta la tnz[nl] sau la zero daca nl nu se afla in intervalul [1,12] */
{
    return nl < 1 || nl > 12 ? tnrz[0]:tnz[nl];
}

nzl::nzl(int an)
/* initializeaza elementele lui tnz cu zilele din lunile calendaristice */
{
    int b = an%4==0&&an%100||an%400==0;

    for(int i=0;i<13;i++) tnz[i]=nzl::tnrz[i];
    tnz[2] += b;
}

char *nzl::retdenl(int n)
/* returneaza denumirica lunii a n-a sau pointerul nul daca n este in afara intervalului [1,12] */
{
    return n < 1 || n > 12 ? 0: nzl::tdenluna[n];
}

```

#### Observație:

Constructorul nu testează apartenența anului la intervalul [1600,4900].  
Acum lucru se presupune că este îndeplinit.

- 23.17 Să se scrie un program care citește un întreg de 4 cifre ce reprezintă un an calendaristic și denumiri ale lunilor calendaristice. Pentru fiecare denumire citită, programul afișează numărul zilelor din luna respectivă.

#### PROGRAMUL BXXIII17

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

```

#include <stdlib.h>
#include "BXXIII16.CPP"
#include "BVIII2.CPP"
#include "BVIII3.CPP"

main()
/* - citește un întreg de 4 cifre ce reprezintă un an calendaristic;
   - citește denumiri ale lunilor calendaristice și afisează numarul zilelor din luna respectiva;
   - programul își întrerupe execuția la întâlnirea sfîrșitului de fisier.*/
{
    int an;
    char d[255];

    if(pcit_int_lim("anul_calendaristic=",1600,4900,&an)==0){
        printf("s-a citit EOF\n");
        exit(1);
    }
    nzl nzluna(an);

    for(;;){
        printf("denumirea lunii pe un rind=");
        if(scanf("%s",d)!=1) break;
        int i = nzluna[d]; // determina numarul de zile din luna
        if(i) printf("luna:%s are %d zile\n",d,i);
        else printf("denumire eronata\n");
    }
}

```

- 23.18 Să se scrie un program care citește următoarele date de la intrarea standard:

- un întreg de cel mult 4 cifre care reprezintă un an calendaristic;
- 12 numere întregi care reprezintă numărul zilelor lucrătoare din lunile anului calendaristic citit anterior.

Programul afișează, pentru fiecare lună calendaristică, următoarele date:

- denumirea lunii calendaristice;
- numărul zilelor lucrătoare;
- numărul orelor efectuate de un angajat dacă se consideră că ziua lucrătoare este de 8 ore.

În final, se afișează totalul orelor efectuate de un angajat în decursul anului definit de datele citite de la intrarea standard.

În acest program funcțiile *pcit\_int* și *pcit\_int\_lim* sunt înlocuite de similarele lor *rcit\_int* și *rcit\_int\_lim*.

#### PROGRAMUL BXXIII18

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include "BXX4.CPP" // rcit_int
#include "BXX5.CPP" // rcit_int_lim
#include "BXXIII16.CPP"

```

```

main()
/* - citeste un intreg de 4 cifre care reprezinta un an calendaristic;
   - citeste 12 numere care reprezinta numarul zilelor lucratoare din lunile anului calendaristic
     citit anterior;
   - afiseaza anul calendaristic;
   - afiseaza pentru fiecare luna:
     - denumirea lunii calendaristice;
     - numarul de zile lucratoare;
     - numarul orclor daca se considera ca o zi lucratoare are 8 ore;
     - totalul orclor lucratoare din anul calendaristic.

*/
{
    int zile_lucrat[13];
    int total_zile = 0;
    int zi;
    int an;

    // citeste intregul care reprezinta anul calendaristic
    if(rcit_int_lim("anul calendaristic=",1600,4900,an)==0){
        printf("s-a tastat EOF\n");
        exit(1);
    }

    nzl_nrzileimplicit(an);

    /* citeste zilele lucratoare existente in cele 12 luni ale anului calendaristic */
    printf("tastati zilele lucratoare\n");
    for(int i=1;i<=12;i++){
        do{
            if(rcit_int_lim(nzl::retdenl(i),0,31,zi)==0){
                printf("s-a tastat EOF\n");
                exit(1);
            }
            if(nrzileimplicit[i] >= zi) break;
            printf("numarul zilelor lucratoare=%d\n",zi);
            printf("depaseste numarul zilelor din luna=%d\n",
                  nrzileimplicit[i]);
        }while(1);
        zile_lucrat[i]=zi;
    } // sfarsit for pentru citirea zilelor lucratoare

    // afiseaza anul calendaristic
    printf("\n anul calendaristic:%d\n",an);
    for(i=1;i<13;i++){

        // afiseaza denumirea lunii calendaristice
        printf("\n%s\n",nzl::retdenl(i));

        // afiseaza numarul de zile lucratoare
        printf("zile lucratoare=%d\n",zile_lucrat[i]);

        // afiseaza orele lucrante
        printf("ore lucrante=%d\n",zile_lucrat[i]*8);
    }
}

```

```

// insumeaza numarul zilelor lucratoare
total_zile += zile_lucrat[i];

printf("\n actionati o tasta pentru a continua\n");
getch();
} // sfarsit for afisare

// afiseaza totalul orclor lucrante
printf("\n\n Total ore lucrante=%d\n",total_zile*8);
}

```

23.19 Să se definească tipul abstract *cuvdif* pentru determinarea frecvențelor de apariție ale cuvintelor citite dintr-un text.

Printr-un cuvînt se înțelege o succesiune de litere. Literele mici nu se consideră distințe de cele mari.

Se consideră tipul utilizator *fcuv* definit astfel:

```

struct fcuv {
    char *cuv;
    int f;
};

```

Pointerul *cuv* pointează spre un șir de caractere care reprezintă un cuvînt citit din textul de intrare.

Componenta *f*, numără aparițiile cuvintului spre care pointează componenta *cuv*.

Tipul *cuvdif* se poate defini ca un tablou de elemente de tip *fcuv*. În acest caz, urmează să se indice numărul maxim de elemente al tabloului. Acesta trebuie să depășească numărul cuvintelor diferite din textul de intrare. În cazul în care acest maxim nu este bine estimat și el este depășit la citirea textului de intrare, programul trebuie să se întrerupă. Pentru a evita astfel de situații se poate proceda ca mai jos.

În loc de a defini un tablou de tip *fcuv* care să aibă un număr maxim de elemente, considerăm un pointer spre o zonă în care se păstrează date de tip *fcuv*. Această zonă se alocă în memoria *heap*, prin constructor, aşa încît ea să fie capabilă să păstreze un număr oarecare, *max*, de elemente.

Elementele de tip *fcuv* se păstrează în zona respectivă pe măsură ce se citesc cuvinte distincte din text. Dacă se ajunge ca zona de memorie rezervată în memoria *heap* prin constructor să fie insuficientă, acest maxim se mărește cu o anumită cantitate, să zicem *delta*:

```

max = max+delta

```

Apoi, se alocă în memoria *heap* o zonă de memorie de dimensiune egală cu valoarea nouă a lui *max*, se copiază elementele din zona veche în cea nouă, iar zona veche se eliberează.

Componentele date ale tipului *cuvdif* sunt:

*pfcuv* - Este pointer spre tipul *fcuv*.

- max*
- Este un intreg care definește dimensiunea zonei alocate în memoria *heap*.
- ndif*
- Este un intreg care definește numărul datelor de tip *fcuv* păstrate în zona *heap*.
- delta*
- Este un intreg care definește valoarea pentru incrementarea lui *max* cind *ndif* > *max*.

Constructorul tipului abstract *cuvdif* are ca parametri valorile lui *max* și *delta*. Operatorul [] este supraincărcat pentru a legaliza o expresie de forma:

*cuvdif*[*cuvint*)

unde:

- cuvint*
- Este un pointer spre un sir de caractere care definește un cuvint.

Această construcție definește o referință la componenta *f* a elementului de tip *fcuv* pentru care *cuv* pointează spre cuvintul identic cu cel spre care pointează *cuvint*.

Funcția de supraincărcare a operatorului [] realizează următoarele:

1. Caută în memoria heap un element de tip *fcuv* pentru care *cuv* pointează spre un sir de caractere identic cu cel spre care pointează *cuvint*. Dacă nu există un astfel de element, se continuă cu punctul 2. Altfel se revine din funcție returnându-se o referință la componenta *f* a elementului respectiv.
2. Se păstrează în memoria heap un element corespunzător cuvintului spre care pointează *cuvint*. *f*=0 pentru acest element, iar variabila *ndif* se incrementează. Se revine din funcție returnându-se o referință la componenta *f* a acestui element nou păstrat în memoria *heap*.

Din cele spuse mai sus rezultă că păstrarea unui element nou în memoria *heap* este posibilă numai dacă *ndif* < *max*. Dacă această condiție nu este indeplinită, atunci se extinde zona cu cantitatea *delta*, aşa cum s-a indicat mai sus.

## FIŞIERUL BXXIII19.H

```
struct fcuv {
    char *cuv;
    int f;
};

class cuvdif {
    fcuv *pfcuv;
    int max;
    int ndif;
    int delta;
public:
    cuvdif(int m=100, int d=50);
```

```
int& operator[] (const char *cuvint);
/* returnaza o referinta la componenta f a elementului de tip fcuv
   pentru care cuv pointeaza spre acelasi sir de caractere ca si cuvint */

void afis_frecv_cuv();
/* afiseaza frecventa cuvintelor citite */

};
```

Funcțiile membru sunt definite în fișierul de mai jos, de extensie CPP.

## FIŞIERUL BXXIII19

```
#ifndef __CUVDIF_H
#include "BXXIII19.H"
#define __CUVDIF_H
#endif

cuvdif::cuvdif(int m,int d)
/* - se rezerva in memoria heap zona pentru m elemente de tip fcuv;
   - d este pasul pentru incrementarea lui max */
{
    max=m; // maxim initial pentru cuvintele diferite
    pfcuv=new fcuv[max];
    ndif=0; // nu sunt cuvinte citite
    delta=d; // increment pentru maxim
}

int& cuvdif::operator[] (const char *cuvint)
/* returnaza o referinta la componenta f a elementului din memoria heap pentru care cuv
   pointeaza spre un sir de caractere identic cu cel spre care pointeaza cuvint */
{
    /* se cauta in memoria heap un element pentru care cuv pointeaza
       spre un sir de caractere identic cu cel spre care pointeaza cuvint */

    fcuv *p;
    int i;

    for(p=pfcuv,i=0;i < ndif;i++,p++)
        if(strcmp(p->cuv,cuvint)==0)
            return p->f; // s-a gasit elementul cautat

    /* nu exista in memoria heap un element pentru care cuv sa pointeze
       spre un sir de caractere identic cu cel spre care pointeaza cuvint */

    if(max==ndif){
        // se marestea zona din memoria heap
        max += delta;

        /* se rezerva zona in memoria heap pentru max elemente de tip fcuv */
        p = new fcuv[max];

        // se copiază datele din zona veche în cea nouă
        for(i=0;i < ndif;i++) p[i] = pfcuv[i];
    }
```

```

// se elibereaza zona veche din memoria heap
delete pfcuv;

// pfcuv pointeaza in continuare spre zona noua
pfcuv = p;

} // sfarsit if pentru incrementarea zonei din memoria heap cu valoarea delta

// rezerva zona heap pentru cuvintul spre care pointeaza cuvint
char *c = new char[strlen(cuvint)+1];

// transfera sirul curent in zona heap
strcpy(c,cuvint);

// construieste elementul din memoria heap
p = &pfcuv[ndif++]; // adresa elementului curent din memoria heap

// pastreaza adresa cuvintului curent
p -> cuv = c;

// pentru cuvintul curent f=0
p -> f = 0;

// returneaza o referinta la f
return p -> f;
}

void cuvdif::afis_frecv_cuv()
/* afiseaza frecventa de aparitie a cuvintelor diferite citite din textul de la intrarea standard */
{
    for(int i=0;i < ndif;i++){
        printf("cuvintul:%-40s are frecventa=%d\n",
               pfcuv[i].cuv, pfcuv[i].f);
        if((i+1)%23==0){
            printf("actionati o tasta pentru a continua\n");
            getch();
        }
    }
}

```

### Observații:

- În acest exemplu nu s-a testat posibilitatea depășirii memoriei *heap*. Într-un program real este necesar ca o alocare de forma:  

```

p = new ...

```

să se realizeze folosind un test de forma:  

```

if((p = new ...) == 0){
    /* mesaj de eroare pentru depasirea memoriei heap urmat,
       eventual, de apelul functiei exit */
}

```
- În principiu, tipul abstract *cuvdif* este necesar să aibă destrutor pentru a elibera zonele din memoria *heap*.

În exemplul de față, se are în vedere o utilizare simplă a tipului respectiv și nu a mai fost nevoie de destrutor.

- Atribuirea:  
 $p[i] = pfcuv[i]$   
 se realizează prin copierea implicită a datelor de tip *fcuv* (copiere bit cu bit).
- Tipul abstract *cuvdif* este un exemplu de implementare a tablourilor dinamice.  
 Un obiect care este o instanțiere a clasei *cuvdif* reprezintă un tablou dinamic alocat în memoria *heap* și care are următoarele attribute:  
  - pfcuv* - Este pointer spre începutul zonei alocate tabloului în care se păstrează elemente de tip *fcuv*.
  - max* - Este un întreg ce reprezintă dimensiunea zonei alocate în memoria *heap*.
  - ndif* - Aceasta conține *max* elemente de tip *fcuv*.
  - delta* - Este un întreg ce reprezintă un increment care se aplică la *max*, ori de câte ori se depășește zona din memoria *heap*.

- 23.20 Să se scrie un program care citește de la intrarea standard un text și afișează frecvența de apariție a cuvintelor diferite din textul respectiv.

Această problemă a fost rezolvată în mai multe moduri folosind:

- un tablou de pointeri;
- o listă simplu înlanțuită;
- o listă dublu înlanțuită;
- un arbore binar;
- o tabelă de dispersie.

În programul de față se folosește tipul abstract *cuvdif*. Acesta are o performanță comparabilă cu programul implementat folosind un tablou de pointeri. Ca avantaj al programului de față este faptul că deși este necesar să indicăm o dimensiune maximă pentru numărul cuvintelor distincte din textul care se citește, această dimensiune se ajustează automat în caz de depășire.

### PROGRAMUL BXXIII20

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include "BXXIII19.CPP"

main()
/* citește un text și afișează frecvența de apariție a cuvintelor diferite din textul respectiv */
{
    char t[255];
    int i;

```

```

int c;
cuvdif text;

// citirea cuvintelor din text
// textul se termina la intalnirea sfarsitului de fisier

for(;){
    // salt peste caractere care nu sunt litere
    while((c = getchar()) != EOF && (c < 'A' || c > 'Z' && c < 'a' || c > 'z'))
        ;
    if(c==EOF) break;

    // citeste literele unui cuvant
    for(i=0;c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z';i++){
        t[i]=c; // pstreaza o litera
        c = getchar();
    }

    // pstreaza caracterul NUL la sfarsitul cuvantului citit
    t[i]='\0';

    // incrementeaza frevena cuvantului citit
    text[t]++;
    if(c==EOF) break;
} // sfarsit for pentru citirea cuvintelor din textul de la intrarea standard

// afisaza frevena cuvintelor
printf("\n Frecventa cuvintelor citite\n");
text.afis_frecv_cuv();
}

```

#### Observație:

Expresia `text[t]` realizează apelul la funcția pentru supraîncarcarea operatorului `[]`. Această funcție căută cuvantul spre care pointează `t` în zona `heap`, corespunzătoare obiectului `text`. Dacă nu există un astfel de cuvint, atunci acesta se pstrează în zona respectivă. Dacă nu există loc în zona alocată pentru obiectul `text`, atunci se mărește zona respectivă. La revenirea din funcție, se returnează o referință la componenta/corespunzătoare cuvantului citit.

Aplicarea operatorului de incrementare expresiei `text[t]`, înseamnă incrementarea componentei `f` pentru care această expresie este o referință.

- 23.21 Să se completeze tipul abstract `cuvdif`, definit în exercițiul 23.19., cu funcția membru `ordcresc` care ordonează datele de tip `fcuv` din zona `heap`, să incit pointerii `cuv` respectivi să pointeze spre cuvintele citite în ordinea crescătoare a acestora (ordinea alfabetică).

Funcția `ordcresc` utilizează metoda de sortare a bulelor. Pentru a obține o sortare eficientă se vor utiliza alte metode, ca de exemplu sortarea Shell sau

rapidă (vezi capitolul 14).

#### FIŞIERUL BXXIII21.H

```

struct fcuv {
    char *cuv;
    int f;
};

class cuvdif {
    fcuv *pfcuv;
    int max;
    int ndif;
    int delta;
public:
    cuvdif(int m=100, int d=50);
    int& operator[](const char *cuvint);
    void ordcresc();
    /* ordoneaza elementele din zona heap in asa fel incit cuvintele spre care pointeaza cuv sa formeze un sir de cuvinte ordonat crescator (in ordine alfabetica) */

    void afis_frecv_cuv();
};

Funcțiile membru, exceptind ordcresc, sunt definite în fișierul BXXIII19.CPP.
În fișierul de mai jos se adaugă definiția funcției ordcresc la definițiile existente în fișierul BXXIII19.CPP.

```

Pentru detalii în legătură cu metoda de sortare a bulelor se poate consulta capitolul 14.

#### FIŞIERUL BXXIII21

```

#ifndef __CUVDIF_H
#include "BXXIII21.H"
#define __CUVDIF_H
#endif
#include "BXXIII19.CPP"

void cuvdif::ordcresc()
/* ordoneaza elementele din memoria heap in asa fel incit cuvintele spre care pointeaza cuv sa formeze un sir de cuvinte ordonat crescator */
{
    int i,ind;

    ind = 1;
    while(ind){
        ind = 0;
        for(i=0;i < ndif-1;i++)
            if(strcmp(pfcuv[i].cuv,pfcuv[i+1].cuv) > 0){
                fcuv temp;
                temp = pfcuv[i];
                pfcuv[i]=pfcuv[i+1];
                pfcuv[i+1]=temp;
                ind = 1;
            }
    }
}

```

```

    }
}

```

- 23.22 Să se scrie un program care citește cuvintele dintr-un text și le afișează în ordine alfabetică împreună cu frecvența lor de apariție.

Programul de față este similar cu cel din exercițiul 23.20.

### PROGRAMUL BXXIII22

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include "BXXIII21.CPP"

main()
/* citeste cuvintele din textul de la intrarea standard si le afiseaza in
ordine alfabetica impreuna cu frecventa lor de aparitie */
{
    char t[255];
    int i;
    int c;
    cuvdif text;

    for(;;) {
        while((c=getchar())!=EOF&&(c < 'A'||c > 'Z' &&
            c < 'a' || c > 'z'))
            ;
        if(c == EOF) break;
        for(i=0;c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z';i++) {
            t[i]=c;
            c=getchar();
        }
        t[i]='\0';
        text[t]++;
        if(c == EOF) break;
    }

    // sortare in ordine alfabetica
    text.ordcresc();
    text.afis_frecv_cuv();
}

```

## 23.4. Supraîncărcarea operatorului () (Operatorul de apel funcție)

O funcție se apelează printr-o construcție de forma:

(1) *nume\_functie(listă\_parametrilor\_efecțiivi)*

Această construcție poate fi privită ca un tot unitar care să formeze un

operand. În realitate, ea este o expresie formată din doi operanzi:

*nume\_functie*

și

*listă\_parametrilor\_efecțiivi*

la care se aplică operatorul binar () .

Ceea ce este specific pentru această interpretare este faptul că cel de al doilea parametru (*listă\_parametrilor\_efecțiivi*) poate fi și vid. Aceasta înseamnă că operatorul () poate avea ca operand secund un operand vid, lucru care nu este posibil pentru ceilalți operatori binari.

Apelul unei funcții se poate realiza folosind formatul de mai jos, în care numele funcției se înlocuiește cu o expresie de tip pointer spre funcția respectivă:

(2) (\*ep)(*listă\_parametrilor\_efecțiivi*)

Acest apel se utilizează în cazul în care nu se cunoaște numele funcției, ci numai un pointer spre funcția respectivă (vezi capitolul 8).

Operatorul () se poate supraîncărca așa încât primul operand să fie un obiect:

(3) *obiect(listă\_parametrilor\_efecțiivi)*

O astfel de expresie este echivalentă cu un apel de forma:

*obiect.operator()* (*listă\_parametrilor\_efecțiivi*)

Funcția care supraîncarcă operatorul () trebuie să fie o funcție membru statică.

*Listă\_parametrilor\_efecțiivi* se tratează ca în apelurile obișnuite de funcții.

Supraîncărcarea operatorului () se utilizează frecvent la definirea așa numitului *iterator*.

Iteratorii se utilizează în legătură cu tipuri abstracte care conțin colecții de elemente. Astfel de tipuri de date sunt listele, arborii, tabelele de dispersie etc.

Problema consultării (căutării) elementelor dintr-o colecție de elemente protejate se rezolvă cel mai simplu cu ajutorul iteratorilor. În afară de faptul că iteratorii asigură protecția datelor, ei oferă un mijloc simplu de acces la elementele unei colecții fără a intra în detaliile legate de implementarea colecției, mod de ordonare a elementelor colecției etc. Aceasta asigură o independență mare a utilizării colecției de implementarea ei.

În principiu, un iterator se implementează printr-o clasă specială atașată unui tip abstract care conține o colecție de elemente. Fie, de exemplu, o clasă pe care o numim *container* și care este o colecție de obiecte de un tip oarecare.

Colecția de obiecte poate fi implementată în mai multe moduri:

- tablou de obiecte de tip *obiect*, de dimensiune fixă sau variabilă;
- listă de noduri de tip *obiect* simplu sau dublu înlățuită;
- arbore binar cu noduri de tip *obiect*, tabelă de dispersie cu înregistrări de tip *obiect* etc.

În funcție de organizarea aleasă pentru implementarea tipului *container*, clasa

*container* are o dată membru protejată de tip *obiect* sau pointer spre *obiect*. De exemplu, dacă tipul *container* se implementează cu un tablou de dimensiune fixă, atunci clasa *container* se definește astfel:

```
(a) class container {
    obiect tab[n];
    ...
};
```

unde:

- Este o constantă întreagă.

În acest caz, un iterator va asigura acces la elementele  $\text{tab}[0], \text{tab}[1], \dots, \text{tab}[n-1]$ . Accesul se realizează secvențial: la primul apel se permite acces la  $\text{tab}[0]$ , apoi la  $\text{tab}[1]$  și așa mai departe.

În cazul în care tipul *container* se implementează ca o colecție dinamică de elemente de tip *obiect*, se poate proceda ca în cazul tipului abstract *cuvdif*, implementat în exercițiul 23.19. Clasa *container* se definește astfel:

```
(b) class container {
    obiect *colectie;
    int max;
    int crt;
    int delta;
    ...
};
```

În acest caz, elementele de tip *obiect* se păstrează într-o zonă din memoria *heap*. Dimensiunea curentă a acestei zone este stabilită în așa fel încât ea să poată conține pînă la *max* obiecte de tip *obiect*.

Pointerul *colectie* pointează spre începutul acestei zone. Variabila *crt* are ca valoare numărul obiectelor de tip *obiect* prezente în memoria *heap*.

În cazul în care zona rezervată în memoria *heap* devine insuficientă, se rezervă o altă zonă de dimensiune *max+delta* în care se transferă obiectele existente, așa cum s-a procedat în exercițiul 23.19.

Ca și în cazul (a), iteratorul va asigura acces succesiv la obiectele: *colectie[0], colectie[1], ..., colectie[crt-1]*.

Dacă tipul *container* se implementează ca o listă simplu înlățuită, atunci nodurile se definesc ca în capitolul 11:

```
struct obiect {
    declaratii
    struct obiect *urm;
};
```

Clasa *container* are ca și date membru pointerii *prim* și *ultim*, care pointează spre capetele listei. Acești pointeri au fost definiți în capitolul 11 ca variabile globale pentru implementarea listelor simplu și dublu înlățuite. Clasa *container* se poate defini astfel:

```
(c) class container {
    obiect *prim, *ultim;
    ...
};
```

În acest caz, un iterator trebuie să asigure acces succesiv la nodurile listei, începînd cu cel spre care pointează *prim* și terminînd cu cel spre care pointează *ultim*.

În cazul listelor dublu înlățuite se pot defini doi iteratori, pentru a permite acces secvențial la nodurile listei în ambele sensuri.

Dacă tipul *container* se implementează ca un arbore, atunci nodurile se pot defini ca în capitolul 12:

```
struct obiect {
    declaratii
    struct obiect *st;
    struct obiect *dr;
};
```

Clasa *container* are, în acest caz, ca dată membru un pointer spre rădâcina arborelui, numit *prad*:

```
(d) class container {
    obiect *prad;
    ...
};
```

În acest caz, se pot defini 3 tipuri de iteratori care să permită accesul la obiectele colecției în *preordine*, *inordine* și *postordine*.

În cazul în care tipul *container* se implementează printr-o tabelă de dispersie, se poate utiliza clasa:

```
(e) class container {
    obiect *thash[M];
    ...
};
```

unde:

*M* - Este o constantă întreagă și care, de obicei, este un număr prim.

Tipul *obiect* se definește la fel ca nodurile listelor simplu înlățuite deoarece *thash* este un tablou de pointeri spre nodurile de început ale listelor simplu înlățuite care sunt alcătuite din noduri ale căror chei intră în coliziune (vezi cap.13).

În acest caz, clasa *container* are o funcție membru pentru calculul *dispersiei*. Iteratorul pentru clasa *container* poate fi implementat în mai multe moduri. Astfel, se poate defini un iterator pentru a permite acces la nodurile unei liste spre începutul căreia pointează un element al tabloului *thash*. Un astfel de iterator poate avea ca parametru un întreg din intervalul  $[0, M-1]$  care este un indice în

tabloul *thash* și definește elementul pointer spre începutul listei ale cărei noduri sunt accesate prin intermediul iteratorului.

O altă posibilitate este ca iteratorul să aibă ca parametru un obiect, iar acesta realizează acces succesiv la obiectele care se află în coliziune cu parametrul. În acest caz, iteratorul apelează funcția membru a clasei *container* pentru calculul dispersiei și determină, în felul acesta, indicele din tabela *thash* care definește pointerul spre lista cu obiectele aflate în coliziune și care sunt accesate prin iterator.

În sfîrșit, să amintim că se poate defini un iterator care să realizeze accesul succesiv la obiectele din listele spre care pointează elementele tabloului *thash*, începând cu pointerul *thash[0]*, continuind cu *thash[1]*, *thash[2]* și așa mai departe pînă la *thash[M-1]*.

Din cele de mai sus, rezultă că un iterator are următoarele proprietăți:

- Este atașat unui tip abstract care corespunde unei colecții de obiecte (mai sus, un astfel de tip a fost numit *container*). O instanțiere a clasei *container* este un obiect care este o colecție de obiecte de tip *obiect*).
- Este capabil să livreze, pe rînd, obiectele elemente ale colecției.

Avind în vedere caracteristicile iteratorilor, putem face o analogie dintre acesteia și operația de citire a fișierelor.

Operația de citire a fișierelor se realizează apelind o funcție specială care, la fiecare apel, livrează înregistrarea următoare din fișier. Iteratorul, la fel ca și funcția de citire, are drept scop să asigure accesul la obiectul următor din colecție.

Citirile înregistrărilor dintr-un fișier trebuie să fie precedate de operația de deschidere a fișierului respectiv. În mod analog, în cazul iteratorilor este necesară o operație care să definească obiectul la care se asigură accesul la primul apel al iteratorului. Această operație se numește *initializarea* iteratorului.

Un mod simplu de a realiza un iterator este acela de a implementa conceptul de iterator printr-un tip abstract. Un astfel de tip este strîns legat de clasa care definește colecția de obiecte pentru care se dorește să se definească accesul.

Mai sus, am numit *container* o astfel de clasă. Numim *iterator* clasa atașată clasei *container* și care definește conceptul de iterator pentru ea.

Funcțiile membru ale clasei *iterator*, de obicei, trebuie să aibă acces la elementele protejate ale clasei *container* și de aceea, ea este o clasă prieten pentru clasa *container*. Acest lucru se definește simplu, de exemplu, clasa *container* de la punctul (b) de mai sus, se completează cu o declarație corespunzătoare, ca mai jos:

```
class iterator; //definicie incompleta
class container {
    obiect *colectie;
    ...
    friend class iterator;
};
```

Initializarea iteratorului este în funcție de modul în care se definește colecția

de obiecte. În toate cazurile, ea se realizează cu ajutorul constructorului clasei *iterator*.

Iteratorul clasei *container* se aplică la instanțierile clasei *container*. De aceea, o instanțiere a clasei *iterator* se realizează pentru o instanțiere a clasei *container*. De aici rezultă că, o instanțiere a clasei *iterator* trebuie să conțină un pointer sau o referință la obiectul curent de tip *container* (pentru care se realizează iterarea).

Aceasta înseamnă că, o clasă *iterator* are o dată membru de forma:

```
container *crtcontainer;
```

sau

```
container& crtcontainer;
```

O altă dată membru a unei clase iterator este destinată pentru a defini starea iterării. Această dată definește obiectul curent din *container*. Ea este actualizată la fiecare apel al iteratorului. Această dată este dependentă de modul în care este definită colecția la care este atașat *iteratorul*. În exemplul de mai sus, data respectivă poate fi un intreg care definește indicele obiectului curent, deci ea va fi declarată ca mai jos:

```
int i;
```

Ambele date membru ale clasei iterator se vor inițializa cu ajutorul constructorului clasei *iterator*.

Pentru a inițializa prima dată membru, constructorul are un parametru de tip pointer sau referință la tipul *container*:

```
container *pcontainer
```

sau

```
container& rcontainer
```

A doua dată membru, de obicei, are o inițializare implicită, dar pot apărea situații cînd se dorește să se impună obiectul din *container* de la care să se pornească iterăția. De aici rezultă că clasa *iterator*, pentru exemplul de mai sus, este de forma:

```
class iterator {
    const container *crtcontainer;
    int i;
public:
    iterator(const container& crtinst,
             int indice = 0)
    {
        crtcontainer = &crtinst;
        i = indice;
    }
    ...
};
```

Funcția membru care permite accesul la obiectele colecției *container* este o

funcție care suprainsarcă operatorul `()`. La fiecare apel al ei, se returnează un pointer sau o referință la obiectul curent din `container`. Totodată, se actualizează data membru a clasei `iterator`, pentru ca la apelul următor, să se returneze pointerul sau referința la obiectul următor din `container`.

În cazul clasei `iterator` de mai sus, vom utiliza următoarea funcție membru pentru suprainsarcarea operatorului `()`:

```
obiect *operator()()
{
    return i < crtcontainer -> crt ? & crtcontainer ->
        colectie[i++] : 0;
}
```

Fie instanțierile:

```
container colectie_objeete;
și
iterator iterator_colectie(colectie_objeete);
```

După memorarea obiectelor în obiectul `colectie_objeete` de tip `container`, conținutul acestora se poate afișa folosind instanțierea `iterator_colectie` atașată obiectului `colectie_objeete`. În acest scop se apelează funcția care suprainsarcă operatorul `()` pentru obiectul `iterator_colectie`. Apelul se realizează cu ajutorul expresiei:

```
iterator_colectie()
```

care poate fi folosit ca un operand obișnuit al unei expresii. Apelul de mai sus, returnează un pointer spre o dată de tip `obiect`.

Fie, de exemplu, declarația:

```
obiect *p;
```

atunci un ciclu de forma:

```
while((p = iterator_colectie())!=0){ ... }
```

atribuie succesiv lui `p`, pointerii spre obiectele păstrate în obiectul `colectie_objeete`.

Se observă avantajul utilizării iteratorilor pentru obiectele de tip `container`. Aceștia permit accesul la obiectele din container fără a avea în vedere detaliile legate de modul în care se realizează trecerea de la un obiect al containerului la un altul.

Propunem cititorului să implementeze clasa `iterator` și pentru celelalte variante de implementare ale clasei `container`.

În încheierea acestui paragraf vom observa că, suprainsarcarea operatorului `()` permite utilizatorului să folosească construcții asemănătoare cu cele pentru apelul funcțiilor, înlocuind numele funcției prin obiecte.

În rest, se utilizează aceleași reguli pentru transferul parametrilor și returnarea rezultatelor ca în cazul apelurilor obișnuite de funcții.

## Exerciții:

23.23 Fie clasa `cuvdif`, definită în exercițiul 23.21. Să se defincească o clasă `iterator` pentru clasa `cuvdif`.

Numim `itercuvdif` clasa iterator atașată clasei `cuvdif`. Clasa `itercuvdif` se definește după modelul clasei `iterator` atașată clasei `container` în paragraful de față.

## FIȘIERUL BXXIII23.H

```
struct fcuv {
    char *cuv;
    int f;
};

class itercuvdif;

class cuvdif {
    fcuv *pfcuv;
    int max;
    int ndif;
    int delta;
public:
    cuvdif(int m=100,int d=50);
    int& operator[](const char *cuvint);
    void ordcresc();
    void afis_frecv_cuv();
    friend class itercuvdif;
};

class itercuvdif {
    const cuvdif *container;
    int i;
public:
    itercuvdif(const cuvdif& crtcontainer,int indice=0);
    fcuv *operator()();
};
```

Funcțiile membru, definite în fișierul BXXIII21.CPP, rămân valabile și în cazul de față. Se mai adaugă funcțiile membru ale clasei iterator:

- constructorul iteratorului

și

- funcția pentru suprainsarcarea operatorului `()`.

## FIȘIERUL BXXIII23

```
#ifndef __CUVDIF_H
#include "BXXIII23.H"
#define __CUVDIF_H
#endif
#include "BXXIII21.CPP"

inline itercuvdif::itercuvdif(const cuvdif& crt,int indice)
{
    container = &crt;
```

```

    i = indice;
}

inline fcuv *itercuvdif::operator()()
{
    return i < container -> ndif ? & container -> pfcuv[i++]:0;
}

```

- 23.24 Să se scrie o funcție care citește un cuvânt de la intrarea standard. Funcția returnează valoarea 0 la întâlnirea sfîrșitului de fișier sau codul ASCII al caracterului citit care urmează imediat după cuvântul citit.

#### FUNCȚIA BXIII124

```

int citcuv(char *cuv)
/* - citește un cuvânt de la intrarea standard;
   - returnează:
     - 0 la întâlnirea sfîrșitului de fișier;
     - codul ASCII al caracterului citit care urmează imediat după cuvântul citit
*/
{
    int i,c;
    i = 0;

    // salt pe caractere care nu sunt litere
    while((c=getchar()) != EOF &&
          (c < 'A' || c > 'Z' && c < 'a' || c > 'z'))
        ;
    if(c == EOF) {      // s-a întâlnit EOF
        cuv[0] = '\0';  // cuvântul vid
        return 0;
    }

    // pastreaza caracterele cuvântului curent
    while(c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z'){
        cuv[i++]=c;
        c=getchar();
    }
    cuv[i]='\0';
    if(c==EOF) return 0;
    else return c;
}

```

- 23.25 Să se scrie un program care citește cuvintele dintr-un text și le afișează în ordine alfabetică împreună cu frecvența lor de apariție.

Programul de față este similar cu cel din exercițiul 23.22. Diferența esențială dintre ele constă în aceea că programul de față folosește clasa *itercuvdif* pentru a lista frecvența cuvintelor, spre deosebire de programul amintit mai sus, care folosește, în același scop, funcția membru *afis\_frecv\_cuv*.

#### PROGRAMUL BXIII125

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include "BXIII123.CPP"
#include "BXIII124.CPP"

main()
/* citește cuvintele din textul de la intrarea standard și le afișează în
ordine alfabetica împreună cu frecvența lor de apariție
*/
{
    char t[255];
    cuvdif text;
    int i;

    /* citește cuvintele din textul de la intrarea standard și determină frecvența lor de apariție */
    for(;;){
        i=citcuv(t);
        if(i==0&&strlen(t)==0) break; // sfîrșitul textului
        text[t]++;
        if(i==0) break; // sfîrșitul textului
    }

    // sortează cuvintele citite în ordine alfabetica
    text.ordcresc();

    // afișează frecvența cuvintelor citite
    itercuvdif obiter(text); /* se atasează un obiect de tip iterator pentru
                                obiectul text */
    fcuv *p; i=0;
    while((p=obiter())!=0){
        printf("cuvântul: %-40s are frecvența= %d\n",
               p->cuv,p->f);
        if(++i%23==0){
            printf("Actionati o tasta pentru a continua\n");
            getch();
        }
    }
}

```

- 23.26 Tipul definit de utilizator, *fcuv*, care a fost declarat în exercițiul 23.23. se modifică astfel:

```

struct fcuv {
    char *cuv;
    int f;
    struct fcuv *urm;
};

```

unde *cuv* și *f* au aceleași semnificații ca în cazul exercițiului 23.23.

Să se modifice clasele *cuvdif* și *itercuvdif* înțind seama de noua definiție a tipului *fcuv*.

Clasele *cuvdif* și *itercuvdif* au aceeași funcții *membru*. Acestea se utilizează în același mod, deci realizează aceeași interfață cu programul care le utilizează ca și cele definite în fișierele BXXIII19.CPP, BXXIII21.CPP și BXXIII23.CPP.

## FIŞIERUL BXXIII26.H

```
struct fcuv {
    char *cuv;
    int f;
    struct fcuv *urm;
};

class itercuvdif;

class cuvdif {
    fcuv *prim,*ultim;
public:
    cuvdif();
    int& operator [] ( char *cuvint);
    void ordcresc();
    void afis_frecv_cuv();
    friend class itercuvdif;
};

class itercuvdif { // clasa iterator
    fcuv *crt;
public:
    itercuvdif(const cuvdif& inceput);
    fcuv *operator () ();
};


```

Mai jos, se definesc funcțiile membru intr-un fișier de extensie *CPP*.

Clasa *cuvdif* implementează tipul abstract *cuvdif* printr-o listă simplu înăntuită. Pentru detalii în legătură cu listele simplu înăntuite se poate consulta capitolul 11.

Constructorul *cuvdif* initializează pointerii *prim* și *ultim* cu pointerul nul.

Funcția pentru suprareîncărcarea operatorului [] realizează o căutare liniară în listă a sirului spre care pointează *cuvint*. Dacă nu există, în listă, un nod în așa fel încit pointerul *cuv* să pointeze spre un sir de caractere identic cu cel spre care pointează *cuvint*, atunci se adaugă un astfel de nod la listă.

Funcția returnează o referință la componenta *f* a nodului căutat sau adăugat la listă.

Funcția *ordcresc*, ordonează nodurile în listă în așa fel încit sirurile de caractere spre care pointează componenta *cuv* să fie în ordine crescătoare.

## FIŞIERUL BXXIII26

```
#ifndef __CUVDIF_H
#include "BXXIII26.h"
#define __CUVDIF_H
#endif
```

```
inline cuvdif::cuvdif()
// initializeaza obiectul cu lista vida
{
    prim=ultim=0;
}

int & cuvdif::operator [] (char *cuvint)
/* - cauta un nod in lista asa incit componenta cuv a nodului sa pointeze spre un sir identic cu
   sirul spre care pointeaza cuvint; daca exista un astfel de nod, atunci se returneaza o referinta
   la componenta f a nodului determinat in acest fel;
   - daca nu se gaseste un nod ca cel indicat mai sus, atunci se adauga la lista un nod corespunzator
   sirului spre care pointeaza cuvint, se atribuie valoarea zero componentei f a nodului adăugat
   si se returneaza o referinta la aceasta componenta.
*/
{
    fcuv *p;
    static char er[]="memorie insuficienta\n";

    // cautare liniara in lista
    for(p=prim;p;p = p->urm)
        if(strcmp(p->cuv,cuvint) == 0) return p->f;

    /* nu s-a gasit nodul cautat in lista; se adauga un nod corespunzator sirului spre care
       pointeaza cuvint */
    char *q;

    // se pastreaza sirul in memoria hcap
    if((q=new char [strlen(cuvint)+1]) == 0){
        printf(er);
        exit(1);
    }
    strcpy(q,cuvint);

    // rezerva zona pentru nod
    if((p=new fcuv)==0){
        printf(er);
        exit(1);
    }

    /* construiește și adaugă nodul corespunzator sirului spre care pointeaza cuvint */
    p->cuv=q;
    p->f=0;
    p->urm=0;
    if(prim == 0) // lista vida
        prim=p;
    else
        /* se adaugă nodul după nodul spre care pointează ultim */
        ultim->urm=p;
    ultim=p; // ultim pointează spre nodul adăugat
    return p->f;
} // sfârșit operator []

void cuvdif::ordcresc()
/* - ordoneaza nodurile listei în așa fel încit sirurile spre care pointează
```

```

componenta cuv a nodurilor sa fie in ordine crescatoare;
-ordonarea se realizeaza prin modificarea corespunzatoare a inlantuirii nodurilor;
- in acest scop se utilizeaza metoda de sortare a bulelor.

*/
{
    fcuv *p,*q,*r;
    int ind;

    if(prim==0) return; // lista vida
    if(prim==ultim) return; // lista are un singur nod
    ind=1;
    while(ind){
        ind=0;

        /* p,q,r pointeaza spre trei noduri consecutive ale listei */

        for(p=0,q=prim,r=q->urm;r;)
            if(strcmp(q->cuv,r->cuv)>0){
                if(p==0) /* se permuta primul nod al listei */
                    prim = r;
                else p->urm=r;
                q->urm = r->urm;
                r->urm=q;
                p=r; r=q->urm;
                if(r == 0) /* s-a ajuns la ultimul nod al listei */
                    ultim = q;
                ind = 1; /* deoarece s-a facut permutare */
            } else { // nu se face permutare
                // se avanseaza in lista
                p=q; q=r;
                r=r->urm;
            }
        } // sfirsit for
    } // sfirsit while
} // sfirsit ordonare

void cuvdif::afis_frecv_cuv()
/* afiseaza frecventa de aparitie a cuvintelor diferite citite de la intrarea standard */
{
    fcuv *p;
    int i=0;

    for(p=prim;p;p->urm){
        printf("cuvintul: %-40s are frecventa= %d\n",
               p->cuv,p->f);
        if(++i%23 == 0){
            printf("Actionati o tasta pentru a continua\n");
            getch();
        }
    }
} // sfirsit afis_frecv_cuv

inline itercuvdif::itercuvdif(const cuvdif& inceput)
// constructor pentru iterator
{

```

```

    crt=inceput.prim;
}

fcuv *itercuvdif::operator () ()
/* - supraincarca operator () ; permite acces la nodul curent din lista;
   - returneaza pointerul la nodul curent din lista sau 0 daca nu exista un astfel de nod.
*/
{
    fcuv *crt1=crt; /* pastreaza pointerul spre nodul curent */
    if(crt) // avans la nodul urmator
        crt=crt->urm;
    return crt1;
}

```

### Observații:

1. Funcția membru *afis\_frecv\_cuv* este dependentă de structura nodului care se definește ca o dată de tip *fcuv* și care este un tip utilizator. Dacă *fcuv* se schimbă, atunci funcția *afis\_frecv\_cuv* poate să nu mai fie necesară sau trebuie schimbată. De aceea, de obicei se renunță la o funcție membru de acest fel și acțiunea realizată de ea se poate face folosind iteratorul. Așa cum s-a văzut mai sus, clasa iterator supraincarcă operatorul () și funcția membru respectivă permite acces secvențial la datele de tip *fcuv*. Acestea sunt neprotejate și deci se poate face acces la componentele lor în mod obișnuit.
2. În mod obișnuit, o clasă iterator are două componente "dată membru": una care este o referință la obiectul colecție curent pentru care se aplică iteratorul și alta care definește obiectul curent din container. În cazul de față este suficientă o singură dată membru și anume pointerul spre nodul curent. Aceasta, deoarece pointerul respectiv, permite el singur accesul la nodul următor din listă.
3. Programul din exercițiul 23.25., se poate modifica pentru a utiliza tipurile abstracte *cuvdif* și *itercuvdif* definite mai sus. În acest scop, este necesar să se includă fișierul BXXIII26.CPP în locul fișierului BXXIII23.CPP. Alte modificări nu sunt necesare deoarece funcțiile membru ale tipurilor abstracte de mai sus, au aceeași utilizare indiferent de faptul că sunt incluse dintr-un fișier sau altul. Ele realizează aceeași interfață pentru programul din exercițiul 23.25.

Propunem cititorului să realizeze aceleași tipuri abstracte folosind alte implementări, cum ar fi:

- listă dublu înlanțuită;
- arbore binar;
- tabelă de dispersie.

## 23.5. Supraîncărcarea operatorului ->

Supraîncărcarea operatorului  $->$  se realizează printr-o funcție membru nestică. La supraîncărcare, acest operator este considerat ca fiind un operator unar care se aplică la operandul care îl precede.

Fie expresia:

(1) obiect -> expresie

unde:

obiect - Este o instanțiere a clasei *cl*.

Această expresie se evaluează înlocuind operandul *obiect* cu apelul funcției membru care supraîncarcă operatorul  $->$  pentru obiectele clasei *cl*. Cu alte cuvinte, expresia de mai sus devine:

(2) (obiect.operator -> ()) -> expresie

În continuare, evaluarea expresiei este dependentă de tipul de dată returnat de apelul:

(3) obiect.operator -> ()

Dacă acest apel returnează un pointer spre o dată oarecare, atunci se aplică operatorul predefinit  $->$ , adică se selectează componenta definită de *expresie* a datei spre care pointează pointerul (3).

Dacă apelul (3) returnează un obiect, să zicem *obiect1*, care este o instanțiere a clasei *cl1*, atunci expresia (2) devine de forma (1) de mai sus:

obiect1 -> expresie

și în continuare urmează să se face evaluarea apelindu-se pentru *obiect1*, funcția membru a clasei *cl1* care supraîncarcă operatorul  $->$ . Dacă nu există o astfel de supraîncărcare, atunci expresia inițială este eronată.

**Exemplu:**

Fie tipul utilizator:

```
struct fcuv {
    char *cuv;
    int f;
    struct fcuv *urm;
};
```

Definim un tablou de pointeri spre elementele de tip *fcuv*:

```
fcuv *tab[MAX];
```

Expresia:

```
tab[i] -> cuv
```

utilizează operatorul  $->$  predefinit. Ea definește sirul de caractere spre care

pointează *cuv*, componentă a datei structurate de tip *fcuv* spre care pointează *tab[i]*.

Fie clasa:

```
class nod {
    fcuv *pe;
    ...
public:
    nod(char *p);
    fcuv *operator -> ()
    {
        return pe;
    }
    ...
};
```

Constructorul alocă în memoria *heap* o dată de tip *fcuv* și *pe* pointează spre ea:

```
nod::nod(char *p)
{
    pe = new fcuv;
    // Mai jos se aplică operatorul -> predefinit
    pe -> cuv = new char [strlen(p)+1];
    strcpy(pe -> cuv,p);
    pe -> f = 1;
    pe -> urm = 0;
}
```

Fie instanțierea:

```
nod crt("apa");
```

Ea aplică constructorul *nod* și se păstrează în memoria *heap* o dată de tip *fcuv*.

Pointerul *pe* pointează spre această dată. El este componentă a instanțierii *crt*, deci *crt.pe* pointează spre data de tip *fcuv* păstrată în memoria *heap*. Această dată are componentele inițializate de constructor astfel:

*cuv* - Pointează spre sirul de caractere "apa", păstrat și el în memoria *heap*.

*f* - Are valoarea 1.

*urm* - Are valoarea zero (pointerul nul).

În concluzie, expresia:

```
crt.pe -> cuv
```

pointează spre sirul "apa" păstrat în memoria *heap*. Cum *pe* este o dată membru protejată, expresia de mai sus poate fi utilizată numai în corpul unei funcții membru sau prieten al clasei *nod*.

Expresia:

```
(4) crt -> cuv
```

este legală și ea se interpretează ca mai jos:

operatorul  $\rightarrow$  se aplică instanțierii *crt* a clasei *nod*.

Deoarece pentru clasa *nod* există o supraincărcare a operatorului  $\rightarrow$ , expresia de mai sus devine:

```
(crt.operator -> ()) -> cuv
```

Apelul:

```
crt.operator -> ()
```

returnează valoarea pointerului *pe*, deci expresia (4) este echivalentă cu expresia:

```
crt.pe -> cuv
```

Aceasta, am văzut că pointează spre șirul de caractere "apa", păstrat prin intermediul constructorului în memoria *heap*.

Expresia:

```
crt -> cuv
```

nu este protejată și prin urmare ea poate fi utilizată în mod obișnuit. De exemplu, instrucțiunea:

```
printf("%s", crt -> cuv);
```

este legală și va afișa cuvintul:

apa

în poziția curentă a cursorului pe ecran.

Din acest exemplu se vede că expresia *crt.pe ->cuv* are aceeași valoare ca și expresia *crt -> cuv* în urma supraincărcării operatorului  $\rightarrow$  pentru clasa *nod*.

Diferența dintre ele se referă numai la modul de utilizare. Astfel, expresia *crt.pe ->cuv* se poate utiliza numai în corpul unei funcții membru sau prieten al clasei *nod*, în timp ce expresia *crt -> cuv* nu este protejată. Același lucru este valabil și pentru expresiile *crt.pe ->f* și *crt -> f*.

## 24. CONVERSII

Compilatoarele C și C++ execută în mod automat o serie de conversii pentru datele de tipuri predefinite. Aceste conversii intervin în următoarele situații:

1. Aplicarea unui operator la operanzi de tipuri diferite.
2. Parametrul efectiv al unei funcții are un tip diferit de parametrul formal care îi corespunde.
3. Tipul din antetul unei funcții diferă de cel al expresiei aflate în instrucțiunea *return* care realizează revenirea din funcție.

Conversiile de la punctul 1, se execută folosind regula conversiilor implicate dacă operatorul nu este de atribuire.

În principiu, această regulă se aplică în așa fel încât operandul de tip "inferior" să se convertească spre tipul "superior" al celuilalt operand. Pentru detalii se poate consulta paragraful 3.2.2.

În cazul operatorului de atribuire, valoarea expresiei din dreapta semnului de atribuire se convertește spre tipul expresiei din stanga semnului de atribuire.

În cazul conversiilor de la punctul 2, valoarea parametrului efectiv se convertește spre tipul parametrului formal care îi corespunde.

Conversiile de la punctul 3, se aplică pentru a converti tipul expresiei din instrucțiunea *return* spre tipul care se află în antetul funcției.

Conversiile pot conduce la trunchieri, exceptând cele care se realizează la punctul 1 (regula conversiilor implicate).

Conversia unei valori de tip "inferior" spre un tip "superior" se face fără trunchiere.

Amintim că utilizatorul poate, în anumite cazuri, să impună realizarea unor conversii cu ajutorul operatorului de forțare a tipului sau de conversie explicită (vezi paragraful 3.2.9.). Acest operator se utilizează în limbajul C în expresii de forma:

*(tip)operator*

Expresiile de această formă se numesc expresii *cast*, iar operatorul:

*(tip)*

il vom numi și operator *cast*.

Conversia explicită este necesară adesea pentru evaluarea corectă a expresiilor.

De exemplu, dacă *n* este o variabilă de tip *int* și se dorește valoarea raportului dintre *n* și *n+1*, atunci nu putem utiliza expresia:

*n/(n+1)*

deoarece ambeii operanzi fiind de tip *int*, se realizează împărțirea întreagă și rezultatul va fi egal cu zero. Valoarea raportului respectiv se obține dacă, de

exemplu, se convertește primul operand spre tipul *double* folosind expresia *cast*:

```
(double)n
```

Expresia devine:

```
(double)n/(n+1)
```

care se evaluatează astfel:

- intii se convertește *n* spre tipul *double*, deoarece operatorul *cast* este priorităr;
- apoi se evaluatează suma *n+1*;
- urmează să se aplice operatorul de impărțire pentru doi operanzi de tipuri diferite:
  - deimpărțitul este de tip *double*,
  - impărțitorul de tip *int*;

conform regulei conversiilor implicate, impărțitorul se convertește spre tipul *double* și apoi se realizează impărțirea flotantă și se obține rezultatul dorit.

Există și alte situații în care este obligatorie utilizarea operatorului *cast*. Așa de exemplu, la o atribuire a unui pointer spre *void* la un alt pointer, care este spre un tip diferit de *void*.

#### Exemplu:

```
int *p;  
void *v;  
...  
p = (int *)v;
```

În limbajul C++ expresiile *cast* au și un alt format decât cel indicat mai sus și anume:

```
[ tip(expresie) ]
```

Efectul acestei expresii constă în aceea că, valoarea expresiei din paranteză se convertește spre tipul *tip*.

Utilizarea obiectelor conduce și ele la situații de felul celor de mai sus în care sunt necesare conversii.

Datorită posibilității supraincărcării operatorilor, pot apărea situații în care un operator trebuie să se execute asupra operanziilor obiecte de tipuri diferite sau cind un operand este un obiect de un tip abstract, iar celălalt o dată de un tip predefinit.

De asemenea, apar situații în care se fac atribuiri în care intervin conversii de obiecte.

Situatiile 2 și 3 pot și ele interveni cind se transferă obiecte prin parametri sau se returneză un obiect la revenirea dintr-o funcție.

Astfel de conversii nu pot fi aplicate automat, deoarece compilatorul C++ nu

are definit modul de realizare al acestora, ca în cazul tipurilor predefinite.

#### Exemplu:

Considerăm tipul *complex* definit ca mai jos:

```
class complex {  
    double real; double imag;  
public:  
    complex(double x,double y)  
    {  
        real = x; imag = y;  
    }  
  
    complex operator +(complex a)  
    /* returncaza un obiect de tip complex care este echivalent cu suma *this+a */  
    {  
        complex rtemp;  
        rtemp.real = real+a.real;  
        rtemp.imag = imag+a.imag;  
        return rtemp;  
    }  
    ...  
};
```

Supraincărcarea operatorului + legalizează utilizarea expresiilor de forma:

```
z1+z2
```

unde:

*z1* și *z2* - Sunt instanțieri ale clasei *complex*.

Expresia de mai sus se realizează prin apelul:

```
z1.operator+(z2)
```

În schimb, o expresie de forma:

```
z1+3
```

nu este legală, deoarece operatorul + nu este supraincarcat decât numai pentru cazul cind ambii operanzi sunt de tip *complex*.

Pentru a legaliza o astfel de expresie putem să mai adăugăm o funcție membru care să supraincarce operatorul + ca mai jos:

```
complex operator +(double a)  
/* returncaza un obiect de tip complex echivalent cu suma *this+a */  
{  
    complex rtemp;  
    rtemp.real = real+a;  
    rtemp.imag = imag;  
    return rtemp;  
}
```

În felul acesta, expresia:

```
z1+3
```

este acceptată. La evaluarea ei se utilizează apelul:

```
z1.operator+(3)
```

Parametrul efectiv 3 se convertește spre *double* (tipul parametrului formal care îi corespunde) și apoi se execută corpul funcției operator + definit mai sus. Se observă că la evaluarea expresiei de mai sus s-a aplicat, în mod automat, o conversie implicită din tipul *int* în tipul *double*.

Funcțiile membru pentru supraincărcarea operatorului + nu legalizează utilizarea expresiei:

```
3+z1
```

Aceasta se datorează faptului că, compilatorul evaluează această expresie cu ajutorul apelului:

```
3.operator+(z1)
```

Care este eronat.

În general, dacă un operator binar este supraincărat printr-o funcție membru nestatică, atunci utilizarea operatorului respectiv în expresii, cere ca primul operand să fie de tipul implementat de clasa pentru care funcția respectivă este funcție membru.

De aceea, pentru a legaliza o expresie de această formă vom folosi o funcție prieten de felul celei de mai jos:

```
complex operator +(double a,complex z)
/* returneaza un obiect de tip complex echivalent cu suma a+z */
{
    complex rtemp;
    rtemp.real = real+a;
    rtemp.imag = imag;
    return rtemp;
}
```

sau mai simplu:

```
complex operator+(double a,complex z)
{
    return z+a;
}
```

în ipoteza că operatorul + este supraincărat pentru expresii în care operandul din stînga lui + este de tip *complex*. Cu alte cuvinte, pentru ca operatorul + să poată fi folosit pentru obiecte de tip *complex* sunt necesare 3 funcții care să supraincarce operatorul respectiv (2 funcții membru nestatice și o funcție prieten). Clasa *complex* devine:

```
class complex {
    double real;
    double imag;
public:
    complex(double x,double y)
```

```
{
    real=x;
    imag=y;
}
complex operator+(complex z);
// legalizeaza expresii de forma complex + complex

complex operator+(double d);
// legalizeaza expresii de forma complex+double

friend complex operator+(double a,complex z);
// legalizeaza expresii de forma double + complex
...;
```

Pentru ceilalți operatori sunt necesare supraincărări analoge.

Deși această soluție este simplă, ea conduce la un număr mare de funcții *operator* pentru supraincărcarea operatorilor binari.

Problema admiterii expresiilor cu operanzi de tipuri diferite pentru operatori binari poate fi rezolvată și într-un alt mod și anume prin definirea de conversii pentru date de tip abstract (obiecte).

Astfel, expresiile:

```
z1+3
```

și

```
3+z1
```

se pot legaliza dacă se definește conversia datelor de tip *int* sau alt tip predefinit (*double*, *float* etc.) spre tipul *complex*. În acest caz, este nevoie ca operatorul + să fie supraincărat numai pentru cazul în care ambii operanzi sunt de tip *complex*. De asemenea, expresiile de mai sus devin legale dacă se definește conversia obiectelor de tip *complex* spre tipul *int* sau un alt tip predefinit.

În principiu, se pot defini astfel de conversii. Utilizatorul trebuie să aibă însă grijă să nu se ajungă la ambiguitate în aplicarea conversiilor de acest tip.

În general, se pot defini următoarele tipuri de conversii:

Conversie	
din	în
1. tip predefinit	tip abstract
2. tip abstract	tip predefinit
3. tip abstract	tip abstract

Odată definite aceste conversii, ele se aplică în toate cele 3 situații, amintite la începutul acestui capitol și anume, la evaluarea expresiilor și la atribuirile, la transferul parametrilor și returnarea rezultatelor la revenirea din funcții.

Conversiile de felul celor de mai sus se numesc *conversii definite de utilizator* sau mai simplu *conversii utilizator*.

## 24.1. Conversia datelor de tipuri predefinite în date de tip abstract

Conversia unei date de un tip predefinit într-o dată de tip abstract se realizează în mod automat prin apelul unui constructor adecvat pentru tipul abstract respectiv.

În cazul clasei *complex* de mai sus, constructorul are doi parametri. El poate fi utilizat pentru a evalua expresiile:

$z1+3$

și

$3+z1$

Astfel, expresiile:

$z1+complex(3,0)$

și

$complex(3,0)+z1$

sunt legale și la evaluarea lor se apelează constructorul clasei *complex* pentru a crea un obiect anonim de tip *complex*. Apoi, se apelează funcția care suprareincarcă operatorul + pentru cazul în care ambii operanți sunt de tip *complex*. O astfel de funcție poate fi o funcție membru sau o funcție *prieten* a clasei *complex*.

Expresiile de mai sus sunt evaluate la fel ca și expresia:

$z1+z2$

unde ambii termeni sunt de tip *complex*.

**Exemplu:**

Fie clasa:

```
class complex {
    double real;
    double imag;
public:
    complex(double x,double y)
    {
        real = x;
        imag = y;
    }

    friend complex operator+(complex c1,complex c2);
    //nu mai există alte funcții pentru suprareincarcarea lui +
    ...
};
```

La evaluarea expresiei:

$z1+z2$

se realizează apelul:

`operator+(z1,z2)`

Transferul obiectelor  $z1$  și  $z2$  se realizează cu ajutorul constructorului de copiere implicit, care este o copiere bit\_cu\_bit și care este suficientă în cazul de față.

La evaluarea expresiei:

$z1+complex(3,0)$

se realizează apelul constructorului existent pentru clasa *complex* pentru a defini un obiect anonim cu partea reală egală cu 3 și cu cea imaginată egală cu 0. Apoi, se apelează funcția operator + sub forma:

`operator+(z1,complex(3,0))`

și se transferă parametri, folosind constructorul de copiere implicit.

În mod analog, se evaluează expresia:

$complex(3,0)+z1$

De aici rezultă că, pentru a legaliza expresiile de forma:

$z1+3$

și

$3+z1$

este necesar să se poată face apelurile:

`operator+(z1,3)`

și respectiv:

`operator+(3,z1)`

La aceste apeluri se construiește un obiect *complex* anonim care are partea imaginată egală cu zero, iar apoi se apelează constructorul de copiere implicit pentru a realiza transferul de parametri în mod implicit. Aceasta se poate realiza folosind un constructor care are valoarea implicită zero pentru parametrul y.

Rezultă că prezența unui astfel de constructor permite tratarea de către compilator a expresiilor de forma:

$z1+3$

și

$3+z1$

fără a mai fi nevoie de suprareincărări corespunzătoare pentru operatorul + (adică suprareincărărea pentru cazurile cind un operand nu este de tip *complex*).

Menționăm că, în exemplele date, se realizează și o conversie de date

predefinite și anume la apelul constructorului, valoarea 3 de tip *int* se convertește spre tipul *double*.

### Exemplu:

```
class complex {  
    double real;  
    double imag;  
public:  
    complex(double x,double y=0)  
    {  
        real=x; imag=y;  
    }  
    complex operator+(complex z);  
    // nu mai există supraincarcări pentru operatorul +  
    ...  
};
```

Definiția clasei *complex*, permite tratarea expresiilor de mai jos:

```
complex z1(1,2),z2(3,4);  
...  
z1+z2  
...  
z1+3  
...
```

Prima expresie implică apelul:

```
z1.operator+(z2)
```

La acest apel, pointerul *this* se încarcă cu adresa lui *z1*(&*z1*), iar *z2* se copiază apelindu-se constructorul de copiere implicit.

La cea de a doua expresie se folosește apelul:

```
z1.operator+(3)
```

La acest apel, pointerul *this* se încarcă ca mai sus, apoi se apelează constructorul clasei pentru a construi un obiect anonim care are partea imaginară egală cu 0. În final, se apeleză constructorul de copiere implicit.

Să observăm că expresiile de forma:

```
3+z1
```

nu sunt legale. Într-adevăr, o astfel de expresie conduce la un apel de forma:

```
3.operator(z1)
```

care nu este acceptat de compilator. În acest caz, compilatorul nu mai apelează constructorul pentru a crea obiectul anonim de tip *complex* corespunzător lui 3. De aceea, pentru a admite toate variantele se recomandă să utilizăm pentru supraincărcarea operatorului + o funcție prieten, aşa cum s-a procedat într-un exemplu precedent. Acest lucru este valabil și pentru ceilalți operatori ai tipului *complex*.

Dacă un operator binar poate avea ca prim operand un operand de tip diferit

deci cel pentru care se supraincarcă, atunci el se va supraincarca printre-o funcție prieten. În caz contrar, se va supraincarca printre-o funcție membru nestatică. Avantajul funcțiilor membru nestatic față de funcțiile prieten constă în aceea că în cazul funcțiilor membru nestatic, primul operand se transferă cu ajutorul pointerului *this*. Acesta se aplică în mod implicit în corpul funcției membru.

Funcțiile membru nestatic se utilizează în mod obligatoriu cind primul operand se modifică de către operatorul pe care-l supraincarcă. Exemplu de astfel de operanzi sunt operatorii de atribuire.

Să observăm că prezența constructorului de mai sus, pentru clasa *complex*, nu intră în contradicție cu funcțiile care supraincarcă operatorul + pentru cazurile cind nu sunt ambii operanzi de tip *complex*.

Într-adevăr, fie clasa *complex*, definită ca mai jos:

```
class complex {  
    double real;  
    double imag;  
public:  
    complex(double x,double y=0)  
    {  
        real=x; imag=y;  
    }  
  
    complex operator+(complex z);  
    // realizeaza suma *this+z  
  
    complex operator+(double a);  
    // realizeaza suma *this+a  
  
    friend complex operator+(double a,complex z);  
    // realizeaza suma a+z  
};
```

și declarațiile:

```
complex c(1,2);  
int n;  
...
```

Expresia:

```
c+n
```

se evaluatează cu ajutorul apelului:

```
c.operator+(n)
```

La acest apel, se convertește valoarea lui *n* spre *double* și apoi se apelează, în mod obișnuit, funcția membru care supraincarcă operatorul +, de antet:

```
complex operator+(double)
```

În mod analog, expresia:

se evaluatează cu ajutorul apelului:

```
operator+(n, c)
```

La acest apel se convertește valoarea lui *n* spre *double*, apoi se apelează funcția prieten care supraincarcă operatorul +.

În ambele cazuri, nu intervine conversia din tipul predefinit *double* spre tipul *complex*.

Conversia dintr-un tip predefinit într-un tip abstract se realizează numai dacă nu se poate selecta o funcție membru sau prieten a cărei parametri să nu necesite o astfel de conversie.

Ca o concluzie de reținut este faptul că, o conversie dintr-un tip predefinit într-un tip abstract se realizează de către un constructor al clasei prin care se implementează tipul abstract respectiv. Acest constructor are un parametru care trebuie să fie de un tip predefinit care se poate converti spre tipul abstract. Constructorul poate avea și alți parametri, dar aceștia vor fi toți implicați. O conversie dintr-un tip predefinit într-un tip abstract poate fi precedată de alte conversii care se realizează între tipuri predefinite. De exemplu, mai sus, constanta 3 a fost convertită din tipul *int* în tipul *double* și apoi în tipul *complex*.

Conversiile de acest tip au o prioritate redusă. Ele se aplică atunci cind supraincărcarea operatorilor nu se aplică în mod direct sau folosind conversii predefinite. De exemplu, atribuirea:

```
z=n
```

unde *z* este de tip *complex*, iar *n* de tip predefinit (de exemplu *int*) se realizează apelind funcția membru nestatică care supraincarcă operatorul de atribuire pentru clasa *complex* și care are antetul:

```
complex& operator = (double a)
```

dacă o astfel de funcție există. În caz contrar, se aplică constructorul care convertește pe *n* într-un obiect anonim de tip *complex* și apoi se face atribuirea.

Conversiile definite de utilizator pe baza aplicării automate a constructorului se numesc conversii utilizator *implicite*.

Obiectele construite prin conversii utilizator implicite se distrug automat cind nu mai este nevoie de ele.

Conversiile utilizator implicite sunt adesea utile deoarece, așa cum s-a arătat mai sus, permit reducerea substanțială a funcțiilor pentru supraincărcarea operatorilor. Cu toate acestea, aplicarea conversiilor utilizator implicite trebuie evaluată în legătură cu eficiența lor față de utilizarea funcțiilor de supraincărcare a operatorilor pe care le elimină.

De asemenea, la aplicarea conversiilor utilizator implicite va trebui să avem în vedere posibilitățile de apariție a ambiguităților.

Fie, de exemplu, funcția *f* supraincarcată ca mai jos:

```
void f(cl1 x);
void f(cl2 x);
```

unde:

*cl1* și *cl2*

- Sunt două clase care au constructori de prototipuri:

*cl1(double);*

și

*cl2(double);*

Acești constructori, având un singur parametru, realizează conversii utilizator implicate din *double* spre tipul abstract *cl1*, respectiv *cl2*.

Un apel de forma:

```
f(3.14159265);
```

este eronat, deoarece nu este definită în mod unic conversia utilizator implicită. În acest caz este necesară aplicarea explicită a constructorului:

```
f(cl1(3.14159265));
```

sau

```
f(cl2(3.14159265));
```

Să observăm că, dacă funcția *f* nu este supraincarcată, de exemplu este valabil numai primul prototip, atunci apelul inițial de mai sus este corect și se va realiza în mod automat conversia din *double* spre *cl1*.

#### Observație:

Fie *cl* o clasă care se definește ca mai jos:

```
class cl {
    ...
public:
    cl();           // constructor implicit
    cl(tip_predefinit); /*constructor pentru conversii utilizator implicit */
    cl(...);
    cl(const cl&); // constructor de copiere
    ...
    cl& operator=(cl); /* supraincărcarea operatorului de atribuire */
    ...
};
```

Constructorii și funcția de supraincărcare a operatorului de atribuire se utilizează ca mai jos:

```
cl ob1,ob2; // se aplică constructorul implicit
tip_predefinit t;
...
cl ob3(t); /* se aplică constructorul pentru conversii utilizator implicit */
cl ob4=t;   /* se aplică constructorul pentru conversii implicit; se obtine
```

```

        un obiect anonim de tip cl, apoi acesta se copiaza bit_cu_bit
        in obiectul ob4; dupa copiere se distrug obiectul anonim */

cl ob5(ob4); // se aplica constructorul de copiere
cl ob6 = ob4; // se aplica constructorul de copiere
ob1 =ob3; /*se apeleaza functia care supraincarca operatorul =
/* se aplica constructorul pentru conversii utilizator implicite si
   se creaza un obiect anonim de tip cl, apoi acesta se copiaza
   bit_cu_bit in obiectul ob2; dupa copiere obiectul anonim se
   distrug */

void f(cl); // prototipul functiei f
...
f(ob1); /*se aplica constructorul de copiere pentru a atribui parametrului
           formal al functiei f, valoarea parametrului efectiv ob1 */
...
f(t); /* se apeleaza constructorul pentru conversii implicite; se obtine
         un obiect anonim de tip cl, apoi acesta se copiaza bit_cu_bit
         in parametrul formal al functiei f; dupa copiere se distrug
         obiectul anonim */

```

### Exerciții:

24.1 Să se definească tipul abstract *complex* suprarecăind operatorii pentru operații cu numere complexe cu un număr minim de funcții.

Tipul *complex* a fost implementat în exercițiul 23.1, folosind cîte 3 funcții pentru fiecare operator binar.

În acest exercițiu, fiecare operator binar va fi suprarecărat printr-o singură funcție, care este o funcție prieten. Excepție de la această regulă o constituie operatorul de împărțire pentru care se definesc două funcții: una care este funcție prieten și se apelează la împărțirea de numere complexe cind ambele operanzi sunt numere complexe sau cind împărțitorul este complex, iar deîmpărțitul nu este complex și una care este funcție membru nestatică și se aplică atunci cind împărțitorul nu este complex. În acest caz, este mai eficient ca împărțirea să nu se facă prin metoda generală de împărțire a două numere complexe.

Celelalte funcții membru sint aceleași ca în exercițiul 23.1.

### FIȘIERUL BXXIV1.H

```

class complex {
    double real;
    double imag;

    static int citdouble(char *s,double& d);
    /* - afiseaza s;
       - citeste un numar si-l atribuie datci referite de d;
       - returneaza:
          0 - la sfîrșit de fisier;
          1 - altfel.
    public: // constructor
        complex(double x=0,double y=0);

```

```

    0,double y=0);
/* se utilizeaza pentru:
   - instantieri de obiecte fara parametri (real=imag=0);
   - instantieri de obiecte cu 1-2 parametri;
   - conversie utilizator implicita dintr-un tip predefinit in tipul complex.
*/

// modulul numarului complex
double modul();

// argumentul numarului complex
double arg();

// citeste un numar complex
int citcomplex(char *s);
/* - afiseaza s;
   - citeste componentele numarului complex;
   - returneaza:
      0 - la sfîrșit de fisier;
      1 - altfel.
*/

// afiseaza un numar complex
void afiscomplex(char *f);
/* - afiseaza componentele numarului complex;
   - se utilizeaza formatul spre care pointeaza f sau un format standard daca f
     pointeaza spre sirul vid. */

// incrementeaza partea imaginara
void ipi();

// decrementeaza partea imaginara
void dpi();
/* - obiectul curent se noteaza cu z;
   - obiectul rezultat se noteaza cu r. */

// operatori unari

// negativare
complex operator -(); // r=-z

// incrementeaza partea reala
complex operator ++();

// decrementeaza partea reala
complex operator --();

// radacina patrata
complex operator !();

// operatori binari

// adunare r = z1+z2

```

```

friend complex operator +(complex z1,complex z2);
// scadere r = z1-z2
friend complex operator -(complex z1,complex z2);

// imnultric r = z1*z2
friend complex operator *(complex z1,complex z2);

// impartire r = z1/z2
friend complex operator /(complex z1,complex z2);

// r = z/d
complex operator /(double d);

// ridicare la putere r = z**i
complex operator ^(int i);

// test de egalitate z1 == z2
friend int operator == (complex z1,complex z2);

// returnaza partea reala
double retreal();

// returnaza partea imaginara
double retimag();
};

Funcțiile membru se definesc în fișierul de mai jos, de extensie CPP.

```

## FISIERUL BXXIV1

```

#ifndef __STDLIB_H
#include <stdio.h>
#define __STDLIB_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __COMPLEX_H
#include "BXXIV1.H"
#define __COMPLEX_H
#endif

```

```

inline complex::complex(double x,double y)
/* constructor utilizat la initializari si conversia utilizator implicita
dintr-un tip predefinit in tipul complex */
{
    real = x; imag = y;
}

inline double complex::modul() // returnaza modulul numarului complex
{
    return sqrt(real*real+imag*imag);
}

double complex::arg() // returnaza argumentul numarului complex
{
    if(real==0&&imag==0) return 0.0;
    if(imag==0)
        if(real > 0) return 0.0;
        else return PI;
    if(real==0)
        if(imag > 0) return PI/2;
        else return (3*PI/2);
    double a = atan(imag/real);
    if(real < 0) return PI+a;
    if(imag < 0) return 2*PI+a;
    return a;
}

int complex::citdouble(char *s,double& d)
/* - afisaza s;
   - citeste un numar si-l atribuie datei referite de d;
   - returnaza:
      0 - la sfarsit de fisier;
      1 - altfel.
*/
{
    char t[255];
    for(;;){
        if(s && *s) printf(s);
        if(gets(t)==0) return 0;
        if(sscanf(t,"%lf",&d)==1) break;
        printf("nu s-a tastat un numar\n");
        printf("se reia citirea\n");
    }
    return 1;
}

int complex::citcomplex(char *s)
/* - afisaza s daca nu este vid; daca este vid se afiseaza text standard:
   Partea reala si Partea imaginara;
   - citeste componentele obiectului complex curent;
   - returnaza:
      0 - la sfarsit de fisier;
      1 - altfel. */

```

```

{
    if(s && *s) { // sirul s nu este vid;
        // se citesc componentele obiectului curent
        for(;;) {
            printf(s);
            int i;
            double d;
            if((i=fscanf("%lf",&d)) == EOF) return 0;
            if(i!=1){
                printf("nu s-a tastat un numar pentru \
                        partea reala\n");
                printf("se reia citirea de la inceput\n");
                fflush(stdin);
                continue;
            }
            else real = d;
            if((i=fscanf("%lf",&d))==EOF) return 0;
            if(i!=1){
                printf("nu s-a tastat un numar\n");
                printf("pentru partea imaginara\n");
                printf("se reia citirea de la inceput\n");
                fflush(stdin);
            }
            else{
                imag=d;
                fflush(stdin);
                return 1;
            }
        } // sfisit for
    } // sfisit if(s && *s)

    // s pointeaza spre sirul vid sau cste pointerul zero

    if(complex::citdouble("Partea reala=",real) == 0) return 0;
    if(complex::citdouble("Partea imaginara=",imag) == 0)
        return 0;
    return 1;
}

void complex::afiscomplex(char *f)
/* - afiseaza componentele obiectului curent cu formatul f;
   - daca f prezinta sirul vid, se utilizeaza un format standard */
{
    if(f && *f) // formatul de afisare nu este vid
        printf(f,real,imag);
    else // format standard
        printf("%g+i*(%g)\n",real,imag);
}
inline void complex::ipi()
/* incrementarea partea imaginara a obiectului complex curent */
{
    imag++;
}

```

```

inline void complex::dipi()
/* decrementarea partea imaginara a obiectului curent */
{
    imag--;
}

complex complex::operator-() // r=-z
{
    complex r;
    r.real = -real; r.imag = -imag;
    return r;
}

inline complex complex::operator++()
/* incrementarea partea reala a obiectului curent */
{
    real++;
    return *this;
}

inline complex complex::operator--()
/* decrementarea partea reala a obiectului complex curent */
{
    real--;
    return *this;
}

complex complex::operator }()
/* calculeaza radacina patrata din obiectul complex curent */
{
    complex r; // real=imag=0
    double d;

    if((d=modul())==0) return r; // obiect complex nul
    double alfa=arg(); // argumentul obiectului complex
    d = sqrt(d); // radacina patrata din modul
    alfa = alfa/2;
    r.real = d*cos(alfa); r.imag = d*sin(alfa);
    return r;
}

complex operator+(complex z1,complex z2) // r=z1+z2
{
    complex r;
    r.real = z1.real+z2.real; r.imag = z1.imag+z2.imag;
    return r;
}

complex operator -(complex z1,complex z2) // r=z1-z2
{
    complex r;

```

```

r.real = z1.real-z2.real; r.imag = z1.imag-z2.imag;
return r;
}

complex operator *(complex z1,complex z2) //r=z1*z2
{
    complex r;

    r.real = z1.real*z2.real-z1.imag*z2.imag;
    r.imag = z1.real*z2.imag+z1.imag*z2.real;
    return r;
}

complex operator /(complex z1,complex z2) //r=z1/z2;r=0 daca z2=0
{
    complex r; //real=imag=0
    double d = z2.real*z2.real+z2.imag*z2.imag;

    if(d==0) //z2=0
        return r;
    r.real = (z1.real*z2.real+z1.imag*z2.imag)/d;
    r.imag = (z1.imag*z2.real-z1.real*z2.imag)/d;
    return r;
}

complex complex::operator /(double d) //r=z/d;r=0 daca d=0
{
    complex r; //real=imag=0

    if(d==0) return r;
    r.real = real/d; r.imag = imag/d;
    return r;
}

complex complex::operator ^ (int i) //r=z**i;r=0 daca z=0
{
    complex r; //real=imag=0
    double d=modul();

    if(d==0) return r;
    int n=i < 0 ? -i : i; //n=abs(i)
    if(n==0) //z**0=1
        return 1;
    if(i==1) //z**1=z
        return *this;
    if(i==-1) //z**(-1)=1/z
        r = *this;
    if(n > 1){
        d = pow(d,(double)n); //d=d**n
        double a = arg()*n; //argument*n

        //reducere la primul cerc

        long x = a/PI;
    }
}

```

```

    a -= x*PI;
    r.real = d*cos(a); r.imag = d*sin(a);
}
if(i < 0) //z**i=1/z**n; r=z**n
    r = 1/r; /* operatorul este definit pentru complex/complex;
               se aplica conversia implicita */
return r;
}

inline int operator == (complex z1,complex z2)
/* returneaza:
   1 - daca z1=z2;
   0 - altfel.
*/
{
    return z1.real == z2.real && z1.imag == z2.imag;
}

inline double complex::retreal()
// returneaza partea reala a obiectului complex curent
{
    return real;
}

inline double complex::retimag()
// returneaza partea imaginara a obiectului complex curent
{
    return imag;
}

```

24.2 Să se scrie un program care citește valorile variabilelor  $n$ ,  $a$  și  $b$ , calculează și afișează rădăcinile ecuației binome:

$$a*x^{**n}+b = 0$$

unde:

$n$  - este un întreg pozitiv;  
 $a, b$  - sint numere complexe.

Rădăcinile ecuației binome se obțin calculind rădăcinile de ordinul  $n$  din numărul complex  $-b/a$ , dacă  $n > 1$ . Pentru  $n=1$  ecuația de mai sus este de grad 1 și are o singură soluție:

$$x = -b/a$$

Rădăcinile de ordinul  $n$  dintr-un număr complex  $z$  se determină din forma trigonometrică a acestuia.

Fie  $r$  și  $\alpha$ , modulul și respectiv argumentul numărului complex  $z$ . Atunci rădăcina a  $k$ -a de ordinul  $n$  din numărul complex  $z$  se calculează astfel:

$$\text{rad}_k = r^n * (\cos(\beta_k) + i \sin(\beta_k))$$

unde:

$r^n$  - Este rădăcina de ordinul  $n$  din  $r$ .

```
beta_k = alfa/n+k*(2*PI/n)
```

În aceste relații,  $k$  variază de la 0 la  $n-1$ .

## PROGRAMUL BXXIV2

```
#include "BXXIV1.CPP"
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXX4.CPP" // rcit_int
#include "BXX5.CPP" // rcit_int_lim

main()
/* - citeste pe: n,a si b;
 - calculeaza si afiscaza radacinile ecuatiei binome:
   a*x**n+b=0
   unde:
     n - intreg pozitiv;
     a,b - numere complexe
*/
{
    int k,n;
    complex a,b,x,z;

    for(;;){
        printf("Actionati o tasta pentru a continua\n");
        getch();
        if(rcit_int_lim("gradul ecuatiei n=",1,32767,n)==0)
            exit(0);
        printf("coeficientul lui x\n");
        if(a.citcomplex("")==0) exit(1); // EOF
        printf("termenul liber\n");
        if(b.citcomplex("")==0) exit(1); // EOF
        if(a==0&&b==0){
            printf("ecuatie nedeterminata\n");
            continue; // se trece la alta ecuatie
        }
        if(a==0){
            printf("ecuatia nu are solutie\n");
            continue; // se trece la alta ecuatie
        }
        z=-b/a;
        if(n==1){ // ecuatie de gradul 1
            printf("ecuatie de gradul 1\n");
            z.afiscomplex("x=%g+i(%g)\n");
            continue; // se trece la alta ecuatie
        }
        // ecuatie de grad mai mare ca 1
        // se determina modulul si argumentul numarului complex z

        double r = z.modul();
    }
}
```

```
double alfa = z.arg();
double rn = pow(r,1.0/n); // radacina de ordinul n din r
alfa = alfa/n;
double doipien = 2*PI/n;

// calculaza si afiscaza cele n radacini
for(k=0;k < n;k++){
    double beta = alfa+k*doipien;
    x=complex(rn*cos(beta),rn*sin(beta));
    printf("x%d=%",k);
    x.afiscomplex("");
    if((k+1)%10==0){
        printf("Actionati o tasta pentru a continua\n");
        getch();
    }
    printf("\n");
}
}
```

24.3 Să se scrie un program care rezolvă ecuații de gradul 2 cu coeficienți complecsi.

## PROGRAMUL BXXIV3

```
#include <conio.h>
#include "BXXIV1.CPP"

main()
/* rezolva ecuatii de gradul 2 cu coeficienți complecsi de forma:
   a*x*x+b*x+c = 0
*/
{
    complex a,b,c,x;

    for(;;){
        printf("Actionati o tasta pentru a continua\n");
        getch();
        if(a.citcomplex("a")==0) exit(0); // EOF
        if(b.citcomplex("b")==0) exit(1); // EOF
        if(c.citcomplex("c")==0) exit(1); // EOF
        if(a==0 && b==0 && c==0){
            printf("ecuatie nedeterminata\n");
            continue;
        }
        if(a==0 && b==0){
            printf("ecuatia nu are solutie\n");
            continue;
        }
        if(a==0){ // ecuatie de gradul 1
            printf("ecuatie de gradul 1\n");
            x = -c/b;
            x.afiscomplex("x=%g+i(%g)\n");
            continue;
        }
    }
}
```

```

        }

        // a != 0 - ecuatie de gradul 2
        complex doia = 2*a;
        complex delta = !(b*b-4*a*c);
        x = (-b+delta)/doia;
        x.afiscomplex("x1=%g+i*(%g)\n");
        x=(-b-delta)/doia;
        x.afiscomplex("x2=%g+i*(%g)\n");
    }
}

```

24.4 Să se implementeze tipul abstract *grad* pentru lucrul cu grade sexagesimale.

Un grad sexagesimal are 60 de minute sexagesimale, iar un minut sexagesimal are 60 de secunde sexagesimale.

Tipul abstract *grad* permite operații cu grade sexagesimale:

- adunare;
- scădere;
- înmulțire și impărțire în care un operand este de tip *grad* și unul este de tip predefinit.

De asemenea, clasa *grad* are funcții membru care permit citiri și afișări de obiecte de tip *grad*.

Clasa are 4 date membru protejate:

- *sgrad*;
- *smin*;
- *ssec*;

și

- *totsec*

Între aceste date există relația:

$$\text{totsec} = \text{sgrad} * 3600 + \text{smin} * 60 + \text{ssec}$$

Funcția membru *grminsec*, determină valorile componentelor *sgrad*, *smin* și *ssec* din valoarea datei membru *totsec*.

Există doi constructori, unul care permite inițializarea obiectelor cu valoarea variabilei *totsec*, iar unul care permite inițializarea obiectelor cu valori pentru *sgrad*, *smin* și *ssec*.

În mod analog, există 2 funcții membru pentru a citi valori numai pentru componenta *totsec* sau pentru a citi valori exprimate în grade, minute și secunde.

#### FISIERUL BXXIV4.H

```

class grad {
    long totsec;
    long sgrad;
    int smin;
    int ssec;
}

```

```

void grminsec();
/* converteste din total secunde in grade, minute si secunde */

public:
    grad(long s=0);
    /* - permite declarari de obiecte care se initializaaza dindu-se numai secundele;
       - permite conversie utilizator implicita. */

    grad(long gr,int min,int sec);
    /* initializaaza obiecte dindu-se gradele, minutele si secundele */

    int citsec();
    /* - citeste un intreg de tip long care reprezinta secundele pentru obiectul curent;
       - returneaza zero la intalnirea sfirsitului de fisier si 1 altfel. */

    int citgms();
    /* - citeste trei intregi care reprezinta grade, minute si secunde si
       care se atribuie respectiv datelor sgrad, smin si ssec ale obiectului curent;
       - returneaza zero la intalnirea sfirsitului de fisier si 1 altfel. */

    void afissec(char *f);
    /* afisaza, folosind formatul f, valoarea variabilei totsec pentru obiectul curent;
       daca f pointeaza spre sirul vid, atunci se utilizeaza un format standard */

    void afisgrad(char *f);
    /* afisaza cu formatul f, valoarea variabilelor sgrad, smin si ssec;
       daca f pointeaza spre sirul vid, atunci se utilizeaza un format standard */

    friend grad operator +(grad g1,grad g2);
    // returneaza suma g1+g2

    friend grad operator -(grad g1,grad g2);
    // returneaza diferența g1-g2

    grad operator -();
    // returneaza obiectul curent negativ

    grad operator *(double a);
    // returneaza produsul g*a; g este obiectul curent

    friend grad operator *(double a,grad g2);
    // returneaza produsul a*g2
    grad operator /(double a);
    // returneaza cîtul g/a; g este obiectul curent

    grad& operator += (grad g2);
    // g = g+g2; g este obiectul curent

    grad& operator -= (grad g2);
    // g = g-g2; g este obiectul curent

    long retsec();
    // returneaza valoarea lui totsec pentru obiectul curent

```

```

long retg();
// returnaza gradele pentru obiectul curent

int retm();
// returnaza minutele pentru obiectul curent

int rets();
// returnaza secundele pentru obiectul curent

friend int operator == (grad g1,grad g2);
// returnaza 1 daca g1=g2 si zero altfel

friend int operator < (grad g1,grad g2);
// returnaza 1 daca g1 < g2 si zero altfel

friend int operator <= (grad g1, grad g2);
// returnaza 1 daca g1 <= g2 si zero altfel
};

```

#### Observație:

Modul în care au fost supraincarcați operatorii \* și / asigură protecție împotriva înmulțirilor sau împărțirilor cu ambii operanzi de tip *grad*.

Mai jos, se definesc funcțiile membru în fișierul de extensie *cpp*.

Tot în acest fișier se apelează două funcții care permit citirea intregilor de tip *long*. Ele sunt asemănătoare cu funcțiile *rcit\_int* și *rcit\_int\_lim* definite în exercițiile 20.4 și respectiv 20.5. Numele lor se obțin din numele funcțiilor respective schimbând cuvintul *int* cu *long*.

#### FIŞIERUL BXXIV4

```

#ifndef __GRAD_H
#include "BXXIV4.H"
#define __GRAD_H
#endif

void grad::grminsec()
/* convertește din totalul secunde in grade, minute si secunde */
{
    sgrad = totsec/3600;
    int rest = totsec%3600;
    smin = rest/60;
    ssec = rest%60;
}

inline grad::grad(long s)
/* initializeaza totsec cu valoarea lui s si apoi si celelalte componente in mod corespunzator */
{
    totsec = s;
    grminsec();
}

```

```

inline grad::grad(long g,int m,int s)
/* asigura instantierea obiectelor cu valori pentru grade, minute si secunde */
{
    totsec = g*3600+m*60+s;
    grminsec();
}

int grad::citsec()
/* - citeste un intreg de tip long care reprezinta secunde pentru obiectul curent;
   - returneaza:
       0 - la intlnirea sfirsitului de fisier;
       1 - altfel.
*/
{
    long s;

    if(rcit_long_lim("Masura in secunde=",
                      -2147483648,2147483647,s)==0)
        return 0;
    totsec=s;
    grminsec();
    return 1;
}

int grad::citgms()
/* - citeste trei intregi care reprezinta grade, minute si secunde si care se
   atribuie respectiv datelor sgrad, smin si ssec ale obiectului curent;
   - returneaza:
       0 - la intlnirea sfirsitului de fisier;
       1 - altfel.*/
{
    long g;

    if(rcit_long_lim("grade=", -2147483648,2147483647,g)==0)
        return 0;

    int m;

    if(rcit_int_lim("minute=", -60,60,m)==0) return 0;
    int s;
    if(rcit_int_lim("secunde=", -60,60,s)==0) return 0;
    totsec = g*3600+m*60+s;
    grminsec();
    return 1;
}

void grad::afissec(char *f)
/* afisaza, folosind formatul f, valoarea variabilei totsec; daca f
   pointeaza spre sirul vid, atunci se utilizeaza un format standard */
{
    if(f&&*f) printf(f,totsec);
    else printf("masura in secunde=%ld\n",totsec);
}

```

```

void grad::afisgrad(char *f)
/* afiseaza, folosind formatul f, masura unghiului in grade, minute si secunde;
   daca f pointeaza spre sirul vid, atunci se utilizeaza un format standard */
{
    if(f&&*f) printf(f,sgrad,smin,ssec);
    else printf("grade=%ld\tminute=%d\tsecunde=%d\n",F 255
                sgrad,smin,ssec);
}

grad operator +(grad g1,grad g2) // returneaza suma g1+g2
{
    grad r;
    r.totsec=g1.totsec+g2.totsec;
    r.grminsec();
    return r;
}

grad operator -(grad g1,grad g2) // returneaza diferența g1-g2
{
    grad r;
    r.totsec = g1.totsec-g2.totsec;
    r.grminsec();
    return r;
}

grad grad::operator-() // returneaza negativul obiectului curent
{
    grad r;
    r.totsec=-totsec;
    r.grminsec();
    return r;
}

grad grad::operator*(double a) // returneaza g*a; g este obiectul curent
{
    grad r;
    r.totsec=totsec*a;
    r.grminsec();
    return r;
}

grad operator *(double a,grad g2) // returneaza a*g2
{
    return g2*a;
}

grad grad::operator/(double a) // returneaza g/a; g este obiectul curent
{
    grad r;
}

```

```

    r.totsec=totsec/a;
    r.grminsec();
    return r;
}

grad& grad::operator += (grad g2) // g=g+g2; g este obiectul curent
{
    totsec += g2.totsec;
    grminsec();
    return *this;
}

grad& grad::operator -= (grad g2) // g=g-g2; g este obiectul curent
{
    totsec -= g2.totsec;
    grminsec();
    return *this;
}

inline long grad::retsec() // returneaza masura unghiului in secunde
{
    return totsec;
}

inline long grad::retg() // returneaza gradele pentru obiectul curent
{
    return sgrad;
}

inline int grad::retm() // returneaza minutele pentru obiectul curent
{
    return smin;
}

inline int grad::rets() // returneaza secundele pentru obiectul curent
{
    return ssec;
}

inline int operator == (grad g1,grad g2)
// returneaza 1 daca g1==g2 si zero altfel
{
    return g1.totsec==g2.totsec;
}

inline int operator < (grad g1,grad g2)
// returneaza 1 daca g1 < g2 si zero altfel
{
    return g1.totsec < g2.totsec;
}

inline int operator <= (grad g1,grad g2)
// returneaza 1 daca g1 <= g2 si zero altfel
{
}

```

```

    return g1.totsec <= g2.totsec;
}

```

- 24.5 Să se scrie o funcție care afișează caracterele unui tablou și citește un întreg de tip *long*. Funcția are doi parametri:

*text*: tablou unidimensional de tip caracter și care are aceeași utilizare ca în exercițiile 6.3, 8.2 și 20.4;  
*x*: parametru de tip referință la întregii de tip *long*; numărul citit se păstrează în zona de memorie alocată parametrului efectiv corespunzător lui *x*.

### FUNCȚIA BXXIV5

```

long rcit_long(char text[], long& x)
/* - afișaza sirul de caractere spre care pointează text;
   - citește un întreg și-l păstrează în zona de memorie alocată parametrului efectiv corespunzător lui x;
   - returnează:
     0 - la intilnirea sfîrșitului de fisier;
     1 - altfel.*/
{
    char t[255];

    for(;;){
        printf(text);
        if(gets(t)==0) return 0;
        if(sscanf(t,"%ld",&x)==1) return 1;
        printf("Nu s-a tastat un întreg\n");
    }
}

```

- 24.6 Să se scrie o funcție care citește un întreg de tip *long* care aparține unui interval dat.

Această funcție este analogă cu funcțiile definite în exercițiile 8.3 și 20.5. Funcțiile respective citesc întregi de tip *int* care aparțin unui interval dat.

### FUNCȚIA BXXIV6

```

long rcit_long_lim(char text[], long inf, long sup, long& rlong)
/* - afișaza sirul de caractere spre care pointează text;
   - citește un întreg de tip long ce aparține intervalului [inf,sup] și-l
     păstrează în zona de memorie alocată parametrului efectiv corespunzător lui rlong;
   - returnează:
     0 - la intilnirea sfîrșitului de fisier;
     1 - altfel.*/
{
    for(;;){
        if(rcit_long(text,rlong)==0) return 0; // s-a intilnit EOF
        if(rlong >= inf && rlong <= sup) return 1;
        printf("intregul tastat nu aparține intervalului:");
        printf("[%ld,%ld]\n",inf,sup);
    }
}

```

```

    printf("se reia citirea\n");
}
}

```

- 24.7 Să se scrie un program care citește măsura unei laturi a unui triunghi, precum și măsurile, în grade sexagesimale, ale unghiurilor alăturate laturii respective și pe baza acestor date calculează și afișează măsurile celorlalte elemente ale triunghiului.

Programul rezolvă succesiv problema enunțată pentru mai multe triunghiuri și se oprește la intilnirea sfîrșitului de fișier.

Fie *a* măsura laturii citite și *B* și *C* măsurile în grade sexagesimale ale unghiurilor alăturate lui *a*. Atunci  $A = 180 - B - C$  și  $a/\sin A = b/\sin B = c/\sin C$ .

Măsurile laturilor *b* și *c* se găsesc conform relațiilor:

$$b = (a/\sin A) * \sin B$$

și

$$c = (a/\sin A) * \sin C$$

Pentru calculul funcției sinus este nevoie ca măsura unghiului să fie în radiani.

Trecerea de la grade sexagesimale în radiani se poate face cu ajutorul relației:

$$r = (\pi/180)*n$$

unde:

*r* Este măsura unghiului în radiani.

*PI* Are valoarea  $3.14159265358979$ .

*n* Este măsura unghiului în grade sexagesimale.

Dacă măsura unghiului se dă în secunde, atunci relația de mai sus devine:

$$r = (\pi/(180*3600))*s$$

unde prin *s* s-a notat măsura unghiului în secunde. Aceasta se mai scrie:

$$r = (\pi/648000)*s$$

### PROGRAMUL BXXIV7

```

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

#include "BXX4.CPP"      // rcit_int
#include "BXX5.CPP"      // rcit_int_lim
#include "BXXIV5.CPP"    // rcit_long
#include "BXXIV6.CPP"    // rcit_long_lim

#define PI 3.14159265358979
#include "BXXIV4.CPP"    // implementarea tipul grad

```

```

main()
/* - citeste:
   - masura laturii a;
   - masurile in grade sexagesimale ale unghiurilor B si C, alaturate laturii a;
   - calculeaza si afisaza masurile celorlalte elemente ale triunghiului */
{
    double a;
    grad A,B,C;
    double factor = PI/648000;
    char er[] = "s-a tastat EOF\n";
    char erunghi[]="Masura unghiului nu este in (0,180)\n";

    for(;;){
        // citeste masura laturii a
        for(;;){
            char t[255];
            printf("masura laturii a=");
            if(gets(t)==0) exit(0); //EOF
            if(sscanf(t,"%lf",&a) == 1 && a > 0) break;
            printf("nu s-a tastat un numar pozitiv\n");
        }
        for(;;){
            // citeste masura unghiului B
            // se tasteaza grade minute si secunde
            for(;;){
                printf("B=");
                if(B.citgms()==0){
                    printf(er);
                    exit(1);
                }
                if(0 < B && B < 180*3600L) break;
                printf(erunghi);
            }
            for(;;){
                printf("C=");
                if(C.citgms()==0){
                    printf(er);
                    exit(1);
                }
                if(0 < C && C < 180*3600L) break;
                printf(erunghi);
            }
            A = 180*3600L-(B+C);
            if(A <= 0){
                printf("Suma B+C depaseste 180 de grade\n");
                printf("Se reia citirea masurilor unghiurilor\n");
                continue;
            }
            break;
        } // sfarsit citirea datelor pentru un triunghi
        // afisaza datele citite
        printf("\n\n Masura laturii a = %g\n",a);
        B.afisgrad("Masura unghiului B: grade=%ld\

```

```

minute=%d secunde = %d\n");
C.afisgrad("Masura unghiului C: grade=%ld\
minute=%d secunde = %d\n");

// afiseaza datele calculate
printf("\n\n Date calculate\n");
A.afisgrad("Masura unghiului A: grade=%ld\
minute=%d secunde = %d\n");
double f=a/sin(A.retsec()*factor);
double b=f*sin(B.retsec()*factor);
double c = f*sin(factor*C.retsec());
printf("Masura laturii b = %g\n",b);
printf("Masura laturii c = %g\n",c);
}
}

```

## 24.2. Conversia dintr-un tip abstract într-un tip predefinit

Conversia datelor de tip abstract spre o dată de tip predefinit se realizează pe baza supraincărcării operatorului *cast* (operatorul de forțare a tipului). Aceasta se supraincarcă folosind o funcție membru nestatică care are un antet specific:

*nume\_clasa::operator nume\_tip\_predefinit()*

Lipsa parametrului rezultă din faptul că operatorul *cast* este unar.

Funcția care supraincarcă operatorul *cast* se apelează pentru un obiect care este o instanțiere a clasei *nume\_clasa* și rezultatul apelului este o dată de tip *nume\_tip\_predefinit* și a cărei valoare este definită prin execuția corpului funcției respective.

Operatorul *cast* obișnuit se aplică de către programator la un operand pentru a impune o anumită conversie dintr-un tip predefinit în altul tot predefinit.

Am văzut că, în acest scop, se pot utiliza două formate:

*(tip)expresie*

sau

*tip(expresie)*

Operatorul *cast* supraincărat pentru o clasă oarecare se aplică pentru o instanțiere a clasei respective fie explicit, fie implicit.

La aplicarea explicită a operatorului *cast* supraincărat se utilizează formate similare cu cele indicate mai sus:

*(nume\_tip\_predefinit)obiect*

sau

*nume\_tip\_predefinit(obiect)*

În lucrările [9] și [14] se indică o utilizare a supraîncărcării operatorului *cast* pentru folosirea intregilor pozitivi de valori mici.

Exemplul de mai jos este inspirat din aceste lucrări.

### Exemplu:

Tipul abstract *intmic* se implementează pentru a putea controla lucrul cu intregi situați în intervalul [0,63].

```
class intmic {
    char v; // valoarea poate fi păstrată pe un octet

    void atribuire(int i);
    /* - atribuie lui v valoarea lui i;
       - dacă i nu aparține intervalului [0,63], se da mesaj și se trunchiază valoarea lui
         i la ultimii 6 biti */

public:
    intmic(int i=0)
    /* - construiește obiectul intmic initializându-l cu valoarea lui i;
       - realizează conversia utilizator implicită din int în intmic */

    {atribuire(i);}

    operator int() // supraîncarcarea operatorului cast
    /* conversie din tipul intmic în tipul int */
    {
        return v;
    }

    intmic& operator = (int i)
    {
        atribuire(i);
        return *this;
    }
};
```

Funcția *atribuire* se definește ca mai jos:

```
void intmic::atribuire(int i)
{
    if(i < 0 || i > 63)
        printf("valoarea: %d nu aparține intervalului [0,63]\n",
               i);
    v=i&63;
}
```

Mai jos, se indică exemple de utilizare a tipului definit cu ajutorul clasei *intmic*.

```
intmic m1; // se instantiază obiectul m1 cu m1.v=0
intmic m2=12; // se instantiază obiectul m2 cu m2.v=12
intmic m3 = 123; // se dă mesaj de eroare
                  // se instantiază obiectul m3 cu m3.v=59
intmic m4=m1; // copiere bit_cu_bit
```

```
intmic m5=m2-2;
/* - m2 se convertește spre int aplicindu-se supraîncarcarea operatorului cast;
   - apoi diferența, se convertește din int în intmic și se copiază bit_cu_bit */

intmic m6=m2*10;
/* - m2 se convertește spre int;
   - se înmulțește cu 10 și se obține 120;
   - se convertește din int în intmic, se afișează mesaj și se atribuie lui m6 valoarea 56 */

int i=m2*10;
/* - m2 se convertește spre int;
   - se înmulțește cu 10 și se obține 120 care este de tip int;
   - se atribuie lui i valoarea 120. */

intmic m7;
m7=m2-2;
/* - m2 se convertește spre int aplicindu-se supraîncarcarea operatorului cast;
   - se face diferența m2-2 și se obține o date de tip int(10);
   - se aplică operatorul = supraîncarcat și se atribuie obiectului m7 valoarea diferenței. */
```

### Exerciții:

24.8 În exercițiul 10.31, s-a definit tipul utilizator *RATIONAL* cu ajutorul declarării:

```
typedef struct {
    double numarator;
    double numitor;
} RATIONAL;
```

În exercițiile 10.31, 10.32, 10.33, 10.34, 10.35 și 10.36 se definesc funcții care permit operații cu date de tip *RATIONAL*.

În exercițiul de față, se definește tipul abstract *rational* care permite operații cu numere analoge cu cele definite pentru tipul utilizator *RATIONAL*.

Datele de tip *RATIONAL* le numim *numere fracționare*. Valoarea unei fracții este egală cu valoarea raportului:

*numarator/numitor*

În mod analog, obiectele care sunt instanțieri ale clasei *rational* le numim fracții, iar valoarea fracției se definește prin același raport. În acest caz, valoarea unei fracții se obține ca rezultat al aplicării supraîncărcării operatorului *cast*. Aceasta realizează conversia obiectelor de tip *rational* în valori de tip *double*.

O supraîncarcare analogă a operatorului *cast* se dă și în cartea [18].

### FISIERUL BXXIV8.H

```
class rational {
    long numarator;
    long numitor;
```

```

void simplifica(); /* imparte numitorul si numitorul cu c.m.m.d.c. al lor */

public:
    rational (long num=0,long numit=1);
    /* constructor utilizat pentru initializarea obiectelor de tip rational si pentru
       conversia utilizator implicita a datelor de tip predefinit in obiecte de tip rational */

    int citrational(char *s);
    /* - afisaza sirul sprc care pointeaza s daca nu este vid; altfel afisaza text standard;
       - citeste componentele obiectului rational curent;
       - returneaza:
          0 - la intilnirea sfarsitului de fisier;
          1 - altfel. */

    void afisrational(char *f);
    /* - afisaza componentele obiectului rational curent;
       - se foloseste formatul sprc care pointeaza f sau un format standard daca
       acesta este vid. */

    friend rational operator +(rational r1,rational r2);
    // returneaza r1+r2

    friend rational operator -(rational r1,rational r2);
    // returneaza r1-r2

    friend rational operator *(rational r1,rational r2);
    // returneaza r1*r2

    friend rational operator /(rational r1,rational r2);
    // returneaza r1/r2

    void operator ++(); // incrementeaza obiectul curent

    void operator --(); // decrementeaza obiectul curent

    // prin r notam obiectul curent
    rational& operator += (rational r2); // r=r+r2
    rational& operator -= (rational r2); // r=r-r2
    rational& operator *= (rational r2); // r=r*r2
    rational& operator /= (rational r2); // r=r/r2
    friend rational operator - (rational r1);
    // returneaza -r1

    operator double();
    /* - supraincarca operatorul cast;
       - conversie din rational in double */

    friend int operator == (rational r1,rational r2);
    // returneaza 1 daca r1 == r2, 0 altfel

    friend int operator < (rational r1,rational r2);
    // returneaza 1 daca r1 < r2, 0 altfel

```

```

friend int operator <= (rational r1,rational r2);
// returneaza 1 daca r1 <= r2, 0 altfel

long retnumitor(); // returneaza numitorul obiectului curent

long retnumarator(); // returneaza numaratorul obiectului curent;
};

Fișierul de extensie CPP conține definițiile funcțiilor membru. Totodată se includ fișiere cu funcții obișnuite care sunt utilizate de către unele dintre funcțiile membru. Acestea sunt:

rcit_long      (exercițiul 24.5);
rcit_long_lim (exercițiul 24.6);
cmmdc         (exercițiul 10.29);
cmmmc         (exercițiul 10.30).

```

## FIŞIERUL BXXIV8

```

#ifndef __RCIT_LONG
#include "BXXIV5.CPP"           // rcit_long
#define __RCIT_LONG
#endif
#ifndef __RCIT_LONG_LIM
#include "BXXIV6.CPP"           // rcit_long_lim
#define __RCIT_LONG_LIM
#endif
#ifndef __CMMDC
#include "BX29.CPP"             // cmmdc
#define __CMMDC
#endif
#ifndef __CMMMC
#include "BX30.CPP"             // cmmmc
#define __CMMMC
#endif
#ifndef __RATIONAL_H
#include "BXXIV8.H"
#define __RATIONAL_H
#endif

void rational::simplifica()
/* imparte numaratorul si numitorul cu c.m.m.d.c. al lor */
{
    long c;

    if(numitor) c= cmmdc(numarator,numitor);
    else{
        printf("numitor nul\n");
        return;
    }
    numarator = numarator/c;
    numitor = numitor/c;
}

```

```

rational::rational(long num1, long num2)
/* constructor pentru initializare si conversie din tip predefinit in tipul rational */
{
    numerator=num1;
    if(num2) numitor = num2;
    else {
        printf("numitor nul; se forteaza la 1\n");
        numitor = 1;
    }
}

int rational::citirational(char *s)
/* - afisaza sirul spre care pointeaza s daca nu este vid; altfel afisaza un text standard;
 - citeste componentele obiectului rational curent;
 - returneaza:
    0 - la intilnirea sfirsitului de fisier;
    1 - altfel.
*/
{
    long ln;

    if(s && *s) { // sirul s nu este vid
        for(;;) { // citeste numeratorul
            printf(s);
            int i;
            if((i=scanf("%ld",&ln)) == EOF) return 0;
            if(i == 1) break;
            printf("nu s-a citit un intreg\n");
            fflush(stdin);
        }
        numerator = ln;
        for(;;) { // citeste numitorul
            int i;
            if((i=scanf("%ld",&ln))==EOF) return 0;
            if(i==1&&ln!=0) break;
            printf("nu s-a tastat un intreg nenul\n");
            fflush(stdin);
        }
        numitor = ln;
        fflush(stdin);
        return 1;
    } // sfirsit if sirul nu este vid

    // s pointeaza spre sirul vid
    if(rcit_long_lim("numerator=", -2147483648, 2147483647,
                      numerator)==0)
        return 0;
    do{
        if(rcit_long_lim("numitor=", -2147483648, 2147483647,
                         numitor)==0)
            return 0;
        if(numitor != 0) return 1;
        printf("numitor nul\n");
        printf("se reia citirea numitorului\n");
    }
}

```

```

    } while(1);

} // sfirsit citirational

inline void rational::afisrational(char *f)
/* - afisaza obiectul rational curent;
 - se foloseste formatul spre care pointeaza f sau un format standard daca f pointeaza
 spre sirul vid. */
{
    if(f && *f) // formatul nu este vid
        printf(f,numerator,numitor);
    else // se utilizeaza un format standard
        printf("numerator = %ld\t numitor = %ld\n",
               numerator,numitor);
}

rational operator+(rational r1,rational r2) // returneaza r1+r2
{
    rational r; // numerator=0, numitor=1

    if(r1.numerator == 0) return r2;
    if(r2.numerator == 0) return r1;
    long m = cmmmc(r1.numitor,r2.numitor);
    r.numerator=r1.numerator*(m/r1.numitor)+r2.numerator*(m/r2.numitor);
    r.numitor = m;
    r.simplifica();
    return r;
}

rational operator -(rational r1) // returneaza -r1
{
    rational r;

    r.numerator = -r1.numerator;
    r.numitor = r1.numitor;
    return r;
}

rational operator -(rational r1,rational r2) // returneaza r1-r2
{
    return r1 + -r2;
}

rational operator *(rational r1,rational r2) // returneaza r1*r2
{
    rational r;

    r.numerator = r1.numerator*r2.numerator;
    r.numitor = r1.numitor*r2.numitor;
    r.simplifica();
    return r;
}

```

```

rational operator / (rational r1,rational r2) // returncaza r1/r2
{
    rational r;
    r.numarator = r1.numarator*r2.numitor;
    r.numitor = r1.numitor*r2.numarator;
    if(r.numitor == 0){ //impartire cu zero
        printf("impartire cu zero\n");
        return 0;
    }
    r.simplifica();
    return r;
}

inline void rational::operator++() // incrementaza obiectul curent
{
    *this = *this+1;
}

inline void rational::operator--() // decrementaza obiectul curent
{
    *this = *this-1;
}

inline rational& rational::operator += (rational r2) // r=r+r2
{
    *this = *this+r2;
    return *this;
}

inline rational& rational::operator -= (rational r2) // r=r-r2
{
    *this = *this-r2;
    return *this;
}

inline rational& rational::operator *= (rational r2) // r=r*r2
{
    *this = *this*r2;
    return *this;
}

inline rational& rational::operator /= (rational r2) // r=r/r2
{
    *this = *this/r2;
    return *this;
}

rational::operator double()
// supraincarcarea operatorului cast
// conversie din rational in double
{
    if(numitor == 0){ // numitor nul
        printf("numitor nul\n");
}

```

```

        return 0.0;
    }
    return (double) numarator/numitor;
}

int operator == (rational r1,rational r2)
// returncaza 1 daca r1 == r2 si zero altfel
{
    if(r1.numarator == 0)
        if(r2.numarator == 0) return 1;
        else return 0;
    if(r2.numarator == 0)
        if(r1.numarator == 0) return 1;
        else return 0;

    rational t1 = r1;
    rational t2 = r2;
    t1.simplifica();
    t2.simplifica();
    return t1.numarator==t2.numarator &&
           t1.numitor==t2.numitor;
}

inline int operator < (rational r1,rational r2)
// returncaza 1 daca r1 < r2, 0 altfel
{
    // se utilizeaza explicit supraincarcarea operatorului cast
    return double(r1) < double(r2);
}

inline int operator <= (rational r1,rational r2)
// returncaza 1 daca r1 <= r2, 0 altfel
{
    return double(r1) <= double(r2);
}

inline long rational::retnumarator()
// returncaza numaratorul obiectului curent
{
    return numarator;
}

inline long rational::retnumitor()
// returncaza numitorul obiectului curent
{
    return numitor;
}

```

#### Observații:

1. Supraincarcarea operatorului `==` se poate realiza la fel ca și supraincarcarea operatorilor `<` și `<=`, adică utilizând supraincarcarea operatorului *cast*:

```
return double(r1) == double(r2);
```

Metoda utilizată mai sus este mai exactă, deoarece împărțirea numărătorului la numitorul fracției conduce adesea la un rezultat aproximativ (15 zecimale exacte).

## 2. O expresie de forma:

$r+d$

unde  $r$  este un obiect de tip rational și  $d$  un număr de un tip predefinit se evaluatează astfel:

- deoarece operatorul  $+$  este supraîncărcat pentru operanzi de tip rational, termenul  $d$  se convertește implicit spre un obiect rational care are numitorul egal cu 1;
- apoi se adună cele două obiecte rationale ( $r$  și cel obținut prin conversia *implicită*) apelindu-se funcția care supraîncarcă operatorul  $+$ .

## 3. Expresia:

`double(r)+d`

se evaluatează astfel:

- obiectul rational se convertește spre `double`;
- apoi se aplică operatorul  $+$ , folosind regula conversiilor implicate.

## 4. $x = r+d$

unde:

- $x$  și  $d$  sunt variabile de tip predefinit;
- $r$  este un obiect de tip rational.

## Expresia:

$r+d$

se evaluatează ca la punctul 2 de mai sus. Apoi, se aplică în mod implicit supraîncărcarea operatorului *cast* care convertește în `double` obiectul rational care este rezultatul evaluării expresiei  $r+d$ . Rezultatul de tip `double` se atribuie lui  $x$  direct dacă acesta este de tip `double` sau făcind o nouă conversie din tipul `double` spre tipul lui  $x$ .

## 24.9 În clasele primare se dă elevilor spre rezolvare exerciții care comportă evaluări de expresii ce conțin ca operanzi fracții și cele 4 operații aritmetice (adunare, scădere, înmulțire și împărțire).

Expresia de mai jos este un astfel de exercițiu. La scrierea ei se utilizează notațiile obișnuite din limbajul C, adică pentru înmulțire se utilizează semnul `"*"`, iar pentru împărțire semnul `"/"`. De asemenea, fracția se scrie sub forma liniară și se include între paranteze rotunde (numărător/numitor):

$$((7/15)+(14/15)+(2/9)*(31/3)-(12/11)*((8/3)-(7/4)))/(((3/7)-(1/4))/(3/28)-1)$$

Notind fracțiile cu  $f1, f2, \dots$  expresia de mai sus se transcrie astfel:

$$((f1+f2+f3)*f4-f5*(f6-f7))/((f8-f9)/f10-1)$$

unde:

$$\begin{aligned} f1 &= (7/15), f2 = (14/15), f3 = (2/9), f4 = (31/3), f5 = (12/11), \\ f6 &= (8/3), f7 = (7/4), f8 = (3/7), f9 = (1/4), f10 = (3/28). \end{aligned}$$

Programul de față evaluează expresia de mai sus și afișează rezultatul.

## PROGRAMUL BXXIV9

```
#ifndef __STDLIB_H
#include <stdio.h>
#define __STDLIB_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIV8.CPP"

main()
/* - evalueaza o expresie cu operanzi fractii;
   - afiseaza rezultatul evaluarii expresiei.
*/
{
    rational f1(7,15),f2(14,15),f3(2,9),f4(31,3),f5(12,11),
    f6(8,3),f7(7,4),f8(3,7),f9(1,4),f10(3,28);
    rational f;
    f=((f1+f2+f3)*f4-f5*(f6-f7))/((f8-f9)/f10-1);
    f.afisrational("");
}
```

## Observație:

Valoarea expresiei este egală cu 1064/45.

## 24.10 Să se scrie un program care rezolvă ecuații de gradul 1 cu coeficienți fraționari.

## PROGRAMUL BXXIV10

```
#ifndef __STDLIB_H
#include <stdio.h>
#define __STDLIB_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
```

```

#include "BXXIV8.CPP"

main()
/* rezolva ecuatii de forma
   a*x+b = 0
   unde a si b sunt obiecte de tip rational
*/
{
    rational a,b,x;

    for(;;) {
        if(a.citrationnal("") == 0) exit(0); //EOF
        if(b.citrationnal("") == 0) exit(1); //EOF
        if(a == 0 && b == 0){
            printf("ecuatie nedeterminata\n");
            continue;
        }
        if(a == 0){
            printf("ecuatie nu are solutie\n");
            continue;
        }
        x = -b/a;
        x.afisrationnal("x=%ld/(%ld)\n");
        // valoarea fractiei
        printf("x=%g\n", (double)x);
        printf("Actionati o tasta pentru a continua\n");
        getch();
    } // sfarsit for
}

```

24.11 Să se scrie un program care rezolvă ecuații de gradul 2 cu coeficienți fraționari.

Deoarece rădăcina pătrată din expresia:

$$b^2 - 4ac$$

în general este un număr real, calculele care urmează se vor face în flotantă dublă precizie.

### PROGRAMUL BXXIV11

```

#ifndef __STDLIB_H
#include <stdio.h>
#define __STDLIB_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include <math.h>

```

```

#include "bxxiv8.cpp"

main() /* rezolva ecuatii de gradul 2 cu coeficienti fractionari */
{
    rational a,b,c;

    for(;;){
        printf("coeficientul lui x patrat\n");
        if(a.citrationnal("") == 0) exit(0); //EOF
        printf("coeficientul lui x\n");
        if(b.citrationnal("") == 0) exit(1); //EOF
        printf("termenul liber\n");
        if(c.citrationnal("") == 0) exit(1); //EOF
        if(a==0 && b==0 && c==0){
            printf("ecuatie nedeterminata\n");
            continue;
        }
        if(a==0 & b==0){
            printf("ecuatie nu are solutie\n");
            continue;
        }
        if(a==0){ //ecuatie de grad 1
            rational x = -c/b;
            printf("ecuatie de gradul 1\n");
            x.afisrationnal("x=%ld/(%ld)\n");
            printf("Actionati o tasta pentru a continua\n");
            getch();
            continue;
        }

        // a != 0; ecuatie de gradul 2
        double delta = b*b-4*a*c;
        double doia = 2*a;
        double minusb = -b;
        if(delta > 0) { //radacini reale
            delta = sqrt(delta);
            double x = (minusb + delta)/doia;
            printf("x1 = %g\n", x);
            x=(minusb - delta)/doia;
            printf("x2 = %g\n", x);
        }
        else
            if(delta==0){ //radacina dubla
                double x = minusb/doia;
                printf("radacina dubla\n");
                printf("x=%g\n", x);
            }
            else{ //radacini complexe
                delta = sqrt(-delta);
                double real = minusb/doia;
                double imag = delta/doia;
                printf("x1=%g+i*(%g)\n", real, imag);
                printf("x2=%g-i*(%g)\n", real, imag);
            }
    }
}

```

```

    printf("Actionati o tasta pentru a continua\n");
    getch();
}

```

#### Observație:

Declarația:

```
double delta=b*b-4*a*c;
```

evaluează expresia:

```
b*b-4*a*c
```

apelind funcțiile care suprareîncarcă operațiile pe care le conține, pentru tipul rational, apoi se apeleză funcția care suprareîncarcă operatorul *cast* pentru a converti valoarea expresiei din *rational* în double.

### 24.3. Conversia dintr-un tip abstract într-un alt tip abstract

Pentru conversia obiectelor dintr-un tip abstract într-un alt tip abstract se pot folosi atât conversiile realizate cu ajutorul constructorilor, cit și cele realizate prin suprareîncarcarea operatorului *cast*.

De exemplu, fie două tipuri abstracte *tip1* și *tip2*. Pentru a realiza conversia obiectelor de tip *tip1* în obiecte de tip *tip2* folosind constructorul clasei *tip2*, se procedează la definirea unui constructor pentru clasa *tip2* care să aibă ca parametru un obiect de tip *tip1*. Deci, clasa *tip2* are un constructor de antet:

```
tip2(tip1 ob)
```

Acest constructor trebuie să aibă acces la datele membru ale clasei *tip1*, date care, de obicei, sunt protejate. De aceea, constructorul respectiv se declară ca funcție prieten a clasei *tip1*.

#### Exemplu:

Alături de tipul abstract *grad*, definit în exercițiul 24.4, considerăm tipul abstract *radian* pentru lucrul cu măsuri de unghiuri exprimate în radiani.

Clasa *radian* se definește în aşa fel încit să se poată converti cu ajutorul constructorului măsurile unghiulare din grade sexagesimale în radiani. În acest scop putem proceda ca mai jos:

```

class radian {
    double rad;
public:
    radian(double d = 0)
        // conversie din tip predefinit în tipul radian
    {
        rad = d;
    }
}

```

```

radian(grad);
// constructor care realizează conversie din tipul grad în tipul radian

```

}

Constructorul:

```
radian(grad)
```

este o funcție prieten pentru clasa *grad*, deoarece aceasta trebuie să aibă acces la data membru *totsec* a clasei *grad* (vezi exercițiul 24.4). Pentru a realiza acest lucru, în definiția clasei *grad* se adaugă declarația:

```
friend radian::radian(grad);
```

Constructorul care definește conversia din tipul *grad* în tipul *radian* se definește simplu, ca mai jos:

```

radian::radian(grad g)
// conversie din grade sexagesimale în radiani
{
    rad=g.totsec*PI/648000;
}

```

unde PI are valoarea 3.14159265358979

Conversia din grade sexagesimale în radiani se poate defini și cu ajutorul suprareîncărării operatorului *cast*.

Pentru a realiza conversia obiectelor din tipul abstract *tip1* în tipul abstract *tip2*, cu ajutorul unei funcții membru nestatică care suprareîncarcă operatorul *cast*, se folosește antetul:

```
tip1::operator tip2()
```

Această funcție este o funcție membru nestatică a clasei *tip1* și o funcție prieten pentru clasa *tip2*.

Pentru a realiza conversia din grade sexagesimale în radiani, se procedează ca mai jos:

- se definește funcția membru nestatică a clasei *grad*, de antet:  
`grad::operator radian()`
- iar în clasa *radian* se declară funcția respectivă ca funcție prieten.

```

class radian;
class grad {
    long totsec;
    long sgrad;
    int smin;
    int ssec;
    void grminsec();
public:
    grad(long sec=0) {
        totsec = sec;
        grminsec();
    }
}

```

```

    }
grad(long g,int m,int s)
{
    totsec=g*3600+m*60+s;
    grminsec();
}
...
operator radian();
...
};

class radian {
    double rad;
public:
    radian(double d=0)
    {
        rad = d;
    }
...
friend grad::operator radian();
...
};

```

În acest caz, funcția care suprareîncarcă operatorul *cast* se definește astfel:

```

grad::operator radian()
{
    radian r = totsec*PI/648000;
    return r;
}

```

Declarația:

```
radian r = totsec*PI/648000;
```

se tratează astfel:

- data membru *totsec* este componentă a obiectului curent, deci i se aplică pointerul *this* în mod implicit:  
`this -> totsec;`
- se evaluatează expresia din dreapta semnului de atribuire și se obține o valoare de tip *double* care reprezintă măsura în radiani a unghiului a cărui măsură în secunde sexagesimale este egală cu valoarea lui  
`this-> totsec.`

Pentru a atribui această valoare obiectului *r* de tip *radian*, se realizează conversia utilizator implicită care convertește o dată din tip *double* într-un obiect de tip *radian*. Aceasta se realizează apelindu-se în mod implicit constructorul clasei *radian* care are un parametru de tip *double*. Obiectul obținut în acest fel se atribuie lui *r* printr-o copiere bit\_cu\_bit.

Mai sus, s-a definit conversia din tipul *grad* în tipul *radian* prin două metode:

- una folosește constructorul clasei *radian* care are un parametru de tip *grad* și acesta este o funcție prieten pentru clasa *grad*;

- cealaltă folosește o funcție membru nestatică, care suprareîncarcă operatorul *cast* pentru clasa *grad* și care este o funcție prieten pentru clasa *radian*.

Utilizatorul va trebui să opteze pentru una din metode, deoarece prezența ambelor metode conduce la ambiguitate și în acest caz, compilatorul semnalizează eroare în loc de a aplica una din metodele de conversie.

Uneori este nevoie să se aplique mai multe conversii implicate pentru a evalua un operand.

Mentionăm că, în acest caz, compilatorul admite o singură conversie implicită definită de utilizator. Prin conversie definită de utilizator înțelegem una din următoarele conversii:

- din tip predefinit în tip abstract;
- din tip abstract în tip predefinit;
- din tip abstract în tip abstract.

#### Exemplu:

Presupunem că, clasa *radian* are o funcție membru care suprareîncarcă operatorul *cast* pentru a converti obiectele de tip *radian* în date de tip *double*. O astfel de funcție se poate defini astfel:

```
radian::operator double()
{
    return rad;
}
```

Fie instanțierea:

```
radian r;
...
```

Apelul:

```
sin(r)
```

este corect și se realizează după conversia prealabilă a obiectului *r*, de tip *radian*, în tipul *double* (funcția *sin* are prototipul *double sin(double)*).

Fie instanțierea:

```
grad g;
...
```

Apelul:

```
sin(g)
```

este eronat, deoarece comportă mai mult de o conversie implicită definită de utilizator:

- conversia lui *g* din *grad* în *radian*;
- conversia din *radian* în *double*.

În schimb apelul:

```
sin(radian(g))
```

este corect, deoarece aici se aplică o singură conversie utilizator *implicită*:

- conversia obiectului *radian(g)* de tip *radian* într-o dată de tip *double*.

#### Observație:

La supraincărcarea operatorului *cast* cu ajutorul funcției membru:

```
operator radian()
```

a clasei *grad* nu este necesar ca funcția să fie declarată ca funcție prieten a clasei *radian*. Aceasta deoarece la realizarea conversiei în radiani nu este necesar să se facă acces la data membru protejată a clasei *radian*. În general este nevoie de acest lucru și atunci funcția care supraincarcă operatorul *cast* va fi o funcție prieten a clasei spre care se face conversia.

#### Exerciții:

24.12 Să se implementeze tipurile abstracte *celsius* și *fahrenheit* care permit utilizarea datelor de tip *celsius* și respectiv *fahrenheit* la exprimarea măsurii temperaturii și la efectuarea de calcule. De asemenea, se definesc conversii în ambele sensuri: din grade Celsius în grade Fahrenheit și invers.

O formulă simplă de conversie din grade Fahrenheit în grade Celsius este următoarea:

$$c = (5/9)*(f-32)$$

unde:

- f* - Reprezintă măsura temperaturii în grade Fahrenheit.  
*c* - Reprezintă măsura temperaturii în grade Celsius.

Relația care asigură conversia inversă este următoarea:

$$f = (9/5)*c+32 \text{ sau } 1.8*c+32$$

Clasa care implementează tipul *fahrenheit* conține funcții pentru supraincărcarea operatorilor aritmetici după cum urmează:

- adunarea obiectelor de tip *fahrenheit*;
- scăderea obiectelor de tip *fahrenheit*;
- inmulțirea unui obiect de tip *fahrenheit* cu o dată numerică de tip predefinit;
- împărțirea unui obiect de tip *fahrenheit* cu o dată numerică de tip predefinit;
- negativarea unui obiect de tip *fahrenheit*.

#### FIŞIERUL BXXIV12.H

```
class celsius;
class fahrenheit {
    double fah;
public:
    fahrenheit(double f=0);
    /* constructor utilizat pentru initializare și conversie din tip predefinit în
```

```
tipul fahrenheit */

fahrenheit(celsius);
/* constructor utilizat pentru conversie din tipul celsius în tipul fahrenheit */

double retfahrenheit();
/* returneaza componenta fah a obiectului curent */

operator celsius();
/* conversie din tipul fahrenheit în tipul celsius */

fahrenheit operator -(); // negativare

friend fahrenheit operator +(fahrenheit f1,fahrenheit f2);
// returneaza f1+f2

friend fahrenheit operator -(fahrenheit f1,fahrenheit f2);
// returneaza f1-f2

fahrenheit operator *(double d);
// returneaza f*d, unde f este obiectul curent

friend fahrenheit operator *(double d,fahrenheit f2);
// returneaza d*f2

fahrenheit operator /(double d);
// returneaza f/d, unde f este obiectul curent
};

class celsius {
    double cel;
public:
    celsius(double d=0);
    /* constructor utilizat pentru initializare și conversie din double în celsius */

    friend fahrenheit::fahrenheit(celsius);
    /* constructorul clasei fahrenheit care face conversie din celsius în
     * fahrenheit trebuie să aibă acces la data membru protejata cel */

    double retcelsius();
    /* returneaza componenta cel a obiectului curent */
};

Mai jos, se definesc funcțiile membru și prieten ale claselor fahrenheit și celsius.
```

#### FIŞIERUL BXXIV12

```
#ifndef __FAHCEL_H
#include "BXXIV12.H"
#define __FAHCEL_H
#endif
```

```

inline fahrenheit::fahrenheit (double d)
/* se utilizeaza la initializarea obiectelor de tip fahrenheit si la conversia din double
   in fahrenheit */
{
    fah = d;
}

inline fahrenheit::fahrenheit(celsius c)
/* se utilizeaza la conversia din celsius in fahrenheit */
{
    fah = c.cel*1.8+32;
}

inline double fahrenheit::retfahrenheit()
/* - returnaza fah pentru obiectul curent;
   - se utilizeaza in locul conversiei din fahrenheit in double */
{
    return fah;
}

fahrenheit::operator celsius() //conversie din fahrenheit in celsius
{
    celsius c=(5.0/9.0)*(fah-32);
    return c;
}

fahrenheit fahrenheit::operator -() //negativare
{
    fahrenheit f = *this;
    f.fah = -f.fah;
    return f;
}

fahrenheit operator +(fahrenheit f1,fahrenheit f2) //returnaza f1+f2
{
    fahrenheit f;
    f.fah = f1.fah +f2.fah;
    return f;
}

fahrenheit operator -(fahrenheit f1,fahrenheit f2) //returnaza f1-f2
{
    fahrenheit f = f1 + -f2;
    return f;
}

fahrenheit fahrenheit::operator *(double d)
//returnaza obiectul curent inmultit cu d
{
    fahrenheit f;
    f.fah = fah*d;
    return f;
}

```

```

}

fahrenheit operator *(double d,fahrenheit f2) //returnaza d*f2
{
    fahrenheit f=f2*d;
    return f;
}

fahrenheit fahrenheit::operator /(double d)
//returnaza obiectul curent impartit la d
{
    fahrenheit f;
    f.fah = fah/d;
    return f;
}

celsius::celsius(double c)
/* constructor utilizat la initializare si la conversia din double in celsius */
{
    cel = c;
}

inline double celsius::retcelius() /* returnaza cel pentru obiectul curent */
{
    return cel;
}

```

#### Observații:

1. Expresii de forma:  
 $f1*f2, f1/f2$   
 unde:  
 $f1$  și  $f2$  - sunt obiecte de tip *fahrenheit*, nu sunt admise.
2. Dacă se supraincarcă operatorul *cast* aşa încit să permită conversia obiectelor de tip *fahrenheit* în date de tip *double*, atunci expresia  $f1/f2$  este admisă și pentru cazul în care  $f1$  și  $f2$  sunt obiecte de tip *fahrenheit*. De asemenea, supraincărcarea operatorului *cast* în acest fel face ca expresia:  
 $f1*f2$   
 să fie ambiguă pentru cazul cind  $f1$  și  $f2$  sunt de tip *fahrenheit*. Ambiguitatea rezultă din faptul că operația de înmulțire este supraincărcată cu 2 funcții care au prototipurile:  
 $fahrenheit operator*(double d)$   
 și  
 $fahrenheit operator*(double d, fahrenheit f2)$   
 Prima, se aplică pentru o expresie de forma:  
 $f*d$   
 iar cea de a doua, pentru o expresie de forma:  
 $d*f$

unde:

- este de tipul *fahrenheit*;
- este de tip *double*.

În condițiile în care operatorul *cast* este supraincarcat pentru a realiza conversia din *fahrenheit* în *double*, ambele funcții care supraincarcă operatorul se pot aplica la evaluarea expresiei  $f1*f2$ :

- se convertește *f2* în *double* și se aplică prima funcție care supraincarcă operatorul \*;
- sau
- se convertește *f1* în *double* și se aplică cea de a doua funcție care supraincarcă operatorul \*.

Din cauza ambiguității, compilatorul semnalează o eroare la întlnirea unei expresii de forma  $f1*f2$ .

3. Se pot utiliza expresii de forma:

$c1+c2, c1-c2, c1*d, d*c1, c1/d$

unde:

- c1, c2* - sint obiecte de tip *celsius*;
- d* - este o dată de tip *double*.

Operanții de tip *celsius* se convertesc în obiecte de tip *fahrenheit* și apoi se realizează evaluarea expresiei folosind supraincărările operatorilor respective pentru obiecte *fahrenheit*.

4. Pentru a interzice utilizarea expresiilor de forma  $f1*f2$  sau  $f1/f2$ ,

unde:

*f1* și *f2* sint obiecte de tip *fahrenheit*,

nu s-a supraincarcat operatorul *cast* pentru conversia obiectelor de tip *fahrenheit* în date de tip *double*.

Pentru a avea acces la componenta protejată *fah*, în lipsa supraincărării operatorului *cast*, s-a definit funcția membru *retfahrenheit*.

- 24.13 Să se scrie un program care afișează o tabelă pentru conversia gradelor *Celsius* în grade *Fahrenheit*.

Tabelarea se face din grad în grad, limitele intervalului în care se face tabelarea sint 2 intregi de tip *int* care se citesc de la intrarea standard.

### PROGRAMUL BXXIV13

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "BXX4.CPP" // rcit_int
#include "BXX5.CPP" // rcit_int_lim
#include "BXXIV12.CPP" // FAHCEL

main()
/* - citeste pe m si n de tip int;
```

- afiseaza o tabela de conversie a gradelor Celsius in grade Fahrenheit;
- gradele Celsius variază, cu pasul 1, in intervalul [m,n].

```
/*
{
    int m,n;

    // citeste limitele m si n
    // limita inferioara
    if(rcit_int_lim("limita inferioara m = ", -32768, 32767,m)==0)
        exit(0);
    if(rcit_int_lim("limita superioara n = ",m,32767,n)==0)
        printf("s-a tasta EOF\n");
        exit(1);
}

// Afiseaza antet
printf("\n\n\t Conversie Celsius-Fahrenheit\n");
printf("\n Grade Celsius\t Grade Fahrenheit\n");
for(int i=m;i <= n;i++){
    celsius c(i);
    fahrenheit f=c; // conversie din Celsius in Fahrenheit
    printf("%13d\t %16g\n",i,f.retfahrenheit());
    if((i-m+1)%23 == 0) {
        printf("Actionati o tasta pentru a continua\n");
        getch();
    }
}
```

- 24.14 Să se scrie un program care citește un sir de intregi ce reprezintă măsurile temperaturii în grade Celsius, afișează măsurile respective în grade Fahrenheit, precum și media lor în grade Celsius și Fahrenheit. Sirul de intregi se termină cu sfîrșitul de fișier.

### PROGRAMUL BXXIV14

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "BXXIV5.CPP" // rcit_long
#include "BXXIV6.CPP" // rcit_long_lim
#include "BXXIV12.CPP" // FAHCEL

main()
/* - citeste un sir de intregi ce reprezinta masuri ale temperaturii in grade Celsius;
   - afiseaza masurile respective in grade Fahrenheit, precum si media lor in grade Celsius si Fahrenheit
*/
{
    int n=0;
    long t;
    fahrenheit sum; // sum.fah = 0
```

```

for(;;) {
    if(rcit_long_lim("temperatura in grade Celsius =",
        -2147483648,2147483647,t)==0)
        break; //EOF
    celsius c(t);
    sum = sum + c;
    fahrenheit f=c; //conversie Celsius in Fahrenheit
    printf("temperatura in grade Fahrenheit =%g\n",
        f.retfaarenheit());
    n++;
}
if(n){ // S-au citit temperaturi
    sum=sum/n; // calculeaza media
    celsius cmedia=sum; // conversia mediei in grade celsius
    printf("\nMedia in grade Celsius = %g\n",
        cmedia.retcelsius());
    printf("\nMedia in grade Fahrenheit = %g\n",
        sum.retfaarenheit());
}
else
    printf("Nu s-a citit nimic\n");
}

```

- 24.15 Să se definișă două tipuri abstracte *dc* și *dczi* pentru reprezentarea datelor calendaristice. Tipul abstract *dc* are trei date membru care definesc ziua, luna și anul calendaristic. O dată calendaristică validă este în intervalul:

1 ianuarie 1600 - 31 decembrie 4900

Tipul abstract *dczi* are o singură dată membru care definește numărul de zile socotite începînd cu 1 ianuarie 1600.

Data de 1 ianuarie 1600 corespunde cu valoarea 1 pentru data membru a tipului *dczi* (originea datelor calendaristice este 31 decembrie 1599).

Tipul abstract *dczi* are funcții membru care suprareîncarcă operațiile de adunare, scădere precum și operațiile de comparație și egalitate. Se definește conversia din tipul *dczi* în tipul *dc* și invers.

Pentru a evita suma a două obiecte de tip *dczi* nu se definește funcția care suprareîncarcă operatorul *cast* pentru conversia din *dczi* într-un tip predefinit. În locul ei, se definește o funcție membru care returnează numărul de zile care corespund unui obiect de tip *dczi*.

Nu se pot aduna două obiecte de tip *dczi*, ci numai o dată predefinită se poate aduna cu un obiect de tip *dczi*. Data de tip predefinit se consideră că reprezintă un număr de zile care se adună la data calendaristică pe care o definește operandul *dczi*.

Diferența a două obiecte de tip *dczi* reprezintă numărul de zile dintre cele două date calendaristice definite de cei doi operanzi de tip *dczi*. Se poate scădea dintr-un obiect de tip *dczi* o dată de tip predefinit care reprezintă un număr de zile.

Se pot compara numai obiecte de tip *dczi*.

Tipul *dc* are funcții membru pentru citirea și afișarea obiectelor de tip *dc*. Se poate face acces la datele membru ale unui obiect de tip *dc* cu ajutorul unor funcții membru.

## FIŞIERUL BXXIV15.H

```

class dc;
class dczi {
    long totzile;
public:
    dczi();
    /* constructor implicit pentru initializarea obiectelor cu
       totzile = 0 */

    dczi(dc);
    /* constructor pentru conversie din dc in dczi */

    long rettotzile();
    /* returnaza totzile pentru obiectul curent */

    // cu z se noteaza obiectul curent

    dczi operator +(long n); // returneaza z+n

    friend dczi operator +(long n,dczi z2); // returneaza n+z2

    long operator -(dczi z2); // returneaza z-z2
    dczi operator -(long n);
    // returneaza z-n, unde z este obiectul curent

    int operator == (dczi z2); // returneaza 1 daca z == z2 si zero altfel
    int operator < (dczi z2); // returneaza 1 daca z < z2 si zero altfel
    int operator <= (dczi z2); // returneaza 1 daca z <= z2 si zero altfel
};

class dc {
    int zi;
    int luna;
    int an;
public:
    dc(int z=0,int l=0,int a=0);
    /* constructor pentru initializarea obiectelor de tip dc */

    dc(dczi); // constructor pentru conversia din dczi in dc

    int retzi(); // returnaza zi pentru obiectul curent

    int retrluna(); // returnaza luna pentru obiectul curent
    int retan(); // returnaza an pentru obiectul curent
}

```

```

int citdc(char *text);
// citeste o data calendaristica dupa afisarea sirului spre care pointeaza text

void afisdc(char *f);
/* - afisaza data calendaristica pentru obiectul curent;
- se foloseste formatul f daca acesta nu pointeaza spre sirul vid;
altfel se utilizeaza un format standard. */

friend dczi::dczi(dc);
// constructor pentru conversii din tipul dc in tipul dczi
};

Funcțiile membru se definesc in fișierul de mai jos, de extensie cpp. Tot in acest fișier se definesc două funcții obișnuite care se apelează pentru a verifica datele calendaristice definite cu ajutorul tipurilor dczi și dc.

```

Intervalul 1 ianuarie 1600 - 31 decembrie 4900 conține 1205666 zile incluzind și zilele limită ale intervalului.  
Într-adevăr, intervalul respectiv conține:

$$4901-1600=3301 \text{ ani};$$

Dacă toți anii ar avea 365 de zile, atunci numărul de zile ar fi:

$$3301*365=1204865 \text{ zile}$$

Pentru a afla anii multiplii de 4, calculăm cîtul:

$$3301/4+1=826$$

Se adună 1 deoarece anul 1600, de început, este multiplu de 4.

Dacă toți anii multipli de 4 ar fi ani bisecți, ar însemna că la numărul de zile determinat mai sus s-ar adăuga 826 de zile.

Există  $3301/100+1=34$  ani care sunt multipli de 100. Se adună 1 deoarece anul 1600 este multiplu de 100. Dintre aceștia numai  $3301/400+1=9$  sunt bisecți (1600, 2000, 2400, 2800, 3200, 3600, 4000, 4400 și 4800). Rezultă că  $34-9=25$  ani multipli de 4 sunt și anii bisecți din cei multipli de 4. Deci se obțin:

$$826-25=801 \text{ zile}$$

Care se adaugă din cauza anilor bisecți.

În total, în intervalul respectiv sunt  $1204865+801=1205666$  zile.

## FIŞIERUL BXXIV15

```

#define MAXZILE 1205666L
#ifndef __DCDCZI_H
#include "BXXIV15.H"
#define __DCDCZI_H
#endif

static int tnrz[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};

static int v_calendzile(long tz)
/* - returneaza 1 daca tz este nenegativ si nu depaseste totalul de zile

```

```

din intervalul 1 ianuarie 1600 - 31 decembrie 4900;
- in caz contrar, se afiseaza un mesaj de eroare si returneaza valoarea zero. */

if(tz <= MAXZILE && tz >= 0) return 1;
printf("Prea multe zile sau numar negativ de zile=%ld\n",tz);
return 0;
}

static int v_calend(int z,int l,int a)
/* - returneaza 1 daca data calendaristica definita de parametri se afla in
intervalul 1 ianuarie 1600 - 31 decembrie 4900;
- in caz contrar, afiseaza un mesaj de eroare si returneaza valoarea zero */

if(a==0 && l==0 && z==0) return 0;
if(a < 1600 || a > 4900){
    printf("anul = %d eronat\n",a);
    return 0;
}
if(l < 1 || l > 12){
    printf("luna=%d eronata\n",l);
    return 0;
}
if(z < 1 || z > tnrz[l] + (l==2 && (a%4 == 0 && a%100 ||
a%400 == 0))){
    printf("ziua=%d eronata\n",z);
    return 0;
}
// data calendaristica valida
return 1;
}

inline dczi::dczi()
/* constructor implicit utilizat la initializarea obiectelor cu zero */
{
    totzile = 0;
}

dczi::dczi(dc d)
/* constructor pentru conversia din dc in dczi */
{
    int ancrt;
    int lunacrt;

    totzile = 0;
    if(v_calend(d.zi,d.luna,d.an)){
        for(ancrt=1600;ancrt < d.an;ancrt++)
            totzile += 365+(ancrt%4 == 0 &&
                           ancrt%100 || ancrt%400 == 0);
        for(lunacrt=1;lunacrt < d.luna;lunacrt++)
            totzile += tnrz[lunacrt]+(lunacrt == 2 &&
                           (ancrt%4==0 && ancrt%100 || ancrt%400==0));
        totzile += d.zi;
    }
}

```

```

inline long dczi::rettotzile() // returneaza totzile pentru obiectul curent
{
    return totzile;
}

dczi dczi::operator +(long n) // returneaza z+n
{
    dczi r;
    r.totzile = totzile+n;
    if(v_calendzile(r.totzile) == 0) r.totzile = 0;
    return r;
}

dczi operator +(long n,dczi z2) // returneaza n+z2
{
    return z2+n;
}

inline long dczi::operator -(dczi z2) // returneaza z-z2
{
    return totzile - z2.totzile;
}

inline dczi dczi::operator -(long n) // returneaza z-n
{
    return *this + -n;
}

inline int dczi::operator ==(dczi z2)
// returneaza 1 daca z==z2 si zero altfel
{
    return totzile == z2.totzile;
}

inline int dczi::operator < (dczi z2)
// returneaza 1 daca z<z2 si zero altfel
{
    return totzile < z2.totzile;
}

inline int dczi::operator <= (dczi z2)
// returneaza 1 daca z<=z2 si zero altfel
{
    return totzile <= z2.totzile;
}

dc::dc(int z,int l,int a)
// constructor pentru initializarea obiectelor de tip dc
{
    if(v_calend(z,l,a) == 0){
        z = 0; l = 0; a = 0;
    }
    zi = z; luna = l; an = a;
}

```

```

}

dc::dc(dczi tz) // constructor pentru conversie din dczi in dc
{
    zi = luna = an = 0;
    long t = tz.rettotzile();
    if(v_calendzile(t)){
        for(an=1600;t > 365+(an%4==0 && an%100 || an%400==0);an++)
            t -= (365+(an%4==0 && an%100 || an%400==0));
        for(luna=1;t > tnrz[luna]+(luna==2 &&
            (an%4==0 && an%100 || an%400==0));luna++)
            t -= (tnrz[luna]+(luna==2 &&
            (an%4==0 && an%100 || an%400==0)));
        zi = t;
    }
}

inline int dc::retzi() // returneaza ziua obiectului curent
{
    return zi;
}

inline int dc::retluna() // returneaza luna obiectului curent
{
    return luna;
}

inline int dc::retan() // returneaza anul obiectului curent
{
    return an;
}

dc::citdc(char *text)
/*- citeste o data calendaristica dupa afisarea sirului spre care pointeaza
text, daca acesta nu este vid;
- returneaza:
    0 - la intinirea sfarsitului de fisier;
    1 - altfel.*/
{
    for(;){
        if(text && *text) printf(text);
        if(rcit_int_lim("ziua=",1,31,zi)==0) return 0; // EOF
        if(rcit_int_lim("luna=",1,12,luna)==0) return 0; // EOF
        if(rcit_int_lim("an=",1600,4900,an)==0) return 0; // EOF
        if(v_calend(zi,luna,an)) break; // data calendaristica valida
            printf("data calendaristica eronata\n");
            printf("se reia citirea datei calendaristice\n");
    }
    return 1;
}

void dc::afisdc(char *f)
/*- afiseaza data calendaristica pentru obiectul curent;

```

```

    se utilizeaza formatul spre care pointeaza f daca acesta nu este vid,
    in caz contrar, un format standard. */
{
    if(f && *f) printf(f, zi, luna, an);
    else printf("ziua=%d\t luna=%d\t anul=%d\n", zi, luna, an);
}

```

24.16 Să se scrie un program care citește date calendaristice și afișează data calendaristică a zilei următoare pentru fiecare dată citită.

### PROGRAMUL BXXIV16

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "BXX4.CPP"      // rcit_int
#include "BXX5.CPP"      // rcit_int_lim
#include "BXXIV15.CPP"   // dczi_dc

main()
/* citeste date calendaristice si afiseaza data calendaristica a zilei urmatoare pentru
   fiecare data citita */
{
    dc d_crt;

    for(;;){
        if(d_crt.citdc(0) == 0) exit(0); //EOF
        dczi d_urz_zile = dczi(d_crt)+1;
        dc d_urz = d_urz_zile;
        printf("ziua urmatoare\n");
        d_urz.afisdc(0);
        printf("Actionati o tasta pentru a continua\n");
        getch();
    }
}

```

#### Observație:

Expresia:

`d_crt+1`

nu este admisă de compilator. Aceasta datorită modului în care a fost supraincarcat operatorul "+". Într-adevăr, operatorul + a fost supraincarcat pentru obiecte de tip `dczi` folosind două funcții de prototip:

`dczi dczi::operator+(long n);`

și

`friend dczi operator+(long n,dczi z2);`

Prima funcție este o funcție membru nestatică și se poate aplica numai dacă obiectul din stanga operatorului + este de tip `dczi`. Ori `d_crt` este un obiect de tip `dc`. Deși există un constructor al clasei `dczi` care convertește obiecte de tip `dc` în

obiecte de tip `dczi`, acesta nu se aplică implicit în această situație și de aceea este nevoie de aplicarea lui explicită. Cea de a doua funcție care supraincarcă operatorul + are un prototip care nu corespunde acestui expresii. Într-adevăr, această funcție se poate apela dacă primul operand este de tip `long` sau se poate converti spre tipul `long`.

Nu există însă o conversie utilizator din tipul `d_crt` spre tipul `long` sau alt tip predefinit.

Dacă ar exista o astfel de conversie, atunci s-ar putea apela funcția prieten deoarece conversia lui 1 spre tipul `dczi` se poate realiza implicit aplicindu-se constructorul corespunzător al clasei `dczi`.

Compilatorul admite expresia:

`1+d_crt`

care are aceeași valoare cu expresia de mai sus. În acest caz, se aplică funcția prieten indicată mai sus, termenul `d_crt` convertindu-se din `dc` în `dczi`.

24.17 Să se scrie un program care citește două date calendaristice și afișează diferența, în număr de zile, dintre cea de a doua data calendaristică și prima.

### PROGRAMUL BXXIV17

```

#include <stdio.h>
#include <stdlib.h>
#include "BXX4.CPP"      // rcit_int
#include "BXX5.CPP"      // rcit_int_lim
#include "BXXIV15.CPP"   // dczi_dc

main()
/* citeste doua date calendaristice si afiseaza diferența in numar
   de zile dintre cea de a doua data calendaristica si prima data calendaristica */
{
    dc d1;
    dc d2;

    if(d1.citdc(0) == 0){
        printf("s-a intilnit EOF\n");
        exit(1);
    }

    if(d2.citdc(0) == 0){
        printf("s-a intilnit EOF\n");
        exit(1);
    }
    printf("data a doua-prima=%ld\n", dczi(d2)-d1);
}

```

## 25. MODIFICATORUL CONST PENTRU OBIECTE ȘI FUNCȚII MEMBRU

În capitolul 20 s-au indicat facilitățile oferite de modificatorul *const* la declararea datelor constante.

În principiu, modificatorul *const* prezent într-o declarație, protejează date din declarația respectivă deoarece ei nu își poate schimba valoarea la execuție în mod "direct". De aceea, ea se comportă la fel ca o constantă.

Modificatorul *const* se poate aplica și la declararea obiectelor pentru a instanța obiecte constante sau pointeri spre obiecte constante.

### Exemplu:

Fie clasa *sir*, definită ca mai jos:

```
class sir {
    char *psir;
    int lungime;
public:
    sir(char *p);
    /* constructor pentru initializarea obiectului cu un sir de caractere */
    sir();           // constructor implicit
    sir(const sir&); // constructor de copiere
    sir& operator = (const sir&);
    // supraincarcarea operatorului =
    void afsir(char *f);
    /* afiseaza sirul curent folosind formatul definit de f */
};
```

Parametrul constructorului de copiere este o referință la un obiect. Acest obiect constituie obiectul sursă care se copiază în obiectul curent.

Constructorul de copiere nu are voie să modifice obiectul sursă. Acest lucru este asigurat folosind modificatorul *const* la declararea parametruului formal.

Modificatorul *const* are aceeași utilizare și în cazul funcției care supraincarcă operatorul *"=*". Într-adevăr, obiectul parametru reprezintă partea dreaptă a operatorului de atribuire. Funcția respectivă nu are voie să modifice operatorul din dreapta semnului de atribuire și de aceea, parametrul corespunzător lui se declară folosind modificatorul *const*.

Funcțiile membru ale clasei *sir* se pot defini la fel ca în exercițiul 23.14.

Fie instanțierile:

```
const sir s1("C++ este un C mai bun");
const sir s2=s1;
sir s3;
```

La prima instanțiere se aplică constructorul care are ca parametru un pointer spre un sir de caractere (char \*). La instanțierea a două se aplică constructorul de copiere. La instanțierea a treia se aplică constructorul implicit.

O atribuire de formă:

```
s3=s1;
```

este corectă. La realizarea ei se apelează funcția care supraincarcă operatorul de atribuire. În schimb, la o atribuire de formă:

```
s1=s3;
```

compilatorul afișează un avertisment deoarece *s1* a fost declarat cu ajutorul modificatorului *const*.

Modificatorul *const* se poate utiliza în antetul unei funcții în două moduri.

O primă utilizare a modificatorului *const* este aceea de a proteja obiectul returnat de funcție. În acest caz, modificatorul *const* precede tipul din antetul sau prototipul funcției respective:

(1) ... **const tip nume\_clasa::nume\_functie(...)**

Cea de a doua utilizare a modificatorului *const* este aceea când acesta se află în antetul sau prototipul funcției după paranteza închisă care termină lista parametrilor formali ai funcției:

(2) ... **nume\_clasa::nume\_functie(...) const**

O astfel de funcție nu poate modifica datele membru ale obiectului pentru care se apelează (obiectul curent).

În exemplul de mai sus, funcția *afsisr* este o funcție care nu trebuie să modifice obiectul pentru care se apelează. De aceea, ea se poate defini utilizând modificatorul *const*:

```
void afsir(char *f) const;
```

O funcție care are antetul de formă (2) o vom numi *funcție const*.

La apelul funcției care nu este *funcție const*, pentru un obiect instanțiat cu ajutorul modificatorului *const*, compilatorul va emite un avertisment. Astfel, dacă vom utiliza funcția *afsisr* pentru clasa *sir* fără a fi definită ca *funcție const*, atunci la apelul:

```
s1.afsir();
```

compilatorul afișează un avertisment.

Dacă modificăm antetul și prototipul funcției *afsisr* adăugind modificatorul *const* după paranteza închisă:

```
void sir::afsisr() const
```

atunci la apelul funcției *afsisr* nu se va mai afișa avertismentul respectiv.

Se recomandă ca toate funcțiile membru care nu modifică nici o dată membru a obiectului curent, să fie realizate ca *funcții const*.

## 26. OPERATORII DE DEREFERENȚIERE A POINTERILOR SPRE MEMBRII CLASELOR (\*. SI ->\*)

Așa cum s-a văzut în capitolul 20, în limbajul C++, operatorul unar & (operatorul adresă) se utilizează nu numai pentru a defini adresa operanților, ci și pentru a defini date de *tip referință*, date care permit realizarea *apelului prin referință* (call by reference). De aceea, în limbajul C++, operatorul unar & se mai numește și *operator de referințiere*.

Operatorul unar \* (operatorul de indirectare) se aplică la un operand de tip pointer și el asigură accesul la data aflată în zona de memorie a cărei adresă de început este valoarea operandului respectiv. În felul acesta operatorul unar \* se poate considera că are un efect invers operatorului unar &.

**Exemplu:**

```
int x;  
int *p; // p este un pointer spre întregi de tip int
```

Fie:

```
p=&x;
```

Prin această expresie, lui *p* i se atribuie adresa zonei de memorie alocată lui *x*. La valoarea lui *x* avem acces folosind fie numele *x*, fie expresia:

```
*p
```

formată din operandul *p* la care se aplică operatorul unar \*.

Înlocuind pe *p* în expresia *\*p* cu *&x* ajungem la concluzia că expresia:

```
*&x
```

are aceeași valoare ca și operandul *x*.

Operatorul unar \*, din expresia de mai sus, are un efect opus față de operatorul unar &. El se mai numește și *operator de dereferențiere*.

În limbajul C++ se pot defini pointeri spre elementele membru ale claselor, pointeri spre datele membru, precum și pointeri spre funcțiile membru ale claselor.

Pointerii spre elementele membru ale claselor se declară folosind numele clasei și operatorul de rezoluție.

Fie clasa cu numele *nume\_clasa* care are o dată membru de tip *tip*. Atunci pointerul *p* spre o dată membru a clasei *nume\_clasa* de tip *tip*, se declară astfel:

```
(1) tip nume_clasa::*p;
```

În mod analog, se definesc pointeri spre funcțiile membru ale clasii. Fie, de exemplu, o funcție membru de prototip:

```
tip nume_clasa::nume_functie(lista tipurilor parametrilor formali);
```

Pointerul *pf* spre funcția de acest prototip se declară astfel:

```
(2) tip(nume_clasa::*pf)(lista tipurilor parametrilor formali);
```

Aceste declarații sunt analoge cu declarațiile de pointeri obișnuite. Diferența dintre ele constă în aceea că se utilizează numele clasei urmat de operatorul de rezoluție.

Pointerilor de acest tip li se atribuie valori ca mai jos.

Fie *d* o dată membru a clasei *nume\_clasa* de tip *tip*.

Atunci o expresie de forma:

```
(3) p = &nume_clasa::d
```

atribuie lui *p* adresa datei membru *d*.

În mod analog, pointerului *pf* i se atribuie o valoare folosind o expresie de forma:

```
(4) pf = nume_clasa::nume_functie
```

unde:

*nume\_functie* - Este o funcție membru a clasei *nume\_clasa*.

- Are prototipul indicat mai sus.

Expresiile (3) și (4) sunt acceptate în corpul funcțiilor membru ale clasei *nume\_clasa*, precum și în orice parte a programului, dacă elementele membru nu sunt protejate (*d* și *nume\_functie*).

Pointerii *p* și *pf* se utilizează aplicând *operatorii de dereferențiere* ".\*" și "->".

Ei se utilizează asemănător cu operatorii ".+" și "->".

Operatorul *punct* permite accesul la o componentă a unei date de un tip definit de utilizator. Numele datei structurate se află în stînga punctului, iar numele componentei în dreapta lui.

În mod analog, în stînga operatorului ".\*" se află instanțierea clasei (un obiect al clasei), iar în dreapta lui, în locul numelui componentei, se află pointerul spre componentă respectivă.

De exemplu, expresia:

```
obiect.*p
```

unde:

*obiect* - Este o instanțiere a clasei:

*nume\_clasa* adică *nume\_clasa obiect*;

*p* - Este pointerul declarat în (1) și la care i se atribuie o valoare prin expresia (3) permite acces la componentă:

```
obiect.d
```

unde:

*d* - Este o componentă a clasei *nume\_clasa*.

Expresia (3) atribuie pointerului *p* deplasamentul componentei *d*, care stabilește poziția ei în instanțierile clasei *nume\_clasa*.

Cu alte cuvinte, pointerul *p* declarat în (1) și la care î se atribuie o valoare prin expresia (3), permite înlocuirea expresiei:

*obiect.d*

cu

*obiect.\*p*

În felul acesta, *d* este schimbat cu *\*p* și apare ca o dereferențiere a pointerului *p*.

Operatorul "*->\**" se aplică atunci cind în partea stîngă se utilizează nu o instanțiere a clasei (un obiect), ci un pointer spre tipul definit de clasa respectivă. Astfel, dacă *pobiect* se declară ca un pointer spre tipul *nume\_clasa*:

*nume\_clasa \*pobiect;*

atunci pointerul *p* declarat în (1) și la care î se atribuie valoare prin expresia (3), permite înlocuirea expresiei:

*pobiect->d*

cu

*pobiect->\*p*

Și în acest caz *d* apare ca o dereferențiere a pointerului *p*.

Operatorii ".*\**" și "*->\**" pot fi utilizati pentru a apela funcții membru folosind declarații de forma (2) și la care î se atribuie valori prin expresii de forma (4).

Apelul:

*obiect.nume\_functie(...)*

unde:

*obiect*

- Este o instanțiere a clasei *nume\_clasa*.

*nume\_functie*

- Este o funcție membru a clasei *nume\_clasa* de prototipul indicat mai sus; se poate exprima folosind pointerul *pf* astfel:  
*(obiect.\*pf)(...)*

În mod analog, apelul:

*pobiect->nume\_functie(...)*

unde:

*pobiect*

- Este un pointer spre *nume\_clasa*, se poate exprima cu ajutorul pointerului *pf*, astfel:  
*(pobiect->\*pf)(...)*

În ambele cazuri, *nume\_functie* este schimbat cu *\*pf* și apare ca o

dereferețiere a pointerului *pf*.

Operatorii ".*\**" și "*->\**" au aceeași prioritate care este imediat mai mică decit a operatorilor unari și imediat mai mare decit a operatorilor multiplicativi (/,\*-binar și %). El se asociază de la stînga la dreapta ca majoritatea operatorilor limbajului C++.

Operatorul ".*\**" nu poate fi suprainscris. Avînd în vedere acest fapt, lista completă a operatorilor care nu pot fi suprainscrisă este următoarea:

. . \* :: ? :

### Exerciții:

26.1 Tipul *complex* se definește mai jos cu următoarele funcții membru:

- constructorul cu parametri implict;
- *sum*: insumează părțile reale sau imaginare a două obiecte;
- funcții care suprainscră operatorii binari + și - și operatorul unar minus;
- *afiscomplex*: afișează componentele unui obiect de tip complex;
- *citdouble*: citește un număr de la intrarea standard;
- *citcomplex*: citește componentele unui obiect de tip complex.

### FIŞIERUL BXVI1.H

```
class complex {  
    double real;  
    double imag;  
  
    void sum(complex c1,complex c2,double complex::*p);  
    /* - p pointeaza spre partea reala sau imaginara a obiectelor complexe;  
     - partile spre care pointeaza p ale obiectelor c1 si c2 se insumeaza si  
     se atribuie obiectului curent. */  
  
    static int citdouble(char *text,double& d);  
    /* - afiseaza text;  
     - citește un număr de la intrarea standard;  
     - returneaza:  
         0 - la EOF;  
         1 - altfel.  
    */  
  
public:  
    complex(double x=0,double y=0);  
    /*constructor utilizat la initializare și conversie dintr-un tip predefinit în tipul  
     complex*/  
  
    friend complex operator +(complex c1,complex c2);  
    // returnează suma c1+c2  
  
    complex operator -(); // returnează negativul obiectului curent  
  
    friend complex operator -(complex c1,complex c2);  
    // returnează suma c1-c2
```

```

void afiscomplex() const; // afiseaza componentele obiectului curent
int citcomplex();
/* - citeste componentele obiectului complex curent;
   - returneaza:
     0 - daca s-a intilnuit sfirsitul de fisier;
     1 - altfel.*/
};

Mai jos, se definesc functiile membru si prieten ale clasei complex intr-un
fisier de extensie .cpp.

```

## FIŞIERUL BXVII1

```

#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __COMPLEX_H
#include "BXVII1.H"
#define __COMPLEX_H
#endif

int complex::citdouble(char *text,double& d)
/* - afiseaza text;
   - citeste un numar de la intrarea standard si-l pastreaza in zona referita de parametrul d;
   - returneaza:
     0 - la intilnirea sfirsitului de fisier;
     1 - altfel.
*/
{
char t[255];
double x;

for(;;){
    if(text && *text) printf(text);
    if(gets(t)==0) return 0; // EOF
    if(sscanf(t,"%lf",&x) == 1) break;
    printf("nu s-a tastat un numar\n");
    printf("se reia citirea\n");
}
d=x;
return 1;
} // sfirsit citdouble

int complex::citcomplex()
/* - citeste componentele obiectului curent;
   - returneaza:
     0 - la intilnirea sfirsitului de fisier;
     1 - altfel.*/

```

```

if(citdouble("Partea reala=",real)==0) return 0; // EOF
if(citdouble("Partea imaginara=",imag)==0) return 0; // EOF
return 1;
}

inline void complex::afiscomplex() const
// afiseaza componentele obiectului curent
{
printf("%g+i*(%g)\n",real,imag);
}

inline void complex::sum(complex c1,complex c2,
                        double complex::*p)
/* - p pointeaza spre partea reala sau imaginara a obiectelor de tip complex;
   - partile obiectelor c1 si c2 spre care pointeaza p se insumeaza si se atribuie obiectului curent*/
{
this->*p = c1.*p+c2.*p;
}

inline complex::complex(double x,double y)
/* constructor utilizat la:
   - initializarea obiectelor;
   - conversia din tip predefinit in tip complex.
*/
{
real = x; imag = y;
}

complex operator +(complex c1,complex c2) // returneaza c1+c2
{
complex r;
double complex::*prelimag;

// sc insumeaza partile reale
prelimag = &complex::real;
r.sum(c1,c2,prelimag);

// sc insumeaza partile imaginare
prelimag = &complex::imag;
r.sum(c1,c2,prelimag);
return r;
}

complex complex::operator -() // returneaza negativul obiectului curent
{
complex r = *this;
r.real = -r.real; r.imag = -r.imag;
return r;
}

inline complex operator -(complex c1,complex c2) // returneaza c1-c2
{
return c1 + -c2;
}
```

}

### Observație:

La supraincărcarea operatorului + se utilizează un pointer spre datele membru ale clasei complex. Acesta s-a declarat în corpul funcției respective astfel:

```
double complex::*prelimag;
```

Apoi, el este încărcat pentru a pointa spre partea reală:

```
prelimag = &complex::real;
```

În continuare se apelează funcția *sum* care insumează părțile reale ale obiectelor care se adună.

Pentru a insuma părțile imaginare ale acelorași obiecte, nu avem decit să modificăm valoarea pointerului *prelimag* aşa ca el să pointeze spre partea imaginară:

```
prelimag = &complex::imag;
```

și apoi să apelăm din nou funcția *sum*.

Menționăm că, metoda de mai sus pentru supraincărcarea operatorului +, este mai complexă decit cea directă, utilizată în exercițiile din capitolul 23. Ea a fost considerată pentru a exemplifica modul în care se utilizează pointerii spre date membru ale unei clase.

Metoda devine utilă cind clasa are mai multe componente care se prelucrează la fel printr-o funcție membru de o complexitate mai mare decit funcția *sum* din acest exemplu.

26.2 Să se scrie un program care citește numere complexe de la intrarea standard și afișează suma lor.

Programul de față utilizează tipul *complex* implementat în exercițiul precedent.

### PROGRAMUL BXXVII

```
#include "BXXVII.CPP"

main() /* citeste numere complexe si afiseaza suma lor */
{
    complex s; // se instantiaza numarul complex nul
    complex z; // se utilizeaza pentru a citi un numar complex

    for(;;)
        if(z.citcomplex()==0) break; // EOF
        s = s+z;
    }
    s.afiscomplex();
}
```

## 27. CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR

Limbajele de programare oferă utilizatorului posibilitatea de a folosi în programare date de diferite tipuri, numite *tipuri predefinite*.

De obicei, toate limbajele de programare conțin tipuri predefinite pentru lucrul cu numere întregi de diferite dimensiuni.

Alte tipuri predefinite, utilizate frecvent, sunt cele pentru lucru pe caractere și cu numere neîntregi.

Datele de un anumit tip predefinit se disting prin modul de reprezentare și prin operațiile care se pot efectua asupra lor.

De exemplu, în limbajele C și C++ se disting date de tip *întreg* care se reprezintă prin *complement față de 2*, precum și date de tip *neîntreg* care au reprezentare diferită, numită *flotantă*. Cu ambele tipuri de date se pot face operații de calcul, cum ar fi cele 4 operații aritmetice, operații de comparație etc. Există însă și operații specifice, cum este de exemplu calculul restului împărțirii întregi (operatorul %) sau lucrul pe biți, care sunt interzise pentru datele de alte tipuri decit cele întregi.

Pe scurt, tipurile predefinite au predefinite atât reprezentarea datelor cit și operațiile care se pot executa asupra lor.

Tipul predefinit poate fi considerat ca o mulțime de date care au o aceeași reprezentare și asupra cărora este definit un set de operații. De aici decurge faptul că utilizarea datelor de un tip predefinit este controlată de compilator, în sensul că orice încercare de a utiliza operații neadmisibile asupra datelor de un tip predefinit va fi semnalată la compilare printr-un mesaj de eroare. În felul acesta se asigură o protecție a datelor de tipuri predefinite față de utilizarea eronată a acestora.

În general, la rezolvarea unei probleme intervin și alte tipuri de date decit cele predefinite.

În cazul limbajelor de programare apărute înaintea limbajului Pascal, concepțele care nu corespund tipurilor predefinite, se exprimă de către programator folosind tipurile predefinite din limbajele respective.

De exemplu, calculele cu numere fracționare sau complexe se pot realiza în orice limbaj folosind tipurile predefinite existente în limbajul respectiv. Astfel, numerele fracționare se definesc cu ajutorul a două date de tip întreg. Operațiile cu numere de acest fel se realizează apelând proceduri (funcții) corespunzătoare, ceea ce este posibil pentru orice limbaj care suportă programarea procedurală.

Pentru a simplifica folosirea în programare a concepțiilor care nu corespund direct tipurilor predefinite în limbajul utilizat, s-a ajuns la ideea de a oferi

utilizatorului posibilitatea reprezentării simple în limbajul de programare a conceptelor respective.

Primul limbaj care a fost proiectat în această idee a fost limbajul Pascal.

În acest limbaj se pot defini tipuri noi de date folosind construcția RECORD (înregistrare).

Această idee a fost preluată și de autorii limbajului C. În limbajul C se pot defini tipuri noi de date cu ajutorul construcției *struct*. Tipurile introduse în acest fel au fost numite *tipuri utilizator*. Ele deși simplifică realizarea de concepte cu ajutorul tipurilor predefinite ale limbajului, între tipurile utilizator și cele predefinite există diferențe mari.

În primul rînd, construcția *struct* definește numai reprezentarea datelor tipului utilizator, nu și operațiile care se pot executa cu datele respective.

De exemplu, tipul *complex*, în limbajul C, se definește ca mai jos:

```
struct complex {  
    double real;  
    double imag;  
};
```

În continuare, se pot declara date de tip *complex* în mod obișnuit:

```
struct complex c1,c2,...;
```

Aceasta înseamnă că fiecare din datele *c1,c2,...*, se compune din două componente de tip *double*. La aceste componente avem acces folosind operatorul punct:

*c1.real* și *c1.imag* - componentele lui *c1*;  
*c2.real* și *c2.imag* - componentele lui *c2*;

și așa mai departe.

În rest nu se mai cunoaște nimic despre tipul *complex* definit ca mai sus.

Utilizatorul poate defini funcții pentru a realiza diferite operații cu datele de tip *complex*.

Între reprezentarea datelor de tip *complex* (2 componente de tip *double*) și funcțiile care definesc operații asupra datelor de acest tip nu este specificată nici o legătură. De aceea, compilatorul nu are nici un control asupra operațiilor care se execută asupra datelor de tip *complex*. De asemenea, operatorii obișnuiți nu pot fi utilizati în cazul datelor de tip *complex*. O expresie de forma:

*c1+c2*

unde:

*c1* și *c2* - Sunt declarații ca mai sus, este eronată.

Un număr mic de operatori sunt predefiniți și pentru date de tip utilizator. Așa de exemplu, operatorul unar *&* (operatorul adresă) poate fi utilizat pentru operanzi care sunt de tip utilizator. Deci expresiile:

*&c1,&c2,...*

sunt admise de compilator și au ca valoare adresa de început a zonei alocate operandului la care se aplică operatorul adresă.

Pasul următor în dezvoltarea limbajelor de programare a fost acela de a propria, pe cît posibil, comportamentul datelor de tip utilizator de cel al datelor de tip predefinit.

În acest scop este necesar, în primul rînd, să se stabilească legătura dintre reprezentarea datelor și operațiile care se pot executa asupra datelor respective.

În al doilea rînd, este util ca operațiile cu aceste date să se poată exprima cu ajutorul operatorilor, ca în cazul datelor de tipuri predefinite. De exemplu, este de dorit ca suma dintre datele de tip complex *c1* și *c2* să se exprime cu ajutorul expresiei:

*c1+c2*

decit printr-un apel de funcție de forma:

*adcomplex(c1,c2)*

În al treilea rînd, este necesar să se definească conversii din tipuri predefinite într-un astfel de tip și invers, iar uneori chiar între două tipuri care nu sunt predefinite. De asemenea, este util ca astfel de conversii să se poată realiza implicit, ca în cazul tipurilor predefinite, iar alteori să poată fi explicitate folosind expresii *cast*.

În al patrulea rînd, este util ca datele de un astfel de tip să poată fi inițializate la declararea lor.

Aceste facilități permit utilizatorului să extindă tipurile predefinite ale limbajului cu tipuri noi care să aibă aceeași comportare ca și tipurile predefinite.

Ideea de a defini legătura dintre reprezentarea datelor unui tip și operațiile care se pot executa asupra datelor respective se realizează relativ simplu prin generalizarea construcției *struct*. Nu avem decit să înșirăm, în construcția *struct*, prototipurile funcțiilor care definesc operații asupra datelor tipului respectiv. Totuși, acest lucru nu este suficient, deoarece nu se asigură interzicerea accesului la datele tipului respectiv în afara funcțiilor a căror prototipuri se indică la definirea tipului. Astfel, o expresie de forma:

*c1.real = 1*

poate fi scrisă în orice punct al programului în care este definit *c1* prin declarația de mai sus.

Pentru a înlătura acest inconvenient este necesar să poată fi *protectat* accesul la date. Acest lucru se realizează utilizând *clasele* în locul construcției *struct* extinsă ca mai sus.

Clasa are același format ca și construcția *struct* extinsă cu prototipuri de funcții, dar în plus asigură protecția la utilizarea componentelor sale.

Clasa are două feluri de componente:

- date

- funcții.

Datele componente ale unei clase se numesc *date membru* ale clasei respective, iar funcțiile componente, *funcții membru*.

*Clasa și construcția struct extinsă cu prototipuri de funcții* sint facilități care sint admise în limbajul C++, ele neputind fi utilizate în limbajul C.

Menționăm că și construcția *union*, din limbajul C, poate fi extinsă în C++ cu prototipurile funcțiilor, ca și construcția *struct*.

Construcțiile *struct* și *union* extinse nu asigură însă protecție la utilizarea componentelor sale spre deosebire de *clase*.

Prin utilizarea claselor se face un prim pas spre apropierea tipurilor care nu sint predefinite de tipurile predefinite.

Un tip predefinit am văzut că are două aspecte:

- reprezentarea datelor

și

- setul de operații admise cu datele respective.

Tot așa, tipurile definite cu ajutorul claselor au aceleași proprietăți, definindu-se ambele aspecte.

În ambele cazuri, reprezentarea datelor constituie așa numita *partea de implementare* a tipului.

Setul de operații admis asupra datelor unui tip constituie *interfața* programului cu tipul respectiv.

Interfața definește utilizarea în program a datelor tipului respectiv.

Această parte a unui tip, de obicei, *nu este protejată*. Se obișnuiește să se spună că este *publică*.

În schimb, reprezentarea datelor trebuie să fie *protejată*, în sensul că, datele membru se pot accesa numai în corpul funcțiilor membru, nu și în afara lor.

În general, partea de implementare a unei clase trebuie să fie protejată, iar partea de interfață să fie publică (neprotejată).

În mod implicit, în cazul claselor toate elementele sale sint protejate. Evident, acest lucru nu este util și de aceea, s-au introdus *modificatori de protecție* care permit utilizatorului să stabilească părțile protejate și publice ale unei clase.

Amintim că, o construcție *struct extinsă* este ca o *clasă* care are toate elementele publice. De aceea, se obișnuiește să se spună că o construcție *struct extinsă* este o *clasă* cu elemente neprotejate.

Modificatorii de protecție pot fi utilizati și în cazul construcției *struct extinsă* și atunci ea se poate echivala cu orice *clasă*.

Construcția *union extinsă* este și ea ca o clasă care are toate elementele publice. În cazul ei nu se mai pot utiliza modificatorii de protecție pentru a proteja anumite elemente.

Protecția datelor membru ale unui tip definit printr-o clasă permite compilatorului să controleze operațiile care se execută cu datele tipului respectiv.

Într-adevăr, utilizatorul poate executa operații asupra datelor unui astfel de tip numai apelind funcții membru ale interfeței publice. De exemplu, definind tipul *complex* printr-o clasă a cărei reprezentare este protejată:

```
class complex { // date membru care sunt protejate în mod implicit
    double real;
    double imag;
public:
    // urmează partea neprotejată
};
```

și dacă definim datele *c1*, *c2* de tip *complex*:

```
complex c1,c2;
```

atunci o expresie de forma:

```
c1.real+c2.real
```

este eronată dacă se scrie în afara unei funcții membru, deoarece *componenta real* este o dată membru protejată.

Expresiile:

```
&c1
c1=c2
```

sunt corecte deoarece operatorii adresă și de atribuire sunt predefiniți și pentru date care sint de un tip care nu este predefinit. Amintim că atribuirea care se realizează în acest caz este numită *atribuire implicită* și aceasta este o copiere care se realizează bit cu bit pentru date de același tip.

Expresiile:

```
c1.real = c2.real;
c1.imag = c2.imag;
```

realizează același lucru ca și atribuirea:

```
c1=c2;
```

dar ele pot fi scrise numai în corpul unei funcții membru datorită protecției componentelor *real* și *imag*. Instrucțiunea de atribuire care nu utilizează componentele protejate respective poate fi scrisă oriunde în program.

O expresie de forma:

```
c1+c2
```

este eronată deoarece operatorul + nu este predefinit și pentru date care sint de un tip, altul decât cele predefinite.

Pentru a realiza suma dintre datele *c1* și *c2*, este necesară o funcție care să fie apelată cu datele *c1* și *c2* și care să returneze suma lor. Această funcție trebuie să fie *neprotejată* pentru a putea fi apelată în orice punct al programului în care este nevoie de suma a două numere complexe. Menționăm că o funcție membru se aplicează totdeauna pentru a prelucra o dată de tipul pentru care ea este funcție

membru. De exemplu, dacă funcția membru pentru adunarea datelor complexe  $c1+c2$  o numim *adcomplex*, atunci ea se apelează pentru primul operand al sumei, iar cel de al doilea operand va fi parametru. De aceea, funcția *adcomplex* are prototipul:

```
complex adcomplex(complex);
```

Apelul funcției *adcomplex* pentru a realiza suma  $c1+c2$ , are formatul:

```
c1.adcomplex(c2)
```

Utilizarea operatorului *punct* în apelul funcțiilor membru este în conformitate cu utilizarea lui în expresii de forma:

```
c1.real
```

și

```
c1.imag
```

În ambele cazuri, în stînga punctului se află o dată de tip *complex*, iar în dreapta, un membru al tipului *complex*.

Să observăm că interfața unui tip definit printr-o clasă ne permite să utilizăm datele de tipul respectiv fără a ne interesa de implementarea lui. Astfel, apelul de mai sus al funcției *adcomplex* nu conține componentele dată (real și imag) ale lui *c1* și *c2*. Dimpotrivă, utilizatorului îi este interzis accesul direct la componentele respective.

În concluzie, datele unui tip definit printr-o clasă se prelucră folosind numai interfața tipului respectiv, făcind *abstracție* de reprezentarea datelor tipului respectiv. În felul acesta, tipul definit printr-o clasă diferă substanțial față de tipurile utilizator care se definesc cu ajutorul construcției *struct* în limbajul C.

Tipurile definite cu ajutorul claselor se numesc *tipuri abstractive de date*.

Din cele de mai sus rezultă că, tipurile abstracte de date răspund primului deziderat al tipurilor de date care este și cel mai important și anume acela de a stabili legătura dintre reprezentarea datelor și operațiile care se pot executa asupra datelor respective.

Datele de tip abstract se mai numesc și *obiecte*. Despre obiectele care sunt de un același tip abstract se spune că sint *instantieri* ale clasei care definește tipul respectiv.

Datorită protecției pe care o asigură clasele datelor membru, se obișnuiește să se spună că datele respective sunt *incapsulate* în clase.

Mentionăm că se pot proteja și funcții membru. În acest caz, funcțiile respective nu pot fi apelate direct din program, ci numai din corpul altor funcții membru.

De asemenea, se pot defini date membru publice (neprotejate), dar se recomandă evitarea situațiilor de acest fel.

Obiectul pentru care se apelează o funcție membru se numește *obiectul curent*. În corpul funcției membru se poate face acces direct, fără nici o restricție, la componente obiectului curent.

Uneori este nevoie să ne referim la obiectul curent (nu la componente ale acestuia). Acest lucru este posibil utilizând pointerul implicit *this*. Aceasta este definit în corpul oricărei funcții membru (nestatică) și are ca valoare adresa obiectului curent.

De exemplu, în momentul apelului:

```
c1.adcomplex(c2)
```

în corpul funcției *adcomplex*, pointerul *this* are ca valoare adresa lui *c1* (*&c1*). De fapt, acest pointer se aplică în mod implicit în corpul funcțiilor membru nestatice la componente obiectului curent.

De exemplu, funcția *adcomplex*, de mai sus, se definește astfel:

```
complex complex::adcomplex(complex c)
{
    complex r; //obiect temporar
    r.real = real + c.real;
    r.imag = imag + c.imag;
    return r;
}
```

Elementele *real* și *imag* utilizate fără operatorul *punct* sunt componente obiectului curent. Lor li se aplică, în mod implicit de către compilatorul C++, operatorul  $\rightarrow$  folosind pointerul *this*. Deci, instrucțiunile respective sunt traduse de compilator ca și cind ar fi scrise sub forma:

```
r.real = this  $\rightarrow$  real + c.real;
r.imag = this  $\rightarrow$  imag + c.imag;
```

Funcțiile membru *static* (declarate cu ajutorul cuvintului *static* în definiția clasei) nu se apelează pentru un obiect al clasei. De aceea, în acest caz nu există noțiunea de obiect curent și nici pointerul *this* nu este definit în corpul lor.

Cel de-al doilea deziderat, ca să se poată utiliza operatorii obișnuiți ai limbajului cu operanzi obiecte, este și el realizat în limbajul C++. În acest caz se pornește de la noțiunea de *supraîncărcare a funcțiilor* (supradefinire, redifinire etc.).

În limbajul C++ se admite ca mai multe funcții să aibă același nume, funcții care să difere între ele prin numărul și/sau tipurile parametrilor formali.

Operatorii din limbajele C și C++ se pot aplica la date de diferite tipuri predefinite. De exemplu, expresia:

```
d1+d2
```

este corectă atât pentru date de tip intreg (*int*, *long*, *unsigned*), cit și pentru date neintregi (de tip *float*, *double* și *long double*). Se spune că operatorul  $+$  este supraîncărcat pentru tipurile *int*, *long*, *unsigned*, *float*, *double* și *long double*. Această supraîncărcare este predefinită. Pentru a putea utiliza expresia de mai sus și în cazul cind operanții *d1* și *d2* sunt obiecte (date de tipuri abstracte) este necesar a supraîncărca operatorul  $+$  pentru tipurile obiectelor respective. Aceasta

se realizează simplu folosind funcții membru, de nume special, pentru ca compilatorul să recunoască sensul funcțiilor respective.

O astfel de funcție are numele format din cuvintul cheie *operator* urmat de operatorul pe care-l supraincarcă. În rest, corpul funcției este obișnuit. De exemplu, pentru a supraincarca operatorul + pentru obiecte de tip *complex*, în locul funcției *adcomplex* de mai sus, vom folosi funcția definită astfel:

```
complex complex::operator +(complex c)
{
    complex r;
    r.real = real + c.real;
    r.imag = imag + c.imag;
    return r;
}
```

La întlnirea unei expresii de forma:

c1+c2

unde *c1* și *c2* sunt obiecte de tip *complex*, compilatorul apelează, în mod automat, funcția membru *operator +*, care are același efect cu apelul funcției *adcomplex* pentru obiectul *c1* și cu parametrul *c2*:

c1.adcomplex(c2)

De altfel, compilatorul, la întlnirea expresiei de mai sus, realizează în mod automat apelul:

c1.operator +(c2)

Se pot supraincarca numai operatorii existenți din limbajul C++, exceptind operatorii:

\* :: ? :

La supraincarcarea operatorilor se poate proceda ca mai sus, folosind funcții membru nestatice. Totuși, există cîțiva operatori care prezintă reguli specifice la supraincarcarea lor. Aceștia sunt:

->, [], (), new, delete, operatorul *cast*, =, +=, -=, \*=, /=, &=, ^=, |=, <<= și >>= (aici virgula a fost folosită ca separator).

De exemplu, operatorii *new* și *delete* se supraincarcă prin funcții *membru statice*, iar ceilalți prin funcții membru nestatice. Amănuite în legătură cu supraincarcarea operatorilor de mai sus se dau în capitolul 23, exceptind operatorul *cast* care este tratat în capitolul 24.

Funcțiile membru nestatice prezintă unele inconveniente la supraincarcarea operatorilor. Într-adevăr, dacă expresia:

c1+c2

poate fi tratată supraincarcind operatorul + ca mai sus, expresiile:

c1+d

și

d+c2

unde *d* este o dată de tip predefinit (*int*, *long*, *double* etc.) nu sunt recunoscute pe baza supraincarcării operatorului + cu ajutorul funcției *operator +* de mai sus.

Expresia:

c1+d

poate fi tratată definind, ca mai jos, încă o funcție membru nestatică pentru supraincarcarea operatorului +:

```
complex complex::operator+(double d)
{
    complex r;
    r.real = real+d; r.imag = imag;
    return r;
}
```

În schimb, expresia:

d+c2

nu se poate trata supraincarcind operatorul + cu funcții membru nestatice. Aceasta deoarece o funcție membru nestatică se apelează totdeauna pentru un obiect care este o instantiere a clasei pentru care funcția respectivă este funcție membru, iar pe de altă parte, funcțiile *operator* nestatice se apelează totdeauna pentru obiectul din stînga operatorului pe care-l supraincarcă.

În cazul expresiei de mai sus, nu se poate defini o funcție operator + care să fie funcție membru nestatică pentru clasa *complex*, deoarece în stînga operatorului + se află o dată care nu este de tip *complex*.

Pentru a putea utiliza expresii de forma celei de mai sus (primul operand este de un tip predefinit), operatorul + se supraincarcă folosind o *funcție prieten* a clasei *complex*.

Funcțiile *prieten* pentru o clasă sunt funcții obișnuite sau funcții membru ale unei alte clase, care au acces la elementele protejate ale clasei respective, la fel ca și funcțiile membru ale acelei clase.

Pentru a declara că o funcție este o funcție prieten a unei clase, prototipul funcției respective se scrie precedat de cuvintul *friend* în definiția acelei clase.

În cazul clasei *complex* se procedează astfel:

```
class complex {
    double real;
    double imag;
public:
    ...
    complex operator +(complex c);
    // ambeii operanzi sunt de tip complex; funcție membru nestatică
```

```

complex operator +(double d);
// primul operand este complex, iar al doilea de un tip predefinit
// functie membru nestatica

friend complex operator +(double d,complex c);
// returneaza suma d+c
// functie prieten a clasei complex; are acces la elementele protejate: real si imag
...
};

Funcția prieten pentru supraincărcarea operatorului + se poate defini astfel:

```

```

complex operator +(double d,complex c) // returneaza suma d+c
{
    return c+d;
}

```

Funcțiile prieten au și alte utilizări decit cele legate de supraincărcarea operatorilor.

O utilizare importantă a funcțiilor prieten este cea cu privire la construirea *iteratorilor*. În acest caz, iteratorul se realizează într-o clasă specială care este o *clasa prieten* a clasei pentru care se implementează iteratorul respectiv (vezi supraincărcarea operatorului ()).

O clasă *cl1* este o clasă prieten pentru clasa *cl2*, dacă se declară ca atare (*friend class cl1;*) în definiția clasei *cl2*.

Dacă o clasă *cl1* este o clasă prieten a clasei *cl2*, atunci toate funcțiile membru ale clasei *cl1* sunt funcții prieten pentru clasa *cl2*.

Cel de-al treilea deziderat pentru tipurile abstracte și anume acela ca să se poată defini conversii pentru obiecte care să se aplice implicit sau explicit, este și el implementat în limbajul C++ și se poate realiza prin intermediul supraincărării operatorului *cast* (vezi capitolul 24), precum și cu ajutorul *constructorilor*.

Constructorii sunt niște funcții membru speciale care au importanță mai ales în legătură cu inițializarea obiectelor. Ei au același nume cu numele clasei pentru care se definesc.

Pentru detalii în legătură cu constructorii se va vedea capitolul 22.

Constructorii se pot utiliza pentru a realiza conversii dintr-un tip predefinit în tipul abstract definit de clasa pentru care sunt funcții membru (vezi capitolul 24).

Un astfel de constructor are un singur parametru de tip predefinit. De exemplu, pentru a realiza conversii dintr-un tip predefinit în tipul *complex* se poate folosi următorul constructor:

```

complex::complex(double d)
{
    real = d; imag = 0;
}

```

Conversiile inverse, dintr-un tip abstract într-un tip predefinit, se realizează pe baza supraincărării operatorului *cast*.

De exemplu, conversia obiectelor de tip *complex* în tipul *double* se poate

realiza folosind funcția de mai jos pentru supraincărcarea operatorului *cast*:

```

complex::operator double() { return real; }

```

Aceste conversii uneori se aplică implicit, iar în alte cazuri este necesar să se aplice explicit operatorul *cast* supraincărat.

Conversiile realizate cu ajutorul constructorilor sau a operatorului *cast* supraincărat se numesc *conversii definite de utilizator* sau mai scurt *conversii utilizator*.

Ca regulă generală, conversiile utilizator implicite se aplică în ultimul rînd în evaluarea expresiilor, adică atunci cînd expresia nu poate fi evaluată pe baza aplicării de conversii implicite pentru date predefinite și pe baza apelării funcțiilor care supraincărcă operatorii pentru operanze la care se aplică. Aceasta înseamnă că, conversiile utilizator sint cele mai puțin prioritare.

### Exemple:

Fie clasa *complex*, definită ca mai jos:

```

class complex {
    double real;
    double imag;
public:
    complex(double x,double y=0) // constructor
    {
        real=x; imag=y;
    }

    complex operator +(complex c) // supraincarcarea operatorului +
    {
        complex r;
        r.real = real+c.real; r.imag = imag+c.imag;
        return r;
    }

    operator double() // supraincarca operatorul cast
    {
        return real;
    }
    ...
};

```

Constructorul clasei *complex* realizează conversii dintr-un tip predefinit, în obiecte de tip *complex*. El se apelează în mod automat la instanțierea obiectelor:

```

complex z1(1,2); // se instantiază z1=1+2i
complex z2(3,4); // se instantiază z2=3+4i
complex z3(0);   // se instantiază z3=0+0i

```

Expresia de atribuire:

```

z3=z1+z2

```

este corectă și se evaluatează apelind funcția care supraincărcă operatorul "+":

```
z3 = z1.operator +(z2)
```

La revenirea din funcție se returnează un obiect care corespunde sumei dintre obiectele complexe *z1* și *z2* și care, în lipsa supraincărcării operatorului "*=*", se atribuie bit cu bit lui *z3*.

O expresie de formă:

```
z3 = z1+123
```

este și ea corectă. Într-adevăr, în acest caz se aplică automat secvența de conversii:

```
123
```

- Se convertește din *int* în *double* (conversie implicită dintr-un tip predefinit într-un alt tip predefinit).
- Rezultatul conversiei precedente se convertește din *double* în *complex* (conversie implicită din *double* în tipul abstract *complex* realizată pe baza apelului constructorului).

Acest sir de conversii implicate rezultă din faptul că operatorul *+* este supraincărat cu o funcție membru nestatică a clasei *complex* precum și din modul de interpretare a expresiei de mai sus. Astfel, expresia de mai sus conduce la următorul apel al funcției *operator +*:

```
z3 = z1.operator +(123)
```

Deoarece funcția *operator +* are un parametru de tip *complex*, parametrul efectiv 123 este supus conversiilor implicate amintite mai sus.

Să presupunem că, clasa *complex* definită mai sus, mai are încă o funcție membru nestatică pentru supraincărcarea operatorului *"+"* definită ca mai jos:

```
complex complex::operator +(double d)
// returneaza obiectul curent adunat cu d
{
    complex r;
    r.real = real+d;
    r.imag = imag;
    return r;
}
```

În acest caz, expresia:

```
z3 = z1+123
```

se realizează fără a mai aplica conversia implicită realizată prin constructor.

Expresia se evaluează apelind funcția de mai sus care are un parametru formal de tip *double*:

```
z3 = z1.operator(123)
```

Parametrul efectiv 123 se convertește spre *double* și apoi se atribuie lui *d*. Acest mod de interpretare decurge din faptul că, conversia realizată prin constructor este o conversie utilizator și ea este mai puțin prioritată față de apelul funcțiilor care supraincarcă operatorii.

În condițiile de mai sus, expresia:

```
z3 = 123+z1
```

este cronată. Aceasta deoarece, în acest caz, expresia se interpretează astfel:

```
z3 = 123.operator(z1)
```

Cu alte cuvinte, într-o astfel de expresie nu se mai aplică în mod implicit constructorul pentru a realiza conversia lui 123 într-un obiect de tip *complex*.

O soluție posibilă este aplicarea explicită a constructorului:

```
z3 = complex(123)+z1
```

O altă soluție posibilă este secvența:

```
z3 = 123;
```

```
z3 = z3+z1
```

Prima instrucțiune conduce la o conversie a numărului 123 în *double* (conversie dintr-un tip predefinit în altul), iar din *double* în tipul *complex* prin apelul implicit al constructorului.

O altă soluție posibilă este de a defini o funcție prieten pentru supraincărcarea operatorului *"+"*, așa cum s-a indicat mai sus.

Prezența constructorului care să realizeze conversia dintr-un tip predefinit în tipul *complex* permite însă o soluție mult mai simplă, cu o singură funcție prieten pentru supraincărcarea operatorului *"+"*. Într-adevăr, fie definiția clasei *complex* de mai jos:

```
class complex {
    double real;
    double imag;
public:
    complex(double x,double y=0)
    {
        real = x; imag = y;
    }
    friend complex operator +(complex z1,complex z2)
    // returneaza z1+z2
    {
        complex r;
        r.real = z1.real + z2.real;
        r.imag = z1.imag + z2.imag;
        return r;
    }
    operator double() // conversie din complex in double
    {
        return real;
    }
    ...
};
```

Considerăm instanțierile *z1*, *z2* și *z3* ca mai sus, adică:

```
complex z1(1, 2);
complex z2(3, 4);
complex z3(0);
```

Supraîncărcarea operatorului "+", împreună cu constructorul clasei *complex* asigură recunoașterea și tratarea corectă a expresiilor de mai jos:

```
z3 = z1+z2; // nu se face conversie
z3 = z1+123; // 123 se converteste spre tipul complex
z3 = 123+z2; // 123 se converteste spre tipul complex
```

Expresiile de mai sus sunt interpretate astfel:

```
z3 = operator +(z1, z2); /* apel în care parametrii efectivi au același tip cu cei formalii*/
z3 = operator +(z1, 123); /* cel de al doilea parametru efectiv se converteste spre
                           tipul complex */
z3 = operator +(123, z2); /* primul parametru efectiv se converteste spre
                           tipul complex */
```

Această soluție pare a fi cea mai simplă și se poate utiliza la supraîncărcarea operatorilor binari obișnuiți. Ea presupune două lucruri:

- existența constructorului care să permită conversia dintr-un tip predefinit în tipul abstract pentru care se definește supraîncărcarea operatorilor binari obișnuiți;
- supraîncărcarea fiecărui operator binar obișnuit printr-o funcție prieten ai cărei parametri formali sunt ambii de tipul abstract pentru care se face supraîncărcarea.

Fie declarația:

```
double x;
```

O expresie de atribuire de forma:

```
x=z1
```

unde *z1* este un obiect *complex* declarat ca mai sus, este corectă și atribuie lui *x* partea reală a lui *z1*.

Această atribuire se realizează pe baza aplicării implicite a funcției membru care supraîncarcă operatorul *cast* pentru conversia obiectelor complexe într-o dată de un tip predefinit.

Variabila *x* poate avea orice tip predefinit. De exemplu, dacă *y* este de tip *long*:

```
long y;
```

atunci expresia:

```
y = z1
```

implică 2 conversii:

- una din *complex* în *double*
- și apoi

- din *double* în *long*.

Să observăm că o expresie de forma:

```
x = z1+123
```

unde *x* și *z1* sunt declarate ca mai sus, se evaluează astfel:

1. Se apelează funcția care supraîncarcă operatorul "+" pentru a realiza suma. Acest apel implică conversia 123 în *double* și apoi în *complex* prin apelul implicit al constructorului.
2. Se apelează operatorul *cast* supraîncărat pentru a converti suma complexă într-o dată de tip *double*.
3. Se atrbuie valoarea rezultată la punctul 2, variabilei *x*.

O evaluare mai economică se realizează dacă obiectul *z1* se convertește în *double* și abea apoi se realizează adunarea, după regulile obișnuite.

În acest scop, se poate folosi secvența:

```
x = z1;
x += 123;
```

O altă posibilitate este aceea de a explicita conversia lui *z1* spre *double* prin utilizarea explicită a operatorului *cast*:

```
x = (double)z1+123
```

sau

```
x = double(z1)+123
```

Conversia dintr-un tip abstract într-un alt tip abstract se poate realiza fie cu ajutorul unui constructor adecvat, fie supraîncăind operatorul *cast* în mod corespunzător.

Fie *t1* și *t2* două tipuri abstracte. Pentru a face conversia obiectelor din tipul *t1* în tipul *t2*, se poate defini pentru clasa *t2* un constructor de prototip:

```
t2(t1);
```

Pentru ca constructorul de acest prototip să poată realiza conversia obiectelor din tipul *t1* în tipul *t2*, de obicei, trebuie să aibă acces la datele membru protejate ale clasei *t1*. De aceea, constructorul de mai sus se va declara funcție prieten a clasei *t1*:

```
friend t2::t2(t1);
```

Un exemplu de astfel de constructor s-a definit în exercițiul 24.15 pentru conversia obiectelor din tipul *dc* în tipul *dczi*. De asemenea, clasa *dc* dispune de un constructor care realizează conversia inversă.

Așa cum s-a amintit mai sus, conversia din tipul *t1* în tipul *t2* se poate realiza și prin supraîncărcarea operatorului *cast*.

În acest caz, clasa *t1* va avea o funcție membru nestatică de antet:

```
operator t2();
```

Functia respectivă, de obicei, trebuie să aibă acces la datele membru protejate ale clasei *t2* și de aceea, ea se declara ca funcție prieten a clasei *t2*:

```
friend t1::operator t2();
```

Un exemplu de supraincărcare a operatorului *cast* se dă în exercițiul 24.12. În acest exercițiu se supraincarcă operatorul *cast* pentru a face conversii din grade *fahrenheit* în grade *celsius*.

Menționăm că, prezența mai multor posibilități de conversii dintr-un tip abstract într-un alt tip abstract (conversie atât prin constructor, cit și prin supraincărcarea operatorului *cast*) poate conduce la ambiguități.

Utilizarea constructorilor la conversii este o metodă folosită frecvent și de obicei ea se aplică în mod implicit.

Uneori aplicarea implicită a conversiilor pe baza apelului implicit al constructorilor poate să conducă la situații nedorite sau mai puțin eficiente. Un astfel de exemplu este evaluarea expresiei de mai jos:

```
x = z1+123
```

unde operandul 123 se convertește spre tipul *complex*, în loc să se procedeze la conversia lui *z1* în *double*.

În legătură cu aplicarea conversiilor definite de utilizator, amintim că dacă la evaluarea unui operand este nevoie să se realizeze mai multe conversii implice, atunci dintre acestea numai una poate implica tipuri abstracte, adică se admite o singură conversie utilizator, celelalte putind fi numai predefinite.

Cel de-al patrulea deziderat se referă la inițializarea obiectelor. Aceasta se realizează cu ajutorul constructorilor. Aceștia sunt funcții membru care se apelează automat la declararea obiectelor.

Datele de inițializare sunt valori pentru parametrii formali ai constructorilor.

Se pot defini mai mulți constructori pentru aceeași clasă, care să difere prin numărul și/sau tipul parametrilor.

În cazul în care nu există nici un constructor definit de utilizator pentru o clasă, compilatorul generează în mod automat un constructor pentru clasa respectivă. Acest constructor nu are parametri. El nu servește la inițializarea obiectelor clasei respective, ci are rol numai la alocarea obiectelor.

Un constructor fără parametri se numește constructor *implicit*.

În cazul în care clasa are cel puțin un constructor definit de programator, compilatorul nu mai generează un constructor pentru clasa respectivă.

În concluzie, clasa poate să nu aibă constructori dacă nu se dorește inițializarea obiectelor la instanțierea lor și în acest caz compilatorul generează un constructor implicit.

Constructorul implicit se definește de programator în cazul în care sunt definiți și alți constructori pentru clasa respectivă care au fiecare cel puțin un parametru neimplicit (fără valoare implicită) și se dorește instanțierea de obiecte neinițializate.

Dacă o clasă are definit un constructor care are toți parametrii implicați, atunci

pentru el nu se mai poate defini și un constructor implicit, deoarece apelul acestora devine ambigu.

**Exemplu:**

```
class complex {  
    double real;  
    double imag;  
public:  
    complex() // constructor implicit  
    {  
    }  
  
    complex(double x=0,double y=0) // constructor cu toti parametrii impliciti  
    {  
        real = x; imag = y;  
    }  
    ...  
};
```

**Declarația:**

```
complex z;
```

devine ambiguă, deoarece la instanțierea lui *z* se poate aplica oricare dintre cei doi constructori ai clasei *complex*. În astfel de situații se renunță la constructorul implicit.

Clasa *complex* definită ca mai jos:

```
class complex {  
    double real;  
    double imag;  
public:  
    complex(double x,double y=0)  
    {  
        real=x; imag=y;  
    }  
    ...  
};
```

permete instanțierea numai de obiecte inițializate.

Pentru a putea declara obiecte neinițializate, va fi nevoie să se definească constructorul implicit.

Așa cum s-a arătat mai sus, constructorii unei clase se pot utiliza la conversiile datelor de tip predefinit în obiecte ale clasei respective. În acest caz, constructorul trebuie să aibă cel puțin un parametru. Dacă sunt mai mulți, cel mult unul poate să nu fie implicit.

Parametrii constructorului pot fi de orice tip, dar care este diferit de tipul abstract implementat de clasa pentru care este constructor.

Totuși, parametrii unui constructor pot fi pointeri sau referințe la tipul abstract implementat de clasa la care aparține constructorul.

### Exemplu:

Un constructor de prototip:

```
complex::complex(complex z);
```

este eronat. În schimb, se poate defini un constructor de următorul prototip:

```
complex::complex(complex& z);
```

Un constructor de prototip:

```
nume_clasa::nume_clasa(const nume_clasa&);
```

se numește *constructor de copiere*.

Acesta se apelează automat în următoarele situații:

- la instanțierea unui obiect al clasei *nume\_clasa* se utilizează, pentru inițializare, un alt obiect al aceleiași clase și care deja este instanțiat;
- la transferul parametrilor efectivi care sunt obiecte ale clasei *nume\_clasa*, transfer care nu implică conversii utilizator;
- la revenirea dintr-o funcție care returnează un obiect al clasei *nume\_clasa*.

Să observăm că, constructorul de copiere nu se apelează la evaluarea atribuirilor.

Dacă pentru o clasă nu există definit un constructor de copiere, atunci compilatorul generează în mod automat un astfel de constructor pe care îl numim *constructor de copiere implicit*. Acesta realizează o copiere bit cu bit, ceea ce adesea este suficientă. De exemplu, pentru tipul *complex*, implementat ca în exemplele precedente este suficientă copierea bit cu bit.

În general, constructorul de copiere este necesar pentru clase care au componente date de tip pointer sau referință.

Un exemplu de astfel de clasă este clasa *sir*, implementată în exercițiul 23.14.

Dacă constructorul se apelează la crearea obiectelor, la dezalocarea (distrugerea) lor se apelează o altă funcție membru numită *destructor*.

În general nu este obligatoriu să definim un destructor pentru o clasă. Dacă o clasă nu are definit un destructor de către programator, atunci compilatorul generează în mod automat un destructor pentru clasa respectivă.

Destructorul generat de compilator se numește *destructor implicit*.

Destructorii sunt funcții membru speciale care nu au parametri (vezi capitolul 22).

Pentru o clasă se poate defini cel mult un destructor.

Destructorii se definesc, de obicei, pentru clase care au date membru pointeri sau referințe. Un exemplu de astfel de clasă este clasa *sir* din exercițiul 23.14.

Programarea care se bazează pe utilizarea tipurilor abstracte de date o vom numi *programare prin abstractizarea datelor*.

În principiu, rezolvarea unei probleme folosind programarea prin abstractizarea datelor presupune ca prim pas, stabilirea conceptelor care intervin în descrierea metodei de rezolvare a problemei respective. La pasul următor se implementează conceptele care nu corespund tipurilor predefinite ale limbajului

ales pentru rezolvarea problemei. Dacă limbajul utilizat este limbajul C++, atunci conceptele care nu corespund tipurilor predefinite se implementează sub formă de tipuri abstracte folosind clase. Apoi, se descrie metoda de rezolvare a problemei respective folosind tipurile abstracte implementate prin clase ca și cum tipurile respective ar exista implementate în limbajul C++ alături de cele predefinite.

Dintre concepții utilizate mai frecvent în programare și care nu sunt implementate ca tipuri predefinite în limbajul C++ amintim:

- siruri de caractere;
- numere rationale;
- numere complexe;
- tablouri dinamice;
- liste simplu înlanțuite;
- liste dublu înlanțuite;
- liste circulare;
- arbori;
- tabele de dispersie.

Un neajuns al programării prin abstractizarea datelor este faptul că nu permite exprimarea legăturilor dintre diferite concepții. Singura legătură dintre concepții care se poate exprima, este aceea că datele membru ale unei clase pot fi obiecte ale unei alte clase. Acest lucru nu este suficient în cazul în care concepții sunt strâns dependente între ele formând structuri "ierarhice".

Exprimarea ierarhiilor conduce la atribuire suplimentare cum sint cele de *moștenire*. Aceasta conduce la un model nou de programare pe care îl numim *programare orientată spre obiecte*.

## 28. PROGRAMAREA ORIENTATĂ SPRE OBIECTE

Istoria dezvoltării științelor marchează o perioadă lungă în care s-au acumulat date despre formele de existență ale materiei și fenomenele care au loc în natură și în viața noastră de zi cu zi.

Perioada următoare se caracterizează prin aceea că, datele acumulate au fost supuse unui proces de clasificare potrivit unor trăsături care se constată că sunt comune pentru mai multe fenomene sau forme ale materiei moarte sau vii. Acest proces conduce în mod natural la stabilirea de *ierarhii* între diferențele fenomene și forme de existență ale materiei.

În virful unei ierarhii se află fenomenul sau forma de existență care are trăsături *comune* pentru toate celelalte componente ale ierarhiei respective. Pe nivelul următor al ierarhiei se află componente care pe lângă trăsăturile comune de pe nivelul superior, mai au și trăsături suplimentare, *specifice*. O ierarhie, de obicei, are mai multe niveluri, iar situația unui element pe un nivel sau altul al ierarhiei este uneori o problemă deosebit de complexă.

Elementul din virful ierarhiei este considerat a fi cel mai *general* dintre elementele ierarhiei. Toate celelalte elemente ale ierarhiei sunt *mai particulare*, *mai specializate*. Acest lucru este valabil în orice punct al ierarhiei. Astfel, un element aflat la un nivel al ierarhiei este mai particular (mai specializat) decât cel aflat pe nivelul imediat anterior lui și este mai general decât elementul aflat pe un nivel imediat următor lui. Se obișnuiește să se spună că un element al ierarhiei *moștenește* trăsăturile elementelor de pe nivelurile anterioare.

În general, un element al ierarhiei poate moșteni trăsături de la mai multe elemente ale ierarhiei. Ierarhia este *arborescentă*, dacă orice element al ei, diferit de elementul din virful ierarhiei, moștenește trăsăturile de la un singur element.

La ora actuală, toate ramurile cunoașterii științifice sunt pline de ierarhii rezultate în urma clasificării cunoștințelor acumulate în perioada lungă de observare a fenomenelor și formelor de existență a lumii materiale. Astfel, clasificările ierarhice ale cunoștințelor pot fi întlnite atât în domeniile care pleacă de la cele mai concrete forme ale lumii materiale, cum sunt botanica, zoologia, biologia etc., cit și în domeniul care se consideră că studiază concepțele cele mai abstracte, cum sunt ramurile matematicii: geometria, algebra abstractă etc.

Un exemplu simplu întlnit în geometrie a fost amintit chiar în introducerea acestei cărți. El se referă la clasificarea patrulaterelor. Astfel, dacă se consideră că *paralelogramul* este în virful ierarhiei, atunci pe nivelul următor se poate așeza *dreptunghiul*. Într-adevăr, dreptunghiul este un paralelogram la care se adaugă condiția, să aibă un unghi drept. În felul acesta, dreptunghiul are (moștenește) toate proprietățile paralelogramului, dar mai are și proprietăți în plus (de exemplu, diagonalele dreptunghiului sunt egale). Paralelogramul este

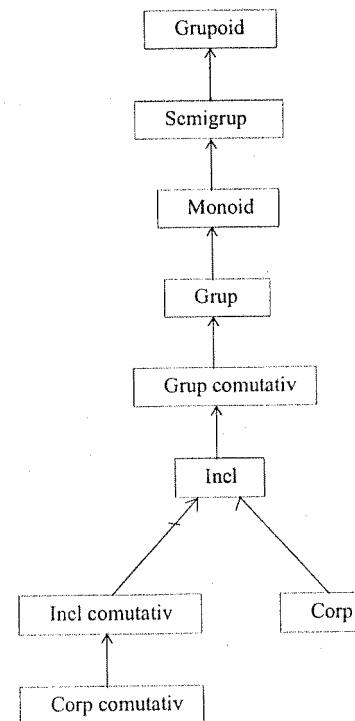
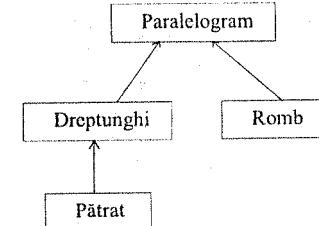
un concept mai general decât dreptunghiul, iar acesta din urmă este un paralelogram particular.

În mod analog, *pătratul* se poate defini ca fiind un dreptunghi particular și anume un dreptunghi care are toate laturile egale. Aceasta înseamnă că pătratul moștenește toate proprietățile dreptunghiului, dar făcând asta, pătratul mai are și alte proprietăți (de exemplu, diagonalele lui sint axe de simetrie).

*Rombul* este și el un paralelogram particular și anume, el este un paralelogram care are toate laturile egale. El moștenește toate proprietățile paralelogramului dar mai are și proprietăți în plus (de exemplu, diagonalele perpendiculare).

Se obține în felul acesta o ierarhie arborescentă ca în figura alăturată.

Un alt exemplu simplu de ierarhie este ierarhia stabilită între concepțele *grupoid*, *semigrup*, *monoid*, *grup*, *inel* și *corp* din algebra abstractă. Această ierarhie este schematizată ca mai jos.



Să amintim pe scurt definițiile acestor concepțe din algebra abstractă.

Ca noțiune primară vom considera noțiunea de *mulțime*.

Fie A și B două mulțimi. Cu  $A \times B$  se notează mulțimea perechilor ordonate de elemente:

$(x,y)$

unde:

- x - Este element al mulțimii A.  
y - Este element al mulțimii B.

Mulțimea  $A \times B$  se numește *produsul cartezian* al mulțimilor A și B.

Fie A și B două mulțimi. Dacă fiecărui element  $x$  din mulțimea A îi punem în corespondență un element  $y$  din mulțimea B și numai unul singur, spunem că am definit o *funcție* sau o *aplicație* pe mulțimea A cu valori în mulțimea B.

Evident, putem defini multe funcții pe o mulțime A și cu valori într-o mulțime B (B poate dифe ри de A sau să coincidă cu A). Pentru a le putea distinge, de obicei, fiecărei funcții i se dă un nume. Faptul că este o funcție definită pe A și cu valori în B, se notează astfel:

$f: A \rightarrow B$

Să observăm că notația, de mai sus, nu definește modul în care se realizează corespondența dintre elementele lui A și B. Ea exprimă numai faptul că prin  $f$  s-a notat o astfel de corespondență.

Fie A o mulțime nevidă. Prin *lege de compoziție* definită pe A se înțelege o aplicație  $f$  definită pe produsul cartezian  $A \times A$  și cu valori în A:

$f: A \times A \rightarrow A$

#### Exemplu:

Cu N notăm mulțimea numerelor naturale  $\{0, 1, 2, \dots\}$ .

Produsul cartezian  $N \times N$  este mulțimea perechilor de numere naturale:

$(m, n)$

Un exemplu de lege de compoziție definită pe mulțimea numerelor naturale este operația de *adunare*, pe care o notăm cu simbolul "+":

$+: N \times N \rightarrow N$

Operația de adunare pune în corespondență fiecărei perechi de numere naturale  $(m, n)$  un al treilea număr natural care se numește suma lor și care se determină în mod unic.

O altă lege de compoziție definită pe mulțimea numerelor naturale este operația de *înmulțire*, pe care o notăm cu simbolul "\*":

$*: N \times N \rightarrow N$

O mulțime se spune că formează un *grupoid* dacă este definită o lege de compoziție internă pe mulțimea respectivă.

Mulțimea numerelor naturale este *grupoid* atât față de operația de adunare cît și față de cea de înmulțire.

Legile de compoziție au, de obicei, diferite proprietăți. Conceptul de *grupoid* este cel mai general. El se află în virful ierarhiei. Se obțin concepte mai particulare în funcție de proprietățile legilor de compoziție.

O proprietate a legilor de compoziție întâlnită frecvent este cea de *asociativitate*. Aceasta se definește ca mai jos.

Fie A o mulțime pe care este definită operația notată cu "#":

$#: A \times A \rightarrow A$

(A este grupoid față de operația notată cu "#").

Legea de compoziție # este *asociativă* dacă:

$(a \# b) \# c = a \# (b \# c)$

oricare ar fi elementele  $a, b, c$  din A.

Un grupoid formează un *semigrup* dacă legea de compoziție este asociativă.

Menționăm că un grupoid față de operația de adunare se numește *grupoid aditiv*, iar un grupoid față de operația de înmulțire se numește *grupoid multiplicativ*.

În mod analog, un *semigrup* care este un grupoid aditiv este un *semigrup aditiv*, iar un semigrup care este un grupoid multiplicativ este un *semigrup multiplicativ*.

Mulțimea numerelor naturale este atât un semigrup aditiv cît și un semigrup multiplicativ, deoarece ambele operații sunt associative.

În general, pot să existe elemente care să aibă un comportament aparte față de legile de compoziție. Astfel de elemente sunt, *elementele neutru*. Dacă notăm cu  $e$  un element neutru față de o lege de compoziție # definită pe A, atunci:

$e \# x = x \# e = x$

unde  $x$  este un element oricare al mulțimii A.

Nu orice semigrup are un element neutru. Semigrupul aditiv și multiplicativ, ambele definite pe mulțimea numerelor naturale, au fiecare un element neutru. Astfel, zero este element neutru pentru semigrupul aditiv, iar unu este element neutru pentru semigrupul multiplicativ.

Se demonstrează că dacă există un element neutru față de o lege de compoziție, atunci el este unic.

Să observăm că mulțimea:

$M = \{1, 2, \dots\} = N - \{0\}$

este un semigrup aditiv care nu are element neutru. În mod analog,

$P = \{0, 2, 3, \dots\} = N - \{1\}$

este un semigrup multiplicativ care nu are element neutru.

Dacă un semigrup are o lege de compoziție care are element neutru, atunci

semigrupul respectiv se spune că formează un *monoid*. Noțiunile de *monoid aditiv* și *monoid multiplicativ* rezultă imediat din cele de semigrup aditiv și respectiv multiplicativ. Multimea N a numerelor naturale formează un monoid atât față de adunare, cit și față de înmulțire. În schimb, multimea M = N - {0} nu este un monoid aditiv, iar P = N - {1} nu este un monoid multiplicativ.

Noțiunea de monoid a apărut ca o particularizare a noțiunii de semigrup.

O altă noțiune importantă în legătură cu legile de compozitie este noțiunea de element *simetrizabil*.

Fie multimea A care este un monoid față de legea de compozitie  $\#$ , cu elementul neutru  $e$ .

Fie  $x$  un element al mulțimii A. Un element  $y$  din A este *simetricul* lui  $x$ , dacă:

$$x\#y = y\#x = e$$

Simetricul unui element, dacă există, este unic.

Un element se spune că este simetrizabil dacă are simetric.

Dacă orice element al unui monoid este simetrizabil, atunci monoidul respectiv este un *grup*.

Multimea N nu este grup nici față de operația de adunare și nici față de operația de înmulțire. Într-adevăr, singurul element simetrizabil din multimea N, față de adunare, este 0, deoarece numai el are proprietatea:

$$0+0 = 0$$

În mod analog, singurul element simetrizabil din multimea N față de înmulțire este 1:

$$1*1 = 1$$

Multimea numerelor intregi, notată cu Z, formează un grup aditiv. Într-adevăr, adunarea numerelor intregi are toate proprietățile necesare pentru a forma un grup:

- adunarea este asociativă, deci formează un semigrup;
- zero este element neutru față de adunare, deci este un monoid;
- dacă  $x$  este un întreg, atunci  $-x$  este simetricul lui, deoarece:  

$$x + (-x) = 0$$

Multimea Z formează un monoid multiplicativ, dar nu și un grup multiplicativ, deoarece numai 1 și -1 sunt simetrizabile față de înmulțire.

Legea de compozitie  $\#$  definită pe multimea A este *comutativă* dacă:

$$x\#y = y\#x$$

oricare ar fi elementele  $x$  și  $y$  din multimea A.

Un grup se numește *comutativ* dacă legea de compozitie definită pe multimea A, față de care A formează grup, este comutativă.

Multimea Z formează un grup aditiv comutativ deoarece adunarea numerelor intregi este comutativă.

O mulțime A este un *inel* dacă pe ea sunt definite două legi de compozitie care au următoarele proprietăți:

- A este grup comutativ față de una dintre legile de compozitie;
- A este monoid față de cealaltă lege de compozitie;
- legea de compozitie față de care A este monoid, este distributivă față de cealaltă lege de compozitie.

De obicei, legea de compozitie față de care A este un grup comutativ se notează cu  $+$ , adică se consideră că grupul este aditiv. Cealaltă lege se obisnuiește să se noteze cu  $*$ , adică se consideră că monoidul este multiplicativ.

Cu aceste notări, proprietatea de distributivitate se exprimă astfel:

$$x*(y+z) = x*y + x*z$$

oricare ar fi  $x$ ,  $y$  și  $z$  elemente ale lui A.

Mulțimile Z și Q (multimea numerelor raționale) sunt exemple de inele.

Un *inel* este *comutativ* dacă operația de înmulțire (\*) este comutativă, adică:

$$x*y = y*x$$

oricare ar fi elementele  $x$  și  $y$  ale lui A.

Inelele Z și Q sunt comutative.

Un exemplu de inel necomutativ este multimea matricelor pătratice de ordinul  $n$  ( $n > 1$ ) ale căror elemente sunt numere aparținând lui Z, Q sau R (multimea numerelor reale).

Elementul neutru al unui monoid aditiv se notează, de obicei cu zero, iar al unui monoid multiplicativ cu unu.

Un inel este *corp*, dacă elementele neutru ale celor două legi de compozitie sunt diferite și dacă orice element diferit de zero (elementul neutru al grupului aditiv) este simetrizabil în raport cu operația de înmulțire.

Ca exemplu de coruri amintim mulțimile Q, R și C (multimea numerelor complexe).

În fine, dacă inelul care este un corp, este comutativ, atunci corpul respectiv este și el comutativ.

Din exemplele de mai sus, rezultă că ierarhizarea conceptelor se realizează din aproape în aproape. La început se stabilesc elementele comune și ele definesc conceptul cel mai general. Apoi, se fac particularizări adăugindu-se proprietăți noi și se ajunge din aproape în aproape la concepe din ce în ce mai particulare (mai specializate).

Caracteristica de bază a unei ierarhii este *proprietatea de moștenire*.

Conform acestei proprietăți, un concept al ierarhiei de pe un nivel are toate proprietățile conceptului aflat pe nivelul imediat superior. De exemplu, dreptunghiul are toate proprietățile paralelogramului, iar pătratul are toate proprietățile dreptunghiului.

Ierarhiile pot apărea la încercarea de a defini conceptele necesare pentru a rezolva o anumită problemă la un calculator.

Am văzut că pentru conceptele care nu corespund tipurilor predefinite din

în limbajul C++, putem defini clase prin care să se implementeze. Până în acest moment, clasele au fost definite considerindu-se independente conceptele pe care le implementează. Este necesar ca ierarhia conceptelor să fie reflectată de către clasele care le implementează. Aceasta implică o noțiune nouă pentru a defini în primul rînd, proprietatea de bază a ierarhiilor - moștenirea. Proprietatea de moștenire se realizează cu ajutorul claselor deriveate. Noțiunea de clasă derivată se definește ca mai jos.

Fie  $c1$  și  $c2$  două concepte ale unei ierarhii aşa încit  $c1$  se află pe un nivel imediat superior față de conceptul  $c2$ . Cu alte cuvinte, conceptul  $c2$  se obține din  $c1$  adăugind caracteristici noi.

Fie  $c1$  și  $c2$  clasele care implementează conceptele  $c1$  și respectiv  $c2$ . Atunci clasa  $c2$  se spune că este o clasă derivată a clasei  $c1$ . Aceasta înseamnă că clasa  $c2$  are ca elemente membru toate elementele membru (date membru și funcții membru) ale clasei  $c1$  (acestea sunt elemente moștenite), precum și alte elemente specifice ei. Se obișnuiește să se spună că clasa  $c2$  moștenește elementele clasei  $c1$ . Despre clasa  $c1$  se spune că este o clasă de bază pentru clasa  $c2$ .

Reluind ierarhia care are în virf conceptul de paralelogram, vom presupune că, concepte din ierarhie se implementează prin clase cu aceleași denumiri. Astfel, conceptul de paralelogram se implementează cu ajutorul clasei paralelogram, conceptul de dreptunghi cu ajutorul clasei dreptunghi și aşa mai departe.

În acest caz, clasa dreptunghi este o clasă derivată a clasei paralelogram, iar clasa paralelogram este o clasă de bază a clasei dreptunghi. În mod analog, clasa romb este o clasă derivată a clasei paralelogram, iar clasa paralelogram este o clasă de bază a clasei romb.

În sfîrșit, clasa pătrat este o clasă derivată a clasei dreptunghi, iar clasa dreptunghi este o clasă de bază a clasei pătrat. În felul acesta, o ierarhie de concepte conduce la o ierarhie între clasele care implementează conceptele ierarhiei respective. Clasa derivată se află totdeauna pe un nivel imediat inferior celui corespunzător clasei de bază. La baza ierarhiei dintre clase se află aceeași proprietate ca în cazul ierarhiei de concepte, adică proprietatea de moștenire.

O ierarhie stabilită între mai multe concepte poate să nu fie unică.

De exemplu, ierarhia care are în virf conceptul de paralelogram poate fi modificată aşa încit pătratul să nu fie un caz particular al dreptunghiului, ci al rombului. Într-adevăr, pătratul se poate defini ca un romb care are un unghi drept. În acest caz, clasa pătrat devine o clasă derivată a clasei romb, iar aceasta la rîndul ei devine o clasă de bază a clasei pătrat.

Așa cum s-a spus mai sus, un concept poate moșteni atribute de la mai multe concepte. Aceasta înseamnă că o clasă poate deriva din mai multe clase de bază. În acest caz, se spune că moștenirea este multiplă (clasa derivată moștenește de la mai multe clase de bază).

Revenind la exemplul de mai sus, să observăm că, conceptul de pătrat, se poate defini ca un dreptunghi care este în același timp și un romb sau invers, adică un romb care este și dreptunghi. Se obține ierarhia din figura de mai jos.

În felul acesta, clasa pătrat este o clasă derivată din clasele dreptunghi și romb.

\* Programarea care permite definirea de ierarhii între tipurile abstrakte de date pe baza conceptului de moștenire, se numește programare orientată spre obiecte.

În limbajul C++, ierarhia dintre tipurile abstrakte de date se exprimă prin ierarhia de clase care implementează tipurile abstrakte respective. Ierarhia dintre clase se stabilește cu ajutorul noțiunilor de clasă derivată și clasă de bază.

B. Stroustrup arată în lucrarea [14] că în cazul în care nu există ierarhii între tipurile abstrakte de date utilizate într-un program (tipuri abstrakte independente), atunci este suficientă programarea prin abstractizarea datelor.

Programarea orientată spre obiecte, având la bază procesul de moștenire, este un salt calitativ față de programarea prin abstractizarea datelor.

Într-adevăr, procesul de moștenire implică un stil nou în programare.

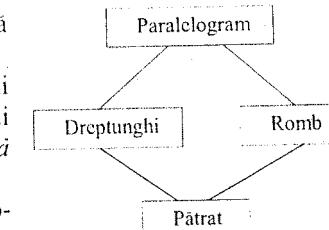
În primul rînd, tipurile abstrakte care au părți comune nu se mai implementează independent. Fiecare tip abstract se implementează moștenind elementele comune de la alte tipuri abstrakte aflate pe un nivel superior în ierarhia din care acesta face parte. În felul acesta, un tip abstract care are elemente comune cu altele se implementează adăugind elemente noi la cele existente. Astfel de adăugări sunt posibile oriind dacă se constată că este nevoie de noi tipuri abstrakte de date. Aceasta înseamnă că, atunci cind se constată că un program trebuie modificat sau dezvoltat, pe baza utilizării de noi tipuri abstrakte de date, tipurile vechi, de obicei, nu se modifică, ci li se adaugă elemente noi. În felul acesta, problema reutilizării componentelor existente nu înseamnă modificarea sau reprogramarea lor, ci completarea lor cu elemente noi.

Programarea orientată spre obiecte utilizează și alte concepte decit moștenirea.

Un alt concept al programării orientate spre obiecte, suportat de limbajul C++, este polimorfismul.

Polimorfismul este un concept care decurge în mod natural din conceptul de moștenire.

Reluind exemplul cu privire la figurile geometrice, vom observa că fiecărei figuri din ierarhie îi se poate calcula perimetru și aria, definind funcții membru corespunzătoare. De asemenea, figurile ierarhiei pot fi afisate pe ecranul display-ului setat în mod grafic, pot fi deplasate etc. În acest scop este necesar să definim funcții membru specifice pentru fiecare figură în parte. Astfel, de exemplu, este nevoie de o funcție membru pentru clasa paralelogram care să afișeze paralelograme, o altă funcție membru pentru clasa dreptunghi care să afișeze dreptunghiuri și aşa mai departe. Aceste funcții pot fi considerate ca fiind versiuni ale aceleleași funcții. Aceasta este în concordanță cu denumirea de



*polimorfism* care vine din greacă și înseamnă "a avea mai multe forme".

Problema care se pune în legătură cu funcțiile de acest fel este cea legată de selectarea versiunilor unei astfel de funcții. Pentru a lămuri acest lucru să presupunem că în cazul ierarhiei de figuri geometrice am defini, pentru fiecare clasă, funcția membru *afiseaza*, care afișează pe display obiectul curent.

Astfel, dacă avem instanțele:

```
paralelogram p(...);  
dreptunghi d(...);  
patrat pt(...);  
romb r(...);
```

atunci un apel de forma:

```
p.afiseaza()
```

afișează paralelogramul definit de obiectul *p*. De asemenea:

- *d.afiseaza()* -afișează un dreptunghi;
- *pt.afiseaza()* -afișează un patrat;
- *r.afiseaza()* -afișează un romb.

La aceste apeluri se utilizează, de fiecare dată, funcția membru *afiseaza* a clasei corespunzătoare obiectului prezent în apel.

În mod analog, se poate defini o funcție *sterge* pentru fiecare din clasele ierarhiei care să anuleze afișarea obiectului curent.

Funcțiile *afiseaza* și *sterge* se pot utiliza pentru a face deplasări pe ecran ale figurilor respective. Numim *deplasare* o funcție care permite deplasarea unei figuri din poziția curentă într-o poziție nouă a ecranului. O astfel de funcție poate fi definită cu doi parametri, *nx* și *ny*, care definesc modificările coordonatelor virfurilor figurii care se deplasează. Numim *modif* funcția care determină datele membru ale unei figuri în urma deplasării figurii respective. Evident, fiecare clasă a ierarhiei are o funcție membru *modif* proprie (specifică).

Funcția *deplasare* se poate defini ca mai jos:

```
void paralelogram::deplasare(int nx,int ny)  
{  
    sterge(); // sterge de pe ecran paralelogramul curent  
    modif(nx,ny); /*modifica coordonatle virfurilor paralelogramului curent cu  
    valorile nx si ny */  
    afiseaza(); /* afiseaza paralelogramul in pozitia noua */  
}
```

Funcțiile *sterge*, *modif* și *afiseaza*, apelate în corpul funcției *paralelogram* sunt funcții membru ale clasei *paralelogram*.

În mod analog, definim o funcție membru *deplasare* pentru clasa *dreptunghi*:

```
void dreptunghi::deplasare(int nx,int ny)  
{  
    sterge(); // sterge de pe ecran dreptunghiul curent  
    modif(nx,ny); /* modifica coordonatle virfurilor dreptunghiului  
    curent cu valorile nx si ny */
```

```
afiseaza(); // afiseaza dreptunghiul in pozitia noua
```

}

În acest caz se apeleză funcțiile membru *sterge*, *modif* și *afiseaza* ale clasei *dreptunghi*.

Pentru celelalte clase (*patrat* și *romb*) se va defini la fel funcția *deplasare*.

Se observă că funcția *deplasare* diferă de la o clasă la alta numai prin antet și prin aceea că funcțiile pe care le apeleză sunt funcții membru ale aceleiasi clase și funcția *deplasare*.

Se pune problema dacă nu se pot înlocui aceste funcții cu una singură și anume cu funcția membru a clasei *paralelogram* care se află în virful ierarhiei.

În principiu, ea se poate aplica pentru obiecte ale claselor derivate, deoarece un astfel de obiect este totdeauna un *paralelogram* (orice dreptunghi, patrat sau romb este un paralelogram).

O funcție membru a unei clase de bază se poate aplica la obiecte ale claselor derivate deoarece acestea o moștenesc.

Problema apare la apelurile funcțiilor *sterge*, *modif* și *afiseaza*, care aşa cum s-a afirmat mai sus, sunt funcții membru ale clasei *paralelogram*. Aceste funcții nu se pot utiliza dacă funcția *deplasare* este apelată pentru un alt obiect decât cel al clasei *paralelogram*, deoarece ele diferă de la o figură geometrică la alta.

Funcțiile membru *sterge*, *modif* și *afiseaza* se selectează la compilare, cind se compilează fiecare funcție membru *deplasare*. De aceea, deși la execuție, funcția membru *deplasare* a clasei *paralelogram* se poate apela pentru un obiect al oricărei clase a ierarhiei, ea se execută corect numai pentru obiectele clasei *paralelogram*. În felul acesta, se pare că moștenirea funcției membru *deplasare* a clasei *paralelogram* pentru clasele derivate din clasa *paralelogram* devine un inconveniut și nu permite deplasarea dreptunghiurilor, a patratelor și a romburilor.

Problema se poate rezolva dacă selectarea funcțiilor care să se apeleze în corpul funcției *deplasare* (*sterge*, *modif* și *afiseaza*) nu s-ar face la compilarea funcției *deplasare*, ci la execuție, la fiecare apel al funcției *deplasare*. În acest scop, s-a introdus noțiunea de funcție *virtuală*.

O funcție virtuală este o funcție membru a unei clase a cărui prototip și antet încep prin cuvântul *virtual*. În cazul exemplului de mai sus, funcțiile *sterge*, *modif* și *afiseaza* se vor defini ca funcții membru virtuale pentru fiecare clasă din ierarhie.

Ca urmare a acestui fapt, apelul unei astfel de funcții nu este rezolvat la compilare, ci la execuție, selectându-se versiunea funcției respective care corespunde obiectului curent. Se obișnuiește să se spună că apelul unei astfel de funcții se realizează folosind *legătura întârziată* (*late binding*) de obiect. Legăturile obișnuite dintre obiect și apelul unei funcții membru se rezolvă la compilare și despre ele se spune că sunt *legături timpurii*. Ele se mai numesc și *legături statice*, iar cele întârziate se mai numesc și *legături dinamice*.

În concluzie, definind funcțiile *sterge*, *modif* și *afiseaza* ca funcții virtuale,

vom defini o singură funcție membru *deplasare* pentru clasa *paralelogram*. Ea se poate aplica pentru un obiect al oricărei clase din ierarhie. La execuția funcției *deplasare* se vor apela acele funcții *sterge*, *modif* și *afiseaza* care sunt funcții membru ale acelei clase la care aparține obiectul curent.

În încheierea acestui capitol amintim că programarea orientată spre obiecte admite și alte concepte decât moștenirea și polimorfismul. Astfel, un concept prezent în limbajele anterioare limbajului C++ și care este suportat de programarea orientată spre obiecte este *identitatea obiectuală* (un exemplu de astfel de limbaj este Smalltalk).

Alte concepte, cum este de exemplu *persistența obiectelor*, sunt suportate de extensii ale limbajului C++. Astfel de extensii sunt limbajele E și O (vezi [6]).

## 29. CLASE DERIVATE ȘI CLASE DE BAZĂ

În capitolul 28 s-a arătat că între diferite concepte utilizate în programare se pot face ierarhizări. La baza ierarhizărilor se află procesul de moștenire, conform căruia un concept, diferit de cel din virful ierarhiei, are toate proprietățile conceptelor care îl precedă în ierarhia respectivă.

Ierarhia dintre concepte induce o ierarhie între clasele care implementează concepțile respective. În cazul claselor, procesul de moștenire înseamnă că o clasă diferită de cea din virful ierarhiei are ca elemente membru, elementele membru ale claselor care o precedă, precum și eventuale alte elemente membru specifice clasei respective. Elementele membru ale clasei aflate în virful ierarhiei sunt elemente comune pentru toate clasele ierarhiei respective.

Ierarhizarea claselor se face cu ajutorul noțiunii de *clasă derivată*.

Fie *cl1* și *cl2* două clase ale unei ierarhii de clase, *cl2* fiind situată pe un nivel imediat inferior nivelului clasei *cl1*. Atunci clasa *cl2* se spune că este o *clasă derivată* a clasei *cl1*, iar *cl1* este o *clasă de bază* a clasei *cl2*. În acest caz, vom spune că clasa *cl2* moștenește clasa *cl1*.

Relația de derivare între două clase se exprimă la definirea clasei derive. În acest scop, după numele clasei derive, se scrie lista claselor de bază pentru clasa respectivă folosind ca separator două puncte. În felul acesta, o clasă derivată se definește ca mai jos:

```
class nume_clasa_derivata: lista_claselor_de_baza {  
    // elemente specifice clasei derive  
};
```

Lista aflată după două puncte indică clasele de bază pentru clasa derivată care se definește. Lista respectivă conține, pentru fiecare clasă de bază, numele clasei care eventual este precedat de un modificador de protecție. Ca modificatori de protecție se pot folosi modifikatorii *public* și *private*. Ca separator între clasele de bază se folosesc virgula.

**Exemple:**

1. `class cl2:cl1 { ... };`  
Clasa *cl2* este o clasă derivată a clasei *cl1*. Clasa *cl1* este singura clasă de bază a clasei *cl2*.
2. `class cl2:private cl1 { ... };`  
Clasa *cl2* este o clasă derivată a clasei *cl1*. Clasa *cl1* este singura clasă de bază a clasei *cl2*.
3. `class cl1,public:cl2 { ... };`  
Clasa *cl1* este o clasă derivată a claselor *cl1* și *cl2*.

Relația de derivare se poate defini și pentru structuri. Aceasta înseamnă că exemplele de mai sus rămân valabile și în cazul în care cuvintul *class* se schimbă cu *struct*. Deci, o clasă de tip *struct* poate fi atât o clasă derivată, cât și o clasă de bază. Nu același lucru se poate afirma despre o clasă de tip *union*. O clasă de tip *union* nu poate fi nici clasă derivată și nici clasă de bază.

În mod implicit, modificatorul de protecție este *private* la definirea unei clase derive și *public* la definirea unei structuri derive. De aici rezultă că exemplele 1 și 2, de mai sus, sunt identice deoarece modificatorul *private* este implicit pentru clase.

Modificatorii de protecție utilizați în *lista\_claselor\_de\_bază* definesc protecția în clasa derivată a elementelor moștenite.

Tabelul de mai jos indică accesul în clasa derivată a elementelor moștenite în funcție de protecția fiecărui element moștenit și de modificatorul de protecție utilizat în *lista\_claselor\_de\_bază*.

Accesul în clasa de bază	Modificatorul de protecție din <i>lista_claselor_de_baza</i>	Accesul în clasa derivată a elementului moștenit
private	private	inaccesibil
protected	private	private
public	private	private
private	public	inaccesibil
protected	public	protected
public	public	public

Din acest tabel se observă că o clasă derivată nu are acces la elementele clasei de bază care au protecția *private*. În schimb, clasa derivată are acces la elementele clasei de bază ce au protecția *protected* sau *public*.

Dacă la definirea clasei derive se utilizează modificatorul de protecție *private*, atunci elementele protejate prin *protected* sau *public* devin protejate prin protecția *private*. Aceasta înseamnă că ele nu mai pot fi accesate de către o clasă derivată a clasei derive în acest fel. De aceea, modificatorul *private* se va folosi în *lista\_claselor\_de\_bază* numai dacă clasa derivată nu se utilizează ca o clasă de bază pentru o nouă derivare.

De obicei, o ierarhie de clase nu este o ierarhie finală, ea putind fi dezvoltată adăugind clase noi, care derivă din clasele terminale. Aceasta este posibil dacă la definirea claselor derive se utilizează, în *lista\_claselor\_de\_bază*, modificatorul *public* în loc de *private*. În acest caz, elementele protejate prin *protected* și *public* se moștenesc în clasa derivată prin aceeași protecție. De aceea, modificatorul de protecție cel mai des utilizat în *lista\_claselor\_de\_baza* este *public*.

Relația dintre clasa derivată și clasele ei de bază ridică unele probleme pe care le vom analiza în continuare.

O primă problemă este aceea a relațiilor dintre *constructorii și destructorii* clasei derive și ai claselor de bază. O a două problemă este legată de *conversiile* obiectelor claselor derive spre cele ale claselor de bază, precum și a pointerilor spre astfel de obiecte.

În sfîrșit, o a treia problemă este legată de *redeclararea* datelor membru și *supraîncarcarea* funcțiilor membru ale claselor de bază în clasa derive.

## 29.1. Relația dintre constructorii și destructorii claselor de bază și ai clasei derive

Constructorii și destructorii sunt funcții membru care nu se moștenesc.

La instanțierea unui obiect al unei clase derive se apelează atât constructorii clasei derive cât și cei ai claselor de bază. Întâi se apelează constructorii claselor de bază și la urmă se apelează și constructorul clasei derive. Ordinea de apel a constructorilor claselor de bază corespunde cu ordinea în care sunt indicate clasele respective în *lista\_claselor\_de\_baza* din definiția clasei derive.

La distrugerea unui obiect al clasei derive, destructorii se apelează în ordine inversă. Deci, întâi se apelează destructorul clasei derive și apoi se apelează destructorii claselor de bază în ordinea inversă față de ordinea de apel a constructorilor la instanțierea obiectului respectiv.

La instanțierea unui obiect se transmit valori parametrilor constructorilor pentru inițializarea datelor membru. În cazul unui obiect al unei clase derive, o parte din valori se folosesc pentru inițializarea datelor membru specific ale clasei derive, iar restul pentru inițializarea datelor membru ale claselor de bază.

Constructorul clasei derive conține parametri pentru toate valorile care se utilizează la inițializarea obiectelor.

Transferul valorilor de inițializare pentru datele membru ale claselor de bază se definește cu ajutorul antetului constructorului clasei derive.

Fie A o clasă derivată din clasele de bază B1,B2,...,Bn:

```
class A:public B1,public B2,...,public Bn {
    ...
    A(...); // prototip pentru constructor
    ...
}
Constructorul A are un antet de forma:
A::A(...):B1(...),B2(...),...,Bn(...)
```

Constructorul A are o listă de parametri completă, adică pentru inițializarea tuturor datelor membru ale unui obiect de tip A.

Expresia Bi(...) (i = 1,2,...,n) din antetul constructorului A conține, în paranteză, o listă de expresii care definesc valori inițiale pentru datele membru ale clasei Bi.

Dacă o clasă Bi nu are constructor, atunci pentru clasa respectivă nu va fi prezentă o expresie de forma Bi(...) în antetul constructorului A. De asemenea, expresia respectivă nu va fi prezentă în antetul lui A dacă datele membru ale clasei Bi se inițializează cu valori implicate sau clasa Bi nu are date membru.

Expresia Bi(...) este un apel explicit al unui constructor al clasei Bi.

Ordinea în care se scriu expresiile Bi(...) în antetul constructorului A este arbitrară. Constructorii se apeleză totdeauna în ordinea în care sunt scrise clasele în *lista\_claselor\_de\_bază*.

O condiție esențială pentru a se putea apela constructorii claselor de bază prin intermediul clasei derivate este ca aceștia să aibă protecția de tip *protected* sau *public*.

Lista care definește apelurile constructorilor claselor de bază nu este prezentă în prototipurile constructorilor claselor derivate ci numai în antetele acestora.

În general, clasele au constructori definiți de programator. Totuși există situații în care o clasă nu are constructori definiți de programator. În acest caz, compilatorul generează în mod automat un constructor implicit (fără parametri) pentru clasa respectivă. Având în vedere această situație rezultă că teoretic pot apărea următoarele cazuri:

1. Clasa derivată și clasele de bază ale ei nu au constructori definiți de programator.
2. Cel puțin o clasă dintre clasele de bază are constructori, iar clasa derivată nu are constructori.
3. Clasa derivată are constructori iar clasele de bază ale ei nu au constructori definiți de programator.
4. Clasa derivată are constructori și cel puțin o clasă de bază a ei are constructori.

În primul caz, la instantierea obiectelor clasei derivate se aplică constructorii implicați generați de compilator.

În cazul al doilea, toți constructorii claselor de bază definiți de programator trebuie să fie implicați sau să aibă toți parametri implicați. La instantierea obiectelor clasei derivate nu se pot face inițializări. La o astfel de instanțiere se aplică constructorul implicit al clasei derivate, generat de compilator și constructorii claselor de bază definiți de programator, folosindu-se valorile implicate ale parametrilor acestora.

În cazul al treilea, constructorul clasei derivate realizează toate inițializările atât pentru datele membru specifice clasei derivate cât și pentru datele membru ale claselor de bază. De aceea, în acest caz este necesar ca clasa derivată să aibă acces la toate datele membru ale claselor de bază care comportă inițializări.

Constructorul clasei derivate are parametri corespunzători pentru datele care comportă inițializări. Valorile lor se atribuie datelor membru în corpul constructorului. Evident, antetul constructorului este format numai din numele lui urmat de lista parametrilor formali, inclusă în paranteze rotunde.

În cazul al patrulea, constructorul clasei derivate conține parametri pentru toate datele membru care comportă inițializări. Valorile parametrilor corespunzători datelor membru specifice clasei derivate se utilizează pentru a inițializa datele respective în mod obișnuit (de obicei în corpul constructorului clasei derivate se inițializează aceste date), iar celelalte valori se utilizează la apelul

explicit al constructorilor claselor de bază.

Apelurile explicite ale constructorilor sunt situate în antetul constructorului clasei derivate, așa cum s-a indicat mai sus.

#### Exemple:

1. Clasa A este o clasă derivată a clasei de bază B. Nici una din clase nu are constructor.

Fie instantierea:

A a;

Obiectul a se instantiază fără a se face inițializări. La instantiere se aplică constructorii implicați ai claselor generați de compilator.

2. Clasa A este o clasă derivată a clasei de bază B.

Clasa B are constructor cu toți parametrii implicați:

```
class B {  
    protected:  
        double x;  
        double y;  
    public:  
        B(double p=0,double q=0)  
        {  
            x = p;  
            y = q;  
        }  
        ...  
};
```

Clasa A nu are constructor:

```
class A:public B{  
    protected:  
        Boolean ecran;  
    public:  
        interior();  
        ...  
};
```

Fie declarația:

A a;

Obiectul a se instantiază apelindu-se constructorul cu parametri implicați al clasei B.

Datele membru x și y se inițializează cu valoarea 0, iar data membru ecran nu este inițializată.

3. Clasa A este o clasă derivată a clasei B. Clasa B nu are constructor definit de utilizator și se definește ca mai jos:

```
class B {  
    protected:  
        double x;  
        double y;  
    public:  
        ... // nu există constructori
```

```

};

Clasa A are un constructor:
class A:public B {
protected:
    Boolean ecran;
    int abs,ord;
public:
    A(double p=0,double q=0,int lx=639,int ly=349)
    {
        // initializeaza datele membru ale clasei de baza
        x=p;
        y=q;

        // initializeaza datele membru specific
        abs=lx;
        ord=ly;

        if(0 <= p && p <= lx && 0 <= q && q <= ly)
            ecran = true;
        else ecran = false;
    }
}

Exemple de instanțieri:
A a; /* a.x=0, a.y=0, a.abs=639, a.ord=349, a.ecran=true */
A b(100,400); /* b.x=100, b.y=400, b.abs=639, b.ord=349, b.ecran = false */

```

4. Clasa A este o clasă derivată a clasei B și ambele clase au constructori definiți de programator.

```

class B {
protected:
    double x,y;
public:
    B(double xx=0,double yy=0)
    {
        x=xx;
        y=yy;
    }
}

class A:public B {
protected:
    Boolean ecran;
    int abs,ord;
public:
    A(double p=0,double q=0,int lx=639,int ly=349):B(p,q)
    {
        abs=lx;
        ord=ly;
        if(0 <= p && p <= lx && 0 <= q && q <= ly)
            ecran = true;
        else ecran = false;
    }
}

```

```

}

...
};

Se pot utiliza aceleași instanțieri ca în exemplul de la punctul 3. Se realizează aceleași inițializări ca la punctul 3.

```

Amintim că datele *abs* și *ord* pot fi inițializate chiar în antetul constructorului, ca mai jos:

```

A(double p=0,double q=0,int lx=639,
    int ly=349):B(p,q),abs(lx),ord(ly)
{
    if(0 <= p && p <= lx && 0 <= q && q <= ly)
        ecran = true;
    else ecran = false;
}
...
};

```

O clasă derivată este necesar să aibă cel puțin un constructor în cazul în care cel puțin o clasă de bază are un constructor care nu este implicit sau nu are toți parametrii implicați.

Punctele 1 - 4 de mai sus, au fost analizate avindu-se în vedere constructorii obișnuiți, adică constructorii care nu sunt de copiere.

În lipsa constructorilor de copiere, compilatorul generează în mod automat constructori de copiere implicați care, atunci cind se aplică, copiază datele membru ale obiectului sursă (care se copiază) în datele membru corespunzătoare ale obiectului destinație (care se crează). O copiere de acest fel a fost numită *copiere bit cu bit*.

Copierea bit cu bit este adesea suficientă și atunci nu este nevoie să se definească constructori de copiere. Există însă cazuri cind copierea bit cu bit nu este suficientă și atunci este nevoie de astfel de constructori. Ei se utilizează atât la inițializarea unui obiect printr-un alt obiect, cât și la transferul prin parametri ai obiectelor și la returnarea acestora la revenirea din funcții.

Constructorii de copiere sunt necesari mai ales atunci cind clasele au ca date membru pointeri spre zone alocate dinamic în memoria *heap*. De asemenea, în condițiile moștenirii unor date membru ale claselor de bază, pot apărea situații în care copierea bit cu bit poate conduce la erori.

În legătură cu apelul constructorilor de copiere vom distinge și în acest caz mai multe situații.

Prima situație, cind nici clasa derivată și nici clasele de bază nu au constructori de copiere, a fost deja amintită mai sus și anume, în acest caz, se face o copiere bit cu bit apelindu-se constructorii de copiere implicați generați de compilator.

Presupunem acum situația în care clasa derivată nu are constructor de copiere, în schimb clasele de bază, o parte din ele sau toate, au constructori de copiere. Dacă este nevoie de un constructor de copiere pentru un obiect al clasei derivate (inițializare prin copiere, transfer de obiecte prin parametri sau obiect returnat de

o funcție), atunci se apelează constructorul de copiere implicit al clasei deriveate. Acesta realizează o copiere bit cu bit a datelor specifice clasei deriveate și a datelor membru moștenite de la clasele de bază care nu au constructori de copiere, iar celelalte date membru se initializează apelindu-se automat constructorii de copiere ai claselor de bază corespunzătoare.

O altă situație este aceea în care clasa derivată are constructor de copiere. În acest caz, se apelează constructorul de copiere al clasei deriveate, cind este nevoie de un astfel de constructor (vezi mai sus), iar utilizarea constructorilor claselor de bază este în funcție de definiția constructorului de copiere al clasei deriveate.

O primă regulă este aceea că, apelul constructorului de copiere al clasei deriveate nu conduce la apelul automat al constructorilor de copiere ai claselor de bază.

Fie A o clasă derivată și B o clasă de bază a ei. Să presupunem că A are un constructor de copiere de antet:

```
A(const A& a)
```

În acest caz, clasa de bază B, fie că nu are nici un constructor definit de programator, fie că are un constructor definit de programator care nu are parametri sau are numai parametri implicați.

La o instanțiere de forma:

```
A b=a;
```

unde a este un obiect al clasei A în prealabil declarat, se apelează constructorul de copiere al clasei A cu antetul indicat mai sus. Apelul constructorului de copiere al clasei A conduce în mod automat la apelul constructorului implicit al clasei B (cel generat de compilator sau cel definit de programator dacă există un astfel de constructor) sau, în caz contrar, la apelul constructorului clasei B care are toți parametrii implicați.

Dacă clasa de bază B are constructori dar nici unul nu este implicit sau cu toți parametri implicați, atunci constructorul de copiere al clasei deriveate A trebuie să conțină apelul explicit al unui constructor al clasei B. Cu alte cuvinte, constructorul de copiere al clasei A va avea antetul:

```
A(const A& a):B(...)
```

### Exerciții:

29.1 Să se implementeze tipul abstract *punct* pentru prelucrarea punctelor din plan.

Clasa *punct* are două date membru care definesc poziția punctelor în plan:

*x* - abscisa

și

*y* - ordonata

Protecția acestor date este de tip *protected* pentru a putea fi moștenite prin derivare.

Clasa are un constructor cu parametrii implicați, valorile implicate fiind zero.

Funcțiile membru ale clasei *punct* sunt:

- *citpct* - citește coordonatele unui punct;
- *afispct* - afișează coordonatele unui punct;
- supraincărcarea operatorilor +, - și == pentru obiecte de tip punct;
- supraincărcarea produsului unui obiect cu un număr;
- supraincărcarea împărțirii unui obiect la un număr;
- *dist* - distanța dintre două puncte;
- *retabs* - returnează abscisa punctului curent;
- *retord* - returnează ordonata punctului curent.

### FIȘIERUL BXXIX1.H

```
class punct {
protected:
    double x0; // abscisa
    double y0; // ordonata

    static int rcit_double (const char *text,double& d);
    /* - afiseaza text;
     * - citeste un numar si-l pastreaza in zona referita de d;
     * - returneaza:
     *   0 - la sfirsitul de fisier;
     *   1 - altfel.
    */

public:
    punct(double abs = 0,double ord=0); // constructor

    int citpct(const char *text);
    /* - afiseaza sirul spre care pointeaza text daca nu este vid;
     * - citeste coordonatele obiectului curent;
     * - returneaza:
     *   0 - la sfirsitul de fisier;
     *   1 - altfel.
    */

    void afispct(const char *f) const;
    /* afiseaza coordonatele obiectului curent folosind formatul f sau
     * un format standard daca f pointeaza spre sirul vid */

    friend punct operator +(punct p1,punct p2); // returneaza p1+p2
    friend punct operator -(punct p1,punct p2);
    // returneaza p1-p2

    punct operator *(double d);
    // returneaza p*d; p este obiectul curent

    friend punct operator *(double d, punct p2);
    // returneaza d*p2
```

```

punct operator /(double d);
// returnaza p/d; p este obiectul curent

double dist(punct p2) const;
/* returnaza distanta dintre punctele p si p2; p este obiectul curent */

double retabs() const;
// returnaza abscisa obiectului curent

double retord() const;
// returnaza ordonata obiectului curent

friend int operator == (punct p1,punct p2);
// returnaza 1 daca p1 si p2 coincid si 0 altfel

punct operator -();
// returnaza simetricul fata de origine al obiectului curent
};

Funcțiile membru se definesc în fișierul de extensie CPP, de mai jos.

```

## FIȘIERUL BXXIX1

```

#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __PUNCT_H
#include "BXXIX1.H"
#define __PUNCT_H
#endif

int punct::rcit_double(const char *text,double& d)
/* - afisaza sirul spre care pointaza text;
   - citeste un numar si-l pastreaza in zona referita de d;
   - returnaza:
     0 - la sfirsit de fisier;
     1 - altfel.
*/
{
    char t[255];
    double x;

    for(;;) {
        printf(text);
        if(gets(t)==0) return 0; //EOF
        if(sscanf(t,"%lf",&x)==1) break;
        printf("nu s-a tastat un numar\n");
    }
    d=x;
}

```

```

return 1;
} // sfirsit rcit_double

inline punct::punct(double abs,double ord) //constructor
{
    x0=abs;
    y0=ord;
} // sfirsit constructor

int punct::citpct(const char *text)
/* - afisaza sirul spre care pointaza text daca nu este vid;
   - citeste coordonatele obiectului curent;
   - returnaza:
     0 - la sfirsit de fisier;
     1 - altfel.
*/
{
    if(text && *text) printf(text);
    if(rcit_double("\nabscisa=",x0)==0) return 0;
    if(rcit_double("\nordonata=",y0)==0) return 0;
    return 1;
} // sfirsit citpct

void punct::afispct(const char *f) const
/* afisaza coordonatele obiectului curent folosind formatul f sau un
   format standard daca f pointeaza spre sirul vid */
{
    if(f && *f) printf(f,x0,y0);
    else printf("x0=%g\ty0=%g\n",x0,y0);
} // sfirsit afispct

punct operator +(punct p1,punct p2) // returnaza p1+p2
{
    punct r;
    r.x0 = p1.x0+p2.x0;
    r.y0 = p1.y0+p2.y0;
    return r;
} // sfirsit operator +

punct punct::operator -() // returnaza simetricul fata de origine al obiectului curent
{
    punct r;
    r.x0 = -x0;
    r.y0 = -y0;
    return r;
} // sfirsit operator - unar

punct operator -(punct p1,punct p2) // returnaza p1-p2
{
    return p1 + -p2;
} // sfirsit operator - binar

```

```

punct punct::operator *(double d) // returneaza p*d; p este obiectul curent
{
    punct r;
    r.x0 = x0*d;
    r.y0 = y0*d;
    return r;
} // sfîrșit operator *(double)

punct operator * (double d,punct p2) // returneaza d*p2
{
    return p2*d;
} // sfîrșit operator *(double,punct)

punct punct::operator /(double d)
// returneaza p/d; p este obiectul curent;
// returneaza originea axelor daca d=0
{
    punct r; // x0=y0=0
    if(d==0) return r;
    r.x0=x0/d;
    r.y0=y0/d;
    return r;
} // sfîrșit operator /

inline double punct::dist(punct p2) const
/* returneaza distanta dintre punctele p si p2; p este obiectul curent */
{
    return sqrt((x0-p2.x0)*(x0-p2.x0)+(y0-p2.y0)*(y0-p2.y0));
} // sfîrșit dist

inline double punct::retabs() const // returneaza abscisa obiectului curent
{
    return x0;
} // sfîrșit retabs

inline double punct::retord() const // returneaza ordonata obiectului curent
{
    return y0;
} // sfîrșit retord

inline int operator == (punct p1,punct p2)
/* returneaza:
   1 - daca p1 si p2 coincid;
   0 - altfel.
*/
{
    return p1.x0==p2.x0 && p1.y0==p2.y0;
} // sfîrșit operator ==

```

29.2 Să se implementeze tipul abstract *gpunct* care pe lingă elementele membru ale clasei *punct*, definită în exercițiul 29.1, conține elemente necesare pentru a afișa un punct în mod grafic pe ecran.

Clasa *gpunct* este o clasă derivată a clasei *punct* definită în exercițiul precedent.

Clasa *gpunct* are următoarele elemente specifice:

- date membru:
  - *culoare* - variabilă de tip *int* care definește culoarea pentru afișarea punctului (pentru detalii se va revedea capitolul 19);
  - *vizibil* - are valoarea 1 dacă punctul este afișat pe ecran și zero altfel.
- funcții membru:
  - *afiseaza* - afișează punctul în mod grafic pe ecran;
  - *sterge* - șterge punctul de pe ecran (culoarea lui devine aceeași cu a fondului);
  - constructor pentru inițializarea elementelor membru ale unui punct; toți parametri sunt implicați;
  - constructor cu doi parametri: un parametru este de tip *punct*, iar celălalt definește culoarea; acesta din urmă este implicit.

## FIȘIERUL BXXIX2.H

```

#ifndef __PUNCT_H
#include "BXXIX1.CPP"
#define __PUNCT_H
#endif
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif

class gpunct:public punct {
protected:
    int culoare;
    int vizibil;
public:
    gpunct(double abs=0,double ord=0,int c=WHITE);
    /* constructor; în mod implicit punctul are coordonatele zero, culoarea este definită cu ajutorul constantei simbolice WHITE(alb) și nu este afișat la instantiere */

    gpunct(const punct& p,int c=WHITE);
    /* constructor utilizat la inițializarea obiectului curent folosind un obiect de tip punct existent */

    void afiseaza();
    // afiseaza obiectul curent conform atributelor sale

    void sterge();
    // face punctul invizibil
};

Funcțiile membru ale clasei gpunct se definesc în fișierul de mai jos, de extensie CPP.

```

## FIŞIERUL BXXIX2

```
#ifndef __GPUNCT_H
#include "BXXIX2.H"
#define __GPUNCT_H
#endif

inline gpunct::gpunct(double abs,double ord,int c):
    punct(abs,ord)
// implicit x0=y0=0 si c=WHITE
{
    culoare=c;
    vizibil=0; // la instantiere punctul nu este vizibil
} // sfirsit constructor

inline void gpunct::afiseaza() // afiseaza punctul curent
{
    putpixel(x0,y0,culoare);
    vizibil=1;
} // sfirsit afiseaza

inline void gpunct::sterge() // punctul devine invizibil
{
    // afiseaza punctul cu culoarea de fond curenta
    putpixel(x0,y0,getbkcolor());
    vizibil=0;
}

gpunct::gpunct(const punct& p,int c):
    punct(p.retabs(),p.retord())
// obiectul curent se initializaaza cu coordonatele obiectului p
{
    culoare = c;
    vizibil = 0;
}
```

- 29.3 Să se scrie un program care citește coordonatele a două puncte și le afișează împreună cu mijlocul segmentului format de cele două puncte. Punctele citite se afișează colorate cu culoarea implicită, iar punctul din mijlocul segmentului se afișează cu culoarea definită de constanta simbolică LIGHTGREEN.

## PROGRAMUL BXXIX3

```
#include <conio.h>
#include <stdlib.h>
#include "BXXIX2.CPP"

main()
/* - citeste coordonatele a doua puncte;
 - afiseaza punctele respective de culoare implicita impreuna cu punctul din mijlocul segmentului format de punctele citite.
 Punctul din mijlocul segmentului se afiseaza cu culoarea definita de constanta simbolica
```

```
LIGHTGREEN. */

gpunct a,b;

if(a.citpct("punctul a")==0) {
    printf("s-a citit EOF\n");
    exit(1);
}
if(b.citpct("punctul b")==0) {
    printf("s-a citit EOF\n");
    exit(1);
}
gpunct c((a+b)/2, LIGHTGREEN);
int gd=DETECT,gm;
initgraph(&gd,&gm, "c:\\borlandc\\bgi");
a.afiseaza();
b.afiseaza();
c.afiseaza();
getch();
closegraph();
}
```

### Observații:

1. Funcția *citpct* se poate apela cu obiecte de tip *gpunct* deoarece ea este moștenită de la clasa *punct* care este clasa de bază pentru *gpunct*.
  2. Supraincărările operatorilor pentru obiectele de tip *punct* se moștenesc pentru clasele derivate și ele se pot aplica și pentru obiecte derivate. Din această cauză, expresia  $(a+b)/2$  este corectă, *a* și *b* fiind instanțieri ale clasei *gpunct* care este o clasă derivată a clasei *punct*.  
Rezultatul expresiei de mai sus este un obiect de tip *punct*.
- 29.4 Să se implementeze tipul abstract *gtext* pentru afișarea de texte pe ecranul setat în mod grafic.

Un text se poate afișa în mod grafic folosind funcția *outtextxy* din biblioteca standard a limbajelor C și C++.

Ea este de prototip:

```
void far outtextxy(int x,int y,char far *sir);
```

unde:

*x si y*

- Definesc coordonatele de inceput ale cimpului în care se afișează textul spre care pointează *sir*.

Pentru a afișa un text în mod grafic este necesar să definim anumite caracteristici și anume:

- setul de caractere;
- dimensiunea caracterelor;
- cadrul caracterelor;
- direcția de scriere a caracterelor;

- culoarea caracterelor.

Aceste caracteristici, exceptând culoarea, se definesc cu ajutorul a două funcții: *settextstyle* și *settextjustify* (pentru detalii se poate consulta paragraful 19.4).

Prima funcție are prototipul:

```
void far settextstyle(int font,int direction,int charsize);
```

Cea de a doua funcție are prototipul:

```
void far settextjustify(int oriz,int vert);
```

Constructorul clasei *gtext* definește valorile pentru parametri acestor funcții.

Funcțiile respective se apelează folosind aceste valori înainte de a afișa textul cu ajutorul funcției *outtextxy*.

Întrucit după afișarea textului pe ecran este nevoie să se revină la caracteristicile vechi pentru afișarea textelor, înainte de a apela funcțiile *settextstyle* și *settextjustify* se vor salva caracteristicile curente cu ajutorul funcției *gettextsettings* de prototip:

```
void far gettextsettings(struct textsettingstype far *textinfo);
```

unde tipul utilizator *textsettingstype* se definește în fișierul *graphics.h* astfel:

```
struct textsettingstype {
    int font;
    int direction;
    int charsize;
    int horiz;
    int vert;
};
```

De aici rezultă că pentru a afișa un text se vor urma pașii:

1. Se memorează caracteristicile curente apelind funcția *gettextsettings*.
2. Se apelează funcțiile *settextstyle* și *settextjustify* pentru a seta caracteristicile obiectului curent.
3. Se afișează textul apelind funcția *outtextxy*.
4. Se refac caracteristicile păstrate la punctul 1 prin apelul funcțiilor *settextstyle* și *settextjustify*.

În mod implicit, se folosește culoarea definită de constanta simbolică WHITE pentru afișarea caracterelor textului.

Se pot afișa texte colorate cu alte culori indicind o valoare corespunzătoare pentru parametrul *culoare* al constructorului. Pentru gestiunea culorilor se folosesc funcțiile *setcolor* și *getcolor* (vezi paragraful 19.2).

#### FIȘIERUL BXXIX4.H

```
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
```

```
#endif

class gtext {
protected:
    int font;
    int directie;
    int dimensiune_caracter;
    int cadrat_oriz;
    int cadrat_vert;
    int culoare_text;
    char text[11];
    int vizibil_text;
public:
    gtext(); // constructor implicit

    gtext(char *sir,
          int culoare = WHITE,
          int f = DEFAULT_FONT,
          int dir = HORIZ_DIR,
          int dim_car = 1,
          int cadrat_o = CENTER_TEXT,
          int cadrat_v = CENTER_TEXT);

    void afisgtext(int x,int y);
    /* afiseaza textul definit de obiectul curent intr-un cimp care
       incepe in punctul de coordonate (x,y) */

    void stergegtext(int x,int y);
    /* face textul invizibil; textul este afisat in cimpul care incepe
       in punctul de coordonate (x,y) */
};
```

Funcțiile membru ale clasei *gtext* se definesc în fișierul de mai jos, de extensie CPP.

#### FIŞIERUL BXXIX4

```
#ifndef __GTEXT_H
#include "BXXIX4.H"
#define __GTEXT_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif

gtext::gtext()
/* - constructor implicit;
   - toate datele membru au valori implice;
   - textul contine sirul vid. */
{
    text[0] = '\0';
    culoare_text = WHITE;
    font = DEFAULT_FONT;
```

```

directie = _HORIZ_DIR;
dimensiune_caracter = 1;
cadraj_oriz = CENTER_TEXT;
cadraj_vert = CENTER_TEXT;
vizibil_text = 0;
}

gtext::gtext(char *sir,int culoare,int f,int dir,
            int dim_car,int cadraj_o,int cadraj_v)
/* constructor cu parametrii impliciti in afara de primul */
{
    strcpy(text,sir,11); /* copiaza cel mult primele 10 caractere ale sirului spre
                           care pointeaza sir*/
    text[10] = '\0'; /* se forteaza sfarsitul sirului de caractere */
    culoare_text = culoare;
    font = f;
    directie = dir;
    dimensiune_caracter = dim_car;
    cadraj_oriz = cadraj_o;
    cadraj_vert = cadraj_v;
    vizibil_text = 0; //textul nu este afisat la instantiere
}

void gtext::afisgtext(int x,int y)
/*afiseaza textul definit de obiectul curent intr-un cimp care incepe in punctul de
   coordonate (x,y)*/
{
    struct textsettingstype caract_crt;
    int culoare_crt;

    /* se salveaza caracteristicile curente */
    gettextsettings(&caract_crt);
    culoare_crt = getcolor();

    /* se seteaza caracteristicile obiectului */
    settextstyle(font,directie,dimensiune_caracter);
    settextjustify(cadraj_oriz,cadraj_vert);
    setcolor(culoare_text);

    /* afiseaza textul */
    outtextxy(x,y,text);

    /* se refac caracteristicile */
    settextstyle(caract_crt.font,
                caract_crt.direction,
                caract_crt.charsize);
    settextjustify(caract_crt.horiz,caract_crt.vert);
    setcolor(culoare_crt);
    vizibil_text = 1; //textul este vizibil
}

void gtext::stergegtext(int x,int y)
/* face invizibil textul care este afisat in cimpul care incepe in punctul de coordonate (x,y)*/
{
}

```

```

int culoare_temp = culoare_text;
// salvaza culoarea textului obiectului curent
culoare_text = getbkcolor();
// culoarea obiectului curent devine culoarea fondului

afisgtext(x,y); // textul devine invizibil
culoare_text = culoare_temp;
// se refac culoarea obiectului curent

vizibil_text = 0; // textul este invizibil
}

```

24.5 Să se implementeze tipul abstract *gtextpunct* care permite afişarea punctelor și a textelor în mod grafic.

Tipul *gtextpunct* se definește printr-o clasă derivată care are două clase de bază: *gpunct* și *gtext*. Clasa *gtextpunct* moștenește elementele membru ale ambelor clase.

#### FISIERUL BXXIX5.H

```

#ifndef __GTEXT_H
#include "BXXIX4.CPP"
#define __GTEXT_H
#endif
#ifndef __GPUNCT_H
#include "BXXIX2.CPP"
#define __GPUNCT_H
#endif

class gtextpunct:public gpunct,public gtext {
protected:
    int vizibil_punct_text;
public:
    gtextpunct(); //constructor implicit
    gtextpunct(char *sir,
               double x = 0,
               double y = 0,
               int c = WHITE,
               int culoare = WHITE,
               int f = DEFAULT_FONT,
               int dir = HORIZ_DIR,
               int dim_car = 1,
               int cadraj_o = CENTER_TEXT,
               int cadraj_v = CENTER_TEXT);

    void afisgtextpunct(int xt,int yt);
    /* afiseaza in mod grafic punctul si textul definit de obiectul curent */

    void stergegtextpunct(int xt,int yt);
    /* face invizibil punctul si textul definit de obiectul curent */
};

```

Functiile membru se definesc in fișierul de extensie *CPP*, de mai jos.

## FIŞIERUL BXXIX5

```
#ifndef __GTEXTPUNCT_H
#include "BXXIX5.H"
#define __GTEXTPUNCT_H
#endif

inline gtextpunct::gtextpunct()
/* - constructor implicit; textul contine sirul vid, iar celelalte date membru au valori implice;
 - se utilizeaza la instantierea obiectelor neinitializate.*/
{
    // obiectul nu se afiseaza la instantiere
    vizibil_punct_text = 0;
}

inline gtextpunct::gtextpunct(char *sir,double x,double y,
                           int c,int culoare,int f,int dir,
                           int dim_car,int cadraj_o,int cadraj_v):
    gtext(sir,culoare,f,dir,dim_car,cadraj_o,
          cadraj_v),gpunct(x,y,c)

// constructor pentru initializarea obiectelor la instantiere
{
    vizibil_punct_text = 0;
}

inline void gtextpunct::afisgtextpunct(int xt,int yt)
/* - afiseaza punctul si textul obiectului curent;
 - textul se afiseaza intr-o zona care incepe in punctul de coordonate (xt,yt). */
{
    afiseaza(); // afiseaza punctul
    afisgtext(xt,yt); // afiseaza textul
    vizibil_punct_text = 1;
}

inline void gtextpunct::stergegtextpunct(int xt,int yt)
/* face invizibil punctul si textul obiectului curent */
{
    sterge(); // face punctul invizibil
    stergetext(xt,yt); // face textul invizibil
    vizibil_punct_text = 0;
}
```

29.6 Să se modifice programul din exercițiul 29.3 în aşa fel încit împreună cu afişarea punctelor să se afișeze și literele care notează punctele respective.

Punctele se notează astfel:

- A - primul punct citit de la intrarea standard;
- B - al doilea punct citit de la intrarea standard;
- M - punctul aflat la mijlocul segmentului AB.

Literele care notează punctele se afișează deplasate față de punctele

respective după cum urmează:

- A - abscisa deplasată în stînga cu 10 pixeli;
- B - ordonata deplasată în jos cu 10 pixeli;
- M - abscisa deplasată în dreapta cu 10 pixeli;
- ordonata deplasată în sus cu 10 pixeli.

## PROGRAMUL BXXIX6

```
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIX5.CPP"

main()
/* - citeste 2 puncte de la intrarea standard;
 - afiseaza punctele respective si punctul situat la mijlocul segmentului
 determinat de ele împreună cu literele:
     A pentru primul punct;
     B pentru al doilea punct;
     M pentru punctul situat la mijlocul segmentului AB;
 - culori:
     - punctele A si B se afiseaza folosind culoarea implicită;
     - punctul M se afiseaza colorat cu culoarea definită de constanta simbolica LIGHTGREEN;
     - caracterele A si B se afiseaza folosind culoarea implicită;
     - caracterul M se afiseaza colorat cu culoarea definită de constanta simbolica YELLOW.
 */
{
    gtextpunct a("A");
    gtextpunct b("B");

    if(a.citpct("punctul A") == 0) {
        printf("s-a citit EOF\n");
        exit(1);
    }
    if(b.citpct("punctul B") == 0) {
        printf("s-a citit EOF\n");
        exit(1);
    }
    gpunct c=(a+b)/2;
    gtextpunct m("M",c.retabs(),c.retord(),LIGHTGREEN,YELLOW);
    int gd = DETECT,gm;
    initgraph(&gd,&gm,"c:\\borland\\bgi");
    a.afisgtextpunct(a.retabs()-10,a.retord());
    b.afisgtextpunct(b.retabs(),b.retord()+10);
    m.afisgtextpunct(m.retabs()+10,m.retord()-10);
    getch();
    closegraph();
}
```

## 29.2. Conversii

În capitolul 24 s-a discutat despre conversiile utilizator care permit efectuarea de conversii de date din tipuri predefinite în date de tipuri abstracte și invers, precum și conversia obiectelor de un tip abstract în obiecte de un alt tip abstract.

În principiu, conversiile dintr-un tip predefinit într-un tip abstract se realizează cu ajutorul constructorilor, iar conversia inversă cu ajutorul supraincărării operatorului *cast*. Conversia dintr-un tip abstract într-un alt tip abstract se poate realiza prin ambele metode.

Ambele metode se pot aplica atât implicit cât și explicit.

În paragraful de față ne referim la conversii care se pot realiza ținând seama de conceptul de derivare.

Fie A o clasă pentru care B este o clasă de bază. Un obiect care este o instantiere a clasei A se convertește în mod implicit într-un obiect al clasei B. Aceasta, deoarece obiectele clasei A sunt obiecte specializate ale clasei B.

Conversia inversă nu se poate realiza fără a fi definită un constructor sau prin supraincărarea operatorului *cast*.

Conceptul de moștenire impune reguli de conversie și pentru pointeri și referințe la obiecte ale claselor derivate și de bază.

Fie declarațiile:

```
A *pa;    A a;
B *pb;    B b;
```

unde:

B - Este o clasă de bază a clasei A.

Compilatorul C++ permite atribuiri de felul celor de mai jos:

```
pa = &a;
pb = &b;
```

Atribuirile de acest fel se realizează pe baza supraincărării隐式的 a operatorului adresă (& unar), care permite ca acesta să se aplique la o dată sau obiect de orice tip.

O atribuire de formă:

```
pb = &a;
```

este acceptată de compilator la fel ca și atribuirea obiectului *a* la obiectul *b*:

```
b = a;
```

În mod analog, atribuirea:

```
pb = pa;
```

este corectă.

În general, un pointer sau o referință la un obiect al unei clase derivate se poate atribui la un pointer sau o referință la un obiect al unei clase de bază a clasei derivate respective. Atribuirile de acest gen se realizează prin conversii implicite corespunzătoare. În schimb atribuirile:

```
pa = &b;
pa = pb;
```

sunt eronate. Ele pot fi acceptate de compilator dacă se utilizează expresii *cast* care realizează conversii explicite de pointeri:

```
pa = (A *) &b;
pa = (A *) pb;
```

Conversiile de acest fel sunt utile cind se creează și prelucră obiecte de tip *colecție*, colecția fiind un set de obiecte de diferite tipuri abstracte derivate dintr-un tip abstract de bază.

## 29.3. Redefinirea datelor membru ale unei clase de bază într-o clasă derivată

O dată membru a unei clase de bază se poate redefini ca dată membru a unei clase derivate.

Exemplu:

```
class B {
protected:
    double x;
    double y;
public:
    B(double xx=0,double yy=0)
    {
        x = xx;
        y = yy;
    }
    ...
};

class A:public B {
protected:
    double x;
    double y;
public:
    B(double dx=0,double dy=0,double bx=0,
       double by=0):B(bx,by)
    {
        x=dx; // x este data membru a clasei derivate
        y=dy; // y este data membru a clasei derivate
    }
    void afis() const;
    ...
};
```

};

Funcția membru *afis* se apelează pentru obiecte ale clasei derivate. Ea afișează datele membru ale clasei derivate, precum și datele membru moștenite de la clasa de bază B. Pentru a face distincție între datele membru ale clasei derivate și cele ale clasei de bază, se folosește operatorul de rezoluție. Astfel:

- B::x - Este dată membru a clasei B.  
x - Este dată membru a clasei A.

Putem defini funcția membru *afis* ca mai jos:

```
void A::afis() const
{
    printf("xbaza=%g\tybaza=%g\n",B::x,B::y);
    printf("xderivat=%g\tyderivat=%g\n",x,y);
}
```

## 29.4. Supraîncărcarea funcțiilor membru ale unei clase de bază într-o clasă derivată

Funcțiile membru ale claselor de bază sunt moștenite de clasele derivate. Astfel, dacă B este o clasă de bază pentru clasa A și *f* este o funcție membru a clasei B, atunci *f* este și o funcție membru a clasei A. Aceasta înseamnă că funcția poate fi utilizată atât cu obiecte ale clasei B, cât și cu obiecte ale clasei A:

```
A a;
B b;
```

*b.f(...)* și *a.f(...)* sunt apeluri corecte ale lui *f*.

În mod analog, funcțiile prieten ale clasei de bază sunt funcții prieten și pentru clasa derivată.

Moștenirea funcțiilor membru și prieten are ca efect faptul că supraîncărcarea operatorilor pentru clasa de bază este valabilă și pentru o clasă derivată a clasei de bază respective.

Cu toate acestea, nu totdeauna o funcție moștenită de la o clasă corespunde pentru a fi apelată cu obiecte ale unei clase derivate. În astfel de situații, funcția respectivă se va supraîncărca pentru clasa derivată. La supraîncărcarea pentru clasa derivată se poate păstra nu numai numele funcției, ci chiar și numărul și tipul parametrilor. Funcțiile supraîncărcate în acest fel se selectează nu numai după numărul și tipul parametrilor ci și după obiectul pentru care sunt apelate. Cu alte cuvinte, reluând exemplul de mai sus, la apelul:

*b.f(...)*

se apelează funcția membru *f* a clasei de bază B, iar la apelul:

*a.f(...)*

se apelează funcția membru *f* a clasei A dacă *f* este supraîncărcată pentru clasa A

și funcția membru *f* a clasei de bază B dacă *f* nu este supraîncărcată pentru clasa A.

O problemă care apare în acest caz este aceea cind în corpul funcției membru *f*, supraîncărcată pentru clasa A, se dorește apelul funcției membru *f* a clasei B:

```
A::f(...)
{
    ...
// în acest punct este nevoie de apelul funcției membru f a clasei B
    ...
}
```

Dacă în corpul funcției *f* utilizăm un apel obișnuit:

*f(...)*

atunci acesta se realizează pentru obiectul curent și conduce la un apel recursiv al funcției *f*. Pentru a apela funcția membru *f* a clasei de bază B pentru obiectul curent care este o instanțiere a clasei derivate A, folosim operatorul de rezoluție:

B::f(...)

Deci, corpul funcției *f* se va defini astfel:

```
A::f(...)
{
    ...
B::f(...) // se apelează funcția membru f a clasei de bază B pentru obiectul curent
           // care este instanțiere a clasei A derivată din B
}
```

### Exemplu:

```
class B {
protected:
    double x;
    double y;
public:
    B(double bx=0,double by=0)
    {
        x=bx; y=by;
    }

    void afis() const
    {
        printf("bx=%g\ty=%g\n",x,y);
    }
};

class A:public B {
protected:
    double xx;
    double yy;
public:
```

```

A(double dx=0,double dy=0,double bx=0,
   double by=0):B(bx,by)
{
    xx=dx;
    yy=dy;
}

void afis() const;
...

```

Funcția membru *afis*, a clasei de bază B, este suprainsarcată pentru clasa derivată A. Funcția *afis* aplicată la o instanțiere a clasei B va afișa valorile lui *x* și *y*.

Funcția *afis* aplicată la o instanțiere a clasei A va afișa atât valorile lui *x* și *y* ale obiectului curent cît și valorile lui *xx* și *yy* ale același obiect. De aceea, la definirea funcției *afis* pentru clasa A vom apela funcția *afis* membru a clasei B.

```

void A::afis() const
{
    B::afis(); // afiseaza pe x si y
    printf("dx=%g\tdy=%g\n",xx,yy);
}

```

Exemple de apel a funcției *afis*:

```

B b(1,2); // b.x=1, b.y=2
A a(1,2,3,4); // a.x=3, a.y = 4, a.xx=1, a.yy=2
b.afis(); // se apeleaza functia membru afis a clasei B

```

Se afișează:

```

bx=1 by=2
a.afis(); // se apeleaza functia membru afis a clasii A

```

Se afișează:

```

bx=3 by=4
dx=1 dy=2

```

În legătură cu suprainsarcarea operatorului de atribuire pentru clasa derivată amintim următoarele:

1. Dacă operatorul de atribuire nu este suprainsarcăt nici pentru clasa derivată și nici pentru clasele de bază, atunci la atribuirea a două obiecte ale clasei derivate se realizează o copiere bit cu bit a obiectului din dreapta semnului de atribuire (obiectul sursă) în obiectul din stînga semnului de atribuire (obiectul destinație).
2. Dacă operatorul de atribuire nu este suprainsarcăt pentru clasa derivată, dar este suprainsarcăt pentru una sau mai multe clase de bază, atunci pentru datele membru moștenite de la clasele de bază care au operatorul de atribuire suprainsarcăt se apelează în mod automat funcțiile de suprainsarcare a operatorului de atribuire corespunzătoare, iar pentru restul datelor membru

se face o copiere bit cu bit.

3. Dacă operatorul de atribuire este suprainsarcăt pentru clasa derivată, atunci funcția care suprainsarcă operatorul respectiv realizează atribuirea obiectului sursă obiectului destinație pentru toate datele membru. În acest caz, nu se mai apelează automat eventualele funcții care suprainsarcă operatorul de atribuire pentru clasele de bază.

### Exerciții:

- 29.7 Să se modifice clasa *punct*, definită în exercițiul 29.1, adăugind construcatorul de copiere și suprainsarcarea operatorului de atribuire.

### FIȘIERUL BXXIX7.H

```

class punct {
protected:
    double x0; double y0;
    static int rcit_double(const char *text, double& d);
public:
    punct() // constructor implicit
    {
        x0 = 0; y0 = 0;
    }
    punct(double abs,double ord=0);
    punct(const punct&); // constructor de copiere
    int citpct(const char *text);
    void afispct(const char *f) const;
    friend punct operator +(punct p1,punct p2);
    friend punct operator -(punct p1,punct p2);
    punct operator *(double d);
    friend punct operator *(double d,punct p2);
    punct operator /(double d);
    double dist(punct p2) const;
    double retabs() const;
    double retord() const;
    friend int operator == (punct p1,punct p2);
    punct operator -();
};

// suprainsarcarea operatorului de atribuire
punct& operator = (const punct&);
}

```

Funcțiile membru noi (constructorul de copiere și suprainsarcarea operatorului de atribuire) se definesc în fișierul de extensie *CPP*, de mai jos. Funcțiile membru vechi, se includ din fișierul *BXXIX2.CPP*.

### FIȘIERUL BXXIX7

```

#ifndef __PUNCT_H
#include "BXXIX7.H"
#define __PUNCT_H
#endif
#include "BXXIX1.CPP" // contine definițiile functiilor membru și prieten vechi

```

```

// se definesc funcțiile membru noi

// constructorul de copiere
punct::punct(const punct& p)
{
    x0=p.x0; y0=p.y0;
}

// supraincarcarea operatorului de atribuire
punct& punct::operator = (const punct& p)
{
    if(this != &p) {
        x0=p.x0; y0=p.y0;
    }
    return *this;
}

```

- 29.8 Să se modifice clasa *gpunct*, definită în exercițiul 29.2, adăugind constructorul de copiere și supraincărcind operatorul de atribuire.

#### FIȘIERUL BXXIX8.H

```

#ifndef __PUNCT_H
#include "BXXIX7.CPP"
#define __PUNCT_H
#endif

#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif

class gpunct:public punct {
protected:
    int culoare;
    int vizibil;
public:
    gpunct(double abs=0,double ord=0,int c=WHITE);
    gpunct(const punct& p,int c=WHITE);

    // constructor de copiere
    gpunct(const gpunct&);

    void afiseaza();
    void sterge();

    // supraincarcarea operatorului de atribuire
    gpunct& operator = (const gpunct&);
};

```

Funcțiile membru noi se definesc în fișierul de extensie *CPP*, de mai jos. Celelalte funcții se includ din fișierul *BXXIX2.CPP*.

#### FIȘIERUL BXXIX8

```

#ifndef __GPUNCT_H
#include "BXXIX8.H"
#define __GPUNCT_H
#endif
#include "BXXIX2.CPP"

inline gpunct::gpunct(const gpunct& gp):punct(gp)
// constructor de copiere
{
    culoare = gp.culoare;
    vizibil = gp.vizibil;
}

gpunct& gpunct::operator = (const gpunct& gp)
// supraincarcarea operatorului de atribuire
{
    if(this != &gp){
        x0=gp.x0; y0=gp.y0;
        culoare=gp.culoare;
        vizibil=gp.vizibil;
    }
    return *this;
}

```

#### Observație:

Constructorul de copiere al clasei *gpunct* apelează în mod explicit constructorul de copiere al clasei *punct*. În acest scop se realizează în mod implicit o conversie a referinței la un obiect de tip *gpunct* într-o referință la un obiect de tip *punct*. Într-adevăr, parametrul *gp* este o referință la un obiect de tip *gpunct*. În antetul constructorului de copiere se face un apel al constructorului de copiere *punct* cu parametrul *gp*. Constructorul de copiere *punct* are ca parametru o referință la un obiect de tip *punct*. De aceea, la apelul constructorului de copiere *punct* se realizează o conversie automată a lui *gp* din referință la obiecte de tip *gpunct* în referință la obiecte de tip *punct*.

- 29.9 Să se modifice clasa *gtextpunct*, definită în exercițiul 29.5, adăugind un constructor care are doi parametri neimpliciti, unul pentru inițializarea datei membru *text*, iar celălalt pentru inițializarea datelor membru ale clasei *gpunct*, un constructor de copiere și supraincărcind operatorul de atribuire.

#### FIȘIERUL BXXIX9.H

```

#ifndef __GTEXT_H
#include "BXXIX4.CPP"
#define __GTEXT_H
#endif

#ifndef __GPUNCT_H
#include "BXXIX8.CPP"
#define __GPUNCT_H

```

```

#endif

class gtextpunct:public gpunct,public gtext {
protected:
    int vizibil_punct_text;
public:
    gtextpunct();
    gtextpunct(char *sir,double x=0,
               double y=0,int c=WHITE,
               int culoare=WHITE,int f=DEFAULT_FONT,
               int dir=HORIZ_DIR,int dim_car=1,
               int cadrej_o=CENTER_TEXT,
               int cadrej_v=CENTER_TEXT);
    gtextpunct(char *sir,const gpunct& gp,
               int culoare=WHITE,int f=DEFAULT_FONT,
               int dir=HORIZ_DIR,int dim_car=1,
               int cadrej_o=CENTER_TEXT,
               int cadrej_v=CENTER_TEXT);

    gtextpunct(const gtextpunct&); // constructor de copiere
    void afisgtextpunct(int xt,int yt);
    void stergegtextpunct(int xt,int yt);
    gtextpunct& operator = (const gtextpunct&);
    // supraincarca operatorul de atribuire
};

Functiile membru noi se definesc in fisierul de extensie CPP, de mai jos. Cele vechi, se includ din fisierul BXXIX5.CPP.

```

## FIŞIERUL BXXIX9

```

#ifndef __GTEXTPUNCT_H
#include "BXXIX9.H"
#define __GTEXTPUNCT_H
#endif
#include "BXXIX5.CPP"

gtextpunct::gtextpunct(char *sir, const gpunct& gp,int culoare,
                      int f,int dir,int dim_car,int cadrej_o,int cadrej_v):
    gpunct(gp),gtext(sir,culoare,f,dir,dim_car,cadrej_o,cadrej_v)
// constructor cu doi parametri neimpliciti
{
    vizibil_punct_text = 0;
}

gtextpunct::gtextpunct(const gtextpunct& gtp): gpunct(gtp)
// constructor de copiere
{
    strncpy(text,gtp.text,11);
    culoare_text = gtp.culoare_text;
    font = gtp.font;
    directie = gtp.directie;
    dimensiune_caracter = gtp.dimensiune_caracter;
}

```

```

cadraj_oriz = gtp.cadraj_oriz;
cadraj_vert = gtp.cadraj_vert;
vizibil_text = gtp.vizibil_text;
vizibil_punct_text = gtp.vizibil_punct_text;
}

gtextpunct& gtextpunct::operator = (const gtextpunct& gtp)
// supraincarca operatorul de atribuire
{
    if(this != &gtp){
        x0 = gtp.x0; y0 = gtp.y0;
        culoare = gtp.culoare;
        vizibil = gtp.vizibil;
        strncpy(text,gtp.text,11);
        culoare_text = gtp.culoare_text;
        font = gtp.font;
        directie = gtp.directie;
        dimensiune_caracter = gtp.dimensiune_caracter;
        cadrej_oriz = gtp.cadraj_oriz;
        cadrej_vert = gtp.cadraj_vert;
        vizibil_text = gtp.vizibil_text;
        vizibil_punct_text = gtp.vizibil_punct_text;
    }
    return *this;
}

```

## 29.10 Să se scrie un program care realizează următoarele:

- citește coordonatele a două puncte A și B;
- citește un număr diferit de -1 pe care îl atribuie variabilei  $k$ ;
- determină punctul C care împarte segmentul AB în raportul  $k$ ;
- afișează punctele A, B, C pe ecran împreună cu notațiile lor.

Punctele A și B se afișează cu culoarea definită de constanta simbolică WHITE, iar punctul C folosind constanta simbolică LIGHTGREEN.

Fie  $ax$  și  $ay$  coordonatele punctului A și  $bx$  și  $by$  ale lui B. Coordonatele  $cx$  și  $cy$  ale punctului C, care împarte segmentul AB în raportul  $k$ , se determină cu ajutorul formulelor:

$$cx = (ax + k * bx) / (k + 1)$$

$$cy = (ay + k * by) / (k + 1)$$

## PROGRAMUL BXXIX10

```

#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIX9.CPP"

```

```

main()
/* - citesc coordonatele punctelor A si B;
 - citesc pe k diferit de -1;
 - determina punctul C care imparte segmentul AB in raportul k;
 - afisaza punctele A, B si C. */
{
    char t[255];

    gtextpunct A("A"), B("B");
    // citeste coordonatele lui A si B

    if(A.citpct("A") == 0) exit(1);
    if(B.citpct("B") == 0) exit(1);
    double k;

    // citeste numarul k
    for(;;){
        printf("raportul k=");
        if(gets(t) == 0) exit(1);
        if(sscanf(t,"%lf",&k) == 1 && k != -1) break;
        printf("nu s-a tastat un numar diferit de -1\n");
    }

    punct p((A+k*B)/(1+k));
    gpunct gp(p,LIGHTGREEN);
    gtextpunct C("C",gp,LIGHTGREEN);
    int gd = DETECT,gm;
    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    A.afisgtextpunct(A.retabs()-10,A.retord());
    B.afisgtextpunct(B.retabs(),B.retord()+10);
    C.afisgtextpunct(C.retabs()+10,C.retord()-10);
    getch();
    closegraph();
}

```

29.11 Să se implementeze tipul abstract *gsegment* pentru lucrul cu segmente de dreaptă.

Un segment de dreaptă este definit de două puncte. Clasa *gsegment* este o clasă derivată din clasa *gpunct*.

Un capăt al segmentului are coordonatele  $x_0, y_0$  măștenite de la clasa *gpunct*. Celălalt capăt al segmentului este definit de data membru *pct* a clasei *gsegment* și care este o instanțiere a clasei *punct*.

Culoarea de trasare pe ecranul grafic a segmentului este definită de data membru *culoare* măștenită de la clasa *gpunct*.

O altă dată membru este variabila *vizibil\_seg*. Dacă segmentul este afișat pe ecran, *vizibil\_seg* are valoarea 1, altfel are valoarea zero.

Pe ecranul grafic se pot trasa linii de grosimi definite cu ajutorul constantelor simbolice *NORM\_WIDTH* și *THICK\_WIDTH*.

În mod implicit, grosimea liniei se definește cu ajutorul constantei simbolice *NORM\_WIDTH*. Stilul trasării este totdeauna cel definit de constanta simbolică

#### SOLID\_LINE.

Caracteristicile curente pentru trasarea liniilor se pot determina cu ajutorul funcției *getlinesettings* de prototip (vezi paragraful 19.7.):

```
void far getlinesettings(struct linesettingstype far *lineinfo);
```

unde structura *linesettingstype* se definește în fișierul *graphics.h* astfel:

```
struct linesettingstype {
    int linestyle;
    unsigned upattern;
    int thickness;
};
```

Caracteristicile respective se pot seta folosind funcția *setlinestyle* de prototip:

```
void far setlinestyle(int stil,unsigned sablon,int grosime);
```

Parametrii *stil* și *sablon* au valorile implicate *SOLID\_LINE* și respectiv zero pentru obiectele de tip *gsegment* și ele nu pot fi modificate de utilizator. În schimb, parametrul *grosime* pentru obiectele de tip *gsegment*, are valoarea implicită *NORM\_WIDTH*, dar poate fi modificat de utilizator la instantierea obiectelor folosind constanta simbolică *THICK\_WIDTH*. În acest scop, clasa *gsegment* are o dată membru corespunzătoare numită *grosime*.

La instantierea obiectelor de tip *gsegment*, constructorul definește coeficienții *m*, *n* ai ecuației dreptei suport a segmentului care se instantiază.

În general, o dreaptă care nu este paralelă cu axa *Oy*, are ecuația:

$$y = mx + n$$

Un segment este paralel cu axa *Oy* dacă abscisele capetelor segmentului sunt egale. În notațiile de mai sus, segmentul este paralel cu axa *Oy* dacă  $x_0$  este egal cu *pct.x0*.

Coeficienții *m* și *n* se determină astfel:

$$\begin{aligned} m &= (pct.y_0 - y_0) / (pct.x_0 - x_0); \\ n &= y_0 - m \cdot x_0 \end{aligned}$$

Dacă segmentul este paralel cu axa *Oy*, atunci dreapta are ecuația:

$$x = x_0$$

și deci coeficienții *m* și *n* nu se mai definesc.

Pentru a putea distinge cazul segmentelor paralele cu axa *Oy* de cele care nu sunt paralele cu axa respectivă, clasa *gsegment* are o dată membru de nume *ypar*.

Variabila *ypar* are valoarea 1 dacă segmentul este paralel cu axa *Oy* și zero în caz contrar. Rezultă că, dreapta suport a segmentului care se instantiază, are ecuația:

$$y = mx + n$$

dacă

$$ypar = 0$$

și

x=x0

dacă

ypar = 1.

## FIŞIERUL BXXIX11.H

```
#ifndef __GPUNCT_H
#include "BXXIX8.CPP"
#define __GPUNCT_H
#endif

class gsegment:public gpunct {
protected:
    punct pct;
    int vizibil_seg;
    int grosime;
    int ypar;
    double m,n;

    void mn(); // calculeaza coeficientii m si n

public:
    gsegment(); // constructor implicit

    gsegment(double xpct,double ypct=0,double x=0,
             double y=0,int c=WHITE,int lat=NORM_WIDTH);
    // constructor utilizat la initializari si conversii

    gsegment(const punct& p0,const punct& p1,
             int c=WHITE,int lat=NORM_WIDTH);
    /* constructor utilizat la initializari cind se dau punctele limite ale segmentului */

    gsegment(const gsegment&);
    // constructor de copiere

    gsegment& operator=(const gsegment&);
    // supraincarca operatorul de atribuire

    void afiseaza();
    // afiseaza segmentul in mod grafic

    void sterge();
    // face invizibil segmentul curent

    double lungime() const;
    // returneaza lungimea segmentului curent

    int operator == (gsegment gs) const;
    /* returneaza 1 daca segmentul curent coincide cu obiectul gs si zero altfel */

    int operator || (gsegment gs) const;
    /* returneaza 1 daca segmentul curent este paralel cu gs si zero altfel */
```

```
int operator && (gsegment gs) const;
/* returneaza 1 daca segmentul curent este perpendicular pe gs si zero altfel */

int operator & (punct p) const;
/* returneaza 1 daca punctul p apartine dreptei suport a segmentului curent si zero altfel */

gsegment operator ^ (gpunct p) const;
/* returneaza un segment care trece prin punctul p si este paralel cu segmentul curent */

gsegment operator | (gpunct p) const;
/* returneaza un segment care trece prin punctul p si este perpendicular pe segmentul curent */

punct operator * (gsegment gs) const;
/* - returneaza punctul de intersecție dintre dreptele suport ale segmentului curent si gs;
   - daca segmentele nu se intersecteaza, obiectul returnat nu este definit */

double retm() const;
/* - returneaza m;
   - rezultatul este nedeterminat daca segmentul este paralel cu axa Oy */

double retn() const;
/* - returneaza n;
   - rezultatul este nedeterminat daca segmentul este paralel cu axa Oy */

int operator != () const;
/* returneaza 1 daca segmentul curent este paralel cu axa Oy si zero altfel */

gpunct retx0y0() const;
// returneaza punctul de coordonate (x,y)

punct retpct() const;
// returneaza punctul pct al obiectului curent

int retculoare() const;
// returneaza culoarea obiectului curent

int retgrosime() const;
// returneaza grosimea obiectului curent

void setculoare (int c);
/* seteaza culoarea obiectului curent la valoarea lui c */

void setgrosime (int g);
/* seteaza grosimea obiectului curent la valoarea lui g */

};

Functiile membru se definesc in fisierul de mai jos, de extensie CPP.
```

Anumite functii efectueaza teste de egalitate in care intervine coeficientul *m*. Avind in vedere faptul ca *m* este de tip *double*, aceste teste sunt afectate de erorile care intervin in mod inevitabil la calculul coeficientului *m*.

De exemplu, două drepte de ecuații:

$$y = m_1x + n_1$$

și

$$y = m_2x + n_2$$

sint perpendiculare, dacă:

$$m_1 \cdot m_2 = -1$$

Un astfel de test este afectat, de obicei, de următoarele erori:

- eroarea de calcul a coeficienților  $m_1$  și  $m_2$ ;
- eroarea de calcul a produsului  $m_1 \cdot m_2$ .

De aceea, testul de mai sus este normal să fie înlocuit cu următorul:

$$\text{abs}(m_1 \cdot m_2 + 1) < \text{EPS}$$

unde:

$\text{abs}(x)$  - Însemnă valoarea absolută a lui  $x$ .

$\text{EPS}$  - Are o valoare dependentă de precizia de calcul.

În cazul de față,  $\text{EPS} = 1.0E-13$ .

Un test asemănător se face în cazul în care se stabilește paralelismul a două drepte. Astfel, dreptele amintite mai sus sunt paralele, dacă  $m_1 = m_2$ . Acest test este afectat de erorile de calcul ale coeficienților  $m_1$  și  $m_2$ . De aceea, testul respectiv se înlocuiește cu următorul:

$$\text{abs}(m_1 - m_2) < \text{EPS}$$

Alte funcții membru care utilizează teste de acest fel sunt:

- funcția care testează apartenența unui punct la o dreaptă (operatorul & supraincărcat);
- funcția care determină un segment perpendicular pe unul dat.

În esență, un test de egalitate de forma:

$$(1) \text{exp1} = \text{exp2}$$

unde:

$\text{exp1}$  și  $\text{exp2}$  - Sunt expresii de tip *double* se înlocuiește cu testul:

$$(2) \text{abs}(\text{exp1} - \text{exp2}) < \text{EPS}$$

Testul (1) se consideră că are valoarea *adevărat* atunci și numai atunci cind testul (2) are valoarea *adevărat*.

Pentru a realiza un test de forma (1) cu ajutorul relației (2), se utilizează o funcție obișnuită numită *testegal*. Ea are prototipul:

$$\text{int testegal(double exp1, double exp2);}$$

Funcția returnează valoarea 1 dacă relația (2) este adevărată și zero altfel.

Relația (2) nu este concluzată dacă  $\text{exp1}$  și  $\text{exp2}$  sunt ele însăși mai mici, în valoare absolută, decit  $\text{EPS}$ . De aceea, înainte de a testa relația (2), funcția testează dacă aceste expresii au valoarea absolută mai mică decit  $\text{EPS}$ . În caz afirmativ, în relația (2), expresia respectivă se înlocuiește cu zero.

## FISIERUL BXXIX11

```
#ifndef __GSEGMENT_H
#include "BXXIX11.H"
#define __GSEGMENT_H
#endif
#define EPS 1.0E-13

int testegal(double exp1, double exp2)
/* returneaza 1 daca relatia:
   abs(exp1-exp2) < EPS
este adevarata;
zero altfel
*/
{
    double d1 = exp1 < 0 ? -exp1:exp1; // abs(exp1)
    double d2 = exp2 < 0 ? -exp2:exp2; // abs(exp2)

    if(d1 < EPS) d1 = 0; // abs(exp1) < EPS
    else d1 = exp1; // abs(exp1) >= EPS
    if(d2 < EPS) d2 = 0; // abs(exp2) < EPS
    else d2 = exp2; // abs(exp2) >= EPS
    if((d1 - d2) < 0) d1 = -d1; // d1 = abs(exp1-exp2)
    return d1 < EPS;
} // sfârșit testegal

void gsegment::mn()
/* - determina coeficienții m și n ai ecuației dreptei care trece prin punctele:
   (x0,y0) - punct mostenit;
   (pct.x0,pct.y0) punct membru;
   - daca segmentul curent este paralel cu axa Oy, atunci m și n
     sunt nedefiniți, iar data membru ypar are valoarea 1;
   - ypar are valoarea -1 daca punctele coincid (x0=pct.x0 si y0=pct.y0);
   - ypar are valoarea zero in restul cazurilor */
{
    if(x0 != pct.retabs())
        // se determina coeficienții m,n și ypar=0
        m=(pct.retord() - y0) / (pct.retabs() - x0);
        n=y0 - m*x0;
        ypar=0;
    }
    else
        if(y0 != pct.retord())
            // segment paralel cu Oy
            ypar = 1;
        else // punctele coincid
            ypar = -1;
} // sfârșit mn

gsegment::gsegment () :pct(639, 0) // constructor implicit
{
    // se instantiază un obiect implicit și anume axa Ox
    x0 = y0 = 0;
```

```

grosime = NORM_WIDTH;
m=n=0;
vizibil = 0; // data mostenita
vizibil_seg=0; // segmentul nu este vizibil la instantiere
ypar=0; // segmentul are capetele distincte si nu este paralel cu Oy
} // sfirsit constructor implicit

inline gsegment::gsegment(double xpct,double ypct,
    double x,double y,int c,int lat):
    gpunct(x,y,c),pct(xpct,ypct),grosime(lat)
// constructor utilizat pentru initializari si conversii
{
    mn(); // determina m, n si ypar
    vizibil_seg=0; // segmentul nu este vizibil la instantiere
} // sfirsit constructor

inline gsegment::gsegment(const punct& p0,
    const punct& p1,int c,int lat):
    gpunct(p0,c),pct(p1),grosime(lat)
/* constructor utilizat la instantieri cind se dau punctele limita ale segmentului */
{
    mn(); // determina m, n si ypar
    vizibil_seg=0; // segmentul nu este vizibil la instantiere
} // sfirsit constructor

gsegment::gsegment(const gsegment& gs):gpunct(gs)
// constructor de copiere
{
    pct = gs.pct;
    grosime = gs.grosime;
    ypar = gs.ypar;
    vizibil_seg=gs.vizibil_seg;
    m=gs.m;
    n=gs.n;
} // sfirsit constructor de copiere

gsegment& gsegment::operator=(const gsegment& gs)
// supraineasca operatorul de atribuire
{
    if(this != &gs) /* obiectul din dreapta semnului = difera de cel din stanga */
        x0=gs.x0; y0=gs.y0;
        culoare=gs.culoare;
        vizibil=gs.vizibil;
        double punct::*p;
        p = &punct::x0;
        pct.*p=gs.pct.*p;
        p = &punct::y0;
        pct.*p=gs.pct.*p;
        grosime=gs.grosime;
        vizibil_seg=gs.vizibil_seg;
        ypar=gs.ypar;
        m=gs.m;
        n=gs.n;
}

```

```

    return *this;
} // sfirsit operator=

void gsegment::afiseaza() // afiseaza segmentul curent
{
    struct linesettingtype temp;

    // sc salveaza caracteristicile curente
    getlinesettings(&temp);
    int c=getcolor();

    // sc seteaza caracteristicile obiectului curent
    setlinestyle(SOLID_LINE,0,grosime);
    setcolor(culoare);

    // sc afiseaza segmentul curent
    double punct::*p;
    double punct::*q;
    p = &punct::x0;
    q = &punct::y0;
    int x1,y1;
    getaspectratio(&x1,&y1);
    double f = (double)x1/y1;

    line(x0,y0*f,pct.*p,pct.*q*f);
    vizibil_seg=1; // segmentul este vizibil
    vizibil=1;

    // sc refac caracteristicile salvate
    setlinestyle(temp.linestyle,temp.upattern,temp.thickness);
    setcolor(c);
} // sfirsit afiseaza

void gsegment::sterge() // face invizibil segmentul curent
{
    int temp=culoare; // salveaza culoarea segmentului curent

    // culoarea obiectului curent devine egala cu cea de fond
    culoare=getbkcolor();
    afiseaza(); /* segmentul curent se afiseaza cu culoarea de fond, devenind invizibil */
    culoare =temp; /* se refac culoarea obiectului curent */
    vizibil_seg=0; vizibil=0;
} // sfirsit sterge

inline double gsegment::lungime() const
// returneaza lungimea segmentului curent
{
    return pct.dist(punct(x0,y0));
} // sfirsit lungime

inline int gsegment::operator == (gsegment gs) const
/* returneaza 1 daca segmentul curent coincide cu obiectul gs si zero altfel */
{
    return pct == gs.pct && x0 == gs.x0 && y0 == gs.y0;
}

```

```

    } // sfirsit operator ==

    int gsegment::operator || (gsegment gs) const
    /* returneaza 1 daca segmentul curent este paralel cu gs si zero altfel */
    {
        if(ypar == 0 && gs.ypar == 0) return testegal(m,gs.m);
        if(ypar == 1 && gs.ypar == 1)
            // ambele segmente sunt paralele cu Oy
            return 1;
        return 0; // segmentele nu sunt paralele
    } // sfirsit operator ||

    int gsegment::operator && (gsegment gs) const
    /* returneaza 1 daca segmentul curent este perpendicular pe gs si zero altfel */
    {
        if(ypar == 0 && gs.ypar == 0) return testegal(m*gs.m,-1.0);
        if(ypar && gs.ypar)
        /* - segmentele sunt paralele cu Oy sau cel putin unul nu este definit;
           - in ambele situatii nu sunt perpendicularare. */
            return 0;

        if(ypar == 1 && gs.ypar == 0) /* segmentul curent este paralel cu Oy */
            return testegal(gs.m,0);

        /* daca testegal returneaza valoarea 1 inseamna ca gs.m=0,
           deci segmentul curent este perpendicular pe gs */

        if(ypar == 0 && gs.ypar == 1) /* segmentul gs este paralel cu Oy */
            return testegal(m,0);

        // in celelalte cazuri segmentele nu sunt perpendicularare
        return 0;
    } // sfirsit operator &&

    int gsegment::operator & (punct p) const
    /* returneaza 1 daca punctul p apartine dreptei suport a segmentului curent si zero altfel */
    {
        double punct::*u;
        double punct::*v;
        u = &punct::x0;
        v = &punct::y0;

        if(ypar == 0) { // segmentul curent nu este paralel cu oy
            double exp1=p.*u;
            exp1=m*exp1+n;
            double exp2=p.*v;
            return testegal(exp1,exp2);
        }

        if(ypar == 1) // segmentul curent este paralel cu Oy
        /* punctul p apartine dreptei suport daca p.x0=x0 deoarece dreapta suport are ecuatie x=x0 */
            return testegal(x0,p.*u);
        // in celelalte cazuri segmentul curent nu este definit
    }

```

```

        return 0;
    } // sfirsit operator &

    gsegment gsegment::operator ^ (gpunct p) const
    /* - returneaza un segment a carui dreapta suport trece prin punctul p si
       este paralela cu segmentul curent;
       - p este un capat al segmentului returnat;
       - celalalt capat al segmentului se determina astfel:
           - daca Oy este paralela cu dreapta suport a segmentului determinat, se considera
             punctul de coordonate (p.x0, p.y0+10);
           - altfel se considera punctul de pe dreapta determinata de abscisa p.x0+10;
           - culoarea segmentului este aceeasi cu a punctului p, iar
             grosimea este cea implicita: NORM_WIDTH.
    */
    {
        double gpunct::*u;
        double gpunct::*v;
        u = &gpunct::x0;
        v = &gpunct::y0;
        int gpunct::*w;
        w = &gpunct::culoare;

        if(ypar == 0) {
        /* - segmentul curent nu este paralel cu Oy;
           - coeficientii m,n sunt determinati pentru segmentul curent;
           - segmentul paralel cu el are dreapta suport de ecuatie:
               y = mx + nn;
               coordonatele punctului p verifică această ecuație, deci:
               nn = p.y0 - m*p.x0
        */
            double nn = p.*v - m*p.*u;
            double a = p.*u + 10;
            double b = m*(p.*u + 10) + nn;
            double c = p.*u;
            double d = p.*v;
            int e = p.*w;
            gsegment r(a,b,c,d,e);
            return r;
        }

        /* - segmentul curent este paralel cu Oy sau nu este definit;
           - segmentul paralel cu el si care trece prin punctul p are drept capete
             punctele de coordonate (p.x0,p.y0) si (p.x0,p.y0+10). */
        double a = p.*u;
        double b = p.*v + 10;
        double c = p.*v;
        int d = p.*w;

        gsegment r(a,b,a,c,d);
        return r;
    } // sfirsit operator ^

```

```

gsegment gsegment::operator | (gpunct p) const
/* - returneaza un segment a carui dreapta suport este perpendiculara pe
segmentul curent si trece prin punctul p;
- p este un capat al segmentului;
- celalalt capat al segmentului se determina astfel:
- daca Ox nu este paralela cu segmentul curent, atunci se considera punctul de
abscisa p.x0+10 de pe dreapta suport a segmentului care se determina;
- altfel se considera punctul de coordonate (p.x0,p.y0+10);
culoarea segmentului este aceeasi cu a punctului p, iar grosimea este cea
implicita: NORM_WIDTH.
*/
{
    double gpunct::*u;
    double gpunct::*v;
    int gpunct::*w;

    u = &gpunct::x0;
    v = &gpunct::y0;
    w = &gpunct::culoare;
    if(ypar == 0 && testegal(m,0)) {
        // segmentul curent este paralel cu axa Ox
        double a = p.*u;
        double b = p.*v + 10;
        double c = p.*v;
        int d = p.*w;

        gsegment r(a,b,a,d);
        return r;
    }
    // segmentul curent nu este paralel cu Ox

    if(ypar == 1 || ypar == -1) {
        /* segmentul curent este paralel cu Oy sau este nedefinit;
        dreapta perpendiculara pe el are ecuatie:
        y = constant;
        segmentul rezultat are drept capete punctele de coordonate:
        (p.x0,p.y0) si (p.x0+10,p.y0)
        */
        double a = p.*u + 10;
        double b = p.*v;
        double c = p.*u;
        int d = p.*w;

        gsegment r(a,b,c,b,d);
        return r;
    }
    /* - segmentul curent nu este paralel cu nici una dintre axe Ox sau Oy;
    - coeficientii m si n sunt determinati pentru segmentul curent si m != 0;
    - dreapta perpendiculara pe segmentul curent are coeficientul unghiular egal cu -1/m;
    ea are ecuatie:
    y = (-1/m)*x + nn

```

- coeficientul nn se determina din conditia ca dreapta sa treaca prin punctul (p.x0,p.y0)

```

double nn = p.*v + (1/m)*p.*u;
double a = p.*u + 10;
double b = (-1/m)* a + nn;
double c = p.*u;
double d = p.*v;
int e = p.*w;

gsegment r(a,b,c,d,e);
return r;
} // sfarsit operator |

punct gsegment::operator * (gsegment gs) const
/* - returneaza punctul de intersectie dintre dreptele suport ale segmentului curent si gs;
- daca segmentele nu se intersecteaza, obiectul returnat nu este definit.
*/
{
    punct r;

    if(ypar == -1 || gs.ypar == -1)
        // cel putin unul dintre segmente nu este definit
        return r;

    if(ypar && gs.ypar)
        // segmentele sunt paralele cu axa Oy
        return r;

    if(ypar == 1) {
        /* - segmentul curent este paralel cu Oy;
        - dreapta suport are ecuatie:
        x = x0;
        - punctul de intersectie are coordonatele:
        (x0,gs.m*x0+gs.n)
        */
        punct r(x0,gs.m*x0+gs.n);
        return r;
    }

    if(gs.ypar == 1) {
        /* - segmentul gs este paralel cu Oy;
        - dreapta suport are ecuatie:
        x = gs.x0;
        - punctul de intersectie are coordonatele:
        (gs.x0,m*gs.x0+n)
        */
        punct r(gs.x0,m*gs.x0+n);
        return r;
    }

    /* - nici unul din segmente nu este paralel cu axa Oy;
    - segmentele sunt paralele daca m = gs.m. */

```

```

if(testegal(m,gs.m))
/* segmentele sunt paralele avind coeficientii unghiulari egali */
    return r;

/* intersectia segmentelor se determina rezolvind sistemul de ecuatii liniare:
   y = m*x+n
   y = gs.m*x+gs.n
avem:
   m*x+n = gs.m*x+gs.n
sau
   x = (gs.n-n)/(m-gs.m)
*/
    double x = (gs.n-n)/(m-gs.m);
    punct rr(x,m*x+n);
    return rr;
} // sfarsit operator*

inline double gsegment::retm() const
/* returneaza coeficientul m; daca segmentul este paralel cu Oy, rezultatul este nedefinit */
{
    return m;
} // sfarsit retm

inline double gsegment::retn() const
/* returneaza coeficientul n; daca segmentul curent este paralel
   cu Oy, atunci rezultatul este nedefinit */
{
    return n;
} // sfarsit retn

inline int gsegment::operator !() const
/* returneaza 1 daca segmentul este paralel cu axa Oy si zero altfel */
{
    return ypar;
} // sfarsit operator !

inline gpunct gsegment::retx0y0() const
/* returneaza punctul de coordonate (x0,y0) */
{
    gpunct r(x0,y0,culoare);
    return r;
} // sfarsit retx0y0

inline punct gsegment::retpt() const
/* returneaza punctul pt al obiectului curent */
{
    return pt;
} // sfarsit retpt

inline int gsegment::retculoare() const
/* returneaza culoarea obiectului curent */
{
    return culoare;
}

```

```

} // sfarsit retculoare

inline int gsegment::retgrosime() const
/* returneaza grosimea obiectului curent */
{
    return grosime;
} // sfarsit retgrosime

inline void gsegment::setculoare (int c)
/* seteaza culoarea obiectului curent la valoarea lui c */
{
    culoare = c;
} // sfarsit setculoare

inline void gsegment::setgrosime (int g)
/* seteaza grosimea obiectului curent la valoarea lui g */
{
    grosime = g;
} // sfarsit grosime

```

29.12 Să se scrie o funcție care citește coordonatele punctelor  $x$ ,  $y$  și  $z$  care reprezintă vîrfurile unui triunghi.

Funcția returnează valoarea 1 dacă punctele respective nu sunt coliniare și zero în caz contrar.

### FUNCȚIA BXXIX12

```

#ifndef __GSEGMENT_H
#include "BXXIX11.CPP"
#define __GSEGMENT_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif

int cit3pct(punct& x,punct& y,punct& z)
/* - citeste coordonatele a 3 puncte de la intrarea standard;
   - returneaza 1 daca punctele nu sunt coliniare si zero altfel. */
{
    char er[] = "s-a citit EOF\n";

    if(x.citpct("primul punct")==0){
        printf(er);
        exit(1);
    }
    if(y.citpct("al doilea punct")==0){
        printf(er);
        exit(1);
    }
    if(z.citpct("al treilea punct")==0){
        printf(er);
        exit(1);
    }
}

```

```

}

/* se face testul de coliniaritate; punctele sint coliniare
daca, de exemplu, x apartine segmentului yz */

gsegment yz(y, z); /* are culoarea si grosimea implicita (WHITE si NORM_WIDTH)*/
if(yz & x) // x apartine segmentului yz
    return 0;

// punctele x,y,z nu sunt coliniare
return 1;
}

```

- 29.13 Să se scrie un program care citește coordonatele punctelor  $a, b, c$  care sunt virfurile unui triunghi și afișează pe ecranul grafic triunghiul respectiv împreună cu medianele sale.

Laturile triunghiului se afișează folosind culoarea implicită și grosimea definită de constanta simbolică THICK\_WIDTH.

Medianele se afișează folosind culoarea definită de constanta simbolică YELLOW și grosimea implicită.

### PROGRAMUL BXXIX13

```

#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXXIX12.CPP"

main()
/* - citește coordonatele punctelor a,b,c;
   - afișaza triunghiul abc și medianele lui;
   - laturile au culoarea implicită și grosimea THICK_WIDTH;
   - medianele au culoarea YELLOW și grosimea implicită.
*/
{
    // citește coordonatele virfurilor triunghiului
    punct a, b, c;

    if(cit3pct(a, b, c)==0){
        printf("punctele a,b,c sunt coliniare\n");
        exit(1);
    }
    int gd=DETECT, gm;
    initgraph(&gd, &gm, "c:\\borlandc\\bgi");

    // se instantiaza laturile triunghiului
    gsegment ab(a, b, WHITE, THICK_WIDTH);
    gsegment ac(a, c, WHITE, THICK_WIDTH);
    gsegment bc(b, c, WHITE, THICK_WIDTH);

    // se afișaza triunghiul pe ecranul grafic
}

```

```

ab.afiseaza();
ac.afiseaza();
bc.afiseaza();
getch();

// determina medianele triunghiului
// mediana relativa la latura bc
punct mbc = (b+c)/2;
gsegment ma(a, mbc, YELLOW);

// mediana laturii ac
punct mac = (a+c)/2;
gsegment mb(b, mac, YELLOW);

// mediana laturii ab
punct mab = (a+b)/2;
gsegment mc(c, mab, YELLOW);

// afișaza medianele
ma.afiseaza();
mb.afiseaza();
mc.afiseaza();
getch();
closegraph();
}

```

- 29.14 Să se scrie o funcție care determină centrul și raza cercului circumscris unui triunghi.

Amintim că centrul cercului circumscris unui triunghi este intersecția mediatoarelor laturilor triunghiului respectiv.

Mediatoarea unui segment este perpendiculara pe mijlocul segmentului.

### FUNCȚIA BXXIX14

```

#ifndef __GSEGMENT_H
#include "BXXIX11.CPP"
#define __GSEGMENT_H
#endif

void cerccircctr(punct x, punct y, punct z, punct& centru, double& r)
/* - determină centrul și raza cercului circumscris triunghiului xyz;
   - funcția presupune ca x,y,z nu sunt coliniare. */
{
    gsegment xy(x, y); // latura xy a triunghiului
    gsegment xz(x, z); // latura xz a triunghiului
    punct mxy = (x+y)/2; // mijlocul segmentului xy
    gsegment mediatoarexy=xy|mxy; // mediatoarea segmentului xy
    punct mxz=(x+z)/2; // mijlocul segmentului xz
    gsegment mediatoarexz=xz|mxz; // mediatoarea segmentului xz

    // centrul cercului este intersecția mediatoarelor laturilor xy și xz
}

```

```

punct c1;
int x1,y1;

c1=mediatoarexy*mediatoarexz;

// raza este distanta de la centru la un virf al triunghiului
r = c1.dist(z);

getaspectratio(&x1,&y1);
double f = (double)x1/y1;
centru = punct(c1.retabs(),c1.retord()*f);
}

```

- 29.15 Să se scrie un program care citește coordonatele a trei puncte care sunt virfurile unui triunghi și afișează triunghiul respectiv împreună cu cercul circumscris acestuia.

Triunghiul se afișează folosind constantele simbolice YELLOW pentru culoare și THICK\_WIDTH pentru grosime. Cercul se trasează folosind caracteristicile implicate ale setării grafice ale ecranului.

### PROGRAMUL BXXIX15

```

#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __CIT3PCT_H
#include "BXXIX12.CPP" // cit3pct.cpp
#define __CIT3PCT_H
#endif
#ifndef __CERCCIRCTR_H
#include "BXXIX14.CPP" // cerecicrtr.cpp
#define __CERCCIRCTR_H
#endif

main()
/* - citeste coordonatele virfurilor unui triunghi;
 - afișaza triunghiul folosind constantele simbolice YELLOW si THICK_WIDTH;
 - afișaza cercul circumscris triunghiului.
*/
{
punct a,b,c;

if(cit3pct(a,b,c) == 0){
    printf("punctele a,b,c sunt coliniare\n");
    exit(1);
}
int gd=DETECT,gm;
initgraph(&gd,&gm,"c:\\borlandc\\bgi");

// se instantiază laturile triunghiului
gsegment ab(a,b,YELLOW,THICK_WIDTH);
gsegment ac(a,c,YELLOW,THICK_WIDTH);

```

```

gsegment bc(b,c,YELLOW,THICK_WIDTH);

// se afișează triunghiul abc
ab.afiseaza();
ac.afiseaza();
bc.afiseaza();
getch();
punct centru;
double raza;

// se determină centrul și raza cercului circumscris triunghiului abc
cerccircctr(a,b,c,centru,raza);

// afișază cercul circumscris triunghiului
circle(centru.retabs(),centru.retord(),raza);
getch();
closegraph();
}

```

- 29.16 O clasă importantă de probleme din geometria euclidiană este clasa referitoare la locuri geometrice. Adesea, este utilă avea o imagine asupra unui loc geometric înainte de a găsi demonstrațiile necesare pentru a determina în mod riguros locul geometric respectiv. Astfel de imagini se obțin simplu folosind relațiile existente între obiectele geometrice (puncte, segmente, distanțe etc.) care intervin în problema de loc geometric.  
Pentru a obține imaginea locului geometric nu avem decit să afișăm deplasările obiectelor geometrice formulate ca date de intrare în problema de loc geometric, precum și a obiectelor rezultat, folosind relațiile existente între obiectele respective.  
În exercițiul de față se trasează un loc geometric întlnit în geometria triunghiului.

Fie  $a$ ,  $b$ ,  $c$  virfurile unui triunghi și se consideră cercul circumscris acestuia. Se cere locul geometric al intersecției medianelor triunghiului dat, cind punctele  $b$  și  $c$  rămân fixe, iar  $a$  se mișcă pe cercul circumscris triunghiului inițial. În acest scop, se calculează centrul și raza cercului circumscris triunghiului inițial  $abc$ , ca în exercițiul 29.15.

Coordonatele unui punct  $p(x,y)$ , de pe acest cerc, satisfac relațiile:

$$(1) \begin{aligned} x &= xc + r \cos(\alpha) \\ y &= yc + r \sin(\alpha) \end{aligned}$$

unde:

$(xc,yc)$  - Sunt coordonatele centrului cercului.

$r$  - Este raza cercului.

$\alpha$  - Este măsura, în radiani, a unghiului format de raza care trece prin punctul  $p$  cu axa Ox.

Într-adevăr, punctul  $p$  este situat la distanța  $r$  de centrul de coordonate  $(xc,yc)$ ,

oricare ar fi măsura  $\alpha$ . Aceasta rezultă ușor din relația (1):

$$\begin{aligned}x - xc &= r \cdot \cos(\alpha) \\y - yc &= r \cdot \sin(\alpha)\end{aligned}$$

sau

$$\begin{aligned}(x - xc)^2 &= r^2 \cdot \cos^2(\alpha) + r^2 \cdot \sin^2(\alpha) \\(y - yc)^2 &= r^2 \cdot \sin^2(\alpha) + r^2 \cdot \cos^2(\alpha)\end{aligned}$$

Însumind, se obține:

$$(x - xc)^2 + (y - yc)^2 = r^2$$

sau extrăgind rădăcina pătrată din ambii membrii se obține:

$$\sqrt{(x - xc)^2 + (y - yc)^2} = r$$

În partea stingă a semnului egal s-a obținut distanța de la punctul  $p(x,y)$  la centrul cercului, de coordonate  $(xc,yc)$ . Această distanță este egală cu  $r$  oricare ar fi măsura unghiulară  $\alpha$ . Cind  $\alpha$  variază de la 0 la  $2\pi$  ( $\pi = 3.1415...$ ), atunci punctul  $p$  descrie cercul cu centru de coordonate  $(xc,yc)$  și de rază  $r$ .

De aici rezultă că, pentru a trasa locul geometric căutat, nu avem decit să afișăm intersecția medianelor fiecărui triunghi care se formează menținind punctele  $b$  și  $c$  fixe, iar coordonatele lui  $a$  se modifică cu ajutorul relațiilor (1), variind pe  $\alpha$  în intervalul amintit mai sus.

Alegem ca pas pentru  $\alpha$  un grad sexagesimal.

Pentru a elimina deformările introduse de dimensiunea pixelilor, vom folosi coeficienții de aspect (vezi 19.7.).

## PROGRAMUL BXXIX16

```
#include <math.h>
#include <dos.h>
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __CIT3PCT_H
#include "BXXIX12.CPP"
#define __CIT3PCT_H
#endif
#ifndef __CERCCIRCTR_H
#include "BXXIX14.CPP"
#define __CERCCIRCTR_H
#endif
#define PI 3.14159265358979

main()
/* - citeste coordonatelor virfurilor unui triunghi;
   - afiseaza triunghiul folosind caracteristicile implice;
   - afiseaza cercul circumscris triunghiului deplasind virful a al triunghiului;
   - afiseaza curba descrisa de intersecția medianelor triunghiurilor formate prin deplasarea
   virfului a pe cercul circumscris triunghiului initial; se utilizeaza culoarea definita de
```

```
constanta simbolica YELLOW. */
{
punct a,b,c;
if(cit3pct(a,b,c) == 0){
    printf("punctele a,b,c sunt coliniare");
    exit(1);
}
gsegment ab(a,b);
gsegment ac(a,c);
gsegment bc(b,c);
int gd = DETECT,gm;
initgraph(&gd,&gm,"c:\\borlandc\\bgi");
int xa,ya;
getaspectratio(&xa,&ya);
double factor = (double)xa/ya;

// sc afiseaza triunghiul initial
ab.afiseaza();
ac.afiseaza();
bc.afiseaza();
getch();

// mijlocul lui bc este fix
punct mbc = (b+c)/2;
punct centru;
double r;

// determina centrul si raza cercului circumscris triunghiului initial
cerccircctr(a,b,c,centru,r);

// sc determina coordonatele centrului
double xc = centru.retabs();
double yc = centru.retord();
double frad = PI/180.0;
double alfa;
int g;
double rm=r*factor;

for(g=0;g < 360;g++){
// afiseaza virful a al triunghiului variabil pe cerc
    alfa = g*frad; // conversie in radiani
    double x = xc+rm*cos(alfa);
    double y = yc+rm*sin(alfa);
    double y1 = yc+rm*sin(alfa);
    punct a(x,y);
    punct am(x,y1);
    gpunct ga(am);
    ga.afiseaza();

    /* determina intersecția medianelor pentru triunghiul curent */
    if((bc & a) == 0) { // punctele nu sunt coliniare
        gsegment ma(a,mbc); // mediana relativă la bc
        punct mab = (a+b)/2;
        gsegment mc(c,mab); // mediana relativă la ab
```

```

gpunct im(ma*mc); // intersectia medianelor
gpunct imm(im.retabs(),im.retord()*factor,YELLOW);
imm.afiseaza(); // se afiseaza intersectia medianelor
}
delay(100); // vizualizare ecran
} // sfirsit for
getch();
closegraph();
}

```

- 29.17 În exercițiul 29.4, s-a definit tipul abstract *gtext* pentru a facilita afișarea de texte pe ecranul grafic. Clasa *gtext* are o funcție membru care afișează textul obiectului curent. Funcția are ca parametrii coordonatele punctului care definește inceputul cimpului în care se afișează textul. Aceleași coordonate se utilizează și în cazul în care se apelează funcția membru *stergegtext* pentru a face textul obiectului curent invizibil.

Mai jos, se definește clasa *gtext* ca o clasă derivată a clasei *punct*, definită în exercițiul 29.7.

Datele membru *x0*, *y0*, moștenite de la clasa *punct*, definesc punctul de inceput al cimpului în care se afișează textul obiectului curent.

## FIŞIERUL BXXIX17.H

```

#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#ifndef __PUNCT_H
#include "BXXIX17.CPP"
#define __PUNCT_H
#endif

class gtext:public punct {
protected:
    int font;
    int directie;
    int dimensiune_caracter;
    int cadraj_oriz;
    int cadraj_vert;
    int culoare_text;
    char text[11];
    int vizibil_text;
public:
    gtext();
    gtext(char *sir,double x=0,double y=0,int c=WHITE,
          int f=DEFAULT_FONT,int dir=HORIZ_DIR,int dim=1,
          int coriz=CENTER_TEXT,int cvert=CENTER_TEXT);

    void afisgtext(double dx=0,double dy=0);
    /* afiseaza textul definit de obiectul curent in cimpul care incepe
       in punctul de coordonate: (x0+dx,y0+dy) */
}

```

```

void stergegtext(double dx=0,double dy=0);
/* face invizibil textul care este afisat in cimpul care incepe in
   punctul de coordonate: (x0+dx,y0+dy) */
gtext(char *sir,const punct& p,int c=WHITE,
      int f=DEFAULT_FONT,int dir=HORIZ_DIR,int dim=1,
      int coriz=CENTER_TEXT,int cvert=CENTER_TEXT);
gtext(const gtext&); // constructor de copiere
gtext& operator = (const gtext&);
// supraincarca operatorul de atribuire
};


```

Funcțiile membru se definesc în fișierul de mai jos, de extensie *CPP*.

## FIŞIERUL BXXIX17

```

#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __GTEXT_H
#include "BXXIX17.H"
#define __GTEXT_H
#endif

gtext::gtext() // constructor implicit
{
    text[0]='\0';
    x0=0;
    y0=0;
    culoare_text = WHITE;
    font = DEFAULT_FONT;
    directie = HORIZ_DIR;
    dimensiune_caracter = 1;
    cadraj_oriz = CENTER_TEXT;
    cadraj_vert = CENTER_TEXT;
    vizibil_text = 0;
} // sfirsit constructor implicit

gtext::gtext(char *sir,double x,double y,int c,int f,
            int dir,int dim,int coriz,int cvert): punct(x,y)
/* constructor utilizat la initializari si conversie din sir de caractere in tipul gtext */
{
    strncpy(text,sir,11);
    text[10]='\0';
    culoare_text = c;
    font = f;
    directie = dir;
    dimensiune_caracter = dim;
    cadraj_oriz = coriz;
    cadraj_vert = cvert;
    vizibil_text = 0;
} // sfirsit constructor

gtext::gtext(char *sir,const punct& p,int c,int f,int dir,
            int dim,int coriz,int cvert): punct(p)

```

```

// constructor utilizat la initializari
{
    strcpy(text,sir,11);
    text[10]='\0';
    culoare_text = c;
    font = f;
    directie = dir;
    dimensiune_caracter = dim;
    cadreaj_oriz = coriz;
    cadreaj_vert = cvert;
    vizibil_text = 0;
} // sfârșit constructor

gtext::gtext(const gtext& gt) : punct(gt) // constructor de copiere
{
    strcpy(text,gt.text,11);
    text[10]='\0';
    culoare_text = gt.culoare_text;
    font = gt.font;
    directie = gt.directie;
    dimensiune_caracter = gt.dimensiune_caracter;
    cadreaj_oriz = gt.cadraj_oriz;
    cadreaj_vert = gt.cadraj_vert;
    vizibil_text = gt.vizibil_text;
} // sfârșit constructor de copiere

void gtext::afisgtext(double dx,double dy)
/* afiseaza textul definit de obiectul curent in cimpul care
   incepe in punctul de coordonate (x0+dx,y0+dy) */
{
    struct textsettingstype car_crt;
    int culoare_crt;

    // se salveaza caracteristicile curente
    gettextsettings(&car_crt);
    culoare_crt = getcolor();

    // se seteaza caracteristicile obiectului
    settextstyle(font,directie,dimensiune_caracter);
    settextjustify(cadraj_oriz,cadraj_vert);
    setcolor(culoare_text);

    // afiseaza textul
    outtextxy(x0+dx,y0+dy,text);

    // se refac caracteristicile
    settextstyle(car_crt.font,car_crt.direction,
                car_crt.charsize );
    settextjustify(car_crt.horiz,car_crt.vert);
    setcolor(culoare_crt);
    vizibil_text = 1; // textul este vizibil
} // sfârșit afistext

```

```

void gtext::stergegtext(double dx,double dy)
/* face invizibil textul obiectului curent care este afisat in
   cimpul care incepe in punctul de coordonate (x0+dx,y0+dy) */
{
    int ctemp=culoare_text;
    // salveaza culoarea textului obiectului curent
    culoare_text = getbkcolor();

    // culoarea obiectului curent devine culoarea fondului
    afisgtext(dx,dy); // textul devine invizibil
    culoare_text=ctemp; // se reface culoarea obiectului curent
    vizibil_text=0; // textul este invizibil
}

gtext& gtext::operator = (const gtext& gt) // supraincarca operatorul =
{
    if(this != &gt){
        x0=gt.x0; y0=gt.y0;
        strcpy(text,gt.text,11);
        text[10]='\0';
        culoare_text = gt.culoare_text;
        font = gt.font;
        directie = gt.directie;
        dimensiune_caracter = gt.dimensiune_caracter;
        cadreaj_oriz = gt.cadraj_oriz;
        cadreaj_vert = gt.cadraj_vert;
        vizibil_text = gt.vizibil_text;
    }
    return *this;
} // sfârșit operator=

```

29.18 Pentru a afișa texte împreună cu punctele, în exercițiul 29.9 s-a introdus tipul abstract *gtextpunct*.

În exercițiul de față se reimplementează tipul *gtextpunct* folosind drept clase de bază clasa *gpunct* definită în exercițiul 29.8 și clasa *gtext* definită în exercițiul 29.17.

Clasele care intervin la implementarea tipului *gtextpunct* formează o ierarhie pe care o reprezentăm în figura alăturată.

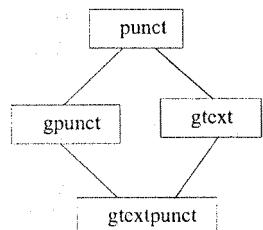
Clasa *gtextpunct* moștenește datele membru *x0* și *y0* de două ori: o dată prin intermediul clasei *gpunct* și o dată prin intermediul clasei *gtext*. Pentru a face distincție între ele se utilizează operatorul de rezoluție:

*gpunct::x0* și *gpunct::y0*

pentru cele moștenite de la clasa *gpunct* și

*gtext::x0* și *gtext::y0*

pentru cele moștenite de la clasa *gtext*.



## FIŞIERUL BXXIX18.H

```

#ifndef __GPUNCT_H
#include "BXXIX8.CPP"
#define __GPUNCT_H
#endif
#ifndef __GTEXT_H
#include "BXXIX17.CPP"
#define __GTEXT_H
#endif

class gtextpunct:public gpunct,public gtext {
protected:
    int vizibil_punct_text;
public:
    gtextpunct();
    gtextpunct(char *sir,double xp=0,double yp=0,
               int cp=WHITE,double xt=0,double yt=0,
               int ct=WHITE,int f=DEFAULT_FONT,
               int dir=HORIZ_DIR,int dim_car = 1,
               int coriz=CENTER_TEXT,int cvert=CENTER_TEXT);
/* constructor utilizat la initializari si conversie a sirurilor de caractere
   in tipul gtextpunct */
    gtextpunct(const gpunct& gp,const gtext& gt);
// constructor utilizat la initializari

    gtextpunct (const gtextpunct& );
// constructor de copiere

    void afisgtextpunct(double dx=0,double dy=0);
/* - afisaza punctul de coordonate:
       (gpunct::x0,gpunct::y0);
   - culoarea punctului este definita de data membru culoarc, mostenita de la
     clasa gpunct;
   - afisaza textul obiectului curent in cimpul care incepe in punctul de coordonate:
       -(gtext::x0+dx,gtext::y0+dy);
   - culoarea textului este definita de data membru culoare_text, mostenita de la
     clasa gtext;
*/
    void stergtextpunct(double dx=0,double dy=0);
/* face invizibil punctul de coordonate:
       (gpunct::x0,gpunct::y0)
   si textul care incepe in punctul de coordonate:
       (gtext::x0+dx,gtext::y0+dy)
*/
    gtextpunct& gtextpunct::operator=(const gtextpunct& );
// suprainscarca operatorul =
};


```

Mai jos, se definesc funcțiile membru ale clasei *gtextpunct* intr-un fișier de extensie *CPP*.

## FIŞIERUL BXXIX18

```

#ifndef __GTEXTPUNCT_H
#include "BXXIX18.H"
#define __GTEXTPUNCT_H
#endif

inline gtextpunct::gtextpunct():gpunct(),gtext() // constructor implicit
{
    vizibil_punct_text = 0;
}

inline gtextpunct::gtextpunct(char *sir,double xp,double yp,
                            int cp,double xt,double yt,int ct,int f,int dir,int dim_car,
                            int coriz,int cvert): gpunct(xp,yp,cp),
                            gtext(sir,xt,yt,ct,f,dir,dim_car,coriz,cvert)
/* constructor utilizat la initializari si conversiile sirurilor de caractere
   in obiecte de tip gtextpunct */
{
    vizibil_punct_text = 0;
}

inline gtextpunct::gtextpunct(const gpunct& gp,const gtext& gt):
gpunct(gp),gtext(gt)

// constructor utilizat la initializari
{
    vizibil_punct_text = 0;
}

gtextpunct::gtextpunct(const gtextpunct& gtp) // constructor de copiere
{
    strncpy(text,gtp.text,11);
    text[10] = '\0';
    double gpunct::*p;
    double gpunct:: *q;
    p = &gpunct::x0;
    q = &gpunct::y0;

    gpunct gp = gtp;
    this ->*p = gp.*p;
    this ->*q = gp.*q;
    double gtext:: *u;
    double gtext:: *v;
    u = &gtext::x0;
    v = &gtext::y0;
    gtext gt = gtp;
    this ->*u = gt.*u;
    this ->*v = gt.*v;

    culoare = gtp.culoare;
    vizibil = gtp.vizibil;
    culoare_text = gtp.culoare_text;
}

```

```

dimensiune_caracter = gtp.dimensiune_caracter;
cadraj_oriz = gtp.cadraj_oriz;
cadraj_vert = gtp.cadraj_vert;
vizibil_text = gtp.vizibil_text;
vizibil_punct_text = gtp.vizibil_punct_text;
} // sfirsit constructor de copiere

void gtextpunct::afisgtextpunct(double dx,double dy)
/* - afisaza punctul de coordonate:
   - (gpunct::x0,gpunct::y0);
   - culoarea punctului este mostenita de la clasa gpunct;
   - afisaza textul obiectului curent in cimpul care incepe in punctul de
     coordonate:
   - (gtext::x0+dx,gtext::y0+dy);
   - celelalte caracteristici ale textului sunt mostenite de la clasa gtext.
*/
{
    afiseaza();           // afiseaza punctul
    afisgtext(dx,dy);    // afiseaza textul
    vizibil_punct_text = 1;
} // sfirsit afisgtextpunct

void gtextpunct::stergtextpunct(double dx,double dy)
/* face invizibil punctul de coordonate:
   - (gpunct::x0,gpunct::y0)
   si textul care incepe in punctul de coordonate:
   - (gtext::x0+dx,gtext::y0+dy)
*/
{
    sterge();           // face punctul invizibil
    stergetext(dx,dy); // sterge textul
    vizibil_punct_text = 0;
} // sfirsit stergtextpunct

gtextpunct& gtextpunct::operator = (const gtextpunct& gtp)
// supraincarca operatorul =
{
    if(this != &gtp) {
        strncpy(text,gtp.text,11);
        text[10] = '\0';
        gpunct gp = gtp;
        double gpunct::*p ;
        double gpunct::*q;
        double gtext::*u;
        double gtext::*v;

        p = &gpunct::x0;
        q = &gpunct::y0;
        u = &gtext::x0;
        v = &gtext::y0;

        this ->*p = gp.*p;
        this ->*q = gp.*q;
    }
}

```

```

culoare = gtp.culoare;
gtext gt = gtp;
this ->*u = gt.*u;
this ->*v = gt.*v;
vizibil = gtp.vizibil;
culoare_text = gtp.culoare_text;
dimensiune_caracter = gtp.dimensiune_caracter;
cadraj_oriz = gtp.cadraj_oriz;
cadraj_vert = gtp.cadraj_vert;
vizibil = gtp.vizibil;
vizibil_text = gtp.vizibil_text;
vizibil_punct_text = gtp.vizibil_punct_text;
}
return *this;
} // sfirsit operator=

```

- 29.19 În exercițiul 29.10 se citesc coordonatele a două puncte A și B și un număr diferit de -1 care se atribuie lui k. Se determină punctul C care împarte segmentul AB în raportul k. Apoi, se afișează punctele A, B, C, împreună cu notațiile lor.

Programul de față realizează același lucru utilizând tipul abstract *gtextpunct* implementat în exercițiul 29.18. În acest caz, se definesc două perechi de coordonate la instanțierea obiectelor clasei *gtextpunct*: o pereche pentru coordonatele punctului și cealaltă pereche pentru a defini începutul cimpului în care se afișeză notația punctului.

La afișare, coordonatele de început ale cimpului în care se afișeză textul se pot modifica cu ajutorul parametrilor.

În programul de față nu se utilizează acești parametri (ei au valorile implicate egale cu zero).

## PROGRAMUL BXXIX19

```

#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIX18.CPP"

main()
/* - citeste coordonatelor punctelor A și B;
   - citeste pe k diferit de -1;
   - determină punctul C care împarte segmentul AB în raportul k;
   - afișează punctele A, B și C.
*/
{
    char t[255];
    punct A,B;

```

```

// citeste coordonatele lui A si B
if(A.citpct("A") == 0) exit(1);
if(B.citpct("B") == 0) exit(1);
double k;

// citeste valoarea lui k
for(;;){
    printf("raportul k=");
    if(gets(t) == 0) exit(1);
    if(sscanf(t,"%lf",&k) == 1 && k != -1) break;
    printf("nu s-a tastat un numar diferit de -1\n");
}

punct C = (A+k*B)/(1+k);
gpunct GA = A; // culoarea implicita (WHITE)
gpunct GB = B; // culoarea implicita (WHITE)
gpunct GC(C,LIGHTGREEN);
gtext GTA("A",A-10,YELLOW);
gtext GTB("B",B+gpunct(0,10),YELLOW);
gtext GTC("C",C+gpunct(10,-10),LIGHTGREEN);
gtextpunct GTPA(GA,GTA);
gtextpunct GTPB(GB,GTB);
gtextpunct GTPC(GC,GTC);
int gd = DETECT,gm;
initgraph(&gd, &gm, "c:\\borlandc\\bgi");

GTPA.afisgtextpunct();
GTPB.afisgtextpunct();
GTPC.afisgtextpunct();
getch();
closegraph();
}

```

## 29.5. Clase virtuale

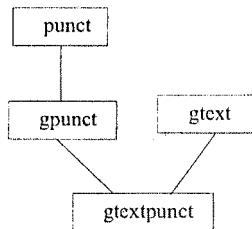
În exercițiul 29.9 a fost definită clasa *gtextpunct* ca o clasă derivată din clasele *gpunct* și *gtext*. Clasa *gpunct* este și ea o clasă derivată și anume, ea derivă din clasa *punct*.

Se obține o ierarhie de clase ca în figura alăturată.

Clasa *punct* are ca date membru două date de tip double *x0* și *y0*, care se folosesc pentru a defini poziția unui punct pe ecranul grafic. Aceste date sunt moștenite de către clasa *gpunct* și prin intermediul ei și de către clasa *gtextpunct*.

Clasa *gtext* conține date membru utilizate la afișarea textelor pe ecranul grafic.

Clasa *gtext* nu are date membru care să definească poziția cimpului în care să se afișeze textul obiectului de tip *gtext*. Cimpul respectiv se definește la apelul funcției membru *afisgtext* cu ajutorul parametrilor.



Clasa *gtextpunct* derivată ca mai sus, se utilizează pentru a afișa un punct pe ecranul grafic, împreună cu un text. Poziția punctului este definită de datele membru *x0* și *y0* moștenite de la clasa *punct*, iar poziția textului se definește cu ajutorul parametrilor la apelul funcției membru *afisgtextpunct*.

Aceeași clasă *gtextpunct* se definește și în exercițiul 29.18, tot ca o clasă derivată din clasele *gpunct* și *gtext*.

De data aceasta, clasa *gtext* este alta decât cea utilizată în exercițiul 29.9. Clasa *gtext* a fost definită în exercițiul 29.17 ca o clasă derivată din clasa *punct*. În felul acesta, clasa *gtext* moștenește datele membru *x0* și *y0* ale clasei *punct*. Aceste date definesc punctul de inceput al cimpului în care se afișează textul obiectului curent. Aceste date pot fi modificate cu ajutorul parametrilor, la apelul funcției membru *afisgtext*.

În acest caz, se obține următoarea ierarhie de clase:

În această ierarhie clasa *gtextpunct* moștenește două perechi de date membru *x0* și *y0* de la clasa *punct*:

- una prin intermediul clasei de bază *gpunct* a clasei *gtextpunct*

și

- una prin intermediul clasei de bază *gtext* a aceleiași clase *gtextpunct*.

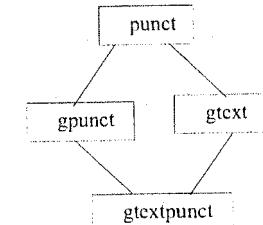
Multiplicarea unor date membru apare în mod frecvent în cazul moștenirii multiple, ca în exemplul de mai sus. Astfel de moșteniri impun instrucțiuni suplimentare pentru a elimina ambiguitățile.

De exemplu, în cazul clasei *gtextpunct*, nu se pot face referiri simple la datele moștenite *x0*, *y0*. O pereche de date *x0* și *y0* se folosește pentru a defini poziția pentru afișarea punctului pe ecranul grafic și acestea sint cele moștenite cu ajutorul clasei *gpunct*, iar cealaltă pereche de date *x0* și *y0* se folosește pentru a defini poziția textului de afișat. Înlăturarea ambiguității, la utilizarea celor două perechi de date *x0* și *y0* moștenite, se poate realiza așa cum s-a procedat la definirea constructorului de copiere sau la supraincărcarea operatorului = pentru clasa *gtextpunct* (vezi exercițiul 29.18).

Într-îndată care se punc în această situație este cea referitoare la definirea clasei *gtext*. Această clasă trebuie să fie o clasă derivată a clasei *punct* sau nu?

În anumite situații este bine ca această clasă să definească poziția în care se afișează textul obiectului curent folosind datele membru *x0*, *y0* moștenite de la clasa *punct*. Astfel de situații intervin atunci cind clasa *gtext* se utilizează pentru a afișa texte independente de clasele care afișează figuri geometrice, cum sint de exemplu clasele *gtext* și *gsegment* definite în exemplele din acest capitol.

În caz contrar, nu este necesar ca clasa *gtext* să aibă date membru moștenite de la clasa *punct* care să precizeze poziția textului pe ecran, poziția respectivă fiind cea a punctului curent care se afișează împreună cu textul, eventual modificată cu ajutorul unor parametri în momentul afișării textului respectiv. De



aici rezultă că ar fi bine să existe două clase pentru afişarea textelor:

- una care să moştenească datele membru  $x_0, y_0$  ale clasei *punct* pentru a defini poziția textului pe ecran și
- una care să nu aibă nevoie de datele respective.

Limbajul C++ oferă utilizatorului o soluție simplă pentru a realiza ambele variante cu o singură clasă *gtext*. Aceasta se realizează cu ajutorul *claselor virtuale*.

Clasile obișnuite devin virtuale în procesul de derivare. Această transformare se realizează pentru clasele de bază care intervin într-o derivare. În acest scop, numele clasei de bază, utilizat la definirea unei clase derivate, va fi precedat de cuvintul cheie *virtual*.

Fie de exemplu, clasa A derivată din clasele de bază B1 și B2, care la rindul lor sunt derivate din clasa de bază B:

```
class B1:public B { ... };
class B2:public B { ... };
class A :public B1,public B2 { ... };
```

Așa cum s-a afirmat mai sus, clasa A moștenește datele membru ale lui B de două ori. Pentru a le moșteni o singură dată, transformăm clasa de bază B în clasă virtuală, ca mai jos:

```
class B1:virtual public B { ... };
class B2:virtual public B { ... };
```

În acest caz, clasa A moștenește datele membru ale lui B o singură dată. Acestea se moștenesc prin intermediul clasei de bază care este scrisă prima în lista claselor de bază. Deci, în exemplul de mai sus, datele membru ale clasei de bază virtuală B sunt moștenite prin intermediul clasei de bază B1.

Dacă clasa A se definește ca mai jos:

```
class A:public B2,public B1 { ... };
```

atunci clasa A moștenește datele membru ale clasei de bază virtuale B prin intermediul clasei de bază B2.

Revenind la clasa *gtextpunct*, vom considera definiția:

```
class gtextpunct:public gpunct,public gtext { ... };
```

unde clasele *gpunct* și *gtext* sunt derivate din clasa *punct* astfel:

```
class gpunct:virtual public punct { ... };
class gtext:virtual public punct { ... };
```

În felul acesta, clasa *gtextpunct* moștenește o singură dată datele membru  $x_0, y_0$  ale clasei *punct*. Moștenirea datelor respective se face prin intermediul clasei de bază *gpunct*.

În legătură cu clasele virtuale trebuie să amintim că există unele particularități în legătură cu ordinea de apel a constructorilor la instantierea obiectelor clasei

derivate.

Ca regulă generală, intuiții se apelează constructorii claselor virtuale în ordinea în care au fost scrise clasele în lista claselor de bază ale clasei derivate. După apelul constructorilor claselor virtuale, se apelează constructorii claselor de bază nevirtuale în ordinea în care clasele respective apar în lista claselor de bază a clasei derivate.

Într-o ierarhie de clase, o clasă poate să apară ca și clasă de bază de mai multe ori. Dacă o clasă de bază apare într-o ierarhie de mai multe ori, atât ca virtuală, cât și ca nevirtuală, atunci la instantierea unui obiect al clasei derivate se va apela constructorul clasei de bază o singură dată pentru toate aparițiile virtuale ale clasei și de atâtea ori cite apariții nevirtuale are clasa de bază respectivă.

### Exerciții:

29.20 Să se modifice clasa *gtextpunct* definită în exercițiul 29.18 folosind clasa de bază virtuală *punct*. Datele membru  $x_0, y_0$  ale clasei *punct* sunt moștenite de clasele *gpunct* și *gtext* derivate din clasa virtuală *punct*.

Clasa *gtextpunct* este derivată, în mod obișnuit, din clasele de bază *gpunct* și *gtext*. Ea moștenește datele membru  $x_0$  și  $y_0$  prin intermediul clasei *gpunct*.

Funcția membru *afisgtextpunct* a clasei *gtextpunct*, afișeză pe ecranul grafic punctul de coordonate  $(x_0, y_0)$  moștenit prin intermediul clasei *gpunct*, precum și textul obiectului curent. Textul se afișează într-un cimp care începe în punctul de coordonate  $(x_0+dx, y_0+dy)$ , unde  $dx$  și  $dy$  sunt parametrii funcției *afisgtextpunct*.

Punctul și textul, afișate cu ajutorul funcției *afisgtextpunct* pot fi facute invizibile apelând funcția membru *stergegtextpunct*.

### FIŞIERUL BXXIX20.H

```
#ifndef __PUNCT_H
#include "BXXIX7.CPP"
#define __PUNCT_H
#endif
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif

class gpunct:virtual public punct {
protected:
    int culoare;
    int vizibil;
public:
    gpunct(double abs=0,double ord=0,int c=WHITE);
    gpunct(const punct&,int c=WHITE);

    // constructor de copiere
    gpunct(const gpunct&);
```

```

void afiseaza();
void sterge();

// supraincarca operatorul de atribuire
gpunct& operator = (const gpunct&);

};

#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#define __GPUNCT_H
#include "BXXIX2.CPP"

class gtext:virtual public punct {
protected:
    int font;
    int directie;
    int dimensiune_caracter;
    int cadrat_oriz;
    int cadrat_vert;
    int culoare_text;
    char text[11];
    int vizibil_text;
public:
    gtext();
    gtext(char *sir,double x=0,double y=0,int c=WHITE,
          int f=DEFAULT_FONT,int dir=HORIZ_DIR,int dim=1,
          int coriz=CENTER_TEXT,int cvert=CENTER_TEXT);

    void afisgtext(double dx=0,double dy=0);
    /* afiseaza textul definit de obiectul curent in cimpul care incepe in punctul de
     * coordonate
     * (x0+dx,y0+dy) */

    void stergetext(double dx=0,double dy=0);
    /* face invizibil textul care este afisat in cimpul care incepe in punctul de coordonate
     * (x0+dx,y0+dy) */

    gtext(const gtext&); //constructor de copiere
    gtext& operator = (const gtext&); // supraincarca operatorul de atribuire
};

class gtextpunct:public gpunct,public gtext {
protected:
    int vizibil_punct_text;
public:
    gtextpunct(); //constructor implicit
    gtextpunct(char *sir,double x=0,double y=0,int cp=WHITE,
               int ct=WHITE,int f=DEFAULT_FONT,int dir=HORIZ_DIR,
               int dim_car=1,int coriz=CENTER_TEXT,
               int cvert=CENTER_TEXT);
    /* constructor utilizat la initializari si conversii */
}

```

```

gtextpunct(const gtextpunct&);
// constructor de copiere

void afisgtextpunct(double dx,double dy);
/* - afiseaza punctul de coordonate (x0,y0);
   - afiseaza textul obiectului curent in cimpul care incepe in punctul de coordonate
   * (x0+dx,y0+dy)
void stergetextpunct(double dx,double dy);
/* face invizibil punctul de coordonate (x0,y0) si textul din
   cimpul care incepe in punctul de coordonate (x0+dx,y0+dy)
   */

gtextpunct& operator=(const gtextpunct&);
// supraincarca operatorul =

};

Mai jos, se definesc functiile membru ale clasei gtextpunct, intr-un fisier de extensie CPP.

```

## FISIERUL BXXIX20

```

#include "BXXIX20.H"

inline gpunct::gpunct(const gpunct& gp):punct(gp)
//constructor de copiere
{
    culoare = gp.culoare;
    vizibil = gp.vizibil;
}

gpunct& gpunct::operator = (const gpunct& gp)
// supraincarca operatorul de atribuire
{
    if(this != &gp){
        x0=gp.x0; y0=gp.y0;
        culoare=gp.culoare;
        vizibil=gp.vizibil;
    }
    return *this;
}

#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#define __GTEXT_H

gtext::gtext() // constructor implicit
{
    text[0]='\0';
    x0=0; y0=0;
    culoare_text = WHITE;
    font = DEFAULT_FONT;
    directie = HORIZ_DIR;
}

```

```

dimensiune_caracter = 1;
cadraj_oriz = CENTER_TEXT;
cadraj_vert = CENTER_TEXT;
vizibil_text = 0;
} // sfarsit constructor implicit

gtext::gtext(char *sir,double x,double y,int c,int f,int dir,
            int dim,int coriz,int cvert):punct(x,y)
/* constructor utilizat la initializari si conversie din sir de caractere in tipul gtext */
{
    strncpy(text,sir,11);
    text[10]='\0';
    culoare_text = c;
    font = f;
    directie = dir;
    dimensiune_caracter = dim;
    cadraj_oriz = coriz;
    cadraj_vert = cvert;
    vizibil_text = 0;
} // sfarsit constructor

gtext::gtext(const gtext& gt):punct(gt) // constructor de copiere
{
    strncpy(text,gt.text,11);
    text[10]='\0';
    culoare_text = gt.culoare_text;
    font = gt.font;
    directie = gt.directie;
    dimensiune_caracter = gt.dimensiune_caracter;
    cadraj_oriz = gt.cadraj_oriz;
    cadraj_vert = gt.cadraj_vert;
    vizibil_text = gt.vizibil_text;
} // sfarsit constructor de copiere

void gtext::afisgtext(double dx,double dy)
/* afisaza textul definit de obiectul curent in cimpul care incepe in
punctul de coordonate (x0+dx,y0+dy) */
{
    struct textsettingstype car_crt;
    int culoare_crt;

    // se salvaza caracteristicile curente
    gettextsettings(&car_crt);
    culoare_crt = getcolor();

    // se seteaza caracteristicile obiectului
    settextstyle(font,directie,dimensiune_caracter);
    settextjustify(cadraj_oriz,cadraj_vert);
    setcolor(culoare_text);

    // afisaza textul
    outtextxy(x0+dx,y0+dy,text);
}

```

```

// se refac caracteristicile
settextstyle(car_crt.font,car_crt.direction,
            car_crt.charsize );
settextjustify(car_crt.horiz,car_crt.vert);
setcolor(culoare_crt);
vizibil_text = 1; // textul este vizibil
} // sfarsit afisgtext

void gtext::stergegtext(double dx,double dy)
/* face invizibil textul obiectului curent care este afisat in cimpul care
incepe in punctul de coordonate (x0+dx,y0+dy) */
{
    int ctemp=culoare_text;
    // salveaza culoarea textului obiectului curent

    culoare_text = getbkcolor();
    // culoarea obiectului curent devine culoarea fondului

    afisgtext(dx,dy); // textul devine invizibil
    culoare_text=ctemp; // se refac culoarea obiectului curent
    vizibil_text=0; // textul este invizibil
}

gtext& gtext::operator = (const gtext& gt) // supraincarca operatorul =
{
    if(this != &gt){
        x0=gt.x0; y0=gt.y0;
        strncpy(text,gt.text,11);
        text[10]='\0';
        culoare_text = gt.culoare_text;
        font = gt.font;
        directie = gt.directie;
        dimensiune_caracter = gt.dimensiune_caracter;
        cadraj_oriz = gt.cadraj_oriz;
        cadraj_vert = gt.cadraj_vert;
        vizibil_text = gt.vizibil_text;
    }
    return *this;
} // sfarsit operator=

inline gtextpunct::gtextpunct() // constructor implicit
{
    vizibil_punct_text = 0;
} // sfarsit constructor implicit

gtextpunct::gtextpunct(char *sir,double x,double y,int cp,
                      int ct,int f,int dir,int dim_car,int coriz,int cvert)
/* constructor pentru initializari si conversii */
{
    strncpy(text,sir,11);
    text[10] = '\n';
    x0 = x; y0 = y;
    culoare = cp;
    culoare_text = ct;
}

```

```

font = f;
directie = dir;
dimensiune_caracter = dim_car;
cadraj_oriz = coriz;
cadraj_vert=cvert;
vizibil = 0;
vizibil_text = 0;
vizibil_punct_text = 0;
} //sfirsit constructor pentru initializari si conversii

gtextpunct::gtextpunct (const gtextpunct& gtp) //constructor de copiere
{
    double punct::*p;
    p = &punct::x0;
    punct pp = gtp;
    x0 = pp.*p;
    p = &punct::y0;
    y0 = pp.*p;
    culoare = gtp.culoare;
    vizibil = gtp.vizibil;
    strncpy(text,gtp.text,11);
    text[10] = '\0';
    culoare_text = gtp.culoare_text;
    font = gtp.font;
    directie = gtp.directie;
    dimensiune_caracter = gtp.dimensiune_caracter;
    cadraj_oriz = gtp.cadraj_oriz; cadraj_vert = gtp.cadraj_vert;
    vizibil_text = gtp.vizibil_text;
    vizibil_punct_text = gtp.vizibil_punct_text;
} //sfirsit constructor de copiere

void gtextpunct::afisgtextpunct (double dx,double dy)
/* afiseaza punctul de coordonate (x0,y0) si textul obiectului curent in
cimpul care incepe in punctul de coordonate
(x0+dx,y0+dy)*/
{
    afiseaza(); // afiseaza punctul (x0,y0)
    afisgtext (dx,dy);

/* afiseaza textul in cimpul care incepe in punctul de coordonate
(x0+dx,y0+dy)*/

    vizibil_punct_text = 1;
} //sfirsit afisgtextpunct

void gtextpunct::stergegtextpunct (double dx,double dy)
/* face invizibil punctul si textul obiectului curent */
{
    sterge(); //face invizibil punctul
    stergegtext (dx,dy); // face textul invizibil
    vizibil_punct_text = 0;
} //sfirsit stergegtextpunct

```

```

gtextpunct& gtextpunct::operator = (const gtextpunct& gtp)
// supraincarca operatorul =
{
    if(this != &gtp){
        double punct::*p;
        p = &punct::x0;
        punct pp=gtp;
        x0 = pp.*p;
        p = &punct::y0;
        y0 = pp.*p;
        culoare = gtp.culoare;
        vizibil = gtp.vizibil;
        strncpy(text,gtp.text,11);
        text[10] = '\0';
        culoare_text = gtp.culoare_text;
        font = gtp.font;
        directie = gtp.directie;
        dimensiune_caracter = gtp.dimensiune_caracter;
        cadraj_oriz = gtp.cadraj_oriz;
        cadraj_vert = gtp.cadraj_vert;
        vizibil_text = gtp.vizibil_text;
        vizibil_punct_text = gtp.vizibil_punct_text;
    }
    return *this;
} // sfirsit operator=

```

### 29.21 Să se scrie un program care realizează următoarele:

- citește, de la intrarea standard, coordonatele a două puncte A și B;
- afișează punctele A și B împreună cu numele lor;
- afișează punctele care împart segmentul AB în raportul  $k$ , unde  $k$  ia valori în intervalul  $[0,5;1,5]$  cu pasul 0,1;
- ultimul punct este afișat împreună cu litera C.

La afișarea punctelor A și B se folosește culoarea implicită. Punctele care împart segmentul AB în raportul  $k$  se afișează folosind culoarea definită de constanta simbolică LIGHTGREEN. Litera C se afișează folosind culoarea definită de constanta simbolică YELLOW. Pozițiile literelor A, B, C se stabilesc în dependență de pozițiile punctelor pe care le notează, astfel:

- punctul A:  $x0-10,y0$ ;
- punctul B:  $x0,y0+10$ ;
- punctul C:  $x0,y0-10$

unde:

$(x0,y0)$  - Sunt coordonatele punctului curent.

### PROGRAMUL BXXIX21

```

#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STRING_H

```

```

#include <string.h>
#define __STRING_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __DOS_H
#include <dos.h>
#define __DOS_H
#endif
#include "BXXIX20.CPP"

main()
/* - citeste coordonatele punctelor A si B;
   - afiseaza punctele A si B impreuna cu numele lor;
   - afiseaza punctele care impart segmentul AB in raportul k; k ia valori
     in intervalul [0,5;1,5] cu pasul 0,1; ultimul punct afisat este notat cu litera C */
{
    punct A,B;

    if(A.citpct("A")==0) exit(1); // s-a tastat EOF
    if(B.citpct("B")==0) exit(1); // s-a tastat EOF
    int gd=DETECT,gm;
    initgraph (&gd,&gm,"C:\\borland\\bgi");
    gtextpunct GTA("A",A.retabs(),A.retord());
    gtextpunct GTB("B",B.retabs(),B.retord());
    GTA.afisgtextpunct(-10,0); // afiseaza A
    GTB.afisgtextpunct(0,10); // afiseaza B
    for(double k=0.5;k < 1.6;k += 0.1){
        punct C=(A+k*B)/(1+k);
        gpunct GC(C,LIGHTGREEN);
        GC.afiseaza();
        delay(100);
        if(k < 1.5) continue;
        gtextpunct GTC("C",C.retabs(),C.retord(),LIGHTGREEN,
                      YELLOW);
        GTC.afisgtextpunct(0,-10); // afiseaza C
        delay(100);
    }
    getch();
    closegraph();
}

```

## 30. FUNCȚII VIRTUALE

În capitolele 22 și 27 s-a arătat că, o funcție membru nestatică a unei clase se apelează, totdeauna, pentru un obiect al clasei pentru care funcția respectivă este funcție membru.

Legătura dintre funcție și obiectul pentru care se apelează funcția se realizează cu operatorul punct (.) sau săgeată (->). Obiectul pentru care se realizează apelul se numește obiect curent.

### Exemplu:

Fie definiția de clasă:

```

class cl {
public:
    ... f(...);
    ... g(...);
    ...
};

```

unde *f* și *g* sunt funcții membru nestatice, și declarațiile:

```

cl obcl;
cl *pcl;

```

Atunci funcțiile *f* și *g* pot fi apelate numai în construcții de felul celor de mai jos:

```

obcl.f(...)
obcl.g(...)

```

sau

```

pcl = &obcl;
pcl -> f(...);
pcl -> g(...);

```

O funcție membru nestatică poate fi apelată în corpul unei alte funcții nestatice fără a mai indica obiectul pentru care se face apelul. Acesta va fi în mod implicit obiectul curent. De exemplu, dacă funcția *f*, din exemplul de mai sus, apelează funcția *g* în felul următor:

```

... f(...)

...
g(...)

...
}

```

atunci la apelul:

```

obcl.f(...)

```

funcția *f* apelează funcția *g* pentru obiectul curent, adică pentru *obcl*.

Acet mecanism se realizează foarte simplu cu ajutorul pointerului implicit *this* care este definit în mod implicit în corpul oricărei funcții membru nestatică. El are ca valoare chiar adresa obiectului curent.

Înseamnă că la apelul de mai sus, în corpul funcției *f*, *this* are ca valoare adresa lui *obcl*. Apelul lui *g* în corpul funcției *f* se realizează în mod implicit sub forma:

```
this -> g(...)
```

de unde rezultă că *g* se apelează pentru obiectul curent.

Regula de apel a funcțiilor membru nestatică are și alte posibilități în cazul în care clasele formează ierarhii.

Să presupunem că A este o clasă derivată din clasa de bază B:

```
class B {
protected:
...
public:
...
    ... fb(...); // funcție membru nestatică
}

class A:public B {
public:
...
    ... fa(...); // funcție membru nestatică
}
```

Fie declarațiile:

```
B obb;
B *pb;
A oba;
A *pa;
```

Conform celor spuse mai sus, următoarele apeluri sunt corecte:

```
obb.fb(...)

...
pb = &obb;
pb -> fb(...)

...
oba.fa(...)

...
pa = &oba;
oba -> fa(...)

...
```

Întrucit un obiect al clasei A este în același timp și un obiect particularizat al clasei B, sunt corecte și apelurile funcției *fb* cu obiecte ale clasei A:

```
oba.fb(...)
```

```
...
pa = &oba;
pa -> fb(...)
```

Corectitudinea acestor apeluri rezultă din faptul că funcția *fb* este moștenită de clasa derivată A.

Apeluri de acest fel întâlnim în exercițiile din capitolul 29. De exemplu, funcția *afisgtextpunct* care este o funcție membru nestatică a clasei *gtextpunct* se definește în exercițiul 29.20, astfel:

```
void gtextpunct::afisgtextpunct(double dx,double dy)
{
    afiseaza();
    afisgtext(dx,dy);
    ...
}
```

Funcția *afiseaza* apelată în corpul funcției *afisgtextpunct*, este o funcție membru nestatică a clasei *gpunct* care este o clasă de bază a clasei *gtextpunct*.

În mod analog, funcția *afisgtext* apelată în corpul aceleiasi funcții, este o funcție membru nestatică a clasei *gtext* care și ea este o clasă de bază a clasei *gtextpunct*.

Ambele funcții (*afiseaza* și *afisgtext*), se apelează pentru obiectul curent care este un obiect al clasei *gtextpunct* deoarece apelurile din corpul funcției *afisgtextpunct* se realizează sub forma:

```
this -> afiseaza()
și
this -> afisgtext(dx,dy)

Apelurile de forma:
obb.fa(...)

și
pb = &obb;
pb -> fa(...)
```

sunt eronate. Într-adevăr, funcția *fa* nu este funcție membru a clasei B și *fa* nu poate fi apelată pentru un obiect care este o instanțiere a clasei B.

Să observăm că, o secvență de forma:

```
pa = &obb;
pa -> fa(...)
```

conduce la eroare, deoarece un pointer spre un obiect al unei clase de bază nu se poate atribui unui pointer spre o clasă derivată. Cu toate acestea, atribuirea de mai sus poate fi forțată printr-o conversie explicită a pointerelor:

```
pa = (A *) &obb;
```

În continuare se poate realiza apelul:

```
pa -> fa(...)
```

Apelurile de acest fel nu sunt recomandate deoarece, de obicei, conduc la erori.

O altă problemă care prezintă interes într-o ierarhie de clase este cea cu privire la supraincărcarea funcțiilor.

Am văzut că în limbajul C++ se pot defini mai multe funcții cu același nume, diferența dintre ele făcându-se după numărul și/sau tipul parametrilor.

Se pot defini funcții membru nestaticice cu același nume și chiar și cu aceeași listă de parametri (vidă sau nu) pentru clase diferite. Aceasta nu constituie o ambiguitate deoarece funcțiile nu se apelează independent, ci legate de un obiect și în felul acesta se selectează, în mod unic, funcția membru care se apelează. Numai funcțiile membru ale aceleiași clase sau cele care nu sunt funcții membru trebuie să difere prin numărul și/sau tipul parametrilor.

#### Exemplu:

Pentru clasele *complex* și *rational* se definește funcția *afiseaza*, fără parametri:

```
class complex {
    double real;
    double imag;
public:
    complex(double x=0,double y=0)
    {
        real=x; imag=y;
    }
    void afiseaza()
    {
        printf("%g+i*(%g)\n",real,imag);
    }
    ...
};

class rational {
    long numarator;
    long numitor;
public:
    rational(long a=0,long b=1)
    {
        numarator = a;
        if(b) numitor = b;
        else numitor = 1;
    }
    void afiseaza()
    {
        printf("(%ld)/(%ld)\n",numarator,numitor);
    }
    ...
};
```

Fie instanțierile:

```
complex z(1,2); // z= 1+2i
rational f(1,2); // f=1/2
...
a.afiseaza(); // sc afiseaza 1+i*(2)
f.afiseaza(); // sc afiseaza (1)/(2)
```

Clasele *complex* și *rational* au, fiecare, cîte o funcție membru *afiseaza* nestatică. Ele nu au parametri. Se apeleză în legătură cu un obiect și acesta permite selectarea lor. Astfel, dacă funcția *afiseaza* se apeleză pentru un obiect *complex*, atunci se apeleză funcția membru *afiseaza* a clasei *complex*. În mod analog, la un apel al funcției *afiseaza* cu un obiect al clasei *rational*, se apeleză funcția membru *afiseaza* a clasei *rational*.

La definirea clasei *complex* se pot defini și alte funcții *afiseaza*, dar acestea vor avea parametri. De exemplu, putem defini pentru clasa *complex*, alături de funcția *afiseaza* de mai sus, următoarea funcție membru:

```
void afiseaza(char *f)
{
    if(f && *f) printf(f,real,imag);
    else printf("%g+i*(%g)\n",real,imag);
}
```

În cazul ierarhiilor de clase se moștenesc în clasa derivată funcții membru din clasa de bază. Aceasta ne permite să moștenim supraincărările operatorilor definite în clasele de bază. De asemenea, se pot supraincărca, în clasa derivată, funcții definite în clasele de bază.

În acest caz, funcțiile membru ale clasei deriveate care supraincarcă funcții din clasele de bază pot dифeri de acestea prin numărul și/sau tipul parametrilor, dar pot și să aibă aceeași listă de parametri (de exemplu, este frecvent cazul cind lista parametrilor este vidă).

În cazul cind funcțiile au aceeași listă de parametri, apar situațiile care se precizează mai jos.

Fie clasele B și A definite astfel:

```
class B {
    ...
public:
    ... f(...);
    ...
};

class A:public B {
    ...
public:
    ... f(...);
    ...
};
```

Funcția *f* este definită în clasa de bază *B* și este suprainscrisă cu aceeași listă de parametri (care poate fi și vidă) în clasa derivată *A*.

Fie declarațiile:

```
B obb;  
B *pb;  
A oba;  
A *pa;  
...
```

La apelurile:

```
obb.f(...)
```

și

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

se apelează funcția membru *f* a clasei *B*.

La apelurile:

```
oba.f(...)
```

și

```
pa->f(...)
```

se apelează funcția membru *f* a clasei *A*.

Selectările de mai sus ale funcției *f* se fac în mod obișnuit ținând seama de obiectul pentru care se face apelul și anume: se selectează funcția membru a clasei căreia îi aparține obiectul curent.

Se pune acum problema apelării funcției membru *f* a clasei de bază *B*, pentru un obiect al clasei deriveate *A*.

Am văzut că acest lucru, în general, este posibil (vezi apelurile funcției *fb* de mai sus) și anume atunci cind funcțiile membru ale celor două clase au nume diferite. În cazul de față se utilizează operatorul de rezoluție:

```
oba.B::f(...)
```

La acest apel se va utiliza, pentru obiectul *oba* al clasei deriveate *A*, funcția membru *f* a clasei de bază *B*.

**Exemplu:**

```
class punct {  
protected:  
    double x0,y0;  
public:  
    punct(double a,double b)  
    {  
        x0 = a; y0 = b;  
    }  
    ...  
    void afiseaza()  
    {  
        printf("(%.2f,%.2f)\n",x0,y0);  
    }  
}
```

```
...  
};  
  
class segment:public punct {  
protected:  
    punct pct;  
public:  
    segment(double a,double b,double c,double d) :  
        punct(a,b),pct(c,d)  
    {  
    }  
    ...  
    void afiseaza()  
    {  
        punct::afiseaza();  
        pct.afiseaza();  
    }  
    ...  
};
```

Funcția *afiseaza* este definită ca funcție membru în clasa de bază *punct* și este suprainscrisă în clasa derivată *segment*.

Fie declarațiile:

```
punct a(100,100);  
segment ab(20,20,200,200);
```

Apelul:

```
a.afiseaza()
```

apeleză funcția membru *afiseaza* a clasei *punct*.

Ca rezultat se afișeză: (100,100)

Apelul:

```
ab.afiseaza()
```

apeleză funcția membru *afiseaza* a clasei *segment*. Ca rezultat se afișeză:

```
(20,20)  
(200,200)
```

Apelul:

```
ab.punct::afiseaza()
```

apeleză funcția membru *afiseaza* a clasei *punct*. Ca rezultat se afișeză datele membru moștenite, adică:

```
(20,20)
```

Să observăm modul de realizare al funcției membru *afiseaza* a clasei *segment*.

Prima instrucțiune este:

```
punct::afiseaza();
```

Ea trebuie interpretată sub forma:

```
this -> punct::afiseaza()
```

adică se apelează, pentru obiectul curent (care este de tip *segment*), funcția membru *afiseaza* a clasei *punct*.

Ca rezultat al acestui apel se va afișa:

(x0,y0)

adică datele membru moștenite de la clasa *punct*.

Cea de a doua instrucțiune:

```
pct.afiseaza();
```

se interpretează sub forma:

```
this -> pct.afiseaza();
```

Aceașa înseamnă că funcția *afiseaza* se apelează pentru data membru *pct* a obiectului curent. Acesta este de tip *punct* și de aceea se apelează funcția membru *afiseaza* a clasei *punct*.

Să observăm că funcția membru *afiseaza* a clasei *segment*, definită ca mai jos:

```
void afiseaza()
{
    afiseaza();
    pct.afiseaza();
}
```

reprezintă un apel recursiv infinit al funcției membru *afiseaza* a clasei *segment*.

Într-adevăr, prima instrucțiune se realizează sub forma:

```
this -> afiseaza();
```

adică se apelează funcția *afiseaza* pentru obiectul curent. Acesta este o instanțiere a clasei *segment*, deci apelul respectiv este o reapelare recursivă a funcției *afiseaza* a clasei *segment*.

Ca o concluzie generală a celor spuse mai sus este faptul că selectările funcțiilor suprăincărcate se tratează, în toate cazurile care au fost analizate, în momentul *compilării*. Ele nu se modifică la execuție și de aceea, astfel de selectări se numesc *static*e. O altă interpretare este aceea că funcția apelată este legată de obiectul pentru care se apelează *timpuriu* (în fază compilării, spre deosebire de legătura *intirziată* care se face la execuție).

Există cazuri cind selectările funcțiilor membru nu se pot face la compilare. Astfel de situații apar frecvent la instanțierea și utilizarea obiectelor de tip *container*.

Un obiect de tip *container* este un set de obiecte neomogene (care nu sunt de un același tip) care aparțin claselor unei ierarhii. Obiectele de acest fel se implementează prin tablouri sau liste care, de obicei, se numesc *eterogene*.

Un tabou eterogen este un tabou obișnuit de pointeri spre tipul situat în rădăcina ierarhiei.

Fie B clasa situată în virful ierarhiei de clase și A1,A2,...,Am clase situate pe diferite niveluri ale ierarhiei. Atunci, un tabou eterogen utilizat pentru a implementa un container de obiecte care aparțin claselor ierarhiei, se declară astfel:

```
B *tab[max];
```

unde:

max

- Este o constantă care definește numărul maxim de obiecte ale containerului.

Definim clasa *container*:

```
class container {
    B *tab[100];
    int n; // indicele elementului liber
public:
    container()
    {
        n = 0;
    }

    int atrib(B *bp)
    {
        if(n < 100){
            tab[n++] = bp;
            return 1;
        }
        else return 0; // container plin
    }

    B *acceselem(int i)
    {
        if(0 <= i && i < n) return tab[i];
        else return 0; // pointerul nul
    }
};
```

Elementele lui *tab* sunt pointeri spre obiecte de tip B. Cum A1,A2,...,Am sunt clase derivate direct sau indirect din B, rezultă că obiectele claselor A1,A2,...,Am sunt obiecte particulare ale clasei B. De aceea, un pointer spre Ai se atribuie în mod obișnuit unui pointer spre B. Aceasta înseamnă că funcția *atrib* poate fi apelată cu parametri efectivi care sunt pointeri spre oricare din obiectele claselor ierarhiei. În particular, funcția *atrib* poate avea ca parametri efectivi expresii de forma:

&OBAi, i = 1,2,...,m

unde prin OBAi am notat un obiect al clasei Ai.

Fie instanțierea:

```
container AB;
```

Un apel de forma:

```
AB.attrib(&OBAi);
```

atribuie elementului:

```
AB.tab[n]
```

pointerul spre obiectul OBAi al clasei Ai. Totodată se incrementează valoarea lui *n*. Cu alte cuvinte, apelul funcției *attrib* de mai sus, poate fi considerat că pune obiectul OBAi, al clasei Ai, în containerul AB.

În containerul AB se pot pune obiecte ale oricărora din clasele ierarhiei.

Funcția *acceselem* returnează pointerul spre elementul tab[i]. Acest pointer permite accesul la elementul respectiv. Funcția se apelează simplu printr-o expresie de forma:

```
B *p = AB.acceselem(indice);
```

Problema care apare în acest moment este aceea a utilizării pointerului *p*. Aceasta este un pointer spre un obiect al uneia din clasele ierarhiei. De exemplu, dacă *p* are ca valoare adresa unui obiect al clasei Ai, atunci vom utiliza pointerul respectiv convertindu-l în mod explicit ca pointer spre Ai:

```
(Ai *)p
```

Presupunem că funcția *f* este o funcție membru nestică definită în clasa B și supraîncărcată în toate clasele Ai ale ierarhiei și cu aceeași listă de parametri. Atunci:

```
(Ai *)p -> f(...)
```

apeleză funcția membru nestică a clasei Ai, pentru obiectul din container de adresă (Ai \*).p.

Evident, dacă valoarea lui *p* returnată de funcția *acceselem* nu este adresa unui obiect al clasei Ai, atunci conversia:

```
(Ai *)p
```

este eronată și apelul:

```
(Ai *)p -> f(...)
```

va fi și el eronat.

De aceea, accesele de acest fel la obiectele unui container sunt surse de erori care nu pot fi depistate decât destul de greu de către programator.

Programatorul trebuie să țină evidența tipurilor obiectelor păstrate în container și pe baza acesteia să realizeze conversia de pointeri corespunzătoare.

Ar fi mult mai convenabil să putem apela funcțiile *f* fără a mai face conversia explicită a lui *p* spre tipul Ai.

Cu alte cuvinte, un apel de forma:

```
p -> f(...)
```

să apeleze funcția membru nestică *f* a clasei Ai, dacă *p* are ca valoare adresa unui obiect al clasei Ai. Aceasta l-ar scuti pe programator de a mai ține evidența tipurilor obiectelor puse în container și de a face conversii de felul (Ai \*).

Apelul:

```
p -> f(...)
```

poate fi tratat într-un singur mod la compilare și anume se apelează funcția membru *f* a clasei de bază B. Aceasta, deoarece *p* este pointer spre obiecte ale clasei B. Selectarea funcției *f* în funcție de valoarea pointerului *p* se poate realiza numai la execuție. De aceea, pentru ca la apelul de forma:

```
p -> f(...)
```

să se selecteze funcția membru *f* a clasei Ai cind *p* are ca valoare adresa unui obiect al clasei Ai, este necesar ca apelul respectiv să nu fie rezolvat la compilare, ci la execuție. În acest caz, legătura dintre obiect și funcția membru apelată nu mai este statică ci *dinamică*. Se obișnuiește să se mai spună că legătura respectivă este *intărită sau ulterioară* (late binding).

Legăturile de acest fel se realizează cu ajutorul funcțiilor *virtuale*. O funcție *virtuală* este o funcție membru nestică a cărei prototip sau antet este precedat de cuvântul cheie *virtual*. O astfel de funcție se definește ca fiind virtuală în clasa de bază și apoi ea poate fi definită în clasele derivate pentru care este nevoie să se utilizeze legătura dinamică. În clasele derivate nu este nevoie să se mai utilizeze cuvântul cheie *virtual* în prototipul sau antetul funcțiilor virtuale. Deci, dacă funcția *f*, din exemplul de mai sus, este declarată ca virtuală în clasa de bază B:

```
virtual ... f(...);
```

atunci funcția *f* definită cu aceeași listă de parametri este în mod automat virtuală pentru orice clasă Ai derivată direct sau indirect din B.

Funcția *f* apare ca o funcție care are mai multe versiuni, una corespunde clasei de bază, iar celelalte claselor derivate.

Se spune că funcțiile virtuale din clasele derivate se suprapun (în engleză *override*) peste funcțiile virtuale din clasele lor de bază. Astfel, dacă B este o clasă de bază și *f* o funcție membru virtuală a acesteia, iar A este o clasă derivată din B pentru care *f* este redefinită ca funcție membru cu aceeași listă de parametri, atunci vom spune că funcția membru *f* a lui A se suprapune peste funcția membru *f* a lui B.

Nu este obligatoriu ca fiecare clasă derivată să aibă o funcție membru care să se suprapună peste funcția virtuală definită în clasa de bază directă sau indirectă. Dacă o clasă derivată nu are o astfel de funcție membru, atunci ea moștenește funcția membru virtuală corespunzătoare primei clase din ierarhie de pe un nivel precedent și care are definită funcția respectivă.

Selectarea versiunii care să se apeleze se face pe baza unor tabele suplimentare construite în program în mod automat pentru funcțiile virtuale. De asemenea, obiectele claselor care conțin funcții virtuale conțin adresele tabelelor

care le corespund. De aceea, utilizarea funcțiilor virtuale necesită atât memorie suplimentară, cât și un anumit timp suplimentar la execuția programului, ambele fiind necesare pentru selectarea versiunilor funcțiilor virtuale.

Rezultă că funcțiile virtuale, deși sunt avantajoase în multe cazuri, este bine să nu se facă exagerări în utilizarea lor deoarece conduc la un consum sporit de memorie și timp de execuție.

În legătură cu funcțiile virtuale este necesar să ținem seama de următoarele reguli:

1. Funcțiile virtuale sunt funcții membru nestatiche.
2. Constructorii nu pot fi funcții virtuale.
3. Destructorii pot fi funcții virtuale.
4. Funcțiile inline nu pot fi funcții virtuale.

#### Exerciții:

30.1 Să se implementeze o ierarhie de clase pentru următoarele patrulatere: paralelogram, dreptunghi, pătrat și romb.

Obiectele acestor clase sunt patrulatere care au două laturi paralele cu axa  $Ox$ . Clasele au funcții membru care permit realizarea următoarelor operații asupra obiectelor:

- afișare pe ecranul grafic;
- stergere de pe ecranul grafic;
- deplasare pe ecranul grafic;
- calculul perimetrelui și ariei figurii geometrice.

Paralelogramul se află în rădăcina ierarhiei și va fi clasa de bază pentru dreptunghi și romb. Pătratul este o clasă derivată din dreptunghi. În felul acesta se obține ierarhia din figura alăturată.

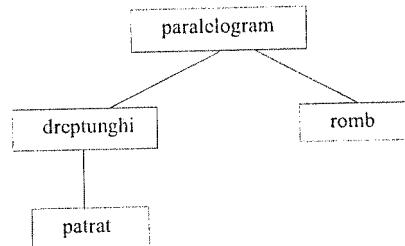
Figurile se definesc cu ajutorul coordonatelor virfurilor. În cazul paralelogramului și rombului se dau coordonatele la toate cele 4 virfuri. În cazul dreptunghiului și pătratului se dau coordonatele a două virfuri diagonale opuse.

Constructorii claselor *paralelogram* și *romb* verifică dacă laturile opuse sunt paralele și dacă segmentul format cu primele două virfuri este paralel cu axa  $Ox$ .

Clasa *paralelogram* are o dată membru numită *valid*, care are valoarea 1 dacă obiectul curent este un paralelogram și zero în caz contrar.

În mod analog, și celelalte clase au date membru care stabilesc validitatea obiectului curent.

Alte date membru ale clasei *paralelogram* definesc caracteristicile pentru afișarea figurilor grafice:



- culoarea liniei;
- și
- grosimea liniei.

#### FIŞIERUL BXXX1A.H

```

#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#ifndef __GSEGMENT_H
#include "BXXIX11.CPP"
#define __GSEGMENT_H
#endif

class paralelogram { // clasa radacina a ierarhiei
protected:
    punct tvirf[4];
    int valid;
    int pvizibil;
    int culoare_latura;
    int grosime_latura;
public:
    int validparalel();
    /* valid = 1 daca obiectul curent este un paralelogram cu 2 laturi paralele cu Ox;
       valid = 0, altfel.
    */
    paralelogram(); // constructor implicit
    paralelogram(double xa,double ya,double xb,double yb,
                 double xc,double yc,double xd,double yd,
                 int c=WHITE,int g=NORM_WIDTH);
    // constructor utilizat la initializari

    paralelogram(const punct& a,const punct& b,
                  const punct &c,const punct& d,
                  int cl=WHITE,int g=NORM_WIDTH);
    // constructor utilizat la initializari

    paralelogram(const paralelogram&); // constructor de copiere

    paralelogram& operator = (const paralelogram&);
    // supraincarca operatorul =

    virtual void afiseaza(); // afiseaza figura curenta

    virtual void sterge(); // face invizibila figura curenta

    virtual double perimetru() const;
    // returneaza perimetruul figurii curente

    virtual double arie() const; // returneaza aria figurii curente
}
  
```

```

void deplaseaza(double dx,double dy);
/* deplaseaza figura curenta modificind coordonatele:
   abscisa cu dx si ordinata cu dy */
};

FISIERUL BXXX1B.H

#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

class romb:public paralelogram {
    int rvalid;
public:
    int validromb();
    /* rvalid=1, daca obiectul curent este romb si are doua laturi paralele cu Ox;
       rvalid=0, altfel. */
    romb(); // constructor implicit
    romb(double xa,double ya,double xb,double yb,
          double xc,double yc,double xd,double yd,
          int c=WHITE,int g=NORM_WIDTH);
    // constructor pentru initializari

    romb(const punct& a,const punct& b,const punct& c,
          const punct& d,int cl=WHITE,int g=NORM_WIDTH);
    // constructor pentru initializari

    romb(const romb&); // constructor de copiere

    romb& operator = (const romb&); // supraincarca operatorul =
    /* - se mostenesc de la clasa paralelogram functiile virtuale:
       afiseaza si sterge
       - se mostenesc functia membru deplaseaza, nevirtuala.
    */
    double perimetru() const;
    /* - returneaza perimetrul rombului curent;
       - este functie virtuala deoarece se suprapune peste functia
         perimetru din clasa de baza care este functie virtuala. */

    double arie() const;
    /* - returneaza aria rombului curent;
       - este functie virtuala deoarece se suprapune peste functia
         arie din clasa de baza care este functie virtuala. */
};

```

### FISIERUL BXXX1C.H

```

#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

```

```

class dreptunghi:public paralelogram {
protected:
    int dvalid;
public:
    int validdrept();
    /* dvalid=1, daca obiectul curent este un dreptunghi si 2 laturi sunt paralele cu Ox;
       dvalid=0, altfel. */
    /*/

    dreptunghi(); // constructor implicit

    dreptunghi(double xa,double ya,double xc,double yc,
               int c=WHITE,int g=NORM_WIDTH);
    // constructor utilizat la initializari

    dreptunghi(const punct& a,const punct& c,
               int cl=WHITE,int g=NORM_WIDTH);
    // constructor utilizat la initializari

    dreptunghi (const dreptunghi&); // constructor de copiere

    dreptunghi& operator = (const dreptunghi&);
    // supraincarca operatorul =

    void afiseaza();
    /* functie virtuala pentru afisarea dreptunghiului curent */

    void sterge();
    /* functie virtuala pentru a face invizibil dreptunghiul curent */

    double perimetru() const;
    /* functie virtuala care returneaza perimetrul figurii curente */

    double arie() const;
    /* functie virtuala care returneaza aria figurii curente */

    // se mosteneste functia obisnuita deplaseaza
};

```

### FISIERUL BXXX1D.H

```

#ifndef __DREPTUNGHI
#include "BXXX1C.H"
#define __DREPTUNGHI
#endif

class patrat:public dreptunghi {
protected:
    int pvalid;
public:
    int validpatrat();
    /* pvalid=1, daca obiectul curent este un patrat si are 2 laturi paralele cu Ox;
       pvalid=0, altfel. */
};

```

```

patrat(); // constructor implicit

patrat(double xa,double ya,double xc,double yc,
       int c=WHITE,int g=NORM_WIDTH);
// constructor utilizat la initializari

patrat(const punct& a,const punct& c,
       int cl=WHITE,int g=NORM_WIDTH);
// constructor utilizat la initializari

patrat(const patrat&); // constructor de copiere

patrat& operator = (const patrat&); // supraincarca operatorul =

/* - se mostenesc functiile virtuale:
   afiseaza si sterge
   de la clasa dreptunghi
   - se mosteneste functia membru deplasare de la paralelogram
*/
double perimetru() const;
/* functie virtuala care returneaza perimetrul obiectului curent */

double arie() const;
/* functie virtuala care returneaza aria obiectului curent */
};

Functiile membru ale ierarhiei de clase se definesc in fisierele de mai jos, de extensie CPP.

```

## FISIERUL BXXX1A

```

#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

// clasa PARALELOGRAM
int paralelogram::validparalel()
/* valid=1, daca obiectul curent are laturile opuse paralele si 2 laturi sunt paralele cu axa Ox */
{
    gsegment OX(1,0,0,0); // axa OX

    // se definesc laturile paralelogramului

    punct A(tvirf[0]);
    punct B(tvirf[1]);
    punct C(tvirf[2]);
    punct D(tvirf[3]);

    gsegment AB(tvirf[0],tvirf[1]);
    gsegment BC(tvirf[1],tvirf[2]);
    gsegment CD(tvirf[2],tvirf[3]);
    gsegment DA(tvirf[3],tvirf[0]);
}

```

```

valid = 0;
if(AB || CD) // segmentele AB si CD sunt paralele
    if(AB || OX) // segmentele AB si CD sunt paralele cu OX
        if(BC || DA) // segmentele BC si DA sunt paralele
            valid = 1;

// test daca punctele sunt diferite
if(A == B) valid = 0;
if(A == C) valid = 0;
if(A == D) valid = 0;
if(B == C) valid = 0;
if(B == D) valid = 0;
if(C == D) valid = 0;
return valid;
}

inline paralelogram::paralelogram() // constructor implicit
{
    valid = 0;
    pvizibil=0;
}

paralelogram::paralelogram(double xa,double ya,double xb,
                           double yb,double xc,double yc,
                           double xd,double yd,int c,int g)
// constructor pentru initializari
{
    tvirf[0] = punct(xa,ya);
    tvirf[1] = punct(xb,yb);
    tvirf[2] = punct(xc,yc);
    tvirf[3] = punct(xd,yd);

    // sc valideaza paralelogramul
    validparallel();
    pvizibil = 0;
    culoare_latura = c;
    grosime_latura = g;
}

paralelogram::paralelogram(const punct& a,const punct& b,
                           const punct& c,const punct& d,
                           int cl,int g)
// constructor folosit la initializari
{
    tvirf[0] = a;
    tvirf[1] = b;
    tvirf[2] = c;
    tvirf[3] = d;

    // sc valideaza paralelogramul
    validparallel();
    pvizibil = 0;
    culoare_latura = cl;
    grosime_latura = g;
}

```

```

}

paralelogram::paralelogram (const paralelogram& p)
// constructor de copiere
{
    for(int i=0;i < 4;i++) tvirf[i]=p.tvirf[i];
    valid = p.valid;
    pvizibil=p.pvizibil;
    culoare_latura = p.culoare_latura;
    grosime_latura = p.grosime_latura;
}

paralelogram& paralelogram::operator = (const paralelogram& p)
// supraincarca operatorul =
{
    if(p.valid && this != &p){
        for(int i=0;i < 4;i++) tvirf[i]=p.tvirf[i];
        valid=p.valid;
        pvizibil=p.pvizibil;
        culoare_latura=p.culoare_latura;
        grosime_latura=p.grosime_latura;
    }
    else
        if(p.valid == 0) valid = 0;
    return *this;
}

void paralelogram::afiseaza()
// functie virtuala care afiseaza paralelogramul curent
{
    if(valid){ // paralelogramul este valid
        gsegment AB(tvirf[0],tvirf[1],culoare_latura,
                     grosime_latura);
        AB.afiseaza(); // afiseaza segmentul AB
        gsegment BC(tvirf[1],tvirf[2],culoare_latura,
                     grosime_latura);
        BC.afiseaza(); // afiseaza segmentul BC
        gsegment CD(tvirf[2],tvirf[3],culoare_latura,
                     grosime_latura);
        CD.afiseaza(); // afiseaza segmentul CD
        gsegment DA(tvirf[3],tvirf[0],culoare_latura,
                     grosime_latura);
        DA.afiseaza(); // afiseaza segmentul DA
        pvizibil=1;
    }
}

void paralelogram::sterge()
// functie virtuala care face invizibil paralelogramul curent
{
    if(valid){ // paralelogramul este valid
        gsegment AB(tvirf[0],tvirf[1],culoare_latura,
                     grosime_latura);
        AB.sterge();
        gsegment BC(tvirf[1],tvirf[2],culoare_latura,
                     grosime_latura);
        BC.sterge();
        gsegment CD(tvirf[2],tvirf[3],culoare_latura,
                     grosime_latura);
        CD.sterge();
        gsegment DA(tvirf[3],tvirf[0],culoare_latura,
                     grosime_latura);
        DA.sterge();
        pvizibil=0;
    }
}

double paralelogram::perimetru() const
/* functie virtuala care returneaza perimetrul paralelogramului curent */
{
    if(valid) {
        double ab = tvirf[0].dist(tvirf[1]);
        double bc = tvirf[1].dist(tvirf[2]);
        return 2*ab+2*bc;
    }
    return 0;
}

double paralelogram::arie() const
/* functie virtuala care returneaza aria paralelogramului curent */
{
    if(valid){
        gsegment CD(tvirf[2],tvirf[3]);
        gpunct A=gpunct(tvirf[0],culoare_latura);

        // se determina inaltimea paralelogramului
        gsegment I = CD|A;
        punct E=I*CD; // intersecția inalțimii din A cu latura CD
        double cd=tvirf[2].dist(tvirf[3]); // lungimea bazei
        double ae=E.dist(A); // lungimea inalțimii
        return cd*ae;
    }
    return 0;
}

void paralelogram::deplaseaza(double dx,double dy)
/* deplasaza paralelogramul modificind abscisa punctelor cu dx si ordonata cu dy */
{
    if(valid) {
        sterge(); // modifica virfurile figurii
        for(int i=0;i < 4;i++) tvirf[i] = tvirf[i]+punct(dx,dy);
        afiseaza();
    }
}

```

## FIŞIERUL BXXX1B

```

#ifndef __PARALELOGRAM_H
#include "BXXX1A.CPP"
#define __PARALELOGRAM_H
#endif
#ifndef __ROMB_H
#include "BXXX1B.H"
#define __ROMB_H
#endif

// clasa ROMB
int romb::validromb()
/* rvalid=1, daca obiectul curent este un romb si are doua laturi paralele cu axa Ox;
   rvalid=0, altfel.
*/
{
    // se verifica daca figura este paralelogram

    validparalel();

    if(valid) {
        // se verifica daca 2 laturi alaturate sunt egale
        double ab=tvirf[0].dist(tvirf[1]);
        double bc=tvirf[1].dist(tvirf[2]);

        if(testegal(ab,bc)) rvalid = 1; // laturi egale
        else rvalid = 0; // nu este romb
    }
    else rvalid = 0; // nu este paralelogram
    return rvalid;
}

inline romb::romb() // constructor implicit
{
    rvalid = 0;
}

inline romb::romb(double xa,double ya,double xb,double yb,
                  double xc,double yc,double xd,double yd,
                  int c,int g):
    paralelogram(xa,ya,xb,yb,xc,yc,xd,yd,c,g)
// constructor utilizat la initializari
{
    if(valid) // figura este paralelogram
        validromb(); /* rvalid=1 daca figura este romb;
                        altfel rvalid=0 */
    else rvalid=0; /* figura nu este un paralelogram si deci nu este nici romb */
}

inline romb::romb(const punct& a,const punct& b,const punct& c,
                  const punct& d,int cl,int g):
    paralelogram(a,b,c,d,cl,g)
// constructor utilizat la initializari

```

```

{
    if(valid) validromb();
    else rvalid=0;
}

inline romb::romb(const romb& r):paralelogram(r)
// constructor de copiere
{
    rvalid=r.rvalid;
}

romb& romb::operator = (const romb& r)
// supraincarca operatorul =
{
    if(r.rvalid && this != &r) {
        for(int i=0;i < 4;i++) tvirf[i] = r.tvirf[i];
        valid = r.valid;
        culoare_latura = r.culoare_latura;
        grosime_latura = r.grosime_latura;
        pvizibil = r.pvizibil;
        rvalid = r.rvalid;
    }
    else
        if(r.rvalid == 0) rvalid = 0;
    return *this;
}

double romb::perimetru() const
// returneaza perimetrul rombului curent
{
    if(rvalid) {
        double p=4*tvirf[0].dist(tvirf[1]);
        return p;
    }
    return 0; // figura nu este romb
}

double romb::arie() const // returneaza aria rombului curent
{
    if(rvalid){
        double d1=tvirf[0].dist(tvirf[2]);
        double d2=tvirf[1].dist(tvirf[3]);
        return d1*d2/2;
    }
    return 0; // figura nu este romb
}

```

## FIŞIERUL BXXX1C

```

#ifndef __PARALELOGRAM
#include "BXXX1A.CPP"
#define __PARALELOGRAM
#endif
#ifndef __DREPTUNGHI
#include "BXXX1C.H"

```

```

#define __DREPTUNGHI
#endif

// clasa DREPTUNGHI
int dreptunghi::validdrept()
/* dvalid=1, daca obiectul curent este un dreptunghi si are doua laturi opuse paralele cu axa Ox;
   dvalid=0, altfel.*/
{
    // se verifica daca figura este paralelogram cu 2 laturi paralele cu axa Ox
    validparalel();
    dvalid = 0;

    if(valid){
        // se verifica daca 2 laturi alaturate sunt perpendicularare
        gsegment AB(tvirf[0],tvirf[1]);
        gsegment BC(tvirf[1],tvirf[2]);
        if(AB && BC) dvalid = 1; // figura este dreptunghi
    }
    return dvalid;
}

inline dreptunghi::dreptunghi() // constructor implicit
{
    dvalid = 0;
}

inline dreptunghi::dreptunghi(double xa,double ya,double xc,
                               double yc,int c,int g):
    parallelogram(xa,ya,xc,ya,xc,yc,xa,yc,c,g)
// constructor utilizat la initializari
{
    if(valid) validdrept();
    else dvalid=0;
}

dreptunghi::dreptunghi(const punct& a,const punct& c,
                      int cl,int g)
// constructor utilizat la initializari
{
    tvirf[0] = a;
    tvirf[1] = punct(c.retabs(),a.retord());
    tvirf[2] = c;
    tvirf[3] = punct(a.retabs(),c.retord());

    validdrept();
    pvizibil = 0;
    culoare_latura = cl;
    grosime_latura = g;
}

dreptunghi::dreptunghi (const dreptunghi& d)
// constructor de copiere
{
    for(int i=0;i < 4;i++) tvirf[i]=d.tvirf[i];
}

```

```

    valid = d.valid;
    dvalid = d.dvalid;
    pvizibil = d.pvizibil;
    culoare_latura = d.culoare_latura;
    grosime_latura = d.grosime_latura;
}

dreptunghi& dreptunghi::operator = (const dreptunghi& d)
// supraincarca operatorul =
{
    if(d.dvalid && this != &d){
        for(int i=0;i < 4;i++) tvirf[i]=d.tvirf[i];
        valid=d.valid;
        dvalid=d.dvalid;
        pvizibil=d.pvizibil;
        culoare_latura=d.culoare_latura;
        grosime_latura=d.grosime_latura;
    }
    else
        if(d.dvalid == 0) dvalid = 0;
    return *this;
}

void dreptunghi::afiseaza() // afiseaza dreptunghiul curent
{
    if(dvalid){ // dreptunghi valid
        struct linesettingstype temp;
        getlinesettings(&temp);
        int c = getcolor();
        setlinestyle(SOLID_LINE,0,grosime_latura);
        setcolor(culoare_latura);
        rectangle(tvirf[0].retabs(),tvirf[0].retord(),
                  tvirf[2].retabs(),tvirf[2].retord());
        setlinestyle(temp.linestyle,
                     temp.upattern,temp.thickness);
        setcolor(c);
        pvizibil =1;
    }
}

void dreptunghi::sterge()
// functie virtuala care face invizibil dreptunghiul curent
{
    if(dvalid){
        int temp = culoare_latura;
        culoare_latura = getbkcolor();
        afiseaza();
        culoare_latura=temp;
        pvizibil=0;
    }
}

double dreptunghi::perimetru() const
/* functie virtuala care returneaza perimetrul dreptunghiului curent */

```

```

    {
        if(dvalid) {
            double baza = tvirf[1].retabs() - tvirf[0].retabs();
            double inaltime = tvirf[3].retord() - tvirf[0].retord();
            if(baza < 0) baza = -baza;
            if(inaltime < 0) inaltime = -inaltime;
            return 2*(baza+inaltime);
        }
        return 0;
    }

    double dreptunghi::arie() const
    /* functie virtuala care returneaza aria dreptunghiului curent */
    {
        if(dvalid){
            double baza=tvirf[1].retabs() - tvirf[0].retabs();
            double inaltime=tvirf[3].retord() - tvirf[0].retord();
            if(baza < 0) baza = -baza;
            if(inaltime < 0) inaltime = -inaltime;
            return baza*inaltime;
        }
        return 0;
    }

```

## FIŞIERUL BXXX1D

```

#ifndef __DREPTUNGHI_H
#include "BXXX1C.CPP"
#define __DREPTUNGHI_H
#endif
#ifndef __PATRAT
#include "BXXX1D.H"
#define __PATRAT
#endif

int patrat::validpatrat()
/* -pvalid=1 daca obiectul curent este un patrat;
   -pvalid=0 altfel.*/
{
    validdrept();
    pvalid=0;
    if(dvalid) {
        // verifică dacă 2 laturi alăturate sunt egale
        double ab=tvirf[1].retabs() - tvirf[0].retabs();
        double da=tvirf[3].retord() - tvirf[0].retord();
        if(ab < 0) ab=-ab;
        if(da < 0) da = -da;
        if(testegal(ab,da)) pvalid=1; // figura este patrat
    }
    return pvalid;
}

inline patrat::patrat() // constructor implicit
{
    pvalid=0;
}

```

```

}

inline patrat::patrat(double xa,double ya,double xc,double yc,
                      int c,int g):dreptunghi(xa,ya,xc,yc,c,g)
// constructor pentru initializari
{
    if(dvalid) validpatrat();
    else pvalid=0;
}

inline patrat::patrat(const punct& a,const punct&c,
                      int cl,int g):dreptunghi(a,c,cl,g)
// constructor utilizat la initializari
{
    if(dvalid) validpatrat();
    else pvalid=0;
}

inline patrat::patrat(const patrat& p):dreptunghi(p)
// constructor de copiere
{
    pvalid=p.pvalid;
}

patrat& patrat::operator = (const patrat& p) // supraincarca operatorul =
{
    if(p.pvalid && this != &p){
        for(int i=0;i < 4;i++) tvirf[i] = p.tvirf[i];
        valid=p.valid;
        dvalid=p.dvalid;
        pvalid=p.pvalid;
        culoare_latura=p.culoare_latura;
        grosime_latura=p.grosime_latura;
        pvizibil=p.pvizibil;
    }
    else
        if(p.pvalid==0) pvalid=0;
    return *this;
}

double patrat::perimetru() const // returneaza perimetrul patratului curent
{
    if(pvalid){
        double latura=tvirf[1].retabs() - tvirf[0].retabs();
        if(latura < 0) latura = -latura;
        return 4*latura;
    }
    else return 0;
}

double patrat::arie() const // returneaza aria patratului curent
{
    if(pvalid){
        double latura=tvirf[1].retabs() - tvirf[0].retabs();

```

```

        return latura*latura;
    }
    else return 0;
}

```

30.2 Să se scrie un program care afișează în jumătatea de sus a ecranului următoarele patrulatere:

- un paralelogram;
- un romb;
- un dreptunghi;
- un patrat.

După afișare se acționează o tastă ASCII oarecare și figurile respective se afișează în jumătatea de jos a ecranului. Acționind din nou o tastă ASCII, se va afișa, pentru fiecare figură geometrică, perimetrul și aria ei.

## PROGRAMUL BXXX2

```

#include <conio.h>
#include "BXXX1B.CPP"
#include "BXXX1D.CPP"

main()
/* - afisaza urmatoarele patrulatere:
   - un paraleogram;
   - un romb;
   - un dreptunghi;
   - un patrat,
   in jumatarea de sus a ecranului;
   - dupa actionarea unei taste ASCII, patrulaterele respective se afisaza in jumatarea
     de jos a ecranului;
   - actionind din nou o tasta ASCII se va afisa perimetru si aria fiecarui patrulater.
*/
{
    paralelogram par(50,30,100,30,70,100,20,100);
    romb r(130,30,180,30,150,70,100,70,YELLOW);
    dreptunghi d(200,30,300,150,LIGHTRED);
    patrat p(350,30,450,130,LIGHTBLUE);
    int gd=DETECT,gm;
    initgraph(&gd,&gm,"C:\\borlandc\\bgi");
    par.afiseaza();
    r.afiseaza();
    d.afiseaza();
    p.afiseaza();
    getch();
    par.deplaseaza(10,170);
    r.deplaseaza(10,170);
    d.deplaseaza(10,170);
    p.deplaseaza(10,170);
    getch();
    closegraph();
    double ppar = par.perimetru();
    double pr = r.perimetru();

```

```

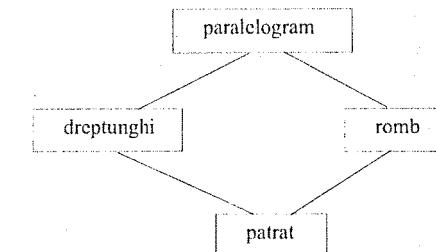
    double pd = d.perimetru();
    double pp = p.perimetru();
    printf("\tperimetru\n");
    printf("paralelogram = %g\n",ppar);
    printf("romb = %g\n",pr);
    printf("dreptunghi = %g\n",pd);
    printf("patrat = %g\n",pp);
    double apar = par.arie();
    double ar = r.arie();
    double ad = d.arie();
    double ap = p.arie();
    printf("\taria\n");
    printf("paralelogram = %g\n",apar);
    printf("romb = %g\n",ar);
    printf("dreptunghi = %g\n",ad);
    printf("patrat = %g\n",ap);
    getch();
}

```

30.3 În exercițiul 30.1, s-a implementat o ierarhie de patrulatere în virful căreia se află clasa *paralelogram*. Clasele *dreptunghi* și *romb* sunt clase derivate din clasa *paralelogram*. Clasa *patrat* este o clasă derivată din clasa *dreptunghi*.

În exercițiul de față se implementează aceeași ierarhie de clase cu singura deosebire că clasa *patrat* este derivată atât din clasa *dreptunghi* cit și din clasa *romb*. Pentru a nu moșteni clasa *paralelogram* de două ori, clasa *paralelogram* se definește ca o clasă de bază virtuală pentru clasele *dreptunghi* și *romb*.

Ierarhia de clase implementată în acest exercițiu este schematizată mai jos:



Pentru a implementa o ierarhie de patrulatere sub forma indicată mai sus, se fac următoarele modificări:

- în fișierele de extensie H, care conțin definițiile pentru clasele *dreptunghi* și *romb*, la derivarea lor, se utilizează clasa de bază virtuală *paralelogram*:

```

class dreptunghi::virtual public
    paralelogram { ... };

```

```

    class romb::virtual public paralelogram { ... };

• se modifică clasa patrat, aceasta devenind o clasă derivată atât din clasa dreptunghi cât și din clasa romb.

```

### FIŞIERUL BXXX3B.H

```

#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

class romb:virtual public paralelogram {
protected:
    int rvalid;
public:
    int validromb();
    romb(); // constructor implicit
    romb(double xa,double ya,double xb,double yb,
          double xc,double yc,double xd,double yd,
          int c=WHITE,int g=NORM_WIDTH);

    romb(const punct& a,const punct& b,const punct& c,
          const punct& d,int cl=WHITE,int g=NORM_WIDTH);

    romb(const romb&);

    romb& operator = (const romb&);

    double perimetru() const;

    double arie() const;
};

```

### FIŞIERUL BXXX3C.H

```

#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

class dreptunghi:virtual public paralelogram {
protected:
    int dvalid;
public:
    int validdrept();
    dreptunghi();
    dreptunghi(double xa,double ya,double xc,double yc,
               int c=WHITE,int g=NORM_WIDTH);

    dreptunghi(const punct& a,const punct& c,
               int cl=WHITE,int g=NORM_WIDTH);

    dreptunghi (const dreptunghi&);

    dreptunghi& operator = (const dreptunghi&);
};

```

```

void afiseaza();

void sterge();

double perimetru() const;

double arie() const;
};


```

### FIŞIERUL BXXX3D.H

```

#ifndef __DREPTUNGHI
#include "BXXX3C.H"
#define __DREPTUNGHI
#endif
#ifndef __ROMB
#include "BXXX3B.H"
#define __ROMB
#endif

class patrat:virtual public dreptunghi,virtual public romb {
protected:
    int pvalid;
public:
    int validpatrat();
    patrat(); // constructor implicit
    patrat(double xa,double ya,double xc,double yc,
           int c=WHITE,int g=NORM_WIDTH);

    patrat(const punct& a,const punct& c,
           int cl=WHITE,int g=NORM_WIDTH);

    patrat(const patrat&);

    patrat& operator = (const patrat&);

    double perimetru() const;

    double arie() const;
};

```

Functiile membru pentru clasele *paralelogram*, *romb* și *dreptunghi* sunt aceleasi ca in cazul exercitiului 30.1, deci ele sunt definite in fisierile:

- BXXX1A.CPP - pentru paralelogram;
- BXXX1B.CPP - pentru romb;
- BXXX1C.CPP - pentru dreptunghi.

Mai jos, se definesc functiile membru noi pentru clasa *patrat*.

### FIŞIERUL BXXX3

```

#ifndef __ROMB
#include "BXXX3B.H" // clasa romb
#define __ROMB
#endif

```

```

#ifndef __DREPTUNGHI
#include "BXXX3C.H"      // clasa dreptunghi
#define __DREPTUNGHI
#endif
#ifndef __PATRAT
#include "BXXX3D.H"      // clasa patrat
#define __PATRAT
#endif
#ifndef __ROMB_H
#define __ROMB_H
#include "BXXX1B.CPP"    // functii membru pentru clasa romb
#endif
#ifndef __DREPTUNGHI_H
#include "BXXX1C.CPP"    // functii membru pentru clasa dreptunghi
#define __DREPTUNGHI_H
#endif

int patrat::validpatrat()
/* - pvalid=1, daca obiectul curent este patrat (este atit dreptunghi cit si romb);
   - pvalid = 0, altfel.
*/
{
    validdrept(); /* stabileste daca obiectul curent este dreptunghi */
    validromb();  /* stabileste daca obiectul curent este romb
    pvalid = rvalid & dvalid;
    return pvalid;
}

inline patrat::patrat() // constructor implicit
{
    pvalid = 0;
}

inline patrat::patrat(double xa,double ya,double xc,
                     double yc,int c,int g):
    dreptunghi(xa,ya,xc,yc,c,g)
// constructor pentru initializari
{
    if(dvalid) validpatrat();
    else pvalid=0;
}

inline patrat::patrat(const punct& a,const punct& c,int cl,
                     int g):dreptunghi(a,c,cl,g)
// constructor utilizat pentru initializari
{
    if(dvalid) validpatrat();
    else pvalid=0;
}

inline patrat::patrat(const patrat& p):dreptunghi(p)
// constructor de copiere
{
    rvalid = p.rvalid;
}

```

```

    pvalid = p.pvalid;
}

patrat& patrat::operator = (const patrat& p)
// supraincarca operatorul de atribuire
{
    if(p.pvalid && this != &p){
        for(int i=0;i < 4;i++) tvirf[i] = p.tvirf[i];
        valid = p.valid;
        rvalid = p.rvalid;
        dvalid = p.dvalid;
        pvalid = p.pvalid;
        pvizibil = p.pvizibil;
        culoare_latura = p.culoare_latura;
        grosime_latura = p.grosime_latura;
    }
    else
        if(p.pvalid == 0) pvalid = 0;
    return *this;
}

inline double patrat::perimetru() const
// returneaza perimetru patratului curent
{
    if(pvalid)
        return romb::perimetru();
    /* se apeleaza functia membru a clasii romb pentru calculul perimetrului
       patratului curent */
    return 0;
}

double patrat::arie() const // returneaza aria patratului curent
{
    if(pvalid){
        double latura=tvirf[1].retabs() -tvirf[0].retabs();
        return latura*latura;
    }
    else return 0;
}

```

30.4 Să se implementeze un container pentru obiecte ale ierarhiei de clase definită în exercițiul 30.3.

În exercițiul de față, containerul se implementează printr-un tablou de pointeri spre obiecte de tip paralelogram.

Dimensiunea maximă a tabloului se ia egală cu 100.

#### FIȘIERUL BXXX4.H

```

#ifndef __PATRAT_H
#include "BXXX3.CPP"
#define __PATRAT_H
#endif

```

```

class container {
    paralelogram *tcontainer[100];
    int n; // indicele elementului liber
public:
    container(); // constructor implicit
    int atrib (paralelogram *p); // punce un obiect in container

    paralelogram *acceselem(int i);
    // permite acces la obiectul spre care pointaza tcontainer[i]
};


```

Funcțiile membru se definesc în fișierul de mai jos, de extensie *CPP*.

#### FIŞIERUL BXXX4

```

#ifndef __CONTAINER_H
#include "BXXX4.H"
#define __CONTAINER_H
#endif

container::container() // constructor implicit
{
    n=0; // container vid
}

int container::atrib (paralelogram *p)
/* - punce obiectul curent in container daca acesta nu este plin;
   - returneaza 1 daca containerul nu este plin;
   - returneaza 0 altfel.
*/
{
    if(n < 100){
        tcontainer[n++] = p;
        return 1;
    }
    else return 0; // container plin
}

paralelogram *container::acceselem(int i)
/* permite accesul la obiectul spre care pointaza tcontainer[i] */
{
    if(0 <= i && i < n) return tcontainer[i];
    else return 0;
}

```

30.5 Să se scrie o funcție care stabilește dacă un obiect de tip *punct* are coordonatele în limitele ecranului.

Funcția returnează 1 dacă coordonatele obiectului parametru aparțin ecranului și zero în caz contrar.

Tipul *punct* este definit în exercițiile 29.1 și 29.7.

#### FUNCTIA BXXX5

```

#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#ifndef __PUNCT_H
#include "BXXIX7.CPP"
#define __PUNCT_H
#endif

int ecran (const punct& p)
/* returneaza 1 daca coordonatele punctului p se afla in limitele
   ecranului si zero in caz contrar */
{
    double x=p.retabs();
    double y=p.retord();

    if(x < 0 || x > getmaxx()) return 0;
    if(y < 0 || y > getmaxy()) return 0;
    return 1;
}

```

30.6 Să se scrie un program care realizează următoarele:

- Citește una din literele P,R,D,T (mică sau mare) care reprezintă tipul unui patrulater:

P - paralelogram;  
R - romb;  
D - dreptunghi;  
T - pătrat.

- Citește coordonatele virfurilor patrulaterului al cărui tip este precizat de litera citită la punctul *a*.
- Păstrează, într-un container, patrulaterul ale cărui coordonate ale virfurilor au fost citite la punctul *b*, dacă acesta are următoarele proprietăți:  
  - este unul din tipurile de patrulater amintite la punctul *a*;
  - patrulaterul are 2 laturi paralele cu Ox;
  - virfurile sunt situate în limitele ecranului grafic;
- Punctele a-c se repetă pînă la întîlnirea sfîrșitului de fișier.
- După întîlnirea sfîrșitului de fișier, pentru fiecare patrulater din container, se afișează pe rînd:  
  - figura pe care o reprezintă patrulaterul;
  - perimetru patrulaterului;
  - aria figurii.

După afișarea datelor unei figuri, se va acționa o tastă ASCII pentru a afișa datele figurii următoare din container.

Culoarea utilizată la afișarea patrulaterelor este definită de constanta simbolică YELLOW.

## PROGRAMUL BXXX6

```
#include <conio.h>
#include "BXXX4.CPP" /* clasa container, precum si clasele ierarhici de patrulatere */
#include "BXXX5.CPP" // functia ecran

main()
/* - citeste una din literele P,R,D,T (mica sau mare) care reprezinta tipul unui patrulater:
   P - paralelogram;
   R - romb;
   D - dreptunghi;
   T - patrat;
- citeste coordonatele virfurilor patrulaterului al carui tip a fost definit de litera citita;
- pastreaza patrulaterul intr-un container daca:
  - este unul din tipurile de patrulater amintite mai sus;
  - are 2 laturi paralele cu Ox;
  - virfurile sunt situate in limitele ecranului grafic;
- se repeta pasii de mai sus pina la intilnirea sfirsitului de fisier;
- la intilnirea sfirsitului de fisier, pentru fiecare patrulater din container, se afiseaza pe rand:
  - patrulaterul respectiv;
  - perimetru si aria patrulaterului
*/
{
    char text1[]="virfurile tastate nu formeaza un\n";
    char text2[]="cu 2 laturi paralele cu Ox\n";
    char car[2],cr;
    int i;
    int gd=DETECT,gm;
    initgraph(&gd,&gm,"C:\\borlandc\\bgi");
    container patrulater;
    punct a,b,c,d; // virfurile unui patrulater
    paralelogram *p;
    romb *r;
    dreptunghi *dr;
    patrat *t;

    for(); { //citeste litera
        for(); {
            printf("una din literele P, R, D,T sau <Ctrl>-Z\n");
            if((i=scanf("%ls",car)) == EOF)
                break; // s-a intilnit sfirsitul de fisier
            cr=car[0];
            if(cr >= 'a') // litera mica
                cr = cr-'a'+'A';
            if(cr=='P' || cr=='R' || cr=='D' || cr=='T') break;
            printf("se cere una din literele:\n");
            printf("\tP\n\tR\n\tD\n\tT\n");
            fflush(stdin);
        } // sfirsit ciclului pentru citirea literei
        if(i == EOF) break; // s-a intilnit sfirsitul de fisier
        // se citesc virfurile patrulaterului si obiectul corespunzator se pune in container
        // daca satisfac conditiile cerute
    }
}
```

```
fflush(stdin);
i = EOF;
switch(cr){
    case 'P': // paralelogram
    case 'R': // romb
        if(a.citpct("a") == 0) break; // EOF
        if(b.citpct("b") == 0) break; // EOF
        if(c.citpct("c") == 0) break; // EOF
        if(d.citpct("d") == 0) break; // EOF
        i=0; // nu s-a intilnit sfirsitul de fisier
        if(ecran(a)==0 || ecran(b)==0 ||
           ecran(c)==0 || ecran(d)==0) {
            // cel putin un virf nu este in limitele ecranului grafic
            printf("cel putin un virf \
                   nu este pe ecran\n");
            break; // iesire din switch
        }
        if(cr=='P'){ // paralelogram
            p=new paralelogram(a,b,c,d,YELLOW);
            if(p->validparalel()==0){
                printf(text1);
                printf(" paralelogram ");
                printf(text2);
                break; // iesire din switch
            }
            // se pune paralelogramul in container daca exista loc
            if(patrulater.atrib(p) == 0)
                i = EOF; // container plin
            break; // iesire din switch
        }
        else{ // romb
            r=new romb(a,b,c,d,YELLOW);
            if(r->validromb()==0){
                printf(text1);
                printf(" romb ");
                printf(text2);
                break; // iesire din switch
            }
            // se pune rombul in container daca este loc
            if(patrulater.atrib(r) == 0)
                i=EOF; // container plin
            break; // iesire din switch
        }
    } // sfirsit else
    default: // dreptunghi sau patrat
        if(a.citpct("a") == 0) break; // EOF
        if(c.citpct("c") == 0) break; // EOF
        i=0; // nu s-a intilnit sfirsitul de fisier
        if(ecran(a) == 0 || ecran(c) == 0) {
            // cel putin un virf nu este in limitele ecranului grafic
            printf("cel putin un virf nu \
                   este pe ecran\n");
            break; // iesire din switch
        }
}
```

```

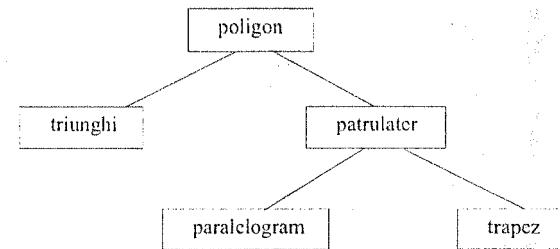
if(cr=='D') { // dreptunghi
    dr=new dreptunghi(a,c,YELLOW);
    if(dr->validdrept()==0){
        printf(text1);
        printf(" dreptunghi ");
        printf(text2);
        break; //iesire din switch
    }
    // se pune dreptunghiul in container daca este loc
    if(patrulater.attrib(dr)==0)
        i=EOF; //container plin
    break; //iesire din switch
}
else{ // patrat
    t=new patrat(a,c,YELLOW);
    if(t->validpatrat()==0){
        printf(text1);
        printf(" patrat ");
        printf(text2);
        break; //iesire switch
    }
    // se pune patratul in container daca este loc
    if(patrulater.attrib(t)==0)
        i=EOF; //container plin
    break; //iesire switch
}
// sfarsit else
} //sfarsit switch
if(i==EOF)
    break; //s-a intilnит sfarsitul de fisier sau containerul este plin
} // sfarsit for pentru citire date

// se afisaza datele pentru patrulaterele din container
cleardevice();
for(i=0;;i++) {
    char temp[80];
    p=patrulater.acceselem(i);
    if(p==0)
        break; /* indicele i depaseste numarul patrulaterelor din container */
    p->afiseaza();
    double per=p->perimetru();
    sprintf(temp,"perimetru=%g",per);
    outtextxy(0,getmaxy()-20,temp);
    double ar=p->arie();
    sprintf(temp,"aria=%g",ar);
    outtextxy(0,getmaxy()-10,temp);
    getch();
    cleardevice(); // sterg tot ecranul
}
closegraph();
}

```

## 30.1. Clase abstracte

Clasele au fost introduse in limbajul C++ pentru a implementa tipuri noi de date, diferite de cele predefinite. Aceste tipuri corespund unor concepte care intervin in mod natural in programarea diferitelor aplicatiilor concrete. Aceste concepte au un caracter mai mult sau mai putin general. O serie de concepte care intervin in rezolvarea unei probleme, nu sunt independente, prezintand elemente comune. Se ajunge in felul acesta la un proces de ierarhizare a conceptelor, in virful ierarhiei aflindu-se conceptul cel mai general, ale carui proprietati sunt moștenite de celelalte concepte ale ierarhiei, mai putin generale sau cum se obisnuiește să se spună, mai specializate. O astfel de ierarhie, intre conceptele utilizate in rezolvarea unei probleme, conduce la o ierarhie a claselor care implementează conceptele respective. Ierarhia dintre clase se definește cu ajutorul noțiunii de derivare. Dacă o clasă A este pe un nivel imediat inferior față de clasa B in ierarhia de clase, atunci se spune că clasa A este o clasă derivată a clasei B, iar despre B se spune că este o clasă de bază a lui A. Esența derivării constă in aceea că, elementele membru (date și funcții) ale clasei de bază sunt moștenite de clasa derivată. Clasele care corespund nivelurilor mai inalte au un grad inalt de generalitate și de aceea anumite funcții membru ale acestora nu pot fi definite. De exemplu, să considerăm o ierarhie pentru figurile geometrice:



Un poligon are elemente comune pentru toate celelalte concepte ale ierarhiei și anume:

- perimetru;
- aria;
- poate fi afișat;
- poate fi facut invizibil după o afișare.

Aceste noțiuni relativ la *poligon* sint prea generale și ele sau nu pot fi definite cu ajutorul unor funcții sau acestea chiar dacă se pot defini sunt ineficiente.

Ele se pot exprima in mod eficient numai la niveluri mai inferioare ale ierarhiei.

De exemplu, pentru conceptul *triunghi* se pot defini funcții pentru elementele indicate mai sus. În mod analog, se pot defini funcții pentru aceleasi elemente și in cazul conceptului *patrulater*. Uncle pot fi definite mai eficient,

dacă se coboară mai jos în ierarhie. Dacă un patrulater este un paralelogram sau un trapez, atunci calculul ariei se poate realiza mai eficient dacă se face separat pentru fiecare din aceste cazuri particulare.

Faptul că noțiunile de arie și perimetru relativ la poligon sunt prea generale și nu le definim decât pentru concepții mai specializate cum sunt triunghiul, paralelogramul, trapezul etc. ne conduce la noțiunea de *funcție virtuală pură*. O *funcție virtuală pură* este o funcție virtuală care nu are definiție ci numai prototip.

Funcțiile membru ale clasei *poligon* pentru calculul perimetrului sau ariei, precum și cele pentru afișarea sau ștergerea unui poligon, sunt funcții virtuale pure. Ele nu sunt definite pentru clasa *poligon*, fiind prea generale și au numai prototipuri.

Funcțiile virtuale pure se definesc ca funcții virtuale la niveluri mai inferioare. În cazul de față, se pot defini funcții virtuale corespunzătoare pentru clasele *triunghi* și *patrulater* sau chiar pentru niveluri mai particulare cum este *paralelogramul* sau *trapezul*.

Prototipul unei funcții virtuale pure are unul din următoarele formate:

**virtual tip nume\_funcție (lista parametrilor formali)=0;**

sau

**virtual tip nume\_funcție (lista parametrilor formali)const=0;**

Cu alte cuvinte, o funcție virtuală pură are un prototip care începe cu cuvântul *virtual* și se termină cu specificația:

=0

urmărată de punctul și virgula care termină orice prototip.

Prototipul unei funcții virtuale pure se scrie în definiția clasei pentru care este funcție membru.

Presupunând că clasa *poligon* are funcțiile virtuale pure amintite mai sus, vom defini această clasă astfel:

```
class poligon {
public:
    virtual double perimetru() const = 0;
    virtual double arie() const = 0;
    virtual void afiseaza() = 0;
    virtual void sterge() = 0;
};
```

O clasă care are cel puțin o funcție membru care este funcție virtuală pură, se numește *clasă abstractă*.

Clasa *poligon*, definită ca mai sus, este un exemplu de clasă abstractă. Clasele abstracte nu au obiecte. Ele nu se pot utiliza la instanțieri ci numai ca și clase de bază în procesul de derivare.

O funcție virtuală pură a unei clase trebuie să fie suprapusă în orice clasă care derivă direct din clasa respectivă. Ea poate fi suprapusă ca o funcție virtuală

obișnuită sau tot ca o funcție virtuală pură. În acest ultim caz, clasa derivată este și ea o clasă abstractă.

Din cele de mai sus rezultă imediat rolul jucat de clasele abstracte și de funcțiile virtuale pure.

O funcție virtuală pură indică o funcție a unei clase care trebuie să fie prezentă la orice clasă derivată din ea. Funcția virtuală pură nu se definește deoarece clasa la care aparține este prea generală pentru a putea fi definită sau se definește neeficient.

Clasele abstracte nu au instanțieri, deoarece ele au cel puțin o funcție nedefinită. Acestea prezintă interes în procesul de derivare impunind suprapunerea funcțiilor virtuale pure în clasele care derivă din ele.

Clasa *triunghi*, din ierarhia de mai sus, se definește astfel:

```
class triunghi:public poligon {
    punct tri[3];
public:
    triunghi();
    triunghi(const punct&,const punct&,const punct&);
    triunghi(const triunghi&);
    double perimetru() const;
    double arie() const;
    void afiseaza();
    void sterge();
    triunghi& operator = (const triunghi&);
};
```

Funcțiile membru *perimetru*, *arie*, *afiseaza* și *sterge* ale clasei *triunghi* sunt funcții virtuale obișnuite care se suprapun peste funcțiile virtuale pure corespunzătoare clasei *poligon*. Aceste funcții trebuie să fie definite în mod obișnuit.

Clasa *triunghi* nu mai este o clasă abstractă, deci ea poate fi instanțiată:

```
triunghi abc;
punct p(1,2),q(3,2),r(2,3);
triunghi pqr(p,q,r);
```

sunt instanțieri corecte ale clasei *triunghi*.

O instanțiere de forma:

```
poligon pol;
```

este eronată, clasa *poligon* fiind o clasă abstractă.

Clasa *patrulater* poate fi definită tot ca o clasă abstractă, ca mai jos:

```
class patrulater:public poligon {
protected:
    punct patru[4];
public:
    patrulater();
    patrulater(const punct& a,const punct& b,
               const punct& c,const punct& d);
    patrulater(const patrulater&);
```

```

    double perimetru() const = 0;
    double arie() const = 0;
    void afiseaza();
    void sterge();
    patrulater& operator = (const patrulater&);
};

Clasa patrulater are două funcții virtuale pure: perimetru și arie. Acestea urmează să fi suprapuse prin funcții virtuale obișnuite (care nu sunt pure) atât în clasa paralelogram cât și în clasa trapez.

```

Funcțiile *afiseaza* și *sterge*, ale clasei *patrulater*, sunt funcții virtuale obișnuite care se definesc folosind funcția standard *drawpoly* care afișează un poligon (vezi paragraful 19.7.).

Un alt exemplu de utilizare a claselor abstracte este cel al ierarhiei care implementează conceptul de *grupoid*.

O mulțime oarecare de elemente formează un *grupoid* dacă pe mulțimea respectivă este definită o lege de compozitie internă. De aici, rezultă că pentru a defini un grupoid este necesar să precizăm mulțimea și legea de compozitie internă.

Spre exemplificare, vom considera mulțimea numerelor intregi, iar ca lege internă alegem operația de scădere. Contraș practiciei incetătenite în matematică, alegind caracterul "+" pentru a nota legea de compozitie internă,  $a+b$  va însemna nu suma dintre  $a$  și  $b$ , ci chiar diferența lor.

Primul pas în definirea grupoidului va fi implementarea unei submulțimi a mulțimii numerelor intregi. Aceasta se face printr-o clasă pe care o denumim *elem*. Fixându-ne asupra mulțimii numerelor intregi de tip *long*, înseamnă că clasa *elem* va avea o dată membru de tip *long*, pe care o numim *gelem*.

Ca funcții membru vom avea și o funcție care permite acces la data membru *gelem*.

Definim clasa *elem* ca mai jos:

```

class elem {
protected:
    long gelem;
public:
    elem() // constructor implicit
    {
    }

    elem(long g) // constructor utilizat la initializari si conversii
    {
        gelem = g;
    }

    elem(const elem& e) // constructor de copiere
    {
        gelem = e.gelem;
    }
};

```

```

long retgelem()
{
    return gelem;
}
};

Pasul următor, în definirea grupoidului, este de a defini o lege de compozitie internă pe mulțimea obiectelor clasei elem, lege pe care convenim să o notăm cu "+".

```

În matematică afirmăm despre o mulțime că este grupoid dacă știm că pe mulțimea respectivă este definită o lege de compozitie internă, făcind abstracție de modul în care este definită legea respectivă. Același lucru îl putem exprima în cazul de față definind clasa *grupoid* ca o clasă abstractă, care derivă din clasa *elem*. Această clasă, în afară de constructori, are o funcție virtuală pură care supraincarcă operatorul "+" pentru obiecte de tip *elem*. Aceasta înseamnă că orice clasă derivată din clasa *grupoid* trebuie să supraincarce operatorul "+". La acest nivel se pot defini clase care să nu mai fie abstracte, având definit modul concret de supraincarcare al operatorului "+".

Clasa *grupoid* poate fi definită astfel:

```

class grupoid:public elem { // clasa abstracta
public:
    grupoid() // constructor implicit
    {
        gelem = 0;
    }

    grupoid(long g):elem(g) // constructor utilizat la conversii
    {
    }

    grupoid(elem e):elem(e) // constructor utilizat la conversii
    {
    }

    virtual elem operator + (elem) = 0;
    /* - funcție virtuală pură;
     - supraincarca operatorul +pentru obiecte de tip elem;
     orice clasa derivată din clasa grupoid trebuie sa supraincarce operatorul +. */
};


```

În momentul de față putem implementa grupoidul anunțat mai sus, care este format din mulțimea numerelor de tip întreg, legea de compozitie fiind operația de scădere. Notăm cu *Zminus* clasa derivată din clasa *grupoid* și ale cărei obiecte formează grupoidul dorit.

Clasa *Zminus* are o singură dată membru și anume pe *gelem* care este moștenită de la clasa *elem* și este de tip *long*.

Funcțiile membru ale clasei *Zminus* sunt constructorii, funcția care supraincarcă operatorul "+" și eventual alte funcții care intervin în utilizarea obiectelor, cum ar fi, de exemplu, funcția de afișare a datei membru *gelem* moștenită de la

clasa *elem*.

Definim clasa *Zminus* astfel:

```
class Zminus:public grupoid {
public:
    Zminus() // constructor implicit
    {
    }

    Zminus(long zm):grupoid(zm)// constructor utilizat la initializare si conversie
    {

    }

    Zminus(elem e):grupoid(e)
    // constructor utilizat la initializare si conversie
    {

    }

    elem operator + (elem zm)
    // suprainscarca operatorul +
    {
        elem v(gelem-zm.retgelem());
        return v;
    }

    void afis()
    // afiseaza data membru gelem a obiectului curent
    {
        printf("%ld\n",gelem);
    }
}
```

Fie instanțierile:

```
Zminus z1(123456789);
Zminus z2(123000000);
```

O expresie de forma:

```
z1+z2
```

este corectă și ea are ca valoare un obiect de tip *elem* pentru care *gelem* are valoarea:

$$z1.gelem - z2.gelem = 123456789 - 123000000 = 456789$$

Pentru a verifica acest lucru, putem folosi sevența de mai jos:

```
Zminus x = z1+z2;
x.afis();
```

Pentru a aplica operatorul "+", suprainscarcat ca mai sus, la obiecte ale clasei *elem*, putem aplica explicit constructorul clasei *Zminus* care convertește un obiect de tip *elem* într-un obiect de tip *Zminus*.

Fie, de exemplu, instanțierile:

```
elem a(123456789);
elem b(123000000);
```

O expresie de forma:

```
a+b
```

este eronată, deoarece obiectul din stînga operatorului trebuie să fie o instanțiere a clasei *Zminus*. De aceea, expresia de mai sus, se scrie apelind explicit conversia din tipul *elem* în tipul *Zminus*:

```
Zminus(a)+b
```

Mulțimea obiectelor clasei *elem* reprezintă o submulțime a numerelor intregi și anume ea corespunde numerelor intregi din intervalul  $[-2^{31}, 2^{31}-1]$ .

Obiectele acestei clase pot fi utilizate în expresii de forma:

(1) *Zminus(a)+b*

unde:

*a* și *b* Sunt instanțieri ale clasei *elem*.

Expresia (1) are ca rezultat un obiect al clasei *elem*, a cărei dată membru are ca valoare diferența dintre datele membru ale obiectelor *a* și *b*. În felul acesta, operatorul "+" din expresia (1), corespunde operației de scădere definită pe mulțimea numerelor intregi. De aceea, se poate considera că mulțimea obiectelor clasei *elem* formează un grupoid față de operația "+" definită prin funcția membru *operator +* a clasei *Zminus*.

Acest *grupoid* corespunde *grupoidului* format de mulțimea numerelor întregi față de operația de scădere definită pe mulțimea respectivă.

**Observație:**

Grupoidul format de obiectele clasei *elem* nu corespunde în mod riguros cu noțiunea de *grupoid* definită în matematică pe mulțimea numerelor intregi față de operația de scădere, deoarece expresia (1) are o valoare eronată cind rezultatul aplicării operatorului "+" ne conduce la o valoare care este în afara limitelor tipului *long*.

Schimbând definiția suprainscărcării operatorului "+", se pot implementa alte grupoide pe mulțimea obiectelor clasei *elem*.

Fie de exemplu, clasa *Zplus*, care diferă de clasa *Zminus* numai prin funcția *operator +*:

```
class Zplus:public grupoid {
public:
    ...
    elem operator + (elem zp) // suprainscarca operatorul +
    {
        elem v(gelem + zp.retgelem());
        return v;
    }
    ...
}
```

};

În acest caz, obiectele clasei *elem* formează un grupoid care corespunde grupoidului definit pe mulțimea numerelor intregi față de adunare. Acest grupoid formează chiar un grup, spre deosebire de cel precedent, care nu formează nici măcar un semigrup, deoarece operația de scădere nu este asociativă.

Obiectele clasei *Zminus* pot și ele forma un grupoid față de operația "+", dacă se adaugă, la clasa respectivă, funcția membru de mai jos:

```
Zminus Zminus::operator + (Zminus zm)
// suprareîncarcă operatorul + pentru obiectele clasei Zminus
{
    Zminus v(*this+elem(zm));
    return v;
}
```

Fie instanțierile:

```
Zminus a(123456789);
Zminus b(123000000);
```

Expresia:

a+b

este corectă și ea se evaluează apelindu-se funcția *operator+* de mai sus, suprareîncărată pentru obiectele clasei *Zminus*.

În felul acesta, se poate afirma că obiectele clasei *Zminus* formează și ele un *grupoid* care corespunde cu grupoidul format de elementele clasei *elem*.

Declarația:

```
Zminus v(*this+elem(zm));
```

din corpul funcției care suprareîncarcă operatorul + conține termenul:

elem(zm)

Acst termen apelează constructorul de copiere al clasei *elem* deoarece *zm* este considerat un obiect "particular" al clasei *elem*, fiind o instanțiere a clasei *Zminus* care derivă indirect din clasa *elem*.

La evaluarea expresiei:

\*this+elem(zm)

se apelează funcția care suprareîncarcă operatorul +, de prototip:

```
elem operator + (elem);
```

Să observăm că, expresia:

\*this+zm

este corectă din punct de vedere sintactic, dar utilizarea ei conduce la un apel recursiv infinit al funcției de prototip:

```
Zminus operator + (Zminus);
```

În încheierea acestui paragraf amintim restricțiile legate de utilizarea claselor abstracte.

Așa cum s-a amintit mai sus, o clasă abstractă poate fi utilizată numai ca o clasă de bază. Nu se pot face instantieri pentru o clasă abstractă.

O clasă abstractă nu poate fi tipul unui parametru. O funcție nu poate returna un obiect al unei clase abstracte.

Un parametru poate fi o referință la o clasă abstractă. O funcție poate returna o referință la o clasă abstractă. Referințele la o clasă abstractă sunt admise numai dacă la utilizarea lor nu se creează obiecte intermediare.

În sfîrșit, să amintim că se pot declara pointeri de tipuri implementate prin clase abstracte.

### Exerciții:

30.7 Să se implementeze conceptul *poligon* printr-o clasă abstractă.

### FIȘIERUL BXXX7.H

```
class poligon {
public:
    virtual int polvalid() = 0;
    virtual double perimetru() const = 0;
    virtual double arie() const = 0;
    virtual void afiseaza() = 0;
    virtual void sterge() = 0;
};
```

30.8 Să se implementeze conceptul de *triunghi* printr-o clasă derivată din clasa abstractă *poligon* definită în exercițiul 30.7.

### FIȘIERUL BXXX8.H

```
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#ifndef __GSEGMENT_H
#include "BXXX11.CPP" // clasa gsegment
#define __GSEGMENT_H
#endif
#ifndef __POLIGON_H
#include "BXXX7.H" // clasa poligon
#define __POLIGON_H
#endif

class triunghi:public poligon {
protected:
    punct tri[3];
    int tvizibil,tvalid;
public:
    int polvalid();
    /* - returneaza 1 daca tri[0],tri[1],tri[2] nu sunt coliniare;
```

```

    - returneaza 0 altfel.
}

triunghi(); // constructor implicit

triunghi(double xa,double ya,double xb,
         double yb,double xc,double yc);
// constructor utilizat la initializari

triunghi(const punct& a,const punct& b,const punct& c);
// constructor utilizat la initializari

triunghi(const triunghi&); // constructor de copiere

double perimetru() const; // returneaza perimetrul triunghiului curent

double arie() const; // returneaza aria triunghiului curent

void afiseaza(); // afiseaza triunghiul curent

void sterge(); // face invizibil triunghiul curent
};

Funcțiile membru se definesc în fișierul de mai jos, de extensie CPP.

```

## FIŞIERUL BXXX8

```

#ifndef __TRIUNGHI_H
#include "BXXX8.H"
#define __TRIUNGHI_H
#endif

#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif

int triunghi::polvalid()
/* - returneaza 1 daca virfurile obiectului curent formeaza un triunghi (nu sunt coliniare);
   - returneaza 0 altfel.
*/
{
    gsegment ab(tri[0],tri[1]);

    if((ab & tri[2]) == 0) {
        tvalid = 1;
        return 1; // virfurile formeaza un triunghi
    }
    tvalid = 0;
    return 0; // virfurile sunt coliniare
}

inline triunghi::triunghi() // constructor implicit
{

```

```

    tvalid = 0;
    tvizibil = 0;
}

triunghi::triunghi(double xa,double ya,double xb,double yb,
                   double xc,double yc)
// constructor utilizat la initializari
{
    tri[0] = punct(xa,ya);
    tri[1] = punct(xb,yb);
    tri[2] = punct(xc,yc);
    polvalid();
    tvizibil = 0;
}

triunghi::triunghi(const punct& a,const punct& b,const punct& c)
// constructor utilizat la initializari
{
    tri[0] = a;
    tri[1] = b;
    tri[2] = c;
    polvalid();
    tvizibil = 0;
}

triunghi::triunghi(const triunghi& t) // constructor de copiere
{
    for(int i=0;i < 3;i++) tri[i]=t.tri[i];
    polvalid();
    tvizibil=t.tvizibil;
}

double triunghi::perimetru() const
// returneaza perimetrul triunghiului curent
{
    if(tvalid){
        double p=tri[0].dist(tri[1]);
        p = p+tri[1].dist(tri[2]);
        p = p+tri[2].dist(tri[0]);
        return p;
    }
    return 0; // virfurile nu formeaza un triunghi
}

double triunghi::arie() const // returneaza aria triunghiului curent
{
    if(tvalid) { // se aplica formula lui Heron
        double a = tri[1].dist(tri[2]);
        double b = tri[0].dist(tri[2]);
        double c = tri[0].dist(tri[1]);
        double p = (a+b+c)/2; // semiperimetru
        return sqrt(p*(p-a)*(p-b)*(p-c));
    }
    return 0; // virfurile nu formeaza un triunghi
}

```

```

}

void triunghi::afiseaza() // afiseaza triunghiul curent
{
    int tri1[8]; // tabel cu coordonate
    int i,j;

    if(tvalid){
        for(i=0,j=0; i < 6;i += 2,j++){
            tri1[i] = tri[j].retabs();
            tri1[i+1] = tri[j].retord();
        }
        tri1[6] = tri[0].retabs();
        tri1[7] = tri[0].retord();
        drawpoly(4,tri1);
        tvizibil=1;
    }
}

void triunghi::sterge() // face invizibil triunghiul curent
{
    int c = getcolor();

    if(tvalid){
        setcolor(getbkcolor());
        afiseaza();
        setcolor(c);
        tvizibil = 0;
    }
}

```

- 30.9 Să se implementeze conceptul de *patrulater* printr-o clasă derivată din clasa abstractă *poligon* care are două funcții virtuale pure: una pentru calculul perimetrului și cealaltă pentru calculul ariei. Funcțiile membru *afiseaza* și *sterge* se definesc la fel ca în cazul clasei *triunghi*, definită în exercițiul 30.8.

### FIŞIERUL BXXX9.H

```

#ifndef __GRAPHICS_H
#include <graphics>
#define __GRAPHICS_H
#endif
#ifndef __GSEGMENT_H
#include "BXXIX11.CPP" // clasa gsegment
#define __GSEGMENT_H
#endif
#ifndef __POLIGON_H
#include "BXXX7.H" // clasa poligon
#define __POLIGON_H
#endif

```

```

class patrulater:public poligon {
protected:
    punct patru[4];
    int valid;
    int pvizibil;
public:
    int polavalid()=0;
    patrulater(); // constructor implicit
    patrulater(double xa,double ya,double xb,double yb,
               double xc,double yc,double xd,double yd);
    // constructor utilizat la initializari

    patrulater(const punct& a,const punct& b,
               const punct& c,const punct& d);
    // constructor utilizat la initializari

    patrulater(const patrulater&); // constructor de copiere

    virtual double perimetru() const = 0; // functie virtuala pura

    virtual double arie() const = 0; // functie virtuala pura

    void afiseaza();

    void sterge();
};

Funcțiile membru se definesc în fișierul de mai jos, de extensie CPP.

```

### FIŞIERUL BXXX9

```

#ifndef __PATRULATER
#include "BXXX9.H" // clasa patrulater
#define __PATRULATER
#endif

inline patrulater::patrulater() // constructor implicit
{
    pvizibil = 0;
    valid = 0;
}

patrulater::patrulater(double xa,double ya,double xb,double yb,
                      double xc,double yc,double xd,double yd)
// constructor utilizat la initializari
{
    patru[0] = punct(xa,ya);
    patru[1] = punct(xb,yb);
    patru[2] = punct(xc,yc);
    patru[3] = punct(xd,yd);
    valid = 1;
    pvizibil = 0;
}

```

```

patrulater::patrulater(const punct& a,const punct& b,
                      const punct& c,const punct& d)
// constructor utilizat la initializari
{
    patru[0] = a;
    patru[1] = b;
    patru[2] = c;
    patru[3] = d;
    valid = 1;
    pvizibil = 0;
}

patrulater::patrulater(const patrulater& p)
// constructor de copiere
{
    for(int i=0;i < 4;i++) patru[i]=p.patrui[i];
    valid = p.valid;
    pvizibil = p.pvizibil;
}

void patrulater::afiseaza() // afiseaza patrulaterul curent
{
    int patrul[10];
    int i,j;

    if(valid){
        for(i=0,j=0; i < 8;i += 2,j++){
            patrul[i] = patru[j].retabs();
            patrul[i+1] = patru[j].retord();
        }
        patrul[8] = patru[0].retabs();
        patrul[9] = patru[0].retord();
        drawpoly(5,patrul);
        pvizibil=1;
    }
}

void patrulater::sterge() // face invizibil patrulaterul curent
{
    int c = getcolor();

    if(valid){
        setcolor(getbkcolor());
        afiseaza();
        setcolor(c);
        pvizibil = 0;
    }
}

```

30.10 Să se implementeze conceptul de *paralelogram* printr-o clasă derivată din clasa *patrulater* definită în exercițiul 30.9.

Obiectele clasei *paralelogram* pot fi afisate sau șterse folosind funcțiile membru *afiseaza* și respectiv *sterge* ale clasei *patrulater*.

## FIŞIERUL BXXX10.H

```

#ifndef __PATRULATER_H
#include "BXXX9.CPP" // clasa patrulater
#define __PATRULATER_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif

class paralelogram:public patrulater {
protected:
    int pvalid;
public:
    int polvalid();
    /* - returneaza 1 daca obiectul curent este un paralelogram;
       - returneaza 0 altfel.
    */

    paralelogram(); // constructor implicit

    paralelogram(double xa,double ya,double xb,double yb,
                 double xc,double yc,double xd,double yd);
// constructor utilizat la initializari

    paralelogram(const punct& a,const punct& b,
                 const punct& c,const punct& d);
// constructor utilizat la initializari

    paralelogram(const paralelogram&); // constructor de copiere

    double perimetru() const;
// returneaza perimetrul paralelogramului curent

    double arie() const;
// returneaza aria paralelogramului curent
};

Mai jos, se definesc funcțiile membru într-un fișier de extensie CPP.

```

## FIŞIERUL BXXX10

```

#ifndef __PARALELOGRAM
#include "BXXX10.H" // clasa paralelogram
#define __PARALELOGRAM
#endif

int paralelogram::polvalid()
/* - returneaza 1 daca obiectul curent este un paralelogram;
   - returneaza 0 altfel.
*/
{
    gsegment ab(patrui[0],patru[1]);
    gsegment bc(patrui[1],patru[2]);

```

```

gsegment cd(patr[2],patr[3]);
gsegment da(patr[3],patr[0]);

if((ab == bc) || (ab == cd) || (ab == da) ||
(bc == cd) || (bc == da) || (cd == da)) {
    pvalid = 0;
    return 0;
}
if(ab || cd)
    if(bc||da){
        pvalid = 1;
        return 1;
    }
pvalid = 0;
return 0;
}

inline paralelogram::paralelogram() // constructor implicit
{
    pvalid = 0;
}

inline paralelogram::paralelogram(double xa,double ya,
    double xb,double yb,double xc,
    double yc,double xd,double yd):
    patrulater(xa,ya,xb,yb,xc,yc,xd,yd)
// constructor utilizat la initializari
{
    polvalid();
}

inline paralelogram::paralelogram(const punct& a,const punct& b,
    const punct& c,const punct& d):
    patrulater(a,b,c,d)
// constructor utilizat la initializari
{
    polvalid();
}

paralelogram::paralelogram(const paralelogram& p)
// constructor de copiere
{
    for(int i=0;i < 4;i++) patr[i]=p.patr[i];
    polvalid();
    pvizibil = p.pvizibil;
}

double paralelogram::perimetru() const
// returneaza perimetru paralelogramului curent
{
    if(pvalid){
        double ab = patr[0].dist(patr[1]);
        double bc = patr[1].dist(patr[2]);
        return 2*(ab+bc);
    }
}

```

```

    }
    return 0;
}

double paralelogram::arie() const // returneaza aria paralelogramului curent
{
    if(pvalid) {
        gsegment CD(patr[2],patr[3]);
        gpunct A(patr[0]);

        // se determina inaltimea paralelogramului din punctul A
        gsegment I = CD | A;
        punct E=I*CD; // intersectia inalitimii din A cu latura CD
        double cd=patr[2].dist(patr[3]); // lungimea laturii CD
        double ae = E.dist(A); // lungimea inalitimii AE
        return cd*ae;
    }
    return 0;
}

```

- 30.11 Să se scrie un program care afișează un triunghi și un paralelogram pe ecranul grafic împreună cu perimetrul și aria figurilor respective. Afisările se fac secvențial, întii triunghiul și apoi paralelogramul, după acționarea unei taste ASCII oarecare.

### PROGRAMUL BXXX11

```

#include <conio.h>
#include "BXXX8.CPP" // clasa triunghi
#include "BXXX10.CPP" // clasa paralelogram

main()
/* - afiseaza un triunghi si un paralelogram impreuna cu perimetrele si arile lor;
   - afisarea se face secvential, intii triunghiul si apoi paralelogramul,
   dupa actionarea unei taste ASCII oarecare.
*/
{
    triunghi t(80,10,100,210,50,150);
    paralelogram p(100,100,200,200,240,220,140,120);
    char tab[80];
    int gd = DETECT,gm;
    initgraph(&gd,&gm,"c:\BORLANDC\BGI");
    t.afiseaza();
    double per = t.perimetru();
    sprintf(tab,"perimetru triunghiului = %g",per);
    outtextxy(0,320,tab);
    double ar=t.arie();
    sprintf(tab,"aria triunghiului = %g",ar);
    outtextxy(0,330,tab);
    getch();
    cleardevice();
    p.afiseaza();
    per = p.perimetru();
    sprintf(tab,"perimetru paralelogramului = %g",per);
}

```

```

outtextxy(0,320,tab);
ar=p.arie();
sprintf(tab,"aria paralelogramului=%g",ar);
outtextxy(0,330,tab);
getch();
closegraph();
}

```

## 31. INTRĂRI/IEȘIRI

Limbajul C++, la fel ca limbajul C, nu are instrucțiuni specifice operațiilor de intrare/ieșire.

În limbajul C, astfel de operații se realizează cu ajutorul unui set de funcții din biblioteca standard a sistemului. Aceste funcții pot fi folosite în același mod și în programele scrise în C++.

În afara acestor funcții, biblioteca standard a limbajului C++ oferă utilizatorului ierarhii de clase care permit realizarea de astfel de operații.

La baza acestor operații se află conceptul de *stream* (flux). Prin *stream* se înțelege un flux de date de la o sursă la o destinație sau *consumator*.

În general, sursa de date poate fi tastatura, un fișier de pe disc sau chiar o zonă de memorie. În mod analog, destinația poate fi ecranul, un fișier pe disc sau o zonă de memorie.

Biblioteca standard a limbajului C++ conține două ierarhii de clase, una are ca rădăcină clasa *streambuf*, iar cealaltă, clasa *ios*.

Clasa *streambuf* furnizează funcții generale pentru lucru cu zonele tampon (bufere) și permite tratarea operațiilor de intrare/ieșire fără a avea în vedere formatari complexe. Din clasa *streambuf* derivă clasele *strstreambuf* și *filebuf*.

Clasa *ios* are un pointer spre *streambuf*. Ea are date membru pentru a gestiona interfața cu *streambuf* și pentru tratarea erorilor. Din clasa *ios* derivă două clase: *istream* pentru gestiunea intrărilor și *ostream* pentru gestiunea ieșirilor. Aceste derivări se realizează utilizând clasa *ios* drept clasă de bază virtuală:

**class istream:virtual public ios**

și

**class ostream:virtual public ios**

O a treia clasă, numită *iostream* derivă din ambele clase:

**class iostream:public istream,public ostream**

Clasele *istream*, *ostream* și *iostream* sunt, fiecare, clase de bază pentru alte trei derivări și anume:

**class istream\_withassign:public istream**

**class ostream\_withassign:public ostream**

**class iostream\_withassign:public iostream**

Clasa *istream* permite realizarea de conversii formatare sau neformatare ale caracterelor încărcate din obiecte de tip *streambuf*.

Clasa *ostream* permite conversii formatare sau neformatare în caractere care se păstrează în obiecte de tip *streambuf*.

Clasa *iostream* moștenește facilitățile ambelor clase de bază permitând operații în ambele direcții.

Clasele cu sufixul *withassign* furnizează "stream"-uri (fluxuri de date)

standard: *cin*, *cout*, *cerr* și *clog*.

Obiectul *cin* (console input) este o instanțiere a clasei *istream\_withassign* și el corespunde fișierului standard de intrare definit de pointerul *stdin*.

În mod analog, obiectul *cout* (console output) este o instanțiere a clasei *ostream\_withassign* și el corespunde fișierului standard de ieșire definit de pointerul *stdout*.

Obiectele *cerr* și *clog* sunt și ele instanțieri ale clasei *ostream\_withassign* și corespund fișierului standard de ieșire definit de pointerul *stderr*. Acestea au facilități suplimentare față de fișierul *stderr*.

Clasele cu sufixul *withassign* se deosebesc de clasele lor de bază prin aceea că ele suprainsarcă operatorul de atribuire.

De exemplu, clasa *istream\_withassign* se definește astfel:

```
class istream_withassign:public istream {
    istream_withassign();
    istream& operator = (istream&);
    istream& operator = (streambuf&);
};
```

O clasă *stream* este o clasă derivată din *istream* sau *ostream*.

Utilizarea ierarhiilor de clase amintite mai sus implică includerea fișierului *iostream.h*. Așa cum se va vedea mai târziu, în cazuri mai complexe de utilizare a acestor ierarhii este nevoie să se includă în program și alte fișiere de extensie *h*.

Utilizarea facilităților oferite de clase, conduce la o flexibilitate și claritate mai mare în realizarea operațiilor de intrare/ieșire față de funcțiile standard obișnuite: *scanf*, *printf*, *gets*, *puts* etc.

Un avantaj imediat al utilizării *stream*-urilor este acela de a elimina erorile care apar în mod frecvent la apelul funcției *printf*, cind numărul specifiicatorilor de format diferă de cel al parametrilor efectivi. La aceasta adăugăm și posibilitățile acestor clase de a realiza operații de intrare/ieșire nu numai cu date de tipuri predefinite, ci și cu obiecte de tipuri abstracte. Utilizatorul are posibilitatea să definiască simplu conversii pentru obiectele asupra cărora se realizează operații de intrare/ieșire.

Obiectele care sunt instanțieri ale claselor ierarhiei care are ca rădăcină clasa *ios* se numesc *stream*-uri. În particular, obiectele *cin*, *cout*, *cerr* și *clog* sunt *stream*-uri standard.

## 31.1. Intrări/ieșiri standard

Prin intrări/ieșiri standard înțelegem schimbul de informații care se realizează cu terminalul de la care s-a lansat programul. Așa cum s-a afirmat mai sus, se pot folosi 4 obiecte standard pentru a realiza intrări/ieșiri standard: *cin*, *cout*, *cerr* și *clog*.

Obiectul *cin* este o instanțiere a clasei *istream\_withassign*, iar celelalte sunt instanțieri ale clasei *ostream\_withassign*. Aceste obiecte sunt inițializate prin

includerea fișierului *iostream.h*. La includerea acestui fișier se apelează constructorii claselor mai sus amintite.

Obiectele *cin*, *cout*, *cerr* și *clog* se instantiază și inițializează o singură dată, chiar dacă fișierul respectiv se include de mai multe ori. În felul acesta, la lansarea programului, obiectele respective sunt disponibile și pot fi utilizate pentru a realiza operații de intrare/ieșire de la terminalul standard.

Menționăm că aceste operații pot fi redirectate spre alte periferice, la fel cum se procedeză cu funcțiile *scanf*, *printf* etc.

### 31.1.1. Ieșire standard

Ieșirile standard se pot realiza folosind operatorul "*<<*". Acesta este suprainsarcăt pentru operații de ieșire. Operandul sting este un obiect al clasei *ostream*.

Operandul din dreapta are un tip pentru care a fost suprainsarcăt operatorul "*<<*". Acesta poate fi un tip predefinit sau un tip abstract.

Operatorul "*<<*" suprainsarcăt pentru operații de ieșire se numește *operator de inserare* (insertion) sau *pune la ...* (put to).

Prototipul funcției membru a clasei *ostream* care suprainsarcă operatorul de inserare este următorul:

*ostream& operator << (tip);*

În mod implicit, operatorul de inserare este suprainsarcăt pentru tipurile predefinite: *char* (cu semn sau fără semn), *short*, *int*, *long*, *unsigned*, *unsigned long*, *char \** (se utilizează pentru siruri de caractere), *float*, *double*, *long double* și *void \** (pentru pointeri, ieșirea se face în hexazecimal).

Amintim că operatorul "*<<*" este în mod normal operatorul de deplasare la stinga. El are prioritatea imediat mai mică decât operatorii binari aditivi (+ și -) și imediat mai mare decât operatorii relaționali. Operatorii de deplasare se asociază de la stinga la dreapta. Aceste reguli rămân valabile și pentru operatorul de inserare. Deoarece la suprainsarcarea operatorului de inserare se returnează o referință la obiectul curent, operatorii de inserare se pot aplica înlanțuți.

**Exemple:**

```
int i;
...
cout << i; // afiseaza valoarea lui i
// are același efect cu apelul
printf("%d", i);
...
cout << "\n"; // afiseaza newline
// are același efect cu apelul
printf("\n");
// sau
```

```

putchar('\n');

double d;
...
cout << d; // afisaza valoarea lui d

// are același efect cu apelul
printf("%g", d);

char c;
...
cout << c; // afisaza caracterul al carui cod ASCII coincide cu valoarea lui c

// are același efect cu apelul
printf("%c", c);

char t[] = "Acesta este un sir de caractere";
...
cout << t; // afisaza sirul spre care pointeaza t

// are același efect cu apelul
printf("%s", t);

int a;
...
cout << &a; // afisaza, in hexazecimal, adresa zonei alocate lui a
cout << i+di; // afisaza valoarea expresiei i+di;
// operatorul << este mai putin prioritara decat +
cout << (i&0377); // afisaza valoarea octetului mai putin semnificativ a lui i

```

În ultima instrucțiune sunt necesare parantezele rotunde datorită faptului că operatorul "<<" este mai prioritara decat "&".

Instrucțiunea expresie:

```
cout << "i=" << i;
```

afisează:

i = valoarea lui i

Are același efect cu apelul:

```
printf("i=%d", i);
```

Instrucțiunea:

```
cout << "i=" << i+10 << "\n";
```

are același efect cu apelul:

```
printf("i=%d\n", i+10);
```

Funcția *printf* oferă facilități de formatare complexă a datelor de ieșire. Astfel de formatari sunt posibile și în cazul operatorului de inserare. În acest caz, se utilizează diferenți indicatori de format. Aceștia sunt definiți în clasa *ios*, ca mai jos:

```

public:
enum{
    skipws=0x0001, // salt peste caractere albe la intrare
    left=0x0002, // ieșire cadrată în stînga
    right=0x0004, // ieșire cadrată în dreapta
    internal=0x0008, // caractere nesemnificative după semn sau indicator de baza
    dec=0x0010, // conversie în zecimal
    oct=0x0020, // conversie în octal
    hex=0x0040, // conversie în hexazecimal
    showbase=0x0080, // se afisează baza
    showpoint=0x0100, // se afisează punctul zecimal
    uppercase=0x0200, // ieșire hexazecimală cu litere mari
    showpos=0x0400, // întregii pozitivi sunt afișați cu + în față
    scientific=0x0800, // numerele flotante se afisează cu exponent
    fixed=0x1000, // numerele flotante se scriu fără exponent
    unitbuf=0x2000, // se videază zonele tampon pentru toate inserările
    stdio=0x4000, // după fiecare inserare se videază stdcout și stderr
};

...

```

Acești enumeratori reprezintă biți din dublul cuvînt *x\_flag* care este dată membru a clasei *ios*. Acești biți sunt, în mod implicit, să încît să se asigure formate standard la ieșire. Acestea sunt aceleași cu formatele definite de următorii specificatori de format utilizati la apelul funcției *printf* sau *scanf*:

%d	pentru int;
%ld	pentru long;
%u	pentru unsigned;
%lu	pentru unsigned long;
%g și %lg	pentru float și double;
%Lg	pentru long double;
%c	pentru char;
%s	pentru șiruri de caractere.

Utilizatorul poate seta biții din data membru *x\_flag* pentru a realiza formatari diferite decât cea implicită.

Setările biților din *x\_flag* se fac cu funcții membru ale claselor ierarhiei care au ca rădăcină clasa *ios*.

În afară de setările biților din data membru *x\_flag*, la definirea formatorilor, utilizatorul are posibilitatea de a defini dimensiunea cîmpului, la fel ca în cazul specificatorilor de format utilizati la apelul funcțiilor *printf* și *scanf*.

În mod implicit, la afișarea unei date, cîmpul de afișare are atîtea caractere cîte sunt necesare pentru afișarea datei respective. Utilizatorul poate impune un cîmp minim pentru afișare cu ajutorul funcției membru *width* a clasei *ios*.

Funcția *width* are două prototipuri:

int width();

și

int width(int w);

Apelul fără parametru al funcției *width* returnează dimensiunea curentă a

cimpului. Ea este definită de data membru *x\_width* a clasei *ios*.

În mod implicit, valoarea dimensiunii cimpului este zero, ceea ce înseamnă că data se afișează pe un cimp de lungime egală cu numărul de caractere necesar afișării ei.

Apelul funcției *width* cu parametrul *w* permite utilizatorului să definească lungimea minimă a cimpului în care se realizează afișarea. În acest caz, funcția *width* returnează dimensiunea curentă a cimpului înaintea modificării ei la valoarea lui *w*.

Dacă dimensiunea curentă a cimpului este mai mică decit numărul de caractere al datei care se afișează, atunci data respectivă se va afișa fără trunchiere, adică pe atitea caractere cîte sint necesare. Dacă dimensiunea curentă a cimpului este mai mare decit numărul de caractere al datei care se afișează, atunci data respectivă se afișează în cimpul respectiv completată cu caractere nesemnificative pînă la dimensiunea cimpului. În mod implicit data se afișează cadrată în dreapta în cimpul curent, iar caracterele nesemnificative sint spații. În felul acesta, funcția *width* asigură compatibilitate pentru interpretarea lungimii cimpului curent la operația de inserare cu cea definită de specifiicatorii de format utilizati la apelul funcției *printf*.

**Exemplu:**

```
int i;
...
i=123;
cout.width(5); // se setează dimensiunea cimpului curent la valoarea 5
cout << i;
```

Valoarea lui *i* se afișează pe un cimp de 5 caractere, cadrată în dreapta.

Primele două caractere ale cimpului conțin spații. Același efect se obține prin apelul de mai jos:

```
printf("%5d", i);
```

După o operație de inserare, dimensiunea cimpului devine în mod automat cea implicită, adică egală cu zero.

Caracterele nesemnificative care însoțesc datele care se afișează se mai numesc și *caractere de umplere*. Ele pot fi definite de utilizator cu ajutorul funcției membru *fill* a clasei *ios*.

Funcția *fill* are două prototipuri:

```
char fill();
și
char fill(char c);
```

Funcția fără parametru returnează codul ASCII al caracterului de umplere curent.

Apelul funcției *fill* cu parametru conduce la definirea caracterului de umplere și anume, caracterul de umplere devine egal cu cel al cărui cod ASCII este egal

cu valoarea lui *c*. Funcția returnează valoarea codului ASCII al vechiului caracter de umplere.

**Exemplu:**

```
int i=123;
...
cout.width(5);
cout.fill('0');
cout << i;
```

Se afișează:

00123

Același efect se obține cu ajutorul apelului funcției *printf* de mai jos:

```
printf("%05d", i);
```

Funcția *fill* oferă facilități superioare pentru a defini caracterele de umplere față de specifiicatorii de format utilizati la apelul lui *printf*. Într-adevăr, funcția *fill* ne permite să utilizăm drept caractere de umplere și alte caractere decit zero și cel implicit, care sint singurele caractere utilizabile folosind funcția *printf*. De exemplu, apelul:

```
cout.fill('*');
```

definește drept caracter de umplere, caracterul asterisc.

Indicatorii de format enumerați mai sus, care definesc un atribut de format, sint grupați în trei grupe. Aceste grupe pot fi referite prin denumiri atribuite lor. Primul grup de biți formează grupa care definește cadrul datei în cimpul de afișare. Ea se numește *adjustfield* și este formată din biții *right*, *left* și *internal*. Acești biți definesc pozițiile caracterelor de umplere: în stînga datei care se afișează (modul implicit), în dreapta datei sau în interiorul ei.

O altă grupă este denumită *basefield* și ea conține biții *dec*, *oct* și *hex*. Ea definește baza de numerație a datei care se afișează.

În sfîrșit, cea de a treia grupă este denumită *floatfield* și conține biții *scientific* și *fixed* care definesc formatul de afișare al numerelor: cu exponent sau fără.

O caracteristică generală a acestor grupe este faptul că în cadrul fiecarei, numai un bit poate fi setat. De exemplu, dacă este setat bitul *left*, atunci ceilalți biți din aceeași grupă nu mai pot fi setați.

Biții indicatori de format pot fi setați folosind funcția membru *setf* a clasei *ios*. Aceasta are două formate:

```
long setf(long f);
și
long setf(long bit, long grupă);
```

Ambele funcții returnează valoarea precedentă a dublu cuvintului *x\_flag* care definește formatul vechi.

Primul format al funcției *setf* modifică formatul curent setind biții setați în

valoarea lui *f*. Ceilalți biți (care corespund biților avind valoarea zero) ai formatului rămân nemodificați.

Cel de al doilea format se utilizează pentru a seta biții din cele trei grupe amintite mai sus: *adjustfield*, *basefield* și *floatfield*. Primul parametru definește bitul din cadrul grupei, iar cel de al doilea grupă.

La apelurile acestei funcții se pot utiliza enumeratoare și denumirile celor trei grupe. Ei vor fi prefixați de numele *ios* urmat de operatorul de rezoluție.

#### Exemple:

```
1. int i = 123;
cout.width(5);
cout.setf(ios::left,ios::adjustfield);
cout.fill('0');
cout << i;
```

Se afișează:

12300

```
2. int i = 123;
cout.width(5);
cout.setf(ios::right,ios::adjustfield);
cout.fill(' ');
cout.setf(ios::showpos);
cout << i;
```

Se afișează:

+123

caracterul + fiind precedat de caracterul spațiu.

```
3. int i = 123;
cout.width(5);
cout.setf(ios::showpos);
cout.fill('0');
cout.setf(ios::internal,ios::adjustfield);
cout << i;
```

Se afișează:

+0123

```
4. int i = 123;
cout.setf(ios::hex,ios::basefield);
cout << i;
```

Se afișează:

7b

Se obține același efect ca la apelul:

```
printf("%x",i);
```

```
5. double d = 123.45;
cout << d;
```

Se afișează:

123.45

```
6. double d = 123.45;
```

```
cout.setf(ios::scientific,ios::floatfield);
cout << d;
```

Se afișează:

1.2345e+02

Numele de zecimale se poate defini cu ajutorul funcției membru *precision* a clasei *ios*. Ea are două prototipuri:

```
int precision();
int precision(int n);
```

În primul caz, se returnează precizia curentă. În cel de al doilea caz, se returnează precizia existentă înaintea modificării. Precizia se modifică la *n* caractere.

#### Exemplu:

```
7. double pi = 3.14159265;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(7);
cout << pi;
Se afișează:
3.1415927
```

#### Exerciții:

31.1 Să se scrie un program care afișează la display datele din exemplele 1-7 din paragraful 31.1.1.

#### PROGRAMUL BXXXI

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

main()
/* afiseaza date folosind icerarhile de clase care implementeaza conceptul de stream */
{
    int i=123;

    // exemplul 1
    cout.width(5);
    cout.setf(ios::right,ios::adjustfield);
    cout.fill('0');
    cout << i;
    cout << "\n Actionati o tasta pentru a continua\n";
    getch();

    // exemplul 2
    cout.width(5);
    cout.fill(' ');
    cout.setf(ios::left,ios::adjustfield);
    cout.setf(ios::showpos);
    cout << i;
```

```

cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 3
cout.width(5);
cout.setf(ios::showpos);
cout.fill('0');
cout.setf(ios::internal,ios::adjustfield);
cout << i;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 4
cout.setf(ios::hex,ios::basefield);
cout << i;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 5
double d = 123.45;
cout << d;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 6
cout.setf(ios::scientific,ios::floatfield);
cout << d;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 7
double pi = 3.14159265;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(7);
cout << pi;
cout << "\n Actionati o tasta pentru a continua\n";
getch();
}

```

### 31.1.1. Manipulatori

Manipulatorii permit definirea formatelor pentru operațiile de intrare/ieșire. Practic, toate facilitățile de definire ale formatelor descrise mai sus, folosind funcțiile membru *width*, *setf*, *fill* etc., pot fi realizate cu ajutorul *manipulatorilor*. Aceștia sunt funcții membru speciale. O astfel de funcție returnează o referință la un stream. În felul acesta, apelurile manipulatorilor se pot înlănuiri în inserări cu scopul de a defini biții de format.

Indicăm mai jos numele, parametri și acțiunea fiecărui manipulator.

Manipulator	Acțiune
<b>dec</b>	setează indicatorul de conversie în zecimal
<b>oct</b>	setează indicatorul de conversie în octal
<b>hex</b>	setează indicatorul de conversie în hexazecimal
<b>ws</b>	setează indicatorul skipws de salt peste caracterele albe
<b>endl</b>	înscrează newline și videază zona tampon a streamului
<b>ends</b>	înscrează caracterul nul care termină șirurile de caractere
<b>flush</b>	videază zona tampon a unui obiect al clasii ostream
<b>setbase(int n)</b>	setează baza de conversie egală cu n; n poate avea una din valorile 0,8,10 sau 16; valoarea zero înseamnă conversie implicită, adică în zecimal
<b>resetiosflags(long a)</b>	șterge biții de format specificați de parametrul a
<b>setiosflags(long a)</b>	setează biții de format specificați de parametrul a
<b>setfill(int n)</b>	setează caracterul de umplere la valoarea lui n
<b>setprecision(int n)</b>	setează precizia la valoarea lui n
<b>(setw(int n))</b>	setează dimensiunica cimpului la valoarea lui n

Manipulatorii *dec*, *oct* și *hex* definesc baza de conversie. Aceasta rămîne valabilă în continuare, pînă la o nouă modificare a ei.

Utilizarea manipulatorilor implică includerea fișierului *iomanip.h*.

### Exerciții:

31.2 Programul de față realizează aceleși afișări ca și programul din exercițiul 31.1, utilizând manipulatori la definirea formatelor.

### PROGRAMUL BXXXI2

```

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

main()
/* afiseaza date folosind ierarhiile de clase care implementeaza conceptul de stream */
{
    int i=123;

```

```

// exemplul 1
cout << setw(5) << resetiosflags(ios::internal|ios::right) <<
    setiosflags(ios::left) << setfill('0') << i;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 2
cout << setw(5) << resetiosflags(ios::internal|ios::left) <<
    setiosflags(ios::right) << setfill(' ') <<
    setiosflags(ios::showpos) << i;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 3
cout << setw(5) << resetiosflags(ios::left|ios::right) <<
    setiosflags(ios::internal|ios::showpos) <<
    setfill('0') << i;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 4
cout << hex << i;
cout << dec; // se reface conversia in zecimal
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 5
double d = 123.45;
cout << d;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 6
cout << resetiosflags(ios::fixed) <<
    setiosflags(ios::scientific) << d;
cout << "\n Actionati o tasta pentru a continua\n";
getch();

// exemplul 7
double pi = 3.14159265;
cout << setprecision(7) << pi;
cout << "\n Actionati o tasta pentru a continua\n";
getch();
}

```

### 31.1.1.2. Ieșire neformatată

Ieșirea unui singur caracter se poate realiza cu ajutorul funcției membru *put* a clasei *ostream*. Aceasta are prototipul:

```
ostream& ostream::put(char c);
```

Apelul:

```
char c='A';
cout.put(c);
```

are același efect cu instrucțiunea expresie:

```
cout << c;
```

dacă biții de format au valori implicate.

Pentru a afișa *n* octeți, se poate utiliza funcția membru *write* a clasei *ostream*. Ea are următoarele prototipuri:

```
ostream& write(const signed char *sir,int n);
```

și

```
ostream& write(const unsigned char *sir,int n);
```

Funcția *write* afișează *n* caractere chiar dacă printre cei *n* octeți spre care pointează *sir* se întâlnește caracterul *nul*.

Esența acestor funcții este faptul că, asupra lor biții de format nu au nici un efect.

### 31.1.1.3. Suprainscrierea operatorului << pentru ieșiri de obiecte

Utilizarea operatorului de inserare pentru tipuri abstracte este posibil dacă acesta se suprainscrie în mod corespunzător.

De exemplu, operatorul << se utilizează în expresii de forma:

(1) stream << data\_de\_tip\_predefinit

El poate fi suprainscrat pentru a fi utilizat cu obiecte care sunt instanțieri ale unei clase. Cu alte cuvinte, operatorul de inserare se suprainscră astfel încât expresia:

(2) stream << obiect

să fie acceptată de compilator.

Operandul *stream* este o instanțiere a clasei *ostream* și de aici rezultă că operatorul << poate fi suprainscrat pentru o clasă oarecare numai printr-o funcție prieten. Din expresia (2), rezultă că funcția care suprainscră operatorul de inserare are doi parametri: primul este o referință la clasa *ostream*, iar cel de al doilea parametru este un obiect al cărui tip este tipul abstract pentru care se face suprainscrierea operatorului respectiv.

Deci, prototipul funcției care suprainscră operatorul de inserare pentru clasa C este următorul:

```
friend ostream& operator << (ostream&,C);
```

Funcția returnează o referință la clasa *ostream* pentru ca operatorul de inserare să se poată înălțui ca în cazul operanzilor de tipuri predefinite.

De exemplu, dacă clasa C este clasa care implementează conceptul de număr *complex*, atunci prototipul funcției care suprainscră operatorul de inserare va fi:

```
friend ostream& operator << (ostream&,complex);
```

Acăstă funcție o putem defini astfel:

```
ostream& operator << (ostream& iesire,complex z)
// supraincarca operatorul de inserare
{
    iesire << z.real << "(" << z.imag << ")i";
    return iesire;
}
```

Clasa *complex* a fost definită în mai multe exerciții din cartea de față. O parte din elementele clasei *complex* se definesc ca mai jos:

```
class complex {
    double real;
    double imag;
public:
    complex(double x,double y)
    {
        real = x; imag = y;
    }

    friend ostream& operator << (ostream&,complex);
    ...
};

Fie instanțierea:
```

```
complex a(1,2);
```

Instrucțiunea expresie:

```
cout << "a=" << a << "\n";
```

Afișează:

```
a = 1+(2)i
```

### 31.1.2. Intrare standard

Intrările standard se pot realiza folosind operatorul ">>". Acesta este supraincarcat pentru intrările standard. Operandul stîng este un obiect al clasei *istream*. Operandul din dreapta are un tip pentru care a fost supraincarcat operatorul ">>". Acesta poate fi un tip predefinit sau un tip abstract.

Operatorul ">>" supraincarcat pentru operații de intrare se numește *operator de extragere* (extraction) sau *obține de la...* (get from).

În mod implicit, operatorul de extragere este supraincarcat pentru tipurile predefinite și pentru siruri de caractere.

Amintim că operatorul ">>" este, în mod normal, operatorul de deplasare la dreapta. El are aceeași prioritate și asociativitate ca și operatorul "<<".

Funcția care supraincarcă operatorul de extragere returnează o referință la obiectul curent și de aceea, operatorii de extragere se pot aplica înlanțuiți.

Prototipul funcției membru a clasei *istream* care supraincarcă operatorul de *extragere* este următorul:

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

În mod implicit, operatorul de extragere realizează avansul peste caracterele albe pînă la primul caracter care nu este alb.

Utilizatorul poate să suprime acest avans prin stergerea bitului *ios::skipws*. În acest scop se poate folosi manipulatorul *resetiosflags* (vezi paragraful 31.1.1.).

Caracterele cîmpului curent trebuie să corespundă tipului de date care formează operandul drept al operatorului de extragere. În caz contrar, se intră în starea de eroare.

**Exemple:**

1. int i;

```
cin >> i;
```

Această instrucțiune expresie are același efect cu apelul:  
`scanf ("%d", &i);`

2. double d;

```
cin >> d;
```

Această instrucțiune expresie are același efect cu apelul:  
`scanf ("%lf", &d);`

3. char c;

```
cin >> c;
```

Această instrucțiune expresie citește primul caracter care nu este alb (dacă bitul *ios::skipws* nu este în prealabil șters); are același efect cu secvența:  
`scanf ("%1s", tc);`  
unde *tc* este un tablou de tip *char* de cel puțin 2 caractere:  
`char tc[2];`

4. char tab[100];

```
cin >> tab;
```

Această instrucțiune expresie are același efect cu apelul:  
`scanf ("%s", tab);`

În legătură cu sirurile de caractere, poate apărea situația cind la intrare se află un sir de caractere care depășește dimensiunea tabloului destinat pentru a păstra caracterele citite. Pentru a preîmpinge acest lucru, se poate defini dimensiunea cîmpului egală cu a zonei receptoare a sirului de caractere care se citește. În acest scop, se va folosi funcția *width* (vezi paragraful 31.1.1.) într-o secvență de forma:

```
char tab[...];
cin.width(sizeof tab);
cin >> tab;
```

În mod implicit, cimpul din care se face transferul incepe la primul caracter care nu este alb și se termină la primul caracter care nu aparține tipului operatorului de extragere sau la întâlnirea unui caracter alb.

Utilizatorul poate defini dimensiunea maximă a cimpului cu ajutorul funcției *width*, ca în exemplul de mai sus.

În cazul operatorului de extragere se pot folosi aproape toți manipulatorii definiti în paragraful 31.1.1.1.

În particular, manipulatorul ws se poate folosi numai în operații de extragere. El permite activarea sau dezactivarea saltului peste caracterele albe.

Manipulatorii care nu pot fi utilizati în operații de extragere decit numai în operații de inserare sunt:

*endl, ends, flush și setbase.*

Un *stream* are asociat o *stare de eroare* care se definește cu ajutorul datei membru *state* a clasei *ios*. Biții acestei date au semnificația indicată mai jos:

```
class ios {
public:
    ...
    enum io_state {
        goodbit = 0x00, // operația de intrare/iesire corecta daca acest bit
                        // este pozitionat celalalt sint stersi
        eofbit=0x01, // s-a intilnit sfîrșitul de fisier intr-o operatie de intrare
        failbit=0x02, // se seteaza daca ultima operatie de intrare/iesire
                        // (extragere sau conversie) a esuat; streamul este utilizabil
                        // in continuare daca se sterge bițul de eroare
        badbit=0x04, // se seteaza daca ultima operatie de intrare/iesire a fost
                        // invalida; streamul poate fi utilizat in continuare dupa
                        // ce se sterge conditia de eroare
        hardfail=0x08 // se seteaza la o eroare irecuperabila
    };
    ...
};
```

Un *stream* intrat într-o stare de eroare nu mai permite operații de intrare/iesire (inserare sau extragere) pînă cînd condiția de eroare este înlăturată și biții de eroare sunt sterși.

Accesul la biții de eroare a datei *state* se obține folosind următoarele funcții:

- > *int good();* - Returnează o valoare diferită de zero dacă operația de intrare/iesire s-a efectuat fără erori (bițul *goodbit* este setat).
- > *int eof();* - Returnează o valoare diferită de zero dacă s-a întilnit sfîrșitul de fisier (bițul *eofbit* este setat).
- > *int fail();* - Returnează o valoare diferită de zero dacă este poziționat cel puțin unul din biții *failbit* sau *hardfail*.
- > *int bad();* - Returnează o valoare diferită de zero dacă este poziționat bițul *badbit*.

Valoarea datei membru *state* este returnată de funcția:

*int rdstate();* - Biții de eroare pot fi modificați folosind funcția membru *clear* a clasei *ios*.

Această funcție are prototipul:

```
void clear(int i= 0);
```

Ea permite setarea biților de eroare.

De exemplu, pentru a seta bițul *failbit* vom utiliza construcția:

(1) *ios::failbit*

Pentru ca ceilalți biți să rămână pe loc, apelăm funcția *rdstate* pentru streamul curent, de exemplu, pentru *cin*:

(2) *cin.rdstate()*

Bițul *failbit* se setează apelind funcția *clear* ca mai jos:

```
cin.clear(ios::failbit|cin.rdstate());
```

Un apel, fără parametri, al funcției *clear* permite anularea tuturor biților de eroare, cu excepția bitului *hardfail* care nu poate fi anulat.

Un stream poate fi testat pentru a stabili dacă se află sau nu într-o stare de eroare, la fel ca o expresie logică.

Aceasta este posibil deoarece clasa *ios* are suprîncărcat operatorul "!", precum și conversia unui *stream* într-un pointer spre *void*.

Funcția membru care suprîncarcă operatorul "!" are prototipul:

```
int operator !();
```

Acest operator poate fi utilizat în expresii de forma:

```
if (!stream)
    ...
```

Expresia *!stream* are o valoare diferită de zero (adevărat), dacă cel puțin unul din biții de eroare ai stării lui *stream* este setat; altfel expresia are valoarea zero (fals).

Funcția de conversie a streamului într-un pointer spre *void* are prototipul:

```
operator void *();
```

Conversia streamului într-un pointer spre *void* are ca rezultat pe zero (pointerul nul) dacă cel puțin unul din biții de eroare ai stării streamului este setat și o valoare diferită de zero în caz contrar. Menționăm că pointerul rezultat printr-o astfel de conversie se poate folosi numai pentru a testa dacă biții de eroare ai stării streamului sunt setați sau nu.

O expresie de forma:

```
cin >> v
```

are ca valoare o referință la streamul *cin*, de tip *istream*. O astfel de referință poate fi utilizată într-o instrucție de forma:

```
if(cin >> v)
```

În acest caz, se aplică automat conversia referinței respective spre un pointer spre *void*. Rezultatul acestei conversii este pointerul nul dacă cel puțin un bit de eroare al stării streamului *cin* este setat.

În caz contrar, pointerul rezultat este nenul.

#### Exerciții:

31.3 Să se scrie o funcție care realizează următoarele:

- afișează un text la terminalul standard;
- citește un întreg de tip *int* de la intrarea standard;
- returnează valoarea zero dacă se întâlnește sfîrșitul de fișier și unu în caz contrar.

Funcția are doi parametri:

*text* - Este pointerul spre sirul de caractere care se afișează la terminal.

*n* - Este referință la intregi de tip *int*.

Funcția păstrează întregul citit în zona de memorie referită de *n*.

#### FUNCȚIA BXXXI3

```
#ifndef __Iostream_H
#include <iostream.h>
#define __Iostream_H
#endif

int citInt(char *text,int& n)
/* - afișaza text;
   - citește un întreg de tip int și-l păstrează în zona referită de n;
   - returnează zero la sfîrșitul de fișier și unu altfel.
*/
{
    char t[255];

    for(;;) {
        cout << text;
        if(cin >> n) return 1; //citire fără eroare
        if(cin.eof()) return 0; //s-a întâlnit EOF

        //sterge starea de eroare
        cin.clear();

        //videază buferul
        cin >> t;

        cout << "Nu s-a tastat un întreg\n";
    }
}
```

31.4 Să se scrie un program care citește, de la intrarea standard, un sir de numere de tip *int* și afișează suma lor. După ultimul număr al sirului se tastează sfîrșitul de fișier.

#### PROGRAMUL BXXXI4

```
#include <conio.h>
#include <stdio.h>
#include "BXXXI3.CPP"

main()
/* citește numere de tip int și afișează suma lor */
{
    int s = 0;
    int i = 1;
    int nr;

    for(;;) {
        char zona[20];
        sprintf(zona,"n%d=",i);
        if(citInt(zona,nr) == 0) break;
        s += nr;
        i++;
    }
    cout << "S-au citit " << i-1 << " intregi\n";
    cout << "Suma lor este = " << s << "\n";
}
```

#### Observație:

După modelul funcției *citInt*, se pot construi funcții care să permită citirii de numere de tip *long*, *float*, *double* sau *long double*. Propunem cititorului să realizeze singur astfel de funcții. Pentru testarea lor se pot folosi programe de felul celui de mai sus.

Instrucțiunea expresie:

```
cin >> t;
```

utilizată în funcția *citInt* pentru avida buferul, extrage din obiectul *cin* caractere din cimpul care începe cu caracterul eronat (care nu corespunde unui întreg) și pină la întâlnirea unui caracter alb. Dacă după acest caracter alb urmează și alte caractere, atunci instrucțiunea de mai sus nu este suficientă pentru vidarea buferului de intrare. O secvență eficientă în toate situațiile se indică în paragraful 31.2.

##### 31.1.2.1. Intrări neformatare

Intrarea unui singur caracter se poate realiza cu ajutorul funcției membru *get* a clasei *istream*. Ea are prototipul:

```
istream& get(char&);
```

Se obișnuiește să se spună că funcția *get* extrage un caracter din streamul curent și-l păstrează în zona de memorie referită de parametrul formal.

Funcția *get* nu realizează avansul peste caracterele albe, deci ea extrage din streamul curent caracterul, indiferent că acesta este un spațiu alb sau nu.

Funcția *get* are și alte prototipuri care permit extragerea unui sir de caractere din streamul curent și anume:

```
istream& get(unsigned char *zona,int max,int term = '\n');
istream& get(signed char *zona,int max,int term = '\n');
```

Această funcție extrage din streamul curent cel mult *max-1* caractere sau pînă la întîlnirea caracterului dat de valoarea lui *term*. Caracterele respective se păstrează în zona de memorie spre care pointează *zona*.

După caracterele extrase din streamul curent se păstrează caracterul NUL ('\0'). De aceea, zona de memorie spre care pointează *zona* trebuie să conțină cel puțin *max* octeți.

Terminatorul definit de *term* nu se păstrează în zona de memorie spre care pointează *zona*. De altfel, dacă extragerea se termină la întîlnirea lui, acesta nici nu este extras din streamul curent.

Extragerea caracterului definit de *term* din streamul de intrare este posibilă cu ajutorul funcției *getline*.

Aceasta are un prototip similar cu funcția *get*:

```
istream& getline(char *zona,int max,int term = '\n');
```

Parametrii formalii ai funcției *getline* au același sens ca în cazul funcției *get*.

Terminatorul definit de *term* nu se transferă în zona de ieșire ci numai se extrage din streamul de intrare.

Extragerea binară a datelor din streamul curent se realizează folosind funcția *read*. Aceasta are prototipurile:

```
istream& read(unsigned char *zona,int n);
istream& read(signed char *zona,int n);
```

Funcția extrage *n* octeți din streamul curent și îi transferă în zona de memorie spre care pointează *zona*.

**Exemple:**

```
1. char c;
   cin.get(c);
```

Are același efect cu apelul:  
c=getchar();

```
2. char tab[100];
   cin.get(tab,100,'\\n');
```

Citește cel mult 99 de caractere de la intrarea standard pe care le păstrează în tabloul *tab*. Dacă se întâlnește caracterul '\n' înainte de a citi 99 de caractere, citirea se întrerupe. Caracterul '\n' rămîne în zona tampon a

streamului curent.

După ultimul caracter transferat din streamul curent se memorează caracterul NUL.

O funcție membru a clasei *istream* care se utilizează frecvent este funcția *putback*. Aceasta permite punerea unui caracter într-un stream de intrare. Streamul intră în starea "fail" dacă nu se poate pune caracterul respectiv în stream.

Funcția *putback* are prototipul:

```
istream& putback(char c);
```

Un apel de forma:

```
cin.putback(c);
```

pune caracterul *c* în streamul *cin*.

Extragerea referitoare la *cin*, care urmează acestui apel, va începe cu caracterul *c*.

Funcția *putback* își găsește utilizări în programe de *analiză lexicală* care recunosc diferite tipuri de date aflate la intrare. De exemplu, dacă la intrare se află o expresie de forma:

a1b2+x

atunci analiza lexicală a acesteia trebuie să recunoască următoarele tipuri de date:

```
a1b2 identificator;
+ caracter special;
x identificator.
```

În general, prin *identificator* se înțelege o succesiune de litere și cifre care incep cu o literă. Pentru a recunoaște un identificator vom proceda la citirea ciclică a caracterelor de la intrare cit timp acestea sunt litere sau cifre. La întîlnirea unui caracter care nu este nici literă și nici cifră, înseamnă că s-a "avansat" peste caracterele identificatorului, citindu-se chiar și caracterul care urmează după identificator. De obicei, acest caracter trebuie pus înapoi, pentru a fi recitat și recunoscut ulterior. În caz contrar, el poate fi pierdut. De exemplu, în cazul de mai sus, după citirea caracterelor identificatorului *a1b2* se citește caracterul "+". Abia în acest moment se știe că identificatorul respectiv s-a terminat. Urmează repunerea caracterului "+" în streamul de intrare pentru a putea fi ulterior recitat și tratat.

Tinând seama de cele de mai sus, putem construi o funcție pentru recunoașterea identificatorilor ca mai jos:

```
void ident(char *id)
/* citește caracterele identificatorului curent și le păstrează în zona spre care pointează id */
{
    *id = '\\0';
    char car;
    car = 0;
```

```

    cin >> car; // salt peste spatii albe
    if(car >= 'A' && car <= 'Z' || car >= 'a' && car <= 'z')
    // litera
        do{
            *id++ = car; // pastreaza caracterul curent al identificatorului
            car = 0;
            cin.get(car);
            int i = car >= 'A' && car <= 'Z' ||
                car >= 'a' && car <= 'a' ||
                car >= '0' && car <= '9';
            /* i=1 pentru cazul cand car este litera sau cifra;
               i=0 altfel */
        } while(i);
    *id = '\0'; // sfarsitul identificatorului

    /* car contine un caracter care nu este nici litera si nici cifra;
       acest caracter este urmator identificatorului extras din streamul cin;
       acesta trebuie repus in streamul cin pentru a putca si recitit ulterior */

    cin.putback(car);
}

```

### 31.1.2.2. Supraincărcarea operatorului >> pentru intrări de obiecte

Utilizarea operatorului de extragere pentru tipuri abstracte este posibilă dacă acesta se supraincarcă în mod corespunzător.

În cazul tipurilor predefinite, operatorul de extragere a fost folosit în expresii de forma:

(1) `cin >> data_de_tip_predefinit`

De aceea, alături de expresia (1), se dorește acceptarea de către compilator a expresiilor de forma:

(2) `cin >> obiect`

unde *obiect* este o instanțiere a unei clase oarecare CL. Acest lucru este posibil dacă se supraincarcă operatorul ">>" în mod corespunzător pentru clasa CL. O astfel de supraincărcare este posibilă printr-o funcție prietenă clasei CL. Antetul funcției va fi următorul:

`istream& operator >> (istream&,CL&);`

Funcția returnează o referință la clasa *istream* ceea ce permite utilizarea înlanțuită a operatorului de *extragere*.

De exemplu, dacă CL este clasa care implementează conceptul de număr complex, atunci prototipul funcției care supraincarcă operatorul de extragere va fi:

`friend istream& operator >> (istream&,complex&);`

### Exerciții:

31.5 Să se implementeze tipul abstract *epunct* pentru gestiunea punctelor pe ecranul grafic.

### FIŞIERUL BXXXI5.H

```

#ifndef __GRAPHIC_H
#include <graphics.h>
#define __GRAPHIC_H
#endif
#ifndef __Iostream_H
#include <iostream.h>
#define __Iostream_H
#endif

class epunct {
protected:
    int x;
    int y;
    int valid; /* valid=1 daca punctul se afla in limitele ecranului grafic;
                  altfel valid=0 */
    int culoare;
    int vizibil;

    void testecran();
    /* defineste valoarea variabili valid pentru obiectul curent */
public:
    epunct(); // constructor implicit
    epunct(int abs,int ord=0,int c=WHITE);
    // constructor pentru initializari si conversii

    epunct(const epunct&); // constructor de copiere
    epunct& operator = (const epunct&);
    // supraincarca operatorul de atribuire

    friend istream& operator >> (istream&,epunct&);
    // supraincarca operatorul de extragere

    void afiseaza(); // afiseaza punctul curent

    void sterge(); // face invizibil punctul curent

    int retabs() const; // returneaza abscisa punctului curent

    int retord() const; // returneaza ordonata punctului curent

    int retculoare() const; // returneaza codul culorii punctului curent

    int retrvalid() const; // returneaza valoarea lui valid
};

Functiile membru se definesc in fisierul de mai jos, de extensie CPP.

```

## FISIERUL BXXXI5

```

#ifndef __EPUNCT_H
#include "BXXXI5.H"
#define __EPUNCT_H
#endif

void epunct::testecran()
/* atribuie 1 variabilei valid daca x si y definesc un punct in limitele ecranului grafic;
 altfel valid = 0 */
{
    if(x < 0 || x > getmaxx() || y < 0 || y > getmaxy())
        valid = 0;
    else valid = 1;
}

inline epunct::epunct() // constructor implicit
{
    valid = vizibil = 0;
}

epunct::epunct(int abs,int ord,int c)
// constructor pentru initializari si conversii
{
    x = abs; y = ord;
    culoare = c;
    vizibil = 0;
    testecran();
}

epunct::epunct(const epunct& ep) // constructor de copiere
{
    x = ep.x; y = ep.y;
    culoare = ep.culoare;
    vizibil = ep.vizibil;
    valid = ep.valid;
}

epunct& epunct::operator = (const epunct& ep)
// supraincarca operatorul =
{
    if(ep.valid && this != &ep){
        x = ep.x; y = ep.y;
        culoare = ep.culoare;
        vizibil = ep.vizibil;
        valid = ep.valid;
    }
    else
        if(ep.valid == 0) valid = 0;
    return *this;
}

istream& operator >> (istream& intrare,epunct& ep)
// supraincarca operatorul de extragere

```

```

{
    int a,b,c;
    char t[255];
    char er[] = "Nu s-a tastat un intreg\n";
    static istream *pnul = 0;

    for(;;) {
        for(;;) { // citeste abscisa
            cout << "abscisa=";
            if(intrare >> a) break;
            if(intrare.eof()) return *pnul; // s-a intilnit EOF
            cout << er;
            intrare.clear(); // sterge starea de eroare
            intrare.getline(t,255,'\'\n'); // videaza streamul de intrare
        }

        // citeste ordonata
        for(;;) {
            cout << "ordonata=";
            if(intrare >> b) break;
            if(intrare.eof()) return *pnul; // s-a intilnit EOF
            cout << er;
            intrare.clear(); // sterge starea de eroare
            intrare.getline(t,255,'\'\n'); // videaza streamul de intrare
        }

        epunct crt(a,b);
        if(crt.valid) break; // crt este in limitele ecranului
        cout << "coordonatele tastate nu sunt in\
                limitele ecranului\n";
        cout << "se reia citirea coordonatelor\n";
    }

    for(;;) { // citeste codul pentru culoarea punctului
        cout << "culoare=";
        if((intrare >> c) && c >= 0 && c <= 15) break;
        if(intrare.eof()) return *pnul; // s-a intilnit EOF
        intrare.clear(); // sterge starea de eroare
        cout << "se cere intreg in intervalul [0,15]\n";
        intrare.getline(t,255,'\'\n');
    }

    ep.x = a; ep.y = b;
    ep.culoare = c;
    ep.valid = 1;
    ep.vizibil = 0;
    return intrare;
}

inline void epunct::afiseaza() // afiseaza pe ecranul grafic punctul curent
{
    if(valid){
        putpixel(x,y,culoare);
        vizibil = 1;
    }
}

```

```

inline void epunct::sterge() // face invizibil punctul curent
{
    if(valid) putpixel(x,y,getbkcolor());
    vizibil = 0;
}

inline int epunct::retabs() const // returneaza abscisa punctului curent
{
    return x;
}

inline int epunct::retord() const // returneaza ordonata punctului curent
{
    return y;
}

inline int epunct::retculoare() const
// returneaza codul culorii punctului curent
{
    return culoare;
}

inline int epunct::retvalid() const
{
    return valid;
}

```

31.6 Să se implementeze tipul abstract *tabpct* care gestionează, în memoria *heap*, un tablou de obiecte ale clasei *epunct* definită în exercițiul 31.5.

Clasa *tabpct* are următoarele date membru:

- |                |   |
|----------------|---|
| <i>n</i>       | - numărul obiectelor de tip <i>epunct</i> păstrate într-o zonă contiguă din memoria <i>heap</i> ;   |
| <i>tab</i>     | - pointer spre zona din memoria <i>heap</i> în care se păstrează cele <i>n</i> obiecte de tip <i>epunct</i> ;                             |
| <i>vtabpct</i> | - variabilă de tip <i>int</i> care are valoarea 1 dacă cele <i>n</i> obiecte din memoria <i>heap</i> sunt afișate și zero în caz contrar. |

#### FIŞIERUL BXXXI6.H

```

#ifndef __EPUNCT_H
#include "BXXXI5.CPP"
#define __EPUNCT_H
#endif

class tabpct {
protected:
    epunct *tab;
    int n;
    int vtabpct;
public:
    tabpct();

```

```

    tabpct(int nr);
    tabpct(const tabpct&);
    ~tabpct();
    tabpct& operator = (const tabpct&);
    void afiseaza(); // afiseaza punctele din zona spre care pointeaza tab
    void sterge(); // face invizibile punctele din zona spre care pointeaza tab
};

Funcțiile membru se definesc în fișierul de mai jos, de extensie CPP.

```

#### FIŞIERUL BXXXI6

```

#ifndef __TABPCT_H
#include "BXXXI6.H"
#define __TABPCT_H
#endif

tabpct::tabpct()
{
    n=0;
}

tabpct::tabpct(int nr)
{
    vtabpct = 0;
    tab = 0;

    if(nr < 1) {
        cout << "numar puncte = " << nr << " eronat\n";
        n = 0;
    }
    else {
        n = nr;

        // se rezerva zona de memorie în memoria heap
        tab = new epunct[n];

        if(tab == 0) {
            cout << "Memorie insuficienta\n";
            n = 0;
        }
        else{ // se citesc coordonatele punctelor
            for(int i=0;i < n;i++)
                if(!(cin >> tab[i])) break;
            if(i < n){
                cout << "S-au citit " << i <<
                    " puncte in loc de " << n << "\n";
                n = i;
                cin.clear();
            }
        } // sfîrșit else
    } // sfîrșit constructor
}

```

```

tabpct::tabpct(const tabpct& tp)
{
    n = tp.n;
    vtabpct = tp.vtabpct;
    tab = new epunct[n];

    if(tab == 0){
        cout << "Memorie insuficienta\n";
        n = 0;
    }
    else
        for(int i=0; i < n;i++) tab[i] = tp.tab[i];
}

tabpct& tabpct::operator = (const tabpct& tp)
{
    if(this != &tp){
        n = tp.n;
        vtabpct = tp.vtabpct;
        if(n > 0) delete tab;
        tab = new epunct[n];

        if(tab == 0){
            cout << "Memorie insuficienta\n";
            n = 0;
        }
        else
            for(int i=0;i < n;i++) tab[i] = tp.tab[i];
    }
    return *this;
}

inline tabpct::~tabpct()
{
    delete tab;
}

void tabpct::afiseaza()
// afiseaza pe ecranul grafic punctele pastrate in zona spre care pointeaza tab
{
    for(int i=0;i < n;i++) tab[i].afiseaza();
    vtabpct = 1;
}

void tabpct::sterge()
// face invizibil punctele pastrate in zona spre care pointeaza tab
{
    for(int i=0;i < n;i++) tab[i].sterge();
    vtabpct = 0;
}

```

31.7 Să se scrie un program care trasează graficul unei funcții care se definește tabelar.

## PROGRAMUL BXXXI7

```

#include <stdlib.h>
#include <conio.h>
#include "BXXXI6.CPP"
#include "BXXXI3.CPP"

main() /* traseaza graficul unei functii date prin puncte */
{
    int nrpuncte = 0;
    if(citInt("numarul punctelor =",nrpuncte) == 0){
        cout << " s-a tastat EOF\n";
        exit(1);
    }
    if(nrpuncte == 0){
        cout << "numarul de puncte = 0\n";
        exit(1);
    }
    int gd = DETECT,gm;
    initgraph(&gd,&gm,"C:\\borlandc\\bgi");
    tabpct functie(nrpuncte);
    cleardevice();
    functie.afiseaza();
    getch();
    closegraph();
}

```

### Observație:

Pentru a obține un grafic care să aibă un aspect apropiat de cel real, este nevoie să se utilizeze coeficienții de aspect.

- 31.8 Punctele afișate cu ajutorul funcției *putpixel* au o vizibilitate redusă datorită dimensiunii mici. Uneori se preferă afișarea unor puncte de dimensiuni mai mari pentru ca acestea să iasă mai bine în evidență pe ecranul grafic. În acest scop, se poate afișa punctul respectiv, împreună cu un cerc, cu centru în acel punct și de rază trei. În acest exercițiu se completează clasa *epunct*, definită în exercițiul 31.6, cu funcțiile membru *cafiseaza* și *csterge*, care afișeză și respectiv face invizibil un punct de pe ecranul grafic împreună cu cercul cu centru în acel punct și de rază trei.

În cele ce urmează vom numi *c\_punct* imaginea unui punct pe ecranul grafic împreună cu a cercului cu centru în punctul respectiv și de rază trei.

## FIȘIERUL BXXXI8.H

```

#ifndef __EPUNCT_H
#include "BXXXI5.CPP" // clasa epunct
#define __EPUNCT_H
#endif
class cepunct:public epunct {
public:
    void cafiseaza(); // afiseaza obiectul curent ca un c_punct
}

```

```

void csterge(); // face invizibil obiectul curent care este afisat ca un c_punct
cepunct();
cepunct(int abs,int ord,int c=WHITE);
}

```

Funcțiile membru se definesc în fișierul de mai jos, de extensie *CPP*.

### FIŞIERUL BXXXI8

```

#ifndef __CEPUNCT_H
#include "BXXXI8.H"
#define __CEPUNCT_H
#endif

inline cepunct::cepunct():epunct()
{
}

inline cepunct::cepunct(int abs,int ord,int c):epunct(abs,ord,c)
{
}

void cepunct::cafiseaza() // afiseaza obiectul curent ca un c_punct
{
    int i,j;

    if(valid) {
        getaspectratio(&i,&j);
        double f = (double) i/j;
        putpixel(x,f*y,culoare);
        int temp = getcolor();
        setcolor(culoare);
        circle(x,f*y,3); // traseaza un cerc de raza 3
        setcolor(temp);
        vizibil = 1;
    }
}

void cepunct::csterge()
// face invizibil obiectul curent care este considerat ca este afisat ca un c_punct
{
    if(valid) {
        int temp = culoare;
        culoare = getbkcolor();
        cafiseaza();
        culoare = temp;
        vizibil = 0;
    }
}

```

#### Observații:

1. Funcția *cafiseaza* utilizează coeficienții de aspect pentru a obține imagini pe ecranul grafic apropiate de cele reale.

2. Acest exemplu ilustrează avantajul conceptului de moștenire: modificările se obțin simplu prin adăugări de facilități noi, nu prin schimbarea și reprogramarea conceptelor existente.

Moștenirea este un concept care încurajează programarea prin componente standardizate care se bucură de o mare extensibilitate.

Se pot crea biblioteci de componente standardizate care se pot utiliza în programe ca atare. Dacă sunt necesare modificări, atunci acestea se fac nu modificând componentele respective, ci prin extensia lor, adăugindu-se facilități noi pe baza conceptului de moștenire. În felul acesta se mărește eficiența în programare pe baza reutilizării simple a componentelor deja existente și testate.

- 3.1.9 Să se extindă tipul abstract *tabpct*, implementat în exercițiul 31.6, cu funcții pentru afișarea și stergerea *c\_punctelor* (pentru noțiunea de *c\_punct* vezi exercițiul 31.8).

### FIŞIERUL BXXXI9.H

```

#ifndef __CEPUNCT_H
#include "BXXXI8.CPP" // clasa cepunct
#define __CEPUNCT_H
#endif
#ifndef __TABPCT_H
#include "BXXXI6.CPP" // clasa tabpct
#define __TABPCT_H
#endif

class ctabpct:public tabpct {
protected:
    cepunct *ctab;
public:
    ctabpct();
    ctabpct(int nr);
    void cafiseaza();
    // afiseaza punctele din zona spre care pointeaza ctab;
    // punctele se afisaza ca niste c_puncte

    void csterge();
    // face invizibile punctele din zona spre care pointeaza ctab;
    // se consideră ca punctele respective sunt afisate ca niste c_puncte
};

```

Mai jos, se definesc funcțiile membru într-un fișier de extensie *CPP*.

### FIŞIERUL BXXXI9

```

#ifndef __CTABPCT_H
#include "BXXXI9.H"
#define __CTABPCT_H
#endif

inline ctabpct::ctabpct()
{

```

```

n=0;
}

inline ctabpct::ctabpct(int nr):tabpct(nr)
{
    (epunct *)ctab = tab;
}

void ctabpct::cafiseaza()
/* afiseaza punctele din zona spre care pointeaza ctab;
   punctele se afiseaza ca niste c_puncte */
{
    for(int i = 0;i < n;i++) ctab[i].cafiseaza();
    vtabpct = 1;
}

void ctabpct::csterge()
/* face invizibile punctele din zona spre care pointeaza ctab;
   se considera ca punctele respective sunt afisate ca niste c_puncte */
{
    for(int i = 0;i < n;i++) ctab[i].csterge();
    vtabpct = 0;
}

```

31.10 Să se implementeze tipul abstract *graficpct* pentru trasarea graficelor funcțiilor date prin puncte.

Prin graficul unei funcții dată prin puncte vom înțelege imaginea de pe ecranul grafic compusă din:

- imaginile punctelor date sub formă de *c\_puncte*;
- linii drepte care unesc punctele consecutive date ale funcției.

Liniile au caracteristicile standard (SOLID\_LINE, NORM\_WIDTH), iar culoarea coincide cu a primului punct dintre cele două puncte consecutive.

## FIŞIERUL BXXXI10.H

```

#ifndef __CTABPCT_H
#include "BXXXI9.CPP"
#define __CTABPCT_H
#endif

class graficpct:public ctabpct {
public:
    graficpct();
    graficpct(int nr);
    void afisgrafic(); // afiseaza graficul prin puncte
    void sterggrafic(); // face invizibil graficul prin puncte
};

```

Funcțiile membru se definesc în fișierul de mai jos, de extensie *CPP*.

## FIŞIERUL BXXXI10

```

#ifndef __GRAFICPCT_H
#include "BXXXI10.H"

```

```

#define __GRAFICPCT_H
#endif

inline graficpct::graficpct():ctabpct()
{
}

inline graficpct::graficpct(int nr):ctabpct(nr)
{
}

void graficpct::afisgrafic() // afiseaza graficul prin puncte
{
    // se afiseaza punctele
    ctabpct::cafiseaza();

    // se unesc punctele consecutive prin segmente de dreapta
    struct linesettingstype temp;
    getlinesettings(&temp);
    int c = getcolor();
    setlinestyle(SOLID_LINE,0,NORM_WIDTH);
    int i,j;
    getaspectratio(&i,&j);
    double f=(double)i/j;

    for(i=0;i < n-1;i++) {
        setcolor(ctab[i].retculoare());
        line(ctab[i].retabs(),f*ctab[i].retord(),
             ctab[i+1].retabs(),f*ctab[i+1].retord());
    }
    setcolor(c);
    setlinestyle(temp.linestyle,temp.upattern,temp.thickness);
}

void graficpct::sterggrafic() /* face invizibil graficul prin puncte */
{
    // se sterg punctele
    ctabpct::csterge();

    // se sterg segmentele de dreapta care unesc punctele consecutive
    struct linesettingstype temp;
    getlinesettings(&temp);
    int c = getcolor();
    setlinestyle(SOLID_LINE,0,NORM_WIDTH);
    setcolor(getbkcolor());
    int i,j;
    getaspectratio(&i,&j);
    double f = (double)i/j;
    for(i=0;i < n-1;i++)
        line(ctab[i].retabs(),f*ctab[i].retord(),
              ctab[i+1].retabs(),f*ctab[i+1].retord());
    setcolor(c);
    setlinestyle(temp.linestyle,temp.upattern,temp.thickness);
}

```

31.11. Să se scrie un program care trasează graficul unei funcții care se definește prin puncte.

Acest program este similar cu cel din exercițiul 31.7. Spre deosebire de acesta, programul de față afișează punctele date ca niște c\_puncte, iar punctele consecutive sunt unite prin segmente de dreaptă.

### PROGRAMUL BXXXI11

```
#include <conio.h>
#include <stdlib.h>
#include "BXXXI10.CPP"
#include "BXXXI3.CPP"
main()
/* traseaza graficul unei functii data prin puncte:
   - punctele se afisaza ca niste c_puncte;
   - punctele consecutive se unesc prin segmente de dreapta.
*/
{
    int nrpuncte = 0;

    if(citInt("numarul punctelor = ",nrpuncte) == 0){
        cout << "S-a tastat EOF\n";
        exit(1);
    }
    if(nrpuncte == 0){
        cout << "Numarul de puncte = 0\n";
        exit(1);
    }
    int gd = DETECT,gm;
    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    graficpct functie(nrpuncte);
    cleardevice();
    functie.afisgrafic();
    getch();
    closegraph();
}
```

#### Observație:

Propunem cititorului să execute programul de mai sus pentru funcția dată prin punctele din tabelul de mai jos.

numărul punctelor = 11

Număr current	Abscisă	Ordonată	Culoare
1	100	60	1
2	200	60	2
3	100	200	3
4	200	200	4

(continuare)

Număr current	Abscisă	Ordonată	Culoare
5	100	60	5
6	100	200	6
7	150	250	7
8	200	200	8
9	200	60	9
10	150	10	10
11	100	60	11

31.12 Tipul abstract *graficpct* permite trasarea pe ecranul grafic a graficelor funcțiilor definite prin puncte. Abscisa și valoarea funcției, precum și culoarea punctului sunt date care se citesc de la intrarea standard. Adesea, coordonatele punctelor rezultă în urma unor calcule. Se pot defini tablouri de tip *epunct* sau *cepunct* ale căror elemente se determină prin calcul.

Pentru a trasa graficul unei funcții definite în acest fel, vom extinde tipul abstract *graficpct*, adăugind un constructor cu doi parametri:

- nr*: - numărul elementelor de tip *epunct* pentru care se trasează graficul;  
*te*: - tabloul de tip *epunct* ale cărui elemente definesc coordonatele celor *nr* puncte ale graficului funcției, precum și culoarea acestora.

Numim *tgrafic* tipul abstract care permite trasarea graficelor funcțiilor fie citind datele de la intrarea standard, fie folosind un tablou de tip *epunct* ale cărui elemente se determină prin calcule.

Clasa *tgrafic* este o clasă derivată a clasei *graficpct*.

Ea are trei constructori:

- constructorul implicit;
- constructor cu un singur parametru, utilizat în cazul cînd datele se citesc de la intrarea standard;
- constructorul cu doi parametri, amintit mai sus.

### FIȘIERUL BXXXI12.H

```
#ifndef __GRAFICPCT_H
#include "BXXXI10.CPP"
#define __GRAFICPCT_H
#endif

class tgrafic:public graficpct{
public:
```

```

tgrafic();
tgrafic(int nr);
tgrafic(int nr,epunct *te);
};

```

### Observație:

Funcțiile membru *afisgrafic* și *stergegrafic* pot fi apelate pentru obiecte ale clasei *tgrafic* deoarece ele sunt moștenite de această clasă.

Constructorii clasei *tgrafic* se definesc în fișierul de mai jos, de extensie *CPP*.

### FIŞIERUL BXXXI12

```

#ifndef __TGRAFIC_H
#include "BXXXI12.H"
#define __TGRAFIC_H
#endif

inline tgrafic::tgrafic():graficpct()
{
}

inline tgrafic::tgrafic(int nr):graficpct(nr)
{
}

tgrafic::tgrafic(int nr,epunct *te)
{
    if(nr <= 0){
        cout << "Numar puncte eronat\n";
        n = 0;
    }
    else{
        n = nr;
        tab = new epunct[nr];
        if(tab == 0){
            cout << "memorie insuficienta\n";
            n = 0;
        }
        else{
            int i;
            for(i=0;i < n;i++) tab[i]=te[i];
            vtabpct=0;
            (epunct *)ctab=tab;
        }
    }
}

```

- 31.13 În ultimul timp se încearcă definirea evoluției "probabile" a unei persoane folosind diferite metode matematice mai mult sau mai puțin sofisticate. O metodă foarte simplă pretinde că definește ascensiunile sau decăderile probabile ale unei persoane în primii 70 de ani de viață folosind date nașterii persoanei respective.

Evident, aceste tendințe pot fi puternic influențate de condițiile concrete ale vieții fiecărui și o ascensiune rezultată din metoda pe care o indică mai jos, poate să nu se adeverescă.

Tendințele de ascensiune și coborire se vor afișa pe ecranul grafic.

Ca date de intrare se dau ziua, luna și anul nașterii persoanei în cauză.

Ziua și luna se concatenează împreună pentru a forma un întreg de 4 cifre.

De exemplu, 15 octombrie (luna a 10-a) formează întregul 1510, 25 septembrie (luna a 9-a), formează întregul 2509.

În cazul în care ziua aparține intervalului [1,9], aceasta se inmulțește cu 10. Deçi, 4 octombrie formează întregul 4010, 4 septembrie formează întregul 4009 etc.

Numărul obținut în acest fel se inmulțește cu anul nașterii care și el este un întreg de 4 cifre. Primele 7 cifre ale produsului obținut în acest fel, reprezintă puncte de ascensiune și descreștere din viața persoanei respective. Acestea aparțin intervalului [0,9] și ele reprezintă puncte pe ordonată (valorile funcției pentru care se trasează graficul). Pe axa absciselor se reprezintă anii de viață din 10 în 10.

Prima cifră a produsului corespunde abscisei de valoare 10, a două cifră, abscisei de valoare 20 și aşa mai departe.

Punctele reprezentate în acest fel se unesc prin segmente de dreaptă.

Conform acestei metode, persoanele născute în zilele de 1, 2 și 3 ale unei luni calendaristice vor avea același grafic cu cele născute în zilele de 10, 20 și respectiv 30 ale aceleiași luni calendaristice.

Data calendaristică se citește apelind funcția *pcit\_data\_calend*, definită în exercițiul 8.4. Aceasta apelează funcțiile definite în exercițiile 8.2 și 8.3. Data calendaristică se validează de către funcția *pcit\_data\_calend* care apelează în acest scop funcția *v\_calend*, definită în exercițiul 6.5. Această funcție folosește tabloul global *nrzile* care definește numărul de zile din lunile calendaristice.

### PROGRAMUL BXXXI13

```

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

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

#include "BVI5.CPP"
#include "BVIII12.CPP"
#include "BVIII13.CPP"
#include "BVIII14.CPP"
#include "BXXXI12.CPP"

main() /* graficul tendințelor evoluțiilor probabile ale unei persoane */
{
    int zi,luna,an;

```

```

if(pcit_data_calend(&zi,&luna,&an) == 0){
    cout << "s-a tastat EOF\n";
    exit(1);
}
char zona[5]; /* zona in care se concateneaza ziua cu luna calendaristica */
char cluna[3]; // luna pe caractere
itoa(luna,cluna,10);

if(zi < 10) // zi in intervalul [1,9]
    zi *= 10;
itoa(zi,zona,10);

if(luna < 10) /* daca luna este in intervalul [1,9], se concateneaza un zero dupa zi */
    strcat(zona,"0");
strcat(zona,cluna);
long zilunan = atol(zona);

/* zilunan are ca valoare intregul format din concatenarea zilei cu luna calendaristica */
zilunan *= an; /* produsul dintre anul nasterii si intregul format din concatenarea
zilei cu luna; este un intreg de cel mult 8 cifre */
char zonacif[9];
ltoa(zilunan,zonacif,10);

/*zonacif contine cifrele care reprezinta ordonatice functiei de reprezentat pe ecranul grafic */
int gd = DETECT,gm;
initgraph(&gd,&gm,"c:\\borlandc\\bgi");
int ax,ay; //coeficientii de aspect
getaspectratio(&ax,&ay);
double f=(double)ax/ay;

// afisaza axele de coordonate
line(40,40*f,40,330*f); // axa ordonatelor
line(20,310*f,530,310*f); // axa absciselor
int crtcul=getcolor(); // salveaza culoarea curenta
setcolor(YELLOW); // defineste culoarea pentru diviziuni

// traseaza diviziunile pe axa ordonatelor
for(int y=1; y < 10;y++){
    char cif[2];
    itoa(y,cif,10);
    int z = 310-30*y;

    // afisaza ordonata
    outtextxy(20,z*f,cif);
    /* prin punctul de divizuire se duc paralela cu axa absciselor */
    line(30,z*f,530,z*f);
}

// traseaza diviziunile pe axa absciselor
for(int x = 10;x < 80;x += 10){
    char ab[3];
    itoa(x,ab,10);
    int z = 30+5*x;
}

```

```

// afiseaza abscisa
outtextxy(z,340*f,ab);

// traseaza diviziunea
line(z+4,310*f,z+4,330*f);

// refac culoarea curenta
setcolor(crtcul);
cepunct hor[7]; // tablou de 7 elemente pentru pastrarea punctelor functiei de trasat
int i;

// se definesc punctele functiei
for(x=10,i=0;x <= 70;x += 10,i++)
    hor[i] = cepunct(34+5*x,
                     (310-30*(zonacif[i]-'0')),LIGHTRED);

// se instantiaza un obiect de tip tgrafic care se initializaaza cu elementele tabloului hor
tgrafic functie(7,hor);

// se traseaza graficul functiei
functie.afisgrafic();
getch();
closegraph();
}

```

## 31.2. Formatarea în memorie

Aşa cum s-a amintit la inceputul capitolului de faţă, din clasa *streambuf* derivă clasa *strstreambuf*. Obiectele acestei clase se folosesc la realizarea operaţiilor de formatare în memorie. Ele sunt utilizate în clase ale ierarhiei de clase care are ca rădăcină clasa *ios*. Astfel, din clasa *ios* derivă clasa *strstreambase*, iar din aceasta derivă alte trei clase: *istrstream*, *ostrstream* și *strstream*. Aceste trei clase mai au, fiecare, încă o clasă de bază și anume:

- clasa *istrstream* derivă din clasa *istream*,
- clasa *ostrstream* derivă din clasa *ostream*,
- clasa *strstream* derivă din clasa *iostream*.

Clasa *ostrstream* se utilizează pentru a realiza conversii din format intern în siruri de caractere care se păstrează în memorie. Aceste conversii sunt analoge cu cele realizate cu ajutorul funcției *sprintf*.

Clasa *ostrstream* are doi constructori de prototip:

```

ostrstream();
și
ostrstream(char *zona,int n);
unde:
zona

```

- Definește adresa zonei de memorie în care se păstrează sirul de caractere rezultat în urma conversiilor datelor din

- formatul intern.
- Definește dimensiunea zonei de memorie spre care pointează *zona*.

În cazul constructorului fără parametri, se definește în mod automat zona de memorie receptoare.

Adresa acestei zone de memorie poate fi cunoscută apelind funcția membru *str* de prototip:

```
char *str();
```

Ea returnează adresa zonei tampon alocată în mod automat la utilizarea unui obiect instantiat fără parametri:

```
ostrstream obiect;
```

Dimensiunea zonei de memorie alocate se obține prin apelul funcției membru *pcount* a clasei *ostrstream* de prototip:

```
int pcount();
```

Clasa *ostrstream*, fiind derivată din clasa *ostream*, moștenește supraincărările operatorului de *inserare* (*<<*). Acestea permit ca operatorul de inserare să poată fi utilizat cu instanțieri ale clasei *ostrstream*.

#### Exemplu:

```
ostrstream obiect;
int i;
...
obiect << i << '\0';
```

Zona de memorie receptoare se alocă automat la execuția acestei instrucțiuni expresie. De aceea, în continuare se poate apela funcția *str* pentru a determina adresa zonei de memorie în care se păstrează caracterele rezultate prin conversia lui *i* din binar în zecimal:

```
char *zona = obiect.str();
```

În continuare avem acces la caracterele din zona receptoare prin intermediul pointerului *zona*.

De exemplu, instrucțiunea:

```
cout << zona;
```

afisează la terminal valoarea lui *i* convertită în zecimal.

Zona de memorie alocată în acest fel trebuie să fie eliberată de utilizator folosind operatorul *delete*:

```
delete zona;
```

Clasa *istrstream* permite realizarea de conversii inverse față de clasa *ostrstream*. Aceste conversii sunt similare cu cele obținute cu ajutorul funcției *sscanf*.

Clasa *istrstream* are doi constructori de prototip:

```
istrstream(char *zona);
istrstream(char *zona,int n);
```

unde:

*zona* - Este pointer spre începutul zonei de memorie care conține sirul de caractere supus conversiilor.

*n* - Este dimensiunea zonei de memorie spre începutul căreia pointează zona.

Clasa *istrstream* este derivată din clasa *istream* și de aceea ea moștenește, printre altele, supraincărările operatorului de *extrageré* (*>>*). În felul acesta, operatorul de extragere se poate folosi cu obiecte ale clasei *istrstream*.

Clasele amintite mai sus, implică includerea fișierului *strstream.h*.

#### Exemplu:

Secvența de mai jos, utilizată la citirea unui întreg de tip *long* de la intrarea standard:

```
...
for(;;){
    char tab[255];
    printf(text);
    if(gets(tab) == 0) exit(1);
    if(sscanf(tab,"%ld",&x) == 1) break;
    printf("Nu s-a tastat un intreg\n");
    printf("Se reia citirea intregului\n");
}
```

se poate scrie ca mai jos, înlocuind funcțiile *gets*, *sscanf* și *printf* cu funcții membru și operatori supraincarcați în clasele definite în fișierul *strstream.h*.

```
...
for(;;){
    char tab[255];
    cout << text;
    if(cin.getline(tab,255,'\'\n') == 0) exit(1); // stare de eroare (EOF)
    istrstream obiect(tab);
    if(obiect >> x) break;
    cout << "Nu s-a tastat un intreg\n";
    cout << "Se reia citirea intregului\n";
}
```

### 31.3. Prelucrarea fișierelor

În cele ce urmează avem în vedere operații de prelucrare a fișierelor înregistrate pe discuri.

La prelucrarea fișierelor se folosesc obiecte ale clasei *filebuf* care este o clasă derivată a clasei *streambuf*.

Obiectele clasei *filebuf* sunt utilizate de clase specifice pentru operații cu fișiere și care aparțin ierarhiei care are ca rădăcină clasa *ios*. În această ierarhie este definită clasa *fstreambase* care este o clasă derivată a clasei *ios*.

Clasa *fstreambase* este o clasă de bază pentru următoarele 3 clase:

- |                 |  |
|-----------------|--|
| <i>ifstream</i> | - folosită la operații de intrare (citire);                |
| <i>ofstream</i> | - folosită la operații de ieșire (scriere);                |
| <i>fstream</i>  | - folosită la operații de intrare/ieșire (citire/scriere). |

ACESTE TREI CLASE, AU FIECARE, ÎNCĂ O CLASĂ DE BAZĂ ȘI ANUME:

- |                 |                                      |
|-----------------|--------------------------------------|
| <i>ifstream</i> | - derivă din clasa <i>istream</i> ;  |
| <i>ofstream</i> | - derivă din clasa <i>ostream</i> ;  |
| <i>fstream</i>  | - derivă din clasa <i>iostream</i> . |

De aici rezultă că pentru obiectele acestor clase se pot aplica funcțiile membru ale claselor lor de bază, inclusiv supraincărările operatorilor de inserare și extragere.

Prelucrarea unui fișier de pe disc începe cu *deschiderea* lui. În acest scop se poate proceda în unul din următoarele moduri:

1. Se utilizează funcția membru *open* cu un obiect al uneia din clasele *ifstream*, *ofstream* sau *fstream* instanțiat fără parametri.
2. Se utilizează parametri pentru deschiderea fișierului la instanțierea obiectelor.

În primul caz, obiectele se instanțiază folosind constructori implicați.

Fie, de exemplu, instanțierea:

```
ifstream obiect;
```

Obiectului *obiect* își atașează un fișier concret la un apel al funcției *open* de forma:

```
obiect.open(...)
```

Funcția *open* este definită în clasa *fstreambase* și este supraincărtă în clasele derivate din aceasta. Ea are prototipul:

```
void open(char *fisier,int mod,int protectie);
```

Parametrul *fisier* este un pointer spre un sir de caractere care definește numele și extensia fișierului care se deschide, precum și calea spre fișierul respectiv, dacă acesta nu se află în directorul curent.

Parametrul *mod* definește modul de deschidere al fișierului. În acest scop se poate folosi enumeratori definiți în clasa *ios* astfel:

```
class ios {  
public:  
    enum open_mode {  
        in=0x01,out=0x02,ate=0x04,app=0x08,trunc=0x10,  
        norecreate=0x20,noreplace=0x40,binary=0x80  
    };  
};
```

Acești enumeratori au următoarele semnificații:

- |                     |  |
|---------------------|--|
| <i>in</i> :         | - fișierul se deschide în intrare (citire);  |
| <i>out</i> :        | - fișierul se deschide în ieșire (scriere);  |
| <i>ate</i> :        | - după deschiderea fișierului, poziția curentă în fișier este sfîrșitul fișierului; acest mod se utilizează pentru a face căutări în fișier începând cu sfîrșitul acestuia;  |
| <i>app</i> :        | - deschidere pentru a adăuga înregistrări la sfîrșitul fișierului;   |
| <i>trunc</i> :      | - dacă fișierul există, conținutul lui se pierde și se creează un fișier nou; acest mod este implicit cind este prezentă opțiunea <i>out</i> (deschidere în scriere); el este incompatibil cu opțiunile <i>ate</i> și <i>app</i> ; |
| <i>norecreate</i> : | - fișierul care se deschide cu această opțiune trebuie să existe; el nu se deschide în creare;   |
| <i>noreplace</i> :  | - dacă fișierul există și este prezentă opțiunea <i>out</i> , atunci deschiderea este admisă numai dacă se utilizează opțiunea <i>ate</i> sau <i>app</i> ;   |
| <i>binary</i> :     | - fișierul se deschide pentru a fi prelucrat binar; în mod implicit fișierele se consideră că sunt prelucrate pe caracter (mod text).  |

La o deschidere de fișier se pot folosi împreună mai multe opțiuni, folosind operatorul "!" (sau logic pe biți).

Pentru obiectele clasei *ifstream*, opțiunea *in* este implicită, iar pentru cele ale clasei *ofstream*, opțiunea *out* este implicită.

Parametrul *protectie* definește modul de acces la fișier. În mod implicit acest parametru are valoarea zero și aceasta înseamnă că fișierul respectiv nu are restricții la acces.

Rezultatul operației de deschidere de fișiere poate fi testat cu ajutorul cuvintului de stare al streamului pentru care s-a apelat funcția *open*.

**Exemple:**

1. 

```
ifstream fisier;  
// se deschide fișierul fis.dat în citire  
fisier.open("fis.dat",ios::norecreate);  
  
// se testează stareea obiectului fisier după deschidere  
if(fisier){
```

```

// deschidere corecta
...
}
else{
...
cout << "incident la deschiderea lui fis.dat\n";
...
}

Optiunea ios::in este implicită.

2. ofstream fis_iesire;
// se deschide în scriere fișierul factura.dat
fis_iesire.open("factura.dat");

if(!fis_iesire){
    cout << "incident la deschiderea lui factura.dat\n";
    ...
}

3. ofstream fis;
// se deschide pentru adaugare fișierul binar stoc.dat
fis.open("stoc.dat",ios::app | ios::binary);

// se testează starea obiectului fis după deschidere
if(!fis){

    cout << "incident la deschiderea lui stoc.dat\n";
    ...
}
else{
...
// deschidere corecta
...
}

```

Cea de a doua posibilitate de a deschide un fișier este aceea de a folosi parametrii corespunzători la instanțierea obiectelor claselor *ifstream*, *ofstream* sau *fstream*. În acest caz, constructorul apelat pentru instanțierea obiectului apelează în mod automat funcția *open*.

Constructorii care deschid fișierele la instanțierea obiectelor claselor amintite mai sus, au aceeași parametru ca și funcția *open*:

*nume\_clasa(char \*fisier,int mod,int protectie);*

unde:

*nume\_clasa* - Este *ifstream*, *ofstream* sau *fstream*.

Parametrul *protectie* este și în acest caz un parametru implicit, avind valoarea implicită egală cu *zero*.

Exemplile de mai sus pot fi rescrise astfel:

1. ifstream fisier("fis.dat",ios::nocreate);
 if(fisier){

```

// deschidere corecta
...
}
else{
...
cout << "incident la deschiderea lui fis.dat\n";
...
}

2. ofstream fis_iesire("factura.dat");
if(!fis_iesire){
    cout << "incident la deschiderea lui factura.dat\n";
    ...
}

3. ofstream fis("stoc.dat",ios::app | ios::binary);
if(!fis){
...
cout << "incident la deschiderea lui stoc.dat\n";
...
}
else{
...
// deschidere corecta
...
}
```

După deschiderea unui fișier se pot face prelucrări asupra fișierului respectiv.

După terminarea prelucrării unui fișier, în conformitate cu modul în care a fost deschis, acesta trebuie inchis. Închiderea unui fișier se realizează cu ajutorul funcției *close*. Aceasta se apelează în mod obișnuit:

*obiect.close();*

sau

*pobiect -> close();*

unde:

*obiect* - Este o instanțiere a uneia din clasele *ifstream*, *ofstream* sau *fstream*.

În mod analog, *pobiect* este un pointer spre unul din tipurile implementate prin aceste clase.

La prelucrarea fișierelor păstrate pe disc se pot folosi funcțiile membru ale claselor *istream*, *ostream* și *iostream*, acestea fiind clase de bază pentru clasele specifice prelucrării fișierelor de pe disc. În particular, se pot folosi operatorii de *inserare* și *extragere* pentru operații de scriere cu format și respectiv citire cu format dintr-un fișier de pe disc.

Acești operatori, precum și funcțiile membru *get*, *getline*, *put*, *read* și *write* amintite în paragraful 31.1, pot fi folosite pentru a realiza operații cu fișiere pe disc. Cu ajutorul acestora se realizează operații de intrare/ieșire cu acces

secvențial.

Accesul aleator se poate realiza folosind funcții membru specifice ale claselor *ostream* și *istream*.

Astfel, clasa *ostream* are funcțiile membru *seekp* și *tellp* care pot fi utilizate la realizarea accesului aleator în scrierea înregistrărilor în fișiere.

În mod analog, clasa *istream* are funcțiile membru *seekg* și *tellg* care pot fi utilizate la realizarea accesului aleator la citirea înregistrărilor din fișiere.

Funcția *tellp* returnează poziția curentă în streamul curent față de începutul lui. Are prototipul:

**long tellp();**

Funcția *seekp* realizează poziționarea în *streamul* curent.

Aceasta are prototipul:

**ostream& seekp(long n,seek\_dir rel= beg);**

unde:

*seek\_dir*

- Se definește în clasa *ios* astfel:  
**enum seek\_dir{beg,cur,end};**

Acești parametri se interpretează astfel:

*n:*

- deplasament în număr de octeți;

*rel:*

- *beg*: deplasamentul *n*, se consideră față de începutul streamului;

- *cur*: deplasamentul *n*, se consideră față de poziția curentă din stream;

- *end*: deplasamentul *n*, se consideră față de sfîrșitul streamului.

Funcțiile membru ale clasei *istream* au prototipuri și semnificații similare:

**long tellg();**

și

**istream& seekg(long n,seek\_dir rel=beg);**

Menționăm că funcțiile *tellp*, *tellg*, *seekp* și *seekg* se pot utiliza și în legătură cu obiectele care sunt instanțieri ale claselor *istrstream*, *ostrstream* și *strstream*.

O altă funcție membru utilă este:

**gcount**

funcție membru a clasei *istream*;

Ea are prototipul:

**int gcount();**

Funcția *gcount* returnează numărul de octeți citiți la ultima operație de intrare.

## Exerciții:

31.14 Să se scrie un program care copiază intrarea standard la ieșirea standard.

Acest program a mai fost realizat în exercițiile 16.1 și 16.4. În exercițiul 16.1 s-au folosit funcțiile *read* și *write* din biblioteca limbajului C și care au prototipurile în fișierul *io.h*. Ele sunt funcții de prelucrare, de nivel inferior, ale fișierelor.

În exercițiul 16.4 se utilizează funcțiile *getc* și *putc* care au prototipurile în fișierul *stdio.h*.

În acest exercițiu se utilizează două variante și anume:

- utilizarea funcțiilor *get* și *put*, prima fiind funcție membru a clasei *istream*, iar cealaltă fiind funcție membru a clasei *ostream* (această variantă este similară cu cea din exercițiul 16.4);
- utilizarea operatorilor de *extragere* și *inserare*.

## PROGRAMUL BXXXI14

```
#include <iostream.h>

main()
/* copiază intrarea standard la ieșirea standard folosind funcțiile membru get și put */
{
    char c;
    while(cin.get(c)) cout.put(c);
}
```

## Observație:

Programul se termină la întlnirea sfîrșitului de fișier. În acest moment, streamul *cin* intră în stare de eroare și referința la el, returnată de funcția *get*, este zero. În felul acesta, ciclul *while* se intrerupe și odată cu el se intrerupe și execuția programului.

## PROGRAMUL BXXXI14A

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

main()
/* copiază intrarea standard la ieșirea standard folosind operatorii de extragere și inserare */
{
    char c;
    while(cin >> resetiosflags(ios::skipws) >> c) cout << c;
}
```

## Observații:

1. Programul se termină la întlnirea sfîrșitului de fișier. În acest moment, streamul *cin* intră în stare de eroare. Operatorul *>>* este supraincarcat în aşa fel încât să returneze o referință la *streamul* la care se aplică. Această referință

- este egală cu zero dacă *streamul* respectiv se află în stare de eroare.
- Acste programe pot fi folosite pentru a copia fișiere de tip text de pe disc sau pe disc, precum și pe imprimantă. În acest scop se vor face redirectări corespunzătoare.

**Exemple:**

- Afișarea pe display a conținutului fișierului *fis.dat* de pe disc:  
>BXXXI14A < fis.dat
- Afișarea pe imprimanta paralelă a conținutului fișierului *fis.dat* de pe disc:  
>BXXXI14A < fis.dat > prn

- 31.15 Să se scrie un program care citește de la intrarea standard datele ale căror formate sunt definite mai jos și le scrie în fișierul *misc.dat* din directorul curent.

Fișierul este organizat binar, fiecare articol conținând următoarele date:

tip	denumire	um	cod	pret	cantitate
I	REZISTENTA	01OKO	123456789	10	10000

Acest program este similar cu cel definit în exercițiul 16.6.

### PROGRAMUL BXXXI15

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#define MAX 50

typedef struct {
    char tip[2];
    char den[MAX+1];
    int val;
    char unit[3];
    long cod;
    float pret;
    float cant;
} ARTICOL;

/* 6 articole în zona tampon */
union {
    ARTICOL a[6];
    char zt[6*sizeof(ARTICOL)];
} buf;

int citart(ARTICOL *str)
/* - citește datele de la intrarea standard și le păstrează în zona spre care pointează str;
   - returnează valoarea zero la întâlnirea sfîrșitului de fișier și unu în caz contrar. */
{
    char t[255];
    float x,y;
```

```
for(;;) {
    cout << "tip den um cod pret cant\n";
    if(cin >> setw(2)>>str-> tip >> setw(50) >> str-> den
       >> str-> val >> setw(3) >> str-> unit
       >> str-> cod >> x >> y){
        str-> pret = x; str-> cant = y;
        return 1;
    }

    // streamul este în stare de eroare
    if(cin.eof()) return 0; // EOF

    // nu este sfârșit de fișier
    // se anulează starea de eroare
    cin.clear();

    // se avansază peste eventualele caractere existente după eroare
    cin.getline(t,255,'\'n');
    cout << "Date eronate; se reia citirea\n";
}

// sfârșit for
} // sfârșit funcție citart

main()
/* creează fișierul misc.dat cu datele citite de la intrarea standard */
{
    // se deschide misc.dat în creare binara
    ofstream misc("misc.dat",ios::binary);
    int n;
    if(!misc){
        cout << "Nu se poate deschide fișierul \
                  misc.dat în creare\n";
        exit(1);
    }
    // se creează fișierul misc.dat
    for(;;) {
        // se umple zona tampon a fișierului
        for(int i=0;i < 6;i++)
            if((n=citart(&buf.a[i])) == 0) break; // s-a întâlnit EOF
        // se scrie zona tampon dacă nu este văzuta
        if(i){ // zona tampon nu este văzuta
            misc.write(buf.zt,i*sizeof(ARTICOL));
            if(!misc){ // eroare la scrierea în fișier
                cout << "Eroare la scrierea în fișierul\
                          misc.dat\n";
                exit(1);
            }
        }
        if(n == 0) break; // s-a întâlnit EOF
    } // sfârșit for
    misc.close();
} // sfârșit main
```

- 31.16 Să se scrie un program care afișează articolele păstrate în fișierul *misc.dat*.

creat în exercițiul precedent, pentru tip = I.

Amintim că *tip* este o literă:

- I - intrări;
- E - ieșiri;
- T - transfer etc.

## PROGRAMUL BXXXI16

```
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#define MAX 50

typedef struct {
    char tip[2];
    char den[MAX+1];
    int val;
    char unit[3];
    long cod;
    float pret;
    float cant;
} ARTICOL;

union {
    ARTICOL a[6];
    char zt[6*sizeof(ARTICOL)];
}buf;

main()
/* citeste si afisaza articolele din fisierul misc.dat pentru tip = I */
{
    ifstream misc("misc.dat", ios::binary);
    int n;

    if(!misc) {
        cout << "Eroare la deschiderea fisierului\\n";
        misc.dat in citire\\n";
        exit(1);
    }

    // se citeste fisierul misc.dat si se listeaza articolele pentru tip = I
    cout << "\\n\\n\\n" << setw(30) << "INTRARI" << "\\n\\n\\n";
    do {
        misc.read(buf.zt, 6*sizeof(ARTICOL));

        // calculeaza numarul articolelor citite
        n = misc.gcount()/sizeof(ARTICOL);
        for(int i=0; i < n; i++)
            if(buf.a[i].tip[0] == 'I'){
                cout.setf(ios::left,ios::adjustfield);
                cout << setw(50) << buf.a[i].den;
                cout.setf(ios::right,ios::adjustfield);
                cout << setw(3) << setfill('0') << buf.a[i].val;
                cout.fill(' ');
                cout << buf.a[i].unit;
                cout << " " << setw(9) << setfill('0')
                    << buf.a[i].cod;
                cout.fill(' ');
                cout << " " << buf.a[i].pret;
                cout << " " << buf.a[i].cant << "\\n";
            } // sfirsit for
    } while(misc);
    if(!misc.eof())
        cout << "incident la citirea din fisierul misc.dat\\n";
    misc.close();
}
```

```
cout.fill(' ');
cout << buf.a[i].unit;
cout << " " << setw(9) << setfill('0')
    << buf.a[i].cod;
cout.fill(' ');
cout << " " << buf.a[i].pret;
cout << " " << buf.a[i].cant << "\\n";
} // sfirsit for
} while(misc);
if(!misc.eof())
    cout << "incident la citirea din fisierul misc.dat\\n";
misc.close();
}
```

31.17 Să se scrie un program care citește siruri de caractere de la intrarea standard și le transferă într-un fișier pe disc.  
Numele fișierului este *textmen.txt*.

## PROGRAMUL BXXXI17

```
#include <fstream.h>
#include <stdlib.h>

main()
/* - citeste texte de la intrarea standard si le pastreaza pe disc in fisierul textmen.txt;
   - programul se termina la infinalarea sfirisitului de fisier.*/
{
    // se deschide fisierul textmen.txt in creare

    ofstream textmen("textmen.txt");
    char buf[255];

    for(;;){
        // citeste o linie de la intrare
        if(cin.getline(buf,255, '\\n') == 0) break; // sfirisitul de fisier
        // se scrie linia in fisier
        for(int i=0;buf[i];i++) textmen.put(buf[i]);
        textmen.put('\\n');
    }
    textmen.close();
}
```

31.18 Să se definească tipul abstract *bara* care permite implementarea unei *bare de meniuri*.

O bară de meniuri este un dreptunghi de lățimea unei linii de pe ecranul setat în mod text. În acest dreptunghi sunt afișate texte care pot fi selectate cu ajutorul tastelor cu săgeți (sägeata spre stînga și spre dreapta). Aceste texte le numim *meniuri*.

Fondul pentru bara de meniuri este afișat folosind culoarea definită de constanta **LIGHTGRAY**.

Textele se scriu folosind culoarea definită de constanta **BLACK**.

Unul dintre meniu este meniu *current*. Pentru acesta, fondul se afisează folosind culoarea definită de constanta CYAN.

Meniu curent se activează prin acționarea tastei *Enter*.

Fiecare meniu își se atașează o funcție care se apelează în momentul activării meniului respectiv. Numim *acțiuni* aceste funcții.

Programul aplicativ conține un tablou global cu pointeri spre aceste acțiuni, numit *tabact*. Acțiunile au prototipul:

```
int actiune(void);
```

Interfața dintre acțiuni se realizează cu ajutorul variabilelor de tip global.

Pentru a putea gestiona simplu acțiunile programului, se introduce tipul PF astfel:

```
typedef int (*PF)();
```

Tabloul *tabact* se declară astfel:

```
int (*tabact[])() = {act1,act2,...,actn};
```

unde:

*act1, act2, ..., actn* - Sunt numele acțiunilor.

astfel

Tipul abstract *bara* are trei date membru:

- *tab*;
- *n*;
- *crt*.

Data membru *tab* este un tablou de elemente de tip *meniu*. Aceste elemente se păstrează în memoria *heap*.

Data membru *n* definește numărul elementelor tabloului *tab*. El este egal cu numărul elementelor tabloului *tabact*.

Data membru *crt* definește meniu curent.

Meniurile se consideră numerotate de la unu.

La afișarea barei de meniu, în mod implicit, meniu cel mai din stînga este selectat (*crt=1*).

Funcțiile membru ale tipului *bara* sint:

- *bara* (constructor);
- *afisbara*;
- *activbara*;
- *apelact*;
- *execmenu*.

Constructorul clasei *bara* rezervă zonă de memorie în memoria *heap* pentru tabloul *tab*. Apoi, citește textele de meniu și păstrează în memoria *heap* elementele de tip *meniu* ale tabloului *tab*.

Textele de meniu se citesc de la intrarea standard.

Funcția *afisbara* afișează bara de meniu în linia unu a ecranului.

Funcția *activbara* afișează meniu selectat, adică al crt-lea meniu.

Funcția *apelact* se apelează la acționarea tastei *Enter*. Ea are ca efect apelul acțiunii corespunzătoare meniului al crt-lea.

Funcția *execmenu* citește de la tastatură caracterele utilizate pentru gestiunea barei de meniu (tastele cu săgeți, tasta *Enter* și tasta *Esc*) și le interpretează în mod corespunzător:

- săgețile permit selectarea meniului curent;
- tasta *Enter* permite apelul acțiunii corespunzătoare meniului curent;
- tasta *Esc* termină execuția aplicației sub controlul barei de meniu.

Menționăm că o aplicație se poate executa sub controlul mai multor bare de meniu.

Tipul abstract *meniu* conține următoarele date membru:

- *text*;
- și
- *pf*.

Data membru *text* este un pointer spre un text de meniu.

Data membru *pf* este un pointer de tip *PF*. Are ca valoare adresa acțiunii atașate meniului.

Funcțiile membru ale tipului *meniu* sint:

- 2 constructori;
- *afis*;
- *apelact*;
- *retlung*.

Clasa *meniu* are un constructor implicit vid și unul care permite inițializarea unui obiect de tip *meniu* cu textul meniului și cu adresa acțiunii meniului.

Textul meniului se păstrează în memoria *heap*.

Funcția *afis* afișează textul meniului.

Funcția *apelact* apelează acțiunea corespunzătoare meniului.

Funcția *retlung* returnează lungimea textului meniului în număr de caractere.

Clașele *meniu* și *bara* se definesc în fișierul de extensie *h* de mai jos, iar funcțiile membru ale lor, în fișierul de extensie *cpp*.

## FIŞIERUL BXXXI18.H

```
typedef int(*PF)();  
  
class meniu {  
    char *text; // pointer spre textul meniului  
    PF pf; // pointer spre acțiunea meniului  
public:  
    meniu();  
    meniu(char *t,PF f);  
    void afis(); // afiseaza textul meniului  
    int apelact(); // apeleaza acțiunea meniului  
    unsigned retlung(); // returneaza lungimea textului meniului  
};
```

```

class bara{
    meniu *tab; // tabloul de meniuri
    int n; // numarul elementelor lui tab
    int crt; // indicele meniului curent
public:
    bara(int s=1); // s defineste indicele meniului curent
    void afisbara(); // afiseaza bara de meniuri
    void activbara(); // afiseaza meniul curent
    int apelact(); // apeleaza actiunea meniului curent
    void execmeniu(); // gestioneaza evenimentele de la tastatura
};

```

## FISIERUL BXXXI18

```

#include <string.h>
#include <iostream.h>
#ifndef __BARA
#include "bxxxi18.h"
#define __BARA
#endif

meniu::meniu() // constructor vid
{
}

meniu::meniu(char *t, PF f)
// constructor utilizat pentru definirea barei de meniuri
{
    // rezerva zona pentru textul meniului
    if((text=new char[strlen(t)+1])==0){
        cout << "memorie insuficienta\n";
        exit(1);
    }
    // transfera textul meniului in memoria heap
    strcpy(text,t);
    // initializeaza pointerul spre actiunea meniului
    pf=f;
}

unsigned meniu::retlung() // returneaza lungimea textului meniului
{
    return strlen(text);
}

void meniu::afis() // afiseaza textul meniului
{
    cprintf(" %s ",text);
}

bara::bara(int a) // constructor
// a este indexul meniului curent
{
    crt=a;
}

```

```

int nr=sizeof tabact/sizeof(PF); // determina numarul meniurilor

if((tab=new meniu[nr])==0)
// rezerva zona pentru bara de meniuri
{
    cout << "memorie insuficienta\n";
    exit(1);
}

for(n=0;n<nr;n++)
// citeste textele meniurilor si initializaaza adresa actiunilor meniurilor
{
    char t[255];
    cout << "textul meniului curent:";
    if(!(cin >> t)) break;
    tab[n]= meniu(t,tabact[n]);
}

void bara::afisbara() // afiseaza bara de meniuri
{
    // ascunde cursor
    _AH=1; _CH=0x20;
    geninterrupt(0x10);
    clrscr();
    int dim=2;

    for(int i=0;i<n;i++) // determina lungimea barei de meniuri
        dim +=tab[i].retlung()+3;
    window(1,1,dim,1); // fereastra pentru bara de meniuri

    textbackground(LIGHTGRAY); // fondul barei
    textcolor(BLACK); // culoarea de afisare a meniurilor
    clrscr();
    for(i=0;i<n;i++){ // afisarea meniurilor
        tab[i].afis();
    }
}

void bara::activbara() // afiseaza meniul curent
{
    int dim=2;

    for(int i=0;i<n;i++) {
        if(crt==i+1) break;
        dim +=tab[i].retlung()+3;
    }
    window(dim,1,dim+tab[i].retlung()+3,1);
    textbackground(CYAN);
    textcolor(BLACK);
    clrscr();
    tab[i].afis();
}

```

```

void bara::execmenu() // functia de gestiune a evenimentelor
{
    char t[255];
    cin.getline(t, 255, '\n'); // videaza buferul de intrare
    char c=0;

    for(;c!=27;){ // seiese la Esc
        c=getch();
        switch(c){
            case 0: // caracter de control
                c=getch();
                switch(c){
                    case 75: // caracterul <
                        if(crt==1) break;
                        crt--; //cursorul pe meniu precedent
                        afisbara();
                        activbara();
                        break;
                    case 77: // caracterul -
                        if(crt==n) break;
                        crt++; //cursorul pe meniu urmator
                        afisbara();
                        activbara();
                }
                break;
            case 13: // caracterul Enter
                apelact();
                break;
            case 27: // caracterul Esc
                break;
        }
    }

    int bara::apelact()
    {
        int i;
        i=tab[crt-1].apelact();
        return i;
    }

    int meniu::apelact()
    {
        int i;
        i=(*pf)();
        return i;
    }
}

```

- 31.19 Să se scrie un program care realizează următoarele transformări:
- măsurile ariilor din hectare în jugăre;
  - și
  - măsurile unghiurilor din radiani în grade sexagesimale.

Programul folosește o bară de meniuri cu următoarele texte:

HaJug RadSexagesimal

La activarea meniului *HaJug* se apelează acțiunea *hajug* care realizează conversia din hectare în jugăre.

La activarea meniului *RadSexagesimal* se apelează acțiunea *radsex* care realizează conversia din radiani în grade sexagesimale.

Ambele acțiuni apeleză funcția *citd* care afișează un text și apoi citește un număr de la intrarea standard. Numărul respectiv este interpretat ca număr de hectare, dacă funcția care face apel este *hajug* și ca grade în radiani dacă apelul se face de către funcția *radsex*.

Rezultatul conversiei se atribuie variabilei globale *rezultat*.

În ambele cazuri, se apeleză funcția *afrez* care afișează un text, împreună cu valoarea variabilei *rezultat*.

Acțiunile *hajug*, *radsex* și funcțiile *citd* și *afrez*, împreună cu funcția principală sunt definite în fișierul de mai jos, de extensie *cpp*.

În acest fișier se mai definesc datele globale:

- *rezultat*;

și

- *tabact*.

Funcția principală instanțiază obiectul *bm*:

bara *bm*;

La această instanțiere se apeleză constructorul clasei *bara* care citește textele meniurilor (*HaJug* și *RadSexagesimal*) și inițializează tabloul *tab* cu textele respective și adresele acțiunilor. Apoi se apeleză funcția *afisbara* pentru a afișa bara de meniuri definită de obiectul *bm*. Apelind în continuare funcția *activbara* pentru același obiect, se afișează meniu curent al căruia fond arc culoarea definită de constanta *CYAN*. După aceea se apeleză funcția *execmenu* pentru obiectul *bm*. În acest moment programul se află în aşteptarea tastării unui caracter.

Se pot actiona tastele cu săgeți (sägeata spre stînga și săgeata spre dreapta), tasta *Enter* și *Esc*.

Alte taste dacă sunt actionate nu au nici un efect.

Cu ajutorul tastelor cu săgeți se poate defini meniu curent.

La actionarea tastei *Enter* se va apela una din acțiunile *hajug* sau *radsex*. La actionarea tastei *Esc*, se revine din funcția *execmenu* și se întrerupe execuția programului.

## FIŞIERUL BXXXI19

```

#include <stdlib.h>
#include <conio.h>
#include <dos.h>

int hajug(); // conversie din ha in jugar
int radsex(); // conversie din radian in sexagesimal

```

```

int (*tabact[])()={hajug,radsexa}; // tabela de actiuni

#include "bxxx18.cpp"      // clasele meniu si bara
#include <iostream.h>

double rezultat;

int citd(char *text,double& d)
/* - afisaza text;
   - citeste un numar;
   - returnaza:
     0 - la EOF;
     1 - altfel */
{
    char t[255];
    double x;

    // afisaza cursor
    _AH=1; _CH=6; _CL=7;
    geninterrupt(0x10);
    window(1,2,80,25);
    textbackground(BLACK);
    textcolor(WHITE);
    clrscr();

    for(;;){
        gotoxy(1,wherey()+1);

        if(text && *text) cout << text;

        if(cin.getline(t,255,'\n')==0){
            // ascunde cursor
            _AH=1; _CH=0x20;
            geninterrupt(0x10);
            return 0;
        }

        istrstream ob(t);
        if(ob >> x) break;
        cout << "nu s-a tastat un numar\n";
    }
    d=x;

    // ascunde cursor
    _AH=1; _CH=0x20;
    geninterrupt(0x10);
    return 1;
}

void afrez(char *s)
{
    gotoxy(1,wherey()+1);
    cout << s << " " << rezultat;
}

```

```

int hajug()
{
    if(citd("aria in Ha:",rezultat)==0) {
        cout << "S-a tastat EOF\n";
        return 0;
    }
    rezultat /= 0.57546415;
    afrez("Aria in Jugare");
    return 1;
}

int radsexa()
{
    if(citd("masura unghiului in radiani:",rezultat)==0) {
        cout << "S-a tastat EOF\n";
        return 0;
    }
    rezultat *= 180.0/3.14159265358979;
    afrez("Unghiul in Grade sexagesimale");
    return 1;
}

main()
{
    // sterge tot ccranul
    window(1,1,80,25);
    textbackground(BLACK);
    textcolor(WHITE);
    clrscr();
    bara bm; // instantiaza bara de meniuri
    bm.afisbara();
    bm.activbarsa();
    bm.execmeniu();

    // afisaza cursor
    _AH=1; _CH=6; _CL=7;

    geninterrupt(0x10);
    window(1,1,80,25);
    textbackground(BLACK);
    textcolor(WHITE);
    clrscr();
}

```

## BIBLIOGRAFIE

1. C.Bohn, G.Jacopini  
**FlowDiagrams, Turning Machines and Languages with Only Two Formation Rules**  
Comm ACM, 9, 5, pag. 366 - 371, 1966
2. Brian W. Kernighan, Dennis M. Ritchie  
**The C Programming Language**  
Prentice-Hall, Inc, Englewood Cliffs, New Jersey 1978
3. Donald E. Knuth  
**Tratat de programarea calculatoarelor, vol. 1 - 3**  
Editura Tehnică, Bucureşti 1974
4. Valentin Cristea și alții  
**Dicționar de informatică**  
Editura științifică și enciclopedică, Bucureşti 1981
5. Narian Gehani  
**C: An Advanced Introduction**  
AT&T Bell Laboratories Murray Hill, New Jersey 1985
6. Joel E. Richardson, Michel J. Carey, Daniel T. Schuh  
**The Design of the E Programming Language**  
Computer Sciences Department  
University of Wisconsin Madison, Wi53706 1989
7. \* \* \*  
**BORLAND INTERNATIONAL TURBO C++: Getting Started**  
1990
8. \* \* \*  
**BORLAND INTERNATIONAL TURBO C++: Programmer's Guide**  
1990
9. Frederic Lung  
**Le Langage C++**  
CNIT Paris - La Defense, 1990
10. Setrag Khoshafian, Razmik Abnous  
**Object Orientation Concepts, Languages, Databases, User Interfaces**  
John Wiley&Sons, Inc. New York  
Chichester Brisbone Toronto, Singapore 1990
11. Aaron M. Tenenbaum, Yedidya Langsam, Moshe J. Augenstein  
**Data Structures Using C**  
Prentice - Hall International, Inc., 1990
12. Clara Ionescu, Ioan Zsako  
**Structuri arborescente cu aplicațiile lor**  
Editura Tehnică, Bucureşti 1990
13. Liviu Negrescu  
**Introducere în limbajul C vol. 1 - 2**  
Colecția GLOB, 1990
14. Stroustrup Bjarne  
**The C++ Programming Language - Second Edition**  
Copyright 1991 by AT&T  
Bell Telephone Laboratories, Incorporated
15. Liviu Negrescu  
**Limbajul C - Culegere de programe; Fasciculele 1 - 2,**  
Cluj-Napoca 1991
16. Liviu Negrescu  
**Limbajul TURBO C**  
Editura LIBRIS, Cluj-Napoca 1992
17. Liviu Negrescu  
**Introducere în mediul de dezvoltare integrat TURBO C**  
Editura LIBRIS, Cluj-Napoca 1992
18. Ionuț Mușlea  
**Initiere în C++ - Programare orientată pe obiecte**  
Editura MicroInformatica, Cluj-Napoca 1992
19. Donald L. Shell  
CACM 2(iulie), pag. 30-32, 1959
20. C. A. R. Hoare  
**Quicksort**  
Comp. j., 5, No. 1, 1962
21. Liviu Negrescu  
**Introducere în limbajul C**  
Editura MicroInformatica, Cluj-Napoca 1993