

Seria Linux este coordonată de dr. Gabriel Ciobanu

Dragoș Acostăchioae este programator de sistem și aplicații Linux, administrator de rețea la firma BIOSFARM din Iași. Predă cursul de „Programarea Calculatoarelor” la Colegiul de Tehnologie Informatică din cadrul Universității „A.I. Cuza”. Este autorul a numeroase articole despre sistemul de operare Linux, având publicată la editura Polirom lucrarea *Administrarea și configurarea sistemelor Linux*. Domenii principale de interes: sistemele deschise, programarea orientată-obiect, aplicațiile client-server și securitatea sistemelor UNIX.

Dragoș Acostăchioae

PROGRAMARE C ȘI C++ PENTRU LINUX

© 2002 by Editura POLIROM

www.polirom.ro

Editura POLIROM
Iași, B-dul Copou nr. 4, P.O. BOX 266, 6600
București, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33, P.O. BOX 1-728, 70700

Descrierea CIP a Bibliotecii Naționale a României:

ACOSTĂCHIOAE, DRAGOȘ

Programare C și C++ pentru Linux / Dragoș Acostăchioae;
cuvânt înainte de Conf. dr. Dorel Lucanu; Iași; Polirom, 2002.
216 p.; 24 cm

ISBN: 973-681-112-3

I. Lucanu, Dorel (pref.)

004.43 C
004.43 C++

Printed in ROMANIA

Cuvânt înainte de Conf. dr. Dorel Lucanu



130252
B.C.U. - IASI

POLIROM
2002

CUPRINS

<i>Cuvânt înainte.....</i>	9
<i>Prefață.....</i>	11
<i>Ce este Linux?</i>	14

Partea întâi

PROGRAMAREA ÎN LIMBAJUL C

Capitolul 1	
<i>Scurt istoric</i>	19
Capitolul 2	
<i>Noțiunile de bază ale limbajului C</i>	20
2.1. Variabile.....	21
2.2. Constante.....	23
2.3. Operatori	24
2.4. Funcții	28
2.5. Tablouri.....	30
2.6. Enumerații.....	30
2.7. Comentarii.....	31
Capitolul 3	
<i>Utilizarea datelor</i>	32
3.1. Domeniul variabilelor	32
3.2. Conversii de tip	33
Capitolul 4	
<i>Controlul fluxului</i>	35
4.1. Instrucțiuni	35
4.2. <i>if-else</i>	35
4.3. <i>switch</i>	36
4.4. <i>while</i>	37
4.5. <i>for</i>	38
4.6. <i>do-while</i>	39
4.7. <i>break</i> și <i>continue</i>	39
4.8. <i>goto</i>	40

Capitolul 5	
Pointeri	41
5.1. Tablouri și pointeri.....	41
5.2. Tablouri de pointeri.....	43
5.3. Pointeri la funcții.....	44
	47
Capitolul 6	
Structurarea programelor: funcțiile	49
6.1. Recursivitatea.....	49
6.2. Funcția <i>main()</i>	49
	50
Capitolul 7	
Structuri	51
7.1. Structuri cu autoreferire	51
7.2. <i>typedef</i>	53
	54
Capitolul 8	
Preprocesorul C	55
Capitolul 9	
Bibliotecile standard C	57
Capitolul 10	
Probleme propuse	63

Partea a doua

PROGRAMAREA ÎN LIMBAJUL C++

Capitolul 1	
Scurt istoric	67
Capitolul 2	
Noțiunile de bază ale programării orientate-obiect	69
2.1. Premisele limbajelor orientate-obiect.....	69
2.2. Concepte fundamentale.....	70
Capitolul 3	
Clase	72
3.1. Declararea claselor.....	72
3.2. Membrii unei clase.....	73
3.3. Crearea și distrugerea obiectelor.....	74
3.4. Conceptul de moștenire.....	77
Capitolul 4	
Programare avansată utilizând clase	80
4.1. Controlul accesului la clase.....	80
4.2. Funcții și clase prietene	82
4.3. Cuvântul-cheie <i>this</i>	83

4.4. Redefinirea operatorilor	84
4.5. Moștenirea multiplă	89
4.6. Conversii de tip definite de programator.....	91
4.7. Constructorul de copiere	93
4.8. Clase abstrakte	93
4.9. Membri statici ai unei clase.....	94

Capitolul 5	
Fluxuri	96
5.1. Introducere	96
5.2. Obiecte standard.....	97
5.3. Redirecțări.....	97
5.4. <i>cin</i>	98
5.5. <i>cout</i>	100
5.6. Operații de intrare/ieșire cu fișiere	102

Capitolul 6	
Tratarea excepțiilor	103

Capitolul 7	
Template-uri	106

Capitolul 8	
Proiectarea și dezvoltarea de aplicații orientate-obiect	121

Capitolul 9	
Probleme propuse	125

Partea a treia

UNELTE DE PROGRAMARE PENTRU LINUX

Capitolul 1	
Editoare	129
1.1. Editorul <i>Emacs</i>	129
1.2. Alte editoare: <i>mcedit</i> , <i>joe</i> și <i>vi</i>	132

Capitolul 2	
Compilatoare	135
2.1. Compilarea programelor C: <i>GCC</i>	136
2.2. Compilarea programelor C++: <i>G++</i>	138
2.3. Crearea de biblioteci: <i>ar</i>	139

Capitolul 3	
Instrumente de programare	140
3.1. Depanarea programelor: <i>GDB</i>	140
3.2. Utilitarul <i>make</i>	143
3.3. <i>autoconf</i> , <i>automake</i> și <i>libtool</i>	148
3.4. Utilitarele <i>diff</i> , <i>patch</i> și <i>diffutils</i>	166

3.5. Sistemul de control al versiunilor CVS.....	170
3.6. Alte programe utile	174
3.7. Documentarea programelor: <i>DOC++</i> și <i>code2html</i>	178
3.8. Verificarea programelor: <i>splint</i>	181

Capitolul 4

Medii integrate de dezvoltare	183
4.1. Depanatorul vizual <i>DDD</i>	183
4.2. Mediul integrat <i>KDevelop</i>	184
4.3. Programul <i>Glade</i>	186

Capitolul 5

Programarea <i>open source</i>.....	187
5.1. Ce este <i>open source</i> ?.....	187
5.2. Managementul Web al proiectelor software – <i>sourceforge.net</i>	190

Anexe

Anexa 1	
Functiile standard ANSI C	195
Anexa 2	
Functiile standard POSIX.....	199
Anexa 3	
Licenta Publica GNU.....	204
Bibliografie	211
Glosar de termeni.....	213

CUVÂNT ÎNAINTE

Mișcarea *free software* este mai mult o filosofie de viață decât una de preț și se referă la realizarea de programe-calculator gratuite. Această filosofie se referă la libertatea de a rula un program pentru orice scop dorit, la libertatea de a studia modul de funcționare al programului, la libertatea de a redistribui copii prietenilor și colegilor, la libertatea de a aduce îmbunătățiri și de a le oferi public în folosul comunității. Scopul acestei mișcări este unul social, în sensul că propune un anumit comportament atât producătorilor de programe-calculator, cât și utilizatorilor acestor programe. De multe ori *free software* este confundat cu termenul (mișcarea) *open source*, care s-ar putea traduce prin *cod-sursă deschis*. Diferența este una de ordin practic: *open source* este o metodologie de elaborare a programelor-calculator pentru care *free software* este doar o subpolitică. Cele două mișcări formează împreună *comunitatea free software*.

Unul dintre fenomenele majore ale mișcării *free software*, dacă nu cumva cel mai important dintre ele, este proiectul GNU. Acesta a fost inițiat de Richard Stallman în 1983 având scopul de a crea un *free software* compatibil cu sistemul Unix (Gnu's Not Unix!). Inevitabil, acest software trebuia să includă și un sistem de operare. La început au fost dezvoltate câteva utilitare: editorul Emacs, un compilator C, un interpretor de comenzi etc. În anii '90, Linus Torvalds a dezvoltat un nucleu de sistem de operare pe care l-a numit *Linux*. Acest nucleu a fost integrat cu celelalte componente și a format un sistem de operare complet, cunoscut ca sistem *GNU bazat pe Linux (linux-based GNU system)*. Astăzi, când spunem sistem de operare Linux, înțelegem de fapt un sistem GNU bazat pe Linux.

Prezenta carte este o invitație de a intra în comunitatea *free software*. Pentru a deveni un membru deplin al acestei comunități, ai nevoie să cunoști instrumentele create și utilizate de către ea. Autorul s-a oprit la limbajele C și C++, cele mai utilizate în momentul de față, și la principalele programe-utilitar strict necesare în dezvoltarea programelor. Accentul este pus pe modul în care sunt utilizate aceste instrumente, fiind prezentate multe exemple tipice. Sunt convins că această carte îi va face pe mulți dintre (viitorii) programatori să se simtă mai liberi în timpul exercitării profesiei.

*Dorel Lucanu
Facultatea de Informatică
a Universității „A.I. Cuza” din Iași*

PREFATĂ

Avându-și rădăcinile în anii de început ai descoperirilor științei calculatoarelor moderne, limbajul C, sistemul de operare UNIX și Internetul se întrepătrund, nici unul dintre ele nu ar putea ființa fără celealte două.

Apariția sistemului Linux a constituit un nou imbold în cadrul evoluției UNIX-ului și a curentului *open source*. Totul se bazează pe un fundament numit *cooperare*, care în definitiv se bazează pe oameni. Cercetători ca Thompson, Brian Kerninghan, Dennis Ritchie și Bjarne Stroustrup, iar mai târziu Richard Stallman și Linus Torvalds au înțeles că rezultatele științei trebuie să împărtășite cu ceilalți, cu membrii comunității.

Fenomenul *open source* nu a condus doar la elaborarea unui număr uriaș de programe de calitate, liber disponibile prin intermediul Internetului. Acest curent a schimbat mentalitatea oamenilor, făcându-i să gândească nu egoist, ci în folosul unei comunități.

Concomitent cu multiplele posibilități de utilizare ca server, sistem personal sau la birou, Linux este o platformă ideală pentru dezvoltarea de programe. Instrumentele de programare GNU își au originile în anii '80, ele nefiind inventate la comandă, ci izvorând din nevoile programatorilor de a-și ușura munca.

Puterea instrumentelor GNU pentru mediile UNIX rezidă în capacitatea extraordinară a acestora de a interacționa. Ideea nu este de a construi monoliți complecsi, greu de întreținut, de dezvoltat, care rezolvă toate problemele, ci multe componente mici, făcute să comunice între ele într-un mod natural, standardizat. Un mediu integrat, spre exemplu, nu implementează un depanator propriu, ci utilizează programul GDB, permitând manipularea acestuia prin comenzi vizuale, dar oferind totodată accesul la linia de comandă. La fel stau lucrurile și în cazul suportului pentru CVS și așa mai departe.

În lucrarea de față sunt trecute în revistă noțiunile de bază ale programării C și C++ și, totodată, particularitățile acestea pe sistemele Linux. De asemenea, sunt prezentate principalele instrumente și medii de programare în sistemele de operare Linux și UNIX, în special pentru limbajele C și C++.

Cartea conține și o prezentare a fenomenului *open source*, îndemnând cititorul să utilizeze software liber și chiar să contribuie activ la dezvoltarea acestui tip de software.

Așteptăm reacțiile, remarcile și eventualele observații sau corecții pe adresa dragos@biosfarm.ro. De asemenea, îi invităm pe cititori să viziteze pagina dedicată acestui volum, <http://www.biosfarm.ro/~dragos/prg>.

Structură generală

Lucrarea este structurată în părți: în prima sunt cuprinse noțiunile de bază ale programării în limbajul C, în timp ce partea a doua tratează elementele esențiale ale programării în limbajul C++. Ambele segmente conțin numeroase exemple și propun o serie de probleme. Partea a treia prezintă principalele instrumente de programare din mediile Linux și UNIX, cum ar fi compilatoarele – GCC, G++, instrumentele pentru depanarea programelor – GDB, cele pentru automatizarea compilării și realizarea de programe complexe – make, autoconf, automake, libtool sau cele destinate lucrului în echipă – CVS. Sunt prezentate totodată o serie de medii integrate de dezvoltare existente pe platformele menționate.

Lucrarea include și o listă de referințe bibliografice necesare pentru însușirea temeinică și profundarea limbajelor C și C++.

Cui se adreseză această carte

Cartea se adreseză atât studenților și elevilor care urmează cursuri de programare, cât și specialiștilor în programare, care doresc fie să învețe sau să utilizeze platforma Linux, fie să își porteze programele pe aceasta.

Convenții utilizate în această carte

Bold-ul este utilizat pentru opțiuni, cuvinte-cheie etc., având rolul de a le evidenția.

Italic-ul este utilizat pentru nume de variabile, constante, cuvinte-cheie etc., care vor trebui în general, să fie înlocuite cu valoarea lor reală de către utilizator.

Courier-ul este folosit pentru nume de comenzi, de fișiere, de utilizatori sau grupuri, precum și pentru texte care trebuie tastate efectiv de către utilizator. Este de asemenea utilizat pentru conținutul unui fișier sau ieșirea unor comenzi, în cadrul exemplelor.

| semnifică SAU logic. Astfel, optiune1 | optiune2 înseamnă „fie optiune1, fie optiune2”.

[...] semnifică un text optional, deci care nu este obligatoriu de folosit.

... semnifică o porțiune de text care a fost omisă pentru a nu îngreuna lizibilitatea sau pentru a reduce din spațiul utilizat.

este promptul implicit al interpreterului de comenzi pentru utilizatorul *root*.

Cum a fost scris acest material

Această lucrare a fost realizată în HTML 4, utilizând doar programe *open source*: Red Hat Linux 7.3, GNU Emacs, XFig, Mozilla, Konqueror, GIMP.

Programele au fost compilate cu GCC și EGCS.

Mulțumiri

Doresc să îmi exprim recunoștință pentru ajutorul acordat în corectarea și definitivarea acestei cărți *Conf. dr. prof. Dorel Lucanu*, precum și colaboratorilor – în același timp prietenilor – *Sabin Buraga* și *Mihaela Brut*. Mulțumesc de asemenea primului meu dascăl de C, *Conf. dr. Claudia Botez*.

*Autorul
Iași, septembrie 2002*

CE ESTE LINUX?

Linux este un sistem de operare gratuit (*open source*), compatibil cu UNIX. UNIX este un sistem multitasking și multiuser, în care utilizatorii au acces la resursele calculatorului de la diferite terminale plasate local sau la distanță, putând executa în mod concurrent un număr nedeterminat de programe. Linux respectă standardele POSIX, suportă toată gama de aplicații GNU și posedă o interfață grafică X Windows System.

Linux a apărut inițial ca un proiect al lui Linus Torvalds, student pe atunci la Universitatea din Helsinki. Prima versiune utilizabilă a fost lansată pe 5 octombrie 1991. Din acel moment, numărul celor care lucrează la Linux, precum și numărul de utilizatori, a crescut în mod impresionant. În momentul față se estimează că există circa 18 milioane de utilizatori de Linux (pentru detalii, a se vedea situl <http://counter.li.org>).

- Spre deosebire de alte sisteme de operare, nici o firmă nu este proprietara sistemului Linux. Mai mult decât atât, el este protejat de Licența Publică GNU (GNU este o fundație care apără interesele autorilor de programe gratuite – pentru informații suplimentare, a se vizita situl <http://www.gnu.org>), care stipulează faptul că se poate copia și utiliza gratuit codul programelor, cu condiția de a se permite și altora să facă același lucru. Textul complet al acestei licențe se găsește în Anexa 3.

Sistemul Linux propriu-zis este alcătuit dintr-un nucleu (*kernel*) și un număr uriaș de programe și utilitare. Este un sistem de operare complet, care conține practic orice tip de aplicație, putând astfel juca o varietate de roluri, printre care:

- **Server.** Sistemul Linux conține un suport excepțional pentru rețea, putând oferi atât serviciile specifice Internetului, cum ar fi Web, FTP, poștă electronică etc., precum și alte servicii de rețea, cum ar fi server de imprimante, server de fișiere (NFS) și.a.m.d. De asemenea, Linux poate substitui un server Windows NT sau Netware. Stabilitatea, fiabilitatea și securitatea sa îi conferă posibilitatea de a fi utilizat pe servere *high-end* și în medii critice.
- **Sistem personal.** Linux poate fi utilizat la redactarea de texte, tipărirea la imprimantă, conectarea și navigarea pe Internet, citirea corespondenței. În plus, este o platformă multimedia perfectă pentru jocuri, pentru vizionarea de filme sau audiuția de piese muzicale.
- **Sistem utilizat la serviciu.** Sistemul Linux conține o suită completă de programe pentru birou, cum ar fi programe de calcul tabelar, redactare de texte, realizarea de prezentări etc. Instrumentele pentru acces la Internet reprezintă și ele un punct forte al acestui sistem.
- **Sistem educațional.** Atuul principal este costul practic nul al sistemului. Instalarea de sisteme proprietare pe multe calculatoare poate fi extrem de

costisitoare. Linux include aplicații, compilatoare și medii integrate de dezvoltare a căror calitate este în general superioară celei a programelor comerciale.

Un sistem Linux, alcătuit dintr-un program de instalare, un nucleu și aplicații gata compilate se numește *distribuție Linux*. Numeroase companii și organizații au realizat asemenea distribuții. Acestea sunt disponibile gratuit pe Internet, dar pot fi achiziționate și contra cost ca pachete formate din CD-ROM-uri și manuale de utilizare. Distribuțiile Linux cele mai răspândite sunt *Red Hat*, *Mandrake*, *SuSE*, *Caldera* și *Slackware*. Distribuția *Red Hat* poate fi descărcată de la adresa <http://www.redhat.com> sau *mirrors* (oglindiri ale unui sit Internet), iar *Mandrake*, de la adresa <http://www.mandrake.com>.

Contra opiniei generale, suportul tehnic pentru programele *open source* are de multe ori o calitate superioară celui oferit pentru aplicațiile comerciale. Prin formularea unei întrebări sau unei cereri de ajutor pe o listă de discuție, se poate obține un răspuns la orice problemă, în câteva ore. De asemenea, există numeroase companii, îndeosebi cele care comercializează distribuții, ce oferă consultanță și suport tehnic profesional, contra cost.

Nucleul Linux a fost adaptat (*portat*) pe foarte multe platforme, începând cu Intel și continuând cu SUN Sparc, PowerPC, Motorola 68000 (Atari și Amiga), MIPS (SGI), DEC Alpha, ARM etc.

În ultimii ani, Linux a cunoscut o largă răspândire și în România. Fiind gratuit, nu există posibilitatea de a intra în conflict cu legea dreptului de autor, ceea ce va conduce cu siguranță la utilizarea sa și în cadrul firmelor, precum și în mediile cu multe calculatoare. În majoritatea centrelor universitare din țară, Linux este extrem de popular, multe materii folosindu-l ca platformă de referință (Sisteme de operare, Programare și.a.m.d.). Cei mai mulți furnizori de servicii Internet folosesc Linux datorită capabilităților sale excelente de rețea, flexibilității în configurare și stabilității sale. De asemenea, profesioniștii științei calculatoarelor îl utilizează pentru instrumentele puternice de programare, procesare de texte etc. de care dispune.

În seria Linux, din care face parte și prezentul volum, au apărut și vor apărea lucrări de referință privind programarea, utilizarea și administrarea în cadrul acestui sistem de operare.

Partea întâi

**PROGRAMAREA
ÎN LIMBAJUL C**

Capitolul 1

SCURT ISTORIC

Fă aşa!
(Jean Luc Picard)

În anul 1970, doi programatori, Brian Kerningham și Dennis Ritchie, de la AT&T (*Bell Laboratories*), au creat limbajul C. Principalul scop pentru care a fost realizat acest limbaj a fost rescrierea sistemului de operare UNIX, pentru a-l face portabil pe toate platformele existente. Din acest motiv, majoritatea programelor pentru UNIX sunt scrise tot în C. Limbajul C este un descendant al limbajelor *B*, *MACRO11* și *BCPL*. Prima versiune a limbajului C a rulat pe un calculator de tip *PDP-11*.

Marele avantaj al limbajului C este acela de a fi extrem de flexibil și de a permite programarea atât la nivel înalt, cât și la nivel scăzut. C-ul este un limbaj procedural, ceea ce face ca un program scris în C să înceapă de obicei cu definirea structurilor de date și să continue cu definirea funcțiilor, pentru lucrul cu aceste structuri.

C este un limbaj de programare cu scop general și nu este dependent în vreun fel de hardware sau de mașina pe care se utilizează. Un program în C ar trebui să se poată compila și executa fără nici un fel de modificări pe orice calculator, atât timp cât se respectă standardele universale acceptate (ANSI C).

Primul manual de programare C a fost realizat în 1978 de către Brian Kerningham și Dennis Ritchie și denumit *The C Programming Language*, cunoscut și sub numele de „cartea albă” sau „K&R.”

În anul 1989, limbajul C a fost standardizat oficial de către comitetul ANSI X3J11.

Capitolul 2

NOȚIUNILE DE BAZĂ ALE LIMBAJULUI C

Nu este corect. Nu este nici măcar greșit.
(Wolfgang Pauli)

C este un limbaj cu un nivel relativ inferior. Astfel, acesta nu posedă operații pentru lucrul cu variabile compuse, cum ar fi siruri de caractere, liste sau tablouri. De asemenea, limbajul nu prevede instrucțiuni de alocare a memoriei, nu conține nici instrucțiuni de intrare/ieșire, și nici de lucru cu fișiere. Toate aceste mecanisme, considerate a fi de nivel înalt, trebuie apelate prin intermediul funcțiilor conținute în biblioteca standard C sau în alte biblioteci. Această caracteristică prezintă avantajul că executabilele (codul-obiect) generate au dimensiune redusă, funcțiile de bibliotecă putând fi legate dinamic de acestea.

Programele C care utilizează doar bibliotecile standard pot fi compilate și executate pe orice combinație mașină-compilator, fără nici o modificare, în situația în care nu utilizează facilități specifice unui anumit compilator. Conversia unui program-sursă de pe o mașină pe alta se numește *portare*.

- Pașii care trebuie urmați în vederea creației unui fișier executabil sunt:
1. Se creează fișierul-sursă, cu ajutorul unui editor de texte.
 2. Se compilează fișierul-sursă, în vederea generării fișierului-obiect.
 3. Se link-editează fișierul-obiect, generându-se fișierul executabil.

Dacă se utilizează un mediu integrat, acesta include un editor de texte. În caz contrar, poate fi folosit orice editor de texte, cum ar fi vi, emacs, joe etc. Editorul de texte nu trebuie totuși să introducă caractere speciale sau alte caractere de formatare, ci să genereze doar text ASCII. Fișierele create cu ajutorul editorului se numesc *fișiere-sursă*.

Pentru a urma tradiția manualelor de C, vom prezenta în continuare cel mai scurt program posibil, care va afișa pe ecran mesajul „hello, world”:

```
main()
{
    printf("hello, world\n");
}
```

Presupunând că fișierul-sursă în care a fost salvat programul de mai sus se numește `hello.c`, compilarea acestuia se poate face prin comanda:

```
gcc -o hello hello.c
```

Fișierul executabil rezultat se va numi `hello` și poate fi lansat în execuție cu ajutorul comenzii:

```
./hello
```

2.1. Variabile

Variabilele sunt unitățile de informație care se pot modifica de-a lungul execuției programului. Ele au o durată de viață bine determinată, în funcție de locul și modul de declarare (vezi *infra*).

Numele variabilelor se specifică prin intermediul identificatorilor și sunt alcătuite din litere și cifre; primul caracter trebuie să fie o literă. Poate fi utilizat și caracterul „_”, care joacă tot rol de literă. Literele mari și mici sunt considerate caractere distințe; în mod convențional, literele mici sunt folosite pentru variabile, iar cele mari, pentru constante simbolice. Numărul de caractere din cadrul numelui este nelimitat. O serie de cuvinte-cheie nu pot fi utilizate ca nume de variabile:

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	unsigned
union	void	volatile	while			

Există doar următoarele tipuri de date de bază în C:

char	un caracter
int	un număr întreg. Numărul de biți în care este reprezentată valoarea depinde de tipul calculatorului gazdă (vezi <i>infra</i>)
float	număr flotant în simplă precizie
double	număr flotant în dublă precizie

În plus, există doi calificatori care pot fi aplicati tipului int: short și long, referindu-se la diferite mărimi de întregi. De asemenea, tipurilor char și int li se poate aplica calificatorul unsigned (fără semn). Cuvântul int poate fi omis în aceste cazuri.

Pe sistemele Linux i386 (compatibile IBM PC):

Tip	Lungime	Plajă valori
char	8 biți	cu semn: -128 – 127, fără semn: 0 – 255
int	32 biți	cu semn: -2147483648 – 2147483647, fără semn: 0 – 4294967295
short	16 biți	cu semn: -32768 – 32767, fără semn: 0 – 65536
long	32 biți	cu semn: -2147483648 – 2147483647, fără semn: 0 – 4294967295
float	32 biți	$3.4 \cdot 10^{-38} – 3.4 \cdot 10^{38}$
double	64 biți	$1.7 \cdot 10^{-308} – 1.7 \cdot 10^{308}$

Pentru a afla valoarea minimă, respectiv cea maximă pe care o poate memoria o variabilă de tip numeric, trebuie consultat fișierul `/usr/include/ limits.h`, care conține definiții de genul `CHAR_BIT` – numărul de biți dintr-un `char`, `CHAR_MAX` – valoarea maximă a unui `char` și.m.d.

Toate variabilele trebuie declarate înainte de a fi utilizate. O declarație menționează un tip și este urmată de una sau mai multe variabile de acel tip, specificate prin numele acestora și separate prin virgulă:

```
int x, y, z;
char s[1000];
```

Variabilele pot fi, de asemenea, inițializate în momentul declarării lor, cu o valoare inițială, precizând după numele acestora semnul egal „=” și o constantă:

```
int i = 0;
char s[20] = "Salut";
```

De remarcat este faptul că la declararea sirului de caractere `s` se alocă în mod automat 20 de caractere, chiar dacă lungimea valorii inițiale este mai mică decât atât.

Variabilele nu sunt inițializate implicit de către compilator. Variabilele neinițializate explicit au valoare nedefinită (engl. *garbage*), cu alte cuvinte programatorul nu trebuie să se bazeze pe o anumită valoare a acestora.

Variabile de tip `const`

Modificatorul de acces `const` este o promisiune a programatorului pentru compilator că valoarea unei variabile nu va fi modificată după inițializarea sa. Dacă aceasta nu este inițializată, compilatorul o va inițializa pe zero. În acest sens, este important de remarcat că declarația de genul `const char *str = "hello"` este un pointer către o informație care nu poate fi modificată. Pointerul în sine poate fi modificat (vezi capitolul 5).

Variabilele de acest tip sunt utile în general pentru declararea parametrilor funcțiilor (vezi paragraful 2.4). Spre exemplu, o funcție care contorizează numărul de caractere dintr-un sir nu trebuie să modifice conținutul acestuia:

```
int lungime_sir(const char *sir)
{
    int contor = 0;

    while(*sir++)
        contor++;

    return contor;
}
```

Variabile de tip `register`

Cuvântul-cheie `register` precizează compilatorului că variabila precizată este des utilizată și ar trebui memorată într-o dintre registrele procesorului. Pe compilatoarele moderne, nu este necesar să fie utilizat. Multe dintre aceste compilatoare ignoră chiar această directivă.

2.2. Constante

O constantă numerică desemnează o valoare numerică fixată.

Sufixul „L” sau „l” atașat unei constante numerice forțează reprezentarea sa ca fiind de tip `long`. Sufixul „U” sau „u” forțează reprezentarea `unsigned`, iar dacă depășește dimensiunea maximă a acestuia, va fi automat de tip `unsigned long`. Cele două sufixe pot fi utilizate și împreună.

Toate constantele în virgulă flotantă sunt prin definiție de tip `double`. Forțarea la tipul `float` poate fi făcută prin adăugarea sufixului „F” sau „f”.

Pentru constantele numerice sunt permise și notatiile de genul `123.456e-7` sau `0.123E3`. Există de asemenea o notație specială care permite exprimarea de valori octale și hexazecimale: „0” la începutul unei constante înseamnă că aceasta este exprimată în octal; prefixul `0x` înseamnă că valoarea este exprimată în hexa.

O constantă de tip caracter este un singur caracter, scris între apostrofe ‘ ’. Valoarea este corespondentul numeric al caracterului în setul de caractere al mașinii. De exemplu, în setul de caractere ASCII, caracterul „0” are valoarea 48. Constantele de tip caracter participă în operațiile numerice ca orice alte valori numerice, de exemplu în cadrul operațiilor de comparație cu alte caractere.

Anumite caractere standard pot fi reprezentate cu ajutorul *secvențelor escape*, cum ar fi \n (linie nouă), \t (tab), \\ (backspace), \0... (corespondentul valorii specificate în octal) sau \0x... (corespondentul valorii specificate în hexa).

O constantă-șir de caractere este o secvență compusă din zero sau mai multe caractere, între ghilimele duble ", ca "Hello, world!" sau "" (un șir nul). Secvențele escape descrise mai sus sunt valabile și în acest caz. Utilă poate fi secvența \\", care înlocuiește caracterul ghilimele duble (imposibil de utilizat ca atare).

Practic, un șir de caractere este un tablou ale căruia elemente sunt caractere de tip char. Compilatorul plasează în mod automat un caracter nul, \0 (având practic valoarea numerică zero), la sfârșitul șirului, caracter denumit *terminator nul*. Funcția următoare, lungime_sir(s), returnează lungimea unui șir de caractere s, primit ca argument, exclusiv terminatorul \0:

```
int lungime_sir(char *s)
{
    int i = 0;

    while(s[i] != '\0')
        i++;

    return i;
}
```

La manipulările de șiruri de caractere trebuie avut în vedere că este necesar a fi alocată suficientă memorie pentru a memora întregul șir, inclusiv acest caracter de terminare.

2.3. Operatori

Operatorii specifică operațiile care se fac cu variabile și constante, conform tipurilor acestora.

2.3.1. Operatorul de atribuire

Operatorul de atribuire este semnul egal „=”. Valoarea din dreapta semnului egal este atribuită variabilei din stânga. Într-o instrucțiune se pot face mai multe atribuiri:

```
rez = x = y;
```

În acest caz, ordinea evaluării este de la dreapta la stânga. Astfel, y va fi atribuit lui x, care la rândul său va fi atribuit lui rez.

2.3.2. Operatori aritmetici

Operatorii aritmetici binari sunt „+”, „-”, „*”, „/” și operatorul modulo, „%”. Există operatorul unar „-”, dar nu există operatorul unar „+”. De notat este faptul că împărțirea întregilor trunchiază orice parte fracționară. Operatorul „%” nu poate fi aplicat la float sau double.

2.3.3. Operatori relaționali

Acești operatori permit compararea a două valori. Dacă rezultatul comparației este fals, valoarea rezultată este 0 (zero), respectiv 1 în caz contrar. Operatorii relaționali sunt: > (mai mare), >= (mai mare sau cel mult egal), < (mai mic), <= (mai mic sau cel mult egal), == (egal), != (inegal).

2.3.4. Operatori logici

Aceștia sunt && și || (ȘI logic), ||| (SAU logic), numiți și conectori logici, și „!” (negație logică). Expresiile care conțin operatori logici sunt evaluate de la stânga la dreapta, iar evaluarea se oprește în momentul în care se cunoaște rezultatul (cu alte cuvinte, dacă rezultatul nu mai poate fi modificat de operațiile rămase, acestea sunt abandonate).

În C nu există un tip de date corespunzător tipului bool, acesta existând în schimb în limbajul C++. În cazul operațiilor logice, o valoare nenulă (în mod convențional 1) este considerată ca având valoarea de adevăr adevărat, iar valoarea 0 desemnează fals.

Operatorul unar de negație „!” convertește un operand non-zero sau adevărat în zero și un operand egal cu zero sau fals în 1. Spre exemplu, expresia:

```
if(!ok)
```

este echivalentă cu:

```
if(ok == 0)
```

2.3.5. Operatorii de incrementare și decrementare

Operatorul de incrementare `++` adună 1 la operandul său, iar `--` scade 1 din operand. Acești doi operatori pot fi utilizati atât ca prefix (înaintea variabilei, e.g. `++i`), cât și ca sufix (după variabilă, e.g. `i++`). Efectul asupra operandului este același, dar folosirea cu prefix modifică variabila înainte de a-i folosi valoarea, în timp ce utilizarea ca sufix modifică variabila după ce valoarea sa a fost folosită în cadrul expresiei.

Folosind această facilitate, putem simplifica unele construcții, cum ar fi funcția de calcul a lungimii unui sir de caractere, descrisă în paragraful 2.2:

```
int lungime_sir(char *s)
{
    int i = 0;

    while(s[i++] != '\0');

    return i;
}
```

2.3.6. Operatori logici pe biți

Operatorii logici pe biți sunt următorii: „`&`” (ȘI bit cu bit), „`|`” (SAU bit cu bit), „`^`” (SAU EXCLUSIV bit cu bit), `<<` (deplasarea bițiilor spre stânga), `>>` (deplasarea bițiilor spre dreapta), și „`~`” (complement față de 1). Acești operatori nu se pot utiliza pentru valori de tip `float` sau `double`.

Operatorul ȘI „`&`” este utilizat în general pentru a masca anumiți biți ai unui număr. Mascarea bițiilor reprezintă selectarea unor anumiți biți dintr-un octet. Pentru aceasta, se aplică operația ȘI bit cu bit cu un număr având setați biții respectivi pe 1. De exemplu, pentru a verifica bitul cel mai puțin semnificativ a variabilei `flags`, efectuăm `flags & 1`.

Operatorul SAU „`|`” este folosit pentru a seta anumiți biți pe 1:

```
x = x | 12;
```

Setează pe 1 biții 2 și 3 din `x`, deoarece $12_{10} = 1100_2$.

Operațiile de deplasare `<<` și `>>` realizează deplasări la stânga, respectiv la dreapta pentru operandul din stânga lor, cu numărul de poziții dat de operandul din dreapta lor. Deplasarea la stânga va provoca umplerea locurilor libere din dreapta cu zero. Deplasarea la dreapta va umple biții liberi din stânga cu zero. Spre exemplu: $4 << 1 = 12$ și $12 >> 2 = 2$, deoarece $2_{10} = 10_2$, $4_{10} = 100_2$ și $12_{10} = 1100_2$.

Operatorul unar „`~`” furnizează complementul față de 1 al unui întreg, adică convertește fiecare bit de 1 în 0 și viceversa.

2.3.7. Operatori compuși de asignare

Majoritatea operatorilor binari permit atașarea unui operator de asignare de forma `operator=: +=, -=, *=, /=, <<=, >>=, &=, |= și ^=`. Astfel, expresii de genul `i = i + 1` pot fi simplificate prin `i += 1`.

Funcția de mai jos calculează numărul de biți setați pe 1 dintr-un număr întreg:

```
int bitcount(unsigned n)
{
    int i;
    for(i = 0; n != 0; n >>= 1)
        if(n & 1)
            i++;
    return i;
}
```

2.3.8. Ponderea operatorilor și ordinea lor de evaluare

În tabelul de mai jos, operatorii aflați pe aceeași linie au pondere egală; liniile tabelului sunt în ordinea descrescătoare a ponderilor:

Operator	Modul de evaluare
<code>() [] -></code>	de la stânga la dreapta
<code>! ~ ++ -- (tip) * & sizeof (operatori unari)</code>	de la dreapta la stânga
<code>* / %</code>	de la stânga la dreapta
<code>+ -</code>	de la stânga la dreapta
<code><<>></code>	de la stânga la dreapta
<code><=&t;= >>=</code>	de la stânga la dreapta
<code>== !=</code>	de la stânga la dreapta
<code>&</code>	de la stânga la dreapta
<code>^</code>	de la stânga la dreapta
<code> </code>	de la stânga la dreapta
<code>&&</code>	de la stânga la dreapta
<code> </code>	de la stânga la dreapta
<code>? :</code>	de la dreapta la stânga
<code>= += -= etc.</code>	de la dreapta la stânga

2.4. Funcții

Funcțiile reprezintă un set de instrucțiuni grupate într-un bloc de program. Ele permit împărțirea programelor în mai multe subroutines mai mici. Funcțiile ascund detalii de implementare ale unor operații specifice, pe care programatorul nu trebuie neapărat să le cunoască. Astfel, funcțiile ușurează munca de scriere a codului, facilitează lucru în echipă și introduc ideea de reutilizare a codului (această idee permite programatorilor să utilizeze munca altora, fără a începe totul de la capăt). Funcțiile pot fi localizate fizic și în fișiere diferite, reducând în acest mod dimensiunile codului-sursă. De asemenea, funcțiile pot să rezide în biblioteci (cum ar fi cele standard).

Funcțiile pot primi un număr nelimitat de argumente, pentru fiecare în parte trebuind declarat tipul și numele. Numele este necesar pentru a putea accesa respectivul argument din cadrul funcției. Argumentele funcției sunt trimise prin valoare, adică funcția apelată primește o copie temporară, locală, a fiecărui argument și nu adresa sa. Ca urmare, funcția nu poate afecta valoarea originală a argumentului din funcția apelantă. Atunci când argumentul este un tablou (e.g. un sir de caractere), este trimisă adresa de început a tabloului, deci apelul se face prin referință, iar elementele nu sunt copiate. Există de asemenea posibilitatea ca o funcție să primească un număr variabil de argumente (vezi subcapitolul 6.2).

Funcțiile pot returna o valoare al cărei tip trebuie să preceadă numele funcției. Dacă o funcție nu returnează nici o valoare, tipul declarat trebuie să fie `void`. Instrucțiunea `return` specifică ieșirea din funcție și ea poate fi urmată de o valoare, dacă este cazul. Nu este obligatoriu ca funcțiile `void` să apeleze instrucțiunea `return` la sfârșitul acestora.

În exemplul următor vom utiliza funcția `lungime_sir`, definită mai sus:

```
#include <stdio.h>
int lungime_sir(char *s)
{
    int i = 0;
    while(s[i++] != '\0');
    return i;
}
void main()
{
    char s[1000];
    printf("Introduceti sirul de caractere: ");
    gets(s);
    printf("Sirul are %d caractere.\n", lungime_sir(s));
}
```

Pentru a putea utiliza o funcție, aceasta trebuie mai întâi declarată. În cadrul declarației, numele argumentelor poate lipsi:

```
int lungime_sir(char *);
```

În mod uzual, declarația unei funcții care este definită în alt fișier se face într-un fișier-antet (*header*), care poate fi inclus în orice fișier-sursă care utilizează funcția respectivă. Fișierul-antet poate conține și declarații de variabile, constante etc.:

functii.h

```
int lungime_sir(char *);
```

global.c

```
#include "functii.h"
int lungime_sir(char *s)
{
    int i = 0;
    while(s[i++] != '\0');
    return i;
}
```

main.c

```
#include <stdio.h>
/* Linia de mai jos poate fi înlocuită cu
   #include "global.c" */
#include "functii.h"

void main()
{
    char s[1000];
    printf("Introduceti sirul de caractere: ");
    gets(s);
    printf("Sirul are %d caractere.\n", lungime_sir(s));
}
```

Compilarea acestor fișiere se va face cu:

```
gcc -o test global.c main.c
```

2.5. Tablouri

Un tablou multidimensional este practic o matrice. Declararea unui asemenea tablou se face cu:

```
tablou[dimensiune1][dimensiune2]...[dimensiuneN]
```

Un tablou multidimensional se initializează printr-o listă de valori inițiale scrise între acolade. Pentru exemplificare vom implementa o funcție care returnează numărul de zile ale unei luni:

```
int nr_zile(int luna, int an)
{
    int tabzile[2][12] = {
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
        { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
    };

    int bisect = __an % 4 == 0 && __an % 100 != 0 || __an % 400 == 0;
    return tabzile[bisect][__luna - 1];
}
```

2.6. Enumerații

Tipul enumerare enum reprezintă o modalitate de a utiliza nume în loc de numere, pentru un tip comun de date:

```
enum figura_geometrica {
    cerc,
    patrat,
    triunghi
};
```

Numele cerc, patrat și triunghi le vor fi asignate în mod automat valorile 0, 1 respectiv 2. Numerele corespunzătoare numelor sunt considerate întregi. Pentru a asigna alte valori, se folosește o construcție de forma:

```
#include <stdio.h>

enum figura_geometrica {
    cerc = 10, patrat = 20, triunghi = 30
};
```

```
main()
{
    enum figura_geometrica g = triunghi;
    printf("Valoarea lui g = %d\n", g);
}
```

2.7. Comentarii

Comentariile reprezintă anumite observații introduse de programator, ele fiind ignorate la compilare.

Începutul unui comentariu este marcat prin /*, iar sfârșitul acestuia prin */. Comentariile pot cuprinde mai multe linii, ca în exemplul de mai jos:

```
/* Comentariu
   care ocupa
   mai multe linii */
```

Pot fi folosite și comentarii imbricate, însă acestea nu sunt permise de către unele compilatoare.

Capitolul 3

UTILIZAREA DATELOR

Nu-ți fă probleme, mă voi gândi la o soluție.
(Indiana Jones)

Un *bloc local* este o porțiune de program C încadrată de paranteza deschisă „{” și de paranteza închisă „}”. O funcție C conține aceste două paranteze, aşadar reprezintă și ea un bloc local.

Un bloc local poate apărea oriunde în cadrul programului. Variabilele declarate în interiorul unui bloc sunt vizibile doar în cadrul acestuia. Altfel spus, viața unei variabile începe odată cu declararea acesteia și se încheie odată cu sfârșitul blocului. Variabilele cu aceeași nume declarate într-un bloc local au precedență față de variabilele declarate în afara blocului.

3.1. Domeniul variabilelor

Unul dintre punctele forte ale limbajului C este flexibilitatea sa în definirea stocării datelor. Sunt două aspecte care pot fi controlate în C: domeniul și timpul de viață a variabilelor. *Domeniul unei variabile* se referă la locurile în care variabila poate fi accesată. *Timpul de viață* se referă la momentul în care variabila poate fi accesată.

Există trei domenii ale variabilelor:

- **extern** – implicit pentru variabilele declarate în afara oricărei funcții. Domeniul variabilei declarate extern este întregul program;
- **static** – domeniul unei variabile declarate static în afara oricărei funcții este restul programului în respectivul fișier-sursă. Domeniul unei variabile declarate static în interiorul unei funcții este restul blocului local;
- **auto** – implicit pentru variabilele declarate în afara unei funcții. Domeniul este restul blocului local.

Timpul de viață al unei variabile **extern** sau **static** durează dinainte ca funcția `main()` să fie apelată și se sfîrșește la ieșirea din program. Timpul de viață al argumentelor funcțiilor durează până la ieșirea din funcție (apelul instrucțiunii

`return`). Timpul de viață al unui variabilă alocate dinamic (pointer) începe odată cu alocarea spațiului de memorie prin apelul `malloc()` și se încheie odată cu apelul funcției `free()` sau la încheierea execuției programului.

O variabilă globală care trebuie să poată fi accesată din mai multe fișiere trebuie declarată într-un *fișier-antet*. O asemenea variabilă trebuie definită doar într-un singur fișier-sursă. Variabilele nu trebuie definite în fișiere-antet.

Declararea unei variabile desemnează precizarea tipului său către compilator, dar nealocarea de spațiu de memorie pentru ea. Definirea variabilei înseamnă declararea acesteia și alocarea de spațiu de memorie pentru ea:

```
extern int i; /* declaratie */
int n;          /* definitie */
```

3.2. Conversii de tip

Atunci când intr-o expresie apar operanzi de tipuri diferite, aceștia sunt convertiți în mod automat într-un tip comun.

În cadrul expresiilor aritmetice, conversia operanzilor se face după următoarele reguli:

Orice operand care nu este de tip `int` sau `double` este convertit astfel:

Tip	Se convertește la tipul
<code>char</code>	<code>int</code>
<code>unsigned char</code>	<code>int</code>
<code>signed char</code>	<code>int</code>
<code>short</code>	<code>int</code>
<code>enum</code>	<code>int</code>
<code>float</code>	<code>double</code>

1. Dacă unul dintre operanzi este de tip `double`, atunci și celălalt este convertit la `double`.
2. În caz contrar, dacă unul dintre operanzi este de tip `unsigned long`, celălalt este convertit și el la `unsigned long`.
3. În caz contrar, dacă unul dintre operanzi este de tip `long`, celălalt este convertit la tipul `long`.

4. În caz contrar, dacă unul dintre operanzi este de tip `unsigned`, celălalt este convertit la `unsigned`.
5. În caz contrar, ambii operanzi sunt de tipul `int`.

Modificarea tipului unui operand (*cast*) dintr-o expresie poate fi făcută și de către programator. Prima utilizare a acestei tehnici este modificarea tipului unui operand dintr-o operație aritmetică astfel încât operația să fie efectuată corect, atunci când compilatorul nu poate decide corect o conversie de tip.

A doua utilizare este modificarea tipului de pointeri către sau dinspre `void *`, pentru a realiza interfață cu funcțiile care au ca argument sau returnează pointeri de tip `void`. De exemplu:

```
struct student *s = (struct student *)malloc(sizeof(struct student));
```

Capitolul 4

CONTROLUL FLUXULUI

*Ferește-te de programatorii
care au șurubelnițe!
(anonim)*

Instrucțiunile de control al fluxului dintr-un limbaj specifică ordinea în care se fac calculele.

4.1. Instrucțiuni

În limbajul C, orice instrucțiune se încheie cu „;”, caracter considerat terminator de instrucțiuni.

Acoladele, „(” și „)” sunt folosite pentru a grupa instrucțiuni și declarații, construcție echivalentă practic cu o singură instrucțiune. După acolada închisă care încheie un bloc nu se pune „;”.

4.2. if-else

Instrucțiunea `if-else` se utilizează pentru luarea condițională de decizii. Sintaxa ei este:

```
if(expresie)
    instructiune1
else
    instructiune2
```

unde secțiunea `else` este optională. Dacă expresia din paranteza de după `if` este adevărată (adică are o valoare nenulă), este executată `instructiune1`. În caz contrar, este executată `instructiune2`:

```
if(i == 1)
    printf("i egal cu 1\n");
else
    return;
```

Deoarece secțiunea `else` este optională, se poate ajunge la ambiguități atunci când avem de-a face cu structuri `if-else` imbricate. În aceste cazuri, este necesară utilizarea accoladelor:

```
if(i > 0)
{
    if(x > y)
        res = x;
}
else
    res = y;
```

Construcția `if-else` poate fi înlocuită (simplificată) prin utilizarea operatorului ternar „`? :`”, în expresia:

```
e1 ? e2 : e3
```

Se evaluatează mai întâi expresia `e1`. Dacă aceasta este adevărată, atunci se evaluatează expresia `e2` și aceasta este valoarea expresiei condiționale. În caz contrar, se evaluatează `e3` și aceasta va fi rezultatul expresiei condiționale. Exemplul următor calculează maximul dintre două numere:

```
max = (a > b ? a : b);
```

4.3. switch

Instrucția `switch` se utilizează pentru a lua decizii multiple, testând dacă o expresie se potrivește cu una dintr-un număr de valori constante și executând corespunzător anumite instrucții. Programul următor contorizează tipurile de caractere dintr-un text introdus de la tastatură:

```
main()
{
    int c, nr_spatii = 0, nr_cifre = 0, nr_car = 0;

    while((c = getchar()) != EOF)
        switch(c)
    {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
```

```
        case '6':
        case '7':
        case '8':
        case '9':
            nr_cifre++;
            break;
        case ' ':
        case '\t': /* tab */
        case '\n': /* linie nouă */
            nr_spatii++;
            break;
        default: /* alte caractere decat cele enumerate mai sus */
            nr_car++;
            break;
    }
    printf("Am intalnit %d cifre, %d spatii si %d alte caractere\n",
           nr_cifre,
           nr_spatii,
           nr_car);
}
```

Piecare caz trebuie să fie „etichetat” cu o constantă întreagă. Dacă un caz se potrivește cu valoarea expresiei din paranteză, execuția este transferată la cazul respectiv, după întâlnirea primului `break` trecându-se la execuția următoarei instrucții de după `switch`. Cazul denumit `default` este executat atunci când nici unul dintre cazuri nu a fost întâlnit. Instrucția `break` provoacă ieșirea din cadrul `switch`, fără a mai verifica și celelalte cazuri.

4.4. while

Sintaxa instrucției `while` este următoarea:

```
while (expresie)
    instructiune
```

Dacă expresia specificată între paranteze este adevărată (nenulă), este executată instrucția. După execuție expresia este evaluată și din nou ciclul continuă până când valoarea expresiei devine falsă (nulă). Exemplul următor va decrementa valoarea contorului i până când aceasta devine 0:

```
while(i > 0)
    i--;
```

4.5. for

Sintaxa instrucției for este:

```
for(e1; e2; e3)
    instructiune
```

Expresia *e1* este evaluată o singură dată, la apelul instrucției for. Dacă expresia *e2* este adevărată (nenulă), este executată *instructiune* și apoi evaluată expresia *e3*, după care ciclul este reluat până când *e2* devine falsă (nulă). De obicei, *e1* și *e3* sunt expresii de asignare, iar *e2* o expresie relațională. Oricare dintre expresii poate fi omisă, dar caracterele „;” trebuie să rămână. Dacă *e2* este omisă, ea este considerată întotdeauna adevărată. Instrucția break în cadrul *instructiune* provoacă ieșirea imediată din ciclul for. Instrucția for este echivalentă cu:

```
e1;
while(e2)
{
    instructiune
    e3;
}
```

Astfel, instrucția de mai jos decrementează variabila *i* până când aceasta devine 0:

```
for(i = 5; i > 0; i--)
    ...
```

În cadrul instrucției for poate fi utilizat un operator special, virgula „,”. Astfel, o succesiune de expresii separate prin virgulă este evaluată de la stânga la dreapta și tipul și valoarea rezultatului sunt tipul și valoarea operandului cel mai din dreapta. Funcția prezentată în exemplul următor inversează caracterele dintr-un sir:

```
void reverse(char *s)
{
    int c, i, j;

    for(i = 0, j = strlen(s) - 1; i < j; i++, j--)
    {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Înăță mai jos o implementare pentru funcția atoi, care convertește un sir de caractere într-un număr întreg:

```
int atoi(char *s)
{
    int i, n, semn = 1;

    /* ignora spatiile de la inceputul sirului */
    for(i = 0; s[i] == ' ' || s[i] == '\t' || s[i] == '\n'; i++)
        ;
    if(s[i] == '+' || s[i] == '-')
        /* verifica semnul */
        semn = (s[i] == '+') ? 1 : -1;
    /* calculeaza partea intreaga */
    for(n = 0; s[i] >= '0' && s[i] <= '9'; i++)
        n = 10 * n + s[i] - '0';

    return semn * n;
}
```

4.6. do-while

Instrucția do-while este similară cu while, dar condiția de terminare este evaluată la sfârșitul ciclului și nu la începutul acestuia:

```
do
    instructiune
while(expresie);
```

4.7. break și continue

Am întâlnit instrucția break în paragrafele precedente. Așa cum am văzut, aceasta determină încheierea imediată a ciclului (buclei).

Instrucția continue are un efect diametral opus lui break, în sensul că face să înceapă următoarea iterație a ciclului. Pentru while și do-while aceasta reprezintă o nouă evaluare a expresiei, iar pentru for înseamnă reinițializarea ciclului, începând cu reevaluarea expresiei *e1* (vezi supra).

4.8. *goto*

Instrucțiunea *goto* efectuează un salt forțat la un anumit punct din program, denumit *etichetă* (*label*). Deoarece C este considerat un limbaj structurat, nu este recomandată folosirea acestei instrucțiuni, practic fiind posibilă în totdeauna evitarea ei. Ca exemplu, *goto* este folosită în cadrul nucleului Linux pentru a forța compilatorul să genereze un cod mai eficient.

```
for(i = 0; i < 10; i++)
    if(x - i < 0)
        goto ok;
ok:
...
```

Capitolul 5

POINTERI

*Aceasta este închisă în memoria mea,
și singur trebuie să găsești cheia.
(Shakespeare, Hamlet, Act I, Scena V)*

Prin definiție, un *pointer* este o variabilă care conține adresa unei alte variabile. Pointerii sunt foarte utilizați în C pentru că simplifică implementarea unor numeroase probleme și conduc la generarea de cod obiect mai compact și mai eficient.

Este posibilă adresarea variabilei conținute de pointer în mod indirect. Presupunem că avem variabila *x* de tip *int*, iar *px* un pointer la tipul *int*. Operatorul unar „*&*” furnizează adresa la care se află o variabilă. Instrucțiunea:

```
px = &x;
```

atribuie variabilei *px* adresa la care este memorată variabila *x*. Vom spune că *px* *pointează pe x*.

Operatorul unar „***” returnează conținutul variabilei memorate la adresa respectivă. În cazul nostru, instrucțiunea:

```
x = *px;
```

atribuie variabilei *x* valoarea memorată la adresa *px*. Declarația:

```
int *px;
```

declară că pointerul *px* memorează o valoare de tip *int*.

Înțial, conținutul variabilei de tip pointer *px* este vid. Pentru a putea folosi această variabilă, trebuie mai întâi alocat spațiul de memorie necesar, cu ajutorul funcției de bibliotecă *malloc()*. Această funcție primește ca argument dimensiunea spațiului de memorie care se dorește a fi alocat și returnează adresa de unde începe zona de memorie alocată, în caz de succes, sau un pointer cu valoarea *NULL* (practic 0) atunci când alocarea nu a reușit. Trebuie menționat aici faptul că această funcție returnează o variabilă de tip *void **, adică un pointer de tip general, *void*. Pentru a se evita generarea de atenționări (engl. *warnings*) de către

compilator, este necesară utilizarea operatorului *cast* pentru modificarea acestui tip în tipul pointerului `px` atunci când se dorește ca `px` să primească drept valoare adresa zonei alocate cu `malloc()`:

```
px = (int *)malloc(2);
```

Operațiunea descrisă mai sus se numește *alocare dinamică*.

Atunci când nu se cunoaște mărimea tipului de variabilă (cum ar fi în cazul unei structuri, vezi capitolul 7), trebuie utilizată instrucțiunea `sizeof`:

```
double *d;
d = (double *)malloc(sizeof(double));
```

Eliberarea zonei de memorie ocupată de conținutul unui pointer se face cu ajutorul funcției `free()`. Eliberarea acestor zone de memorie se face în mod automat la ieșirea din program.

```
free(d);
```

Pointerii rezolvă și anumite inconveniente în folosirea parametrilor unei funcții, deoarece aceștia sunt transmiși prin valoare, nefiind posibilă modificarea adreselor memorate prin intermediul lor de către funcție. Trimînd ca parametri variabile de tip pointer, adresele spre care pointează aceștia nu pot fi schimbate, dar conținutul lor poate fi modificat fără nici o restricție. Funcția din exemplul următor interchimbă valorile parametrilor primiți:

```
void swap(int *px, int *py)
{
    int tmp;
    tmp = *px;
    *px = *py;
    *py = tmp;
}
```

De asemenea, utilizarea pointerilor rezolvă problema că o funcție nu poate returna decât o valoare, iar programatorul dorește ca funcția să poată returna mai multe. Această dilemă se poate rezolva trimînd ca argumente funcției pointeri al căror conținut va fi modificat de funcție cu valorile care practic se doresc și să fie returnate.

Se observă incrementarea lui `s`, operațiune corectă cu un pointer, care nu afectează conținutul acestuia, incrementând adresa `s`, aceasta memorând o copie

5.1. Tablouri și pointeri

În C, există o relație foarte strânsă între pointeri și tablouri, după cum vom observa în cele ce urmează.

Un tablou poate fi utilizat ca și cum ar fi un pointer, iar un pointer poate fi indexat ca și cum ar fi un tablou. Valoarea unei variabile de tip tablou reprezintă adresa primului element al tabloului:

```
int t[20];
int *pt;
int x;
```

În urma instrucțiunii:

```
pt = &t[0];
pt = t;
```

`pt` va pointa la primul element al tabloului `t`. Cele două atribuiri sunt echivalente. Instrucțiunea:

```
x = *pt;
```

atribuie lui `x` valoarea primului element al tabloului, iar:

```
x = *(pt + 1);
```

atribuie lui `x` valoarea celui de-al doilea element al tabloului. Dacă `pt` este un pointer, el poate fi accesat cu ajutorul indicilor sub forma `pt[i]`, ceea ce este echivalent cu `*(pt + i)`.

Pentru a ilustra mai bine aceste noțiuni de așa-zisă *aritmetică a pointerilor*, vom descrie funcția `lungime_sir()` care calculează lungimea unui șir de caractere:

```
int lungime_sir(char *s)
{
    int n;
    for(n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

locală în cadrul funcției a parametrului actual de la apelul funcției. De remarcat că notația `char *s` este echivalentă cu `char s[]`. Dacă `p` este un pointer, atunci `p++` desemnează următorul „obiect” pointat, indiferent de tipul acestuia.

Pentru a evidenția ușurința programării cu pointeri, vom implementa o funcție pentru copierea unui sir de caractere într-un alt sir, furnizând mai întâi varianta fără pointeri, iar apoi varianta cu pointeri:

```
void my strcpy1(char s[], char t[])
{
    int i = 0;

    while((s[i] = t[i]) != '\0')
        i++;
}

void my strcpy2(char *s, char *t)
{
    while(*s++ = *t++)
}
```

5.2. Tablouri de pointeri

Pointerii pot fi utilizați și în *tablouri de pointeri*. Spre exemplu, un tablou de pointeri poate constitui un text, fiecare linie a acestuia fiind reprezentată ca un pointer la primul său caracter. Vom prezenta în continuare un program de sortare lexicografică a acestor linii. Pointerii simplifică această operațiune, deoarece atunci când două linii neordonate trebuie inversate, vor fi inversați doar pointerii corespunzători în tablou, nu și conținutul liniilor de text:

functii.h:

```
/*
compara cele două siruri primite ca argumente
returneaza zero daca sunt identice, un numar pozitiv daca s1>s2 si
un numar negativ daca s1<s2
*/
int strcmp(char *s1, char *s2);

/*
returneaza lungimea sirului primit ca argument
*/
int strlen(char *s);
```

```
/*
citeste de la tastatura linii de text, cel mult numarul specificat,
pana cand intalneste EOF, memorandu-le in argumentul text
returneaza numarul de linii introduse
*/
int readlines(char **text, int max);

/*
sorteaza liniile textului primit ca argument
returneaza zero in caz de succes, -1 in caz de eroare
*/
int sort(char **text, int n);
```

functii.c:

```
#include <stdio.h>

int strcmp(char *s1, char *s2)
{
    for(; *s1 == *s2; s1++, s2++)
        if(*s1 == '\0')
            return 0;

    return (*s1 - *s2)
}

int strlen(char *s)
{
    int n;

    for(n = 0; *s != '\0'; s++)
        n++;

    return n;
}

int readlines(char **text, int max)
{
    int len, nlines = 0;
    char s[255];

    while(nlines < max && (len = strlen(gets(s))) > 0)
    {
        if((text[nlines++] = (char *)malloc(len)) == NULL)
            /* daca alocarea de memorie a esuat */
            return -1;
        strcpy(text[nlines++], s);
    }

    return nlines;
}
```

```

int sort(char **text, int n)
{
    int i, j, dist;
    char *tmp;

    for(dist = n / 2; dist > 0; dist /= 2)
        for(i = dist; i < n; i++)
            for(j = i - dist; j >= 0; j -= dist)
            {
                if(strcmp(text[j], text[j + dist]) <= 0)
                    break;
                tmp = text[j];
                text[j] = text[j + dist];
                text[j + dist] = tmp;
            }
}

void printlines(char **text, int n)
{
    int i = 1;

    while(--n >= 0)
        printf("%d: %s\n", i, *text++, i++);
}

```

main.c:

```

#include <stdio.h>
#define MAX_LINES 1000

int main()
{
    char *text[MAX_LINES];
    int i, nlines;

    nlines = readlines(text, MAX_LINES);
    sort(text, nlines);
    printlines(text, nlines);

    return 0;
}

```

5.3. Pointeri la funcții

În C, există posibilitatea de a defini un pointer la o funcție, care poate fi manipulat ca orice altă variabilă. Pentru exemplificare, vom modifica programul de sortare a liniilor unui text, prezentat mai sus, pentru a putea sorta orice fel de date, nu doar siruri de caractere. Algoritmul de sortare va fi independent de operațiile de comparare și de inter-schimbare a liniilor de text, astfel încât, prin transmiterea acestor două funcții ca argumente, se va putea realiza sortarea pe diferite criterii.

```

#include <stdio.h>
#define MAX_LINES 1000

int nrcmp(char *s1, char *s2)
{
    double n1, n2;

    n1 = atof(s1);
    n2 = atof(s2);
    if(n1 < n2)
        return -1;
    else
        if(n1 > n2)
            return 1;

    return 0;
}

void schimba(char **s1, char **s2)
{
    char *tmp;

    tmp = *s1;
    *s1 = *s2;
    *s2 = tmp;
}

int sort(char **text, int n, int (*compara)(), (*schimba)())
{
    int i, j, dist;

    for(dist = n / 2; dist > 0; dist /= 2)
        for(i = dist; i < n; i++)
            for(j = i - dist; j >= 0; j -= dist)
            {
                if((*compara)(text[j], text[j + dist]) <= 0)
                    break;
                (*schimba)(&text[j], &text[j + dist]);
            }
}

```

```

int main(int argc, char **argv)
{
    char *text[MAX_LINES];
    int i, nlines, numeric = 0; /* 1 daca valorile sunt numerice */
    if(argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n')
        numeric = 1;
    nlines = readlines(text, MAX_LINES);
    if(numeric)
        sort(text, nlines, nrcmp, schimba);
    else
        sort(text, nlines, strcmp, schimba);
    printlines(text, nlines);
    return 0;
}

```

Declarația `int (*compara)()` desemnează că argumentul `compara` este un pointer la o funcție care returnează un `int`. Declarația `(*schimba)()` indică un pointer la o funcție care returnează un `void`.

Capitolul 6

STRUCTURAREA PROGRAMELOR: FUNCTIILE

*Natura utilizează doar cele mai lungi căi
pentru a-și țese şablonul, astfel că fiecare mică bucată
de ţesătură dezvăluie organizarea întregii tapiserii.*
(Richard Feynman)

6.1. Recursivitatea

În limbajul C, funcțiile pot fi apelate recursiv. Aceasta înseamnă că o funcție se poate apela pe ea însăși. Atunci când o funcție se autoapelează, fiecare apel generează alte instanțe ale variabilelor declarate în interiorul funcției. Recursivitatea este utilă în special pentru prelucrarea structurilor de date recursive, cum ar fi arborii sau grafurile.

Trebuie avut în vedere faptul că utilizarea de funcții recursive duce la consum ridicat de memorie, deoarece pentru fiecare apel de funcție se alocă spațiu pentru stiva funcției precum și pentru variabilele declarate în interiorul ei.

Exemplul următor implementează funcția `calcul`, care decrementează un număr întreg până când acesta devine zero:

```

#include <stdio.h>

void calcul(int *contor)
{
    if(*contor == 0)
        return;
    else
    {
        (*contor)--;
        calcul(contor);
    }
}

int main()
{
    int i;

    printf("Introduceti numarul de cicluri: ");
    scanf("%d", &i);
}

```

```

calcul(&i);
printf("Valoarea lui i este: %d\n", i);

return 0;
}

```

6.2. Funcția `main()`

S-a putut observa în multe exemple de până acum apelul `main()`. Aceasta nu este altceva decât o funcție specială, care definește programul principal. Apelul acestei funcții începe o dată cu execuția programului și încheiează la apelul instrucțiunii `return` din cadrul funcției `main()` sau la încheierea programului.

În mod normal, funcția `main()` trebuie să returneze un număr întreg, reprezentând rezultatul execuției programului. În general, aceasta este zero pentru o execuție normală.

Funcția `main()` poate primi argumente de forma:

```
int main(int argc, char **argv)
```

unde `argc` este numărul de parametri primiți de program la apelarea acestuia, iar `argv` un tablou de siruri de caractere (vezi capitolul 5) conținând parametri. Primul element al tabloului, `argv[0]`, reprezintă numele executabilului asociat programului, inclusiv calea completă în care acesta se află. Programul de mai jos va afișa toți parametrii primiți în linia de comandă.

```

main(int argc, char **argv)
{
    int i;

    for(i = 1; i < argc; i++)
        printf("Parametrul %d: %s\n", i, argv[i]);
}

```

Presupunând că fișierul executabil corespunzător acestui program este denumit `afisparam`, apelul:

```
afisparam param1 param2
```

va afișa la ieșirea standard:

```
Parametrul 1: param1
Parametrul 2: param2
```

Capitolul 7

STRUCTURI

*Regula este: îl faci cât de simplu poți,
dar nu mai simplu de-atât.*
(Albert Einstein)

O structură este o colecție de date de tipuri diferite. O structură se declară utilizând sintaxa:

```
struct nume_structura {
    declaratii membri
    ...
}
```

De exemplu:

```
struct student {
    char *nume;
    char *prenume;
    char *adresa;
    char *telefon;
}
```

Pentru a declara o variabilă de tip `student` și a o inițializa:

```
struct student s = { "Buraga", "Sabin", "Iasi, str. Pacurari",
    "0744123456" };
```

Membrii structurii pot fi accesati prin construcții de forma:

```
nume_structura.nume_membru
```

Spre exemplu:

```
printf("Nume si prenume: %s %s\n", s.nume, s.prenume);
```

În mod evident, pot fi declarati și pointeri la structuri. Pentru accesarea membrilor trebuie folosit în acest caz operatorul `->`:

```
struct student *s;
s = (struct student *)malloc(sizeof(struct student));
s->nume = (char *)malloc(10);
strcpy(s->nume, "Buraga");
```

O structură poate conține și membri de tip structură:

```
struct data {
    int zi, luna, an;
};

struct student {
    char *nume;
    char *prenume;
    char *adresa;
    char *telefon;
    struct data data_nastere;
} s;
```

Ultima declarație, cea a structurii student, conține și o definiție, cea a variabilei s de tip student. Membrii structurii data se accesează astfel:

```
s.data_nastere.zi
```

Manipularea structurilor este supusă unui număr de restricții. Astfel, structurile nu pot fi asignate sau copiate ca un întreg, elementele acestora trebuind copiate individual de către programator. Este permisă utilizarea structurilor în cadrul argumentelor funcțiilor, precum și ca tip de dată returnată de o funcție:

```
struct student citire_student();
int verificare_student(struct student *s);
```

Conținutul unei variabile de tip structură poate fi inițializată astfel:

```
struct student s = { "Buraga", "Sabin", "Iasi", "0744123456",
{ 07, 01, 1974 } };
```

7.1. Structuri cu autoreferire

Autoreferirea desemnează posibilitatea ca o structură să aibă ca membru un pointer la o structură de același tip cu ea:

```
struct nod {
    int val;
    struct nod *stanga, *dreapta;
};

#include <stdio.h>
#include <string.h>

struct nod {
    char *linie;
    struct nod *stanga, *dreapta;
};

struct nod *inserare_linie(struct *rad, char *sir)
{
    struct nod *nou;

    if(rad == NULL)
    {
        /* alocam memorie pentru un nod */
        if((nou = (struct nod *)malloc(sizeof(struct nod))) == NULL)
            /* daca alocarea de memorie a esuat */
            return -1;

        /* alocam memorie pentru sir */
        if((nou->linie = (char *)malloc(strlen(sir))) == NULL)
            /* daca alocarea de memorie a esuat */
            return -1;

        strcpy(rad->linie, sir);
        rad->stanga = rad->dreapta = NULL;
    }
    else
    {
        if(strcmp(sir, rad->linie) < 0)
            rad->stanga = inserare_line(rad->stanga, sir);
        else
            rad->dreapta = inserare_line(rad->dreapta, sir);
    }

    return rad;
}

void afiseaza_linie(struct nod *rad)
{
    if(rad->stanga)
```

```

    afiseaza_linie(rad->stanga);
    printf("%s\n", rad->linie);
    if(rad->dreapta)
        afiseaza_linie(rad->dreapta);
}

int main()
{
    int len;
    char s[256];
    struct nod *rad = NULL;

    while((len = strlen(fgets(s, 255, stdio))) > 0)
        rad = inserare_linie(rad, s);
    afiseaza_linie(rad);

    return 0;
}

```

7.2. *typedef*

Într-un mod similar cu macrodefinițiile, limbajul C permite crearea unor nume noi pentru diferite tipuri de date, denumite *typedef*. Astfel, declarația:

```
typedef char* SIR;
```

face ca numele de tip SIR să fie sinonim pentru `char *`. Vom utiliza *typedef* pentru structura arborelui de mai sus:

```

typedef struct nod {
    char *linie;
    struct nod *stanga, *dreapta;
} NOD_ARBORE;

```

Declarațiile *typedef* sunt utile în primul rând pentru tipuri de date care pot fi dependente de platformă, modificările trebuind efectuate doar asupra declarației *typedef*, și în al doilea rând pentru a crește claritatea unui program atunci când în cadrul acestuia intervin structuri mai complicate de date.

Capitolul 8

PREPROCESORUL C

Un algoritm trebuie să fie văzut pentru a putea fi crezut.
 (Donald Knuth)

C permite utilizarea unor extensii de limbaj prin intermediul *macrourilor*. Aceste macrouri sunt analizate înainte de compilare, în faza de *preprocesare*.

Includerea fișierelor-antet

Pentru includerea fișierelor-antet (dar nu exclusiv) se pot utiliza următoarele două macrouri:

```
#include "fisier"
#include <fisier>
```

Practic, un astfel de macrou va fi înlocuit în faza de preprocesare cu conținutul fișierului specificat. Prima variantă a macroului face presupunerea că fișierul specificat se găsește în directorul curent, iar cea de-a doua variantă cauta fișierul în căile cunoscute de compilator (în cazul GCC, folosind opțiunea `-I`).

Macrosubstituțiile

Macrosubstițuirea desemnează înlocuirea unui nume cu un sir de caractere. Aceasta se poate realiza prin definiții de forma:

```
#define MAX 100
```

Atunci când în codul-sursă va fi întâlnit cuvântul MAX, acesta va fi înlocuit cu 100. Se pot defini de asemenea și macrouri cu argumente, ca în exemplul de mai jos, care calculează maximul dintre două numere:

```
#define max(x, y) (x > y ? y : x)
```

Compilare condiționată

O expresie de forma:

```
#if constanta
```

verifică dacă respectiva constantă este non-zero. O expresie de forma:

```
#ifdef simbol
```

verifică dacă simbolul este definit (printr-o directivă `#define`). În mod similar, o expresie de genul:

```
#ifndef simbol
```

verifică dacă simbolul nu este definit. Toate aceste trei directive preprocesor pot fi urmate de linii de cod-sursă care pot fi următoare optional de `#else` și apoi obligatoriu de `#endif`. În cazul în care condiția este adevărată, liniile de cod dintre `#else` și `#endif` sunt ignorate. În cazul în care condiția este falsă, liniile dintre directivă și `#else` sunt ignorate.

Echivalentă cu expresia `#ifdef simbol` este construcția:

```
#if defined(simbol)
```

Această formă este mai permisivă, în sensul că sunt valide exprimări de genul:

```
#if defined(simbol1) || defined(simbol2)
```

Capitolul 9

BIBLIOTECILE STANDARD C

scansare.

(Al. Viro pe lista linux-kernel)

După cum am mai spus, operațiunile de intrare/ieșire sunt implementate în biblioteca standard C și sunt accesibile prin intermediul funcțiilor din cadrul acesta. Pentru a putea folosi această bibliotecă, trebuie inclus fișierul-antet `stdio.h` printr-o directivă `#include <stdio.h>`. O serie de funcții au fost utilizate în programele prezentate în cadrul acestei părți a lucrării de față.

Ieșirea formatată: `printf`

Sintaxa funcției `printf` este următoarea:

```
printf(const char *format, arg1, arg2, ...)
```

`printf` convertește, formatează și tipărește la ieșirea standard valorile argumentelor `arg1`, `arg2`, ..., conform formatului specificat prin primul argument. Acest format conține caractere obișnuite, care vor fi copiate la ieșire, și caractere de control, care determină modul de conversie și tipărirea următoarelor argumente. Fiecare specificație de conversie este introdusă prin caracterul „%” și încheiată printr-un caracter de conversie.

Între „%” și caracterul de conversie pot fi:

- un semn minus „-”, care semnifică alinierea la stânga a argumentului convertit;
- un număr care specifică lungimea minimă a sirului. Argumentul va fi aliniat la dreapta și va fi completat cu spații albe până la lungimea câmpului. Dacă dimensiunea câmpului a fost prefixată cu un „0”, caracterul de completare va fi „0”.

Caracterele de conversie sunt următoarele:

- d argumentul este convertit în zecimal;
- o argumentul este convertit în octal fără semn;
- x argumentul este convertit în hexazecimal fără semn;
- u argumentul este convertit în zecimal fără semn;

- c argumentul este considerat ca fiind un singur caracter;
 - s argumentul este un sir de caractere. Caracterele din cadrul argumentului corespunzător acestui format de conversie sunt preluate până la întâlnirea caracterului \0 sau până când este atinsă lungimea specificată;
 - f argumentul este considerat în virgulă mobilă, implicit cu 6 cifre după virgulă.
- Asemănătoare cu funcția printf sunt funcțiile:
- sprintf(char *sir, const char *format, arg1, arg2, ...)
 - trimite ieșirea către sirul de caractere specificat;
 - fprintf(FILE *fisier, const char *format, arg1, arg2, ...)
 - trimite ieșirea către identificatorul de fișier specificat (vezi *infra*).

Intrarea formatată: scanf

scanf citește caractere de la intrarea standard, le interpretează conform formatului specificat și le memorează în argumentele următoare, care practic sunt pointeri care indică adresele unde vor fi memorate datele convertite. Sintaxa funcției scanf este următoarea:

```
int scanf(const char *format, arg1, arg2, ...)
```

Formatul primului argument este identic cu cel descris la funcția printf. Funcția scanf returnează numărul de „obiecte” citite de la intrare. Important de remarcat este faptul că spațiul de memorie alocat pentru pointerii trimiși ca argumente trebuie să fie destul de mare pentru a putea memora datele citite.

```
#include <stdio.h>

int main()
{
    int x, y;

    printf("Introduceti cele doua valori: ");
    scanf("%d %d", &x, &y);
    printf("%d + %d = %d\n", x, y, x + y);

    return 0;
}
```

Asemănătoare cu funcția scanf sunt funcțiile:

- sscanf(char *sir, const char *format, arg1, arg2, ...)
- citește date din sirul de caractere specificat;
- fscanf(FILE *fisier, const char *format, arg1, arg2, ...)
- citește date din identificatorul de fișier specificat (vezi *infra*).

Funcții pentru citirea și scrierea de caractere

- int getchar() citește un caracter de la intrarea standard;
- int fgetc(FILE *f) citește un caracter din fișierul specificat;
- int putchar(int c) scrie un caracter la ieșirea standard;
- int fputc(int c, FILE *f) scrie un caracter în fișierul specificat.

Funcții pentru citirea și scrierea de siruri de caractere

- char *gets(char *s) citește un sir de caractere de la intrarea standard, memorându-l în argumentul s. Nu face vreo verificare asupra dimensiunii spațiului alocat sirului s;
- char *fgets(char *s, int max, FILE *f) citește din fișierul f un sir de caractere a cărui lungime nu va depăși numărul maxim de caractere, specificat ca argument;
- int puts(char *s) scrie un sir de caractere la ieșirea standard;
- int fputs(char *s, FILE *f) scrie un sir de caractere în fișierul f.

Lucrul cu fișiere

Înainte de a putea citi sau scrie într-un fișier, acesta trebuie deschis cu ajutorul funcției fopen. Această funcție returnează un pointer la o structură de tip FILE, care conține informații despre fișierul deschis, utilizate intern de către bibliotecă:

```
FILE *fopen(const char *nume_fisier, const char *mod)
```

Modul de deschidere a fișierului se alege în funcție de modul în care se intenționează a fi utilizat:

- r doar citire;
- w scriere;
- a adăugare.

Dacă fișierul nu există și este deschis pentru scriere sau adăugare, acesta va fi creat. Dacă se deschide un fișier existent pentru scriere, conținutul vechi va fi șters. Dacă la deschidere apare o eroare, funcția fopen va returna valoarea NULL.

Închiderea unui fișier atunci când nu mai este folositor se realizează cu ajutorul funcției fclose, care primește ca argument pointerul la FILE care specifică fișierul respectiv. Conținutul acestuia va fi eliberat.

În momentul începerii execuției unui program C, sunt deschise în mod automat trei fișiere: intrarea standard (stdin), ieșirea standard (stdout) și ieșirea

standard pentru erori (stderr). Acestea pot fi folosite ca orice alt fișier, dar nu pot fi deschise sau închise (ele sunt practic constante).

Vom realiza în cele ce urmează un program care simulează comanda cat, pentru concatenarea a două fișiere:

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

void copie(FILE *f)
{
    int c;

    while((c = fgetc(f)) != EOF)
        putc(c, stdout);
}

int main(int argc, char **argv)
{
    FILE *f;

    if(argc == 1)/* nu am primit argumente, utilizam intrarea standard */
        copie(stdin);
    else
        /* va fi afisat intai continutul fisierului precizat prin argv[1], apoi continutul fisierului argv[2] */
        while(--argc > 0)
            if((f = fopen(++argv, "r")) == NULL)
            {
                fprintf(stderr, "cat: nu pot deschide fisierul '%s'\n",
                        *argv);
                return -ENOENT; /* fisierul nu exista */
            }
            else
            {
                copie(f);
                fclose(f);
            }

    return 0;
}
```

Vom prezenta mai jos un program care extrage fișierele conținute într-o arhivă tar:

```
/*
untar v1.0
Dezarchiveaza o arhiva tar furnizata ca parametru
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
    FILE *in, *out;
    char buff[256];
    char *currdir, *name;
    char c;
    unsigned long i, currpos = 0;

    printf("untar v1.0 -- Dezarchiveaza o arhiva tar\n\n");
    if(argc != 2)
    {
        printf("Numar invalid de parametri.\n");
        printf("Mod de utilizare: untar fisier\n");
        return -1;
    }
    if((in = fopen(argv[1], "r")) == NULL)
    {
        printf("untar: nu pot deschide fisierul '%s'\n");
        return -1;
    }
    while(!feof(in))
    {
        fseek(in, currpos, SEEK_SET);
        i = 0;
        /* determina numele fisierului sau directorului continut in arhiva */
        while((c = fgetc(in)) != '\0')
            buff[i++] = c;
        if(i == 0)
            break;
        if(buff[i - 1] == '/')
        {
            buff[i - 1] = '\0';
            currdir = buff;
            /* daca este un director, il creez */
            if(mkdir(currdir, 0755))
                printf("untar: nu pot crea directorul '%s'\n", currdir);
            else
                printf("%s/\n", currdir);
            strcat(currdir, "/");
        }
        else
        {
            buff[i] = '\0';
            name = buff;
            printf("%s\n", name);
            /* daca este un fisier, il scriu pe disc */
        }
    }
}
```

```

if((out = fopen(name, "w")) == NULL)
{
    printf("untar: nu pot deschide fisierul '%s' pentru
           citire\n", name);
    fclose(in);
    return -1;
}
currpos += 0x200;
fseek(in, currpos, SEEK_SET);
while((c = fgetc(in)) != '\0')
{
    if(c == EOF)
    {
        printf("untar: sfarsit neasteptat de fisier\n");
        fclose(in);
        fclose(out);
        return -1;
    }
    fputc(c, out);
}
fclose(out);
currpos = ftell(in);
while((currpos % 0x200) != 0)
{
    currpos++;
    currpos -= 0x200;
}
currpos += 0x200;
fclose(in);
return 0;
}

```

Întrucât scopul lucrării de față nu este învățarea limbajului C, recomandăm cititorului să parcurgă anexele 1 și 2, care conțin lista tuturor funcțiilor de bibliotecă prevăzute de standardele ANSI și POSIX, precum și resursele bibliografice.

Capitolul 10

PROBLEME PROPUSE

1. Să se implementeze funcția `char *elimina_caracter(const char *s, const char c)`, care elimină toate aparițiile caracterului `c` din sirul `s`.
2. Să se implementeze funcția `double atof(const char *s)`, care convertește un sir de caractere într-un număr de tip `double` (vezi implementarea funcției `atoi()` în paragraful 4.5).
3. Să se modifice programul prezentat în subcapitolul 5.2 pentru sortarea unor linii de text, astfel încât să sorteze liniile de text primite de la intrarea standard.
4. Să se scrie un program care să definească structura pentru un arbore și să implementeze operațiile uzuale pentru aceasta (creare, căutare, stergere, inserare, parcurgere). Nodurile arborelui vor conține elemente de tip `int`.
5. Să se modifice programul precedent pentru lucrul cu arbori, pentru a putea salva, respectiv încărca informațiile într-un fișier. Nodurile arborelui vor conține elemente de tip sir de caractere.
6. Să se scrie o variantă simplificată a utilitarului `bc`, adică un program care să aștepte de la tastatură expresii de forma `operand_stanga operator operand_dreapta` și să afișeze rezultatul operației, acceptând operatorii `+, -, /, ^` și `^`.
7. Să se implementeze o variantă a comenzi `grep` care caută într-un fișier pe baza unor criterii un subșir de caractere, liniile găsite fiind afișate. Opțiunile suportate vor fi:
`-v` afișează doar liniile care nu satisfac criteriul;
`-c` afișează numai numărul liniilor găsite, nu și liniile;
`-i` ignoră literele mici de cele mari.
Subșirul de căutat va putea include următoarele meta-caractere:
 - `“*”` înlocuiește mai multe caractere;
 - `“?”` înlocuiește un singur caracter.
8. Utilizând structura definită în capitolul 7, să se realizeze un program care să permită evidența studenților dintr-o facultate. Programul trebuie să implementeze și salvarea, respectiv încărcarea datelor de pe disc.
9. Să se scrie un program denumit `aat`, care să aibă sintaxa `aat [timp comanda | delete]`. Acesta va executa comanda la timpul specificat (poate fi de forma `AAAALLZZOOMM`). Dacă este menționată opțiunea `delete`, va fi ștersă planificarea comenzi la momentul specificat. Fără parametri, vor fi afișate comenziile planificate și executate.

10. Să se implementeze un program care simulează comanda `tar`, care să gestioneze un fișier-arhivă care include mai multe fișiere. Se vor implementa opțiunile:

- c creează o arhivă care va îngloba conținutul fișierelor din directorul curent;
- x dezinchidează arhiva, restaurând fișierele conținute de aceasta;
- t listează conținutul arhivei.

Pentru simplificarea programului, nu se va căuta recursiv în subdirectoare. De asemenea, în arhivă vor fi memorate doar numele și dimensiunea fișierelor.

Partea a doua

PROGRAMAREA ÎN LIMBAJUL C++

Capitolul 1

SCURT ISTORIC

Începutul este momentul în care trebuie acordată cea mai mare atenție corectitudinii echilibrelor.
(Frank Herbert, *Dune*)

Așa cum am văzut în partea I a acestei cărți, C este un limbaj procedural, structurat. Creșterea complexității programelor a făcut însă necesară elaborarea altor tipuri de limbaje. Astfel, destinate inteligenței artificiale, au apărut limbajele care au la bază noțiunea de „cadru” și cele care pleacă de la ideea de „actor”. Primele implementează operații asupra unor *modele de entități*; celelalte presupun faptul că *obiectele* nu sunt simple elemente pasive asupra cărora se fac anumite prelucrări, ci, dimpotrivă, că menirea acestor *obiecte* constă în a realiza prelucrările asupra lor înlătărită.

De aici a pornit ideea de a grupa structurile de date cu operațiile care prelucrează respectivele date. Astfel s-au născut noțiunile de *obiect* și de *clăsă*. Proiectarea de programe utilizând clase se numește *programare orientată-obiect* (*OOP, Object Oriented Programming*).

Primele limbaje orientate-obiect au fost *SIMULA* (1965) și *SIMULA-2* (1967). În anii '70 a apărut și celebrul limbaj *SMALLTALK*. Cel mai mare dezavantaj al acestora a fost faptul că au fost proiectate ca limbaje de sine stătătoare, având o răspândire relativ redusă. Din acest motiv, puțini programatori erau dispuși în acea vreme să renunțe la limbajele consacrate doar pentru a lucra obiectual.

În anul 1980, Bjarne Stroustrup a conceput limbajul „*C with Classes*” („*C cu clase*”). Acest limbaj a dus la îmbunătățirea C-ului prin adăugarea unor noi facilități, printre care și lucrul cu clase. În vara lui 1983, C-with-classes a pătruns și în lumea academică și a instituțiilor de cercetare. Astfel, acest limbaj a putut să evolueze datorită experienței acumulate de către utilizatorii săi. Denumirea finală a acestui limbaj a fost *C++*.

Succesul extraordinar de care se bucură limbajul C++ a fost asigurat de faptul că a extins cel mai popular limbaj al momentului, C. Programele scrise în C funcționează și în C++, și ele pot fi transformate în C++ cu eforturi minime.

Cele mai recente etape în evoluția acestui limbaj sunt limbajele *JAVA*, realizat de firma *SUN*, și *C#*, propus de *Microsoft*, care nu sunt altceva decât limbaje C++ puțin modificate și extinse.

Interesantă este următoarea afirmație a lui Stroustrup: „*utilizatorii au început să folosească C++ înainte ca specialiștii să aibă timpul necesar să-i instruiască pentru a-l folosi cu randament maxim*”. Într-adevăr, s-a constatat că o mare parte dintre compilatoarele de C++ existente nu sunt folosite decât pentru dezvoltarea de software structurat, și nu orientat-obiect (într-un spus, se lucrează în C pe un compilator de C++).

Programatorii au descoperit ulterior că aplicațiile orientate-obiect sunt mai ușor și mai rapid de scris, și nu în ultimul rând mai ușor de înțeles.

Capitolul 2

NOTIUNILE DE BAZĂ ALE PROGRAMĂRII ORIENTATE-OBIECT

Procesul programării are mai multe faze:

se începe cu definirea conceptelor,

după care se trece la stabilirea relațiilor dintre ele.

Abia după aceea se poate trece la scrierea codului.

(Bjarne Stroustrup)

2.1. Premisele limbajelor orientate-obiect

În ultimii ani, programarea orientată-obiect a devenit foarte populară, mai ales datorită avantajelor sale utile dezvoltării proiectelor actuale, ce devin din ce în ce mai complexe. Acest stil de programare propune împărțirea aplicațiilor în mai multe module, astfel încât cel ce dezvoltă un modul nu trebuie să cunoască detaliile de implementare a altor module.

Trebuie totodată să amintim că programarea orientată-obiect este un concept foarte natural. În lumea înconjurătoare, zi de zi, în orice moment, avem de-a face cu *Obiecte*. În jurul nostru se găsesc o multitudine de obiecte, *interconectate*, comunicând unele cu altele într-un fel sau altul.

Domeniul în care acest stil de programare s-a dovedit cel mai util este dezvoltarea interfețelor utilizator și a aplicațiilor bazate pe acestea.

Programarea structurată este bazată pe ecuația enunțată de Niklaus Wirth:

Structuri de date + Algoritmi = Program

Programarea structurată a fost o etapă ce a trebuit depășită, deoarece este deficitară în ceea ce privește posibilitatea reutilizării programelor, scalabilității și extinderii unor module de program, atribute de neînlătuit în realizarea aplicațiilor complexe. Principala deficiență a programării structurate constă în tratarea separată a algoritmilor și a structurilor de date ce se prelucră. De obicei, în natură, o entitate este caracterizată atât prin structură, cât și printr-un anumit comportament. În mod normal, obiectele evoluează în timp, adeseori modificându-și structura și funcționalitatea.

2.2. Concepte fundamentale

Ideea de bază de la care pleacă programarea orientată-obiect este de a grupa structurile de date cu operațiile care prelucrează respectivele date. Un asemenea ansamblu poartă denumirea de *clasă*. Proiectarea de programe utilizând clase se numește *programare orientată-obiect (OOP)*.

În mod frecvent, pentru structurile de date se utilizează denumirea de *date membre* sau *câmpuri*, iar pentru procedurile ce prelucrează aceste date, termenul de *funcții membre* sau *metode*.

În analogie cu ecuația programării structurate, se poate considera ca valabilă următoarea relație:

$$\text{Date} + \text{Metode} = \text{Clasă}$$

Această relație este bazată pe principiul fundamental al *încapsulării datelor*, conform căruia accesul la datele membre se poate face numai prin intermediul setului de metode asociat. Acest principiu determină o abstractizare a datelor în sensul că o clasă este caracterizată complet de specificațiile metodelor sale, detaliile de implementare fiind ascunse utilizatorului. Acest aspect este hotărâtor în cazul proiectelor complexe, de dimensiuni mari, care nu pot fi realizate decât cu ajutorul unor echipe de programatori. Aplicațiile pot fi împărtite cu ușurință în module, astfel încât cel ce dezvoltă un modul nu trebuie să cunoască detaliile de implementare a celoralte module. Consecințele imediate sunt scăderea timpului de dezvoltare a aplicațiilor, simplificarea activității de întreținere a modulelor și creșterea calității programelor.

Privind limbajele orientate-obiect ca pe o evoluție a limbajelor structurate, constatăm că noțiunea de clasă este o generalizare a noțiunii de structură de date. O clasă descrie un ansamblu de obiecte similare. Un obiect este aşadar o variabilă de un anumit tip. În mod uzual, se folosește exprimarea că *un obiect este instanțierea unei clase*.

Un alt concept important în cadrul programării orientate-obiect este cel de *polimorfism*, care se referă la posibilitatea de a opera cu mai multe variante ale unei funcții, care efectuează o anumită operărie specifică pentru anumite obiecte.

Evoluția și ierarhizarea claselor de obiecte se bazează pe conceptul de *moștenire*. Astfel, procedeul numit *derivare* permite definirea unei noi clase (*clasa derivată*) pornind de la o clasă existentă (*clasa de bază*), prin adăugarea de noi date și metode membre, eventual prin redefinirea unor metode ale clasei de bază. Clasa derivată moștenește de la clasa de bază structura de date și metodele aferente. Este posibilă totodată și derivarea unei clase din mai multe clase de bază, această operărie fiind denumită *moștenire multiplă*.

Așadar, dintr-o clasă de bază pot fi derivate mai multe clase și fiecare clasă derivată poate deveni la rândul ei o clasă de bază pentru alte clase derivate. Se poate astfel realiza o *ierarhie de clase*, care să modeleze sisteme complexe. Construirea ierarhiei de clase constituie activitatea fundamentală de realizare a unei aplicații orientate-obiect, reprezentând în fapt fază de proiectare a respectivului sistem.

În capitolele următoare vom discuta despre implementarea acestor concepte în limbajul C++.

Capitolul 3

CLASE

Limbajul C își permite să te împuști singur cu ușurință în picior; C++ face acest lucru mai dificil, însă atunci când se întâmplă îți spulberă întreg piciorul.
 (Bjarne Stroustrup)

3.1. Declararea claselor

O sintaxă simplificată a declarării unei clase este următoarea:

```
class NumeClasa
{
  ...
  declaratii variabile membre
  ...
  declaratii functii membre
  ...
}
```

După cum se poate observa din această alcătuire a declarației, clasa este asemănătoare cu o structură din limbajul C, dar care poate avea în componență să membri atât de tip variabilă, cât și de tip funcție. Așa cum spuneam și în capitolul precedent, pentru datele din interiorul clasei se utilizează de obicei termenul de *date membre*, iar pentru funcții, denumirea de *funcții membre* sau *metode*. Spunem aşadar că o clasă permite *încapsularea* în interiorul său a datelor și a funcțiilor membre.

Întocmai ca în limbajul C, pentru a putea utiliza efectiv un tip de date (în cazul de față o clasă), trebuie să definim o variabilă de acel tip. Într-un mod similar declarației:

```
int i;
```

putem scrie:

```
NumeClasa variabila
```

Vom considera de acum încolo că *variabila* poartă numele de *obiect*. Exprimarea ușuală este că *un obiect este o instanță a unei clase*.

3.2. Membrii unei clase

Accesarea membrilor unei clase se face ca în cazul structurilor din limbajul C:

```
obiect.VariabilaMembra = valoare
```

pentru *accesul* la o variabilă membră, și:

```
obiect.FunctieMembra()
```

pentru *apelarea* unei funcții membre.

Pentru exemplificare să considerăm o implementare a noțiunii de punct. Ca variabile membre avem nevoie doar de coordonatele x și y care definesc poziția în spațiu a unui punct. Declaram o funcție care calculează aria dreptunghiului având colțurile (0, 0) și (x, y).

```
class Point
{
  unsigned x, y;
  unsigned long Arie()
  {
    return x * y;
  }
  unsigned GetX();
  unsigned GetY();
  void SetX(unsigned X);
  void SetY(unsigned Y);
};

unsigned Point::GetX()
{
  return x;
}

unsigned Point::GetY()
{
  return y;
}

void Point::SetX(unsigned X)
{
  x = X;
}
```

```
void Point::SetY(unsigned Y)
{
    y = Y;
}
```

Am folosit un operator nou, specific C++, `::`, numit *operator de rezoluție*, numit și *operator de acces* sau de *domeniu*. El permite accesul la un identificator, dintr-un bloc în care acesta nu este vizibil datorită unei alte declarații locale. Un exemplu de folosire este următorul:

```
char *sir = "variabila globala";
void functie()
{
    char *sir = "variabila locala";
    printf("%s\n", ::sir); // afiseaza variabila globala
    printf("%s\n", sir); // afiseaza variabila locala
}
```

Pentru definițiile funcțiilor membre aflate în afara declarației clasei este necesară specificarea numelui clasei urmat de acest operator înaintea numelui fiecarei funcții, indicând faptul că funcția are același domeniu ca și declarația clasei respective și este membră a ei, deși este definită în afara declarației.

O întrebare care poate apărea se leagă de motivul pentru care am realizat funcțiile `GetX()`, `GetY()`, `SetX()`, `SetY()`, când putem utiliza direct variabilele membru `x` și `y`. Răspunsul este furnizat de una din regulile programării C++, adoptată în general de către specialiști, care cere protecția variabilelor membru (a se vedea capitolul următor), acestea neputând fi accesate decât prin intermediul unor funcții, care au rolul de *metode de prelucrare a datelor încapsulate în interiorul clasei*. Această regulă reprezintă principiul fundamental al *încapsulării datelor*.

3.3. Crearea și distrugerea obiectelor

Să considerăm următorul program C++:

```
void main()
{
    Point p;
```

În momentul definirii variabilei `p`, va fi alocat automat spațiul de memorie necesar, acesta fiind eliberat la terminarea programului. În exemplul de mai sus,

variabila `p` este de tip static. În continuare vom modifica acest program pentru a folosi o variabilă dinamică (pointer).

```
void main()
{
    Point *p;

    p = new Point;
    p->x = 5;
    p->y = 10;
    printf("Aria = %d\n", p->Aria());
    delete p;
}
```

Se poate observa utilizarea unor operatori pe care nu îi cunoaștem din limbajul C: `new` și `delete`. C++ introduce o metodă nouă pentru gestiunea dinamică a memoriei, similară celei utilizate în C (`malloc()` și `free()`), dar superioară și special construită pentru programarea orientată-obiect. Operatorul `new` este folosit pentru alocarea memoriei, iar sintaxa acestuia este:

```
variabila = new tip;
variabila = new tip(valoare_initiala);
variabila = new tip[n];
```

Este ușor de intuit că prima variantă alocă spațiu pentru *variabilă* dar nu o initializează, a doua variantă îi alocă spațiu și o initializează cu valoarea specificată, iar a treia alocă un tablou de dimensiune *n*. Acest operator furnizează ca rezultat un pointer conținând adresa zonei de memorie alocate, în caz de succes, sau un pointer cu valoarea *NULL* (practic 0), atunci când alocarea nu a reușit.

Eliminarea unei variabile dinamice și eliberarea zonei de memorie aferente se realizează cu ajutorul operatorului `delete`. Sintaxa acestuia este:

```
delete variabila;
```

Deși acești doi operatori oferă metode flexibile de gestionare a obiectelor, există situații în care aceasta nu rezolvă toate problemele (de exemplu, obiectele care necesită alocarea unor variabile dinamice în momentul creării lor). De aceea, pentru crearea și distrugerea obiectelor în C++ se folosesc niște funcții membre speciale, numite *constructori* și *destructori*.

Constructorul este apelat automat la instantierea – fie statică, fie dinamică – a unei clase.

Destructorul este apelat automat la eliminarea unui obiect, adică la încheierea timpului de viață în cazul static, sau la apelul unui `delete` în cazul dinamic.

Din punct de vedere cronologic, constructorul este apelat după alocarea memoriei necesare, deci în faza finală a creării obiectului, iar destructorul înaintea eliberării memoriei aferente, deci în faza inițială a distrugerii sale.

Constructorii și destructorii se declară și se definesc similar cu celelalte funcții membre, dar prezintă o serie de caracteristici specifice:

- numele lor coincide cu numele clasei căreia îi aparțin; destructorii se disting de constructori prin faptul că numele lor este precedat de caracterul „~”;
- nu pot returna nici un rezultat;
- nu se pot utiliza pointeri către constructori sau destructori;
- constructorii pot avea parametri, destructorii însă nu. Un constructor fără parametri poartă denumirea de *constructor implicit*.

De remarcat este faptul că în cazul în care o clasă nu dispune de constructori sau destructori, compilatorul de C++ generează automat un constructor, respectiv un destructor implicit.

Să completăm în continuare clasa *Point* cu un constructor și un destructor:

```
Point::Point()
// constructor implicit
{
    x = 0;
    y = 0;
}

Point::Point(unsigned x, unsigned y)
{
    x = x;
    y = y;
}

Point::~Point()
```

Se poate observa cu această ocazie și modul de marcarea a comentariilor în C++: tot ce se află după sirul // este considerat comentariu.

De notat este faptul că definiții de forma:

```
Point p;
```

sau

```
Point *p = new Point();
Point *p = new Point;
```

duc la apelarea constructorului implicit.

3.4. Conceptul de moștenire

Dacă întrebăm un zoolog ce este un câine, ne va răspunde că este un reprezentant al speciei *canine domesticus*. Un câine este un tip de carnivor, un carnivor este un tip de mamifer, și aşa mai departe. Zoologul împarte animalele în regnuri, clase, ordine, familii, genuri și specii. Această ierarhie stabilește o relație de genul „este un/este o”. Putem remarcă acest tip de relație oriunde în lume: Mercedes este un tip de mașină, care la rândul său este un tip de autovehicul, și exemplele pot continua. Astfel, când spunem că ceva este un tip de altceva diferit, spunem că este o specializare a aceluia lucru, aşa cum o mașină este un tip mai special de autovehicul.

Conceptul de câine *moștenește*, deci primește în mod automat, toate caracteristicile unui mamifer. Deoarece este un mamifer, cunoaștem faptul că se mișcă și respiră aer – toate mamiferele se mișcă și respiră aer prin definiție. Gîndindu-ne la un câine îl asociem totodată cu ideea de lătrat, cu mișcatul din coadă, și aşa mai departe. Putem clasifica mai departe câinii în câini de pază și câini de vânătoare, iar câinii de vânătoare, în cockeri și dobermani etc.:

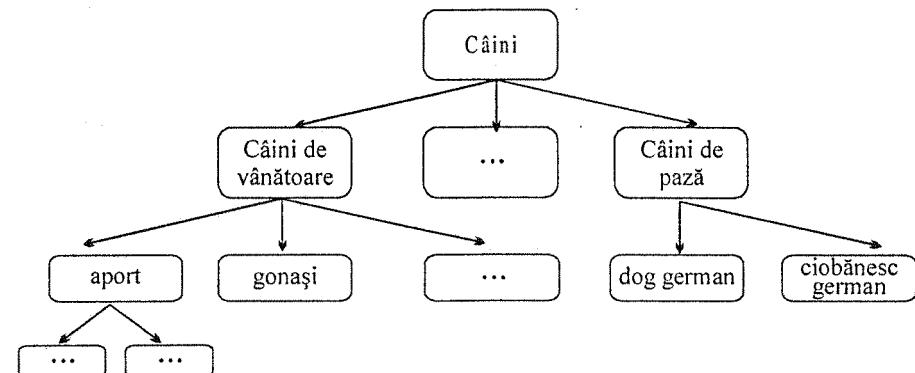


Figura 3.1. Ilustrare a conceptului de moștenire

Așa cum am văzut mai sus, conceptul de moștenire este o noțiune foarte naturală, pe care o întâlnim în viața de zi cu zi. În C++ întâlnim noțiunea de *derivare*, care este în fapt o abstractizare a noțiunii de moștenire. O clasă care adaugă proprietăți noi la o clasă deja existentă vom spune că este *derivată* din clasa originară. Clasa originară poartă denumirea de *clasă de bază*.

Clasa derivată moștenește toate datele și funcțiile membre ale clasei de bază (care nu sunt declarate ca *private* în aceasta); ea poate adăuga noi date la cele existente și poate suprascrie sau adăuga funcții membre. Clasa de bază nu este afectată în nici un fel în urma acestui proces de derivare și ca urmare nu trebuie recomplată. Declarația și codul obiect sunt suficiente pentru crearea clasei derivate, ceea ce permite reutilizarea și adaptarea ușoară a codului deja existent, chiar dacă fișierul-sursă nu este disponibil (spre exemplu, în cazul unei biblioteci). Astfel, nu este necesar ca programatorul unei clase derivate să cunoască modul de implementare a funcțiilor membre din componenta clasei de bază.

O noțiune nouă legată de derivare este cea de *supraîncarcare* sau *suprascrivere* a funcțiilor membre. Aceasta se referă, în mod evident, la redefinirea unor funcții ale clasei de bază în clasa derivată. De notat este faptul că funcțiile originale din clasa-părinte sunt în continuare accesibile în clasa derivată, deci caracteristicile clasei de bază nu sunt pierdute.

Dintr-o clasă de bază pot fi derivate mai multe clase și fiecare clasă derivată poate servi mai departe ca bază pentru alte clase derivate. Se poate astfel realiza o *ierarhie de clase*, care să modeleze adekvat sisteme complexe. Pornind de la clase simple și generale, fiecare nivel al ierarhiei acumulează caracteristicile claselor-părinte și le adaugă un anumit grad de specializare. Mai mult decât atât, în C++ este posibil ca o clasă derivată să moștenească simultan proprietățile mai multor clase de bază, procedură numită *moștenire multiplă*. Construirea ierarhiei de clase reprezintă activitatea fundamentală de realizare a unei aplicații orientate-obiect, reprezentând în fapt fază de proiectare a respectivului sistem.

Sintaxa simplificată a derivării este:

```
class NumeClasaDerivata : NumeClasaDeBaza
```

În continuare vom deriva din clasa Point o clasă specializată, GraphicPoint, care va „ști” să deseneze efectiv punctul pe ecran:

```
class GraphicPoint : Point
{
    unsigned color;
    GraphicPoint(unsigned X, unsigned Y, unsigned Color);
    ~GraphicPoint();
    void Draw();
    void SetX(unsigned X);
    void SetY(unsigned Y);
};

GraphicPoint::GraphicPoint(unsigned X, unsigned Y, unsigned Color) :
    Point(X, Y)
{
    color = Color;
}

GraphicPoint::~GraphicPoint()
```

```
}
```

```
GraphicPoint::Draw()
{
    // ...
    // apelare primitive grafice pentru desenarea efectiva a punctului
}
```

```
GraphicPoint::SetX(unsigned X)
{
    Point::SetX(); // apelul functiei SetX() apartinand clasei de baza
    Draw();
}
```

```
GraphicPoint::SetY(unsigned Y)
{
    Point::SetY();
    Draw();
}
```

Se observă din exemplul de mai sus că am adăugat o variabilă nouă față de clasa Point, color, pentru a putea memora culoarea cu care se face desenarea punctului. De asemenea, am suprascris constructorul și destructorul clasei-părinte. În constructorul derivat, am apelat constructorul original folosind construcția:

```
ClasaDerivata::ClasaDerivata() : ClasaDeBaza()
```

În clasa GraphicPoint am adăugat o funcție membră nouă, Draw(), care desenează efectiv punctul pe ecran. Am suprascris funcțiile SetX() și SetY(), apelând în ambele funcțiile originare, utilizând sintaxa:

```
ClasaDeBaza::FunctieMembra()
```

apelând apoi funcția de desenare Draw().

Regulile de funcționare a constructorilor și destructorilor, descrise în paragraful precedent, rămân valabile și în cazul claselor derivate, cu două observații privind ordinea de apelare a acestora:

- la instantierea clasei derivate se apelează mai întâi constructorul clasei de bază, apoi se apeleză propriul constructor;
- la distrugerea unui obiect al unei clase derivate, este apelat mai întâi propriul constructor, și apoi destructorul clasei de bază (deci în ordine inversă celei de la crearea obiectului).

Capitolul 4

PROGRAMARE AVANSATĂ UTILIZÂND CLASE

*Dacă un general nu este curajos,
el nu va putea să biruie incertitudinile
sau să realizeze planuri bune.*
(Shen Pao-hsu)

4.1. Controlul accesului la clase

Spre deosebire de limbajele orientate-obiect pure, C++ permite controlul accesului la membrii claselor. În acest scop, s-au creat trei specificatori de control al accesului:

- **public:** membrul poate fi accesat de orice funcție din domeniul declarației clasei;
- **private:** membrul este accesibil numai funcțiilor membre și prietene ale clasei;
- **protected:** similar cu private, însă accesul se extinde și la funcțiile membre și prietene ale claselor derivate.

De remarcat este faptul că o funcție membră a unei clase are acces la toți membrii clasei, indiferent de specificatorul de acces.

Așadar, sintaxa declarației unei clase derivate incluzând controlul accesului este:

```
class NumeClasaDerivata : SpecifierAcces NumeClasaDeBaza
```

unde *SpecifierAcces* poate fi **public** sau **private**.

Atributul din clasa de bază	Modificator de acces	Accesul moștenit de clasa derivată	Accesul din exterior
private protected public	private private private	inaccesibil private private	inaccesibil inaccesibil inaccesibil
private protected public	public public public	inaccesibil protected public	inaccesibil inaccesibil accesibil

Se observă că pentru a oferi clasei derivate acces la un membru al clasei de bază, acesta trebuie declarat **protected** sau **public**. Pentru respectarea principiului încapsulării datelor, datele membre pentru care se oferă acces claselor derivate se declară în clasa de bază cu atributul **protected**. De asemenea, pentru a conserva dreptul de acces în urma derivării, se utilizează derivarea **public**. Accesul poate fi stopat pe orice nivel al ierarhiei de clase printr-o derivare **private**.

Stabilirea atributelor de acces ale membrilor unei clase, precum și ale derivărilor, trebuie să se facă astfel încât dezvoltarea ierarhiei de clase să nu afecteze principiul încapsulării datelor.

Vom relua în continuare exemplul din capitolul precedent, completat cu specificatori de acces:

```
class Point
{
protected:
    unsigned x, y;
public:
    Point();
    Point(unsigned x, unsigned y);
    ~Point();
    unsigned long Arie();
    unsigned GetX();
    unsigned GetY();
    void SetX(unsigned X);
    void SetY(unsigned Y);
};

class GraphicPoint : public Point
{
    unsigned color;
public:
    GraphicPoint(unsigned X, unsigned Y, unsigned Color);
    ~GraphicPoint();
    void Draw();
    void SetX(unsigned X);
    void SetY(unsigned Y);
};
```

Se observă că variabilele membru **x** și **y** sunt declarate **protected**, așa încât vor fi vizibile și vor avea același atribut în clasa **GraphicPoint** (deși nu sunt utilizate). În mod normal, **x** și **y** ar trebui să fie declarați **private**, însă nu sunt utilizati decât în interiorul clasei **Point**. Funcțiile din **GraphicPoint** nu accesează acești doi membri direct, ci prin intermediul metodelor publice de accesare oferite de clasa **Point**.

Important de notat este faptul că, implicit, dacă nu este utilizat nici un specificator de acces, membrii sunt considerați **private**.

```
void main()
{
    Point *p;
    p = new Point;
    p->x = 5; // operatie imposibila: x este membru privat
    p->y = 8; // operatie imposibila: y este membru privat
    p->SetX(5); /* corect: acces la variabila x prin intermediul
                   functiei SetX() */
    p->SetY(8);
    printf("Aria = %d\n", p->Aria());
    delete p;
}
```

Am demonstrat prin acest exemplu de program că din exteriorul unei clase nu pot fi accesate datele membre **private** sau **protected**.

4.2. Funcții și clase prietene

În paragraful precedent, am afirmat că principiul încapsulării datelor este bine de respectat în cadrul elaborării ierarhiei de clase. Cu toate acestea, există situații în care este greu să se respecte acest principiu. De aceea, Bjarne Stroustrup a introdus un concept menit să rezolve și aceste situații particulare, pentru a oferi soluții elegante în vederea rezolvării tuturor situațiilor posibile. Acest concept este cel de **friend**, care permite practic abateri controlate de la ideea protejării datelor prin încapsulare. Este recomandat ca mecanismul **friend** să fie folosit **numai** în cazul în care nu există altă soluție!

Mecanismul **friend** (sau *prietenie*) a apărut datorită imposibilității de a face ca o metodă să fie membră a mai multe clase.

Funcțiile prietene sunt funcții care nu sunt metode ale unei clase, dar care au totuși acces la membrii privați ai acesteia. Orice funcție poate fi *prietenă* a unei clase, indiferent de natura acesteia.

Sintaxa declarării unei funcții prietene în cadrul declarației unei clase este următoarea:

```
friend NumeFunctie
```

Iată și un exemplu:

```
class Point {
    friend unsigned long Calcul(unsigned X, unsigned Y);
public:
    friend unsigned long AltaClasa::Calcul(unsigned X, unsigned Y);
```

```
; ...
unsigned long Calcul(unsigned X, unsigned Y)
{
    return X * Y / 2;
}

unsigned long AltaClasa::Calcul(unsigned X, unsigned Y)
{
    ...
}
```

După cum se vede din exemplul de mai sus, nu are nici o importanță în cadrul cărei secțiuni este declarată funcția prietenă.

Clasele prietene sunt clase care au acces la membrii privați ai unei clase. Sintaxa declarării unei clase prietene este:

```
friend class NumeClasaPrietena
```

Iată și un exemplu:

```
class PrimaClasa {
    ...
};

class ADouaClasa {
    ...
    friend class PrimaClasa;
};
```

În exemplul de mai sus, clasa PrimaClasa are acces la membrii privați ai clasei ADouaClasa.

Important este să remarcăm că *relația de prietenie nu este tranzitivă*. Dacă o clasă A este prietenă a clasei B, iar clasa B este prietenă a unei clase C, aceasta nu înseamnă că A este prietenă a clasei C. De asemenea, *proprietatea de prietenie nu se moștenește în clasele derivate*.

4.3. Cuvântul-cheie *this*

Toate datele membre ale unei clase primesc un parametru ascuns, pointer-ul **this**, care reprezintă adresa obiectului în cauză. Acesta poate fi utilizat în cadrul funcțiilor membre.

Iată și un exemplu:

```
unsigned long Point::Arie()
{
    return this->x * this->y;
}
```

4.4. Redefinirea operatorilor

După cum știți deja, C/C++ au definite mai multe tipuri de date: *int*, *char* etc. Pentru utilizarea acestor tipuri de date sunt definiți mai mulți operatori: adunare (+), înmulțire (*) etc. C++ permite programatorilor să își redefină (supraîncarce) acești operatori pentru a lucra cu propriile clase.

Sintaxa supraîncărcării unui operator este:

operator Simbol

unde *Simbol* este simbolul oricărui operator C++, exceptând: „.”, „.*”, „::”, „?:”. Această definire se face în cadrul clasei, întocmai ca pentru o funcție membră.

Operatorii supraîncărcați în interiorul unei clase vor respecta reguli de asociativitate similare celor pentru operatorii predefiniți, după cum se poate constata din tabelul de mai jos:

Tipul operatorului	Simbolul operatorului	Asociativitate	Observații
Binar	() [] ->	->	Se definesc doar ca funcții membre
Unar	+ - ~ * & (tip)	<-	
Unar	++ --	<-	
Unar	new delete	<-	Poate fi supradefinit și pentru o clasă
Binar	-> * / % + - & &&	->	
Binar	<< >> < <= > >= == !=	->	
Binar	= += -= *= /= %= &= ^= = <<= >>=	<- -	Se definesc doar ca funcții membre
Binar	,	->	

După cum vom vedea în continuare, există două variante de definire a operatorilor:

- ca funcții membre ale clasei;
- ca funcții prietene ale clasei.

Pentru exemplificare, ne propunem să extindem clasa Point cu utilizarea unor operatori:

```
class Point {
    Point& operator += (Point __p);
    Point& operator -= (Point __p);
    Point operator + (Point __p);
    Point operator - (Point __p);
    Point& operator = (Point __p);
    int operator == (Point __p);
    int operator != (Point __p);
    int operator < (Point __p);
    int operator > (Point __p);
    int operator <= (Point __p);
    int operator >= (Point __p);
};

Point& Point::operator += (Point __p)
{
    x += __p.x;
    y += __p.y;
    return *this;
}

Point& Point::operator -= (Point __p)
{
    x -= __p.x;
    y -= __p.y;
    return *this;
}

Point Point::operator + (Point __p)
{
    return Point(x + __p.x, y + __p.y);
}

Point Point::operator - (Point __p)
{
    return Point(x - __p.x, y - __p.y);
}

int Point::operator == (Point __p)
{
    return x == __p.x && y == __p.y;
}

int Point::operator != (Point __p)
```

```

{
    return !(*this == __p);
}

int Point::operator < (Point __p)
{
    return x < __p.x && y < __p.y;
}

int Point::operator > (Point __p)
{
    return x > __p.x && y > __p.y;
}

int Point::operator <= (Point __p)
{
    return x <= __p.x && y <= __p.y;
}

int Point::operator >= (Point __p)
{
    return x >= __p.x && y >= __p.y;
}

main()
{
    Point p1 = Point(5, 7), p2(1, 3);

    if(p1 > p2)
        p1 -= p2;
    else
        p2 -= p1;
}

```

Am utilizat mai sus varianta cu funcții membre. În continuare vom descrie implementarea operatorului + folosind cea de-a două variantă, cu funcții prietene.

```

class Point {
    ...
    friend Point operator + (Point __p1, Point __p2);
}

Point operator + (Point __p1, Point __p2)
{
    return Point(__p1.x + __p2.x, __p1.y + __p2.y);
}

```

Definirea operatorilor ca fiind funcții membre ale unei clase prezintă o restricție majoră: primul operand este obligatoriu de tipul clasă respectiv.

Supraredefinirea operatorilor este supusă în C++ unui set de restricții:

- nu este permisă introducerea de noi simboluri de operatori;
- patru operatori nu pot fi redefiniți (vezi *supra*);

- caracteristicile operatorilor nu pot fi schimbată: pluralitatea (nu se poate supraredefini un operator unar ca operator binar sau invers), precedența și asociativitatea;
- funcția operator trebuie să aibă *cel puțin* un parametru de tipul clasă căruia îi este asociat operatorul supraredefinit.

Programatorul are libertatea de a alege natura operației realizate de un operator, însă este recomandat ca noua operație să fie apropiată de semnificația inițială.

4.4.1. Redefinirea operatorului „=”

Operatorul „=” este deja predefinit în C++, inclusiv pentru operanzi de tip clasă. Dacă nu este supraredefinit, atribuirea se face membru cu membru, în mod similar cu initializarea obiectului efectuată de către compilator. Pot exista situații în care se dorește o atribuire specifică clasei, mai ales în cazul în care clasa are date membru de tip pointer, ca atare poate fi supraredefinit.

```

Point& Point::operator = (Point __p)
{
    x = __p.x;
    y = __p.y;
    return *this;
}

main()
{
    Point p1(4, 7), p2;

    p2 = p1;
}

```

4.4.2. Redefinirea operatorului []

Operatorul de indexare [] se folosește în cazul în care există ca membru al clasei un tablou, pentru a se returna elementul de un anumit indice al acestuia, definindu-se astfel:

```
int &operator[](int)
```

Iată un exemplu de utilizare a acestui operator:

```
#define MAX 10

class SirDeIntregi
{
```

```

int sir[MAX];
public:
    SirDeIntregi()
    {
        for(i = 0; i < MAX; i++)
            sir[i] = 0;
    }
    int &operator[](int i)
    {
        return sir[i + 1];
    }
};

main()
{
    SirDeIntregi s;
    s[1] = 2;
}

```

4.4.3. Redefinirea operatorilor *new* și *delete*

Acești doi operatori pot fi supradefiniți pentru a realiza operații specializate de alocare/eliberare dinamică a memoriei pentru obiectele unei clase, mai ales în cazul în care clasa are date membre de tip pointer. Funcția operator new trebuie să primească un argument de tipul `size_t` care să precizeze dimensiunea în octeți a obiectului alocat și să returneze un pointer de tip `void` conținând adresa zonei alocate:

```
void *operator new(size_t)
```

cu mențiunea că `size_t` este definit în `stdlib.h`. Chiar dacă parametrul de tip `size_t` este obligatoriu, calculul dimensiunii obiectului în cauză și generarea sa se fac de către compilator.

Funcția operator delete trebuie să primească ca prim parametru un pointer de tipul clasei în cauză sau `void`, conținând adresa obiectului de distrus, și un al doilea parametru, optional, de tip `size_t`. Funcția nu întoarce nici un rezultat.

```
void operator delete(void *, size_t)
```

Trebuie să menționăm aici că operatorii `new` și `delete` supradefiniți păstrează toate proprietățile operatorilor `new` și `delete` standard.

Exemplul de mai jos implementează operatorii `new` și `delete` folosind funcțiile clasice din C, `malloc` și `free`:

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

```

void* operator new(size_t marime)
{
    void *nou;
    if((nou = malloc(marime)) == NULL)
        printf("Nu am putut aloca %d octeti!\n", marime);
    return nou;
}

void operator delete(void *p)
{
    free(p);
}

main()
{
    char *s;
    s = new char(255);
    strcpy(s, "test");
    delete s;
}

```

4.4.4. Redefinirea operatorilor unari

Operatorii unari pot fi supradefiniți utilizând o funcție membră fără parametri sau o funcție prietenă cu un parametru de tipul clasă. Trebuie subliniat că pentru operatorii `++` și `--` dispare distincția dintre utilizarea ca prefix și cea ca postfix, de exemplu între `x++` și `++x`, respectiv `x--` și `--x`, după cum menționam și în tabela cu operatori de mai sus.

4.5. Moștenirea multiplă

Limbajul C++ permite crearea de clase care moștenesc proprietățile mai multor clase de bază. Moștenirea multiplă crește astfel flexibilitatea dezvoltării ierarhiei de clase. Dacă derivarea normală duce la construirea unei ierarhii de tip arbore, derivarea multiplă va genera ierarhii de tip graf.

Sintaxa completă pentru operația de derivare este următoarea:

```
class NumeClasaDerivata : ListaClaseDeBaza
```

unde *ListaClaseDeBază* este:

```
SpecificatorAcces NumeClasaDeBaza, ...
```

4.5.1. Clase virtuale

Utilizarea moștenirii multiple se poate complica o dată cu creșterea dimensiunii ierarhiei de clase. O situație care poate apărea este derivarea din două clase de bază, Clasa1 și Clasa2, care la rândul lor sunt deriveate dintr-o clasă comună, ClasaDeBaza. În acest caz, noua clasă, ClasaNouă, va conține datele membre ale clasei ClasaDeBaza duplicate. Dacă prezența acestor date duplicate este utilă, ele pot fi distinse evident cu ajutorul operatorului de rezoluție, `::`. Totuși, în cele mai multe cazuri, această duplicare nu este necesară și duce la consum inutil de memorie. De aceea, în C++ a fost creat un mecanism care să evite această situație, prin intermediul conceptului de *clasă virtuală*. Sintaxa este:

```
class NumeClasaDerivata : SpecificatorAcces virtual NumeClasaDeBaza
```

Această declarație nu afectează clasa în cauză, ci numai clasele deriveate din aceasta. Astfel, clasele Clasa1 și Clasa2 considerate vor fi declarate virtuale. Trebuie menționat faptul că declararea *virtual* a acestor clase va afecta definirea constructorului clasei ClasaNouă, deoarece compilatorul nu poate hotărî care date vor fi transferate către constructorul ClasaDeBaza, dintre cele specificate de constructorii Clasa1 și Clasa2 (putând exista, după cum am precizat, date redundante). Constructorul ClasaNouă va trebui modificat astfel încât să trimită datele pentru constructorul ClasaDeBaza. De asemenea, trebuie precizat că într-o ierarhie de clase deriveate, constructorul clasei virtuale este întotdeauna apelat primul.

```
class ClasaDeTest
{
public:
    virtual bool Test(int i)
    {
        return (i == 0);
    }
};

class ClasaDerivata
{
public:
    virtual bool Test(int i)
    {
        return (i == 1);
    }
};
```

```
main()
{
    ClasaDerivata c;
    c.Test(1);
}
```

4.6. Conversii de tip definite de programator

După cum știți, în C/C++ există definit un set de reguli de conversie pentru tipurile fundamentale de date. C++ permite definirea de reguli de conversie pentru clasele create de programator. Regulile astfel definite sunt supuse unor restricții:

- într-un sir de conversii nu este admisă decât o singură conversie definită de programator;
- se recurge la aceste conversii numai după ce se verifică existența altor soluții (de exemplu, pentru o atribuire, se verifică mai întâi supraîncărcarea operatorului de atribuire și în lipsa acestuia se face conversia).

Există două metode de a realiza conversii de tip, pe care le vom prezenta în cele ce urmează.

4.6.1. Supraîncărcarea operatorului *unar cast*

Sintaxa este:

```
operator TipData()
```

respectiv:

```
operator (TipData)
```

Operatorul *cast* este *unar*, așadar are un singur operand, adresa obiectului prin intermediul căruia este apelat, și întoarce un rezultat de tipul operatorului. Ca urmare, prin această metodă se pot defini numai conversii dintr-un tip clasă într-un tip de bază sau un alt tip clasă.

De remarcat este faptul că, în cazul conversiei dintr-un tip clasă într-un alt tip clasă, funcția operator trebuie să aibă acces la datele membre ale clasei de la care se face conversia, deci trebuie declarată prietenă a clasei respective.

Exemplul de mai jos implementează un operator care convertește un obiect de tip *Point* într-un întreg:

```
class Point
{
    int x, y;
```

```

public:
    // transforma un Point intr-un int
operator int ()
{
    return x * y;
}
};

main()
{
    Point p(10, 72);
    printf("Arie: %d\n", (int)p);
}

```

4.6.2. Conversii de tip folosind constructori

Aceasta metodă constă în definirea unui constructor ce primește ca parametru tipul de la care se face conversia. Constructorul întoarce întotdeauna ca rezultat un obiect de tipul clasei de care aparține. Ca urmare, folosind aceasta metodă se pot realiza numai conversii dintr-un tip de bază sau un tip clasă într-un tip clasă.

Trebuie menționat faptul că, în cazul conversiei dintr-un tip clasă într-un alt tip clasă, constructorul trebuie să aibă acces la datele membre ale clasei de la care se face conversia, deci trebuie declarată prietenă a clasei respective.

```

class CuloareRGB
{
    int rosu, verde, albastru;
public:
    CuloareRGB(int c_rosu, int c_verde, int c_albastru)
    {
        rosu = c_rosu;
        verde = c_verde;
        albastru = c_albastru;
    }
    CuloareRGB()
    {
        rosu = verde = albastru = 0;
    }
};

class Culoare
{
    int culoare;
public:
    // transforma un tip CuloareRGB in Culoare
    Culoare(CuloareRGB rgb)
    {
        return rgb.rosu * 256 + rgb.verde * 16 + rgb.albastru;
    }
};

```

4.7. Constructorul de copiere

O situație care poate apărea deseori este inițializarea unui obiect cu datele membre ale unui obiect de același tip. În paragraful 4.4.1 am descris suprăîncărcarea operatorului de atribuire. Există totuși situații în care acest operator nu poate fi utilizat, cum ar fi la transferul unui obiect ca parametru sau la crearea unei instanțe temporare a unei clase, atunci când copierea membru cu membru nu este adecvată (de exemplu, datorită existenței unor membri de tip pointer). Pentru a rezolva aceste situații, C++ a introdus un constructor special, numit *constructorul de copiere*.

Sintaxa este:

```
NumeClasa::NumeClasa (NumeClasa &NumeObiectSursa)
```

În continuare vom completa clasa Point cu un constructor de copiere:

```

Point::Point(Point &p)
{
    x = p.x;
    y = p.y;
}

```

În cazul în care clasa nu dispune de constructor de copiere, compilatorul generează automat un constructor de copiere care realizează copierea membru cu membru.

4.8. Clase abstracțe

În C++ există posibilitatea de a defini clase generale, care sunt destinate exclusiv creării de noi clase prin derivare, ele neputând fi instantiate și utilizate ca atare (pot fi declarați doar pointeri către astfel de clase, care vor fi inițializați însă cu adresele unor obiecte ale claselor derivate). Clasele de acest gen se numesc *clase abstracțe*. Ele se constituie ca bază în cadrul elaborării de ierarhii de clase, putând fi folosite, spre exemplu, pentru a impune anumite restricții în realizarea claselor derivate.

În vederea construirii unor astfel de clase, s-a introdus conceptul de *funcție virtuală pură*. O astfel de funcție este declarată în cadrul clasei, dar nu este definită. O clasă care conține o funcție virtuală pură este considerată abstractă. Sintaxa definirii acestor funcții este:

```
virtual TipData NumeFunctieMembra() = 0
```

Se impune aici observația că funcțiile virtuale pure trebuie redefinite în clasele derivate, altfel și acestea vor fi considerate abstracte.

```
// clasa abstracta
class XEventListener
{
public:
    // functie virtuala pura
    bool ProcessEvent(XEvent *__Event) = 0;
};

class XObjectEventListener : public XEventListener
{
public:
    virtual bool ProcessEvent(XEvent *__Event);
    virtual bool ProcessMouseEvent(XMouseEvent *__Event);
    virtual bool ProcessKeyboardEvent(XKeyboardEvent *__Event);
    virtual bool ProcessMessage(XMessage *__Event);
    virtual bool ProcessCommand(XMessage *__Event);
    virtual bool ProcessSignal(XEvent *__Event);
    virtual bool ProcessBroadcast(XEvent *__Event);
};
```

4.9. Membri statici ai unei clase

În mod normal, datele membre ale unei clase sunt alocate în cadrul fiecărui obiect. În C++, se pot defini date membre cu o comportare specială, numite *date statice*. Acestea sunt alocate o singură dată, existând sub forma unei singure copii, comună tuturor obiectelor de tipul clasă respectiv, iar operațiunile de creare, inițializare și acces la aceste date sunt independente de obiectele clasei. Sintaxa este:

```
static DeclarareMembru
```

Funcțiile membre statice efectuează de asemenea operații care nu sunt asociate obiectelor individuale, ci întregii clase (de aceea, nu pot accesa decât datele membre statice ale clasei, nu și pe cele obișnuite). Funcțiile exterioare clasei pot accesa membrii statici ai acesteia astfel:

```
NumeClasa::NumeMembru
Obiect::NumeMembru
```

Funcțiile membre statice nu primesc ca parametru implicit adresa unui obiect, aşadar în cadrul lor cuvântul-cheie `this` nu poate fi utilizat. De asemenea, membrii normali ai clasei nu pot fi referiți în exterior decât specificând numele unui obiect, nu doar prin intermediul numelui clasei.

```
class ClasaDeTest
{
    // o variabilă membră statică
    static int rezultat;
};
```

Capitolul 5

FLUXURI

*Ferîji-vă de bug-urile din programul următor;
eu doar am demonstrat că este corect, nu l-am încercat.*
(Donald Knuth)

5.1. Introducere

Fluxurile (*streams*) au în principal rolul de a abstractiza operațiile de intrare/ieșire. Ele oferă metode de scriere și citire a datelor independente de dispozitivul I/O și chiar independente de platformă. Fluxurile încapsulează (ascund) problemele specifice dispozitivului cu care se lucrează, sub biblioteca standard `iostream`.

Alt avantaj al folosirii fluxurilor este datorat implementării bibliotecii `iostream`, care utilizează un sistem de buffer-e. Se știe că în general operațiile de intrare/ieșire care utilizează dispozitivele periferice sunt relativ mari consumatoare de timp, astfel încât aplicațiile sunt uneori nevoite să aștepte terminarea acestor operațiuni. Informațiile trimise către un flux nu sunt remise imediat dispozitivului în cauză, ci sunt transferate într-o zonă de memorie tampon, din care sunt descărcate către dispozitiv abia în momentul umplerii acestei zone de memorie.

În C++, fluxurile au fost implementate utilizând clase, după cum urmează:

- clasa `streambuf` gestionează buffer-ele;
- clasa `ios` este clasa de bază pentru clasele de fluxuri de intrare și de ieșire. Clasa `ios` are ca variabilă membru un obiect de tip `streambuf`;
- clasele `istream` și `ostream` sunt derivate din `ios`;
- clasa `iostream` este derivată din `istream` și `ostream` și oferă metode pentru lucrul cu terminalul;
- clasa `fstream` oferă metode pentru operații cu fișiere.

5.2. Obiecte standard

Când un program C++ care include `iostream.h` este lansat în execuție, sunt create și inițializate automat patru obiecte:

- `cin` gestionează intrarea de la intrarea standard (tastatura);
- `cout` gestionează ieșirea către ieșirea standard (ecranul);
- `cerr` gestionează ieșirea către dispozitivul standard de eroare (ecranul), neutrizând buffer-e;
- `clog` gestionează ieșirea către dispozitivul standard de eroare (ecranul), utilizând buffer-e.

5.3. Redirectări

Dispozitivele standard de intrare, ieșire și eroare pot fi *redirectate* către alte dispozitive. Erorile sunt de obicei redirectate către fișiere, iar intrarea și ieșirea pot fi *conduse* (engl. *piped*) către fișiere utilizând comenzi ale sistemului de operare (utilizarea ieșirii unui program ca intrare pentru altul).

Sintaxa pentru operații de ieșire, `cout`:

```
cout << InformatieDeTrimisLaIesire;
```

respectiv pentru intrare, `cin`:

```
cin >> NumeVariabila;
```

De fapt, `cin` și `cout` sunt niște obiecte definite global, care au supraîncărcat operatorul `>>` respectiv `<<` de mai multe ori, pentru fiecare tip de parametru în parte (`int`, `char *` etc.):

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

De exemplu:

```
#include <iostream.h>
void main()
{
```

```

int NumarIntreg;

cout << "NumarIntreg = ";
cin >> NumarIntreg;
cout << "\nCe ai introdus = " << NumarIntreg << endl;
}

```

Acest scurt program citește de la intrarea standard o valoare întreagă, pe care o trimite apoi către ieșirea standard. Se observă posibilitatea de a utiliza simbolurile „\n”, „\t”, §.a.m.d. (ca la printf, scanf etc.). Utilizarea simbolului endl va forța golirea zonei tampon, adică trimiterea datelor imediat către ieșire.

Atât operatorul >>, cât și << returnează o referință către un obiect al clasei istream. Deoarece cin, respectiv cout sunt și ele obiecte istream, valoarea returnată de o operație de citire/scriere din/în stream poate fi utilizată ca intrare/ieșire pentru următoarea operație de același fel.

5.4. cin

Funcția cin.get()

Funcția membră get() poate fi utilizată pentru a obține un singur caracter din intrare, apelând-o fără nici un parametru, caz în care returnează valoarea utilizată, sau cu un parametru de tip pointer la tipul char.

- **get() fără parametri**

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

```

#include <iostream.h>

void main()
{
    char c;

    while((c = cin.get()) != EOF)
    {
        cout << "c = " << c << endl;
    }
}

```



Citirea de șiruri de caractere utilizând get()

Operatorul >> nu poate fi utilizat pentru a citi corect șiruri de caractere de la intrare deoarece spațiile sunt interpretate ca separator între diverse valori de intrare. În astfel de cazuri trebuie folosită funcția get(). Sintaxa de utilizare a funcției get, în acest caz, este următoarea:

```

cin.get(char *PointerLaSirulDeCaractere, int LungimeMaxima, char Sfarsit);

```

Primul parametru este un pointer la zona de memorie în care va fi depus șirul de caractere. Al doilea parametru reprezintă numărul maxim de caractere ce poate fi citit plus unu. Cel de-al treilea parametru este caracterul de încheiere a citirii, care este optional (implicit considerat „\n”). În cazul în care caracterul de încheiere este întâlnit înainte de a fi citit numărul maxim de caractere, acest caracter nu va fi extras din flux. Există o funcție similară funcției get(), cu aceeași sintaxă, numită getline(). Funcționarea sa este identică cu get(), exceptând faptul că acel ultim caracter menționat mai sus este și el extras din flux.

Funcția cin.ignore()

Această funcție se utilizează pentru a trece peste un număr de caractere până la întâlnirea unui anumit caracter. Sintaxa sa este:

```

cin.ignore(int NumarMaximDeCaractere, char Sfarsit);

```

Primul parametru reprezintă numărul maxim de caractere ce vor fi ignorate, iar al doilea parametru, caracterul care trebuie găsit.

Funcția cin.peek()

Această funcție returnează următorul caracter din flux, fără însă a-l extrage.

Funcția cin.putback()

Această funcție inserează în flux un caracter.

5.5. cout

5.5.1. Funcții membre ale cout

Funcția cout.flush()

Funcția cout.flush() determină trimiterea către ieșire a tuturor informațiilor aflate în zona de memorie tampon. Această funcție poate fi apelată și în forma cout << flush.

Funcția cout.put()

Funcția cout.put() scrie un caracter către ieșire. Sintaxa sa este următoarea:

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

Deoarece această funcție returnează o referință de tip ostream, pot fi utilizate apeluri succesive ale acesteia, ca în exemplul de mai jos:

```
#include <iostream.h>
void main()
{
    cout.put('H').put('i').put('!').put('\n');
}
```

Funcția cout.write()

Această funcție are același rol ca și operatorul <<, cu excepția faptului că se poate specifica numărul maxim de caractere ce se doresc scrise. Sintaxa funcției cout.write() este:

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

5.5.2. Formatarea ieșirii

Funcția cout.width()

Această funcție permite modificarea dimensiunii valorii trimise spre ieșire, care este considerată implicit exact mărimea câmpului în cauză. Ea modifică dimensiunea numai pentru următoarea operație de ieșire. Sintaxa este:

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

Funcția cout.fill()

Funcția cout.fill() permite modificarea caracterului utilizat pentru umplerea eventualului spațiu liber creat prin specificarea unei dimensiuni mai mari decât cea necesară ieșirii, în funcția cout.width(). Sintaxa acesteia este:

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

5.5.3. Opțiuni de formatare a ieșirii

Pentru formatarea ieșirii sunt definite două funcții membre ale cout, și anume:

Funcția cout.setf()

Această funcție activează o opțiune de formatare a ieșirii, primită ca parametru:

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

unde Optiune poate fi:

- showpos determină adăugarea semnului plus (+) în fața valorilor numerice pozitive;
- left, right, internal schimbă alinierarea ieșirii;
- dec, oct, hex schimbă baza de numerație pentru valori numerice;
- showbase determină adăugarea identificatorului bazei de numerație în fața valorilor numerice.

Funcția cout.setw()

Această funcție modifică dimensiunea ieșirii, fiind similară funcției cout.width(). Sintaxa sa este:

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

În continuare vom exemplifica utilizarea funcțiilor pentru formatarea ieșirii:

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
    int number = 783;
```

```

cout << "number = " << number;
cout.setf(ios::showbase);
cout << "number in hexa = " << hex << number;
cout.setf(ios::left);
cout << "number in octal, aligned to the left = " << oct <<
    number;
}

```

5.6. Operații de intrare/ieșire cu fișiere

Operațiile asupra fișierelor se efectuează prin intermediul clasei `ifstream` pentru citire, respectiv `ofstream` pentru scriere. Pentru a le utiliza, aplicațiile trebuie să includă `fstream.h`. Clasele `ofstream` și `ifstream` sunt derivate din clasa `iostream`, ca urmare toți operatorii și toate funcțiile descrise mai sus sunt moștenite și de această clasă.

Sintaxele pentru constructorii acestor două clase sunt:

```

ofstream Variabila(char *NumeFisier, ios::Mod);
ifstream Variabila(char *NumeFisier);

```

Acești constructori au rolul de a deschide fișierul specificat ca parametru. Cel de-al doilea parametru al constructorului `ofstream` este optional și specifică modul de deschidere a fișierului:

- `ios::append` – adaugă la sfârșitul fișierului;
- `ios::atend` – poziționează pointer-ul la sfârșitul fișierului, însă informațiile pot fi scrise oriunde în cadrul fișierului;
- `ios::truncate` – este modul de deschidere implicit: vechiul conținut al fișierului este pierdut;
- `ios::nocreate` – dacă fișierul nu există, atunci operația eșuează;
- `ios::noreplace` – dacă fișierul există deja, atunci operația eșuează.

Pot fi utilizate prescurtările `app` pentru `append`, `ate` pentru `atend` și `trunc` pentru `truncate`.

Pentru a închide aceste fișiere trebuie apelată funcția membră `close()`.

Rezultatul operațiilor de intrare/ieșire poate fi testat prin intermediul a patru funcții membre:

- `eof()` verifică dacă s-a ajuns la sfârșitul fișierului;
- `bad()` verifică dacă s-a executat o operație invalidă;
- `fail()` verifică dacă ultima operație a eșuat;
- `good()` verifică dacă toate cele trei rezultate precedente sunt false.

Capitolul 6

TRATAREA EXCEPTIILOR

*Isaac Newton a recunoscut în secret cătorva prietenii: el înțelegea comportamentul gravitației, dar nu și modul de funcționare.
(Lily Tomlin, *The Search for Signs of Intelligent Life in the Universe*)*

Este un fenomen „natural” ca în programe să se strecoare erori, de diverse naturi. Activitatea de programare implică și acțiuni mai puțin plăcute, adică testarea, depanarea și corectarea erorilor. Costurile de îndepărțare a erorilor cresc de obicei direct proporțional cu întârzierea momentului din cadrul procesului de dezvoltare când sunt descoperite.

Trebuie însă înțeleasă diferența dintre erori (bug-uri) și exceptii. Exceptiile sunt situațiile neașteptate apărute în cadrul sistemului care rulează un program. Programele trebuie să fie pregătite pentru a trata aceste situații excepționale.

În C++ s-a realizat un mecanism facil de tratare a exceptiilor. Astfel, o excepție este un obiect a cărui adresă este trimisă dinspre zonă de cod unde a apărut problema către o zonă de cod care trebuie să o rezolve.

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

1. Se identifică acele zone din program în care se efectuează o operație despre care se cunoaște că ar putea genera o excepție, și se marchează în cadrul unui bloc de tip `try`. În cadrul acestui bloc, se testează condiția de apariție a excepției și, în caz pozitiv, se semnalizează apariția excepției prin intermediul cuvântului-cheie `throw`.
2. Se realizează blocuri de tip `catch` pentru a capta exceptiile atunci când acestea sunt întâlnite.
3. Blocurile `catch` urmează un bloc `try`, și în cadrul lor sunt tratate exceptiile.

Sintaxa pentru `try`:

```

try
{
    // cod
    throw TipExcepție;
}

```

Sintaxa pentru `throw`:

```
throw TipExceptie;
```

Sintaxa pentru `catch`:

```
catch(TipExceptie)
{
    // cod tratare exceptie
}
```

Dacă `TipExceptie` este „...”, este captată orice excepție apărută.

După un bloc `try`, pot urma unul sau mai multe blocuri `catch`. Dacă excepția corespunde cu una dintre declarațiile de tratare a excepțiilor, aceasta este apelată. Dacă nu este definită nici o rutină de tratare a excepției, este apelată rutina predefinită, care încheie execuția programului în curs. După ce rutina este executată, programul continuă cu instrucțiunea imediat următoare blocului `try`.

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

```
class TipExceptie {};
```

În continuare prezentăm un exemplu de program care utilizează tratarea excepțiilor.

```
#include <iostream.h>

#define MAXX 80
#define MAXY 25

class Point
{
public:

    class xZero {};
    class xOutOfScreenBounds {};

    Point(unsigned __x, unsigned __y)
    {
        x = __x;
        y = __y;
    }

    unsigned GetX()
    {
        return x;
    }

    unsigned GetY()
    {
        return y;
    }

    void SetX(unsigned __x)
```

```
{
    if(__x > 0)
        if(__x <= MAXX)
            x = __x;
        else
            throw xOutOfScreenBounds();
    else
        throw xZero();
}

void SetY(unsigned __y)
{
    if(__y > 0)
        if(__y <= MAXY)
            y = __y;
        else
            throw xOutOfScreenBounds();
    else
        throw xZero();
}

protected:
    int x, y;
};

main()
{
    Point p(1, 1);

    try
    {
        p.SetX(5);
        cout << "p.x successfully set to " << p.GetX() << ".\n";
        p.SetX(100);
    }

    catch(Point::xZero)
    {
        cout << "Valoare zero!\n";
    }

    catch(Point::xOutOfScreenBounds)
    {
        cout << "Out of screen bounds!\n";
    }

    catch(...)
    {
        cout << Exceptie necunoscuta!\n";
    }
}
```

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

Capitolul 7

TEMPLATE-URI

Aceia dintre noi care suntem arhitecti avem poate această dorință aproape de centrul vieții noastre: că într-o zi, undeva, cuiva, vom construi o clădire care este extraordinară, frumoasă, un loc în care oamenii pot merge și visa timp de secole.

(Christopher Alexander)

Template-ul implementează acesta-zisul concept de *tip parametrizat* (engl. *parametrized type*). Un template reprezintă o familie de tipuri sau funcții, cu alte cuvinte, un şablon sau model. Acest concept a fost introdus în primul rând pentru a crește gradul de reutilizabilitate a codului. De exemplu, pentru a implementa o listă de numere întregi este necesară în mod normal realizarea unei clase speciale (să spunem *ListOfIntegers*), iar pentru o listă de siruri, altă clasă (să spunem *ListOfStrings*). Conceptul de *template* permite realizarea unei clase generale (să spunem *List*), care să accepte orice tip de element, inclusiv tipuri necunoscute la momentul implementării acesteia. Tipul template-ului este stabilit în momentul instanțierii sale. Template-urile sunt foarte utile pentru realizarea de biblioteci care trebuie să ofere metode generice de prelucrare a datelor.

Sintaxa generală de declarare a unui template este următoarea:

```
template < ListaDeParametri > Declaratie
```

unde *Declaratie* reprezintă declararea sau definirea unei clase sau funcții, definirea unui membru static al unei clase template, definirea unei clase sau funcții membre a unei clase template, sau definirea unui membru template al unei clase.

Clasele parametrizate (sau clasele template) se declară astfel:

```
template <class NumeParametru>
class NumeClasa
{
    // ...
    // definirea clasei
}
```

Particularizarea (= stabilirea tipului) clasei template se face prin intermediul unei construcții de genul:

```
NumeClasa <NumeParametru>
```

unde *NumeParametru* reprezintă tipul obiectului.

Funcțiile template se declară astfel:

```
template <class NumeParametru>
// ...
// declaratia functiei
```

Să considerăm în continuare ca exemplu implementarea unei stive generice folosind template-uri.

```
#include <iostream.h>

template <class T> class StackItem
{
public:
    StackItem *Next;
    T *Data;
    StackItem(T __Data, StackItem <T> *__Next)
    {
        Data = new T(__Data);
        Next = __Next;
    }
};

template <class T> class Stack
{
public:
    T pop()
    {
        T result = *(Data->Data);
        StackItem <T> *temp = Data;
        Data = Data->Next;
        delete temp;
        return result;
    }

    T top()
    {
        return *(Data->Data);
    }

    void push(T __Data)
    {
        Data = new StackItem <T>(__Data, Data);
    }
};
```

```

int isEmpty()
{
    return Data == 0;
}

Stack()
{
    Data = 0;
}

private:
    StackItem <T> *Data;
};

main()
{
    Stack <int> anIntegerStack;
    anIntegerStack.push(5);
    anIntegerStack.push(7);
    if(anIntegerStack.isEmpty())
        cout << "Stiva goala" << endl;
    else
        cout << anIntegerStack.pop() << endl;
    cout << anIntegerStack.top() << endl;
}

```

În exemplul următor a fost implementată o listă generică (List). Ca elemente ale listei s-au folosit obiecte de tip Point (clasa definită în capitolul 4). Pentru parcurgerea ușoară a listei, a fost implementată o clasă de tip *iterator*, care poate fi considerată ca fiind un „cursor” care străbate lista. Funcția List.begin() returnează un iterator poziționat pe primul element al listei, List.end() pe ultimul element al listei. Saltul la următorul element al listei se face cu ajutorul operatorului ++ din clasa Iterator.

```

#include <iostream.h>

class Point
{
    friend ostream& operator << (ostream& output, Point p);

protected:
    unsigned x, y;

public:
    Point()
    {
        x = 0;
        y = 0;
    }

    Point(unsigned X, unsigned Y)

```

```

    {
        x = X;
        y = Y;
    }

    ~Point() {}

    unsigned GetX()
    {
        return x;
    }

    unsigned GetY()
    {
        return y;
    }

    void SetX(unsigned X)
    {
        x = X;
    }

    void SetY(unsigned Y)
    {
        y = Y;
    }
};

ostream& operator << (ostream& output, Point p)
{
    output << "(" << p.x << ", " << p.y << ")";
    return output;
}

template <class T> class Item {
public:
    Item *Next;
    T *Data;
    Item(T __Data, Item <T> *__Next)
    {
        Data = new T(__Data);
        Next = __Next;
    }
};

template <class T> class List {
public:
    T pop_front()
    {
        T result = *(Data->Data);
        Item <T> *temp = Data;
        Data = Data->Next;
        delete temp;
        return result;
    }
};

```

```

T front()
{
    return *(Data->Data);
}

void push_front(T __Data)
{
    Data = new Item <T>(__Data, Data);
}

int empty()
{
    return Data == 0;
}

List()
{
    Data = 0;
}

class Iterator
{
    friend class List <T>;
protected:
    Item <T> *Current;

    Iterator(Item <T> *x)
    {
        Current = x;
    }

public:
    Iterator() {}

    int operator == (Iterator& x)
    {
        return Current == x.Current;
    }

    int operator != (Iterator& x)
    {
        return Current != x.Current;
    }

    T operator *()
    {
        return *(Current->Data);
    }

    Iterator& operator ++(int)
    {
        Current = Current->Next;
        return *this;
    }
};

```

```

Iterator begin()
{
    return Iterator(Data);
}

Iterator end()
{
    Item <T> *temp;
    for(temp = Data; temp; temp = temp->Next);
    return Iterator(temp);
}

private:
    Item <T> *Data;
};

main()
{
    List <Point> anPointList;
    List <Point>::Iterator index, end;

    anPointList.push_front(Point(1, 1));
    anPointList.push_front(Point(3, 14));
    index = anPointList.begin();
    end = anPointList.end();
    if(anPointList.empty())
        cout << "Lista vida" << endl;
    else
        for(; index != end; index++)
            cout << *index << " ";
    cout << endl;
}

```

Clasele template pot avea trei tipuri de prieteni (*friends*):

- o clasă sau o funcție care nu este de tip template;
- o clasă sau o funcție template;
- o clasă sau o funcție template având tipul specificat.

Dacă este necesară particularizarea unei funcții template sau a unei funcții membre a unei clase template pentru un anumit tip, funcția respectivă poate fi supraîncărcată pentru tipul dorit.

Trebuie remarcat de asemenea că în cazul în care o clasă template conține membri statici, fiecare instanță a template-ului în cauză va conține propriile date statice.

În cele ce urmează vom prezenta o clasă utilizată pentru conectarea la un server SQL, în cazul nostru PostgreSQL, precum și un scurt program care o utilizează. Se remarcă comentariile în stil DOC++ în fișierul-antet *toolbox.h*, care vor ajuta la generarea automată a documentației aferente.

toolbox.h:

```

#ifndef _TOOLBOX_H
#define _TOOLBOX_H

#include <libpq-fe.h>
#include <libpq-int.h>

/** Returneaza timpul exprimat in secunde, pornind de la reprezentarea
    tip AAAA-LZ-ZZ
*/
unsigned long get_time(const char *__s);

/// Returneaza timpul exprimat in secunde
unsigned long get_time(unsigned long __year,
    unsigned long __month,
    unsigned long __day);

/// Extrage anul
int get_year(unsigned long __time);

/// Extrage luna
int get_month(unsigned long __time);

/// Extrage ziua
int get_day(unsigned long __time);

/// Extrage ora
int get_hour(unsigned long __time);

/// Extrage minutul
int get_minute(unsigned long __time);

/// Extrage secunda
int get_second(unsigned long __time);

/// Returneaza o valoare de tip caracter
char get_char(const char *__s);

/// Returneaza o valoare de tip bool
bool get_bool(const char *__s);

/// Returneaza o valoare de tip long
long get_long(const char *__s);

/// Returneaza o valoare de tip double
double get_double(const char *__s);

/// Returneaza o valoare de tip sir de caractere
char *get_string(char *__s);

/// clasa care descrie o conexiune la o baza de date

```

```

class SqlDatabase {
public:
    /// face conectarea la baza de date specificata
    SqlDatabase(const char *__name);

    /// inchide conexiunea la baza de date
    ~SqlDatabase();

    /// trimite serverului o interogare SQL
    ExecStatusType Exec(const char *__query);

    /** trimite serverului o interogare SQL si returneaza 1
        daca aceasta a fost executata cu succes
    */
    int ExecCommandOk(const char *__query);

    /** trimite serverului o interogare SQL care extrage date
        dintr-o baza de date si returneaza 1 daca aceasta
        a fost executata cu succes
    */
    int ExecTuplesOk(const char *__query);

    /** Returneaza mesajul de eroare generat de ultima interogare
    */
    const char *ErrorMessage();

    /** Returneaza numarul de tuple ("inregistrari") extrase la
        ultima interogare
    */
    int Tuples();

    /** Returneaza numarul de campuri extrase la ultima interogare
    */
    int Fields();

    /// Returneaza numele unui camp
    const char *FieldName(int __field_num);

    /// Returneaza identificatorul unui camp
    int FieldNum(const char *__field_name);

    /// Returneaza tipul unui camp
    Oid FieldType(int __field_num);

    /// Returneaza tipul unui camp
    Oid FieldType(const char *__field_name);

    /// Returneaza lungimea unui camp
    int FieldSize(int __field_num);

    /// Returneaza lungimea unui camp
    int FieldSize(const char *__field_name);

    /** Returneaza valoarea unui camp dintr-o inregistrare specificata
    */

```

```

const char *GetValue(int __tup_num, int __field_num);
/** Returneaza valoarea unui camp dintr-o inregistrare specificata */
const char *GetValue(int __tup_num,
    const char *__field_name);

/** Returneaza lungimea unui camp dintr-o inregistrare specificata */
int GetLength(int __tup_num, int __field_num);

/** Returneaza lungimea unui camp dintr-o inregistrare specificata */
int GetLength(int __tup_num, const char *__field_name);

/// Elibereaza memoria alocata pentru inregistrari
void Clear();

/// Verifica daca o conexiune a esuat
int ConnectionBad();

/// Returneaza starea ultimei interogari
unsigned long OidStatus();

/// Returneaza starea conexiunii
ConnStatusType Status();
protected:
    PGconn *pgConn;
    PGresult *pgResult;
    ExecStatusType pgStatus;
};

void log(char *__fmt, ...);

extern char *username;

#endif

```

toolbox.cc:

```

#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <syslog.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#include "toolbox.h"

char *username;

```

```

int get_year(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_year + 1900;

    return result;
}

int get_month(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_mon + 1;

    return result;
}

int get_day(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_mday;

    return result;
}

int get_hour(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_hour;

    return result;
}

int get_minute(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_min;

    return result;
}

int get_second(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_sec;

    return result;
}

char get_char(const char *__s)
{
    return __s[0];
}

bool get_bool(const char *__s)

```

```

{
    return (_s[0] == 't' || _s[0] == 'T' || _s[0] == 'Y' ||
           _s[0] == 'y');
}

long get_long(const char *__s)
{
    return strtol(__s, 0, 10);
}

double get_double(const char *__s)
{
    return strtod(__s, 0);
}

char *get_string(char *__s)
{
    unsigned long i;

    if(__s == 0)
        return 0;
    i = strlen(__s) - 1;
    while(__s[i] == ' ')
        __s[i--] = '\0';

    return __s;
}

unsigned long get_time(const char *__s)
{
    int day, month, year;
    struct tm t;

    if(sscanf(__s, "%4d-%2d-%2d", &year, &month, &day) == 3)
    {
        t.tm_year = year - 1900;
        t.tm_mon = month - 1;
        t.tm_mday = day;
        t.tm_hour = 0;
        t.tm_min = 0;
        t.tm_sec = 0;
        t.tm_isdst = -1;
        return mktime(&t);
    }

    return 0;
}

unsigned long get_time(unsigned long __year, unsigned long __month,
                      unsigned long __day)
{
    struct tm t;
    t.tm_year = __year - 1900;
}

```

```

t.tm_mon = __month - 1;
t.tm_mday = __day;
t.tm_hour = 0;
t.tm_min = 0;
t.tm_sec = 0;
t.tm_isdst = -1;

return mktime(&t);
}

SqlDatabase::SqlDatabase(const char *__name)
{
    pgConn = PQsetdb(0, 0, 0, 0, __name);
}

SqlDatabase::~SqlDatabase()
{
    PQfinish(pgConn);
}

ExecStatusType SqlDatabase::Exec(const char *__query)
{
    extern int PenguinsDone();

    if(PQstatus(pgConn) == CONNECTION_BAD)
        return (pgStatus = PGRES_FATAL_ERROR);
    pgStatus = PQresultStatus((pgResult = PQexec(pgConn, __query)));
    if(pgStatus == PGRES_BAD_RESPONSE ||
       pgStatus == PGRES_FATAL_ERROR ||
       pgStatus == PGRES_NONFATAL_ERROR)
    {
        // incercam sa restabilim legatura cu serverul PostgreSQL
        PQreset(pgConn);
        if(PQstatus(pgConn) == CONNECTION_BAD)
        {
            log("Eroare fatala: am pierdut legatura cu serverul de
                baze de date");
            exit(0);
        }
        pgStatus = PQresultStatus((pgResult = PQexec(pgConn,
            __query)));
    }
    return pgStatus;
}

int SqlDatabase::ExecCommandOk(const char *__query)
{
    return (Exec(__query) == PGRES_COMMAND_OK);
}

int SqlDatabase::ExecTuplesOk(const char *__query)
{
    return (Exec(__query) == PGRES_TUPLES_OK);
}

```

```

const char *SqlDatabase::ErrorMessage()
{
    return PQerrorMessage(pgConn);
}

void SqlDatabase::Clear()
{
    PQclear(pgResult);
}

int SqlDatabase::Fields()
{
    return PQnfields(pgResult);
}

int SqlDatabase::Tuples()
{
    return PQntuples(pgResult);
}

const char *SqlDatabase::FieldName(int __field_num)
{
    return PQfname(pgResult, __field_num);
}

int SqlDatabase::FieldNum(const char *__field_name)
{
    return PQfnumber(pgResult, __field_name);
}

Oid SqlDatabase::FieldType(int __field_num)
{
    return PQftype(pgResult, __field_num);
}

Oid SqlDatabase::FieldType(const char *__field_name)
{
    return FieldType(FieldNum(__field_name));
}

const char *SqlDatabase::GetValue(int __tup_num, int __field_num)
{
    return PQgetvalue(pgResult, __tup_num, __field_num);
}

const char *SqlDatabase::GetValue(int __tup_num,
    const char *__field_name)
{
    return GetValue(__tup_num, FieldNum(__field_name));
}

int SqlDatabase::GetLength(int __tup_num, int __field_num)
{
}

```

```

    return PQgetlength(pgResult, __tup_num, __field_num);
}

int SqlDatabase::GetLength(int __tup_num, const char *__field_name)
{
    return GetLength(__tup_num, FieldNum(__field_name));
}

int SqlDatabase::FieldSize(int __field_num)
{
    return PQfsize(pgResult, __field_num);
}

int SqlDatabase::FieldSize(const char *__field_name)
{
    return FieldSize(FieldNum(__field_name));
}

int SqlDatabase::ConnectionBad()
{
    return (pgConn->status == CONNECTION_BAD);
}

unsigned long SqlDatabase::OidStatus()
{
    return get_long(PQoidStatus(pgResult));
}

ConnStatusType SqlDatabase::Status()
{
    return pgConn->status;
}

void log(char *__fmt, ...)
{
    va_list argptr;
    char s[1024];

    va_start(argptr, __fmt);
    vsprintf(s, __fmt, argptr);
    va_end(argptr);
    syslog(LOG_NOTICE, "%s: %s", username, s);
}

```

pgtest.cc:

```
#include <pwd.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

#include "toolbox.h"

int main()
{
    SqlDatabase *db;
    passwd *pswd;
    int i;

    // obtin numele utilizatorului curent
    pswd = getpwuid(geteuid());
    username = pswd->pw_name;

    // initiez o conexiune la baza de date 'biosfarm'
    db = new SqlDatabase("biosfarm");

    // trimite o interogare serverului SQL
    // extrage toate inregistrările din tabela 'clienti'
    db->Exec("select * from clienti order by denumire");

    // afiseaza denumirea si numarul de telefon
    printf("Denumire client      Numar telefon\n");
    for(i = 0; i < db->Tuples(); i++)
        printf("%20s %15s\n", db->GetValue(i, "denumire"),
               db->GetValue(i, "telefon"));

    // elibereaza zona de memorie alocata pentru inregistrari
    db->Clear();

    // inchide conexiunea la serverul SQL
    delete db;

    return 0;
}
```

Pentru compilarea acestui program se va utiliza comanda:

```
g++ -o test -I/usr/include/pgsql -I/usr/include/pgsql/internal
pgtest.cc toolbox.cc -lpq
```

Capitolul 8**PROIECTAREA ȘI DEZVOLTAREA DE APLICAȚII ORIENTATE-OBIECT**

De vorbit este ușor. Arată-mi codul!
 (Linus Torvalds, într-un mesaj trimis pe linux-kernel)

Dezvoltarea de sisteme orientate-obiect pare a fi, la prima vedere, mai complicată și de durată mai mare decât dezvoltarea aplicațiilor tradiționale. În realitate, durata și costurile dezvoltării de aplicații orientate-obiect sunt mult mai mici.

Etapa fundamentală în cadrul acestui proces o constituie cea de proiectare a sistemului, chiar dacă este evident că structura internă a acestuia este irelevantă pentru utilizatori. S-a constatat de asemenea că succesul aplicațiilor orientate-obiect depinde în principal de doi factori:

1. Existența unei viziuni arhitecturale coerente și bine definite

Arhitectura unui sistem orientat pe obiecte cuprinde atât structura claselor și interacțiunea dintre obiecte, cât și împărțirea aplicației în module și niveluri de abstractizare. Iată câteva condiții în vederea realizării unei arhitecturi corecte:

- niveluri de abstractizare bine definite;
- clase având interfețe bine definite, a căror modificare provoacă schimbări minime asupra celoralte clase;
- modificarea modului de implementare a unei clase nu are repercusiuni asupra interfeței sau implementării celoralte clase;
- arhitectura sistemului este simplă, realizată prin abstracții și mecanisme obișnuite.

**2. Urmarea unui ciclu de dezvoltare atât iterativ
cât și incremental bine administrat**

Există în principal două tipuri de cicluri de dezvoltare:

- ciclu de dezvoltare nedefinit. În acest caz, este imposibil de știut viteza dezvoltării sistemului, momentul în care va fi finalizat, calitatea sistemului rămânând permanent sub semnul întrebării. Este posibil ca o parte dintre eforturile depuse să fie ineficiente, aşadar costurile dezvoltării să fie foarte mari;

- reguli clare care stabilesc fiecare aspect al ciclului. În acest al doilea caz, este împiedicată creativitatea și experimentul, care ar putea produce o aplicație având calitate sporită. Cerințele utilizatorilor ajung cu dificultate la nivelul programatorilor ce realizează aplicația, îngreunând procesul de dezvoltare și mărindu-i costurile.

În realitate, nu vom întâlni nicăieri vreunul dintre aceste cazuri distinct. În cadrul dezvoltării de sisteme software, aceste două tipuri de cicluri de dezvoltare se întrepătrund, înclinând mai mult sau mai puțin spre una dintre extreame, în funcție de deciziile luate de conducătorii acestor proiecte. S-a tras în multe locuri concluzia că un ciclu de dezvoltare ideal este atât de tip iterativ cât și de tip incremental.

Ce înseamnă că un ciclu este iterativ? Un proces iterativ presupune îmbunătățirea succesivă a arhitecturii obiectuale, utilizând experiența și rezultatele obținute în fiecare etapă sau versiune în etapa următoare de analiză și dezvoltare. Ce reprezintă un ciclu incremental? În cadrul unui proces incremental, fiecare trecere printr-un astfel de ciclu de analiză/dezvoltare conduce la îmbunătățirea deciziilor, rezultând astfel în final o soluție care întrunește adevăratele cerințe ale utilizatorilor, și are o arhitectură clară, este eficientă și ușor de întreținut.

De obicei, sistemele comerciale sunt realizate urmând un ciclu mai clar definit, deoarece sunt realizate în cadrul unor companii cu număr mare de programatori și trebuie executate într-un interval predefinit de timp. Spre deosebire de acestea, sistemele *open source* sunt construite după un ciclu mai vag definit, dar care de cele mai multe ori conduce la sisteme mai extensibile, mai flexibile și de calitate superioară celor comerciale.

Sintetizat, etapele dezvoltării unui sistem orientat pe obiecte sunt:

I. Analiza

- Identificarea obiectelor din cadrul sistemului.
- Identificarea acțiunilor efectuate de fiecare obiect.
- Identificarea interacțiunilor dintre aceste obiecte (mesajele prin care comunică obiectele).

II. Abstractizarea

- Stabilirea claselor ale căror instanțiere sunt aceste obiecte.
- Elaborarea ierarhiei de clase.

III. Implementarea

- Împărțirea pe module (clase) a ierarhiei de clase.
- Elaborarea claselor de bază (fundamentale).
- Elaborarea celorlalte clase din ierarhie (în aceasta etapă se determină și disfuncțiile în proiectare/implementare a claselor de bază și este posibilă revenirea la punctul 2).
- Asamblarea într-un tot unitar a modulelor (claselor).

IV. Testarea (se realizează și pe parcursul etapei III: 2-4)

V. Scrierea de documentație (eventual în timpul etapelor III și, eventual, IV)

Perioada de viață a unui sistem nu se rezumă însă la proiectarea și dezvoltarea sa; ea continuă cu lansarea de noi versiuni, corectarea erorilor ce nu au fost detectate în cadrul etapei de testare, adaptarea sa, adăugarea de noi facilități la cererea utilizatorilor, și acest ciclu nu se încheie niciodată.

În cadrul etapei de proiectare a sistemului orientat pe obiecte, nu trebuie să uităm că un sistem complex bine construit este alcătuit din mai multe componente simple (atomice) care interacționează. Sistemele monolitice au costurile de proiectare, implementare și mai ales de întreținere mult mai mari decât sistemele modulare. Programarea orientată pe obiecte oferă toate avantajele în vederea creării de sisteme modulare.

În ceea ce privește ierarhiile de clase, există două categorii: în prima, toate sau aproape toate clasele sunt derivate dintr-o clasă de bază, rădăcină; într-o două, pot exista mai multe ierarhii de clase distincte. Avantajul de a avea o singură clasă de bază este acela că se poate evita cu ușurință moștenirea multiplă; dezavantajul este că de multe ori în cadrul procesului de implementare a claselor derivate poate apărea necesitatea modificării clasei de bază.

În cadrul etapei de implementare a aplicației, este important să se stabilească o convenție unitară privind stilul de programare utilizat. De cele mai multe ori nu are importanță stilul adoptat, însă un stil unitar poate ușura foarte mult interconectarea modulelor componente și poate micșora costurile de întreținere a sistemului. În continuare vom face câteva sugestii:

- Indentarea**
 - dimensiunea tabularii să fie de 2-4 caractere.
- Acoladele**
 - accoladele corespunzătoare să fie aliniate vertical;
 - pe liniile conținând o accoladă să nu apară și cod.
- Liniile lungi**
 - dimensiunea unei linii de cod să fie mai mică decât lățimea ecranului;
 - dacă o linie este împărțită pe mai multe rânduri, atât cel de-al doilea rând cât și următoarele să fie indentate.
- Codul-sursă**
 - să nu existe spații înainte și după operatorii unari;
 - să existe spațiu înainte și după operatorii binari;
 - să existe un spațiu după virgule și punct și virgulă, dar nu înainte;
 - să nu existe spații înainte și după paranteze;
 - să existe spațiu înainte și după cuvintele-cheie;
 - să fie utilizate linii libere pentru a separa diverse module de cod-sursă și a mări lizibilitatea programului.

- **Comentariile**

- textul unui comentariu să fie separat de „//” cu un spațiu;
- pe cât posibil, să fie utilizate comentariile stil C++, „//”, în loc de cele în stil C, „/* */”;
- să fie la obiect și de nivel cât mai înalt;
- să indice operația realizată de către funcție, efectele secundare, tipul parametrilor, valorile pe care le poate returna.

- **Denumirile identificatorilor**

- denumirile să fie cât mai descriptive posibil;
- să fie evitate abrevierile criptice (de exemplu, este de preferat contor în loc de c);
- să nu fie utilizată notația „maghiară” (care prevede includerea tipului unei variabile în denumirea sa, de exemplu pszSir, adică o variabilă pointer la un sir de caractere terminat cu caracterul nul).

- **Drepturile de acces la membri**

- să se declare mai întâi membrii **public**, apoi cei **protected**, iar apoi cei **private**;
- datele membre să apară după declararea metodelor;
- prima metodă declarată să fie constructorul, apoi destructorul.

- **Definirea claselor**

- ordinea definirii metodelor să fie aceeași cu cea a declarării acestora.

Un ultim sfat: nu este de ajuns să citiți cursuri, cărți sau cod-sursă! Cea mai bună metodă pentru a învăța un limbaj de programare este de a scrie efectiv cod.

Capitolul 9

PROBLEME PROPUSE

1. Să se rescrie programul de concatenare a două fișiere prezentat în partea I, capitolul 9, utilizând fluxuri.
2. Să se definească o clasă de tip număr complex, care să implementeze operatorii +, -, *, <, > și ==.
3. Să se definească o clasă de tip arbore care să implementeze operațiile fundamentale asupra arborilor (creare, parcurgere, căutare, stergere, inserare). Nodurile arborelui vor conține elemente de tip întreg.
4. Utilizând clasa Point, descrisă în capitolele precedente, să se implementeze clasa Rectangle pentru a descrie un dreptunghi, conținând operatorii =, ==, += și -=, precum și funcții pentru intersecție și uniune.
5. Să se definească clasa String, care să implementeze operatorul de copiere, precum și operatorii =, <, > și ==.
6. Să se scrie un program care să poată sorta orice tip de dată (siruri de caractere, întregi etc.), utilizând template-uri.
7. Să se modifice programul de la punctul 3 pentru lucru cu arbori, pentru a putea salva, respectiv încărca informațiile dintr-un fișier. Nodurile arborelui vor conține elemente de tip sir de caractere.
8. Să se definească structurile de stivă și coadă ca și cazurile particulare ale listei liniare. Să se transforme apoi clasele definite în clase parametrizate (template-uri).
9. Să se definească clasa Client, care să conțină membrii și metodele necesare manipulării informațiilor despre un beneficiar al unei societăți comerciale, utilizând clasa String definită la punctul 5. Clasa trebuie să implementeze și operatorii-iteratori ++ și -- pentru parcurgerea listei de clienți.
10. Să se implementeze clasele Factura (care să conțină membrii nr_doc și data) și PlataFactura (care să conțină membrii nr_doc, data și tip), care să definească facturile emise la un beneficiar și respectiv plățile efectuate de către acesta. Să se modifice clasa Client, definită la punctul precedent, astfel încât să conțină lista de facturi emise. La rândul său, fiecare factură trebuie să poată conține o listă de încasări cu care a fost achitată. Programul trebuie să poată citi de la tastatură (utilizând fluxuri) aceste date și să afișeze apoi un centralizator de plăți pentru fiecare beneficiar în parte. Acesta trebuie să includă lista de facturi emise beneficiarului, precum și documentele cu care acestea au fost achitate.

Partea a treia

**INSTRUMENTE DE PROGRAMARE
PENTRU LINUX**

Capitolul 1

EDITOARE

*Sper că vei găsi curajul de a trăi în continuare
în ciuda existenței acestei facilități.*
(Richard Stallman)

1.1. Editorul *Emacs*

GNU Emacs este un editor de texte extensibil. El a fost creat de către Richard Stallman, prima versiune funcțională apărând în anul 1985.

Emacs nu este doar un editor de texte. Din Emacs se pot compila programe, se pot executa și depana programe, se poate citi e-mailul și a.m.d. Extensiile Emacs sunt scrise în limbajul Lisp. Există o versiune de Emacs pentru terminal și una pentru mediul grafic X Window, xemacs.

Pentru folosirea acestui program trebuie instalat pachetul Emacs. Lansarea în execuție a editorului se face prin comanda cu același nume.

Ecranul Emacs este alcătuit în principal din mai multe ferestre de lucru, denumite *buffere*. Este posibil ca unele buffere să nu fie vizibile, unul dintre ele fiind considerat curent, cu care lucrează utilizatorul. Editorul Emacs poate lucra cu mai multe fișiere deschise simultan, fiecare aflându-se într-un buffer diferit. Fiecare buffer este afișat într-o fereastră (*window*). De asemenea, fiecare fereastră conține în partea inferioară o linie de stare (*mode line*), care indică diferite informații utile despre buffer: dacă a fost modificat, linia și coloana curentă etc.

Ecranul Emacs mai conține și un așa-zis *mini-buffer*, practic o zonă în care Emacs raportează rezultatele comenziilor, combinațiile de taste apăsate parțial etc.

În marginea superioară a ecranului se găsește meniul Emacs.

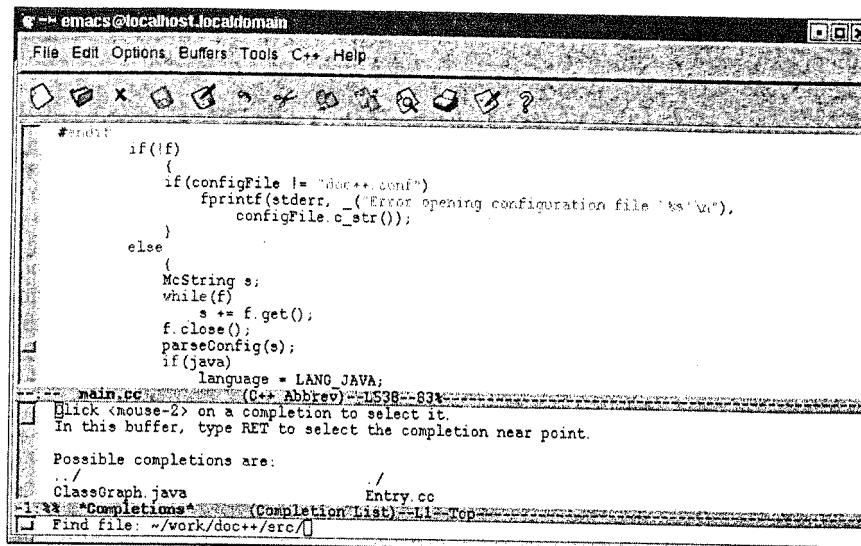


Figura 1.1. Editorul de texte Emacs

Vom prezenta în continuare combinațiile principale de taste folosite în Emacs.

În Emacs (dar nu numai) sunt utilizate două taste speciale, și anume Control și Meta, care se folosesc doar în combinație cu alte taste. Control corespunde tastei Control de pe majoritatea tastaturilor, iar Meta tastei Alt de pe PC-uri. Codul generat de tasta Meta împreună cu altă tastă se poate obține (fiind necesar, de exemplu, în cazul conexiunilor la distanță) prin apăsarea tastei Escape (ESC), apoi a tastei respective. Din acest motiv, tasta Escape se mai numește și *metafy*. Pentru a urma convențiile, vom urma și noi notația echivalentă C = Control, M = Meta.

Orice comandă are un nume simbolic, și poate fi apelată prin acesta. Unele comenzi nu au o combinație de taste atașată. Pentru a apela o asemenea comandă se tastează M-x *nume_comanda*. Poate fi utilizată și tasta Tab pentru a completa aceste nume. Numele complet al comenzi va fi afișat în acest caz în mini-buffer.

O comandă des folosită este C-x, care este o comandă prefix, adică este întotdeauna urmată de altă combinație de taste. Majoritatea comenziilor care încep cu C- se referă la unități de text (litere, rânduri). Toate comenziile care încep cu M- se referă la unități de limbă (cuvinte, fraze).

C-@	Setează un marcat	M-w	Copiază regiunea în bufferul de ștergere
C-a	Salt la începutul liniei	C-x 0	Șterge fereastra curentă
M-a	Salt la începutul frazei	C-x 1	Păstrează numai fereastra curentă
C-b	Salt un caracter înapoi	C-x 2	Aranjează ferestrele vertical
M-b	Salt un cuvânt înapoi	C-x 3	Aranjează ferestrele orizontal
C-d	Șterge un caracter	C-x b	Selectează un buffer
M-d	Șterge un cuvânt	C-x C-b	Listează toate bufferele
DEL	Șterge caracterul precedent	C-x C-c	Iese din Emacs
M-DEL	Șterge cuvântul precedent	C-x C-f	Caută un fișier și îl încarcă
C-e	Salt la sfârșitul liniei	C-x C-w	Salvează fișierul curent
M-e	Salt la sfârșitul frazei	C-x d	Apelaază managerul de fișiere
C-f	Salt la următorul caracter	C-x k	Șterge conținutul bufferului curent
M-f	Salt la următorul cuvânt	C-x o	Mută cursorul la altă fereastră
C-g	Anulează comanda curentă	C-x s	Salvează conținutul tuturor bufferelor
C-h i	Apelaază sistemul de ajutor	C-x u	Undo, adică inversează efectul ultimei comenzi
C-k	Șterge linia curentă	C-y	Restaurează ultimul text șters
M-k	Șterge fraza	M-y	Restaurează ultimul text din bufferul de ștergere
C-l	Reafisează ecranul	C-z	Opriște temporar Emacs
C-n	Salt la următoarea linie	M-/	Completează automat cuvântul
C-p	Salt la linia precedentă	M--	Idem cu C-u i
C-r	Căutare înapoi	M-c	Modifică cuvântul în caractere mari
C-s	Căutare înainte	M-h	Selectează paragraful curent ca regiune
C-SPAC	Setează un marcat la cursor	M-l	Modifică cuvântul în caractere mici
E			
C-v	Salt la pagina următoare	M-x	Apelaază o comandă după numele acesteia
M-v	Salt la pagina precedentă	M-<	Salt la începutul bufferului
C-w	Șterge conținutul regiunii	M->	Salt la sfârșitul bufferului

1.2. Alte editoare: *mcedit, joe* și *vi*

1.2.1. Programul *mcedit*

Programul *mcedit* este editorul din cadrul managerului de fișiere *mc*. Este un editor simplu, oferind o serie de comenzi de bază, util fiind mecanismul de *syntax highlighting*, adică diferite scheme de colorare a instrucțiunilor, parantezelor etc. Există și posibilitatea de a defini macrouri și de a emula comenziile Emacs.

Acest editor face parte din pachetul *mc*. Apelarea lui se poate face fie prin lansarea comenzi *mcedit*, fie din cadrul *MC* cu tasta F4 sau Shift-F4.

Comenziile principale sunt:

- F1 apeleză pagina de ajutor
- F2 salvează conținutul fișierului
- F3 marchează începutul, respectiv sfârșitul unui bloc
- F4 caută și înlocuiește un sir de caractere
- F5 copie conținutul blocului
- F6 mută conținutul blocului
- F7 caută un sir de caractere
- F8 șterge conținutul blocului
- F9 apeleză meniul *mcedit*
- F10iese din editor
- Ctrl-sageti stanga/dreapta salt la cuvântul precedent/următor
- Shift-sageti selecteză un bloc
- Ctrl-Insert copie blocul selectat în buffer (*clipboard*)
- Shift-Insert insereză conținutul din buffer
- Meta-L salt la o anumită linie

Pentru semnificația tastei Meta, a se consulta paragraful despre Emacs.

1.2.2. Programul *joe*

Programul *joe* este un editor minimal care permite următoarele comenzi:

- Ctrl-Z salt la cuvântul următor
- Ctrl-X salt la cuvântul precedent
- Ctrl-K F căutare în text
- Ctrl-L continuarea căutării

- Ctrl-K L salt la o anumită linie
- Ctrl-K B marchează începutul blocului
- Ctrl-K K marchează sfârșitul blocului
- Ctrl-K M mută conținutul blocului
- Ctrl-K C copie conținutul blocului
- Ctrl-K Y șterge conținutul blocului
- Ctrl-Y șterge linia curentă
- Ctrl-K X salvează fișierul și părăsește editorul

1.2.3. Programul *vi*

vi este un editor de texte care se găsește în majoritatea sistemelor UNIX. Există peste 20 de implementări de *vi*. Prima versiune a fost dezvoltată la Universitatea Berkeley în anul 1980.

În Linux este răspândită implementarea numită *VIM (Vi IMproved)*. Pentru a avea acces la editorul *vim*, trebuie instalate pachetele *vim-common*, *vim-minimal* și eventual *vim-enhanced*.

Ecranul *vi* conține un buffer – zona în care apare textul în curs de editare – și o linie de comandă, în partea de jos a ecranului. Pe această linie sunt afișate informații de stare și pot fi introduse comenzi. În zona bufferului, liniile care sunt goale (spre exemplu, atunci când textul are mai puține linii decât ecranul) vor conține caracterul *tilda*, „~”.

Editorul *vi* are trei moduri de lucru: modul de inserare, utilizat pentru scrierea textului, modul de comandă, folosit pentru a „naviga” în cadrul textului, pentru a salva fișierul în curs de editare, pentru a marca blocuri de text etc., și modul numit *ex*, care se utilizează pentru funcții adiționale, cum ar fi căutarea unui sir de caractere în cadrul textului etc.

vi pornește în mod comandă. Pentru a intra în modul de inserare se folosește tasta „i” (*Insert*). Pentru întoarcere la modul de comandă se apasă tasta Esc (*Cancel*). Tastele mai importante de navigare în cadrul textului (valabile în modul de comandă) sunt:

- h un caracter la stânga
- j o linie în jos
- k o linie în sus
- l un caracter la dreapta
- w un cuvânt înainte
- b un cuvânt înapoi
- \$ sfârșitul liniei
- 0 începutul liniei
- G începutul bufferului

Alte comenzi importante:

- x șterge un caracter
- d șterge un obiect. Prin obiect se înțelege un bloc de text, desemnat cu ajutorul comenziilor de navigare. Spre exemplu, comanda dw înseamnă „șterge următorul cuvânt”.
- dd șterge linia
- D șterge textul dintre cursor și sfârșitul liniei
- u reface ultima modificare adusă textului

Comenzile pentru ieșirea din vi:

- :wq salvează fișierul înainte de ieșire
- :q ieșe din vi doar dacă nu s-au făcut modificări și nu au fost salvate
- :q! ieșe din program chiar dacă modificările nu au fost salvate

Comenzile pentru salvarea fișierului sunt:

- :w salvează fișierul pe disc
- :w *fisier* salvează bufferul în fișierul specificat

Mai multe informații despre vi pot fi obținute din pagina de manual *man vi*.

Capitolul 2

COMPILATOARE

*Eu am o regulă simplă în viață:
dacă nu înțeleg ceva, înseamnă că este rău.
(Linus Torvalds)*

Compilarea programelor C este formată din trei faze independente, realizate de obicei de trei programe distincte:

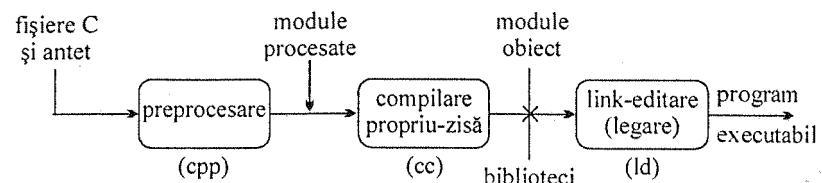


Figura 2.1. Fazele compilării

Utilizând anumite opțiuni (vezi paragraful următor), compilatorul poate fi determinat să parcurgă doar anumite faze.

Preprocesarea

Preprocesorul primește un text și generează tot un text. El analizează macrourile (care încep cu caracterul „#”, vezi partea I, capitolul 8 pentru detalii) și le expandează. Astfel, rezultatul preprocesării este tot un program-sursă C/C++, dar care nu mai include directive preprocesor, ci rezultatul acestora (de exemplu, o directive #include va avea ca efect includerea efectivă a codului C/C++ conținut în fișierul-antet corespunzător). Faza de preprocesare poate fi apelată prin:

```
gcc -E main.c > main.i
```

Compilarea

Rezultatul compilării este un *fișier-obiect* care reprezintă un amestec de cod mașină și alte informații necesare în faza de editare de legături (link-editare).

Fișierul-obiect conține în primul rând o listă a simbolurilor nesatisfăcute (adică declarate dar nedefinite), precum și locurile în care acestea sunt utilizate. Aceste simboluri sunt externe (definite în alte module). De asemenea, fișierul-obiect conține și o listă a simbolurilor exportate de către modul.

Fișierul-obiect mai conține și *informații de relocare*, acestea fiind necesare pentru că toate salturile se fac la niște adrese care se vor modifica în fază de editare de legături.

Nu în cele din urmă, obiectele pot conține și informații necesare pentru depanarea programului rezultat (vezi subcapitolul 3.1). Faza de compilare poate fi apelată cu:

```
gcc -c main.c
```

Rezultatul va fi un fișier-obiect cu numele `main.o`.

Editarea de legături

Legarea modulelor constă în completarea referințelor obiectelor globale, relocarea codului și adunarea tuturor modulelor într-un singur fișier executabil. Editarea de legături se face cu ajutorul programului `ld`, dar care nu trebuie apelat separat (lista fișierelor-obiect este foarte mare, fiindcă poate include numeroase biblioteci).

Bibliotecile C

O *bibliotecă* este o colecție de module-obiect deja compilate, strânse la un loc într-un *fișier-arhivă*. În general, declarațiile globale (variabile, constante, funcții etc.) se găsesc într-un fișier-antet furnizat împreună cu fișierul-arhivă. Crearea bibliotecilor se face cu ajutorul programului `ar` (pentru detalii vezi subcapitolul 2.3).

Există posibilitatea de a crea biblioteci speciale, denumite *biblioteci partajate* (*shared libraries*). Există un număr mare de funcții, în special în bibliotecile standard, foarte des utilizate în majoritatea programelor. Funcțiile trebuie încărcate în memorie o singură dată de către sistemul de operare, pentru a nu lega codul lor la toate executabilele, ceea ce ar duce la mărirea dimensiunii acestora. Ca rezultate pozitive, putem enumera: reducerea drastică a dimensiunii executabilelor, micșorarea timpului de încărcare a acestora și reducerea consumului de memorie.

În Linux, bibliotecile partajate sunt stocate în directoarele `/lib` și `/usr/lib`.

2.1. Compilarea programelor C: *GCC*

Cel mai răspândit compilator din lumea UNIX este *GCC*, implementare *open source* realizată de către Free Software Foundation. Aceasta este un compilator extensibil, existând extensii pentru C++, Objective-C, Pascal, Fortran etc. Executabilele generate de către acest compilator nu trebuie neapărat să fie *open source*, chiar dacă includ librăriile standard C sau C++.

Prima versiune a compilatorului *GCC* a fost scrisă de către Richard Stallman, inițiatorul proiectului *GNU*, prin 1986.

Sintaxa generală de apelare a compilatorului este:

```
gcc [ opțiuni nume_fisier ]
```

Tipul fișierelor de intrare este determinat după sufixul acestora, care sunt prelucrate după cum urmează:

- `.c` – sursă C: preprocesare, compilare, asamblare;
- `.C` – sursă C++: preprocesare, compilare, asamblare;
- `.cc` – sursă C++: preprocesare, compilare, asamblare;
- `.cxx` – sursă C++: preprocesare, compilare, asamblare;
- `.m` – sursă Objective-C: preprocesare, compilare, asamblare;
- `.i` – sursă C preprocesată: compilare, asamblare;
- `.ii` – sursă C++ preprocesată: compilare, asamblare;
- `.s` – sursă în limbaj de asamblare: asamblare;
- `.S` – sursă în limbaj de asamblare: preprocesare, asamblare.

Fișierele având alte sufixe, cum ar fi `.o` (fișier-obiect) sau `.a` (fișier-arhivă), sunt trimise către link-editor.

Pentru a se evita o parte din etapele prelucrării pot fi utilizate următoarele opțiuni:

- `-c` compilează și asamblează fișierele-sursă, dar nu le link-editează;
- `-S` compilează fișierele-sursă, fără însă a le asambla;
- `-E` nu lansează compilarea, ci doar preprocesarea.

Implicit, `gcc` generează un fișier executabil având denumirea `a.out`; fișierele-obiect corespunzătoare intrării `sursa.sufix` sunt denumite `sursa.o`, iar fișierele asamblate `sursa.s`.

Denumirea ieșirii poate fi schimbată cu ajutorul opțiunii `-o fisier`.

Spre exemplu, pentru compilarea unui program C:

```
gcc -o cap7_1 cap7_1.c
```

Alte opțiuni des utilizate:

Opțiuni la compilare:

- `-Idirector` adaugă `director` la lista directoarelor în care sunt căutate fișierele `.h`;
- `-Ldirector` adaugă `director` la lista directoarelor în care sunt căutate bibliotecile;
- `-g` include informații de depanare în fișierul-obiect generat. Această opțiune este necesară dacă se intenționează depanarea programului cu GDB, de exemplu;

- `-O, -O1, -O2, -O3` activează diverse niveluri de optimizare a codului. `-O3` este nivelul cel mai avansat de optimizare;
- `-O0` dezactivează optimizarea codului.

Opțiuni la editarea de legături:

- `-lbiblioteca` utilizează biblioteca specificată în cadrul etapei de editare de legături.

Opțiuni la preprocesare:

- `-Dmacro` definește macroul specificat ca fiind 1;
- `-Dmacro=valoare` definește macroul cu valoarea specificată;
- `-Umacro` anulează macroul specificat.

Pentru exemplificare, compilarea unui program care utilizează biblioteca ncurses:

```
gcc -o test test.c -I/usr/include/ncurses -lncurses
```

2.2. Compilarea programelor C++: G++

Compilatorul G++ se utilizează pentru compilarea programelor C++. Sintaxa de utilizare este identică cu cea a GCC.

Opțiunile mai des utilizate sunt:

- `-fall-virtual` tratează toate funcțiile membre ca fiind virtuale;
- `-fthis-is-variable` permite utilizarea pointer-ului `this` ca o variabilă obișnuită;
- `-fexternal-templates` produce cod obiect mai mic pentru declarațiile de template-uri, generând doar o singură copie a fiecărei funcții template acolo unde aceasta este definită. Pentru a putea utiliza această opțiune, trebuie marcate toate fișierele-sursă care utilizează template-uri cu directiva `#pragma implementation` (acolo unde sunt definite efectiv template-urile), respectiv `#pragma interface` (acolo unde sunt declarate template-urile). Atunci când este utilizată această opțiune, toate instanțierile de template-uri sunt considerate `external`. De aceea, toate aceste instanțieri trebuie să fie realizate în cadrul fișierului marcat cu `implementation`, de exemplu prin utilizarea unor declarații `typedef` care să facă referire la fiecare instanțiere;

- `-falt-external-templates` are un comportament similar cu cel al opțiunii precedente, cu excepția faptului că este generată o singură copie a fiecărei funcții template acolo unde aceasta este definită pentru prima oară.

2.3. Crearea de biblioteci: ar

Utilitarul `ar` se folosește pentru a crea, modifica și extrage fișiere dintr-o arhivă, în speță bibliotecă de tip static.

Pentru a crea o bibliotecă statică, trebuie mai întâi generate fișierele-obiect, apoi creată arhiva:

```
ar -r libtest.a functii.o test.o
ranlib libtest.a
```

Prima comandă va crea arhiva `libtest.a`, conținând fișierele-obiect `functii.o` și `test.o`. Cea de-a doua comandă generează un index pentru arhiva nou creată.

Sintaxa programului `ar` este următoarea:

```
ar comanda arhiva [membru...]
```

unde `comandă` poate fi:

- `d` sterge module din arhivă;
- `p` afișează membrii specificați (sau toți membrii dacă nu este specificat nici unul) din arhivă;
- `r` inserează un nou membru în arhivă;
- `t` afișează conținutul unei arhive;
- `x` extrage un membru din arhivă.

Pentru a crea biblioteci partajate, se va folosi compilatorul `gcc` (sau `g++` pentru programe C++):

```
gcc -shared -o libtest.so functii.o test.o
```

Bibliotecile partajate trebuie instalate în genere în directorul `/usr/lib`, apoi trebuie executată comanda `ldconfig` pentru a actualiza baza de date de biblioteci (`cache`).

Capitolul 3

INSTRUMENTE DE PROGRAMARE

Ideea că există o singură modalitate corectă de a rezolva orice problemă pentru oricare utilizator este fundamental greșită.
(Bjarne Stroustrup)

3.1. Depanarea programelor: GDB

Un program de depanare, cum ar fi GDB, se utilizează pentru a putea monitoriza ce se întâmplă într-un program în timpul execuției sale sau în momentul în care și-a încheiat execuția în mod forțat (*crashed*). Autorul original al programului GDB este Richard Stallman, care a început dezvoltarea acestuia în anul 1988.

Pentru folosirea acestui program, trebuie instalat pachetul `gdb`.

GDB poate fi utilizat atât pentru depanarea programelor scrise în C cât și a celor scrise în C++. Acesta poate opri din execuție un program în momentul întâlnirii unei condiții specificate de programator, examina conținutul variabilelor la momentul respectiv, și chiar modifica valoarea acestora și continua execuția programului, pentru a putea descoperi mai ușor erorile din program în anumite condiții. GDB poate fi folosit și pentru a depana programe care sunt executate pe mașini aflate la distanță.

Pentru ca un program să poată fi depanat, trebuie să fie compilat cu opțiunea `-g`, care activează includerea informațiilor de depanare în fișierul-object generat.

Modalitatea cea mai simplă de a lansa în execuție GDB este de a-i furniza ca parametru numele fișierului executabil al programului pe care dorim să îl executăm:

`gdb program`

sau:

`gdb numar_proces`

Pentru a depana procesul specificat. De asemenea, pentru a putea trimite parametri programului apelat, se folosește sintaxa:

`gdb --args program argumente`

Pentru a executa programul într-un alt terminal decât cel curent, pe care va rula GDB, se utilizează opțiunea `-tty dispositiv`. Aceasta este utilă dacă, spre exemplu, programul utilizează biblioteca ncurses și face afișarea pe tot ecranul, caz în care ar fi imposibilă execuția acestuia pe același terminal cu GDB. Sub Linux, este ușor de comutat între terminale. Ca exemplu, GDB poate fi lansat în execuție de pe consola virtuală 1 (tty1), iar programul să aibă ieșirea și intrarea pe consola virtuală 7 (tty7). Comutarea se face cu ajutorul combinației de taste Alt+F1, respectiv Alt+F7. De asemenea, această facilitate este utilă pentru a executa programul sub o altă fereastră X Window. Pentru a stabili terminalul pe care va rula programul după intrarea în GDB, se utilizează comanda internă `tty terminal` (vezi *infra*).

GDB are o interfață de tip *shell*, care permite execuția de comenzi interne. Pot fi accesate cu ajutorul tastelor săgeți sus, respectiv jos comenziile GDB apelate în trecut. Promptul afișat pentru a arăta că se așteaptă comenzi este `(gdb)`. Ieșirea din GDB se face prin comanda `quit`.

Lansarea în execuție a programului se face prin comanda `run`. În cadrul GDB, un program poate fi oprit datorită unui semnal, unui punct de oprire (*breakpoint*) sau la solicitarea persoanei care îl depaneză. Starea programului poate fi consultată în orice moment cu ajutorul comenzi `info program`.

Puncte de oprire

Un punct de oprire (*breakpoint*) oprește programul atunci când ajunge la un anumit punct din codul-sursă. Acesta se definește cu ajutorul comenzi `break`. Astfel, `break funcție` va seta un punct de oprire la intrarea în funcția specificată. Comanda `break numar_linie` stabilește un punct de oprire în cadrul fișierului-sursă curent. În mod similar, `break fisier:numar_linie` respectiv `break fisier:funcție` stabilesc puncte de oprire în linia, respectiv funcția din fișierul specificat. `break` fără argumente setează un punct de oprire la următoarea instrucțiune care se va executa. Comanda `info breakpoints` afișează punctele de oprire definite.

Un punct de verificare (*watchpoint*) oprește programul atunci când se modifică valoarea unei expresii, fără a ști cu precizie locul unde se întâmplă acest eveniment. Asemenea puncte de verificare se stabilesc prin comanda `watch expresie`. Atunci când expresia este întâlnită în program și valoarea ei se modifică, programul va fi oprit. Comanda `info watchpoints` afișează punctele de verificare definite.

Un punct de interceptare (*catchpoint*) oprește programul atunci când intervin evenimente speciale, cum ar fi o excepție C++ sau încărcarea unei biblioteci.

Comanda `catch throw` va opri programul atunci când se tratează o excepție C++, `catch catch` când se întâlnește o interceptare a unei excepții C++.

Pentru a șterge punctele de oprire, de verificare sau de interceptare, se folosește comanda `clear`, cu sintaxa `clear [fisier:]functie, clear [fisier:]functie` sau `clear [interval]` care șterge intervalul de puncte de oprire specificate, sau toate punctele dacă intervalul nu este specificat.

GDB permite de asemenea detectarea semnalelor UNIX apărute în timpul execuției programului și modificarea comportamentului său la întâlnirea unui asemenea semnal. Astfel, comanda `handle semnal mod_tratare` stabilește modul de tratare a semnalului specificat. Aceasta poate fi specificat atât prin valoarea sa numerică, cât și prin numele său. Poate lua și valoarea `all`, care desemnează toate tipurile de semnale. Parametrul `mod_tratare` poate lua una dintre următoarele valori:

- `nostop` programul nu se va opri la apariția semnalului;
- `stop` programul se va opri la apariția semnalului;
- `print` GDB va afișa un mesaj la apariția semnalului;
- `no/print` nu afișează nici un mesaj;
- `pass` permite ca programul să detecteze semnalul. Este posibil ca execuția programului să se încheie dacă acesta nu a prevăzut tratarea respectivului semnal;
- `ignore` nu permite programului să detecteze semnalul.

Continuarea execuției programului

Continuarea execuției programului atunci când acesta s-a oprit se face cu ajutorul comenzi `continue`. Comanda `step` continuă execuția programului până la următoarea linie de cod. Comanda `next` este similară cu `step`, cu excepția faptului că apelurile de funcții sunt executate fără oprire (cu alte cuvinte, fără a intra în funcția respectivă).

Inspectarea variabilelor

În orice moment, pot fi inspectate valorile variabilelor din cadrul programului. Aceasta se face cu ajutorul comenzi `print expresie`. Comanda poate primi ca parametru și o expresie (în limbajul C), nu neapărat o variabilă. Este permisă și utilizarea operatorului de rezoluție, `::` din C++. Variabilele statice dintr-un anumit fișier sau funcție pot fi inspectate folosind formatul `fisier::variabila`, respectiv `functie::variabila`. Spre exemplu, pentru a vizualiza valoarea variabilei globale `count`, definite în fișierul `xobject.cc`:

```
print 'xobject.cc'::count
```

Vom reda în continuare un exemplu de depanare a programului `atoi`, prezentat în paragraful 4.2:

```
$ gcc -g -o atoi atoi.c
$ gdb atoi
GNU gdb Red Hat Linux (5.1.90CVS-5)
Copyright 2002 Free Software Foundation, Inc.
(gdb) break 14
Breakpoint 1 at 0x8048499: file atoi.c, line 14
(gdb) run
Starting program: /home/dragos/atoi

Breaking 1, atoi (s=0x8048598 "1952") at atoi.c:14
14      for(n = 0; s[i] >= '0' && s[i] <= '9'; i++)
(gdb) print i
$1 = 0
(gdb) continue
Continuing.
x = 1952

Program exited normally.
(gdb) quit
```

Mai multe informații despre GDB pot fi obținute apelând comanda `info gdb`.

3.2. Utilitarul `make`

Utilitarul `make` se folosește în principal pentru a întreține un set de fișiere cu interdependențe, care fac parte dintr-un proiect software. `Make` oferă un sistem de codificare a relațiilor de legătură dintre fișiere precum și a comenziilor care trebuie utilizate pentru prelucrarea acestor fișiere atunci când sunt modificate. Acest utilitar servește la automatizarea compilării programelor complexe. Majoritatea programelor distribuite în format-sursă din Linux utilizează acest utilitar. Prima implementare a `GNU make` a fost realizată de Richard Stallman și Roland McGrath, dezvoltarea fiind continuată de către Paul D. Smith.

În vederea utilizării programului `make`, trebuie instalat pachetul cu același nume.

Sistemul constă dintr-un fișier `makefile`. Acesta conține mai multe linii, care pot fi de patru tipuri: scop (`target`), comenzi shell, macrouri și directive. Comentariile încep cu caracterul `„#”`.

La apelarea comenzi `make`, aceasta caută în directorul curent mai întâi fișierul `makefile`, iar apoi fișierul `Makefile`.

Linii de scop

Acestea specifică ceea ce poate fi construit (e.g. compilat). Linile de scop constau dintr-o listă de fișiere scop, următe de caracterul „:”, apoi de lista de dependințe. Această listă poate fi și vidă. De exemplu:

```
doc++: comment.cc main.cc readfiles.cc
```

Lista de dependințe conține scopuri, care trebuie să fie mai vechi decât scopul curent după execuția comenzi make. Scopul trebuie să fie mai recent decât scopurile de care depinde. Dacă vreuna dintre dependințe este mai recentă decât scopul curent sau dacă dependința nu există, aceasta trebuie construită mai întâi. Dacă lista de dependințe este vidă, scopul este întotdeauna construit.

În cadrul listei de dependințe pot fi utilizate și caractere wildcard, similare cu cele utilizate în interpretorul de comenzi (*shell*) Bash: „*”, „?” și „[]”:

```
doc++: *.cc
```

Fișierele din lista de dependințe pot fi și de tip obiect, de tip header etc., însă nu este obligatoriu ca acestea să intre în componenta scopului (e.g. executabilului).

Reguli

Make oferă posibilitatea utilizării unor scopuri generice, cunoscute sub numele de *reguli de sufix*, *reguli de inferență* sau doar *reguli*. Regulile sunt modalități de a specifica doar o singură dată modul de construire al unui scop. Spre exemplu, regula de mai jos specifică modul de transformare a unui fișier-sursă .cc într-un fișier-obiect .o:

```
# genereaza fisierele obiect .o prin compilarea
# tuturor fisierelor-sursa cu extensia .cc
.cc.o:
    g++ -c $< -I. -I/usr/local/include
# genereaza fisierul executabil doc++
# prin link-editarea fisierelor-obiect componente
doc++: comment.o main.o readfiles.o
    g++ comment.o main.o readfiles.o
```

construcție care este echivalentă cu:

```
# compileaza comment.cc
comment.o: comment.cc
    g++ -c comment.cc -I. -I/usr/local/include
# compileaza main.cc
main.o: main.cc
    g++ -c main.cc -I. -I/usr/local/include
# compileaza readfiles.cc
```

```
readfiles.o: readfiles.cc
    g++ -c readfiles.cc -I. -I/usr/local/include
# genereaza fisierul executabil doc++
# prin link-editarea fisierelor obiect componente
main: comment.o main.o readfiles.o
    g++ comment.o main.o readfiles.o
```

În exemplul de mai sus a fost utilizat un macrou special, \$<, care substituie fișierul-sursă curent.

Tipul fișierului este determinat după extensia sa (șirul de caractere de după punct „.”). Dacă există vreun fișier de tipul respectiv, regula de sufix va fi aplicată.

Linii de comenzi shell

Linile de comenzi shell, cunoscute și sub numele de comenzi, definesc acțiunile care sunt executate pentru a construi un scop. Textul care urmează după caracterul „;” în cadrul unei linii de scop este considerat o comandă. De asemenea, toate liniile care urmează după un scop și care încep cu caracterul *tab* sunt considerate comenzi care vor fi executate în vederea construirii scopului.

Liniile lungi de comenzi pot fi continuăte pe linia următoare adăugând la sfârșit caracterul „\”.

În mod normal, comenziile sunt afișate la ieșirea standard. Acest lucru poate fi prevenit prin prefixarea comenzi cu caracterul *at*, „@”.

Macrouri

Macrourile au următoarele roluri:

- scutesc munca programatorului care scrie fișierul makefile;
- ușurează întreținerea fișierului permitând specificarea informațiilor într-un singur loc. Atunci când valoarea unui macrou se modifică, aceasta se propagă în mod automat în tot fișierul. Spre exemplu, lista de fișiere-obiect care vor fi construite poate fi specificată prin intermediul unui macrou;
- oferă un mod de a introduce variabile într-un makefile, cunoscând informațiile care vor fi modificate. De exemplu, modul de compilare a unui program cu sau fără informații de depanare poate fi determinat printr-un macrou. Modificând valoarea acestuia, în linia de comandă sau printr-o variabilă de mediu, modul de compilare poate fi stabilit fără a modifica fișierul makefile;
- ușurează posibilitatea de citire a fișierului makefile, reducând numărul de caractere conținute.

Macrourile pot fi definite prin definiții interne fișierului makefile, prin variabile de mediu, sau în linia de comandă. În cadrul fișierului acestea pot fi definite astfel:

```
nume = valoare
```

Numele macrourilor sunt caractere mari prin convenție. *valoare* poate fi și o listă de valori. De asemenea, definițiile pot include alte macrouri:

```
INCLUDES=/usr/local/include /usr/include/Xterminal
OBJECTS=test.o xterminal.o
OBJS=$(OBJECTS) main.o
```

După cum s-a putut observa și din exemplul de mai sus, macrourile pot fi referite prin intermediu construcțiilor:

```
$(macro)
${macro}
```

Macrouri predefinite

Utilitarul make oferă o serie de macrouri speciale, predefinire, pentru a facilita utilizarea regulilor generice. Aceste macrouri pot fi referite fără a mai utiliza parantezele. De asemenea, acestora nu le poate fi atribuită nici o valoare, aceasta depinzând de starea în care se află make în momentul în care macroul este evaluat. Macrourile predefinite sunt următoarele:

Macrou	Descrierea macroului
\$@	Numele scopului curent
\$%	Valoarea membrului curent al arhivei, doar în cazul în care scopul curent este o bibliotecă
\$?	Lista de dependințe care sunt învechite pentru scopul curent
\$<	Fișierul-sursă curent, altfel spus fișierul care se prelucrează în momentul întâlnirii acestui macrou. Fișierul-sursă curent reprezintă fișierul care este învechit în comparație cu scopul curent
\$*	Rădăcina scopului curent. Spre exemplu, dacă scopul se numește <code>readfiles.o</code> , rădăcina are sufixul <code>.o</code> șters, deci este <code>readfiles</code>
\$\$@	Este similar cu \$@, însă se utilizează pentru executabilele alcătuite dintr-un singur fișier-sursă

Mai există câteva macrouri predefinite, dar care pot fi modificate de către programator, dintre care mai importante sunt:

Macrou	Descrierea macroului
VPATH	Lista de directoare în care make va căuta fișierele dependință. Elementele listei se despart cu ajutorul caracterului „:”
SHELL	Interpretorul de comenzi utilizat pentru a executa liniile de comandă

Directive

Fiecare implementare a utilitarului make (sub Linux se utilizează GNU make, gmake) conține propriile directive. Directivele „#” (comentariu) și include sunt prezente totuși în toate implementările. Astfel, directiva include se utilizează pentru a include într-un makefile definițiile făcute într-un alt makefile, pentru a reduce redundanța acestora.

Argumente în linia de comandă

Argumentele uzuale sunt:

- `-f fisier` stabilește numele fișierului makefile care va fi procesat;
- `-p` afișează toate definițiile de macrouri și regulile;
- `-n` nu execută comenzi, doar le afișează la ieșirea standard;
- `-s` activează modul de lucru silentios, în sensul că la ieșire comenziile nu vor fi afișate.

Mai multe informații privind make pot fi obținute apelând `info make`.

Iată un exemplu de fișier makefile care are drept scop „compilarea” unui fișier LaTeX în vederea obținerii documentului Postscript final:

```
# fisier folosit pentru compilarea cartii
# versiunea 0.9 - Sabin Cornelius Buraga <busaco@infoiasi.ro>
# ultima actualizare: 17 iulie 2001

# fisierul principal al cartii
SRC = main

# operațiunile complete (make)
all: tex idx ps

# doar compilare (make tex)
tex: $(SRC).tex
    latex $(SRC).tex

# generare index (make idx)
idx:
    makeindex $(SRC).idx
    latex $(SRC).tex

# generare PostScript (make ps)
ps:
    dvips $(SRC).dvi -o $(SRC).ps

# generare PDF (make pdf)
pdf:
    ps2pdf $(SRC).ps

# vizualizare PostScript cu gv (make gv)
gv:
```

```

gv $(SRC).ps &

# archiveaza fisierele (make arh)
arh:
    tar -zcf retele-$(SRC).tgz *.tex Makefile *.sty *.cls eps/*
    tar -zxf retele-$(SRC).tgz

# dezarchiveaza fisierele (make unarch)
unarch:
    tar -zxf retele-$(SRC).tgz

# sterge fisierele inutile (make clean)
clean:
    rm *.aux *.dvi *.i?? *.l?? *.toc */

```

3.3. autoconf, automake și libtool

Programele autoconf, automake și libtool sunt utile pentru a simplifica compilarea aplicațiilor complexe pe diferite platforme și pentru a crește gradul de portabilitate a acestora.

Autoconf ușurează portarea programelor pe diferite platforme prin determinarea caracteristicilor sistemului pe care se face compilarea înainte ca aceasta să înceapă (versiunile bibliotecilor instalate, existența anumitor biblioteci etc.). Codul-sursă se poate adapta astfel mai ușor la diferențele existente între platforme. Prima versiune autoconf a fost realizată în anul 1992 de către David MacKenzie.

Pentru a putea folosi aceste programe, trebuie instalate pachetele cu aceleași nume.

Automake este un instrument care generează în mod automat fișiere makefile, simplificând mult munca de realizare a acestora. El oferă de asemenea diverse funcții utile precum determinarea automată a dependențelor dintre fișiere. Dezvoltarea programului automake a fost începută în anul 1994 de către David MacKenzie și Tom Tromey.

Libtool este o interfață pentru compilator și link-editor care permite generarea ușoară a bibliotecilor, fie ele statice sau dinamice, independent de platformă. Utilitarul libtool a fost creat de către Gordon Matzigkeit în anul 1996.

Aceste trei instrumente sunt programe distincte, însă între ele există o legătură strânsă (vezi *infra*).

Vom numi în continuare un arbore de cod-sursă a unui program, împreună cu alte fișiere cum ar fi documentații, fișiere de configurare etc., *pachet*. Un pachet construit cu autoconf conține un script denumit *configure*. Pentru a construi (compila) programul, trebuie executat mai întâi acest script pentru a determina particularitățile sistemului și a pregăti codul-sursă în acest sens. Pentru compilare se va apela comanda *make*.

Opțiunile cele mai des utilizate la apelarea scriptului *configure* sunt:

- *--prefix=director* stabilește directorul în care vor fi instalate fișierele independente de arhitectură ale pachetului (e.g. documentații);
- *--exec-prefix=director* stabilește directorul în care vor fi instalate fișierele dependente de arhitectură, cum ar fi executabile sau biblioteci. Dacă nu este folosită, va fi utilizată valoarea furnizată opțiunii *--prefix*;
- *--bindir=director* specifică destinația în care vor fi instalate fișierele executabile care vor fi apelate de către utilizatori;
- *--libexecdir=director* specifică directorul în care vor fi instalate bibliotecile utilizate de către programele generate;
- *--libdir=director* atunci când pachetul furnizează biblioteci, această opțiune stabilește directorul în care acestea vor fi instalate;
- *--disable-optiune* dezactivează includerea opțiunii (*feature*) specificate. Aceste opțiuni pot activa anumite porțiuni de cod sau implica compilarea condiționată a unor chestiuni experimentale, de exemplu *--disable-debug* va dezactiva generarea informațiilor pentru depanare;
- *--enable-optiune* este similară cu opțiunea precedentă, dar activează opțiunea specificată. Există posibilitatea ca această opțiune să primească un argument, de exemplu *--enable-buffers=64*;
- *--with-pachet* este similară cu opțiunea *--enable-optiune*, dar se referă în general la includerea unei biblioteci sau a unui alt pachet;
- *--without-pachet* opusul opțiunii *--with-pachet*.

La execuția scriptului *configure*, sunt generate o serie de fișiere și eventual directoare. Fișierele generate sunt:

- *config.cache* memorează variabilele dependente de sistem generate, pentru a evita determinarea lor de mai multe ori;
- *config.log* conține jurnalul testelor și rezultatelor acestora, incluzând comenzi shell apelate și mesajele generate;
- *config.status* este un script în care fișierele sunt generate conform configurațiilor curente;
- *config.h* fișier-antet (C/C++) care conține directive preprocesor care descriu configurațiile curente. Acestea sunt necesare pentru programele din cadrul pachetului pentru a determina respectivele configurații. Iată un fragment dintr-un asemenea fișier-antet:

```

/* Define if you have the <unistd.h> header file. */
#define HAVE_UNISTD_H

/* Name of the package */
#define PACKAGE "doc++"

/* Version number of the package */

```

```
#define VERSION "3.4.9"

/* Define if the GNU gettext() function is already present or
preinstalled */
#define HAVE_GETTEXT 1
```

- Makefile fișierul makefile care va fi utilizat de comanda make pentru a construi fișierele necesare. În general este generat pornind de la fișierul Makefile.in.

În cadrul fișierului makefile generat, sunt definite câteva scopuri în vederea manipulării pachetului:

- make all construiește toate fișierele necesare;
- make install instalează pachetul;
- make clean șterge toate fișierele generate.

Pentru a înțelege mai ușor structura și modul de funcționare a perechii de programe autoconf-automake, vom considera un proiect ipotetic denumit aa.

Scriptul configure este generat în mod automat cu ajutorul comenzi autoconf, pornind de la un fișier denumit configure.in. Acesta conține macrouri și cod shell.

Iată în continuare conținutul fișierului configure.in pentru proiectul în discuție:

```
dnl Comentariile incep cu dnl
dnl Acest fisier trebuie procesat cu ajutorul comenzi autoconf
dnl pentru a produce scriptul de configurare
AC_INIT(main.c)
AM_INIT_AUTOMAKE(aa, 1.0)
AC_PROG_CC
AC_OUTPUT(Makefile)
```

Macroul AC_INIT realizează o serie de inițializări și stabilește ca fișier principal al proiectului main.c. Acesta este utilizat pentru a determina dacă fișierele-sursă se găsesc în directoarele corespunzătoare. Apelul acestui macrou este obligatoriu.

Macroul AC_INIT_AUTOMAKE inițializează automake, specificând numele proiectului și versiunea curentă. Si apelul acestui macrou este obligatoriu dacă se utilizează automake.

Macroul AC_PROG_CC verifică instalarea corectă a compilatorului de C. Macroul AC_OUTPUT stabilește care este fișierul care va fi generat. Fișierul makefile se generează pornind de la un fișier makefile „șablon”, denumit Makefile.in.

Fișierul Makefile.in este generat în mod automat de către utilitarul automake pornind de la fișierul denumit Makefile.am. Mai jos prezentăm conținutul acestui fișier pentru proiectul nostru:

```
bin_PROGRAMS = aa
aa_SOURCES = main.c functii.c functii.h
aa_LDADD = -lncurses
```

Prima directivă, bin_PROGRAMS, specifică lista de programe executabile care vor fi generate. A doua directivă, având formatul binar_SOURCES, specifică fișierele-sursă necesare pentru a construi executabilul în cauză, inclusiv cele antet. În final, directiva aa_LDADD desemnează bibliotecile care trebuie link-editate împreună cu executabilul final.

După crearea acestor două fișiere, trebuie executată comanda:

```
aclocal; autoconf
```

Comanda aclocal se apelează pentru a procesa macrourile care nu sunt cunoscute de autoconf, cum este în cazul nostru macroul AC_INIT_AUTOMAKE. Această comandă generează scriptul aclocal.m4, în limbaj M4.

După aceasta, vom apela comanda automake --add-missing pentru a crea fișierul Makefile.in și a copia câteva scripturi care vor fi necesare ulterior. Opțiunea menționată a fost inclusă deoarece în directorul curent al proiectului nostru nu se află o serie de fișiere standardizate prin convenție:

- AUTHORS lista de persoane, însăși de regulă și de adresele lor de e-mail, care au contribuit la proiect;
- ChangeLog jurnalizează modificările aduse proiectului;
- README include informații generale despre proiect, inclusiv modul de instalare etc.;
- NEWS lista de modificări importante, vizibile utilizatorilor.

ACEste fișiere sunt conforme standardelor GNU, dar nu influențează în vreun fel funcționarea programelor autoconf sau automake.

Este important să notăm aici faptul că, după orice modificare adusă fișierelor de configurare autoconf și automake, trebuie apelate comenziile menționate mai sus pentru a regenera fișierele.

Structurarea apelurilor de comenzi și macrouri în cadrul fișierului configure.in se face în următoarele secțiuni, în ordinea menționată:

1. Macrourile de bază: AC_INIT (care trebuie să fie primul macrou apelat), AM_INIT_AUTOMAKE, AC_CONFIG_HEADER.
2. Opțiunile din linia de comandă configure, cum ar fi AC_ARG_ENABLE.
3. Programele care sunt necesare în cadrul procesului de configurare, sau pentru programul care va fi compilat, utilizând macrouri de genul AC_CHECK_PROG sau AC_PATH_TOOL.
4. Bibliotecile necesare în cadrul testelor din procesul de configurare, sau de programul care va fi compilat, prin macroul AC_CHECK_LIB.

5. Fișierele-antet (dacă acestea există).
6. Definițiile de tipuri sau structuri.
7. Funcțiile utilizate, cu ajutorul macroului AC_CHECK_FUNCS.
8. Fișierele generate, cum ar fi fișierele makefile, utilizând macroul AC_OUTPUT.

Este bine ca la începutul lui `configure.in` să se apeleze macroul `AC_CANONICAL_HOST`, care determină numele standardizat (*canonical system name*) al sistemului sau *numele configurației*. Aceste nume sunt alcătuite din trei părți, având formatul *procesor-producător-sistem_de_operare*. De asemenea, pe sistemele în care nucleul diferă de sistemul de operare, cum ar fi GNU/Linux, numele poate avea formatul *procesor-producător-nucleu-sistem_de_operare*. Spre exemplu, numele standardizat pentru un sistem Linux este de genul `i586-pc-linux-gnu`. Alt exemplu este `sparc-sun-solaris2.4`.

Să ne reamintim conținutul fișierului `Makefile.am` pentru proiectul aa:

```
bin_PROGRAMS = aa
aa_SOURCES = main.c functii.c functii.h
aa_LDADD = -lncurses
```

Într-un mod similar cu declararea programelor care vor fi construite (compilate) prin directiva `bin_PROGRAMS`, există posibilitatea realizării și de alte scopuri, după cum urmează:

- DATA definește fișierele care fac parte din proiect (cu alte cuvinte vor fi instalate), dar nu necesită vreo prelucrare pentru a fi construite;
- HEADERS declară fișierele-antet;
- SCRIPTS declară scripturile executabile;
- MANS declară fișierele manual ale proiectului. Există două moduri de utilizare a acestui macrou:
 - man_MANS atunci când fișierele au deja prefixul care semnifică secțiunea de manual căreia îi aparțin, stabilind, cum ar fi `docify.1`
 - man1_MANS atunci când fișierele nu au prefixul stabilit, cum ar fi `equate.man`.

Atunci când proiectul are mai multe subdirectoare, fișierul `Makefile.am` din directorul (sau directoarele) rădăcină trebuie să conțină un macrou de forma:

```
SUBDIRS = doc src
```

Subdirectoarele vor fi procesate în ordinea specificată.

În mod tradițional, structura unui arbore-sursă pentru un proiect conține un director denumit `src` pentru fișiere-sursă, un director `doc` pentru documentații și, eventual, un director `test` pentru programe de test.

Vom prezenta în cele ce urmează programul libtool.

Pentru a utiliza libtool, la începutul fișierului `configure.in` trebuie apelat macroul `AC_PROG_LIBTOOL`. Vor fi disponibile mai multe opțiuni noi, dintre care mai importante sunt:

- `--enable-shared` activează construirea de biblioteci partajate;
- `--disable-shared` este complementul opțiunii precedente;
- `--enable-static` activează construirea de biblioteci statice;
- `--disable-static` este reversul opțiunii precedente.

În cele ce urmează vom presupune că proiectul nostru este o bibliotecă. Fișierul `Makefile.am` va avea următorul conținut:

```
lib_LTLIBRARIES = libaa.la
libaa_la_SOURCES = functii.cc
```

Se observă similitudinea cu declarațiile `bin_PROGRAMS`, respectiv `aa_SOURCES`. Proiectul nostru nu conține decât fișierul-sursă `functii.cc`.

În mod tradițional, pe lângă directoarele menționate mai sus, arborele-sursă mai trebuie să conțină și directorul `include`, care va include fișierele-antet (în general cele care vor fi instalate, adică vizibile altor programatori care vor utiliza biblioteca). Fișierul `Makefile.am` din cadrul acestui director trebuie să conțină o directivă `include_HEADERS`.

După crearea fișierelor `configure.in` și `Makefile.am`, trebuie copiat conținutul scriptului `libtool.m4`, integrat în pachetul `libtool` (fișier localizat în general în directorul `/usr/share/libtool/`), în directorul rădăcină al proiectului, sub numele de `acinit.m4`. Pasul următor este execuția comenziilor:

```
aclocal
libtoolize
```

În urma execuției comenzi `libtoolize` poate fi remarcată apariția unor fișiere noi, `ltconfig` și `ltmain.sh`.

În cazul bibliotecilor, este importantă gestionarea corectă a versiunilor, deoarece o schimbare de versiune poate aduce și modificări ale funcțiilor, fișierelor-antet etc. Un program care utilizează biblioteca va avea probleme la execuție sau este posibil chiar să nu funcționeze deloc. `libtool` posedă un sistem propriu de control al versiunilor, permitând chiar existența mai multor versiuni de biblioteci instalate.

Formatul versiunii unei biblioteci este:

```
current:revizie:varsta
```

unde:

- *current* reprezintă interfața curentă exportată de către bibliotecă (conținutul fișierelor-antet vizibile altor programatori);
- *revizie* reprezintă implementarea curentă a interfeței exportate (dacă interfața rămâne neschimbătă dar se modifică implementarea internă a unor funcții, *current* va rămâne nemodificată și va fi incrementat *revizie*);
- *vârstă* semnifică numărul de interfețe mai vechi suportate de versiunea curentă a bibliotecii (cu alte cuvinte, compatibile cu aceasta). *vârstă* trebuie să fie mai mic sau cel mult egal cu *current*.

Regulile de modificare a acestor trei câmpuri sunt următoarele:

1. În cazul în care codul-sursă a fost modificat, trebuie incrementat câmpul *revizie*. Vom spune că avem o nouă revizie a interfeței curente.
2. Dacă interfața a fost modificată, *current* trebuie incrementat, iar *revizie* pus pe zero. Spunem că este vorba de prima revizie a unei noi interfețe.
3. Dacă noua interfață a adăugat noi elemente, dar nu a modificat interfața versiunii precedente (deci programele vechi sunt compatibile cu noua versiune), trebuie incrementat câmpul *vârstă*. Spunem aşadar că această versiune este compatibilă cu versiunea precedentă.
4. Dacă în noua interfață s-au eliminat componente, programele vechi nu vor mai fi compatibile cu aceasta, și va trebui ca *vârstă* să fie pus pe zero.

Pentru o mai bună înțelegere a utilizării acestor trei programe, vom prezenta în continuare două exemple reale, unul pentru un program executabil și unul pentru o bibliotecă.

Primul exemplu este programul SAL, pentru evidența personalului și a salariilor. Iată structura de directoare a arborelui surselor, în care fișierele care ne interesează au fost îngroșate:

```
..:  
acconfig.h config.guess configure doc lista missing  
aclocal.m4 config.status configure.in install Makefile.am  
mkinstalldirs  
build config.sub devel install-sh Makefile.in src  
  
.build:  
coeficienti.data create_global create_luna create_luna_install  
impozit.data  
  
.devel:  
sal-1.0.spec  
  
.doc:  
ChangeLog  
  
.install:  
install main.cc Makefile.am
```

```
./src:  
bonuri.cc fisafisc1.cc general.h main.h pontaj.cc stat.cc  
cas.cc fisafisc2.cc init.cc Makefile.am salarii.cc useradaug.cc  
config.h.in fluturi.cc initluna.cc Makefile.in salcalc.cc  
usersterg.cc  
conflunian.cc general.cc main.cc personal.cc stamp-h.in
```

Makefile.am:

```
AUTOMAKE_OPTIONS = foreign  
SUBDIRS = install src
```

acconfig.h:

```
/* The package name. */  
#undef PACKAGE  
  
/* The package version. */  
#undef VERSION  
  
/* Define if you have the `crypt' function. */  
#undef HAVE_CRYPT  
  
/* Define if you are using `PostgreSQL' as SQL server. */  
#undef HAVE_LIBPQ
```

configure.in:

```
dnl configure.in for SAL  
dnl Copyright (c) 1999-2001 Dragos Acostachioae  
  
dnl Declara fisierul principal al proiectului  
AC_INIT(src/main.cc)  
dnl Declara limbajul in care este realizat proiectul  
AC_LANG_CPLUSPLUS  
dnl Prefixul implicit de instalare  
AC_PREFIX_DEFAULT(/usr)  
  
dnl Versiunea de program  
VERSION="1.0"  
  
dnl Initializari automake  
AM_INIT_AUTOMAKE(SAL, $VERSION)  
dnl Fisierul antet de configurare  
AM_CONFIG_HEADER(src/config.h)  
  
dnl Determina numele standardizat al masinii
```

```

AC_CANONICAL_HOST

dnl Verifica existenta programelor
dnl Verifica existenta compilatorului de C
AC_PROG_CC
dnl Verifica existenta compilatorului de C++
AC_PROG_CXX
dnl Verifica existenta programului make
AC_PROG_MAKE_SET

dnl Verifica existenta fisierelor-antet
dnl Verifica existenta fisierelor standard C
AC_HEADER_STDC
dnl Verifica existenta fisierelor-antet getopt.h
dnl si sys/time.h
AC_CHECK_HEADERS(getopt.h sys/time.h)

dnl Verifica existenta bibliotecilor
dnl Verifica existenta bibliotecii crypt
AC_CHECK_LIB(crypt, main, [
    AC_DEFINE(HAVE_CRYPT)
    LIBS="-lcrypt"])

dnl Verifica existenta bibliotecii pq
AC_CHECK_LIB(pq, main, [
    AC_DEFINE(HAVE_LIBPQ)
    LIBS="$LIBS -lpq"],
    AC_MSG_ERROR(This version of SAL requires libpq library))
dnl Stabileste bibliotecile care trebuie link-editate
dnl impreuna cu executabilele generate
LIBS="$LIBS -lXterminal -lncurses -lgpm -lpenguins"
dnl Stabileste optiunile pentru preprocesor
CPPFLAGS="-I/usr/include/Xterminal -I/usr/include/ncurses
-I/usr/include/pgsql -I/usr/include/Penguins"

dnl Verifica typedef-uri, structuri si caracteristici
dnl compilatorului
AC_HEADER_TIME

dnl Verifica existenta anumitor functii
AC_CHECK_FUNCS(getopt_long gettimeofday)

dnl Stabileste optiunile pentru compilatorul C++
CXXFLAGS="-O2 -w"
if test "$CC" = "gcc"; then
    CXXFLAGS="$CXXFLAGS -funsigned-char -falt-external-templates"
fi

dnl Fisierele care vor fi generate
AC_OUTPUT(Makefile src/Makefile)

```

install/Makefile.am

```

bin_PROGRAMS = config
config_SOURCES = main.cc

```

src/Makefile.am

```

bin_PROGRAMS = sal
sal_SOURCES = bonuri.cc cas.cc conflunian.cc fisafisc1.cc \
              fisafisc2.cc fluturi.cc general.cc init.cc initluna.cc \
              main.cc personal.cc pontaj.cc salarii.cc salcalc.cc \
              stat.cc useradaug.cc usersterg.cc

clean-local:
    rm -f *.lst

distclean-local:
    rm -f *.lst

maintainer-clean-local:
    rm -f *.lst

```

src/config.h.in

```

/* src/config.h. Generated automatically by configure. */
/* src/config.h.in. Generated automatically from configure.in by
autoheader. */

/* Define if you have the ANSI C header files. */
#define STDC_HEADERS 1

/* Define if you can safely include both <sys/time.h> and
<time.h>. */
#define TIME_WITH_SYS_TIME 1

/* Define if you have the `crypt' function. */
#define HAVE_CRYPT 1

/* Define if you are using `PostgreSQL' as SQL server. */
#define HAVE_LIBPQ 1

/* Define if you have the getopt_long function. */
/* #undef HAVE_GETOPT_LONG */

/* Define if you have the gettimeofday function. */
/* #undef HAVE_GETTIMEOFDAY */

```

```
/* Define if you have the <getopt.h> header file. */
#define HAVE_GETOPT_H 1

/* Define if you have the <sys/time.h> header file. */
#define HAVE_SYS_TIME_H 1

/* Name of package */
#define PACKAGE "SAl"

/* Version number of package */
#define VERSION "1.0"
```

Al doilea exemplu este biblioteca Xterminal, o interfață utilizator în mod terminal, realizată în limbajul C++. Redăm mai jos structura de directoare a arborelui surselor, în care fișierele care ne intereseză au fost îngroșate (unele directoare au fost omise pentru acuratețea descrierii):

```
:
acconfig.h configure doc install-sh Makefile.am NEWS
aclocal.m4 configure.in example lista Makefile.in README
config.guess COPYING.LIB include ltconfig missing src
config.sub CREDITS INSTALL ltmain.sh mkinstalldirs

./doc:
ChangeLog manual objects.tree TO-DO

./doc/manual:
[...]

./example:
exapp.cc exdlg.cc exfreeapp.cc exlist.cc extab.cc Makefile.am
exbar.cc exeevent.cc exhlist.cc exmyapp.cc exwin.cc Makefile.in

./include:
Makefile.am xevent.h xtdesktop.h xtlist.hxttexteditor.h
Makefile.in xobject.h xdialog.h xtmenu.h xttools.h
RegionalInfo.hxtapp.h xterminal.h xtother.h xtwindow.h
stamp-h.in xtbar.h xtfield.h xtpassive.h
xbclass.h xtbutton.h xthlist.h xtscreen.h
xconfig.h.in xtcombobox.h xtlib.h xttab.h

./src:
Makefile.am xobject.cc xtdesktop.cc xtlist.cc xttab.cc
Makefile.in xtapp.cc xtdialog.cc xtmenu.cc xttexteditor.cc
RegionalInfo.cc xtbar.cc xtfield.cc xtother.cc xttools.cc
xbclass.cc xtbutton.cc xthlist.cc xtpassive.cc xtwindow.cc
xevent.cc xtcombobox.cc xtlib.cc xtscreen.cc
```

Makefile.am

```
AUTOMAKE_OPTIONS = foreign
SUBDIRS = include src
clean-local:
    cd example; $(MAKE) clean
distclean-local:
    cd example; $(MAKE) distclean
examples:
    cd example; $(MAKE)
```

acconfig.h

```
/* The package name. */
#undef PACKAGE

/* The package version. */
#undef VERSION

/* Define if you want to catch the signals and translate them into
events. */
#undef SIGNAL_CATCHING

/* Define if you don't want to use that ugly ncurses hacks. */
#undef NO_NCURSES_HACKS

/* Define if you want to generate debugging informations. */
#undef DEBUG

/* Define if you are using 'ncurses' as curses library. */
#undef HAVE_NCURSES

/* Define if you are using 'slang' as curses library. */
#undef HAVE_SLANG

/* Define if you want to use the Linux console mouse support. */
#undef HAVE_LIBGPM

/* Define if your curses have 'ESCDELAY'. */
#undef HAVE_ESCDELAY

/* Define if the use of 'TIOCGWINSZ' requires '<sys/ioctl.h>' */
#undef GWINSZ_IN_SYS_IOCTL
```

configure.in

```
dnl configure.in for Xterminal library
dnl Copyright (c) 1997-2001 Dragos Acostachioiaie
```

```

dnl Process this file with autoconf to produce a configure script.
dnl Declara fisierul principal al proiectului
AC_INIT(src/xobject.cc)
dnl Declara limbajul in care este realizat proiectul
AC_LANG_CPLUSPLUS
dnl Prefixul implicit de instalare
AC_PREFIX_DEFAULT(/usr)

dnl Stabileste versiunea bibliotecii
VERSION="1.0.6"
CURRENT=16
REVISION=0
AGE=10
AC_SUBST(CURRENT)
AC_SUBST(REVISION)
AC_SUBST(AGE)

dnl Initializari automake
AM_INIT_AUTOMAKE(Xterminal, $VERSION)
dnl Declara fisierul-antet de configurare
AM_CONFIG_HEADER(include/xconfig.h)

dnl Determina numele standardizat al masinii
AC_CANONICAL_HOST

dnl Verifica existenta compilatorului de C
AC_PROG_CC
dnl Verifica existenta compilatorului de C++
AC_PROG_CXX

dnl Initializeaza libtool
dnl Implicit vor fi construite biblioteci dinamice
AM_ENABLE_SHARED
AM_DISABLE_STATIC
dnl Verifica existenta libtool
AM_PROG_LIBTOOL

dnl Verifica existenta fisierelor antet
AC_HEADER_STDC
AC_CHECK_HEADERS(sys/ioctl.h string.h strings.h unistd.h)
AM_HEADER_TIOCGWINSZ_NEEDS_SYS_IOCTL

dnl Verifica typedef-uri, structuri si caracteristici ale
dnl compilatorului
AC_STRUCT_TM

dnl Verifica existenta functiilor
AC_TYPE_SIGNAL
AC_FUNC_VPRINTF
AC_CHECK_FUNCS(strdup strtod)

dnl Stabileste optiunile pentru compilator
CXXFLAGS="-O2 -w"

```

```

dnl Introduce optiunea --disable-signals,
dnl care va dezactiva conversia semnalelor in evenimente
dnl in cadrul bibliotecii
AC_ARG_ENABLE(signals,
  [ --disable-signals Turn off signal translation into events],
  AC_DEFINE(SIGNAL_CATCHING)
)

dnl Introduce optiunea --disable-ncurses-hacks,
dnl care va dezactiva utilizarea structurilor interne
dnl ale bibliotecii ncurses
AC_ARG_ENABLE(ncurses-hacks,
  [ --disable-ncurses-hacks Don't use ncurses internal structures],
  AC_DEFINE(NO_NCURSES HACKS)
)

dnl Verifica daca se genereaza informatii
dnl de depanare
AC_ARG_ENABLE(debug,
  [ --enable-debug Turn on debugging],
  AC_DEFINE(DEBUG)
  CXXFLAGS="$CXXFLAGS -g"
)

dnl Verifica daca se utilizeaza slang
dnl ca biblioteca curses
AC_ARG_WITH(slang,
  [ --with-slang Use slang library instead of ncurses],
  withslang=x$withval,
  withslang=xno
)
if test $withslang = xyes; then
  AC_CHECK_LIB(slang, main, [
    AC_DEFINE(HAVE_SLANG)
    LIBS="-lslang"
    CPPFLAGS="$CPPFLAGS -I/usr/include/slang"],
    AC_MSG_WARN(Slang library not found, trying ncurses)
  )
fi

dnl Verifica existenta bibliotecii ncurses
if test "$ac_cv_lib_slang_main" != "yes"; then
  AC_CHECK_LIB(ncurses, main, [
    AC_DEFINE(HAVE_NCURSES)
    LIBS="-lncurses"
    CPPFLAGS="$CPPFLAGS -I/usr/include/ncurses"],
    AC_MSG_ERROR(No curses library found)
  )
fi

dnl Verifica daca se ofera suport pentru mouse
AC_ARG_WITH(gpm,
  [ --without-gpm Disable GPM mouse support],
  withgpm=x$withval,
)

```

```

withgpm=xyes
}
if test $withgpm = xyes; then
  AC_CHECK_LIB(gpm, main, [
    AC_DEFINE(HAVE_LIBGPM)
    LIBS="$LIBS -lgpm"])
fi

dnl Verifica existenta functiei resizeterm
AC_CHECK_FUNCS(resizeterm)

dnl Verifica existenta variabilei globale ESCDELAY
dnl in cadrul bibliotecii curses
AC_MSG_CHECKING(for ESCDELAY support)
if test "$ac_cv_lib_ncurses_main" = "yes"; then
  AC_CACHE_VAL(ac_cv_have_escdelay,
  AC_TRY_LINK([
#include <curses.h>], [
{
  ESCDELAY = 0;
}],
  ac_cv_have_escdelay=yes,
  ac_cv_have_escdelay=no)
)
fi
if test "$ac_cv_lib_slang_main" = "yes"; then
  ac_cv_have_escdelay=yes
fi
AC_MSG_RESULT($ac_cv_have_escdelay)
if test "$ac_cv_have_escdelay" = "yes"; then
  AC_DEFINE(HAVE_ESCDELAY)
fi

dnl Optiunile pentru compilatorul de C
if test "$CC" = "gcc"; then
  CXXFLAGS="$CXXFLAGS -funsigned-char -fthis-is-variable
            -falt-external-templates"
fi

includedir="$includedir/Xterminal"

dnl Fisierele care vor fi generate
AC_OUTPUT(Makefile src/Makefile include/Makefile example/Makefile)

```

example/Makefile.am

```

LDFLAGS = -lXterminal
bin_PROGRAMS = exapp exbar exdlg exeevent exfreeapp exhlist exlist \
              exmyapp extab exwin
exapp_SOURCES = exapp.cc
exbar_SOURCES = exbar.cc
exdlg_SOURCES = exdlg.cc

```

```

exeevent_SOURCES = exeevent.cc
exfreeapp_SOURCES = exfreeapp.cc
exhlist_SOURCES = exhlist.cc
exlist_SOURCES = exlist.cc
exmyapp_SOURCES = exmyapp.cc
extab_SOURCES = extab.cc
exwin_SOURCES = exwin.cc

```

include/Makefile.am

```

include_HEADERS = xbclass.h xconfig.h xevent.h xobject.h xtapp.h \
                  xtblbar.h xtbutton.h xtcombobox.h xtdesktop.h \
                  xtdialog.h xterminal.h xtfield.h xthlist.h xtlib.h \
                  xtlist.h xtmenu.h xtother.h xtpassive.h xtscreen.h \
                  xtab.h xttexteditor.h xttools.h xtwindow.h \
                  RegionalInfo.h

uninstall-local:
  rmdir $(includedir)

```

xconfig.h.in

```

/* include/xconfig.h.in. Generated automatically from
   configure.in by autoheader. */

/* Define if you don't have vprintf but do have _doprnt. */
#undef HAVE_DOPRNT

/* Define if you have the vprintf function. */
#undef HAVE_VPRINTF

/* Define as the return type of signal handlers (int or void). */
#undef RETSIGTYPE

/* Define if you have the ANSI C header files. */
#undef STDC_HEADERS

/* Define if your declares struct tm. */
#undef TM_IN_SYS_TIME

/* Define if you want to catch the signals and translate them into
   events. */
#undef SIGNAL_CATCHING

/* Define if you don't want to use that ugly ncurses hacks. */
#undef NO_NCURSES_HACKS

/* Define if you want to generate debugging informations. */
#undef DEBUG

```

```

/* Define if you are using `ncurses' as curses library. */
#undef HAVE_NCURSES

/* Define if you are using `slang' as curses library. */
#undef HAVE_SLANG

/* Define if you want to use the Linux console mouse support. */
#undef HAVE_LIBGPM

/* Define if your curses have `ESCDELAY'. */
#undef HAVE_ESCDELAY

/* Define if you have the resizeterm function. */
#undef HAVE_RESIZETERM

/* Define if you have the strdup function. */
#undef HAVE_STRDUP

/* Define if you have the strtod function. */
#undef HAVE_STRTOD

/* Define if you have the <string.h> header file. */
#undef HAVE_STRING_H

/* Define if you have the <strings.h> header file. */
#undef HAVE_STRINGS_H

/* Define if you have the <sys/ioctl.h> header file. */
#undef HAVE_SYS_IOCTL_H

/* Define if you have the <unistd.h> header file. */
#undef HAVE_UNISTD_H

/* Name of package */
#undef PACKAGE

/* Version number of package */
#undef VERSION

/* Define if TIOCGWINSZ requires sys/ioctl.h */
#undef GWINSZ_IN_SYS_IOCTL

```

src/Makefile.am

```

lib_LTLIBRARIES = libXterminal.la
libXterminal_la_SOURCES = xbclass.cc xevent.cc xobject.cc xtapp.cc \
    xtbar.cc xtbutton.cc xtcombobox.cc \
    xtdesktop.cc xtdialog.cc xtfield.cc \
    xthlist.cc xtlib.cc xtlist.cc xtmenu.cc \
    xtother.cc xtpassive.cc xtscreen.cc xttab.cc \
    xttexteditor.cc xttools.cc xtwindow.cc \
    RegionalInfo.cc

```

```

libXterminal_la_LDFLAGS = -version-info @CURRENT@:@REVISION@:@AGE@

```

Mai multe informații despre autoconf, automake și libtool pot fi obținute din manualul info pentru aceste utilitare.

3.3.1. Asigurarea portabilității programelor

Portabilitatea este o calitate a codului-sursă care îi permite să fie compilat și executat pe o varietate de platforme. În contextul cărții de față, portabilitatea se referă în general la calitatea de a rula pe diferite sisteme compatibile UNIX, uneori și Windows.

Prezentăm în cele ce urmează o serie de idei care trebuie urmate în vederea asigurării unui nivel de portabilitate cât mai ridicat.

1. Trebuie utilizate funcții de bibliotecă conforme specificațiilor ANSI (vezi Anexa 1) și trebuie evitate funcțiile specifice unei anumite platforme.
2. Dacă trebuie folosite funcții specifice unei anumite platforme sau biblioteci, existența acestora trebuie testată în cadrul scriptului *configure*. Codul va trebui să includă fișierul-antet *config.h* generat și să efectueze teste prin intermediul directivelor preprocesor ca în exemplul de mai jos:

```

#ifdef HAVE_NCURSES
strcpy(s, termname());
#else
strcpy(s, getenv("TERM"));
#endif // HAVE_NCURSES

```

3. De asemenea, anumite fișiere-antet au denumiri diferite pe diferite platforme:

```

#ifdef HAVE_STRING_H
#include <string.h>
#else
#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif
#endif

```

4. Este indicat includerea în toate fișierele-antet a câtorva directive preprocesor pentru a preveni includerea multiplă:

```

#ifndef FUNCTII_H
#define FUNCTII_H 1
...
#endif /* FUNCTII_H */

```

5. Pentru ca un program C++ să poată utiliza o bibliotecă compilată cu un compilator C, este necesar ca orice simbol exportat să fie declarat `extern "C"`:

```
#ifdef __cplusplus
extern "C" {
#endif
...
#ifndef __cplusplus
}
#endif
```

3.4. Utilitarele *diff*, *patch* și *diffutils*

3.4.1. Utilitarul *diff*

Utilitarul *diff* determină diferențele dintre două fișiere text. Acest program este util pentru a evidenția modificările aduse unui program-sursă cu ocazia unei versiuni mai noi, sau cele efectuate de diferiți programatori. Programul *diff* a fost scris de către Mike Haertel, David Hayes, Richard Stallman, Len Tower, Paul Eggert și Wayne Davison, prima versiune datând din anul 1992.

Acest program se găsește în pachetul *diffutils*.

diff trimite la ieșirea standard rezultatul comparației, care poate fi codificat utilizând mai multe formate (vezi *infra*). Setul de diferențe generat este denumit de obicei *diff* sau *patch*. Dacă fișierele sunt identice, nu este trimis nici un rezultat la ieșire.

Setul de diferențe generat poate fi utilizat pentru a distribui modificări (sau *update-uri*) ale programelor-sursă către alte persoane. Această metodă este utilă atunci când diferențele sunt mici în comparație cu fișierele-sursă. De asemenea, *diff* poate compara fișierele aflate pe două structuri de directoare (*source trees*). Pentru a „*aplica*” setul de diferențe unui fișier „*origine*” în vederea generării fișierului modificat, se utilizează utilitarul *patch*.

Spre exemplu, comanda *diff fisier1_original fisier_modificat* generează setul de diferențe dintre cele două fișiere specificate în linia de comandă.

După cum afirmam mai sus, pot fi utilizate mai multe formate pentru ieșirea generată. Varianta implicită, numită „*normală*”, include doar liniile modificate, șterse sau adăugate. Pentru exemplificare, să considerăm următoarele două fișiere-sursă:

hello.c:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

hello_ro.c:

```
#include <stdio.h>

int main()
{
    printf("Salut!\n");
    return 0;
}
```

Iată un exemplu de *patch* generat în urma execuției comenzi *diff hello.c hello_ro.c*.

```
5c5
<printf("Hello, world!\n");
---
>printf("Salut!\n");
```

În mod uzual, atunci când studiem diferențele dintre două fișiere, dorim să vedem liniile de text apropiate de cele care au fost modificate, pentru a înțelege exact ce a fost schimbat. Aceste liniile de text apropiate sunt denumite „*context*”. Programul *diff* oferă două formate pentru setul de diferențe care includ contextul modificărilor: formatul „*context*” și formatul *unified*. Folosirea acestor două formate mai are avantajul că programul *patch* poate aplica modificările efectuate chiar dacă s-au efectuat modificări ale fișierului original (e.g. s-au adăugat liniile de text).

Generarea setului de modificări în format *context*: *diff -c fisier_origine fisier_modificat*. Prezentăm mai jos un exemplu:

```
*** hello.c Tue Jul 23 12:56:49 2002
--- hello_ro.c Tue Jul 23 12:57:09 2002
*****
*** 2,8 ****
int main()
{
    printf("Hello, world!\n");
    return 0;
```

```

}
--- 2,8 ----

int main()
{
    printf("Salut!\n");

    return 0;
}

```

Caracterul de început al liniilor de text modificate poate fi:

- „!”: o linie de text care face parte dintr-un grup de una sau mai multe linii care au fost modificate;
- „+”: o linie de text introdusă;
- „-”: o linie de text ştersă.

Formatul *diff* unificat este mai compact decât cel context. Generarea se face cu: *diff -u fisier_origine fisier_modificat*. Iată un exemplu:

```

--- hello.c Tue Jul 23 12:56:49 2002
+++ hello_ro.c Tue Jul 23 12:57:09 2002
@@ -2,7 +2,7 @@
int main()
{
-    printf("Hello, world!\n");
+    printf("Salut!\n");

    return 0;
}

```

Caracterul de început al liniilor de text modificate poate fi:

- „+”: o linie de text introdusă;
- „-”: o linie de text ştersă.

Pentru ca diferențele listate de comanda *diff* să fie memorate într-un fișier în loc să fie afișate pe ecran, poate fi utilizată comanda de redirectare a ieșirii – „>” –, specificându-se numele noului fișier (de tip *patch*) care va fi creat:

```
diff [ opțiuni ] fisier_origine fisier_modificat > fisier_patch
```

Alți parametri utili ai comenzi *diff* sunt următorii:

- q raportează doar faptul că un fișier a fost modificat, nu și detaliile modificării;
- r parcurge recursiv directoarele, atunci când se face comparația între două directoare.

Mai multe informații despre utilitarul *diff* pot fi obținute apelând comanda *info diff*.

3.4.2. Utilitarul *patch*

Pentru a aplica un fișier *diff* unui fișier sau arbore de fișiere original, se utilizează utilitarul *patch*. Acesta a fost implementat pentru prima oară în anul 1992 de către Larry Wall și Paul Eggert.

Acest program se găsește în pachetul *diffutils*.

Numele fișierelor care vor fi modificate sunt în mod usual preluate din fișierul *diff*. De asemenea, formatul utilizat pentru *patch* este determinat în mod automat. Sintaxa simplificată este *patch < fisier_patch*.

Parametrii mai importanți sunt:

- b creează fișiere de siguranță *backup*;
- c consideră că fișierul *patch* este în format context;
- n consideră că fișierul de intrare este în format normal;
- pnumar elimină căile de directoare conținând numărul de niveluri (caractere „/”) specificat din numele fișierelor. Acest parametru este util atunci când fișierele sunt stocate într-o ierarhie de directoare la persoana care a generat *patch*-ul și în alt mod la persoana care îl utilizează. Spre exemplu, dacă numele fișierelor din *patch* este /home/dragos/work/doc++/src/doc2xml.11, utilizând parametrul –p3, acesta va fi transformat în doc++/src/doc2xml.11;
- R consideră că *patch*-ul a fost creat cu fișierele original respectiv modificat schimbate între ele;
- u consideră formatul fișierului de intrare ca fiind *diff* unificat.

Mai multe informații despre utilitarul *patch* pot fi obținute apelând comanda *info patch*.

3.4.3. Utilitarul *diffstat*

Utilitarul *diffstat* realizează o analiză statistică a unui fișier *patch*, și anume a numărului de inserări, ștergeri și modificări ale tuturor fișierelor incluse în *patch*. Spre exemplu, comanda *diffstat patch-3.4.10-pre1.gz* va afișa o statistică a fișierului *patch* specificat (*diffstat* va decomprima automat fișierul):

CREDITS		2
README		15
REPORTING-BUGS		6
doc/ChangeLog		13

```

po/fr.po      |  4
po/ja.po      |  4
po/ro.po      |  6
po/ru.po      |  4
src/cpp.11    | 10
src/doc2db.11 | 1594 -----
src/doc2dbsgml.11 | 1594 ++++++=====
src/doc2dbxml.11 | 1656 ++++++=====
src/equate.cc |  4
src/main.cc   | 32
14 files changed, 3250 insertions(+), 1594 deletions(-)

```

3.5. Sistemul de control al versiunilor CVS

CVS (*Concurrent Versioning System*) este un sistem de control al versiunii, care permite păstrarea tuturor versiunilor unor fișiere (în general cod-sursă) și realizarea unui jurnal cu modificările efectuate de diferiți utilizatori. Acest sistem permite ca un grup de persoane (echipă) să lucreze la un grup de fișiere (proiect) comun. Sistemul CVS a fost creat în 1989 de către Brian Berliner și Jeff Polk, pe baza unor scripturi shell realizate în 1986 de către Dick Grune.

Pentru a putea utiliza CVS trebuie instalat pachetul `cvs`.

Sistemul consistă dintr-un *depozit (repository)*, stocat pe un server, care conține cea mai recentă versiune a grupului de fișiere. Utilizatorii se pot conecta în orice moment la acest depozit și pot prelua fișierele pe care le conține, sau își pot actualiza o versiune mai veche, existentă pe discul local. Fiecare utilizator efectuează modificări asupra fișierelor și, în momentul în care dorește, trimite modificările către depozit.

O sesiune CVS începe aşadar cu obținerea unei copii a arborelui conținând fișierele-sursă ale proiectului `doc++`:

```
cvs checkout doc++
```

Această comandă va crea un director `doc++` în directorul curent care va conține programele-sursă ale proiectului menționat mai sus. În directorul nou creat precum și în subdirectoarele componente există câte un director numit CVS. În mod normal, conținutul acestui director nu trebuie modificat.

Pot fi modificate fișiere-sursă prin intermediul unui editor, după care aceste modificări pot fi trimise (*committed*) către depozitul CVS utilizând comanda:

```
cvs commit
```

Sistemul CVS va lansa un editor care va solicita introducerea unui mesaj ce va fi introdus în jurnalul de operațiuni. În mod ușual mesajul trebuie să conțină o descriere a modificărilor efectuate. Programul care va fi apelat ca editor se stabilește prin intermediul variabilei de mediu `CVSEDITOR`. Pentru a se evita execuția editorului poate fi utilizată comanda (care va actualiza doar un singur fișier care a fost modificat):

```
cvs commit -m "Optimizat algoritmul de sortare" src/cpp.11
```

Pentru a actualiza copia locală a fișierelor-sursă, se folosește comanda:

```
cvs update
```

Pentru a adăuga un fișier nou creat și în depozitul CVS, se utilizează comanda:

```
cvs add src/doc2dbxml.11
```

Pentru a șterge un fișier eliminat din arborele local și din depozitul CVS, se utilizează comanda:

```
cvs remove src/doc2db.11
```

Pentru a vizualiza diferențele dintre un fișier local (original) și un fișier aflat în CVS (modificat):

```
cvs diff doc/ChangeLog
```

Ieșirea generată este în format *diff unified* (vezi *supra*).

Alte opțiuni utile ale comenzi `cvs`:

- `-d director` specifică locația depozitului. `director` poate avea și formatul `utilizator@mașină:director`. Parametrul opțiunii `-d` poate fi specificat și prin intermediul variabilei de mediu `CVSROOT`. Spre exemplu, un director local va fi specificat ca `/usr/local/cvsroot`, iar unul aflat pe o altă mașină `adragos@cvs.docpp.sourceforge.net:/cvsroot/docpp`.

Informații detaliate privind utilizarea serviciului CVS pot fi obținute apelând comanda `info cvs`.

Pentru a utiliza sistemul CVS poate fi folosit și programul Cervisia (meniu Development :: Cervisia (CVS Frontend)):

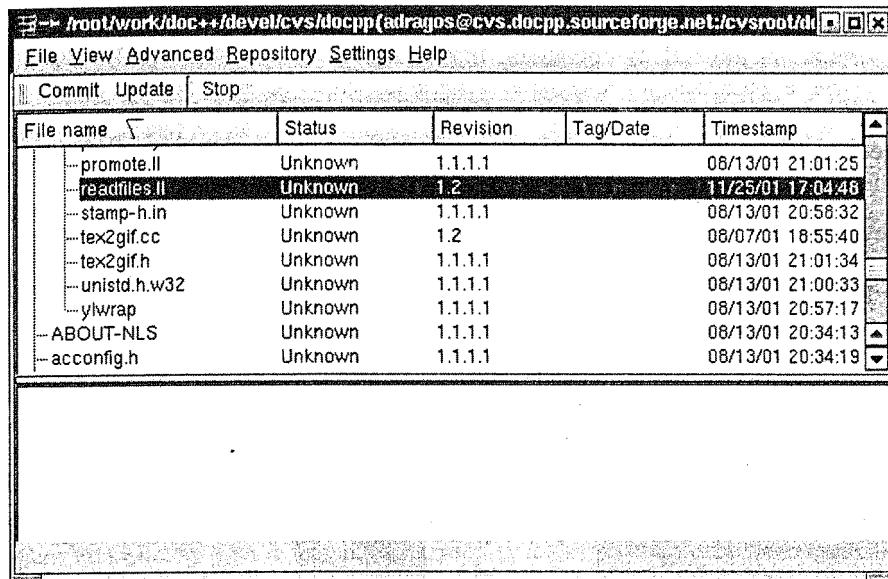


Figura 3.1. Programul Cervisia

3.5.1. Configurarea unui server CVS

Această secțiune prezintă modul de configurare a unui calculator care va stoca un depozit CVS și va servi cereri CVS.

Pentru a configura CVS ca server, trebuie modificat în primul rând fișierul /etc/inetd.conf, pentru ca inetd să execute comanda cvs pserver atunci când recepționează o cerere de conexiune pe portul 2401 (portul standard CVS):

```
2401 stream tcp nowait root /usr/bin/cvs
cvs -f --allow-root=/var/lib/cvsroot/docpp pserver
```

Oțiunea --allow-root specifică directorului CVSROOT permis. Acest director se referă la directorul în care se află stocat un anumit depozit, în exemplul nostru /var/lib/cvsroot/docpp. Directorul se stabilește de către administratorul sistemului. Dacă există mai multe depozite la care se dorește permiterea accesului, oțiunea trebuie repetată.

Dacă se dorește referirea portului ca un nume simbolic de serviciu, se poate adăuga în fișierul /etc/services liniile următoare:

```
cvspserver 2401/tcp
cvspserver 2401/udp
```

După modificarea fișierului /etc/inetd.conf, daemonul inetd trebuie repornit.

Sistemele mai noi oferă, ca alternativă la inetd, daemonul xinetd. Pentru configurarea serverului CVS este necesară crearea unui fișier denumit cvspserver în directorul /etc/xinetd.d, cu următorul conținut:

```
service cvspserver
{
    disable          = no
    socket_type     = stream
    wait             = no
    protocol        = tcp
    port             = 2401
    user             = root
    server           = /usr/bin/cvs
    server_args      = -f --allow-root=/var/lib/cvsroot/docpp pserver
    log_on_success   += USERID
    log_on_failure   += USERID
}
```

Pentru mai multe detalii privind configurarea serviciilor inetd recomandăm consultarea lucrării Dragoș Acostăchioie, *Administrarea și configurarea sistemelor Linux*, Editura Polirom, Iași, 2002.

Pentru autentificarea utilizatorilor se folosesc în general un fișier separat de parole, astfel încât utilizatorii nu vor folosi parola lor obișnuită de intrare în sistem. Fișierul de parole CVS este \$CVSROOT/CVSROOT/passwd, care folosește un sistem similar cu /etc/passwd, exceptând faptul că include mai puține câmpuri: numele de utilizator CVS, parola (optional) și un nume de utilizator sub care va rula CVS (optional). Parolele sunt criptate utilizând mecanismul standard crypt. Iată un fișier exemplu:

```
anonymous:
dragos:weknEDYU47q
mihaela:37cLKDUEE:pubcvs
```

Prima linie din fișier permite accesul oricărui client CVS care încearcă să se autentifice cu utilizatorul anonymous, fără a solicita vreo parolă (acest gen de utilizator este folosit de obicei pentru a oferi acces doar pentru citire – read-only – la depozitul CVS).

A treia linie va acorda drept de acces utilizatorului mihaela, dar operațiile CVS vor fi executate pe server sub utilizatorul pubcvs. Nu este necesar ca utilizatorul mihaela să existe în sistem, dar utilizatorul cu numele pubcvs trebuie să fie definit. Aceasta permite definirea de mai mulți utilizatori CVS fără a fi necesar definirea acestora ca utilizatori ai sistemului.

CVS poate folosi și autentificarea oferită de sistemul de operare. Astfel, dacă utilizatorul specificat în \$CVSROOT/CVSROOT/passwd nu este găsit, sau dacă acest fișier de parole nu există, serverul va presupune că utilizatorul există în sistem și va încerca autentificarea acestuia.

În mod evident, pentru a asigura securitatea depozitului CVS, trebuie corect configurate drepturile de acces ale fișierelor și subdirecțoarelor din cadrul directorului \$CVSROOT/CVSROOT.

Pentru a putea stabili care utilizatori au permisiune de scriere în depozitul CVS, sau care au doar permisiune de citire, sunt utilizate două fișiere. Primul fișier, \$CVSROOT/CVSROOT/readers, declară utilizatorii care au dreptul de citire, câte unul pe fiecare linie, ca în exemplul de mai jos:

```
anonymous
dragos
mihaela
```

Într-un mod similar, fișierul \$CVSROOT/CVSROOT/writers definește utilizatorii care au drept de scriere:

```
dragos
mihaela
```

3.6. Alte programe utile

3.6.1. Programul *indent*

Programul *indent* se utilizează pentru a formați programul-sursă C/C++ după un anumit stil. Utilizarea unui stil unitar conduce la o lizibilitate mai mare a codului și, ca urmare, la un cost de mențenanță mai redus.

Sintaxa generală este:

```
indent [ opțiuni ] fișiere_sursa
```

Dacă nu este specificată nici o opțiune, fișierele-sursă vor fi formatați conform stilului GNU (vezi documentul *The GNU Coding Standards*: <http://www.gnu.org/prep/standards.html>). Pentru a formați codul-sursă conform recomandărilor Kernighan & Ritchie (vezi *The C Programming Language*), se poate utiliza opțiunea -kr.

Opțiunile mai importante (grupate după categoria acestora) sunt:

Linii vide

- -bad introduce o linie vidă după fiecare bloc de declarații;
- -bap introduce o linie vidă după fiecare funcție;
- -bbb introduce o linie vidă înainte de fiecare bloc de comentarii.

Comentarii

indent formează atât comentariile C /* ... */, cât și cele C++ // ...). Comentariile de tip bloc (sau cutie) nu sunt modificate:

```
*****  
* comentariu *  
*****
```

sau:

```
/*  
* comentariu  
*/
```

Marginea din dreapta (practic numărul de caractere pe o linie) pentru comentarii este implicit 78, această valoare putând fi modificată prin intermediu opțiunii -lc. Aceasta poate fi utilă spre exemplu pentru a putea tipări codul-sursă.

Dacă linia pe care se află comentariul nu conține cod-sursă, acesta va fi aliniat la coloana pe care este aliniat și codul-sursă din blocul curent. Acest stil de alinieră poate fi modificat cu ajutorul opțiunii -d, care specifică numărul de caractere cu care comentariul va fi mutat la stânga.

Dacă linia pe care se află comentariul conține cod-sursă, comentariile vor fi aliniate implicit la coloana 33. Această valoare poate fi modificată cu ajutorul următoarelor trei opțiuni:

- -c specifică coloana de alinieră pentru comentariile care urmează după cod;
- -cd specifică coloana de alinieră pentru comentariile care urmează după declarații;
- -cp stabilăște coloana de alinieră pentru comentariile care urmează după directive ale preprocesorului.

Opțiunea -ccb mută delimitatorii comentariilor pe linii separate. Astfel, un comentariu de genul /* comentariu */ va fi transformat în:

```
/*  
* comentariu  
*/
```

Opțiunea `-sc` va introduce un caracter „*” la începutul fiecărei linii dintr-un comentariu. Astfel, prin utilizarea opțiunilor combinate `-cdb -sc`, vom obține:

```
/*
 * comentariu
 */
```

Instrucțiuni

Opțiunea `-br` formatează acoladele astfel:

```
if(i == 0) {
    i++;
}
```

Opțiunea `-bl` formatează acoladele astfel:

```
if(i == 0)
{
    i++;
}
```

Numărul de spații cu care acoladele sunt indentate poate fi modificat cu opțiunea `-bli`. În exemplul de mai sus s-a folosit opțiunea `-bli4`.

Indentarea etichetelor `case` într-o construcție `switch` se poate face cu ajutorul opțiunii `-cliN`. Indentarea acoladelor de sub etichetele `case` se modifică cu opțiunea `-cbin`.

În urma folosirii opțiunii `-pcs` va fi introdus un spațiu după numele funcțiilor și caracterul „(“ la apelarea acestora, cum ar fi `printf ("text\n");`. Opțiunea `-cs` introduce un spațiu după un operator cast, iar `-bs` între cuvântul `sizeof` și argumentul său.

Opțiunea `-saf` introduce un spațiu între `for` și paranteza care urmează. `-sai` inserează un spațiu între `if` și paranteza care urmează. `-saw` introduce un spațiu după instrucțiunea `while`.

Opțiunea `-prs` separă printr-un spațiu parantezele de conținutul dintre ele.

Declarații

Identifierii pot fi aliniați la coloana specificată cu opțiunea `-di`. Spre exemplu, opțiunea `-di8` va forma declarațiile ca mai jos:

```
int i;
char *s;
```

Argumentele unei funcții pot fi aranjate câte unul pe fiecare linie utilizând opțiunea `-bfda`, util pentru funcții cu număr mare de argumente. Astfel, funcția:

```
int main(int argc, char **argv)
```

va fi formatată ca:

```
int main(int argc,
         char **argv)
```

Pozitia acoladelor din cadrul declarațiilor `struct` poate fi modificată cu ajutorul opțiunii `-brs`:

```
struct student {
    char *nume;
    char *prenume;
};
```

Opțiunea `-bls` produce următorul format:

```
struct student
{
    char *nume;
    char *prenume;
};
```

Indentarea liniilor de cod

Atunci când în cadrul codului-sursă apar instrucțiuni de tipul `if` sau `for`, nivelul de indentare este incrementat cu valoarea specificată de opțiunea `-i`. Spre exemplu, `-i8` va indenta codul-sursă cu opt caractere. Atunci când o expresie ocupă mai multe linii, a doua linie și următoarele sunt indentate cu un număr suplimentar de spații, specificat cu ajutorul opțiunii `-ci` (implicit 0).

Despărțirea liniilor lungi

Lungimea maximă a unei linii de cod poate fi stabilită utilizând opțiunea `-l`. Aceasta nu include comentariile care urmează după codul efectiv.

Dezactivarea formatării automate

Programul `indent` permite dezactivarea formatării pe o porțiune din codul-sursă. Pentru a realiza acest lucru, trebuie plasat comentariul de control `/* *INDENT-OFF* */` (sau echivalentul C++ `// *INDENT-OFF*`), comentariu după care codul nu va fi formatat. Această dezactivare este activă până la sfârșitul fișierului-sursă sau până la întâlnirea comentariului `/* *INDENT-ON* */`.

Informații suplimentare despre indent pot fi obținute consultând pagina [info indent](#).

3.6.2. Alte utilitare

Alte utilitare folosite sunt cele din pachetele `binutils` și `fileutils`:

- `strip` – elimină toate simbolurile dintr-un fișier-obiect. Acestea includ simbolurile pentru depanare precum și altele care nu sunt utile pentru utilizatorul final al executabilului;
- `strings` – afișează toate sirurile de caractere conținute de un fișier-obiect;
- `touch` – modifică timpul de acces sau modificare – implicit timpul curent – a unor fișiere.

3.7. Documentarea programelor: *DOC++* și *code2html*

3.7.1. *DOC++*

DOC++ este un generator de documentații pentru C și C++, producând ieșire atât în format TeX, pentru a putea genera versiuni tipărite ale documentației, cât și HTML pentru navigare ușoară prin documentație. Documentația este extrasă direct din fișierele-antet sau sursă C/C++. Programul *DOC++* a fost conceput în anul 1995 de către Roland Wunderling și Malte Zöckler, dezvoltarea acestuia fiind continuată începând cu 1998 de către Dragoș Acostăchioae (autorul cărții de față).

Acest utilitar poate fi descărcat de pe Internet de la adresa <http://docpp.sourceforge.net/download.html>. Pachetul care conține programul se numește `doc++`.

Programul *DOC++* permite programatorilor să scrie documentație, putându-se concentra în același timp pe dezvoltarea programelor. Pentru a realiza acest deziderat, documentația este localizată chiar în cadrul codului-sursă al proiectului pe care îl dezvoltă. Urmând un astfel de model, un programator poate scrie documentație pentru clasele, funcțiile sale etc. și să o actualizeze în funcție de modificările aduse codului-sursă.

Pentru a putea face distincție între comentariile obișnuite, legate în general de implementare, *DOC++* utilizează niște tipuri speciale de comentarii, care au formatul:

```
/** ... */
/// ...
```

Vom numi un asemenea comentariu, *comentariu DOC++*. Fiecare dintre acestea se utilizează pentru a specifica documentația pentru declarația care urmează.

Fiecare comentariu *DOC++* definește o *intrare în manual*, care constă în documentația furnizată de comentariul în sine și diferite informații din declarația care urmează. Intrările în manual sunt structurate în mai multe câmpuri: unele dintre ele sunt completeate în mod automat de către *DOC++*, iar altele pot fi specificate de către programator:

Tabelul 3.1. Intrări în manuale *DOC++*

Numele câmpului	Furnizor	Descriere
<code>@type</code>	DOC++	tipul (returnat de) obiect
<code>@name</code>	DOC++, programator	numele obiectului
<code>@args</code>	DOC++	argumentele primite de obiect
<code>@memo</code>	programator	documentația pe scurt
<code>@doc</code>	programator	documentația completă
<code>@return</code>	programator	documentația pentru valoarea returnată de funcție
<code>@param</code>	programator	documentația pentru un argument al funcției
<code>@exception</code>	programator	documentația pentru o excepție interceptată
<code>@see</code>	programator	referință
<code>@author</code>	programator	autorul
<code>@version</code>	programator	versiunea

DOC++ permite gruparea intrărilor în manual, cu ajutorul construcției:

```
/** @name numele_grupului
   (documentatia pentru grup)
 */
//{@
   (alte intrari in manual)
//@}
```

Va fi creat un grup având numele specificat, care conține toate celelalte intrări ca sub-intrări.

Există posibilitatea incluzării altor fișiere conținând documentație, folosind sintaxa:

```
//@Include: lista_fisiere
/*@Include: lista_fisiere */
```

Pentru formatarea documentației pot fi utilizate atât macrouri (tag-uri) de formatare HTML, cât și LaTeX, ele fiind convertite în mod automat la tipul ieșirii generate.

```
/** O clasa de baza oarecare.
 Documentatia pentru aceasta clasa. A se vedea si situl
 proiectului de la adresa \URL{http://commonbase.sourceforge.net}
 */
class CommonBase {
private:
    /** acest membru va fi listat in documentatie doar daca
     utilizam optiunea '--private'
    */
    int privateMember();

protected:
    // o variabila oarecare
    double variable;

public:
    // o functie membra publica
    int publicMember();
};

/** O functie globala
 Aceasta este documentatia functiei

 @param c caracterul de intrare
 @return 0 daca totul este OK
 @return -1 in caz de eroare
 @author Dragos Acostachioaie
 @version 5.7.1
 @see CommonBase
 */
int function(const int c);
```

Pentru a genera documentația, vom apela programul `doc++`, astfel:

```
doc++ --dir html lista_fisiere
doc++ --tex --output.doc.tex lista_fisiere
```

Prima variantă generează documentația în format HTML, iar a doua în format LaTeX.

Alte opțiuni utile sunt:

- `-A` generează intrări de manual pentru toate declarațiile din codul-sursă, chiar dacă acestea nu sunt documentate;
- `-c` utilizează comentariile C/C++ ca fiind comentarii DOC++;
- `-p` include și membrii privați în documentație.

Pentru mai multe informații privind utilizarea `DOC++`, recomandăm consultarea documentației localizate în `/usr/share/doc/doc++-versiune/manual/html/`.

3.7.2. `code2html`

Utilitarul `code2html` convertește fișiere-sursă C și C++ în cod HTML, folosind mecanismul numit *syntax highlighting*, altfel spus aplică diferite scheme de colorare pentru instrucțiuni, paranteze etc. Programul `code2html` este scris în Perl și a fost dezvoltat începând cu anul 1999 de către Peter Palfrader.

Programul `code2html` poate fi descărcat de la adresa <http://code2html.sourceforge.net>.

Cel mai simplu exemplu de utilizare a acestui program este:

```
code2html atoi.c out.html
```

care generează fișierul HTML corespunzător programului C `atoi.c`.

Alte opțiuni des utilizate sunt următoarele:

- `-l` specifică limbajul în care este scris fișierul-sursă (util în cazul în care `code2html` nu poate determina automat tipul acestuia). Sunt recunoscute „c” și „c++” (precum și o serie de alte limbiage);
- `-n` inserează și numărul liniei din codul-sursă.

Alte programe pentru documentare automată:

- `doxygen`: <http://doxygen.sourceforge.net>;
- `cxx2html`: <http://aips2.aoc.nrao.edu/RELEASED/cxx2html>.

3.8. Verificarea programelor: `splint`

Programul `splint` este un instrument pentru verificarea programelor C. El poate descoperi erori de programare și posibile vulnerabilități cum ar fi pointeri utilizați înainte de a fi alocați, depășirea dimensiunii maxime a șirurilor de caractere, cicluri infinite etc. Programul nu verifică corectitudinea sintaxei C, ci erorile de programare. Aceste erori nu pot fi întotdeauna detectate de către programator, nici chiar în urma depanării programelor, deoarece programele nu pot fi testate în toate condițiile posibile. Utilitarul `splint` (*Secure Programming Lint*) este succesorul programului `LCLint`, și este dezvoltat de *Secure Programming Group*, condus de către David Evans, în cadrul Universității din Virginia.

Programul `splint` se găsește în pachetul cu același nume.

splint primește în mod ușor ca argument lista fișierelor C care vor fi verificate. Spre exemplu:

```
# splint test.c
Splint 3.0.1.6 --- 26 Feb 2002

test.c: (in function main)
test.c:5:12: Unallocated storage s passed as out parameter to
strcpy: s
An rvalue is used that may not be initialized to a value on some
execution path. (Use -usedef to inhibit warning)
test.c:6:2: Path with no return in function declared to return int
There is a path through a function declared to return a value on
which there is no return statement. This means the execution may
fall through without returning a meaningful result to the caller.
(Use -noret to inhibit warning)

Finished checking --- 2 code warnings
```

După cum se poate observa, splint a descoperit două erori:

1. Am folosit la apelul funcției strcpy argumentul s, un sir de caractere pentru care nu am alocat spațiu de memorie. Cu alte cuvinte, apelul strcpy va avea rezultate imprevizibile (în general va duce la terminarea forțată a programului).
2. Funcția main trebuie să returneze un int, dar noi nu am returnat explicit nici o valoare. Altfel spus, valoarea returnată în urma execuției programului este incertă.

Pentru mai multe informații privind splint, recomandăm manualul de utilizare, localizat în genere în /usr/share/doc/splint-versiune/manual.html.

Alt program care poate fi folosit cu succes (dar nu atât de evoluat ca splint) este clint.

Capitolul 4

MEDII INTEGRATE DE DEZVOLTARE

Mulți oameni, incluzând mulți programatori, profesori și manageri, refuză pur și simplu să înțeleagă complexitatea dezvoltării de programe.
(Bjarne Stroustrup)

4.1. Depanatorul vizual DDD

DDD (*Data Display Debugger*) este un așa-zis *debugger vizual*, cu alte cuvinte o interfață grafică intuitivă pentru programul de depanare GDB. DDD dispune atât de interfață în mod terminal, cât și în mediu grafic X Window. Autorii originali ai DDD sunt Dorothea Lütkehaus și Andreas Zeller, care au început dezvoltarea acestuia în anul 1995.

Acest program este localizat în pachetul ddd.

DDD se lansează în execuție prin comanda ddd. Ecranul DDD conține trei ferestre, și anume cea pentru inspectarea variabilelor (în partea de sus), cea pentru programul-sursă și cea pentru consola pentru mesajele GDB. Programul conține și un panou de comandă, conținând funcțiile mai des utilizate.

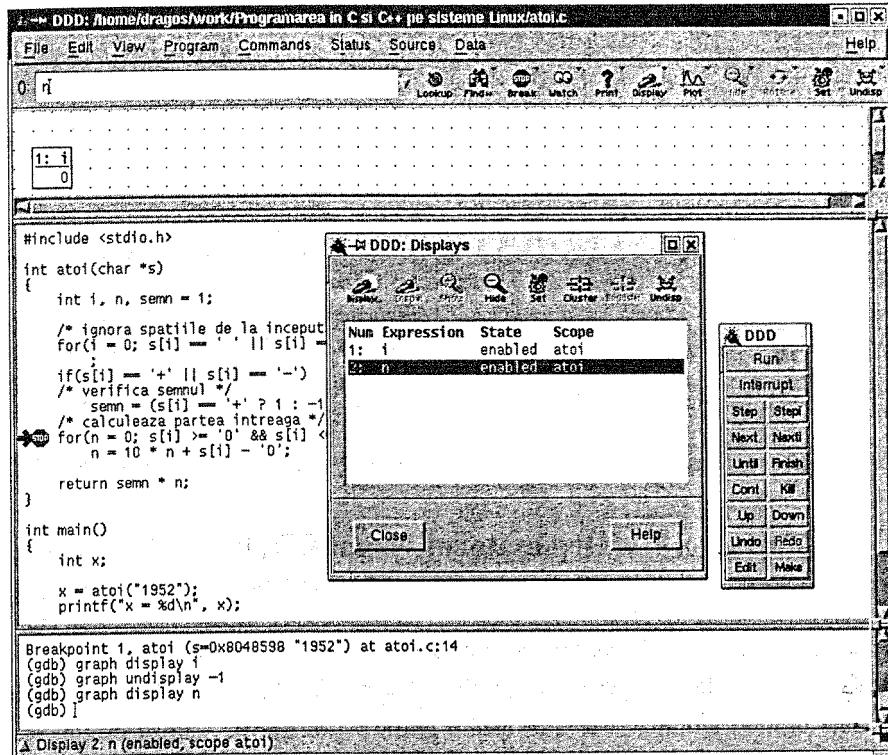


Figura 4.1. Programul DDD

Pot fi gestionate cu ușurință punctele de oprire, utilizând tehnologia *drag-and-drop*. Acestea pot fi salvate pentru sesiuni viitoare de lucru. Programul oferă modalități avansate de inspectare a variabilelor, în special a structurilor și tablourilor.

4.2. Mediul integrat KDevelop

KDevelop este un mediu integrat de dezvoltare (*IDE*) pentru C și C++. El include un editor de texte, o interfață pentru depanatorul GDB, un manager de proiect și un browser sofisticat pentru sursele programelor (browser de clase C++ etc.). KDevelop permite dezvoltarea ușoară de aplicații KDE bazate pe biblioteca Qt, cu interfețe grafice performante. De asemenea, programul include și suport pentru CVS. Programul KDevelop a fost conceput de către Sandy Meier, prima versiune operațională fiind lansată în anul 1998.

Pentru a utiliza acest program trebuie ca pachetul kdevelop să fie instalat. Se lansează în execuție prin comanda cu același nume.

KDevelop conține *vrăjitori* (*wizards*) care permit generarea automată a scheletului proiectului, a structurii de directoare a acestuia, a fișierelor de configurare autoconf și automake, în funcție de interfața proiectului (KDE sau terminal) și a limbajului în care acesta este implementat (C sau C++). De asemenea, codul corespunzător interfeței grafice a programului poate fi generat în mod automat cu ajutorul unui vrăjitor vizual de editare.

Ecranul KDevelop este alcătuit din trei zone:

1. Browser-ul proiectului, conținând mai multe tab-uri: arborele de clase, arborele de fișiere al proiectului, documentații, fereastră de inspectare a variabilelor.
2. Editorul de texte sau mini-navigatorul Web pentru vizualizarea de documentații, după caz.
3. Zona rezervată mesajelor, conținând mai multe tab-uri: mesaje generate de KDevelop, stdout, stderr, consolă (shell), mesaje de depanare etc.

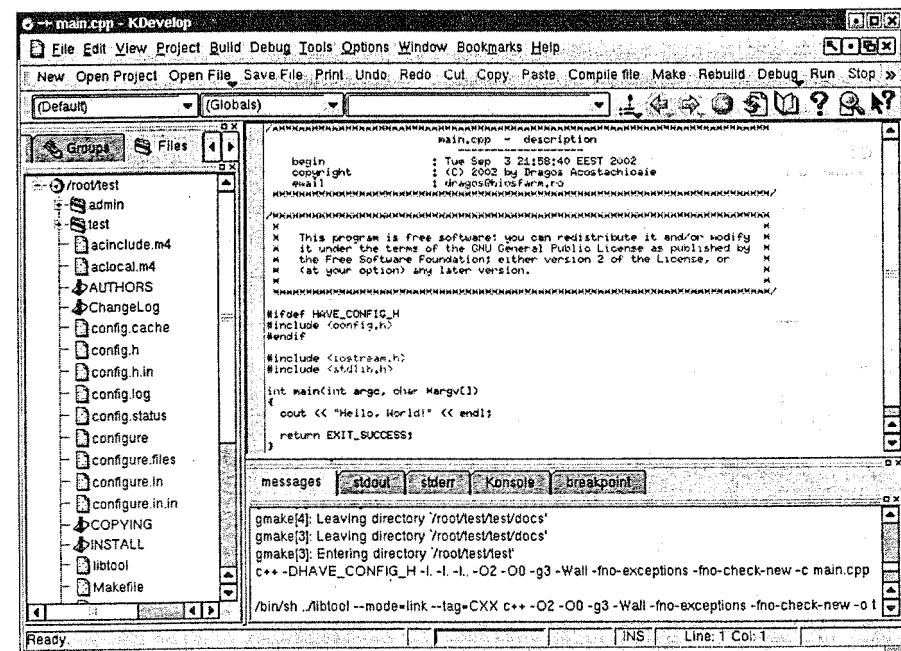


Figura 4.2. Programul KDevelop

4.3. Programul Glade

Glade este un program pentru crearea de interfețe utilizator grafice folosind bibliotecile Gt k+ și GNOME. Glade permite realizarea rapidă a acestor interfețe și generarea C codului corepunzător. Programul Glade a fost creat de către Damon Chaplin în anul 1998.

În vederea utilizării programului Glade, trebuie instalat pachetul glade. Executabilul poartă același nume.

Programul conține trei ferestre: fereastra principală, fereastra Palette, care conține obiectele care pot fi introduse în interfață, și editorul Properties pentru modificarea proprietăților obiectelor (cum ar fi mărimea, culoarea etc.).

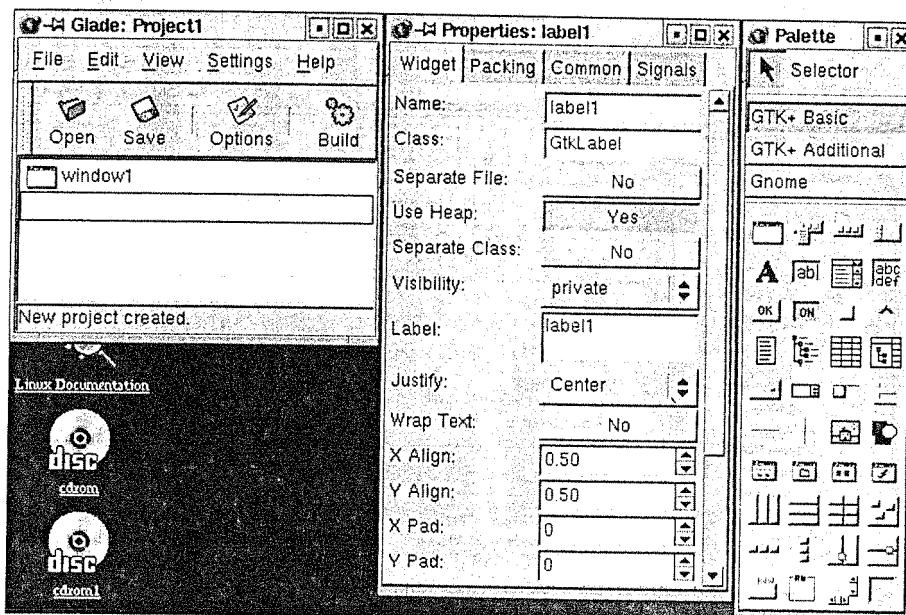


Figura 4.3. Programul Glade

Capitolul 5

PROGRAMAREA OPEN SOURCE

Modificarea comportamentului oamenilor durează întotdeauna mult mai mult decât am vrea să credem. De asemenea, mulți oameni își fac speranțe nerezonabile în privința revoluțiilor.
(Bjarne Stroustrup)

5.1. Ce este *open source*?

5.1.1. Scurt istoric

În anul 1984, Richard Stallman, un cercetător de la Laboratoarele Institutului de Tehnologii din Massachusetts (celebrele MIT Labs), a inițiat proiectul denumit *GNU* (denumire recursivă, conform unei vechi tradiții a hackerilor, însemnând *GNU is Not Unix*). Scopul acestui proiect a fost ca tot software-ul să fie gratuit, din considerentul că toate cunoștințele înglobate într-un program trebuie să fie publice.

Primul program realizat de către Stallman în cadrul proiectului GNU a fost Emacs, care a devenit operațional în 1985. Emacs a fost făcut disponibil pe serverul `ftp prep.ai.mit.edu`. Fiindcă multe persoane erau interesate de acest editor de texte, dar nu aveau acces la Internet, Stallman a pus la dispoziția publicului o bandă (*tape*) conținând programul, contra sumei de 150 \$. În acest mod a început prima afacere de distribuire de *software free*.

Prima intenție a proiectului GNU a fost aceea de a implementa un sistem UNIX complet, de la nucleu și până la aplicațiile pentru utilizatori finali. În anul 1990, proiectul era aproape finalizat, singura componentă majoră care lipsea fiind nucleul unui sistem de operare. Această problemă a fost rezolvată o dată cu apariția nucleului Linux, adoptat în 1992 ca nucleu oficial al proiectului GNU. Acest sistem de operare complet – alcătuit din nucleul Linux și din aplicațiile GNU – este denumit *GNU/Linux*.

În 1997, câțiva dintre liderii lumii software-ului liber (dintre care Eric Raymond, Tim O'Reilly și Larry Augustin) s-au întâlnit în California pentru a găsi o cale de a promova ideile de bază ale acestui concept către persoanele care erau opuși lui. Grupul a concluzionat că, într-o bună măsură, neajunsurile se datorau lipsiei unei campanii de marketing care să aibă drept scop câștigarea nu neapărat de procente de piață, ci de procente de acceptare intelectuală. De asemenea, s-a hotărât renunțarea la termenul *software free*, în favoarea termenului *open source*.

5.1.2. Ce este *open source*?

Termenul *open source* (care poate fi întâlnit și sub numele de *software free*) desemnează programele de calculator fără restricții de distribuție. Cuvântul „free” se referă la libertate, nu la preț. Oricine are libertatea de a distribui copii a software-ului *open source*, să facă rost de codul-sursă, să modifice și să utilizeze programul după bunul său plac.

Întrebarea pe care o poate pune un utilizator, mai ales dacă a lucrat până acum cu programe proprietare, poate suna astfel: *Este legal să folosesc și să distribui programe open source?* Este perfect legal atât timp cât programele sunt protejate de o licență care să permită să faceți aceste lucruri. *The Free Software Foundation* a creat *GNU General Public License* (pe scurt *GPL*), funcție care protejează aproape toate programele din lumea Linux, care garantează că veți avea întotdeauna libertatea de a utiliza și distribui programele, fără a avea dreptul să îngădăți altor persoane accesul la acestea. De asemenea, licența *GPL* stipulează că orice produs dezvoltat pe baza programului sub această licență trebuie să fie tot un produs aflat sub licență *GPL*.

Mai există și alte câteva licențe speciale derivate din *GPL*. Astfel, pentru biblioteci (cum ar fi biblioteca standard C) este utilizată licența *LGPL* (*GNU Library General Public License*), care permite utilizarea bibliotecii de către un program proprietar sau compilarea unui program pentru a putea rula pe un sistem *GNU*. Manualele *GNU* sunt protejate de asemenea de *GPL*, dar folosesc o versiune simplificată deoarece complexitatea licenței nu se justifică în acest caz.

Programele (proiectele) *open source* sunt distribuite în format-sursă, putând fi descărcate fără restricții de pe Internet. Multe dintre acestea pot fi găsite deja compilate (ca fișiere .rpm – pentru distribuțiile Red Hat, Mandrake etc., .deb – pentru Debian, .tgz – pentru Slackware etc.). În majoritatea cazurilor, aceste programe sunt dezvoltate de zeci sau chiar sute de programatori din toată lumea, coordonați prin intermediul Internetului. Proiectele *open source* sunt întreținute și coordonate de către una sau mai multe persoane, numite *maintainers*. Acestea sunt alese în mod democratic, prin acord comun al membrilor comunității implicați în respectivul proiect. Ele au responsabilitatea de a lansa versiuni noi, de a recepționa și sortă modificările efectuate de către alți programatori și.m.d.

În modelul de dezvoltare *open source*, împărtășirea codului-sursă facilitează creativitatea. Programatorii care lucrează la proiecte concurente pot utiliza rezultatele celorlalți, sau pot combina resursele într-un singur proiect. Un proiect nu este abandonat atunci când un programator nu mai lucrează la el. Având la dispoziție codul-sursă, oricine îl poate prelua și continua.

Cuvântul în jurul căruia se învârté totul este *cooperare*. Ideea principală a modelului *open source* este că dacă toți cooperează, toți au de câștigat.

Este foarte important să remarcăm că atât conceptul *open source* și conceptul *open system* (sistem deschis) pot fi întâlnite nu doar în lumea calculatoarelor, ci în

aproape orice domeniu de activitate. Din realitatea dezvoltării programelor urmând modelul *open source*, a rezultat un nou concept: *open science* (știință deschisă).

De cele mai multe ori, calitatea programelor *open source* este cu mult superioară celei a produselor comerciale (proprietare).

Aproape de fiecare dată, proiectele *open source* sunt scrise de către persoane care au nevoie de respectivele programe pentru propriile lor interese, și mai rar de către persoane care au fost plătită pentru aceasta. Faptul determină programatorii să realizeze soft cât mai potrivit pentru nevoile lor și care să conțină cât mai puține bug-uri. Deoarece programele sunt distribuite în format-sursă, oricine descoperă un bug și poate corecta și poate trimite corecțiile sale *maintainer-ului*, și imediat versiunea corectată este făcută publică (fie prin lansarea unei versiuni noi, fie prin intermediul unui fișier *patch*, fie printr-un arbore CVS). În contrast cu aceasta, mariile companii dezvoltătoare de soft răspund mult mai greu la sugestii și la raportări de bug-uri, chiar dacă mulți utilizatori depind de corectarea respectivelor erori. Mai multe facilități, algoritmi mai performanți, corecții sunt mult mai simplu de implementat în cazul programelor *open source*, iar respectivele modificări pot fi integrate cu ușurință în programe, făcându-le mai bune pentru toată lumea, într-un timp foarte scurt.

De altfel, a plăti un programator să facă ceva nu înseamnă neapărat că va scrie cod de calitate. Când contribui la realizarea unui soft *open source*, nimeni nu te forțează să scrii cod. Îți place și ai timp pentru asta. Dacă toți cooperează, toți au de câștigat.

Disponibilitatea surselor mai generează și un alt avantaj: permite elevilor, studenților și programatorilor să citească și să utilizeze codul-sursă. Astfel, ei pot învăța din acel cod mai multe decât să citească cărți de genul „Învățăți C în 21 de zile”, de o calitate îndoieifică. Nu este nevoie să „reinventeze roata”; ei pot prelua porțiuni din codul care este deja scris și să le folosească pentru propriile nevoi.

Multe companii nu vor să aibă de-a face cu programe *open source* pentru că își închipuie că nimeni nu le oferă asistență tehnică. În general, persoanele care lucrează în domeniul *open source* sunt mult mai *open minded* (deschise) decât companiile care produc și comercializează programe proprietare. Dacă puneți o întrebare pe o listă de discuții sau cereți ajutorul însuși autorului, în mod sigur cineva vă va ajuta cu plăcere, atât cât îi stă în putință. Este posibil chiar să primiți ajutor din mai multe părți o dată! Pe de altă parte, există și companii care oferă asistență tehnică pentru soft *open source*.

Faptul că programele sunt libere nu implică nicidcum că programatorii să muncească „pe degeaba” și alții să profite de munca lor, cum s-ar putea crede la prima vedere. Chiar dacă nu se poate vinde programul în sine, distribuția și oferirea de asistență tehnică sunt activități profitabile. Companii ca Red Hat sau Cygnus dovedesc că software-ul *open source* nu este o utopie.

Fenomenul *open source* este o adevărată revoluție. El a schimbat modul de gândire al oamenilor și a introdus modele noi de a face afaceri. Astfel, Richard Stallman a început prima afacere bazată pe programe *open source* prin distribuția

editorului de texte Emacs. Compania americană Red Hat Software Inc. produce și distribuie distribuția Red Hat Linux, însăși de un manual de utilizare. Evident, distribuția poate fi descărcată și de pe Internet, în mod gratuit. Totodată, firma Red Hat dezvoltă cod nou (e.g. GNOME), care este *open source* și care se întoarce în cadrul comunității Linux. Compania Cygnus își bazează afacerea pe îmbunătățirea și oferirea de suport tehnic pentru cel mai bun compilator de C/C++ din lume, GCC. Îmbunătățirile aduse sunt integrate apoi în cadrul GCC. Co-fondatorul acestei companii, John Gilmore, descria filosofia companiei astfel: „*Să facem soft-ul liber profitabil*”. Firma VA Research își livrează serverele și stațiile de lucru cu sistemul Linux preinstalat, oferind totodată și suport tehnic pentru acesta. Exemplul pot continua, cu IBM (prin portarea DB2 pe Linux și prin suportul pentru platforma AS/400). Netscape (migrarea navigatorului Netscape pe dezvoltare *open source*) și.a.m.d., ele dovedind eficiența acestui model de afacere.

5.2. Managementul Web al proiectelor software – sourceforge.net

sourceforge.net este un serviciu gratuit pentru dezvoltatorii de programe *open source*, oferind un acces facil la CVS, liste de discuții, urmărirea bug-urilor, forumuri de mesaje, gestiunea sarcinilor, arhivarea fișierelor, precum și administrarea tuturor acestora prin intermediul Web-ului.

sourceforge.net este proprietatea VA Linux Systems Inc., SUA, una dintre firmele care susțin sistemul de operare Linux și își bazează toate produsele pe acesta.

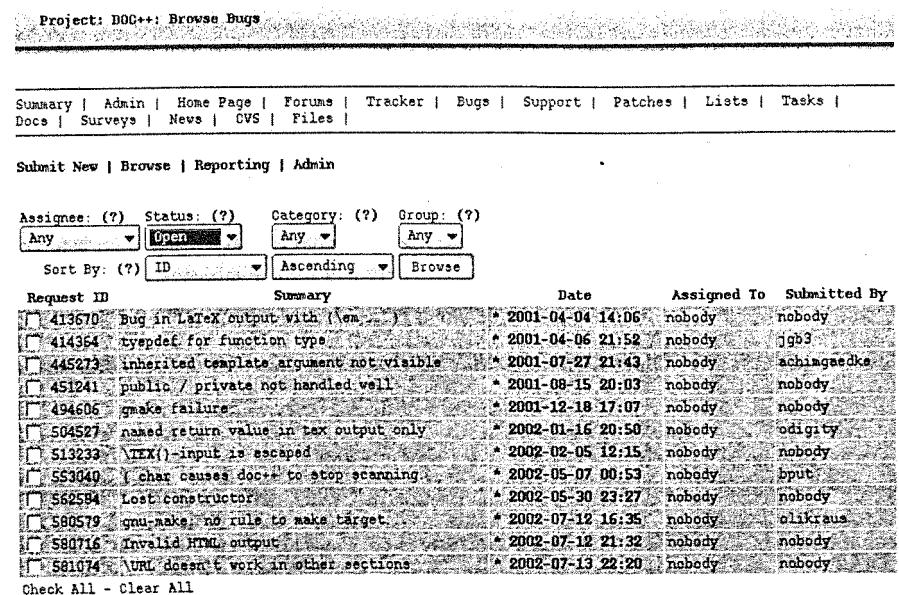
Iată gradul de utilizare al sourceforge.net înregistrat în luna august 2002:

- peste 45.000 proiecte găzduite;
- peste 450.000 utilizatori înregistrați;
- peste 3 milioane de pagini vizualizate pe zi;
- peste 300.000 fișiere transferate pe zi;
- peste 800.000 e-mail-uri trimise pe listele de discuții pe zi.

sourceforge.net găzduiește în acest moment câteva mii de proiecte *open source*. Sistemul utilizează și el doar programe *open source*.

Fiecare proiect *open source* găzduit primește un host virtual cu numele <http://numeproiect.sourceforge.net>. Fiecare proiect are propriul spațiu pentru conținutul Web și pentru script-uri CGI. De asemenea, sunt oferite script-uri PHP pentru a putea construi o prezență mai rafinată a proiectului pe Web. Din cadrul script-urilor PHP pot fi accesate și baze de date MySQL. Fiecare proiect dispune în total de 100 MB de spațiu. sourceforge.net oferă de asemenea acces la mașini care rulează diferite sisteme de operare, permitând dezvoltatorilor să compileze și să își testeze proiectele pe multiple platforme.

Accesul la fișierele proiectului și la CVS se face fie prin SSH, pentru securitate sporită, fie printr-o interfață Web. Administrarea proiectului se realizează în totalitate prin intermediul unei interfețe Web.

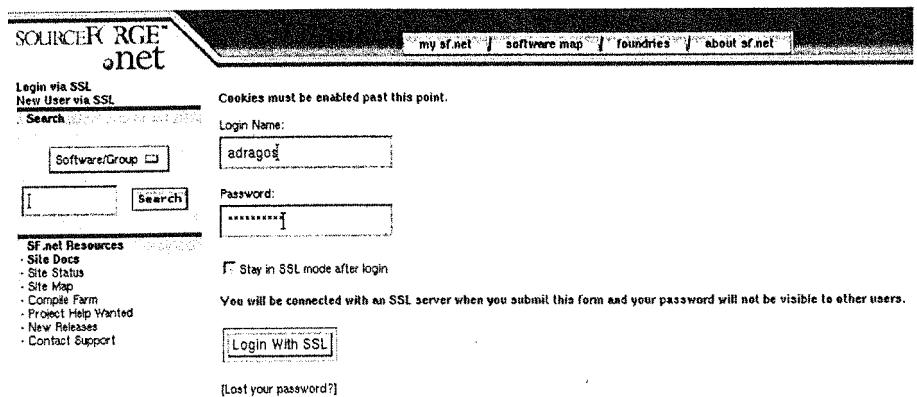


The screenshot shows a web-based bug tracking system for the DOG++ project. At the top, there's a header with links for Summary, Admin, Home Page, Forums, Tracker, Bugs, Support, Patches, Lists, Tasks, Docs, Surveys, News, CVS, and Files. Below the header, there are search filters for Assignee, Status, Category, and Group, and a sort dropdown for Request ID, Date, Summary, Assigned To, and Submitted By. A table lists several bugs, each with a checkbox, an ID, a summary description, a date, and the names of the assignee and submitter. The bugs listed include various LaTeX and make-related issues. At the bottom of the table, there are links for 'Check All' and 'Clear All'.

Request ID	Summary	Date	Assigned To	Submitted By
413670	Bug in LaTeX output with (\emph{...})	2001-04-04 14:06	nobody	nobody
414364	typedef for function type	2001-04-06 21:52	nobody	jgb3
445273	Inherited template argument not visible	2001-07-27 21:43	nobody	achingaedke
451241	public / private not handled well	2001-08-15 20:03	nobody	nobody
494606	make failure	2001-12-18 17:07	nobody	nobody
504527	named return value in tex output only	2002-01-16 20:50	nobody	odigitv
513233	TEX() - input is escaped	2002-02-05 12:15	nobody	nobody
553049	\char causes docvt to stop scanning	2002-05-07 00:53	nobody	bput
562584	Lost constructor	2002-05-30 23:27	nobody	nobody
580579	gnu-make: no rule to make target	2002-07-12 16:35	nobody	olikraus
580716	Invalid HTML output	2002-07-12 21:32	nobody	nobody
581074	URL doesn't work in other sections	2002-07-13 22:20	nobody	nobody

Figura 5.1. Gestionearea unui proiect cu sourceforge.net

Pentru a putea utiliza serviciile oferite de sourceforge.net, trebuie înregistrat un utilizator. Autentificarea se poate face prin intermediul SSL:



The screenshot shows the SSL login page for sourceforge.net. At the top, there's a banner with links for my sf.net, software map, founders, and about sf.net. Below the banner, there's a "Login via SSL" section with fields for "Login Name" (set to "adragos") and "Password". There are checkboxes for "Software/Group", "Search", "Cookies must be enabled past this point.", "Stay in SSL mode after login", and "You will be connected with an SSL server when you submit this form and your password will not be visible to other users.". At the bottom, there's a "Login With SSL" button and links for "Lost your password?" and "New Account".

Figura 5.2. Intrarea în sourceforge.net

Înregistrarea de noi proiecte software se face cu ușurință, prin completarea unui formular, cererea urmând a fi aprobată în decurs de 1-2 zile (personalul sourceforge.net va verifica dacă proiectul nu este deja înregistrat, dacă este *open source* etc.).

Misiunea sourceforge.net este aceea de a ajuta comunitatea *open source*, oferind dezvoltatorilor un sistem centralizat și uniform pentru gestiunea și controlul dezvoltării de programe *open source*. Creatorul acestui sistem, VA Linux Systems, a înțeles că rolul serviciilor oferite de Internet este unul fundamental pentru dezvoltarea software-ului *open source*.

Anexe

Anexa 1

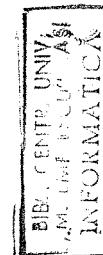
FUNCTIILE STANDARD ANSI C

Nume funcție	Fișier-antet	Descriere
abort	stdlib.h	Încheie forțat execuția procesului
abs	stdlib.h	Calculează valoarea absolută
acos	math.h	Calculează arc cosinus
asctime	time.h	Returnează sirul corespunzător unui moment de timp
asin	math.h	Calculează arc sinus
assert	assert.h	Stabilește o expresie care va încheia execuția programului
atan	math.h	Calculează arc tangenta
atan2	math.h	Calculează arc tangenta
atof	stdlib.h	Convertește un sir de caractere în double
atoi	stdlib.h	Convertește un sir de caractere într-un int
atol	stdlib.h	Convertește un sir de caractere într-un long
bsearch	stdlib.h	Sortează un sir de valori
calloc	stdlib.h	Alocă spațiu de memorie
ceil	math.h	Rotunjeste un double
clearerr	stdio.h	Anulează indicatorul de eroare
cos	math.h	Calculează cosinus
cosh	math.h	Calculează cosinus hiperbolic
ctime	time.h	Returnează sirul corespunzător unui moment de timp
exit	stdlib.h	Încheie execuția programului
exp	math.h	Calculează exponentiala
fabs	math.h	Calculează valoarea absolută
fclose	stdio.h	Închide fișierul

feof	stdio.h	Verifică dacă s-a ajuns la sfârșitul fișierului
ferror	stdio.h	Verifică indicatorul de eroare
fflush	stdio.h	Scrie pe disc datele nesalvate
fgetc	stdio.h	Citește un caracter dintr-un fișier
fgets	stdio.h	Citește un sir de caractere dintr-un fișier
floor	math.h	Rotunjește în minus
fmod	math.h	Calculează restul împărțirii
fopen	stdio.h	Deschide un fișier
fprintf	stdio.h	Scrie date formatare într-un fișier
fputc	stdio.h	Scrie un caracter într-un fișier
fputs	stdio.h	Scrie un sir de caractere într-un fișier
fread	stdio.h	Citește informații dintr-un fișier
free	stdlib.h	Eliberează spațiul de memorie alocat pentru un pointer
freopen	stdio.h	Redeschide un fișier
frexp	math.h	Convertește un număr în parte fracțională și parte întreagă
fscanf	stdio.h	Citește date formatare dintr-un fișier
fseek	stdio.h	Setează poziția pointerului într-un fișier
ftell	stdio.h	Returnează poziția cursorului într-un fișier
fwrite	stdio.h	Scrie informații într-un fișier
getc	stdio.h	Citește un caracter dintr-un fișier
getchar	stdio.h	Returnează un caracter citit de la intrarea standard
getenv	stdlib.h	Citește conținutul unei variabile de mediu
gets	stdio.h	Citește un sir de caractere de la intrarea standard
gmtime	time.h	Transformă o valoare de timp în structura corespunzătoare
isalnum	ctype.h	Verifică dacă un caracter este alfanumeric
isalpha	ctype.h	Verifică dacă un caracter este literă
iscntrl	ctype.h	Verifică dacă un caracter este caracter de control
isdigit	ctype.h	Verifică dacă un caracter este cifră
isgraph	ctype.h	Verifică dacă un caracter este tipăribil exceptând spațiu
islower	ctype.h	Verifică dacă un caracter este literă mică

isprint	ctype.h	Verifică dacă un caracter este tipăribil, inclusiv spațiu
ispunct	ctype.h	Verifică dacă un caracter este tipăribil, exceptând spațiu sau caractere alfanumerice
isspace	ctype.h	Verifică dacă un caracter este caracter de spațiere
isupper	ctype.h	Verifică dacă un caracter este literă mare
isxdigit	ctype.h	Verifică dacă un caracter este cifră hexazecimală
ldexp	math.h	Multiplică un număr cu o putere a lui 2
localtime	time.h	Transformă o valoare de timp în structura corespunzătoare
log	math.h	Calculează logaritmul natural
log10	math.h	Calculează logaritmul natural în baza 10
longjmp	setjmp.h	Sare la o adresă în afara programului curent
malloc	stdlib.h	Alocă spațiu de memorie pentru un pointer
mktime	time.h	Transformă o structură de timp în valoarea corespunzătoare
modf	math.h	Extrage părțile întregi și fractionale
perror	stdio.h	Trimite un mesaj de eroare la ieșirea de eroare standard
pow	math.h	Ridică un număr la putere oarecare
printf	stdio.h	Scrie date formatare la ieșirea standard
putc	stdio.h	Scrie un caracter într-un fișier
putchar	stdio.h	Scrie un caracter la ieșirea standard
puts	stdio.h	Scrie un sir de caractere la ieșirea standard
qsort	stdlib.h	Sorteză un tablou de elemente
rand	stdlib.h	Returnează un număr aleatoriu
realloc	stdlib.h	Modifică dimensiunea spațiului de memorie alocat pentru un pointer
remove	stdio.h	Șterge un fișier
rename	stdio.h	Redenumește un fișier
rewind	stdio.h	Pozitionează pointerul de fișier la începutul acestuia
scanf	stdio.h	Citește date formatare de la intrarea standard
setbuf	stdio.h	Atașează un buffer pentru fișier
setjmp	setjmp.h	Salvează mediul curent

setlocale	locale.h	Stabilește localizarea de țară
sin	math.h	Calculează sinusul
sinh	math.h	Calculează sinusul hiperbolic
sprintf	stdio.h	Scrie date formatațate într-un șir
sqrt	math.h	Calculează radicalul
srand	stdlib.h	Setează „sămânța” pentru generatorul de numere aleatorii
sscanf	stdio.h	Citește date formatațate dintr-un șir
strcat	string.h	Concatenează două șiruri
strchr	string.h	Caută un caracter într-un șir
strcmp	string.h	Compară două șiruri
strcpy	string.h	Copiează un șir într-un altul
strcspn	string.h	Returnează numărul de caractere dintr-un șir care este inclus într-un alt șir
strftime	time.h	Scrie echivalentul unui moment de timp într-un șir
strlen	string.h	Returnează lungimea unui șir
strncat	string.h	Concatenează două șiruri, cu o lungime maximă specificată
strncmp	string.h	Compară un număr de caractere din două șiruri
strncpy	string.h	Copiează un număr de caractere dintr-un șir într-altul
strpbrk	string.h	Caută un șir de caractere într-un alt șir
strrchr	string.h	Caută un caracter într-un șir
strspn	string.h	Returnează numărul de caractere dintr-un șir care este inclus într-un alt șir
strstr	string.h	Caută un șir de caractere într-un alt șir
strtok	string.h	Caută un şablon într-un șir
tan	math.h	Calculează tangenta
tanh	math.h	Calculează tangentă hiperbolică
time	time.h	Returnează timpul curent
tmpfile	stdio.h	Deschide un fișier temporar
tmpnam	stdio.h	Returnează un nume unic de fișier temporar
tolower	ctype.h	Convertește un caracter în literă mică
toupper	ctype.h	Convertește un caracter în literă mare
ungetc	stdio.h	Pune un caracter înapoi într-un fișier



Anexa 2

FUNCȚIILE STANDARD POSIX

Nume funcție	Fișier-antet	Descriere
access	unistd.h	Verifică dacă identificatorul de utilizator sub care se execută procesul curent are dreptul de acces specificat asupra fișierului furnizat ca parametru
alarm	unistd.h	Programează trimiterea semnalului SIGALRM după un interval de timp
cfgetispeed	termios.h	Returnează viteza de transfer pentru intrare (<i>baud rate</i>) a terminalului
cfgetospeed	termios.h	Returnează viteza de transfer pentru ieșirea a terminalului
cfsetispeed	termios.h	Stabilește viteza de transfer pentru intrare
cfsetospeed	termios.h	Stabilește viteza de transfer pentru ieșire
chdir	unistd.h	Schimbă directorul curent
chmod	sys/types.h, sys/stat.h	Modifică drepturile de acces ale unui fișier
close	unistd.h	Închide un descriptor de fișier
closedir	sys/types.h, dirent.h	Închide un descriptor de director
creat	sys/types.h, sys/stat.h, fcntl.h	Creează și deschide pentru scriere un fișier
ctermid	stdio.h, unistd.h	Returnează numele terminalului curent
dup	unistd.h	Realizează o copie a unui descriptor de fișier
dup2	unistd.h	Închide un descriptor de fișier și realizează o copie a sa
execle, execve, execvp, execv,	unistd.h	Apelează un fișier executabil, suprascriind procesul actual în execuție
execve, execvp		
_exit	unistd.h	Încheie execuția procesului curent și închide toate fișierele deschise

fcntl	sys/types.h, unistd.h, fcntl.h	Controlează sau returnează atrитеle unui fișier deschis
fdopen	stdio.h	Deschide un flux utilizând un descriptor de fișier deja deschis
fileno	stdio.h	Returnează descriptorul de fișier asociat unui flux
fork	sys/types.h, unistd.h	Creează un proces copil, care este o copie a procesului curent
fpathconf	unistd.h	Returnează valoarea unei variabile pentru un anumit descriptor de fișier
fstat	sys/types.h, sys/stat.h	Obține starea curentă a unui fișier deschis
getcwd	unistd.h	Returnează calea absolută pentru directorul curent
getegid	sys/types.h, unistd.h	Returnează identificatorul de grup efectiv al procesului curent
getenv	stdlib.h, unistd.h	Returnează valoare unei anumite variabile de mediu
geteuid	sys/types.h, unistd.h	Returnează identificatorul de utilizator efectiv al procesului curent
getgid	sys/types.h, unistd.h	Returnează identificatorul de grup real al procesului curent
getgrgid	grp.h	Caută un identificator de grup în baza de date de grupuri (/etc/group)
getgrnam	grp.h	Caută un nume de grup în baza de date de grupuri
getgroups	sys/types.h, unistd.h	Obține grupurile suplimentare asociate cu procesul curent
getlogin	unistd.h	Returnează numele utilizatorului curent
getpgrp	sys/types.h, unistd.h	Returnează identificatorul de grup al procesului curent
getpid	sys/types.h, unistd.h	Returnează identificatorul procesului curent
getppid	sys/types.h, unistd.h	Returnează identificatorul procesului părinte
getpwnam	pwd.h	Caută un nume de utilizator în baza de date de utilizatori (/etc/passwd)
getpwuid	pwd.h	Caută un identificator de utilizator în baza de date de utilizatori

getuid	sys/types.h, unistd.h	Returnează identificatorul de utilizator real al procesului curent
isatty	unistd.h	Verifică dacă un descriptor de fișier are asociat un terminal
kill	sys/types.h, signal.h	Trimite un semnal unui proces specificat
link	unistd.h	Creează o legătură simbolică
lseek	unistd.h, sys/types.h	Pozitionează pointerul curent în cadrul unui descriptor de fișier
mkdir	sys/types.h, sys/stat.h	Creează un director
mkfifo	sys/types.h, sys/stat.h	Creează un FIFO
open	sys/types.h, sys/stat.h, fcntl.h	Deschide un fișier
opendir	sys/types.h, dirent.h	Deschide un director
pathconf	unistd.h	Returnează valoarea unei variabile pentru o anumită cale
pause	unistd.h	Blochează procesul până la recepționarea unui semnal
pipe	unistd.h	Creează un pipe
read	unistd.h	Citește informații dintr-un fișier
readdir	sys/types.h, dirent.h	Citește următoarea intrare din director
rename	unistd.h	Redenumește un fișier
rewinddir	sys/types.h, dirent.h	Recitește conținutul unui director
rmdir	unistd.h	Sterge un director
setgid	sys/types.h, unistd.h	Setează grupul sub care este executat procesul
setpgid	sys/types.h, unistd.h	Setează grupul pentru un proces specificat
setsid	sys/types.h, unistd.h	Modifică sesiunea din care face parte procesul într-o nouă sesiune
setuid	sys/types.h, unistd.h	Modifică identificatorul de utilizator sub care se execută procesul
sigaction	signal.h	Asignează tabelul de tratamente ale semnalelor

sigaddset	signal.h	Adaugă un semnal la o tabelă de tratamente ale semnalelor
sigdelset	signal.h	Sterge un semnal dintr-o tabelă de tratamente ale semnalelor
sigemptyset	signal.h	Sterge toate semnalele dintr-o tabelă de tratamente ale semnalelor
sigfillset	signal.h	Adaugă toate semnalele cunoscute într-o tabelă de tratamente ale semnalelor
sigismember	signal.h	Verifică dacă un semnal este membru al unei tabele de tratamente ale semnalelor
siglongjmp	setjmp.h	Execută salt la un context salvat anterior cu sigsetjmp
sigpending	signal.h	Modifică tabelul de tratamente ale semnalelor
sigsetjmp	setjmp.h	Salvează contextul curent
sigsuspend	signal.h	Modifică masca de semnale și blochează procesul până la recepționarea unui semnal
sleep	unistd.h	Blochează procesul pentru un interval de timp
stat	sys/types.h, sys/stat.h	Obține informații de stare privind un anumit fișier
sysconf	unistd.h	Returnează valoarea unei variabile sistem
tcdrain	termios.h	Blochează procesul până când toate informațiile scrise într-un descriptor de fișier au fost transmise
tcflow	termios.h	Oprește și repornește operațiunile de intrare/ieșire cu un anumit descriptor de fișier
tcflush	termios.h	Termină operațiunile de intrare/ieșire rămase de efectuat cu un anumit descriptor de fișier
tcgetattr	termios.h	Obține valorile atributelor asociate unui terminal
tcgetpgrp	termios.h	Returnează identificatorul de grup al procesului asociat unui terminal
tcsendbreak	termios.h	Trimită către un terminal un sir de octeți având valoarea zero, pentru un anumit timp
tcsetattr	termios.h	Stabilește valorile atributelor asociate unui terminal
tcsetpgrp	termios.h	Stabilește identificatorul de grup al procesului asociat unui terminal

time	time.h	Returnează timpul curent (exprimat în secunde trecute de la începutul epocii, adică 1 ianuarie 1970)
times	sys/times.h	Obține timpul utilizator și timpul sistem asociate procesului, precum și cele asociate proceselor copil
ttyname	unistd.h	Returnează numele unui terminal
tzset	time.h	Stabilește informațiile de conversie a timpului
umask	sys/types.h, sys/stat.h	Stabilește masca pentru drepturile de acces la crearea fișierelor
uname	sys/utsname.h	Obține informații de identificare asupra sistemului
unlink	stdio.h	Sterge un fișier
utime	sys/types.h, utime.h	Stabilește timpul de acces, respectiv de modificare a unui fișier
wait	sys/types.h, sys/wait.h	Blochează procesul curent până când se încheie un proces copil
waitpid	sys/types.h, sys/wait.h	Obține informații despre un anumit proces copil care și-a încheiat execuția sau a fost oprit
write	unistd.h	Scrie informații într-un fișier

Anexa 3

LICENȚA PUBLICĂ GNU

Aceasta este o traducere neoficială a Licenței Publice Generale GNU în limba română. Nu a fost publicată de Free Software Foundation și nu specifică termenii legali de distribuire a programelor care folosesc GNU GPL – numai textul original în limba engleză al GNU GPL face acest lucru. Totuși, sperăm că această traducere să ajute vorbitorii de limba română să înțeleagă mai bine GNU GPL. Textul traducerii a fost preluat de la adresa: <http://www.roedu.net/gplro.html>

Versiunea originală a acestei licențe poate fi găsită la adresa: <http://www.gnu.org/licenses/gpl.html>.

Licența publică generală GNU

Versiunea 2, iunie 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Este permisă copierea acestui document, dar este interzisă modificarea lui.

Prefață

Licențele majorității programelor sunt concepute pentru a vă priva de libertatea de a modifica și distribui programele respective. În contrast, intenția Licenței Publice Generale GNU este de a vă garanta libertatea de a distribui și modifica programele libere și de a se asigura că programele sunt libere pentru toți utilizatorii. Această Licență Publică Generală se aplică majorității programelor aparținând Free Software Foundation precum și tuturor celorlalte programe ai căror autori decid să o folosească. Alte programe aparținând Free Software Foundation sunt puse sub Licența Publică Generală GNU pentru Biblioteci. Această Licență poate fi de asemenea folosită pentru programele dumneavoastră.

Libertatea programelor nu implică neapărat absența costului. Licențele noastre sunt concepute să vă garanteze libertatea de a distribui copii ale programelor libere (și de a oferi

acest serviciu contra cost, dacă dorîți), de a obține codul-sursă, de a schimba programul sau a folosi porțiuni din el în noi programe libere, și de a ști că puteți face toate lucruri.

Pentru a vă proteja drepturile, trebuie să impunem restricții împotriva oricui ar încerca să vă conteste aceste drepturi sau să vă ceară să renunțați la ele. Aceste restricții implică anumite responsabilități pentru dumneavoastră dacă distribuiți copii ale programelor, sau dacă le modificați.

De exemplu, dacă distribuiți copii ale unui program, indiferent dacă o faceți gratuit sau contra unei sume de bani, trebuie să dați beneficiarilor toate drepturile pe care le aveți dumneavoastră. Trebuie să vă asigurați că ei primesc, sau pot primi, codul-sursă al programului. În plus, trebuie să le arătați care sunt termenii în care primesc programul, pentru ca ei să știe care le sunt drepturile.

Drepturile dumneavoastră sunt protejate în două etape: (1) prin stabilirea drepturilor de autor pentru program, și (2) prin această Licență care vă dă dreptul legal de a copia, distribui și/sau modifica programul.

De asemenea, pentru propria noastră protecție cât și pentru cea a autorilor, vrem să ne asigurăm că toată lumea înțelege că nu există nici un fel de garanție pentru acest program liber. Dacă programul este modificat de altcineva și distribuit mai departe, vrem ca beneficiarii programului să știe că ceea ce au nu este originalul, în aşa fel încât nici o problemă introdusă de altcineva nu va avea un efect negativ asupra reputației autorilor inițiali.

Orice program liber este în mod constant amenințat de patentele software. Noi vrem să evităm pericolul ca cei ce redistribuie programe libere să obțină patente, practic transformând programul într-unul aflat sub controlul total al persoanei sau instituției ce dețin patentul (engl. *proprietary*). Pentru a preveni această situație, facem clară poziția noastră conform căreia orice patent trebuie acordat fie în aşa fel încât să poată fi folosit gratuit și fără restricții de oricine, fie deloc.

Termenii și condițiile exacte de copiere, distribuire și modificare sunt specificate în următoarele paragrafe.

Licența publică generală GNU

Termeni și condiții pentru copiere, distribuire și modificare

0. Această Licență se aplică oricărui program sau proiect ce conține o mențiune a deținătorului drepturilor de autor spunând că poate fi distribuit în termenii acestei Licențe Publice Generale. Prin „Program” vom înțelege orice asemenea program sau proiect, iar prin „proiect bazat pe Program” vom înțelege fie programul fie orice alt proiect derivat din Program conform cu legea drepturilor de autor: un proiect ce conține Programul sau porțiuni din el, fie în forma originală, fie modificată și/sau tradusă în altă limbă (în restul acestui document traducerile vor fi incluse fără restricții în termenul „modificare”). Fiecare persoană autorizată de această Licență va fi desemnată prin termenul „dumneavoastră”.

Activitățile care nu sunt de copiere, distribuire și modificare se află în afara scopului acestei Licențe. Activitatea de executare a programului nu este restricționată, iar rezultatul programului (engl. *output*) este acoperit de licență doar în cazul în care conținutul său constituie un proiect bazat pe Program (independent de faptul că a fost obținut prin rularea Programului). În ce măsură acest lucru este adevărat depinde de natura Programului.

1. Puteți copia și distribui copii nemodificate ale codului-sursă al Programului în forma în care îl primiți, prin orice mediu, cu condiția să specificați vizibil pe fiecare copie autorul și lipsa oricărei garanții, să păstrați intacte toate notele referitoare la această Licență și la absența oricărei garanții și să distribuiți o copie a acestei Licențe cu fiecare copie a Programului.

Puteți pretinde o retribuție finanică pentru actul fizic de transfer al unei copii, și puteți oferi garanție contra cost.

2. Puteți efectua modificări asupra copiilor Programului (sau asupra oricăror portiuni ale sale), creând astfel un „proiect bazat pe Program”. Copiera și distribuirea unor asemenea modificări sau proiecte se pot face conform termenilor secțiunii precedente (1), doar dacă toate condițiile următoarele sunt îndeplinite:

- toate fișierele modificate trebuie să conțină note foarte vizibile menținând faptul că dumneavaoastră le-ați modificat, precum și data fiecărei modificări;
- orice proiect pe care îl distribuiți sau publicați, care în întregime sau în parte conține sau este derivat din Program (sau orice parte a acestuia), trebuie să poată fi folosit de oricine, gratuit și în întregime, în termenii acestei Licențe;
- dacă programul modificat citește comenzi în mod interactiv, trebuie să îl modificați în aşa fel încât atunci când este pornit în mod interactiv să afișeze un mesaj referitor la drepturile de autor precum și o notă menținând lipsa oricărei garanții (sau să menționeze faptul că dumneavaoastră oferiți o garanție). De asemenea trebuie specificat faptul că utilizatorii pot redistribui programul în aceste condiții precum și o explicație a modalității în care poate fi obținut textul acestei Licențe (excepție: dacă Programul este interactiv dar nu afișează în mod normal un asemenea mesaj, nu este necesar ca proiectul bazat pe Program să afișeze un mesaj).

Aceste cerințe se aplică Programului modificat în întregime. Dacă pot fi identificate secțiuni ale proiectului care nu sunt derivate din Program, și pot fi considerate de sine stătătoare, atunci această Licență și termenii săi nu se aplică acelor secțiuni când sunt distribuite ca proiecte separate. Când distribuiți aceleași secțiuni ca parte a unui întreg care este un proiect bazat pe Program, distribuirea întregului proiect trebuie să fie făcută în acord cu termenii acestei Licențe, ale cărei permisiuni pentru alte licențe se extind asupra întregului, și deci asupra fiecărei secțiuni în parte, indiferent de autor.

Astfel, nu este în intenția acestei secțiuni să pretindă drepturi sau să conteste drepturile dumneavaoastră asupra unui proiect efectuat în întregime de dumneavaoastră. Intenția este de a exercita dreptul de a controla distribuția proiectelor derivate sau collective bazate pe Program.

În plus, pura agregare (pe un mediu de stocare sau distribuție) cu Programul (sau cu un proiect bazat pe Program) al unui alt proiect care nu este bazat pe Program nu aduce acel proiect sub incidența acestei Licențe.

3. Puteți copia și distribui Programul (sau un proiect bazat pe el, conform Secțiunii 2) în format obiect sau executabil conform termenilor Secțiunilor 1 și 2 de mai sus, cu condiția să îndepliniți una dintre condițiile de mai jos:

- să îl oferiți însotit de codul-sursă corespunzător, în format citibil de către mașină, care trebuie să fie distribuit în termenii Secțiunilor 1 și 2 de mai sus pe un mediu de distribuție uzual transportului de software; sau
- să îl oferiți însotit de o ofertă scrisă (valabilă pentru cel puțin trei ani, pentru o taxă care să nu depășească costul fizic al efectuării distribuției sursei), de a oferi o copie completă, în format citibil de către mașină, a codului-sursă, distribuit în termenii Secțiunilor 1 și 2 de mai sus, pe un mediu de distribuție uzual transportului de software; sau
- să îl oferiți însotit de informația pe care ați primit-o referitoare la oferta de a distribui codul-sursă corespunzător. (Această alternativă este permisă numai pentru distribuiri necomerciale și doar dacă ați primit programul în format obiect sau executabil împreună cu această ofertă, în conformitate cu Subsecțiunea b de mai sus.)

Codul-sursă al unui proiect este forma preferată în care se fac modificări asupra proiectului. Pentru un proiect executabil, codul-sursă complet însamnă codul-sursă al tuturor modulelor pe care le conține, împreună cu toate fișierele asociate conținând definiții ale interfețelor și scripturile folosite pentru a controla compilarea și instalarea executabilului. Cu toate acestea, ca o excepție, nu este obligatorie distribuirea împreună cu codul-sursă a acelor componente care sunt în mod normal distribuite (în format-sursă sau binar) cu componente majore (compilator, nucleu etc.) ale sistemului de operare sub care rulează executabilul, exceptând situația în care acea componentă acompaniază executabilul.

Dacă distribuția executabilului sau codul-obiect este făcută prin oferirea permisiunii de copiere dintr-un loc dedicat, atunci oferirea permisiunii de copiere a codului-sursă din același loc este considerată distribuire a codului-sursă, chiar dacă beneficiarul nu este obligat să copieze codul-sursă împreună cu codul-obiect.

4. Nu puteți copia, modifica, sub-autoriza sau distribui Programul decât aşa cum este prevăzut în această Licență. Orice încercare de a copia, modifica, sub-autoriza sau distribui Programul în alti termeni va duce la anularea drepturilor ce vă revin conform acestei Licențe. Cu toate acestea, nu vor fi anulate drepturile celor ce au primit copii sau drepturi de la dumneavaoastră conform cu această Licență, atât timp cât rămân în conformitate cu ea.

5. Nu sunteți obligat să acceptați această Licență, deoarece nu ați semnat-o. Totuși, numai această Licență vă permite să modificați Programul sau proiectele derivate din el. Aceste acțiuni sunt interzise prin lege dacă nu acceptați această Licență. În consecință, prin modificarea sau distribuirea Programului (sau a oricărui proiect bazat pe Program), indicați în mod implicit acceptarea acestei Licențe și a tuturor termenilor și condițiilor de copiere, distribuire sau modificare a Programului sau proiectelor bazate pe el.

6. De fiecare dată când redistribuiți Programul (sau orice proiect bazat pe Program), beneficiarul primește o licență de la licențiatorul original care îi permite să copieze, distribue sau modifice Programul în aceeași termen și condiții. Nu puteți impune nici o restricție adițională asupra exercitării drepturilor pe care destinatarul le primește prin această Licență. Nu sunteți responsabil cu impunerea respectării acestei Licențe de către o terță parte.

7. În cazul în care, ca o consecință a unei decizii judecătoarești, sau pretinsă încălcare a unui patent, sau pentru orice altă cauză (nu neapărat limitată la chestiuni legate de patente), vi se impun condiții (prin hotărâre judecătoarească, înțelegere sau alte mijloace) care contravin condițiilor acestei Licențe, acest lucru nu vă permite nerespectarea condițiilor Licenței. Dacă nu puteți face în aşa fel încât să satisfaceti simultan obligațiile din această

Licență și alte obligații pertinente, atunci, ca o consecință, vă este interzisă distribuirea Programului. De exemplu, dacă o autorizație de folosire a unui patent nu vă permite redistribuirea gratuită a Programului de către oricine îl primește de la dumneavoastră, direct sau indirect, atunci singurul mod în care puteți satisface simultan aceste condiții și Licența de față este să nu distribuiți Programul în nici un fel.

Dacă vre o portiune a acestei secțiuni este invalidată sau de neaplicat în anumite circumstanțe, restul secțiunii continuă să se aplice, iar secțiunea în întregime se aplică în toate celelalte circumstanțe.

Nu este în intenția acestei secțiuni să vă determine să încălcați vreun patent sau alte pretenții de drepturi de proprietate, sau să contestați valabilitatea oricărora asemenea pretenții; această secțiune are drept unic scop protejarea integrității sistemului de distribuire de programe libere, care este implementat prin licențe publice. Multe persoane au contribuit generos la spectrul larg de programe distribuite prin acest sistem, bazându-se pe aplicarea sa consistentă; este la latitudinea autorului/donatorului să decidă dacă este dispus să distribuie programe prin orice alt sistem, și o persoană autorizată să folosească acele programe nu poate impune acea decizie.

Intenția acestei secțiuni este de a clarifica ceea ce este considerat a fi o consecință a restului acestei Licențe.

8. Dacă distribuia și/sau folosirea Programului sunt restricționate în anumite țări din cauza patentelor sau din cauza unor interfețe aflate sub incidență unor drepturi de autor restrictive, deținătorul drepturilor de autor ce plasează Programul sub această Licență poate adăuga o limitare geografică a distribuirii ce exclude acele țări, în așa fel încât distribuirea este permisă doar în (sau între) țările care nu sunt excluse. Într-un asemenea caz, Licența încorporează această limitare ca și cum ar fi scrisă în corpul acestei Licențe.

9. Free Software Foundation poate publica din când în când noi versiuni (sau versiuni revăzute) ale Licenței Publice Generale. Asemenea versiuni noi vor fi similare în spirit versiunii prezente, dar pot difera în anumite detalii, pentru a adresa noi probleme sau situații.

Fiecarei versiuni îi este asociat un număr unic. Dacă programul specifică faptul că i se aplică un număr de versiune al acestei Licențe și „orice versiune ulterioară”, aveți opțiunea de a urma termenii și condițiile acelei versiuni sau ale oricărei versiuni ulterioare publicate de Free Software Foundation. Dacă Programul nu specifică un număr de versiune, puteți alege orice versiune publicată vreodată de Free Software Foundation.

10. Dacă doriți să incorporați părți ale Programului în alte programe libere ale căror condiții de distribuție sunt diferite, cereți permisiunea autorului. Pentru programe ale căror drepturi de autor aparțin Free Software Foundation, cereți permisiunea de la Free Software Foundation; uneori facem excepții pentru aceasta. Decizia noastră va fi ghidată de cele două scopuri de a prezerva statutul liber al tuturor proiectelor derivate din programele noastre libere și de a promova distribuirea și refolosirea programelor în general.

Nici o garanție

11. DEOARECE PROGRAMUL ESTE OFERIT SUB O LICENȚĂ CE NU IMPLICĂ NICI UN COST, NU EXISTĂ NICI O GARANȚIE PENTRU PROGRAM, ÎN MĂSURA PERMISĂ DE LEGILE CE SE APPLICĂ. EXCEPTÂND SITUAȚIILE UNDE ESTE

SPECIFICAT ALTFEL ÎN SCRIS, DEȚINĂTORII DREPTURILOR DE AUTOR ȘI/SAU ALTE PĂRTI IMPLICATE OFERĂ PROGRAMUL „ÎN FORMA EXISTENTĂ” FĂRĂ NICI O GARANȚIE, EXPLICITĂ SAU IMPLICITĂ, INCLUZÂND, DAR FĂRĂ A FI LIMITATĂ LA GARANȚII IMPLICITE DE VANDABILITATE ȘI CONFORMITATE UNUI ANUMIT SCOP. VĂ ASUMAȚI ÎN ÎNTREGIME RISCUL ÎN CEEA CE PRIVEȘTE CALITATEA ȘI PERFORMANȚA ACESTUI PROGRAM. ÎN CAZUL ÎN CARE PROGRAMUL SE DOVEДЕSTE A FI DEFECT, VĂ ASUMAȚI ÎN ÎNTREGIME COSTUL TUTUROR SERVICIILOR, REPARAȚIILOR ȘI CORECȚIILOR NECESARE.

12. ÎN NICI O SITUAȚIE, EXCEPTÂND CAZURILE ÎN CARE ESTE CERUT DE LEGEA APLICABILĂ SAU CA REZULTAT AL UNEI ÎNTELEGERI SCRISE, UN DEȚINĂTOR AL DREPTURILOR DE AUTOR, SAU ORICE ALTĂ PARTE CARE POATE MODIFICA ȘI/SAU REDISTRIBUI PROGRAMUL CONFORM PERMISSIUNILOR DE MAI SUS, NU VA FI FĂCUT RĂSPUNZĂTOR PENTRU PAGUBELE DUMNEAVOASTRĂ, INCLUSIV CELE GENERALE, SPECIALE, ÎNTÂMPLĂTOARE SAU REZULTANTE, APĂRUTE DIN FOLOSIREA SAU INABILITATEA DE A FOLOSI PROGRAMUL (INCLUZÂND, DAR FĂRĂ A FI LIMITAT LA PIERDEREA SAU DETERIORAREA DATELOR, SAU PIERDERILE SUFERITE DE DUMNEAVOASTRĂ SAU TERȚE PERSOANE, SAU O INCAPACITATE A PROGRAMULUI DE A INTEROPERA CU ALTE PROGRAME), CHIAR DACĂ DEȚINĂTORUL SAU TERȚA PARTE AU FOST PREVENIȚI ASUPRA POSIBILITĂȚII UNOR ASEMEÑEA PAGUBE.

Sfârșitul termenilor și condițiilor

Cum să Aplicați Acești Termeni Noilor Dumneavoastră Programe?

Dacă dezvoilați un nou program, și doriți să fie de cea mai mare utilitate publicului, cea mai bună metodă de a realiza acest lucru este să-l faceți liber, în așa fel încât oricine să-l poată redistribui și modifica în acești termeni.

Pentru a face acest lucru, atașați începutului fiecărui fișier-sursă pentru a transmite excluderea garanției; de asemenea, fiecare fișier ar trebui să conțină cel puțin linia conținând drepturile de autor și o referire la locul unde poate fi găsită întreaga notă.

<o linie cu numele programului și o scurtă prezentare a ceea ce face.>

Copyright (C) 19yy <numele autorului>

Acest program este liber; îl puteți redistribui și/sau modifica în conformitate cu termenii Licenței Publice Generale GNU, așa cum este publicată de Free Software Foundation; și versiunea 2 a Licenței, fie (la latitudinea dumneavoastră) orice versiune ulterioară.

Acest program este distribuit cu speranța că va fi util, dar FĂRĂ NICI O GARANȚIE, fără garanție implicită de vandabilitate și conformitate unui anumit scop. Citiiți Licența Publică Generală GNU pentru detalii.

Ar trebui să fi primit o copie a Licenței Publice Generale GNU împreună cu acest program; dacă nu, scrieți Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

De asemenea, specificați modalitățile în care puteți fi contactat prin poștă normală și electronică.

Dacă programul este interactiv, modificați-l în aşa fel încât să genereze o scurtă notă atunci când pornește în mod interactiv:

Gnomovision versiunea 69, Copyright (C) 19yy numele autorului Gnomovision nu oferă ABSOLUT NICI O GARANTIE; pentru detalii tastați „*arată g*”. Acest program este liber, și sunteți bine veniți să-l redistribuiți în anumite condiții; tastați „*arată c*” pentru detalii.

Comenzile ipotetice „*arată g*” și „*arată c*” ar trebui să arate porțiunile corespunzătoare din Licență Publică Generală. Bineînțeles, comenzile pe care le veți folosi pot fi denumite altfel decât „*arată g*” și „*arată c*”; ele pot fi plasate pe meniuri, acțiuni ale mouse-ului – orice este potrivit pentru programul dumneavoastră.

De asemenea, ar trebui să obțineți un document semnat de instituția pentru care lucrăți (dacă lucrăți ca programator) sau de școală/universitate, prin care aceasta renunță la drepturile de autor pentru programul dumneavoastră. Urmează un exemplu; modificați numele:

Yoyodyne, Inc., prin aceasta renunță la toate drepturile de autor asupra programului „Gnomovision” (care face cu ochiul compilatoarelor) scris de James Hacker.

<semnatura lui Ty Coon>, 1 April 1989 Ty Coon, President of Vice

Această Licență Publică Generală nu permite incorporarea programului dumneavoastră în programe aflate sub controlul restrictiv al instituțiilor (engl. *proprietary programs*). Dacă programul dumneavoastră este o subrutină a unei biblioteci, puteți considera mai util să permiteți legarea la bibliotecă a programelor aflate sub controlul restrictiv al instituțiilor. Dacă aceasta este ceea ce doriți, folosiți Licență Publică GNU pentru Biblioteci în locul acestei Licențe.

BIBLIOGRAFIE

1. Satir Gregory, Brown Doug, *C++: The Core Language*, O'Reilly & Associates, Sebastopol, 1995
2. Qualline Steve, *Practical C++ Programming*, O'Reilly & Associates, Sebastopol, 1997
3. Bruce Eckel, *Thinking in C++*, Second Edition, Prentice Hall, New Jersey, 2000
4. Danny Kalev, *ANSI/ISO C++ Professional Programmer's Handbook*, Macmillan Computer Publishing, 1999
5. Coplien O. James, *Advanced Programming Styles and Idioms*, Addison-Wesley Publishing, Massachusetts, 1992
6. Booch Grady, *Object-Oriented Analysis and Design – With applications*, Benjamin/Cummings Publishing, Redwood City, 1994
7. Booch Grady, *Object Solutions – Managing the object-oriented project*, Addison-Wesley Publishing, Menlo Park, 1996
8. Salomie Ioan, *Tehnici Orientate pe Obiecte*, microInformatica, Cluj-Napoca, 1995
9. Fred Zlotnick, *The POSIX.1 Standard – A Programmer's Guide*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1991
10. ***, *Open source. Voices from the Open source Revolution*, O'Reilly & Associates, Inc., 1999: <http://www.oreilly.com/catalog/opensources/book/toc.html>

Resurse Web

1. <http://std.dkuug.dk/JTC1/SC22/WG14>: standardul ISO/IEC JTC1/SC22/WG14-C (ANSI C)
2. <ftp://ftp.research.att.com/dist/c++/std/WP/CD2/>: standardul ANSI C++
3. <http://www.msoe.edu/~tritt/cplang.html>: C++ Language Reference
4. <http://www.sgi.com/Technology/STL>: C++ STL Reference
5. <http://www.cs.umd.edu/users/cml/style/>: ghid privind stilul de programare în C și C++
6. <http://www.doc.ic.ac.uk/lab/cplus/c++.rules>: reguli și recomandări pentru programarea în C++
7. <http://www.programmingtutorials.com>: tutoriale de programare pentru C, C++ precum și pentru alte limbiage
8. <http://www.gnu.org/prep/standards.html>: standardele de programare GNU
9. <http://www.snippets.org>: exemple de programe C

10. <http://www.cs.bell-labs.com/who/dmr/index.html>: situl personal al lui Dennis Ritchie
11. <http://www.research.att.com/~bs>: situl personal al lui Bjarne Stroustrup
12. <http://www.cuj.com>: C/C++ Users Journal
13. <http://www.lysator.liu.se/C/>: numeroase legături către situri referitoare la C
14. <http://www.linux.org>: situl oficial al sistemului de operare Linux
15. <http://www.gnu.org>: situl fundației GNU
16. <http://www.gnu.org/software/automake>: situl Automake
17. <http://www.cvshome.org>: situl CVS
18. <http://www.gnu.org/software/ddd>: situl DDD
19. <http://docpp.sourceforge.net>: situl DOC++
20. <http://www.gnu.org/software/gcc/gcc.html>: situl GCC
21. <http://glade.gnome.org>: situl Glade
22. <http://www.kdevelop.org>: situl KDevelop
23. <http://www.splint.org>: situl SPLint
24. <http://sourceforge.net>: cel mai puternic sit dedicat dezvoltării de programe *open source*
25. <http://freshmeat.net>: cea mai mare bază de date de programe *open source*
26. <http://www.ibiblio.org>: cea mai mare bibliotecă de informații publice

GLOSAR DE TERMENI

Iată semnificațiile termenilor importanți folosiți în această lucrare:

Asamblare

Compilare din limbaj de asamblare în cod mașină (engl. *assembly*).

ANSI

American National Standards Institute, instituție americană care se ocupă de crearea standardelor.

Bibliotecă

O colecție de module-obiect grupate într-un fișier-arhivă. Conține codul unor funcții și definițiile unor variabile utile (engl. *library*).

Buffer

Zonă tampon de memorie, utilizată în general pentru stocarea unor date temporare.

Bug

Eroare de proiectare, de programare etc., într-un program.

Compilare

Transformarea unui program dintr-un limbaj în cod mașină (engl. *compilation*).

Declarație

O construcție care anunță existența unui obiect (funcție sau variabilă), fără însă a-l defini (construi sau aloca spațiu de memorie) (engl. *declaration*).

Definiție

O structură care construiește sau alocă spațiu de memorie pentru un obiect (engl. *definition*).

Fișier-antet

Un fișier care grupează declarațiile obiectelor exportate de unul sau mai multe module (engl. *header file*).

Link-editare

Combinarea mai multor module-obiect într-unul singur (fișier executabil sau bibliotecă) (engl. *linking*).

HTML (HyperText Markup Language)

Lingua franca a spațiului WWW, aplicație majoră a SGML (Standard Generalized Markup Language).

Mașină

Exprimare echivalentă la calculator sau sistem (engl. *host*).

Modul

O parte dintr-un program, constând dintr-un singur fișier, care grupează de obicei obiecte (variabile, funcții etc.) folosite împreună (engl. *module*).

Platformă

Desemnează o combinație specifică de hardware și software (tip sau versiune de sistem de operare – Linux, MacOS, Windows – și/sau compilator ori limbaj de programare – GNU C/C++, Perl, Java etc.) (engl. *platform*).

Preprocesare

O fază anterioară compilării, care prelucrează textul programului-sursă C, obținând o nouă sursă C, rezolvând macrourile din cadrul acesteia (engl. *preprocessing*).

Program executabil

Un fișier (exprimat în limbaj mașină) care poate fi direct executat de către sistemul de operare (engl. *executable file, binary*).

Script

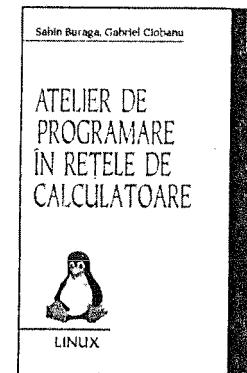
Un limbaj de tip *script* este destinat să prelucreze, să automatizeze și să integreze facilitățile oferite de un anumit sistem. Pentru sistemul de operare Linux/UNIX este disponibilă o pleiadă de interprotoare de scripturi: bash, Perl, Tcl/Tk, Python, M4 etc.

UNIX

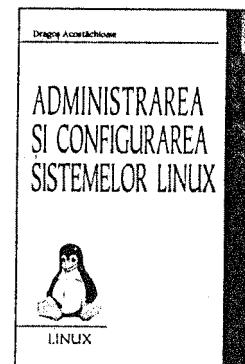
Sistem de operare interactiv apărut în anul 1968, leagănul protocolelor de comunicație în rețea și a celor mai multe dintre conceptele teoriei și practicii sistemelor de operare. Părinții sistemului UNIX sunt considerați Ken Thompson și Dennis Ritchie (acesta din urmă este co-inventatorul limbajului C, împreună cu Brian Kernighan). De-a lungul anilor, sistemul a suferit mutații importante și s-a diversificat în diverse subtipuri (implementări) precum AIX, BSD, HP-UX, Linux, System V sau Xenix. Este utilizat de cei mai mulți dintre furnizorii de servicii Internet și în mediul academic. Una dintre implementările de succes este Linux, conceput de Linus Torvalds, fiind disponibil gratuit pe Internet, din 1991. Linux se află în plină dezvoltare și este disponibil în mai multe „distribuții” dintre care se pot enumera Red Hat, Mandrake, SuSE ori Slackware.

La Editura POLIROM

au apărut:



- Unix și Linux
- Gestionația fișierelor
- Procese
- Semnale
- Comunicarea între procese. *Pipe*-uri
- Duplicarea descripților. Redirectări
- Interfață *socket*
- Modelul client/server TCP
- Modelul client/server UDP
- Multiplexarea intrărilor/ieșirilor
- RPC – Apelul procedurilor la distanță
- Utilizarea bibliotecii MySQL
- Biblioteca ncurses
- Mediul Glade



- Instalarea Linux-ului • Gestionarea pachetelor de programe
- Administrarea utilizatorilor • Sistemul X Windows • Nucleul Linux
- Legătura la rețea • Configurarea dispozitivelor hardware
- Realizarea de copii de siguranță
- Tipărire la imprimantă • Sisteme de baze de date • Sisteme de fișiere în rețea
- Accesarea de la distanță a sistemului • Sistemul numelor de domenii
- Linux ca server Web • Interacțiunea cu sisteme Windows • Sistemul de poștă electronică
- Execuția de proceduri automate • Autentificarea utilizatorilor
- Securizarea sistemului • Configurarea unui firewall



- Protocolul HTTP (HyperText Transfer Protocol)
- Limbajele de marcare și aplicațiile lor • Standardul CGI (Common Gateway Interface)
- Cookie-urile
- Programarea CGI în bash
- Scripturi CGI în bash
- Limbajul și modulele Perl
- Programarea CGI în Perl • Perl și bazele de date relaționale
- Prelucrarea documentelor XML

www.polirom.ro

Coperta: Ionuț Broștianu

Redactare: Andrei Dimitriu

Tehnoredactare: Irina Lăcătușu

Bun de tipar: octombrie 2002. Apărut: 2002
Editura Polirom, B-dul Copou nr. 4 • P.O. Box 266, 6600, Iași

Tel. & Fax (0232) 21.41.00; (032) 21.41.11;

(0232) 21.74.40 (difuzare); E-mail: office@polirom.ro

București, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33 • P.O. BOX 1-728, 70700
Tel.: (021) 313.89.78, E-mail: polirom@dnt.ro

Tiparul executat la SC LUMINA TIPO s.r.l.
str. Luigi Galvani nr. 20 bis, sect. 2, București
Tel./Fax: 211.32.60, 212.29.27, E-mail: lumina-lex@fx.ro
