

7

TEHNICI SPECIALE ÎN PASCAL

Pentru practica proiectării, punerii la punct și dezvoltării programelor, mediul integrat Turbo Pascal oferă programatorului o serie de tehnici de lucru speciale și instrumente puternice care să îl asiste în acest proces complex.

7.1 Tehnici de reacoperire în construirea programelor complexe

Programele complexe, de dimensiuni mari (uneori peste spațiul de memorie disponibil), pot fi concepute modularizat, într-o structură arborescentă, în care se disting un *modul monitor* și mai multe *module operaționale*. Modulul monitor este rezident în memorie pe parcursul execuției programului și apelează succesiv modulele operaționale care pot sau nu să fie prezente simultan în memoria principală.

Reacoperirea este tehnica prin care două sau mai multe *unități* se încarcă la aceeași adresă de memorie, evident nu simultan. Construirea structurilor de reacoperire este posibilă numai pentru modulele constituite ca unități ale utilizatorului (unitățile standard nu pot face obiectul reacoperirii).

În general, un program realizat cu unități poate fi construit ca program executabil în două forme: cu structură liniară, respectiv cu structură arborescentă. La programele cu *structură liniară*, spațiul de memorie necesar este dat de suma lungimilor tuturor unităților, la care se adaugă lungimea programului principal. Programul, cu toate componentele sale, va fi încărcat complet în memorie și va rămâne rezident pe tot parcursul execuției. La programele cu *structură arborescentă* se poate realiza o *structură cu reacoperire (overlay)*, în care se disting două tipuri de componente: componenta rezidentă (rădăcina arborelui), care rămâne în memorie pe tot parcursul execuției programului; componentele reacoperibile (ramurile arborelui), care folosesc pe rând aceeași arie de memorie, în care se încarcă succesiv pe măsura referirii lor, cu eventuala suprascriere (reacoperire) a celor încărcate anterior. Componentele reacoperibile sunt formate din unități ale utilizatorului.

Pe exemplul din figura 7.1, presupunând că partea rezidentă este formată din programul principal și unitatea C, iar celelalte unități se reacoperă între ele, necesarul

de memorie internă este de 55 Ko, dat de suma lungimilor componentei rezidente și a celei mai mari ramuri.

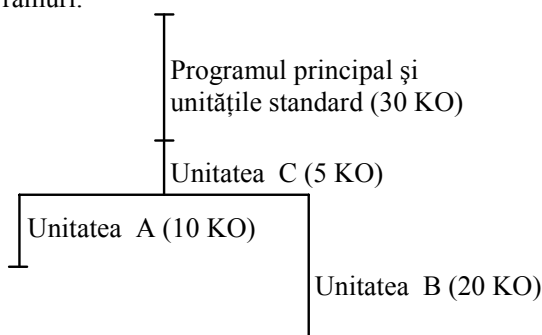


Fig. 7.1 Exemplu de structură cu reacoperire

Pentru construirea și gestiunea structurilor cu reacoperire, mediul de dezvoltare Turbo Pascal are o componentă specializată (**Overlay Manager**), ale cărei proceduri și funcții sunt reunite în unitatea **Overlay**. Folosirea acestei componente presupune:

a) Pe planul declarațiilor:

- Includerea unității Overlay în clauza USES a programului principal, cu obligația de a fi prima referită.
- Introducerea directivelor **{ \$O nume_unitate }**, după clauza USES. Fiecare directivă nominalizează unitatea care va fi inclusă în structura de reacoperire (se generează instrucțiuni care vor include unitatea în fișierul Overlay).
- Fiecare unitate care va fi inclusă în reacoperire se va compila cu directiva **{ \$O+ }**. Prezența acesteia nu realizează includerea unității în reacoperire, ci doar semnalează compilatorului să asigure o structură a codului obiect, care să permită eventuala sa includere într-o astfel de structură. În concluzie, o aceeași versiune de unitate, compilată cu **{ \$O+ }**, va putea fi folosită atât în structuri cu reacoperire, cât și în structuri liniare.

• Pentru asigurarea generării tuturor apelurilor după modelul FAR (cu adresă de segment și cu deplasare), se va folosi directiva de compilare **{ \$F+ }**, atât în programul principal, cât și în unitățile care urmează a fi incluse în reacoperire.

b) Pe planul instrucțiunilor executabile, folosirea tehnicii de reacoperire presupune apelul unor proceduri și funcții ale unității Overlay, în cadrul programului principal (în componenta rezidentă).

• În primul rând este necesară inițierea programului de gestiune a reacoperirii segmentelor, înaintea execuției altor instrucțiuni (inclusiv a celor de inițializare din unități). Acest lucru se realizează cu procedura *OvrInit*, definită astfel: **OvrInit(nume_fisier:STRING)**, unde *nume_fisier* este numele fișierului, cu extensia implicită .OVR, în care sistemul încarcă unitățile incluse în structura de reacoperire.

Pentru exemplul din figura 7.1, pot fi concepute, principal, structurile din figura 7.2. În cazul în care una dintre unitățile apelate conține parte de inițializare, datorită restricției ca procedura *OvrInit* să se execute prima, corelat cu faptul că partea de inițializare a unei unități se lansează în execuție cu prioritate, va trebui construită o unitate separată, care să conțină în partea de inițializare apelul procedurii *OvrInit* și care să fie invocată prima în clauza USES a programului principal (această unitate va conține USES Overlay).

{\$O+,F+}	{\$O+,F+}	{\$O+,F+} (* Optional *)
UNIT A;	UNIT B;	UNIT C;
Interface	Interface	Interface
.....
Implementation	Implementation	Implementation
.....
END.	END.	END.

a) Unitățile

```

{$O+,F+}
PROGRAM PP; {Apelatorul}
USES Overlay, A,B,C
{$O A}
{$O B}
.....
BEGIN
  OvrInit('ALFA.OVR');
.....
END.
```

b) Programul

Fig. 7.2 Exemplu de principiu de realizare a unei structuri cu reacoperire

Pe parcursul execuției programului, folosirea tehnicii de reacoperire se traduce în încărcarea succesivă a unităților, pe măsura referirii lor, într-un *buffer al structurii de reacoperire*, căruia i se asociază memorie la începutul zonei *heap*. La apelul procedurii *OvrInit*, mărimea buffer-ului de reacoperire este setată implicit la mărimea celei mai mari unități incluse în reacoperire, la care se adaugă o zonă suplimentară pentru informații de interfață. Dimensiunea curentă a buffer-ului de reacoperire în număr de octeți poate fi determinată, în orice moment pe parcursul execuției programului, prin apelul funcției *OvrGetBuf*, definită astfel: **OvrGetBuf:Longint**.

Referitor la mecanismul de folosire a buffer-ului, trebuie menționat că o unitate deja încărcată va fi ștearsă (suprascisă), numai dacă nu există suficient spațiu liber în restul buffer-ului pentru a încărca următoarea unitate. Dacă există spațiu

suficient, noua unitate va fi încărcată, evitând reluarea încărcării unora dintre unități de fiecare dată când sunt referite.

Dacă utilizatorul dorește să mărească dimensiunea buffer-ului, tocmai pentru a rezolva asemenea probleme, se poate folosi procedura *OvrSetBuf*, definită astfel: **OvrSetBuf(bufsize:Longint)**, unde *bufsize* precizează noua mărime a buffer-ului, care trebuie să fie mai mare decât cea implicită, fără a depăși valoarea returnată de funcția *MemAvail*: $\text{bufsize} \leq \text{MemAvail} + \text{OvrGetBuf}$ (*MemAvail* returnează volumul total de memorie nealocată în *heap*). Procedura *OvrClearBuf* eliberează buffer-ul, pentru utilizări în alte scopuri.

Pentru a elimina repetarea operațiilor de I/E necesare încărcării unităților din fișierul asociat structurii de reacoperire, este posibilă folosirea memoriei EMS (**E**xpanded **M**emory **S**ystem), prin apelul procedurii *OvrInitEMS* (fără parametri). În acest caz, se va asigura încărcarea fișierului în memoria EMS (dacă aceasta are spațiu suficient), preluarea unităților în buffer-ul structurii de reacoperire realizându-se foarte rapid, direct din această memorie, fără operații de I/E. Dacă nu există spațiu suficient, procedura este inefectivă.

Execuția procedurilor și funcțiilor de gestiune a structurilor de reacoperire se poate încheia cu succes sau cu eroare, situație care poate fi determinată pe baza folosirii variabilei predefinite **OvrResult**. În unitatea Overlay sunt declarate mai multe constante simbolice asociate unor valori ale variabilei *OvrResult* (anexa 5).

7.2 Tehnica gestiunii proceselor

În cele ce urmează, prin *proces* se desemnează un program executabil. În cazul unor sisteme de programe, utilizatorul are de ales între două variante de lansare în lucru a programelor componente:

- *lansare independentă*, caz în care intercondiționările dintre programe se rezolvă prin ordinea de execuție a acestora. Întrucât procesele nu pot comunica între ele, eventuale informații privind modul lor de încheiere vor fi preluate și prelucrate de sistemul de operare.

- *lansare din interiorul altor programe*, stabilindu-se raporturi de tipul proces părinte - proces fiu. Pentru sistemele de programe se poate constitui, la limită, un singur proces părinte, cu rol de monitor, care lansează succesiv în execuție procesele fiu. Programul monitor poate prelucra informațiile referitoare la modul de încheiere a fiecărui proces fiu și poate lua decizii privind modul de continuare. Unitatea Dos oferă posibilitatea implementării acestei soluții prin intermediul unor proceduri și funcții specifice.

- *Lansarea în execuție a unui proces fiu* se poate realiza cu procedura al cărei antet este: **Exec(specif,parametru:STRING)**. Procedura încarcă în memorie și lansează în execuție programul al cărui format executabil se găsește în fișierul cu

specificatorul *specif*. Argumentul *parametru* conține șirul de caractere corespunzând eventualilor parametri dați prin linia de comandă la lansarea acestui program. Întrucât este necesară asigurarea de spațiu în memorie și pentru procesul fiu, în programul corespunzând procesului părinte trebuie să se reducă dimensiunea zonei *heap*, folosind directiva \$M.

- *Salvarea vectorilor de întrerupere* se realizează prin procedura *SwapVectors*. Procedura este destinată asigurării independenței dintre procesul părinte și un proces fiu în folosirea unor rutine de întrerupere instalate de către celălalt proces. Se apelează înainte și după fiecare apel al procedurii *Exec* și schimbă între ei vectorii de întrerupere curenți cu pointerii **SaveIntxx** din unitatea *System*.

- *Preluarea codului de retur al unui proces fiu* se poate realiza prin funcția: **DosExitCode:WORD**. Codul de retur al procesului fiu poate fi poziționat prin parametrul folosit la apelul rutinei *Halt*.

Exemplu:

7.1. Se consideră un program director (*Monitor*) care lansează în execuție trei programe, în cadrul unei aplicații multifuncționale. Cele trei programe sunt memorate în fișierele *Fis1.EXE*, *Fis2.EXE*, *Fis3.EXE*.

```
PROGRAM Monitor;
{$M 2048,0,0}
USES CRT;
VAR
  c:CHAR;
BEGIN
  REPEAT
    ClrScr;
    WriteLn(' Functiile realizate de program sunt:');
    WriteLn(' ':5,'1 - Semnificatie functia 1');
    WriteLn(' ':5,'2 - Semnificatie functia 2');
    WriteLn(' ':5,'3 - Semnificatie functia 3');
    WriteLn(' ':5,'4 - STOP');
    Write ('Tastati functia dorita [1-4] : '); Readln( c );
    CASE c OF
      '1': Exec('Fis1','');
      '2': Exec('Fis2','');
      '3': Exec('Fis3','');
      '4': WriteLn('<< Program terminat>>')
    ELSE
      WriteLn('***Cod eronat de functie***')
    END
  UNTIL c='4'
END.
```

7.3 Lansarea programelor în execuție prin linie de comandă

În general, lansarea în execuție a programelor Turbo Pascal este posibilă în două moduri: din interiorul mediului de programare, prin una din comenzile meniului **Run**; la nivelul sistemului de operare, printr-o linie de comandă. Prima variantă este utilă pe parcursul procesului de punere la punct a programelor, iar a doua este preponderentă în exploatarea curentă a unui produs-program.

Pentru lansarea prin linie de comandă, utilizatorul are posibilitatea includerii, după numele fișierului .EXE, a unor parametri destinați programului. Linia de comandă are formatul general:

...>nume parametru [parametru]...

Parametrii sunt șiruri de caractere, separate prin spații sau tab-uri, care sunt accesibili din programele Turbo Pascal prin două funcții aparținând unității System. Numărul de parametri din linia de comandă poate fi determinat folosind funcția *ParamCount*, definită astfel: **ParamCount:WORD**. Accesul la un anumit parametru este posibil prin funcția *ParamStr*, al cărei antet are forma: **ParamStr(n:WORD):STRING**. Funcția returnează un șir de caractere care conține al n-lea parametru de pe linia de comandă. Este posibil și apelul de forma *ParamStr(0)*, caz în care șirul returnat conține calea și numele fișierului .EXE asociat programului.

Exemplu:

7.2. Se consideră programul *ExDir*, care poate prelua din linia de comandă directorul în care se găsesc fișierele aplicației, transformându-l în director curent. Se asigură salvarea directorului în uz de la momentul lansării, respectiv restaurarea lui la încheierea execuției programului. Dacă noul director nu există, este prevăzută posibilitatea creării și activării lui ca director curent. În cadrul programului, procedura *OpDir* asigură următoarele operații: determinarea directorului curent ('D'), schimbarea directorului curent ('S'), crearea unui nou director ('C'). Procedura returnează o variabilă de eroare (**rez**) când operațiile asupra directoarelor nu se pot executa.

```
PROGRAM ExDir;
VAR
  r : INTEGER;  c : CHAR;
  dinc,dnou: STRING;
PROCEDURE OpDir(op:CHAR; VAR d:STRING; VAR rez:INTEGER);
BEGIN
  CASE UpCase(op) OF
    'D': GetDir(0,d);
    'S': BEGIN  {$I-} ChDir(d);  {$I+}  rez:=IOResult END;
    'C': BEGIN  {$I-} MkDir(d);  {$I+}  rez:=IOResult END
  END
END;
PROCEDURE DetDir;
```

```

VAR
    dir: STRING;
BEGIN
    OpDir('d',dir,r);
    WriteLn('Director curent: ',dir)
END;
BEGIN
    WriteLn('Program in executie curenta: ',ParamStr(0));
    DetDir;
    IF ParamCount <> 0 THEN
        BEGIN
            dnou:=ParamStr(1);
            OpDir('d',dinc,r);
            OpDir('s',dnou,r);
            IF r <> 0 THEN
                BEGIN
                    WriteLn('Director inexistent: ',ParamStr(1));
                    Write('Se creeaza ca director nou?(Y/N): ');
                    ReadLn(c);
                    IF UpCase(c) = 'Y' THEN
                        BEGIN
                            OpDir('c',dnou,r);
                            IF r = 0 THEN
                                OpDir('s',dnou,r)
                            ELSE
                                BEGIN
                                    WriteLn('Eroare la creare director. Executie
intrerupta!' );
                                    RunError(r)
                                END
                            END
                        END
                    END;
                END;
            END;
            DetDir
        END; {restul programului se scrie normal}
    IF ParamCount <> 0 THEN OpDir('s',dinc,r);
    DetDir
END.

```

7.4 Tehnica tratării întreruperilor

Microprocesorul gestionează un sistem de întreruperi. Datorită complexității proceselor ce se pot desfășura în cadrul sistemului, unitatea centrală de prelucrare (UCP) nu poate asigura simultan un control complet, fiind focalizată la un moment dat pe o singură activitate. Pentru a suplini această lipsă, calculatorul este proiectat astfel încât, la apariția oricărui eveniment ce presupune o anumită acțiune, să se genereze un semnal de întrerupere prin care este informată UCP. Aceasta suspendă temporar alte activități pentru a trata evenimentul apărut, transferând controlul la o rutină de tratare a întreruperii, care poate provoca afișarea unui mesaj de avertizare, reluarea sau încheierea execuției programului (normală, cu eroare sau forțată) etc. La încheierea execuției rutinei, UCP revine la activitatea inițială.

Microcalculatoarele IBM PC și compatibile acceptă 256 de întreruperi, cu coduri din intervalul 0..255 (în hexa, \$00..\$FF). În literatura de specialitate, întreruperile se regăsesc clasificate în diverse moduri: • hardware, software; • ale

microprocesorului, hardware, software, DOS, Basic, de adresă, de uz general; • ale microprocesorului, BIOS, ale sistemului de operare (din care fac parte și cele referitoare la funcțiile DOS) etc.

Pentru asigurarea accesului la rutinele de întrerupere, prima zonă de 1 Ko din memoria internă conține o tabelă de pointeri către fiecare din rutinele disponibile, dimensionată pentru a putea memora toate cele 256 de adrese teoretic posibile și numită *tabela vectorilor de întrerupere*. Considerând o rutină cu codul *i* din intervalul 0..255, pointerul asociat ei în tabelă se află la adresa **4*i**.

Utilizatorul poate determina adresa unei rutine de întrerupere prin procedura *GetIntVec* definită în unitatea DOS astfel: **GetIntVec(int:BYTE;VAR adrint:POINTER)**, unde *int* reprezintă codul întreruperii precizat în apelator, iar în *adrint* se returnează adresa rutinei de întrerupere corespunzătoare.

Pentru ilustrarea utilizării acestei proceduri, se prezintă programul *TabVectInt* care afișează, în mai multe ecrane succesive, întreaga tabelă a vectorilor de întrerupere, cu precizarea codului și a adresei rutinei asociate fiecărei întreruperi, cu componentele de segment și deplasare, exprimate în hexazecimal (conversia se realizează de către funcțiile *HexOctet* și *HexCuvint* incluse în unitatea *Hex*).

```
PROGRAM TabVectInt;
USES Dos,Crt,Hex;
VAR
  i : BYTE;
  adrint: POINTER;
PROCEDURE Defilare;
VAR
  car: CHAR;
BEGIN
  GotoXY(1,25);
  Write('Press any key ...');
  car:=ReadKey;
  ClrScr
END;
BEGIN
  ClrScr;
  FOR i:=0 TO 255 DO
    BEGIN
      GetIntVec(i,adrint);
      IF (i+1) MOD 24 = 0 THEN Defilare;

      WriteLn(HexOctet(i):5,HexCuvant(Seg(adrint^)):5,
              HexCuvant(Ofs(adrint^)):5)
    END;
  Defilare
END.

UNIT Hex;
INTERFACE
FUNCTION HexOctet(nroct:BYTE):STRING;
FUNCTION HexCuvint(nrcuv:WORD):STRING;
IMPLEMENTATION
FUNCTION CifraHex(cifra:BYTE):CHAR;
```



```

BEGIN
  IF cifra < 10 THEN CifraHex:=Chr(48+cifra)
                    ELSE CifraHex:=Chr(55+cifra)
END;
FUNCTION HexOctet;
BEGIN
  HexOctet:=CifraHex(nroct DIV 16) + CifraHex(nroct
MOD 16);
END;
FUNCTION HexCuvant;
BEGIN  HexCuvint:=HexOctet(Hi(nrcuv))+HexOctet(Lo(nrcuv))
END
END.

```

Apelul rutinei de întrerupere se realizează prin procedura *Intr*, definită în unitatea System astfel: **Intr(ni:BYTE; VAR reg:Register)**, unde *ni* este codul întreruperii, iar *reg* este o variabilă care specifică, într-o formă standard, diverși parametri. Tipul *Registers*, definit în unitatea Dos (anexa 1), permite accesul la registrele unității centrale. Înainte de apel, trebuie pregătite anumite registre, în funcție de parametrii solicitați de întrerupere. Astfel, în semiregistrul Reg.AH trebuie încărcat codul funcției, în Reg.AL codul subfuncției (când există) etc.

Principalele caracteristici ale întreruperilor (folosindu-se ultima clasificare prezentată anterior) sunt:

- întreruperile microprocesorului sunt generate de el însuși: 0 corespunde situației de împărțire la zero, 2 corespunde erorii de paritate în memorie, 1 și 3 sunt folosite în depanarea programelor; 5 provoacă tipărirea conținutului ecranului pe imprimantă etc. Alte întreruperi sunt generate de diverse componente hardware din configurația sistemului.

- rutinele **BIOS** sunt realizate de către producătorul echipamentului și memorate în ROM. La începutul unei sesiuni de lucru cu calculatorul, adresele din ROM ale rutinelor BIOS sunt încărcate în tabela vectorilor de întrerupere. Când un program cheamă o rutină BIOS, Turbo Pascal preia adresa acesteia din tabelă și o execută. Exemple de întreruperi BIOS: \$09 - apăsarea unei taste; \$10 - servicii video; \$12 - determinarea dimensiunii memoriei existente în sistem; \$13 - gestiunea discurilor; \$15 - determinarea unor caracteristici ale memoriei etc.

În programele *Int15* și *Int12* sunt prezentate exemple de activare a întreruperilor \$15 și \$12. Pentru întreruperea \$15, se apelează funcția \$88, care returnează în registrul AX dimensiunea memoriei extinse exprimată în Ko. Întreruperea \$12 nu folosește registrul AH.

```

PROGRAM Int15;
USES Dos;
FUNCTION
ExtendedMemory:WORD;
VAR
  reg: Registers;
BEGIN
  reg.AH:=$88;
  Intr($15,Reg);

ExtendedMemory:=reg.AX
END;
BEGIN
  WriteLn('Memorie
extinsa= ',
ExtendedMemory, ' Ko')
END.

```

```

PROGRAM Int12;
USES Dos;
FUNCTION MemorySize: WORD;
VAR
  reg: Registers;
BEGIN
  Intr($15,Reg);
  MemorySize:=reg.AX
END;
BEGIN
  WriteLn('Memorie interna= ',
MemorySize, ' Ko')
END.

```

• rutinele de întrerupere ale sistemului de operare sunt memorate pe disc și sunt încărcate în memoria internă la începutul fiecărei sesiuni de lucru. Acestea corespund întreruperilor software. Exemple de întreruperi SO: \$20 - terminare program; \$21 - funcții DOS; \$22 - furnizarea adresei de terminare; \$24 - rutina de tratare a erorilor critice.

Una dintre cele mai utilizate întreruperi este \$21 (rutine DOS). Adresele rutinelor, corespunzând diferitelor servicii DOS, sunt memorate în tabela vectorilor de întrerupere, în pozițiile neocupate de rutinele BIOS. Exemple de funcții DOS: \$00 - terminare program; \$01 - citire cu ecou de la tastatură; \$08 - citire fără ecou de la tastatură; \$25 - setarea vectorului de întrerupere; \$2B - setarea datei din sistem; \$2C - citirea datei din sistem; \$2D setarea orei din sistem etc.

Apelul unei rutine DOS se poate realiza cu `Intr($21,Reg)` sau cu o procedură specială definită în unitatea DOS astfel: **MsDos(VAR Reg;Registers)**. Întrucât toate rutinele corespund unor funcții, este obligatorie încărcarea prealabilă a codului acestora în semiregistrul AH, înaintea apelului procedurii *MsDos* sau *Intr*.

Aproape toate cele 80 de funcții DOS se regăsesc în cele peste 200 de proceduri și funcții ale unităților standard din Turbo Pascal. Analizând avantajul posibilităților ce se deschid programatorului care folosește Turbo Pascal, la extreme se constată următoarele variante de abordare: folosirea exclusivă a procedurilor și funcțiilor din unitățile standard ale mediului Turbo Pascal (care apelează, la rândul lor, funcțiile DOS); folosirea directă a rutinelor de întrerupere asociate funcțiilor DOS prin apeluri adecvate ale procedurilor *MsDos* (sau *Intr*). Prima soluție este mai simplă de utilizat în scrierea programului sursă, dar cea de-a doua conduce la creșterea vitezei de execuție a programelor. În concluzie, se poate reține recomandarea ca a doua soluție, care presupune cunoașterea a o serie de detalii tehnice, să fie folosită doar atunci când este importantă asigurarea unui timp de răspuns cât mai scurt. Evident, la limită, este posibilă "ocolirea" integrală a procedurilor și funcțiilor din bibliotecile Turbo Pascal, dar nici una din variante nu poate fi absolutizată.

Pentru analiza comparată a celor două variante se ilustrează folosirea lor pe exemplul funcțiilor DOS de acces la data din sistem. Accesul din Turbo Pascal se poate realiza cu procedurile *GetDate*, pentru preluarea datei curente, respectiv *SetDate*, pentru modificarea acesteia. Procedura *GetDate* are următorii parametri, pentru care se precizează și limitele domeniului de valori admis: anul (1980..2099), luna (1..12), ziua în cadrul lunii (1..31), ziua în cadrul săptămânii (0..6, cu duminică=0, luni=1, ..., sâmbătă=6). Procedura *SetDate* folosește numai primii trei parametri.

În programul *Calendar_1* se ilustrează apelul acestor proceduri pentru determinarea calendarului unei luni oarecare, din intervalul de ani acceptat (1980..2099).

```
PROGRAM Calendar_1;  
USES Dos,Crt;  
VAR  
    ac,lc,zc,zsc: WORD;
```

```

    an,ln,zn,zsn: WORD;
    i,nrz: BYTE;
BEGIN
    REPEAT
        Write('Luna, an: '); ReadLn(ln,an);
    UNTIL (an > 1979) AND (an < 2100) AND (ln IN [1..12]);
    zn:=1;
    GetDate(ac,lc,zc,zsc);
    SetDate(an,ln,zn);
    GetDate(an,ln,zn,zsn);
    SetDate(ac,lc,zc);
    CASE ln OF
        1,3,5,7,8,10,12: nrz:=31;
        4,6,9,11       : nrz:=30;
        2               : IF an MOD 4 = 0 THEN nrz:=29 ELSE nrz:=28
    END;
    ClrScr;
    WriteLn('      ',ln:2,' - ',an:4,#13#10);
    WriteLn(' L M M J V S D');
    WriteLn(' -----');
    IF zsn = 0 THEN zsn:=7;
    FOR i:=1 TO nrz+zsn-1 DO
        BEGIN
            IF i <= zsn-1 THEN Write(' ')
                           ELSE Write(i-zsn+1:3);
            IF i MOD 7 = 0 THEN WriteLn
        END;
        WriteLn;
        WriteLn(' -----')
    END.

```

Primul apel al procedurii *GetDate* asigură salvarea datei curente (în variabilele **ac**, **lc**, **zc**, respectiv **zsc**), care se reface prin al doilea apel al procedurii *SetDate*. Algoritmul folosit se bazează pe faptul că, intern, se asigură determinarea zilei din săptămână asociată oricărei date din intervalul de ani acceptat. Astfel, după solicitarea lunii (**ln**) și a anului (**an**) pentru care se cere afișarea calendarului, se va determina ziua din săptămână (**zsn**) pentru prima zi a lunii, urmată de toate celelalte.

Rutinele de întrerupere DOS corespunzătoare celor două proceduri au codurile **\$2A**, respectiv **\$2B**. După apelul rutinei **\$2A**, rezultatele sunt furnizate în următoarele (semi)registre: anul în CX, luna în DH, ziua în cadrul lunii în DL, iar ziua în cadrul săptămânii în AL. Corespunzător, apelul rutinei **\$2B** trebuie precedat de încărcarea valorilor asociate noii date din sistem în CX, DH și DL.

Folosirea acestei soluții presupune: adăugarea unei declarații de forma: reg1,reg2: Registers; înlocuirea celor patru instrucțiuni de apel al procedurilor *GetDate* și *SetDate* cu următoarea secvență (restul programului rămânând nemodificat):

```

reg1.AH:=$2A;
MsDos(reg1);
ac:=reg1.CX;
lc:=reg1.DH;
zc:=reg1.DL;
zsc:=reg1.AL;
reg2.AH:=$2B;
reg2.CX:=an;
reg2.DH:=ln;
reg2.DL:=zn;
MsDos(reg2);

```

```

reg1.AH:=$2A;
MsDos(reg1);
zsn:=reg1.AL;
reg2.AH:=$2B;
reg2.CX:=ac;
reg2.DH:=lc;
reg2.DL:=zc;
MsDos(reg2);

```

7.5 Tehnici de lucru la încheierea execuției programelor

Încheierea execuției unui program este un eveniment important, care trebuie tratat de programator cu toată atenția. Turbo Pascal asigură utilizatorului posibilitatea implicării directe, atât în declanșarea acestui eveniment, cât și în realizarea unor prelucrări suplimentare asociate lui.

Terminarea unui program poate fi de mai multe categorii: normală, cu eroare la execuție, forțată. În primele două cazuri, decizia de terminare a execuției aparține sistemului, iar în ultimul terminarea se execută înaintea atingerii sfârșitului logic al programului, la solicitarea expresă a utilizatorului.

Variabilele **ExitCode** și **ErrorAddr**, definite în unitatea System, conțin informații privind modul de încheiere a execuției. La încheierea cu eroare la execuție, *ExitCode* va conține numărul erorii, iar *ErrorAddr* punctează pe codul asociat instrucțiunii care a generat această eroare. La lansarea în execuție a unui program, cele două variabile au valorile inițiale **0**, respectiv **nil**. La încheiere normală, situația celor două variabile este *ExitCode*=0 și *ErrorAddr*=nil. Mediul Turbo Pascal determină dacă este necesară afișarea unui mesaj de eroare la execuție, pe baza valorii variabilei pointer *ErrorAddr*. Mesajul de eroare se afișează numai dacă *ErrorAddr* ≠ nil.

Teoretic, încheierea forțată a execuției, înaintea atingerii sfârșitului logic al programului, contravine principiilor programării structurate. Totuși, practica programării conduce la situații când o asemenea încheiere devine necesară, dar se recomandă limitarea folosirii ei numai la cazurile când nu sunt posibile și alte soluții.

- Procedura de încheiere a execuției blocului curent are antetul de forma: **Exit**. În cazul subprogramelor, apelul procedurii provoacă revenirea în apelator (blocul din care a fost apelat subprogramul), iar pentru programe determină încheierea execuției.

- Procedura de încheiere a execuției cu un cod de retur precizat de utilizator este definită astfel: **Halt[(ExitCode:WORD)]**. La apelul procedurii cu parametru, *ExitCode* va primi valoarea transferată la execuția apelului, iar în cazul apelului fără parametru rămâne pe valoarea 0; în ambele cazuri, *ErrorAddr* rămâne cu valoarea nil. Prin apelul procedurii (care poate fi plasat chiar la sfârșitul logic al programului, fără a încălca principiile programării structurate), se poate face distincția între mai multe moduri de terminare normală a unui program.

Exemplu:

7.3. Fiind dat un vector de mari dimensiuni, memorat într-un fișier binar, să se realizeze programul de ventilare a fișierului cu obținerea a două fișiere binare, cu valorile pozitive, respectiv negative ale fișierului (valorile nule vor fi neglijate). Programul *Ventilare* realizează încheierea execuției cu apelul procedurii *Halt*, care,

prin poziționarea codului de retur, permite distincția între diversele moduri de încheiere a execuției: ambele fișiere nevide, ambele fișiere vide, respectiv unul sau altul dintre fișierele de ieșire vid.

```
PROGRAM Ventilare;
VAR
  fi,fp,fn: FILE OF INTEGER;
  v : INTEGER;
  np,nn:WORD;
BEGIN
  Assign(fi,'FI.DAT');
  Assign(fp,'FP.DAT');
  Assign(fn,'FN.DAT');
  Reset(fi); Rewrite(fp); Rewrite(fn);
  WHILE NOT Eof(fi) DO
    BEGIN
      Read(fi,v);
      IF v > 0 THEN Write(fp,v)
      ELSE IF v < 0 THEN Write(fn,v)
    END;
  np:=FileSize(fp); nn:=FileSize(fn);
  IF (np <> 0) AND (nn <> 0)
    THEN Halt
    ELSE IF np <> 0
      THEN Halt(1)
      ELSE IF nn <> 0
        THEN Halt(2)
        ELSE Halt(3)
    {0 - ambele fișiere nevide}
    {1 - fisierul FN vid}
    {2 - fisierul FP vid}
    {3 - ambele fișiere vide}
END.
```

La revenirea în contextul de lansare a programului în execuție, există soluții, oferite de mediul Turbo Pascal (prin variabila **DosExitCode**) sau de sistemul de operare DOS, pentru testarea valorii codului de retur și luarea unor decizii adecvate de continuare a prelucrărilor. În programul *GestProc* se creează fișierul binar inițial, se lansează în execuție programul *Ventilare* (considerat memorat în fișierul VENT.EXE) și se prelucrează codul de retur.

```
PROGRAM GestProc;
USES Dos;
{$M 2048,0,0}
VAR
  fi : FILE OF INTEGER;
  vi,n,i: INTEGER;
  r : WORD;
BEGIN
  Write('Numar de elemente si valoare initiala:');
  ReadLn(vi,n);
  Assign(fi,'FI.DAT');
  Rewrite(fi);
  FOR i:=vi TO vi+n-1 DO Write(fi,i);
  Close(fi);
  SwapVectors;
  Exec('VENT','');
  SwapVectors;
  r:=DosExitCode;
  IF r = 0 THEN
    WriteLn('OK!')
  ELSE IF r = 1 THEN
```

```
      WriteLn('Fisierul FN vid')
    ELSE IF r = 2 THEN
      WriteLn('Fisierul FP vid')
    ELSE
      WriteLn('Ambele fisiere vide')
    END.
```

• Procedura de încheiere cu eroare de execuție este definită astfel: **RunError[(ExitCode:WORD)]**. La apelul fără parametru, *ExitCode* rămâne cu valoarea 0, iar în cazul apelului cu parametru valoarea parametrului real va fi atribuită variabilei *ExitCode*.

Exemplu:

7.4. În programul *Valid* se realizează introducerea cu validare a unei valori numerice, cu reluare până la furnizarea unei valori corecte, respectiv până la depășirea unui număr limită de reluări.

```
PROGRAM Valid;
VAR  n,i,er : BYTE; x : REAL;
BEGIN
  Write('Numar maxim admis de reluari:'); ReadLn(n); i:=0;
  REPEAT
    er:=0;
    Write('x:');    {$I-} ReadLn(x); {$I+}
    IF IOResult <> 0 THEN
      BEGIN
        Write('Valoare nenumERICA');
        er:=1
      END;
    Inc(i)
  UNTIL (er = 0) OR (i > n);
  IF i > n THEN
    BEGIN
      Write('Depasire numar de reluari admis');
      RunError(106);
    END
  END.
```

Soluția propusă conduce la întreruperea forțată prin apelul *RunError*, la depășirea numărului de repetări admis, cu mesajul standard de eroare corespunzător codului dat de utilizator:

Runtime error 106: Invalid numeric format

• *Includerea rutinelor proprii de încheiere a execuției programelor*. Pe lângă operațiile realizate de sistemul de operare MS-DOS, Turbo Pascal este prevăzut cu câteva rutine de serviciu, a căror lansare, într-o anumită succesiune, este declanșată implicit la terminarea execuției unui program. La apariția evenimentului de terminare a unui program, controlul este transferat primei proceduri din lanțul rutinelor de încheiere a execuției (**exit procedures**), a cărei adresă este memorată în variabila *ExitProc*.

Utilizatorul are posibilitatea să includă propria rutină de tratare a terminării, al cărei apel se va insera la începutul lanțului de apeluri. În acest scop, variabila

ExitProc va fi setată pe adresa procedurii proprii, cu asigurarea refacerii valorii inițiale după încheierea execuției acestei proceduri, asigurându-se astfel continuarea cu procedurile implicite de încheiere.

Procedurile de încheiere proprii oferă programatorului un mijloc de a controla - nu și de a preveni - terminarea unui program. De exemplu, dacă programul folosește o imprimantă, în acest mod se poate asigura sesizarea momentului pentru extragerea ultimei pagini. Tehnica de includere în lanț a procedurii proprii este ilustrată în programul *Ex_ExitProgram*.

```
PROGRAM Ex_ExitProgram;
VAR
  ExitOrig: POINTER;
{F+}
PROCEDURE ExitPropriu;
{$F-}
BEGIN
  {Instrucțiuni ale utilizatorului la terminarea programului }
  WriteLn('Terminarea programului...');
  ExitProc:= ExitOrig; {restaurarea adresei rutinei implicite
                        de încheiere a execuției}
END;
BEGIN
  ExitOrig:=ExitProc; {salvarea adresei rutinei implicite de
  încheiere a execuției}
  ExitProc:=@ExitPropriu; {includerea rutinei proprii de
  încheiere la începutul lanțului de apeluri}
  {Restul programului se scrie normal}
END.
```

Deși procedura *ExitPropriu* nu este chemată explicit, ea va fi prima procedură apelată la terminarea execuției programului, efectul vizibil fiind afișarea mesajului inclus în cadrul ei. Se remarcă folosirea directivei {\$F}, necesară pentru a forța tratarea procedurii *ExitPropriu* ca "îndepărtată" (far). Acest lucru este necesar deoarece rutinele de încheiere implicite aparțin segmentului de cod al unității System, în timp ce procedura *ExitPropriu* este inclusă în segmentul de cod al programului principal.

7.6 Punerea la punct a programelor Turbo Pascal

În raport de momentul în care se manifestă, pot fi identificate trei categorii de erori: de compilare, la execuție și de logică.

Erorile de compilare (Compile-Time Errors) sunt, în general, rezultatul unor greșeli lexicale sau sintactice. Erorile lexicale apar când un identificator, un literal sau un simbol folosit în program este ilegal. De exemplu, considerând declarația:

```
CONST hex1=$FFHH;
```

se produce eroare lexicală, deoarece în specificarea valorii constantei hexa apar caractere nepermise (HH). Eroarea este semnalată printr-un mesaj de forma: *Error 7: Error in integer constant*. O eroare similară ar produce și folosirea unui literal hexa **\$-12**, deoarece acești literalii nu pot avea semn.

Declarația de constantă simbolică:

```
CONST sir='pret unitar=;
```

generează eroare lexicală, compilatorul interpretând absența apostrofului de închidere a șirului, în sensul următorului mesaj: *Error 8: String constant exceeds line*.

Erorile sintactice corespund cazurilor în care unele construcții folosite în program încalcă regulile formale ale limbajului, cum ar fi specificarea unui număr greșit sau de tipuri neadecvate de parametri în apelul unor proceduri sau funcții, omiterea caracterului ";" la încheierea unei instrucțiuni sau folosirea lui într-un context nepermis etc. Mesajele nu sunt întotdeauna foarte explicite, dar deplasarea cursorului în fereastra Turbo Pascal, exact în locul în care a fost sesizată eroarea, îl va ajuta pe utilizator să determine cauza acesteia. În plus, la folosirea tastei **F1** se afișează o fereastră de informare cu explicații suplimentare privind semnificația erorii. De semnalat faptul că unele erori, care la o primă analiză par lexicale, sunt interpretate de compilator ca erori de sintaxă. Se consideră declarația:

```
VAR pret-unitar: REAL;
```

În sens strict, s-a produs o eroare lexicală, identificatorul fiind construit incorect ("-" în loc de "_"). Totuși, mesajul de eroare al compilatorului este *Error 86: ":" expected* și nici cel suplimentar, afișat la apăsarea tastei **F1**, *A colon doesn't appear where it should* nu este cu mult mai explicit. Compilatorul interpretează că identificatorul este **pret**, după care, fiind în secțiunea VAR a unui program, așteaptă caracterul ":", care ar trebui să urmeze în sintaxa declarației. Deoarece cursorul se plasează sub caracterul "-", este de presupus că utilizatorul va sesiza ușor adevărata cauză a erorii.

În multe cazuri, eroarea sintactică este sesizată după locul strict în care s-a produs, în momentul în care compilatorul constată absența unui element pe care îl aștepta în contextul respectiv. De exemplu, considerând secvența:

```
VAR
  pret_unitar: REAL
BEGIN
```

absența caracterului ";" de încheiere a declarației variabilei se sesizează doar la începutul liniei următoare, mesajul de eroare fiind: *Error 85: ":" expected*, iar cursorul se va plasa la începutul cuvântului BEGIN.

Pentru o instrucțiune de forma:

```
IF a > b THEN WriteLn(a,'> ',b);
ELSE WriteLn(a,'<= ',b);
```

mesajul de eroare este: *Error 113: Error in statement*. Cauza erorii este folosirea incorectă a terminatorului ";" la încheierea ramurii THEN a instrucțiunii IF, iar eroarea este sesizată la întâlnirea lui ELSE, cu plasarea cursorului la începutul

acestui. La folosirea tastei **F1** se afișează fereastra de informare cu mesajul *This symbol can't start a statement*, care, de această dată, este suficient de explicit.

Erorile la execuție (Run-Time Errors) se mai numesc și semantice și apar atunci când instrucțiuni valide pe plan sintactic sunt combinate încorect. Printre cazurile mai frecvente din această categorie pot fi menționate: încercarea de a realiza operații I/O cu fișiere nedeschise, tentativa de citire dincolo de sfârșitul unui fișier, depășirea domeniului de valori permis pentru unele tipuri de variabile, depășirea stivei, împărțire la zero etc. În toate cazurile, execuția programului se încheie anormal, se afișează un mesaj de eroare, iar cursorul se deplasează în fereastra de editare a mediului Turbo Pascal, în contextul din care s-a generat eroarea, așteptându-se intervenția utilizatorului. Trebuie menționat că, depinzând de modul de folosire a unor directive de compilare, programatorul poate inhiba sau nu semnalarea unora dintre erori sau poate prelua controlul în caz de eroare, evitând încheierea anormală și prematură a programului.

Un exemplu tipic din prima categorie este directiva `{SR}`, pentru controlul depășirii domeniului de valori al datelor de tip întreg. Întrucât valoarea ei implicită este `{SR-}`, execuția programului:

```
VAR a: BYTE;  
BEGIN  
    a:=255; a:=a+1;  
END.
```

se va încheia fără eroare, deși la a doua operație de atribuire se depășește domeniul de valori admis pentru tipul `BYTE`. Introducând în program o primă linie cu directiva `{SR+}`, execuția se încheie cu eroare, afișându-se mesajul *Error 201: Range check error*, iar cursorul se va plasa la începutul instrucțiunii care a generat eroarea.

Un exemplu pentru evitarea încheierii anormale îl poate constitui directiva `{SI}`, care permite programatorului preluarea controlului în caz de eroare la realizarea operațiilor de intrare/ieșire (programul *Valid*).

Mesajele erorilor de execuție au forma: *Runtime error 106 at ssss:dddd*, unde **ssss** reprezintă segmentul, iar **dddd** deplasarea corespunzând adresei instrucțiunii la care s-a produs eroarea. Execuția se întrerupe, iar cursorul se va deplasa pe linia din programul sursă cu instrucțiunea care a generat eroarea (în exemplul considerat, linia cu instrucțiunea de apel al procedurii `ReadLn`).

Erorile de logică (Logic Errors) sunt cel mai greu de îndepărtat, deoarece programul se poate încheia "normal", dar rezultatele nu sunt cele așteptate. Acestea sunt generate de greșelile de proiectare a programelor. Pentru depistarea lor este necesară, de multe ori, folosirea unor tehnici speciale de depanare.

Depanarea interactivă se realizează în cadrul unei sesiuni de depanare, care presupune execuția supravegheată a programului. Utilizatorul are posibilitatea să aleagă între: folosirea unui *depanator integrat* (Integrated Debugger), inclus în mediul de programare; folosirea unui *depanator autonom* (Standalone Debugger), respectiv pachetul de programe **Turbo Debugger** al firmei Borland.

Folosirea depanatorului integrat, abordată în cele ce urmează, presupune setarea comutatorului `Options|Debugger|Integrated`. În procesul de depanare

interactivă pot fi identificate două etape: pregătirea sesiunii de depanare, respectiv desfășurarea acesteia.

- *Pregătirea sesiunii de depanare interactivă* corespunde operațiilor ce se desfășoară pe parcursul fazelor de compilare și editare de legături și care, prin informațiile produse, condiționează desfășurarea sesiunii propriu-zise. Informațiile necesare procesului de depanare sunt generate de directive de compilare sau de comenzi echivalente din meniul **Options**. Folosirea depanatorului integrat depinde de modul de specificare a directivelor de compilare **{SD}** și **{SL}**, cărora le corespund comutatoarele **Options|Compile|Debug Information**, respectiv **Options|Compile|Local Symbols**.

Directiva {SD} activează sau inhibă generarea unor informații de depanare, care constau dintr-o tabelă ce precizează corespondența dintre numerele liniilor din programul sursă ce conțin instrucțiuni executabile și adresa codului obiect generat pentru fiecare dintre ele. Valoarea implicită a acestei directive este **{SD+}** și folosirea ei permite execuția pas cu pas și definirea punctelor de întrerupere în programele compilate în acest mod. În plus, la apariția unor erori la execuție, comanda **Compile|Find Error** asigură localizarea automată a instrucțiunii ce a produs eroarea. Directiva condiționează și obținerea așa numitei "mape" a editorului de legături, respectiv un fișier cu extensia **.MAP**, ale cărui caracteristici și conținut depind de modul de setare a butoanelor radio din grupul **Options|Linker|Map File**:

- (.) **Off**
- () **Segments**
- () **Public**
- () **Detailed**

Butonul **Off** este selectat implicit și are ca efect neproducerea fișierului **.MAP**. La selectarea butonului **Segments**, fișierul va conține informații privind adresa de start, lungimea și numele segmentelor de cod, date și stivă, precum și adresa de început a zonei heap. Alegerea butonului **Public** face ca, pe lângă informațiile privind segmentele, în fișierul **.MAP** să se includă un tabel cu adresele obiectelor "publice", adică ale subprogramelor interne, ale constantelor cu tip și ale variabilelor globale definite în program, precum și ale variabilelor globale ale unităților folosite (în absența clauzei **USES**, numai cele ale unității **System**). De asemenea, se precizează adresa punctului de lansare în execuție a programului (*program entry point*). Cu butonul **Detailed** se afișează, în plus, un tabel cu numerele liniilor cu instrucțiuni executabile din programul sursă, urmate de adresa codului obiect generat în cadrul segmentului de cod al programului principal. Sunt incluse și instrucțiunile executabile ale eventualelor subprograme interne.

Directiva {SL} activează sau inhibă generarea unor informații despre simbolurile locale, adică variabilele și constantele declarate în cadrul subprogramelor interne ale unui program. Valoarea implicită a directivei este **{SL+}** și folosirea ei condiționează examinarea și modificarea variabilelor locale, precum și urmărirea succesiunii de generare și execuție a apelurilor de proceduri și funcții interne.

Directivele **{SD}** și **{SL}** se folosesc de obicei împreună, cu precizarea că **{SL}** este ignorată dacă **{SD}** nu este activă. Pentru depanare, pentru a nu fi

condiționați de eventualele setări implicite pe parcursul sesiunii de lucru, se recomandă includerea la începutul programului a configurației: {\$D+,L+}. Deoarece atât {\$D+} cât și {\$L+} generează spațiu de memorie suplimentar, după punerea la punct a programelor se recomandă resetarea acestora.

• *Desfășurarea sesiunii de depanare interactivă.* Depanatorul integrat oferă utilizatorului mai multe comenzi incluse în meniurile **Run**, **Debug** și **Window** ale mediului Turbo Pascal, utile în procesul de depanare: execuția pas cu pas a programelor, cu sau fără execuția în aceeași manieră a subprogramelor; vizualizarea valorilor unor variabile (mai general, expresii), cu eventuala modificare a acestora; crearea unor puncte de întrerupere; vizualizarea conținutului stivei. În practica programării, comenzile se grupează pe parcursul sesiunii de depanare în funcție de obiectivele urmărite. Sesiunea de depanare este formată din următoarele componente: deschiderea sesiunii; sesiunea propriu-zisă; închiderea sesiunii. Chiar dacă tehnicile de depanare sunt diferite, în cadrul lor se regăsesc cel puțin două elemente comune: execuția pas cu pas a instrucțiunilor, urmărirea în paralel a rezultatelor obținute.

a) *Deschiderea sesiunii* se realizează cu una din comenzile meniului **Run**:

- **Run|Trace Into** (F7) sau **Run|Step Over** (F8), când se urmărește depanarea de la începutul programului;
- **Run|Go to Cursor** (F4), în cazul unei depanări dintr-un anumit punct.

În primul caz, efectul comenzii este afișarea unei bare de execuție (**run bar**), care supraluminează linia BEGIN a programului principal. În al doilea caz, anterior deschiderii sesiunii are loc deplasarea cursorului pe o anumită linie, iar la folosirea comenzii se execută instrucțiunile în mod normal pînă la linia marcată, după care se intră într-o manieră de lucru asemănătoare variantei anterioare.

O variantă a depanării dintr-un anumit punct o constituie crearea punctelor de întrerupere (**breakpoints**), cu ajutorul cărora se selectează instrucțiunile din program corespunzând locurilor unor prezumtive greșeli de logică. Crearea punctelor de întrerupere se realizează anterior deschiderii sesiunii de depanare propriu-zise, prin deplasarea cursorului în textul sursă în contextul respectiv și folosirea comenzii **Debug|Toggle breakpoint** (sau <Ctrl><F8>). Punctul de întrerupere va fi marcat printr-o bară luminoasă, iar în timpul sesiunii de depanare execuția programului se va întrerupe înainte de executarea instrucțiunilor de pe linia marcată. Comanda poate realiza și eliminarea punctelor de întrerupere, dacă este folosită pentru linii unde există deja definite asemenea puncte. Punctele de întrerupere pot fi gestionate și cu comanda **Debug|Breakpoints**, care, prin intermediul unei ferestre de dialog, permite ștergerea unor puncte (butonul de comandă **Delete**), definirea unor noi puncte de întrerupere sau modificarea caracteristicilor celor existente (butonul de comandă **Edit**).

b) *Sesiunea propriu-zisă* corespunde aplicării unor tehnici de depanare, bazate pe folosirea într-o anumită succesiune a unor comenzi din meniurile **Debug** și **Window**, după deschiderea acestora cu una din comenzile menționate ale meniului

Run. Urmărirea valorilor variabilelor este asociată cu execuția pas cu pas, de la început sau dintr-un anumit punct, fiind posibilă urmărirea modului cum evoluează valorile unor variabile (mai general, expresii) pe măsura execuției instrucțiunilor. În consecință, fie de la început, fie pe parcursul sesiunii, este necesară precizarea variabilelor (expresiilor) ale căror valori vor fi afișate într-o fereastră de urmărire (**watch window**). Se folosesc comenzile din submeniul **Debug|Watch**, prin care se poate specifica: includerea unui nou element (**Add watch**); ștergerea unui element (**Delete watch**); modificarea unui element (**Edit Watch**).

Submeniul are o fereastră proprie de dialog cu un câmp de intrare în care se realizează operații asupra elementului curent din lista expresiilor de urmărit, marcat printr-o bară luminoasă. La prima introducere a unui element în listă are loc automat deschiderea ferestrei de urmărire și amplasarea acestuia la începutul listei, cu afișarea valorii sale (la începutul sesiunii valoarea este nedefinită).

Desfășurarea sesiunii presupune execuția pas cu pas a instrucțiunilor, de la început sau din punctul ales, cu afișarea valorilor variabilelor (expresiilor) incluse în fereastra de urmărire. Activarea ferestrei de urmărire, atât pentru urmărirea prin defilare a tuturor valorilor cât și pentru operații de editare asupra listei, se realizează cu comanda **Window|Watch**.

Urmărirea conținutului stivei este legată de lucrul cu subprograme proprii, permițând controlul succesiunii apelurilor de proceduri și funcții, până la atingerea apelului procedurii aflată curent în execuție. Se bazează pe folosirea comenzii **Window|Call stack (<Ctrl><F3>)**, care deschide o fereastră pentru afișarea succesiunii apelurilor de subprograme.

c) *Închiderea sesiunii* se realizează cu comanda **Run|Program reset (<Ctrl><F2>)**.