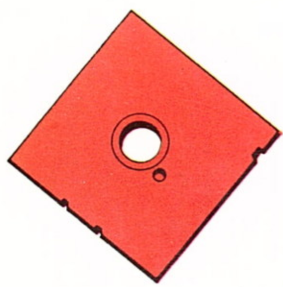


PROGRAMOZZUNK TURBO PASCAL NYELVEN!

KEZDŐKNEK * KÖZÉPHALADÓKNAK

VERZIÓ 5.0, 5.5, 6.0

ComputerBooks

ÉRTÉKESÍTÉS
SZERVÍZ
HÁLÓZATÉPÍTÉS
HÁLÓZATOPTIMALIZÁLÁS
ADATVÉDELEM

TRIGON

TRIGON HARDWARE KFT.

Budapest 1202 Nagykörösi út 114. T: 185-82-93, Fax: 163-69-26

Bemutatóterem és iroda: 1031 Bp. Kadosa u. 57. T: 160-74-5

Levélcím: 1775 Bp. Pf:

BENKŐ TIBORNÉ
BENKŐ LÁSZLÓ
TÓTH BERTALAN
VARGA BALÁZS

PROGRAMOZZUNK TURBO PASCAL NYELVEN!

VERZIÓ 5.0, 5.5, 6.0

KEZDŐKNEK * KÖZÉPHALADÓKNAK

LEKTOR
TÓTH BERTALAN

COMPUTERBOOKS
Budapest, 1993

Kiadó: ComputerBooks Kiadói Kft
1126 Bp. Tartsay V. u. 12.
Tel:1751–564, Tel/fax:1753–591

Felelős kiadó: a ComputerBooks ügyvezetője

Borítógrafika: Székely Edith

ISBN: 963 7642 42 0

Köszönetnyilvánítás

Ezúton szeretnénk köszönetet mondani Somodiné Hajdú Éva, a Kaffka Margit Gimnázium matematika és számítástechnika szakos tanárnőjének, aki sok ötletet adott a kézirat gyakorlati feladatokkal való bővítésére, elősegítve ezzel könyvünk felhasználását a Turbo Pascal nyelv oktatásában.

Köszönetet szeretnénk mondani Bokker Tamás és Berkovics Ákos tanulóknak, akik a készülő kézirat első kilenc fejezetéből tanultak programozni, megválaszolva az ellenőrző kérdéseket és a gyakorlati feladatok tanulmányozása után önálló programokat készítettek. A tapasztalat azt mutatta, hogy az ellenőrző kérdések hasznosan segítették elméleti tudásuk elmélyülését.

Vizi Bernadett tanuló vállalta a játékprogramok tesztelését és szívesen játszott a programokkal.

Végül köszönetet szeretnénk mondani szerzőtársunknak, Benkő Lászlónak, aki vállalta a kézirat szerkesztetését és a könyv vonalas ábráinak rajzolását.

TARTALOMJEGYZÉK

BEVEZETÉS 1

1. ALGORITMUS ÉS A PROGRAM 3
1.1. Mielőtt számítógépet programoznánk 3
1.2. A program megtervezése 3
1.2.1. A programozás alapja 3
1.2.2. A program folyamatábrája 4
1.2.3. A számítógépes program készítése 6
1.3. Mintafeladat 9

2. GONDOLATOK A PASCAL NYELVRŐL 13
2.1. A programokról általában 13
2.2. A Pascal nyelvről röviden 13
2.3. A Pascal program szerkezete 14
2.4. Az első Pascal program 15
2.4.1. A változók deklarálása 17
2.4.2. Olvasás billentyűzetről 20
2.4.3. Kiírás képernyőre 22
2.4.4. Számítás 26

3. A TURBO PASCAL NYELV ELEMEI 35
3.1. A nyelv jelkészlete 37
3.2. A foglalt szavak 37
3.3. A program sorai 38
3.4. Azonosítók 39
3.5. Számok 40
3.6. Sztringek - karaktersorozatok 41
3.7. Cimkék 42
3.8. Operandusok 43
3.9. Operátorok 43
3.10. Kifejezések 43
3.11. Megjegyzések 44

4. A TURBO PASCAL PROGRAM SZERKEZETE 47

5. ADATTIPUSOK ÉS DEKLARÁCIÓK 51
5.1. Egyszerű adattípusok 52
5.1.1. Numerikus információ tárolása 53
5.1.1.1. Egész típusok 53
5.1.1.2. Valós típusok 54
5.1.2. Logikai információ tárolása 54
5.1.2.1. Logikai típus 55
5.1.3. Szöveges információ tárolása 55
5.1.3.1. Karakter típus 55
5.1.3.2. String típus 55
5.1.4. Sorszámozott típus 56
5.1.4.1. Felsorolt típus 57
5.1.4.2. Résztartomány típus 58
5.1.5. Mutató típus 59
5.2. Struktúrált típusok 59
5.2.1. Tömb típus 59
5.2.2. Rekord típus 61
5.2.3. Halmaz típus 63
5.2.4. File típus 63
5.3. Felhasználói típus létrehozása (type) 64
5.3.1. A type használata felsorolt típus esetén 64
5.3.2. A type használata résztartomány típus esetén 65
5.3.3. A type használata tömb esetén 65
5.3.4. A type használata record esetén 66
5.3.5. A type használata halmaz esetén 67
5.4. Konstansok deklarálása 67
5.4.1. Típusos konstansok 68
5.5. Típuskonverzió 70

6. PASCAL KIFEJEZÉSEK 75
6.1. Operátorok 75
6.2. Elsőbbségi szabály 76
6.3. Operátorok csoportosítása 76
6.3.1. Aritmetikai operátorok 77
6.3.2. Bitenkénti logikai operátorok 83
6.3.3. Boolean operátorok 85
6.3.4. Relációs (összehasonlító) operátorok 89

6.3.5. Pointer operátorok ... 90
6.3.6. Sztring összekapcsolás operátora ... 91
6.3.7. Halmaz műveleti operátorok ... 91
6.3.7.1. Halmazokra vonatkozó vizsgálatok ... 94
6.4. Standard (szabványos) függvények használata ... 95

7. ALAPVETŐ I/O MŰVELETEK ... 105
7.1. Írás képernyőre: Write és Writelen eljárás ... 105
7.1.1. Szöveg kiírása a képernyőre ... 106
7.1.2. Egész típusú eredmények kiiratása ... 108
7.1.3. Valós típusú eredmények kiiratása ... 110
7.1.4. Boolean típusú eredmények kiiratása ... 112
7.2. Olvasás billentyűzetről: Read, Readln eljárások ... 117

8. A PASCAL NYELV UTASÍTÁSAI ... 127
8.1. Egyszerű utasítások ... 128
8.1.1. Üres utasítás ... 128
8.1.2. Értékadó utasítás ... 128
8.1.2.1. Aritmetikai értékadás ... 129
8.1.2.2. Logikai értékadás ... 133
8.1.3. Goto utasítás ... 134
8.1.4. Eljáráshívás ... 135
8.2. Struktúrált utasítások ... 135
8.2.1. A blokk utasítás ... 135
8.2.2. Feltételes utasítások ... 135
8.2.2.1. If utasítás ... 136
8.2.2.2. Case utasítás ... 141
8.2.3. Ciklusutasítások ... 144
8.2.3.1. For utasítás ... 144
8.2.3.2. While utasítás ... 147
8.2.3.3. Repeat utasítás ... 149
8.2.4. With utasítás ... 150

9. ELJÁRÁSOK ÉS FÜGGVÉNYEK ... 157
9.1. Függvények ... 164
9.2. Eljárások ... 168
9.3. Típusdefiníció használata a paraméterlistán ... 178

9.4. Eljárás paramétere: függvény 179
9.5. Forward deklaráció - előre hivatkozó 182
9.6. Rekurzív alprogramok 182
9.6.1. A rekurzív alprogramok csoportosítása 186
9.7. Globális és lokális változók, az azonosítók érvényességi köre 188
9.8. Sztring használata Turbo Pascal-ban 191
9.8.1. Szabványos eljárások és függvények a sztringek kezelésére 192

10. MODULOK HASZNÁLATA A TURBO PASCAL-BAN 205
10.1. A Turbo Pascal modulok felépítése 205
10.1.1. A modulok fejléce - a modulok közötti kapcsolat 206
10.1.2. Az interface rész 207
10.1.3. Az implementation rész 208
10.1.4. Az inicializációs rész 210
10.2. A modulok használatát bemutató példaprogram 211
10.3. Szabványos modulok 213

11. FILE-KEZELÉS 217
11.1. Turbo Pascal szöveges file-ok 217
11.1.1. A szöveg file azonosítása 220
11.1.2. A szöveg file megnyitása 220
11.1.3. A szöveg file I/O műveletei 223
11.1.4. Szöveg file lezárása 225
11.1.5. A szabványos szöveg file-ok: input és output 226
11.1.6. Példaprogramok text típusú file-ok használatára 230
11.2. Típusos file-ok 234
11.2.1. Típusos file deklarálása és megnyitása 236
11.2.2. File-műveletek 238
11.2.3. A file lezárása 240
11.2.4. Példák típusos file-ok használatára 240
11.3. Típus nélküli file-ok 248
11.4. Eszközök (device) használata 252

11.5. File-ok törlése, átnevezése 255
11.6. Könyvtárak kezelése 255

12. A TURBO PASCAL MEMÓRIAHASZNÁLATA 259
12.1. A mutató típus - dinamikus változók 259
12.1.1. Tömb a halomterületen 261
12.1.2. Mutatótömb használata 262
12.1.3. A lista tárolási szerkezet 263
12.1.4. A saját verem kialakítása 267
12.2. További lehetőségek a memória elérésére 269
12.2.1. Amit a 8086 mikroprocesszorról tudni kell 269
12.2.2. A Turbo Pascal és a szegmentált memória 271
12.2.3. Speciális lehetőségek a memória elérésére 274
12.2.3.1. A Mark és a Release eljárások használata 274
12.2.3.2. A Getmem és a Freemem eljárások használata 275
12.2.3.3. A mutatókról bővebben 277
12.2.3.4. Az absolute deklaráció 282
12.2.3.5. A memória és a portok közvetlen elérése 283
12.2.3.6. Típus nélküli paraméterek használata 284

13. A CRT UNIT HASZNÁLATA 289
13.1. A szöveges üzemmód 289
13.2. Szöveges mód változói, konstansai, függvényei és eljárásai 291
13.2.1. Szöveges mód változói 291
13.2.2. Szöveges mód konstansai 293
13.2.3. Szöveges mód eljárásai és függvényei 295
13.3. A szöveges mód programozása 296
13.3.1. Mintaprogramok a Crt unit használatára 296
13.3.2. Adat beolvasása és ellenőrzése 299
13.3.3. Menükezelés 302

14. GRAPH UNIT HASZNÁLATA ... 305
14.1. Grafikus mód ... 305
14.2. Graph unit ... 305
14.3. Grafikus vezérlők típusa ... 306
14.4. A grafikus könyvtár eljárásainak és függvényeinek csoportosítása ... 309
14.5. Grafikus programok készítése ... 319
14.5.1. Színkezelés különböző vezérlők esetén ... 319
14.5.2. Jelentősebb mód konstansok ... 319
14.5.3. Grafikus program felépítésének vázlata ... 320
14.5.4. Grafikus üzemmód hibajelzései ... 324
14.6. Grafikus mintafeladatok ... 325
14.6.1. Szöveg kiírása grafikus módban ... 325
14.6.2. Szöveges és grafikus mód váltása ... 329
14.6.3. CGA.BGI és a LITT.CHR programba fordítása ... 330
14.6.4. Grafikus kurzor mozgatása ... 332
14.6.5. Alakzat mozgatása ... 334
14.6.6. Képernyő torzításának kiküszöbölése ... 334
14.6.7. Alakzatok rajzolása ... 335
14.6.8. Kép kivágása és újrahelyezése ... 336

15. MINTAFELADATOK ... 339
15.1. Másodfokú egyenlet megoldása ... 339
15.2. Adatok rendezése ... 347
15.2.1. Rendezés cserével ... 348
15.2.2. Sztring típusú adatok rendezése ... 351
15.2.3. Különféle rendező algoritmusok ... 353

16. A TURBO PASCAL SPECIÁLIS LEHETŐSÉGEI ... 361
16.1. Overlay használata ... 361
16.2. Rendszerhívások ... 367
16.3. Megszakítási rutinok készítése - TSR programok ... 370
16.4. A DOS unit file-kezelési lehetőségei ... 375
16.5. Programok indítása ... 378

17. JÁTÉKPROGRAMOK KÉSZÍTÉSE 381
17.1. Számkirakó játékprogram 381
17.2. Memória játék 384
17.3. Türelem tüske játékprogram 387
17.4. Rex játékprogram 391
17.5. Játékok keretprogramja 397

FÜGGELÉK
F1. A TURBO PASCAL SZABVÁNYOS ELJÁRÁSAI ÉS FÜGGVÉNYEI 399
F1.1. Matematikai függvények 399
F1.2. Függvények a megszámlálható típusokra 402
F1.3. String kezelések 404
F1.4. Byte és regiszter műveletek 407
F1.5. Könyvtárak kezelése 408
F1.6. File kezelések 412
F1.7. DOS és rendszer paraméterek kezelése 423
F1.8. Overlay kezelés 432
F1.9. Pointerek kezelése 435

F2. A CRT UNIT ELJÁRÁSAI ÉS FÜGGVÉNYEI 441

F3. A GRAPH UNIT ELJÁRÁSAI ÉS FÜGGVÉNYEI 447

F4. A TURBO PASCAL FORDÍTÓ DIREKTÍVÁI 473
F4.1. A kapcsoló direktívák 474
F4.2. Paraméter direktívák 482
F4.3. Feltételes fordítás 483
F4.3.1. Feltételes szimbólumok 484
F4.3.2. Feltételes direktívák 485

F5. ÖSSZEFOGLALÓ TÁBLÁZATOK 487
F5.1. A Turbo Pascal futás közbeni hibaüzenetei 487
F5.2. A funkcióbillentyűk visszatérési kódjai 490
F5.3. IBM karakterkódok táblázata 492
IRODALOMJEGYZÉK 493
TÁRGYMUTATÓ 494

BEVEZETÉS

Jelen könyv a Pascal magasszintű programozási nyelvet ismerteti. 1968-ban N. Wirth, a Zürichi Műszaki Egyetem tanára készítette el a nyelv vázlatát az ALGOL programozási nyelv filozófiájának felhasználásával. 1970-ben munkatársaival létrehozta a nyelv első fordító programját. A nyelvet Blaise Pascal-ról, a XVII. század jelentős francia tudósáról nevezte el, aki több tudományban is kiemelkedő eredményeket ért el. Sorelméleti kutatásaival alkotta meg a Pascal-háromszöget, a nyomás egyenletes terjedésének törvényeit rögzítette a Pascal-tételben, a barometrikus magasságmérés kidolgozásának állít emléket a róla elnevezett mértékegység. A valószínűségszámítás és a filozófia is a nagyjai között tartja nyilván, egy programnyelv névadójává mégis talán azért lett, mert ő készítette el az első működő (mechanikus) számológépet.

1973-ban definiálták a szabványos (standard) Pascal nyelvet. A nyelv gyorsan elterjedt és hamarosan az egyik legnépszerűbb magasszintű programozási nyelvvé vált az egész világon. Népszerűségét és hatékonyságát nagyban növelte az amerikai Borland cég által IBM PC-re készített Turbo Pascal változat, mely a szövegszerkesztő és fordítóprogram összekapcsolásával és számos más kiegészítéssel igen kényelmes fejlesztő eszközt adott a programozók kezébe.

A könyvünk elsősorban azoknak szól, akik segítségünkkel kívánják megismerni a Pascal nyelv rejtelmeit és szépségeit. A nyelv leírása mellett a legalapvetőbb programozással kapcsolatos fogalmakat is tisztázzuk. A Pascal nyelv szintaktikájával és szabályaival foglalkozó fejezetek példákon keresztül illusztrálják az elméletet. Különösen ajánljuk az első két fejezet áttanulmányozását azok számára, akik ezideig semmilyen programozási nyelvvel nem foglalkoztak.

Akik már foglalkoztak más programozási nyelvvel, vagy ismerik a Pascal utasításait, de még nem tudnak önállóan programozni, azoknak is ajánljuk, hogy ismételjék át a Pascal nyelv ismertetését leíró fejezeteket és önállóan oldják meg az ott közölt gyakorló feladatokat. A megoldást csak akkor érdemes megnézni, amikor már átgondolták a feladatot és ellenőrizni kívánják a munkájukat. Természetesen a megoldás sikertelen próbálkozás

esetén is segítséget nyújt, hiszen megértve annak gondolatmenetét az Olvasó programozási tapasztalatokra tesz szert. Kezdő programozók a kilencedik fejezettel bezárólag eljutnak a Pascal nyelv alapjainak megismeréséig. Mielőtt tovább lépnének érdemes a fejezetek végén található ellenőrző kérdések és feladatok önálló feldolgozásával átismételni az olvasottakat.

Azoknak is ajánljuk a könyvet, akik a Pascal nyelvet használják egyszerűbb feladatok megoldására, de még nem ismerik a Turbo Pascal rendszer által nyújtott lehetőségek teljes tárházát.

Hasznos ismereteket nyújt a könyv mindazoknak, akik idáig nem értették meg a rekurzív rutinok és a pointerek használatát. Részletesen tárgyalja a nagy programok készítéséhez elengedhetetlenül szükséges modulok felhasználását és készítését. Kiemelést érdemel a Turbo Pascal file-kezelési lehetőségeit bemutató fejezet.

Külön fejezetben tárgyaljuk a komolyabb programok felhasználói felületének kialakításához szükséges szöveges és grafikus könyvtárak alkalmazását. Azoknak is ajánljuk a könyvet, akik numerikus számításokra írtak már programot, de nem foglalkoztak grafikus programok készítésével. Külön fejezetek foglalkoznak ezeknek az elsajátításával.

Játékprogramok példáján keresztül is bemutatjuk, hogyan használható a számítógép szöveges és grafikus üzemmódban.

A lemezmelléklet a fejezetben található példák és feladatok forrás programjait tartalmazza.

Szeretnénk felhívni a tisztelt Olvasó figyelmét, hogy jelen kötetünk a ComputerBooks által kiadásra kerülő Pascal trilógia bevezető kötetét jelenti. A trilógia második kötete a már megjelent az "Objektum-Orientált programozás Turbo Pascal 6.0-ban, Turbo Vision" című könyv, a harmadik kötete pedig a Windows programozásával ismerteti meg az Olvasót Turbo Pascal for Windows rendszerben.

1. ALGORITMUS ÉS A PROGRAM

1.1. Mielőtt számítógépet programoznánk...

Mielőtt mélyebben belemerülnénk a számítógép programozásába, tudomásul kell vennünk a számítógép sajátosságát, vagyis azt hogy mit ért meg, és mit nem. Sokat beszélnek az *okos* és az *intelligens* számítógépről. Ez megfogalmazás így nem egészen igaz. Maga a számítógép nem okos, viszont ezt a hiányosságát pótolni igyekszik a gyorsaságával és a pontosságával. Okossá a számítógépet a rajta futó programok teszik. Jó programot viszont csak akkor tudunk írni, ha tudjuk, hogy a számítógép megérti a szándékainkat, és nem fogja félreérteni azokat. Ehhez viszont jól át kell gondolni, hogy mit szeretnénk csinálni és azt hogyan tudjuk véghezvinni.

1.2. A program megtervezése

Tisztázzunk a témakörrel kapcsolatos néhány fontosabb alapfogalmat, mielőtt a számítógépes problémamegoldás főbb lépéseinek tárgyalására rátérnénk.

1.2.1. A programozás alapjai

A *programozás* azt a folyamatot jelenti, amikor a feladatot a számítógép számára érthető formában írjuk le.

A programozás eredményeként létrejön a *program*. A program legkisebb funckionális egysége az *utasítás*. A program az utasítások olyan sorozatából áll, amely a meghatározott feladatot tartalmazza. A leírás nyelvét *programozási nyelvnek* nevezzük. A programozási nyelvek szolgálnak arra, hogy a programozó munkáját megkönnyítsék a feladat megfogalmazásakor. Például a Pascal egy magasszintű programozási nyelv, amelyet könnyű megtanulni.

Mivel a számítógép csak a gépikódú információkat tudja közvetlenül feldolgozni, ezért a kész programot át kell alakítani, le kell fordítani a gép nyelvére. A *fordítóprogramokat* (*compiler*) erre a célra készítették.

A programozási nyelvnek mint, minden nyelvnek, pontosan meghatározott szabályai, lényegében nyelvtana van, amit *szintaxisnak* nevezünk.

Szintaktikus hibáról beszélünk, ha a szabályoktól eltérő módon írjuk meg a program valamelyik utasítását. *Szemantikus hiba* akkor lép fel, ha a nyelv elemi szabályai ellen nem vétünk (a program szintaktikusan helyes), azonban az utasítás, vagy utasítások sorozata a megoldandó feladat szempontjából helytelen (például előjelhiba, elsőbbségi szabály megsértése, stb).

1.2.2. A program folyamatábrája

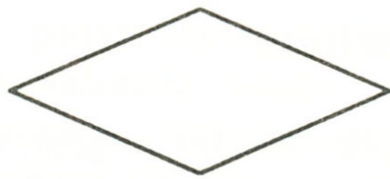
Számítógépes programok és a szervezési folyamatok tervezésénél használjuk a folyamatábrát, amely segíti a programok áttekinthetőségét, kiegészíti a dokumentálást, a programban esetleges későbbi javításokat is megkönnyíti.

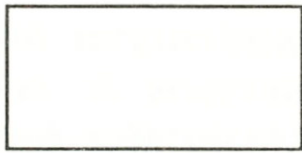
A folyamatábra bemutatja a program folyamatát minden elágazásával - rendszerint anélkül, hogy magát az utasítássorrendet is részletezné. Ha a programot utasításokra bontva ábrázoljuk, akkor utasításdiagramról beszélünk. Ismerkedjünk meg a leglényegesebb szimbólumokkal, amelyeknek felhasználásával rajzoljuk meg a folyamatábrákat (1.1. ábra).

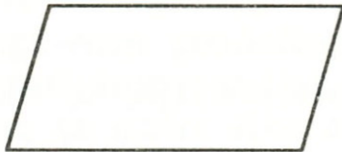
Bemutatjuk a szimbólumokat az

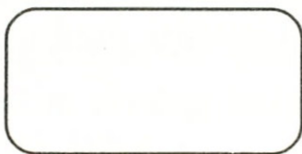
- elágazás,
- művelet,
- Input/Ouput (be-/kimenet) művelet,
- program kezdete és vége,
- folyamat iránya,
- összekapcsolás

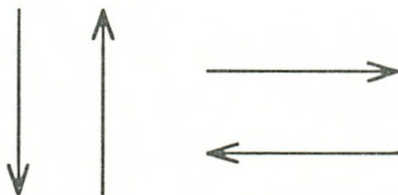
tervezésére.


Elágazás


Művelet


Input/Output művelet


A program kezdete és vége


A folyamat iránya

Összekapcsolás

1.1. ábra A folyamatábra elemei

1.2.3. A számítógépes program készítése

A megoldandó probléma egyértelmű megfogalmazásától a gépi megoldásig a következő főbb lépéseket kell megtenni:

1) A feladat megfogalmazása és elemzése.
(meglepő, de talán ez a legnehezebb)
2) A matematikai modell kialakítása. Milyen módszert alkalmazunk, milyen részekre lehet a feladatot bontani?
3) Az adatstruktúra kialakítása. Milyen adatokra van szükség a program futásához?
4) A program eredményközlése. Meg kell tervezni, hogy a program a legjobban kiértékelhető formában adja vissza az eredményt.
5.) A program folyamatábrájának elkészítése.
6.) A program megírása.
7.) A program tesztelése, hibakeresése, futtatási eredmények kiértékelése.
8.) A program dokumentálása.

A feladat megfogalmazásán azt a folyamatot értjük, amikor a rendelkezésre álló bemenő és kimenő adatok figyelembevételével mindennapi nyelven részletesen leírjuk. A feladatot elemezni kell, meg kell határozni a lehetséges megoldások módját, egyáltalán létezik-e megoldás. Ha a feladatnak vannak olyan részei, amely a programozás szempontjából kiritikus, azokat külön feljegyezzük.

Ezután következik a feladat matematikai modelljének kialakítása. Meg kell határozni a számítógépre a leginkább alkalmas matematikai megoldási módszereket. Ezeknek megalkotása a programozás egyik legnehezebb és legszerteágazóbb feladata.

A programozónak sokrétű ismeretekkel kell rendelkeznie, hogy a legmegfelelőbb matematikai modellt építse fel. Természetesen a feladattól függően lehet, hogy utána kell nézni a numerikus analízis megfelelő módszereinek.

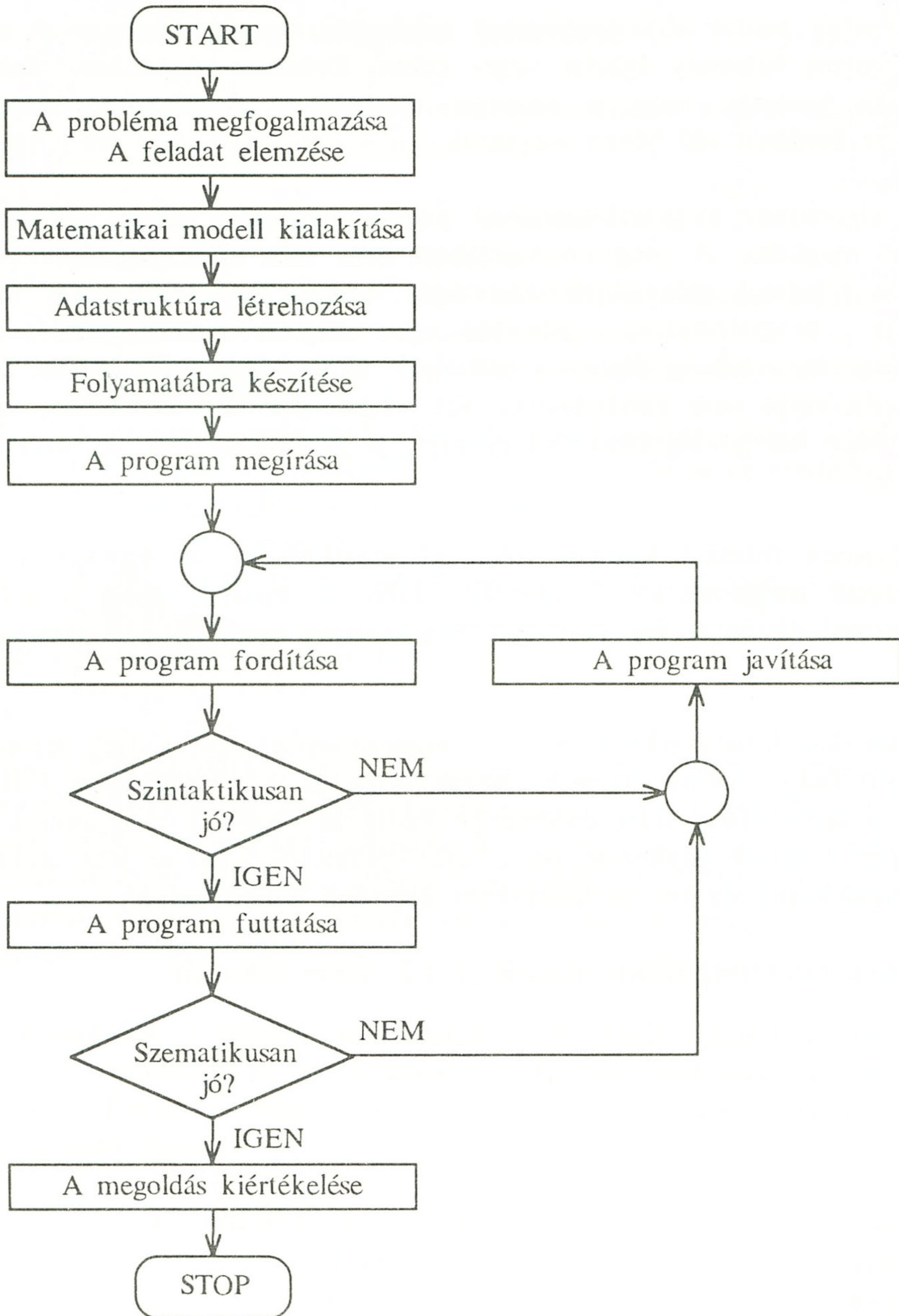
Az első négy pontot *algoritmizálásnak* nevezzük. Az *algoritmus* olyan előírás, amely szerint valamely feladat véges számú lépésben megoldható. Sok éves tapasztalat mutatja, hogy a program írásánál a jó algoritmus megszerkesztésére fordított idő bőven megtérül.

Ha az algoritmus megszerkesztésével készen vagyunk, akkor következik a program megírása. A program valójában nem más, mint az algoritmusban megadott feladatok szövegének számítógép számára érthető formája. Itt már az adott programnyelv - általában igen szigorú - szabályainak pontos betartására van szükség. Éppen a szabályok szigorúsága, valamint az, hogy a számítógép maga nem gondolkozik, csak végrehajtja az utasításokat, mutatja azt is, hogy bár jó algoritmusból könnyű jó programot írni, de azért lehet rosszat is.

A programok írásánál kétfajta hibát követhetünk el, az egyiket a nyelv szabályainak megsértésével (szintaktikai hiba), a másikat pedig a helytelen algoritmussal illetve elvileg helytelen programlépés eredményével (szemantikai hiba).

A szintaktikai hiba eredményeképp a programunkat nem tudjuk lefordítani, azaz nem tudjuk olyan állapotba hozni, hogy az a számítógépen futtatható legyen, a szemantikai hiba eredménye pedig az, hogy a programunk futni fog ugyan a számítógépünkön, de a futás eredménye nem az lesz, mint amit a mi szándékaink szerint eredményként kapnunk kellett volna.

A program megtervezésének lépéseit az 1.2. ábrán láthatjuk.



1.2. ábra A program tervezésének lépései

Példaként vizsgáljuk meg egy levél feladásának algoritmusát.

START
Boríték vétele.
Boríték megcímzése.
A levél összehajtása.
A levél borítékba tétele.
Bélyeg van?
ha nincs, bélyeg vétele
Bélyeg borítékra ragasztása.
A levél feladása.
STOP

Tekintsük meg a levélfeladás algoritmusának blokkdiagramját az 1.3. ábrán .

1.3. Mintafeladat

Nézzünk meg egyszerű numerikus feladatot, amely adatokat olvas és eredményt ír ki.

Feladat: Az *a* és *b* oldal ismeretében számítsuk ki a téglalap területét.

Megoldás: A feladat megoldásához ismerni kell az *a* és *b* oldal adatait és a téglalap területét meghatározó matematikai képletet.

A téglalap területe az *a* és *b* oldal ismeretében:

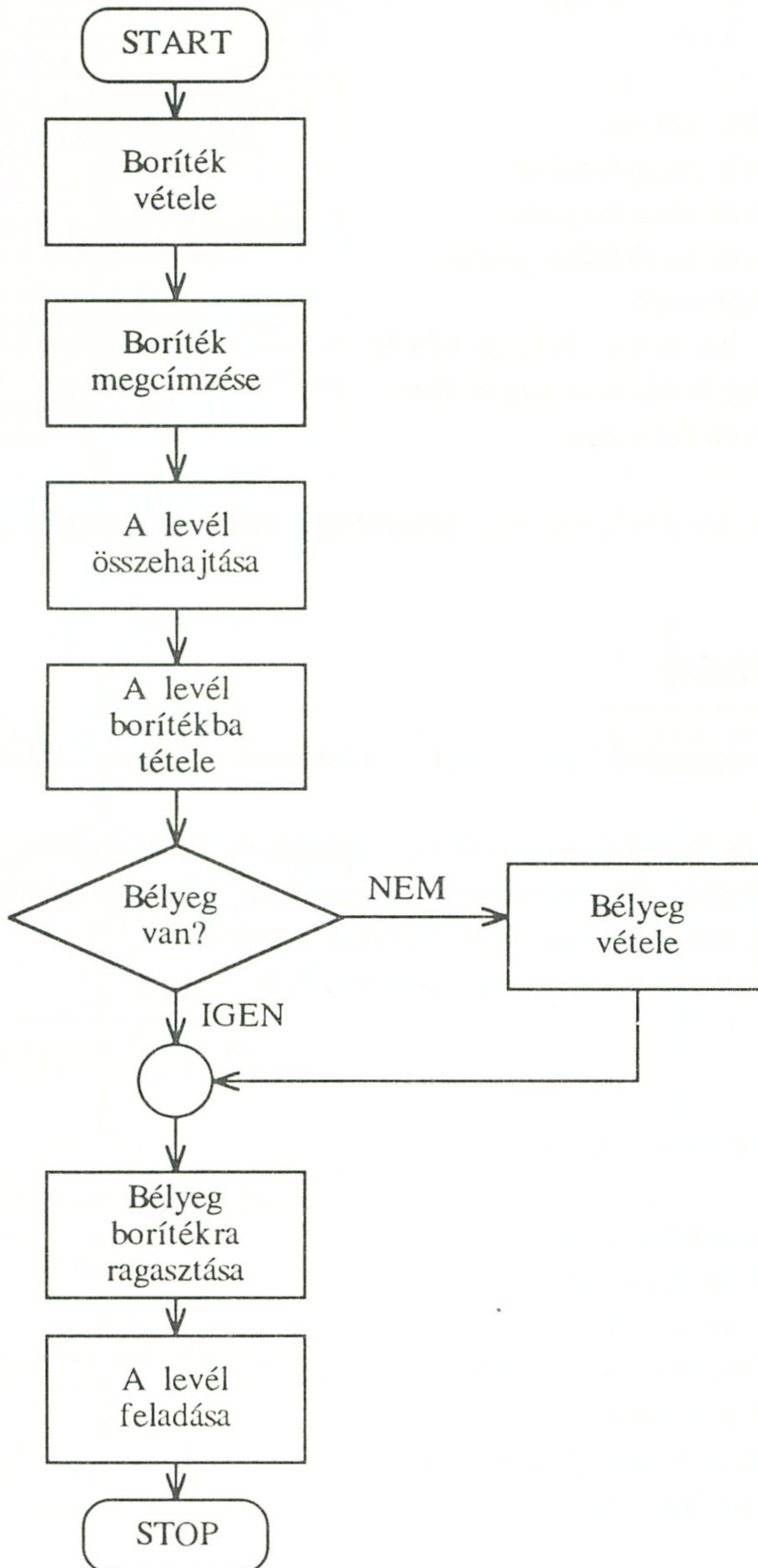
```
t = a. b
```

Algoritmus:

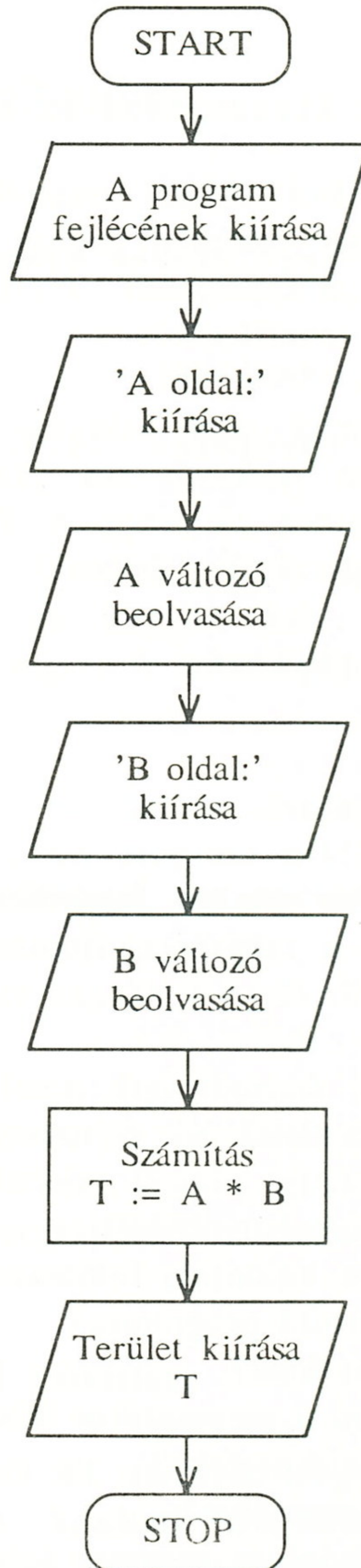
Bontsuk lépésekre a feladatot

START
Irja ki a program fejlécét.
Az a oldal beolvasását jelezze.
Olvassa be az *a* adatot.
A b oldal beolvasását jelezze.
Olvassa be a *b* adatot.
Számítsa ki a téglalap területét.
Irja ki az eredményt.
STOP

A program algoritmusának blokkdiagramja a 1.4. ábrán látható.



1.3. ábra A levélfeladás algoritmusa



1.4. ábra A téglalap program blokkdiagramja

A feladat Pascal programja:

```
(* demo.pas *)
{ a teglalap teruletenek kiszamitasa az a es b oldalbol }
program teglalap;
var a,b,t: real;
begin
     writeln('A teglalap teruletenek szamitasa');
     writeln;
     write('a oldal: '); readln(a);
     write('b oldal: '); readln(b);
     t:=a*b;
     writeln('A teglalap terulete: ',t:6:1);
end.
```

A program futási eredménye:

```
A teglalep teruletenek szamitasa

a oldal: 12.5
b oldal: 10
A teglalap terulete:  125.0
```

A mintaprogramok és a játékprogramok fejezetben bonyolultabb algoritmusok leírárával is megismerkedünk.

Ellenőrző kérdések:

1. Mit értünk programozás alatt?
2. Miből áll a program?
3. Mire szolgál a programozási nyelv?
4. Milyen célból készítenek különféle felhasználói nyelveket?
5. Miért van szükség a fordító programra?
6. Mit jelent a szintaktikus hiba?
7. Jelzi-e a fordító program a szemantikus hibát?
8. Milyen jellegű a szemantikus hiba?
9. Mire használható a folyamatábra?
10. Egy bonyolultabb számítógépes program tervezéséhez milyen lépéseket kell megtenni?
11. Mit jelent az algoritmizálás?

2. GONDOLATOK A PASCAL NYELVRŐL

2.1. A programokról általában

A számítógép egy nagyon hasznos segédeszköz lehet az élet szinte minden területén. Az viszont, hogy milyen hasznos, nagyon erősen függ attól, hogy milyen programmal (*szoftver*) rendelkezünk, amellyel a számítógépet használjuk.

Önmagában a számítógép semmire sem képes, ugyanis csak azt tudja végrehajtani, amire utasítjuk, ezt azonban nagyon pontosan és gyorsan gondolkodás nélkül végzi el. A gyors és pontos munkához cserébe viszont nekünk is pontosnak kell lennünk. Tudomásul kell vennünk, hogy a számítógép mit ért meg, és hogyan kell leírnunk, hogy azt a mi elgondolásunk szerint hajtsa végre. A számítógép csak parancsokat hajt végre, vagy programokat futtat.

Ebben a bevezető fejezetben röviden és olvasmányosan ismerkedik meg a Pascal programozási nyelvvel kis program példákon keresztül az az olvasó, akik még semmilyen programozási nyelvet nem ismert ezideig. Azonban ajánljuk azoknak is, akik második nyelvként a Pascalt választották és ebben a fejezetben egy átfogó képet kapnak a nyelvről.

A további fejezetek részletesen foglalkoznak a szükséges elméleti kérdésekkel, példákon keresztül mutatják be az utasítások használatát. A függelékben adjuk meg a gyakorlatként kiadott feladatok megoldását. A könyvben hivatkozott összes program megtalálható a lemezmellékleten.

2.2. A Pascal nyelvről röviden

Minden program utasításokból áll, amelyik megmondja pontosan, hogy mi is történjen. Ha egy programot kezdünk írni, akkor pontos szabályok állnak a rendelkezésünkre, hogy milyen utasításokat használhatunk, azokat pontosan hogyan kell leírni. Például, ha egy Pascal programban ki akarjuk írni a képernyőre, hogy "Rendben" akkor ezt a következő formában kell

megtennünk:

```
Write('Rendben');
```

A **Write** eljárás szolgál arra, hogy az általunk kívánt szöveg a képernyőn jelenjen meg. Minden utasításnak megvan a maga szintaktikája. Nézzük meg, hogy a **Write** eljárást ebben az esetben hogyan kell használni.
A **Write** után közvetlenül egy zárójelnek kell következnie, és a zárójelben aposztróf jelek között legyen a szöveg, amit ki akarunk írni, majd bezárjuk a zárójelet. Az utasítást pontosvessző zárja. Ha ebben az utasításban bármit másképpen csinálunk, például hibásan írjuk le a **Write** kulcsszavat, vagy elhagyunk egy zárójelet vagy egy aposztrófot, akkor a fordítóprogram hibát fog jelezni és a programot nem tudjuk futtatni, míg a jelzett hibát ki nem javítjuk. Ha csak a szöveget gépeltük be rosszul, akkor természetesen a program lefut, de az elgépelt szöveg fog megjelenni.

A másik olyan fogalom amelyikkel minden programban találkozunk, az a változók fogalma. A változó fogalmát egy példán keresztül lehet talán a legjobban megérteni. Ha két számot össze akarunk adni, ahhoz a programban olyan utasítást kell írni, hogy a két szám összegét számítsa ki, azaz az egyik számhoz adja hozzá a másik számot. Egy Pascal programban ezt egy értékadó utasítással végezhetjük el, amely a következőképpen néz ki:

```
osszeg:=egyik+másik;
```

Ebben a sorban három változó szerepel, rendre az *osszeg*, az *egyik* és a *másik*. Minden változónak van egy neve (ezt a változó azonosítójának szokták nevezni), jellemzi a típusa (ez lehet például egész szám, valós szám, szöveg stb.), és a típustól függően tartalmaz egy értéket (például az összeg értéke lehet 5 vagy 319.43 is).

2.3. A Pascal program szerkezete

Maga a Pascal program három egymástól jól elkülöníthető részből áll.
Ezeknek a részeknek a neve a következő:

programfej
definíciós illetve deklarációs rész
programtörzs

Az egyszerűbb megértés kedvéért hátulról visszafelé nézzük meg, hogy melyik rész mire szolgál.

A programtörzs tulajdonképpen a program legérdekesebb része. Ebben a részben írjuk le a program utasításait, azaz itt mondjuk meg, hogy a programunk mit hajtson végre, valamint azt is, hogy milyen módon.

A definíciós ill. a deklarációs részben mondjuk meg a fordítóprogramnak, hogy milyen változókat fogunk használni, illetve itt írjuk le az úgynevezett eljárásokat és függvényeket is.

A programfej a Pascal programokban csak egy sor, amelyik a **program** kulcsszóval kezdődik, és utána a program azonosítója következik.

2.4. Az első Pascal program

Most nézzük meg a fenti fogalmakat néhány igen egyszerű program segítségével. Az első program kizárólag példaként szolgál egy legrövidebb Pascal programra, amelyik kiírja a képernyőre, hogy "Rendben". A program olyan egyszerű, hogy még a deklarációs rész is hiányzik belőle.

Ajánlatos a mintaprogramok kipróbálása céljából egy alkönyvtárat létrehozni. Az **md** (*make directory*) parancs hozza létre az alkönyvtárat, melynek a neve 8 karakternél nem lehet hosszabb. Legyen a neve proba:

C:\>**md proba**

Lépjünk be az alkönyvtárba a **cd** (*change directory*) paranccsal:

C:\>**cd proba**

C:\PROBA>_

Bent vagyunk a PROBA alkönyvtárban, a villogó kurzor a további parancsok megadását várja. Mivel a Turbo Pascal kényelmes fejlesztői környezetében programozunk indítsuk el a **turbo** programot:

C:\PROBA>**turbo**

A **turbo** indításával beléptünk a Turbo Pascal integrált környezetébe, ekkor a menürendszeren keresztül elérhető minden szükséges eszköz: a szövegszerkesztő a program írásához, a Pascal fordító, a futtató és a hibakereső program.

Az **F3** funkció gomb megnyomására a *Load file név* ablakban a programunk számára adunk egy file nevet. A szövegszerkesztő a megadott névvel és .PAS kiterjesztéssel elhelyezi abban az alkönyvtárban, ahonnan a **turbo** programot indítottuk el, kivéve, ha nem váltottunk könyvtárat a menüből. Legyen az első programunk neve

legelso

és *Enter*-t ütve a *legelso.pas* file számára a szerkesztőbe kerülünk és a villogó kurzor mutatja a bal felső sarokban a programunk első karakterének helyét és várja, hogy gépeljünk.

```
program legelso;
begin
    Write('Rendben');
end.
```

Ha a program begépelésével készen vagyunk, akkor a *Save* almenüvel vagy az *F2* funkció gomb leütésével a kijelölt nevű, példánkban a *legelso.pas* néven a program tárolásra kerül a lemezen.

A program rövid ugyan, de néhány dolgot jól be lehet rajta mutatni. Egyik fontos dolog, hogy az utasítások majdnem kivétel nélkül pontosvesszővel végződnek. A másik, amiről idáig nem volt szó, az a **begin** és **end** utasítások. Ezek az ún. utasítás zárójelek, a tulajdonképpeni feladatuk az, hogy utasítások csoportjait egybefogják. A példa nem erre szolgált, hiszen egy utasítást nem lehet igazán összefogni, azonban a Pascalban a főprogram egy blokk, amely mindig **begin**-nel kezdődik és a program utolsó utasítása mindig az **end.**, amelyet az általánostól eltérően nem pontosvessző, hanem pont zár le.

A program fordítása és futtatása a *Run* menüben történik. Ha a programot már begépeltük, a fordítást és a futtatást kétféleképpen is végrehajthatjuk, vagy az F10 funkció gombbal fellépünk a főmenübe és kiválasztjuk a *Run* almenüpontot, vagy a Ctrl F9 ún. forró gombbal (a két billentyűt együtt kell lenyomni úgy, hogy a Ctrl gomb nyomvatartása mellett üssük le az F9-et) mindezt azonnal végrehajthatjuk. Ha a fordítóprogram hibát talál, akkor ezt a hibakódjával szövegesen is jelzi. Ilyen esetben a program utasításait újra

gondosan vizsgáljuk át, javítsuk ki a megtalált hibát és újra adjuk ki a Ctrl F9 parancsot. Ha a fordító hibátlannak találta a programot, akkor azt az integrált keretrendszer le is futtatja.

A program eredménye a felhasználói (ún. DOS) ablakban jelenik meg. Ez az ablakváltás olyan gyors, hogy szemmel nem lehet követni. A program futása után ismét a szövegszerkesztőben találjuk magunkat. Ez jó jel, hiszen a program lefutott, csak még az eredményt nem láttuk. A felhasználói ablakot, tehát ahová az eredmény íródott az ALT F5 gombok egyidejű lenyomásával nézhetjük meg és bármely billentyű leütésével visszatérhetünk a Turbo Pascal integrált környezetébe és folytathatjuk a programunk bővítését, módosítását, vagy új program írását.

Ha a program eredménye elfér egy képernyőn, érdemes beállítani az *Output* ablakot. Ennek a beállítása a következő: az **F6** funkció billentyűvel átlépünk a *Watch* ablakba, majd az **ALT F6** együttes megnyomásával beállítjuk a *Watch* ablak helyén az *Output* ablakot. Így a program szövege alatt kényelmesen kiértékelhető a lefuttatott program. Az *Output* ablakot is görgethetjünk a kurzor nyíl billentyűkkel. Az *Edit* és az *Output* ablak közötti oda-/visszaváltás az **F6** billentyűvel történik (Turbo Pascal 5.0 és 5.5 verzióknál).

2.4.1. A változók deklarálása

A következő program csak annyiban térjen el az előzőtől, hogy a szöveget egy *szoveg* nevű változóba írjuk:

```
Program masodik;
var szoveg  :string;
begin
           szoveg:='Rendben';
           Write(szoveg);
end.
```

Itt először nézzük meg valamivel részletesebben a program második sorát. Ez az úgynevezett változódeklaráció.

A változódeklaráció arra szolgál, hogy megmondjuk a programnak, hogy milyen változókat fogunk használni. A Pascalban minden változót deklarálni kell. A deklaráció mindig a **var** kulcsszóval kezdődik, ezt követi a változó azonosítója, majd a változóazonosítótól kettősponttal elválasztva következik a változó típusa, végül pedig az utasítászáró pontosvessző. Itt az alkalom, hogy megnézzük a leglényegesebb egyszerű változótípusokat. (Azért csak ezeket, mert a Pascal igen gazdag lehetőségeket kínál a változók deklarációjára, viszont az érthetőség kedvéért jobb mindent a maga idejében megemlíteni.)
Az öt alapvető változótípus a következő:

integer - egész szám -32768..32767 között
real - valós szám $2.939*10^{-39}..1.701*10^{38}$ között
char - egy karakter (pl. betű vagy valamilyen jel az ASCII kódtábla szerint.)
string - egy legfeljebb 255 karakter hosszúságú idézet
boolean - logikai változó, amelynek az értéke csak igaz (ezt *true*-nak írjuk) vagy hamis (*false*) lehet;

A másik újdonság ebben a programban a program negyedik sorában szereplő úgynevezett értékadó utasítás. Általában ezzel az utasítással adunk egy adott nevű változónak értéket. Az ilyen utasításoknak a bal oldalán mindig egy változó neve szerepel, ez lesz az a változó, amelynek az értéket adunk. Ezt a nevet a := jel követi. Az értékadásban ez a két jel összetartozik és mindig ezt kell használni. Ezután következik az az érték, ami a változó új értéke lesz.

Példánkban tehát a *szoveg* nevű változó értéke 'Rendben' lesz. Ennél az értékadásnál meg kell még jegyezni, hogy ha **string** típusú változó értékét mindig '' jelek közé kell tennünk, ugyanis különben a program változónévnek veszi azt, és például a

```
szoveg:=Rendben;
```

programsor esetén, a *szoveg* nevű változó értékét meg akarja változtatni úgy, hogy az a *Rendben* nevű változó értékével egyezzen meg, ami természetesen hiányzik a programból.

Bonyolítsuk egy kicsit tovább ezt a programot:

```
program harmadik;
var  i,j  : integer;
     x    : real;
begin
     i:=9;
     j:=6;
     x:=i/j;
     Write(x);
end;
```

Ez a program az előzőtől először is annyiban különbözik, hogy több változót deklaráltunk. Itt lehet látni, hogy több azonos típusú változó deklarációja úgy történik, hogy vesszővel elválasztva felsoroljuk a változók azonosítóit, majd ezt követi a típus megadása.

A program második különbsége az utolsó értékadásban szereplő kifejezés, azaz az hogy itt az *x* változónak nem egy konkrét értéket adunk, hanem kiszámíttatjuk az *i* és *j* változók hányadosát.

Idáig tehát tudunk változót deklarálni, annak értéket adni, illetve azt kiírni. de még nagyon hiányzik az, hogy egy változó értékét menet közben adjuk meg. Ezt a beolvasó utasítással érhetjük el.

Bővítsük ki az előző programot úgy, hogy az *i* értékét a klaviatúráról olvassuk be:

```
program negyedik;
var  i,j  : integer;
      x   : real;
begin
     ReadLn(i);
     j:=6;
     x:=i/j;
     Write(x);
end.
```

Könnyen kitalálható, hogy a ***ReadLn***(i) utasítás szolgál arra, hogy az *i* változóba a klaviatúráról értéket olvassunk be.

Ezzel a programunk már teljessé vált, hiszen egyik rész sem hiányzik belőle. Megvan a programfej, a deklarációs rész, a programtörzs, valamint a programtörzsön belül van beolvasás, számítás és kiírás. Most tekintsük át az utóbbiakat külön-külön.

2.4.2. Olvasás billentyűzetről

Itt két eljárást kell megemlíteni. Az egyik a ***Read*** a másik a ***ReadLn***. Szintaktikájuk (a programbeli használatuknak illetve leírásmódjuknak szabálya) a következő:

> ***Read***(*változóazonosító*);
> illetve
> ***ReadLn***(*változóazonosító*);

Mindkét eljárás a megadott azonosítójú változó értékét teszi egyenlővé a klaviatúráról beolvasott értékkel, a különbség mindössze annyi, hogy a ***ReadLn*** eljárás az *Enter* billentyűig kiolvassa a beírtakat, és az *Enter*-t figyelmen kívül hagyja.
A különbség érzékeltetésére nézzünk meg két rövid programot:

```
program beolvas1;
var ch1,ch2  : char;
begin
     Read(ch1);
     Read(ch2);
     Write(ch1,ch2);
end.
```

Ha ezt a programot lefuttatjuk, és a klaviatúrán egy ***a*** betűt és utána egy *Enter*-t ütünk, akkor a program a *ch1* változóba beolvas egy karaktert, ez lesz az *a* betű, a *ch2* változóba beolvas egy másik karaktert, ez lesz az *Enter*, majd ezeket kiírja és megáll.

Ha viszont a két ***Read*** eljáráshívást ***ReadLn***-ra cseréljük, akkor a program viselkedése megváltozik.

Nézzük meg így a programot:

```
program beolvas2;
var ch1,ch2  : char;
begin
     ReadLn(ch1);
     ReadLn(ch2);
     Write(ch1,ch2);
end.
```

Tegyük ugyanazt, mint az előbb, üssünk le egy *a* betűt és egy *Enter*-t. Ennek hatására a gép végrehajtja az első ***ReadLn*** utasítást, azaz a *ch1* változóba beírja az *a* betűt, az *Enter*-t pedig - valamint ha valamit írtunk az *a* betű és az *Enter* közé - elfelejti. Most le kell ütnünk mondjuk a *b* billentyűt és az *Enter*-t, ennek hatására a *ch2* értéke *b* lesz és a képernyőn megjelenik a program erdeménye, az

```
ab
```

kiírás.

A legnagyobb keveredést a ***Read*** működése okozhatja, ugyanis ha a program egy ***Read*** utasításhoz ér akkor megáll, és várakozik mindaddig, ameddig jelzést nem kap, hogy a kért értéket a klaviatúráról leolvashatja. Ez a jelzés pedig éppen az *Enter* leütése, azaz ameddig *Enter*-t nem ütünk, addig a változó értékével semmi sem történik. Igen ám, de ahogy az első példában is látszott, ekkor a változóba betöltődik az adott érték, és legalább egy *Enter* ottmarad, és összezavarhatja a következő beolvasást.

A ***Read*** illetve a ***ReadLn*** eljárásokkal egyszerre több változót is beolvashatunk.
Ekkor a szintaktikájuk a következő lesz:

Read(*változó1,változó2, ... ,változón*)
illetve
ReadLn(*változó1,változó2, ... változón*)

Karakter típusú változó esetén egyértelműen az *n* darab változóba az első *n* darab karakter kerül. Numerikus (**real**, **integer**) változók esetén az egyes

értékeket egymástól el kell választani, ugyanis pl. ha három **integer** típusú változóba olvasunk be és a klaviatúrán az "1234" számsorozatot írjuk le, akkor nem lehet eldönteni, hogy az vajon 1 2 34, 1 23 4, vagy 12 3 4 akart lenni.
Az értékek elválasztása a szóköz vagy az *Enter* billentyűvel történik. Numerikus érték beolvasásánál valamilyen számot mindenképpen be kell írnunk, ugyanis ha csak szóköz illetve *Enter* billentyűt ütünk le, akkor a gép tovább várakozik, egyéb, nem szám leütése esetén pedig hibát jelez és a programunk leáll.

2.4.3. Kiírás képernyőre

Megint két eljárást kell ismertetnünk: a ***Write*** és a ***WriteLn***.

Ezen eljárások hívása a következő:

> ***Write***(*kiírandó*);
> illetve
> ***Writeln***(*kiírandó*);

A két eljárás között mindössze annyi a különbség, hogy az első használata esetén a következő kiírás közvetlenül az előző kiírást követi, a második utasítás használata esetén a legközelebbi képernyőre íráskor pedig új sor kezdődik.
Most nézzük meg, hogy mit és milyen formában tudunk kiírni, azaz mi szerepelhet a zárójelek között levő "kiírandó" helyén:
először is, miként már láttuk szerepelhet " (aposztrófok) közé zárt szöveg, ebben az esetben a szöveg kerül kiírásra, aposztrófok nélkül.
Például a

```
WriteLn('Rendben');
```

utasítás hatására a képernyőn a

```
Rendben
```

szöveg jelenik meg.

Láttuk azt is, hogy a "kiírandó" helyén szerepelhet változóazonosító is azaz, ha a *szoveg* egy **string** típusú változó akkor a

```
szoveg:='Rendben';
WriteLn(szoveg);
```

programrészlet hatása az előzővel megegyező.

Természetesen nem csak szövegeket írhatunk ki, hanem számokat is. A

```
WriteLn(56);
```

utasítás eredménye a képernyőn

```
56
```

és ugyanígy ha i **integer** típusú, akkor a

```
i:=56;
WriteLn(i);
```

hatására ugyancsak

```
56
```

jelenik meg a képernyőn. Ha ezek után, a sikereken felbuzdulva kipróbáljuk a

```
WriteLn(3.14);
```

utasítást, joggal várhatjuk, hogy az eredmény

```
3.14
```

lesz.
Ehelyett azonban a képernyőn

```
3.1400000000E+00
```

jelenik meg. Ennek az az oka, hogy ha egy számban tizedespont is van, akkor azt a Pascal automatikusan **real** típusúnak veszi, és olyan formát keres amelyikben minden **real** típusú számot képes kiírni, ez a forma pedig a szám úgynevezett normál alakja. Ennek az első része tartalmazza a szám értékes jegyeit, a második pedig azt, hogy a tíz hányadik hatványával kell ezt a számot szorozni. Például a 123.456 ebben a formában 1.2345600000E+02. A számítógép szempontjából ez a módszer nagyon praktikus, az egyszerű felhasználó azonban nehezen tudja elviselni - így

természetesen létezik jobb megoldás is. Ezt úgy hívják, hogy a kiírási formátum beállítása. Ismét egy példával szemléltetve, ha *x* **real** típusú változó akkor a következő programrészlet:

```
x:=3.14;
WriteLn(x:6:2);
```

hatására *x* értéke összesen 6 karakteren lesz kiírva, és ezen belül 2 karaktert foglalnak le a tizedesjegyek, azaz a kiírt érték:

```
3.14
```

lesz. Magától érthetődik, hogy nem csak a **real** típusú számokat írhatjuk ki adott hosszúságon, hanem más típust is. A

```
szoveg:='Rendben';
WriteLn(szoveg:12);
```

programrészlet hatására a "Rendben" kiírás 12 karakteren jelenik meg, úgy hogy először 5 szóközt látunk, majd utána a szöveget. Itt természetesen a második kettőspont a hozzá tartozó számmal együtt fölösleges, sőt hiba, hiszen az a tizedesjegyek számát határozza meg, az pedig kizárólag **real** típusú számok esetén létezik.
Ugyanúgy ahogy a ***Read*** illetve ***ReadLn*** eljárásnak több beolvasandó változója lehet, a ***Write*** illetve a ***WriteLn*** eljárás képes több értéket is kiírni, ekkor a kiírandó értékeket vesszővel választjuk el.

Természetesen itt is figyelni kell, mert például az alábbi rövid program:

```
program hibasiras;
var a,b : integer;
begin
     a:=11; b:=22;
     Writeln(a,b);
end.
```

eredménye a képernyőn

```
1122
```

lesz, és nem lehet látni, hogy hol a két szám között a határ.

Ugyanez a program a

```
Writeln(a:3,b:3);
```

utasítás hatására

```
 11 22
```

formájú eredményt szolgáltat. Nézzük meg most az előbbiekben leírt negyedik példaprogramot kicsit kibővítve ***Read*** illetve ***Write*** eljárásokkal.

```
program irolvas;
var i,j  : integer;
    x    : real;
begin
     Write('Az osztando:');
     ReadLn(i);
     j:=6;
     x:=i/j;
     WriteLn('Az oszto:6');
     Writeln('Az eredmény:',x:5:2);
end.
```

A programot elindítva a számítógép kiírja a képernyőre a következő üzenetet:

```
Az osztandó:
```

és várakozik. Ezután be kell gépelnünk egy **integer** típusú számot és egy *Enter*-t. Ez a szám közvetlenül a kettőspont után jelenik meg, mivel **Write** eljárást használtunk.

```
Az osztando:9
```

Tételezzük fel, hogy a 9 számot írtuk be. az *Enter* leütése után a gép végrehajtja a hátralevő utasításokat és kiírja:

```
Az osztó:6;
Az eredmény: 1.50
```

Érdemes megfigyelni, hogy az idézeten belül (*'Az osztó:6'*) szereplő :6 nem azt vezérli, hogy milyen hosszon kerül az idézet kiírásra, mert ehhez a kiírandó idézet után kellene állnia, hanem az idézet részeként lesz kiírva.

2.4.4. Számítás

Értelemszerűen ez a program legösszetettebb része, hiszen programunk itt végez el minden olyan számítást, amiért létrehoztuk, a többi rész (deklaráció, beolvasás, stb ...) csak kiegészítés, bár feltétlenül szükséges. A számításoknál először az úgynevezett értékadó utasításokkal kell foglalkozni, amelyek segítségével változóknak lehet az értékét megadni. Értékadó utasításra példát már láttunk ebben a részben, hiszen gyakorlatilag képtelenség nélküle programot írni.

Most nézzük meg az értékadó utasításokra vonatkozó legfontosabb szabályokat.
Az értékadó utasítás formája mindig

változóazonosító:=kifejezés;

ahol a *változóazonosító* annak a változónak az azonosítója, amelynek értéket akarunk adni, a := jel azt jelenti, hogy a jel bal oldalán álló változó a jel jobb oldalán lévő kifejezés értékét kell hogy felvegye.

Itt a következőkre kell nagyon figyelni. Az első, nem az értékadás formájával összefüggő szabály az, hogy minden változónak adjunk értéket. Ez nyilvánvalónak tűnik, mégis nagyon sokan elrontják, mert feltételezik, hogy a változó értéke a program futásának kezdetén pl. numerikus változó esetén nulla. Ez nagyon súlyos hiba, és adott esetben nehezen megtalálható hiba, mert a gép a deklarációs részben létrehozza a változót, és értékül azt adja, ami éppen azon a memóriacímen volt.
A másik, már formai szabály az, hogy egy értékadó utasítással csak egy változónak adhatunk értéket, azaz például ha az *a*, *b*, *c* és *d* változónak is nullát akarunk értékül adni, akkor azt csak négy utasítással tudjuk megtenni:

```
a:=0;  b:=0;  c:=0;  d:=0;
```

(megjegyzendő, hogy nem hiba, ha ezt a négy utasítást egy sorba írjuk).
A harmadik fontos szabály az, hogy néhány esettől eltekintve az értékadó utasítás jobb és bal oldalán álló kifejezésnek azonos típusúnak kell lennie,

azaz például az egyik leggyakoribb hibát az alábbi programban nézhetjük meg:

```
Program hiba1;
var i,j,k  : integer;
begin
      Write('i:');
      ReadLn(i);
      Write('j:');
      ReadLn(j);
      k:=i/j;
      WriteLn('k=',k:5);
end.
```

Ezt a programot nem tudjuk lefuttatni, ugyanis az osztás eredménye, azaz az értékadó utasítás jobb oldalán álló kifejezés értéke mindig **real** típusú, függetlenül attól, hogy a kifejezés értékének tört része nulla-e vagy sem. Ilyen hibával már eleve el sem indul a program, mivel nem lehet a programot lefordítani. A kritikus sort helyesen a

```
k:=round(i/j);
```

formában írhatjuk, ahol a round(i/j) az i/j kifejezést a kerekítés szabályai szerint kerekíti, és **integer** típusúvá alakítja.

Az utolsó, ami ide kívánkozik, az értékadó utasítás működésével kapcsolatos. Az értékadó utasítás a változó értékét olyan módon határozza meg, hogy először kiszámolja a jobb oldalon álló kifejezés értékét, és a bal oldalon álló változó ezután veszi fel ezt az értéket. Ez nyilvánvalónak látszik, és azt hogy miért kell kiemelni, a következő utasítás mutatja meg:

```
a:=a/2+b;
```

Itt (és ez eléggé gyakori) az *a* változó szerepel az utasítás bal, valamint a jobb oldalán is, és beláthatatlan következményekkel járna, ha az *a* értéke a kifejezés értékének kiszámítása alatt megváltozhatna.

Az eddigiek segítségével már írható komplett működőképes program, de ezek a programok még semmi másra nem jók, csak arra, hogy egy egyszerű számítást elvégezzenek.

Legyen most a feladat a signum függvény meghatározása, azaz olvasson be a program egy **real** típusú számot, és ha a szám pozitív, akkor a *sign(x)=1*, ha a szám negatív, akkor a *sign(x)=-1*, ha pedig nulla akkor a *sign(x)=0* üzenetet írja ki.

A feladat megoldása nem okoz nagy gondot, hiszen ha *x* értéke nullától eltér akkor a *sign(x)* értéke az *abs(x)/x* kifejezéssel kiszámítható. A problémát az jelenti, hogy ha *x* értéke nulla akkor ezt a műveletet a számítógép nem tudja elvégezni, és hibaüzenettel leáll. Ahhoz hogy ezt elkerüljük, rendelkezésünkre áll az úgynevezett feltételes elágazás.

Ekkor egy feltételtől függően hajt végre a gép egy elágaztatást, vagy pedig a következő két utasítás valamelyikét. Ennek az utasításnak a formája:

if *feltétel* **then** *utasítás*;

Ebben az esetben a **then** kulcsszó után álló utasítás csak akkor hajtódik végre, ha az **if** és **then** között álló feltétel teljesül. A másik esetben

if *feltétel* **then** *utasítás1*
else *utasítás2;*

Itt az **if** és **then** között álló feltétel teljesülése esetén csak az *utasítás1* hajtódik végre, az *utasítás2* nem, ha pedig az említett feltétel nem teljesül, akkor csak az *utasítás2* hajtódik végre, az *utasítás1* nem. Nézzük meg ezeket az előbb említett *signum* függvény kapcsán:

```
Program signum1;
var a,sign  : real;
begin
     Write('A szám:');
     ReadLn(a);
     sign:=0;
     if a<>0 then sign:=abs(a)/a;
     WriteLn('sign(x)=',sign:3:0);
end.
```

illetve

```
Program signum2;
var a  : real;
begin
     Write('A szám:');
     ReadLn(a);
     if a=0 then  WriteLn('sign(x)= 0.')
            else  WriteLn('sign(x)=',abs(a)/a:3:0);
end.
```

Az első programban a *sign(x)* értékét nullának feltételeztük, és ha a beolvasott szám nullától eltért, akkor változtattuk meg az értékét. A második esetben nem feltételeztünk semmit, hanem a beolvasott szám értékétől függően írtuk ki *sign(x)* értékét. Fontos formai szabály, hogy a pontosvessző az egész **if** utasítást zárja le, tehát második forma használata esetén az **else** kulcsszó előtt semmiképpen sem szerepelhet pontosvessző.

A feltételes elágazásoknak a Pascal ismeri egy másik formáját is. Az **if** utasítás ugyanis legfeljebb kétirányú elágazást enged meg, hiszen egy feltétel mindig egy logikai kifejezés (**boolean** típus) amely vagy igaz, vagy hamis. Kézenfekvő, hogy legyen egy olyan elágazás is, amely más típus esetén is lehetővé teszi, hogy egy kifejzés értékétől függően hajtódjon végre a következő utasítás.
Ennek formája

case *kifejezés* **of**
érték1 : utasítás1;
érték2 : utasítás2;
...
értékn : utasításn;
else *utasitás*
end;

Az utasítás használatát ismét egy példán mutatjuk be. Irjunk programot, amely egy pont koordinátáit olvassa be, és kiírja, hogy a koordinátarendszer melyik negyedébe esik a pont, illetve, ha a pont éppen valamelyik tengelyre esik, azt is jelzi. Az *x* és az *y* változók tartalmazzák a pont kordinátáit. Ha a pont egyik tegelyre sem esik, akkor

(sign(x)+1)*10+(sign(y)+1) értéke rendre a 22, 20, 0 illetve a 2 értéket

veheti fel. Nézzük most a programot:

```
program siknegyed;
var x,y         : real;
    signx,signy : integer;
begin
    Write('x=');
    ReadLn(x);
    Write('y=');
    ReadLn(y);
    if x=0 then signx:=0
          else signx:=round(abs(x)/x);
    if y=0 then signy:=0 else signy:=round(abs(y)/y);
    case (signx+1)*10+(signy+1) of
         22 : WriteLn('Elsô síknegyed');
         20 : WriteLn('Második síknegyed');
          0 : WriteLn('Harmadik síknegyed');
          2 : WriteLn('Negyedik síknegyed');
    else WriteLn('A pont éppen tengelyre esik')
    end;
end.
```

A **case** utasítás használatánál a következőkre kell figyelni. Először, hogy a **case** kulcsszót követő kifejezés (ezt hívják az utasítás *szelektorának*) csak úgynevezett sorszámozott típusú lehet. Nem lehet például az eddig tárgyalt típusok közül **real** vagy **string**. A másik fontos szabály a kettőspontok és pontosvesszők használatára vonatkoznak. Az utasítás törzsében fel kell sorolni a szelektor azon az értékeit, amelyek szerint végre akarjuk hajtani az utasítást. A jelen példában, ha a szelektor értéke 22, 20, 0 illetve 2 akkor meghatározott szöveget kell kiírni. Bármely más értékre az **else** után álló szöveg kerül a képernyőre. A szelektor lehetséges értékeit rendre egy-egy kettőspont követi. A kettőspont után található az az utasítás, amelyet az adott érték esetén végre kell hajtani, majd minden egyes utasítást pontosvessző zár.

Az utolsó választási lehetőség az **else** ág. Ez csak akkor hajtódik végre, ha a szelektor egyik felsorolt értékkel sem egyezett meg. A lehetséges szelektorértékeknél alkalmazott szabállyal ellentétben az **else** szót nem követheti kettőspont, és az utána álló utasítást nem kell pontosvesszővel zárni. Az **if** utasítás szabályaival ellentétben a **case** utasításnál nem hiba, ha közvetlenül az **else** kulcsszó előtt pontosvessző áll. Ha a felsorolt

szelektorértékektől eltérő érték esetén nem akarunk semmit sem csinálni, akkor lehetőségünk van az **else** részt kihagyni. A **case** utasítást mindig **end** kulcsszóval zárjuk, ez jelzi, hogy a program ne keressen több szelektorértéket.

Az eddigi ismeretek birtokában elvileg már majdnem minden feladatot meg lehet oldani. A programozás alapvető részeiről szóló fejezethez még egy kiegészítés kívánkozik. Mint már jó néhányszor, most is nézzünk meg egy példát. Olvassunk be tíz számot, számítsuk ki az átlagát, és határozzuk meg az átlagtól való legnagyobb eltérést.

A program algoritmusa a következő:
Először beolvassuk a tíz értéket, és ugyanakkor össze is adjuk egymással. Ezután ezt az összeget elosztjuk tízzel, így megkapjuk az átlagot. Végül ismét végignézzük a számokat, és megvizsgáljuk az átlagtól való eltérést.

Most gondoljuk végig, hogy milyen változókat használjunk. Először is be kell olvasni a tíz számot. Ez az eddigiek alapján tíz ***real*** típusú változót jelentene, de ezt nagyon körülményes lenne kezelni, ezért egy, az eddigiekben nem szereplő változótípust fogunk használni, a tömböt. Ennek lényege, hogy például a példánkban szereplő tíz változót azonos azonosítóval, valamint egy sorszámmal jelöljük mégpedig olyan módon, hogy az első beolvasott szám a *szam[1]*, a második a *szam[2]* és így tovább. Természetesen, miután a tömb elemek csoportjából áll, a tömböt alkotó elemeknek van saját típusa, azaz tömböt képezhetünk **real**, **integer** illetve tetszőleges egyéb típusú elemekből is. A tömbváltozók deklarálása a deklarációs részben történik, mégpedig - a jelen példára hivatkozva - a következő formában:

```
var szam  : array [1..10] of real;
```

ahol a *szam* a tömb azonosítója, az **array** kulcsszó jelzi, hogy tömbváltozóról van szó, a szögletes zárójelek közé [ ] írt két szám pedig a lehetséges legkisebb és legnagyobb indexet jelöli ki. A megadás módjából kitalálhatjuk, hogy az index tetszőleges számtól tetszőleges számig terjedhet, így például nem hibás a

```
var tomb  : array [-9..100] of integer;
```

deklaráció sem. Ezután a kitérő után térjünk vissza a programunk változólistájának a megtervezéséhez. Az eddigiek alapján lesz egy tízelemű **real** számokból álló tömbünk. Ezenkívül kell egy változó, amelyik az átlagot fogja tartalmazni, ennek neve *atl* lesz, típusa pedig **real**;

A következő változónk pedig az eltérés értékét mutatja meg, ennek neve legyen *delta*, típusa pedig szintén **real**. A program írása során még egy változót fogunk használni, amelyik azt mutatja meg, hogy a tömbnek éppen hányadik elemével foglalkozunk (ezt segédváltozónak hívjuk, mert a számításokban közvetlenül nem vesz részt). A segédváltozónak az *i* azonosítót adjuk, és **integer** típusúnak deklaráljuk.

Mielőtt azonban a program írásához fognánk, térjünk vissza az algoritmushoz. A program első részében be kell olvasnunk 10 számot. Ezt az eddigi ismereteink szerint beolvashatjuk egyenként a következő programrészlet segítségével:

```
Write('-');
ReadLn(szam[1]);
Write('-');
ReadLn(szam[2]);
...
```

Ezt a megoldást azonban rögtön el is kell felejteni, annyira rossz. A tíz szám beolvasása ugyanis ugyanannak az utasításnak a tíz esetben történő megismétlését jelenti.

Ilyen ismétlés a programok írása során rendkívül sokszor előfordul, ezért az ilyen ciklikusan ismétlődő műveletekhez a programnyelvek segítséget nyújtanak. Ezeket az utasításokat hívjuk átfogó néven ciklusutasításoknak. A Pascal programnyelv háromféle ciklusutasítással rendelkezik, most nézzük azt, amelyikre a programunkhoz szükségünk van.
Ennek formája a következő:

for *ciklusváltozó*:=*kifejezés1* **to** *kifejezés2* **do** *utasítás*;
illetve
for *ciklusváltozó*:=*kifejezés1* **downto** *kifejezés2* **do** *utasítás*;

Ezeknek az utasításoknak a hatására a **for**-t követő ciklusváltozó értéke először a *kifejezés1* értékét veszi fel, ezután végrehajtódik az utasítás, majd

az említett változó értéke **to** esetén eggyel nő, **downto** esetén eggyel csökken, és újra végrehajtódik az utasítás, mindaddig, míg a változó értéke a *kifejezés2* értékét el nem éri.

Nézzük ezután a program listáját:

```
program atlag1;
var atl,delta   : real;
    szam        : array [1..10] of real;
    i           : integer;
begin
    atl:=0;
    for i:=1 to 10 do begin
                       Write(i,'. szám: ');
                       ReadLn(szam[i]);
                       atl:=atl+szam[i];
                      end;
    atl:=atl/10;
    delta:=abs(atl-szam[1]);
    for i:=2 to 10 do
      if delta<abs(atl-szam[i]) then delta:=abs(atl-szam[i]);
    WriteLn('Az átlag=',atl:8:3);
    WriteLn('A legnagyobb eltérés=',delta:8:3);
end.
```

Ebben a programban használtuk a fejezet elején említett, de részletesen nem tárgyalt **begin-end** utasítászárójeleket, ezeknek a szerepe a következő:

A Pascal a feltételes és a ciklusutasításokban (egy rövidesen szóba kerülő kivételtől eltekintve) egy utasítás feltételtől függő illetve ciklikus végrehajtását teszi lehetővé. Ez azonban az esetek többségében nem elegendő, így, például a példánkban is ugyanabban a ciklusban ki kellett írni a képernyőre valami információt, amely jelzi, hogy a gép adatot vár, be kellett olvasni az adatot, és az *atl* nevű változó értékét meg kellett változtatni. Erre szolgált az utasítászárójel, amelynek hatására a Pascal a **begin** és **end** között álló utasításokat egy *összetett utasításként* kezeli.

A teljesség kedvéért meg kell még említeni a Pascalban használt másik két ciklust is. Ezek, az említett **for** ciklustól eltérően nem adott számú esetben ismétlik meg a ciklusmagban szereplő utasítást, hanem egy feltételtől függően.

Az első ilyen ciklusutasítás a

while *feltétel* **do** *utasítás*;

Itt a program az utasításhoz érve először megvizsgálja a feltételt, amely mindig egy logikai kifejezés. Ha ennek éréke *true*, azaz a kifejezés teljesül, akkor végrehajtja az utasítást és újra vizsgálja a feltételt, és ezt ismétli mindaddig amíg a feltételként szereplő kifejezés értéke *false* nem lesz. A másik itt említett ciklusutasítás a

repeat
utasítás1;
utasítás2;
. . .
utasításn;
until *feltétel*;

alakú. Ez az utasítás az előbb említett kivétel, itt a ciklusmagja nem csak egy - egyszerű vagy összetett - utasítást tartalmazhat, hanem tetszőleges számút. A ciklusban a **repeat** és az **until** kulcsszavak közötti utasítások fognak ismétlődni. Ennek a ciklusnak a működése is eltér a **while** ciklustól, ugyanis itt először végrehajtódnak a ciklusban szereplő utasítások, utána a gép megvizsgálja az **until** után szereplő kifejezés értékét, és ha ez *false* (hamis) akkor ismétli meg a ciklus utasításait, ha pedig *true* (igaz), akkor elhagyja a ciklust, és a program a ciklus utáni utasítással folytatódik.

Az eddig leírtak alapján már nagyon sok program algoritmusa, szerkezete felépíthető.

A könyvben megtalálható a Pascal nyelv összes utasítása, ezeknek szabályai, működése, és sok példa, remélhetőleg ezeknek alapján mindenki eljut odáig, hogy önállóan meg tudjon oldani minden programozási feladatot, és még örömét is lelje benne.

3. A TURBO PASCAL NYELV ELEMEI

Mielőtt megismerkednénk a szabványos Pascal nyelv továbbfejlesztett változatával, a Turbo Pascal programozási nyelvvel, először definiálnunk kell a nyelv jelkészletét.

A Turbo Pascal az ASCII karakterkészletet használja. Egy karakter egy **byte**-ban kerül tárolásra. Tekintsük át röviden a bináris és a hexadecimális számok képzését.

A bináris és a hexadecimális számok

A mindennapi életben használt tizes számrendszert a számítógép közvetlenül nem tudja használni, ugyanis a számítógépek úgynevezett logikai áramkörei csak azt tudják megvizsgálni, hogy egy adott helyen (egy adott vezetékben) a feszültség értéke 0 vagy ettől eltérő. Ezt számokkal helyettesítve úgy lehet megfogalmazni, hogy a számítógép azt tudja eldönteni, hogy egy adott érték 0 vagy 1. Azt a számrendszert, amelyik csak ezt a két értéket használja, kettes vagy más néven *bináris számrendszernek* nevezzük. A *bit* (*binary digit*) egy kettes számrendszerbeli számot jelent, melynek az értéke 0 vagy 1 lehet. 8 *bit* alkot egy *byte*-ot. A memóriából kiolvasható legkisebb egység a *byte.*

A bináris számrendszerbeli számok tizes (másnéven decimális) számrendszerbeli értékének kiszámítását egy példán mutatjuk meg.

$$
\begin{array}{l}
1 * 2^7 = 128 \\
0 * 2^6 = 0 \\
1 * 2^5 = 32 \\
1 * 2^4 = 16 \\
0 * 2^3 = 0 \\
1 * 2^2 = 4 \\
1 * 2^1 = 2 \\
1 * 2^0 = 1 \\
\text{-------} \\
10110111 \quad - \quad 183
\end{array}
$$

A példából is látható, hogy a bináris szám 1-es helyein lévő kettőhatványnak megfelelő helyiértékeit adtuk össze a decimális érték kiszámításához.

A számítógép mindig bináris számokat használ, ez azonban számunkra nagyon kényelmetlen, hiszen például egy tizes számrendszerben ötjegyűként ábrázolt szám 14 - 17 jegyű bináris számot jelent. Keresni kell tehát egy olyan megoldást, amelyiket a számítógép is és mi is viszonylag könnyen kezelünk. Ennél a keresésnél vegyük figyelembe még azt a szempontot, hogy a számítógép felépítése olyan, hogy egyszerre mindig 8 illetve számítógéptől függően 16, 32 vagy 64 bináris számjegyet (másnéven bitet) használ.

Az előbbi két szempont figyelembevételével az alkalmazott számábrázolási forma a 16-os, másnéven *hexadecimális* számrendszer lett. Ez egy eléggé tömör számábrázolás, és könnyen kettes számrendszerbeli számokká alakítható, ugyanis minden hexadecimális számjegynek négy bináris számjegy felel meg, és viszont. A bináris - hexadecimális számok konverzióját ismét egy példán mutatjuk meg.

```
binária               bináris
 szám   értéke         szám    értéke
 8421                  8421
-------------       ------------------
 0000      0           1010      A (10)
 0001      1           1011      B (11)
 0010      2           1100      C (12)
 0011      3           1101      D (13)
 0100      4           1110      E (14)
 0101      5           1111      F (15)
 0110      6
 0111      7
 1000      8
 1001      9

 1011 0111 = B7 =  11*16 + 7 = 183

  B    7
```

Belátható, hogy egy **byte**-ba beírható legkisebb szám 0 és a legnagyobb szám az, ha a byte minden bitje 1:

1111 1111 = 128+64+32+16+8+4+2+1 = 255

Tehát egy *byte* 0-tól 255-ig vehet fel értékeket. Az ASCII karakterkészlet elemei egy byte-ban kerülnek ábrázolásra, mivel egy byte-ban összesen 256 különféle jelkombináció írható, ezért a karakterkészlet 256 elemet tartalmazhat maximálisan.

Először ismerkedjünk meg a speciális szimbólumokkal és a foglalt szavakkal, mert ezeknek az ismerete is szükséges a további definíciók megértéséhez.

3.1. A nyelv jelkészlete

A Turbo Pascal az ASCII karakterkészletből az alábbi szimbólumokat használja:

Betűk - A .. Z és a .. z az angol ABC betűi,
Számok - 0 .. 9,
Hexadecimális számok - 0 .. 9, A .. F és a .. f,
Szóköz (ASCII 32) és az összes ASCII vezérlőkarakter
(ASCII 0..31), beleértve a sorvége vagy a return karaktert (ASCII 13).

Egyszerű speciális szimbólumok:

+ - * / = . , : ; ' ^ _ @ $ # < > [ ] ( ) { }

A következő karakterpárok szintén speciális szimbólumok:

:= <= >= <> .. (* *)

A speciális szimbólumok tartalmazzák a műveleti jeleket is. Megjegyezzük, hogy helytelen az =<, =>, >< szimbólumok használata. Hibajelzést kapunk akkor is, ha a speciális két szimbólum közé üres helyet teszünk: < =

3.2. A foglalt szavak

A foglalt szavak tulajdonképpen a Pascal nyelv utasításai és a deklarációkat bevezető kulcsszavak.

A foglalt szavak:

absolute	**end**	**inline**	**procedure**	**type**
and	**external**	**interface**	**program**	**unit**
array	**file**	**interrupt**	**record**	**until**
begin	**for**	**label**	**repeat**	**uses**
case	**forward**	**mod**	**set**	**var**
const	**function**	**nil**	**shl**	**while**
div	**goto**	**not**	**shr**	**with**
do	**if**	**of**	**string**	**xor**
downto	**implementation**	**or**	**then**	
else	**in**	**packed**	**to**	

A szabványos Pascal **packed** kulcsszava a Turbo Pascalban hatástalan. A foglalt szavak a könyv szövegében kiemelve kerülnek kiírásra. A Pascal nem érzékeny a kis- és nagybetűre, így a programban a foglalt szavakat, illetve az utasításokat kis- és nagybetűvel is használhatjuk. A foglalt szavakat azonban más célra nem szabad használni, ilyen nevű azonosítók használata nem megengedett.

3.3. A program sorai

A program sorai tartalmazzák az utasításokat, amelyet egyenként részletesen ismertetünk és példákon keresztül bemutatjuk a működésüket.

Pascalban az utasítások leírásának nincsenek merev szabályai, mivel egy utasítást több sorban is írhatunk, illetve egy sorban több utasítás is szerepelhet. Az utasítást a ; (pontosvessző) zárja. Kivételt képez a program végét lezáró **end** utasítás, amely után . (pontot) kell tenni.

Mielőtt az utasításokat ismertetnénk, meg kell ismerkednünk az azonosítók fogalmával, különböző típusú számkonstansok ábrázolásával, a műveletek végrehajtásához szükséges operátorokkal és a program működését magyarázó megjegyzésekkel.

3.4. Azonosítók

Változóknak, konstansoknak, típusoknak, eljárásoknak, függvényeknek, unit-oknak, programnak és a **record**-ban szereplő mezőknek is nevet kell adnunk, hogy hivakozni tudjunk rájuk. Természetesen ezeknek a neveknek egyedieknek kell lennie, ami azt jelenti, hogy nem adhatjuk ugyanazt a nevet például változónak és függvénynek is. Tehát az *azonosító* azt jelenti, hogy az általunk megadott névvel azonosítjuk a különféle Pascal elemeket.

Az azonosítók bármilyen hosszúak lehetnek, de csak az első 63 karaktert veszi figyelembe a fordítóprogram. Megvan a lehetőségünk, hogy olyan hosszú azonosító neveket használjunk, hogy a programjaink olvashatóak legyenek. Ez azt jelenti, hogy a különféle azonosítók neve utalnak a bennük tárolt tartalomra.

Az azonosítók képzésénél nem használhatjuk fel a teljes ASCII készletet, meg kell ismerkednünk tulajdonképpen a névalkotásásának szabályaival. Az azonosítónak betűvel vagy _ (aláhúzás) kell kezdődnie és nem tartalmazhat szóközt (space). Betűk, számok és _ (aláhúzás) következhetnek az első karakter után. Hasonlóan a foglalt szavakhoz, az azonosítókban sincs megkülönböztetve a kis- és nagybetű.

Azonosítóként a foglalt szavakat nem szabad felhasználni!

Az azonosítók helyes használata:

```
a, a1, b1a, hosszu, a_oldal, vektor1
```

Az alábbi nevek ugyanazt az azonosítót jelentik:

```
a_oldal, A_oldal, A_Oldal, A_OLDAL
```

Hibás azonosítók:

```
a/1, b.23, a-b, a1*s, end
```

mivel / . - * jeleket tartalmaznak illetve nem lehet az **end** azonosító.

3.5. Számok

A számkonstansok lehetnek egész és valós típusúak, illetve decimálisak és hexadecimálisak. Megadásuk a következőképpen történik.

Egész típusú szám megadása előjellel (pozitív előjel elhagyható) és az utána következő számjegyekkel történik:

```
2  -3  +10  1992
```

Valós típusú szám megadása a következő részekből állhat:

előjel	egészrész	tizedespont	törtrész	kitevő
-	**1**	.	**234**	**E+02**

A kitevőrészben a 10 hatványát az E vagy az e betű jelenti - pozitív előjelű kitevő esetén a pozitív jelet nem kötelező kitenni:

```
-12.56e-5 = -12.56E-5 = -0.0001256

2.3e+3 = 2.3e3 = 2.3E+3 = 2.3E3 = 2300.0
```

Ha elhagyjuk az előjelet, akkor a szám pozitív lesz. A valós számban a tizedespont és a kitevőrész közül legalább az egyiknek szerepelnie kell, de ugyanez vonatkozik a egész részre és a törtrészre is. Az *e* betű a kitevőhöz tartozik.

```
-12.5    1.0     0.5  1.34E-3
-2.4e-2  3.56e3  4e5  5E+4
```

Gyakorlat:

1. Alakítsuk át az alábbi normál alakú számokat tizedes törtté:
   a. 2.4E3
   b. 0.56E-6
   c. -12.06E-4
   d. -2.345E7

Megoldás:

a. $2.43E3 = 2.43 \cdot 10^3 = 2430.0$
b. $0.56E-6 = 0.56 \cdot 10^{-6} = 0.00000056$
c. $-12.06E-4 = -12.56 \cdot 10^{-4} = -0.001206$
d. $-2.345E7 = -2.345 \cdot 10^7 = -23450000.0$

2. Az alábbi decimális számokat írjuk át normál alakúra:

a. -1240.5
b. 0.00367
c. -0.1254
d. 1.897

Megoldás:

a. $-1240.5 = -1.2405 \cdot 10^3 = -1.2405E3$
b. $0.00367 = 3.67 \cdot 10^{-3} = 3.67E-3$
c. $-0.1254 = -1.254 \cdot 10^{-1} = -1.254E-1$
d. $1.897 = 1.897 \cdot 10^0 = 1.897E0$

Hibásan megadott valós számkonstansok:

.5 -.25 .1E+3 12E+3.0

Hexadecimális egész szám a $00000000..$FFFFFFFF határon belül adható meg. A szám előtt álló $ jel jelzi, hogy hexadecimális megadást használunk.

3.6. Sztringek - karaktersorozatok

A sztring nulla vagy több karaktert tartalmaz a kiterjesztett ASCII karakterkészletből, amelyeket egy sorba kell írni a programban és '' (aposztrófok) közé kell tenni. Ha a sztring aposztrófot tartalmaz, akkor az aposztrófnak párosan kell szerepelni.

Ha a sztring semmit nem tartalmaz az aposztrófok között, azt üres sztringnek nevezzük: '' .

A sztring hossza az aposztrófok közötti karakterek számának felel meg:

'ez egy sztring'

a hossza 14 karakter.
A Turbo Pascal megengedi, hogy a karakter sztring vezérlő karaktereket is tartalmazzon. A # után közvetlenül álló, előjel nélküli egész konstans (0-255 határ között) egy ASCII karaktert jelöl meg. A #13 egy új sor beiktatását jelenti a karakterek közé.
Néhány példa a sztringre:

```
'Turbo Pascal'
'feladatok'
''''

'.'
 '
''

#13#13
'első sor'#13'második sor'
#7#7'csenget'#7#7
```

3.7. Címkék

A címke számokból álló szekvencia (0-9999), a 0 a szám elején hatástalan. A Turbo Pascal-ban megengedett a címke azonosítóval való jelölése. A programban a címke az utasítás megjelölésére szolgál. A címkéket a **goto** utasításban használjuk. A **goto** utasítás megszakítja a program végrehajtásának folyamatát, és a vezérlés az utasításban megadott címke által kijelölt utasításra adódik át. A címkéket a **label** kulcsszó mellett kell felsorolni.

A Turbo Pascal nyelvben használható címkék:

label 12, ide, oda,99,Vege, vissza;

3.8. Operandusok

A kijezésekben szereplő változókat, konstansokat (számokat, sztringeket, stb) és függvényehívásokat operandusoknak nevezzük.

```
a + b*c/sqr(x)+1.34
```

A fenti aritmetikai kifejezés operandusai: az *a,b,c* és *x* változók, az ***sqr*** függvényhívás (négyzetre emelő függvény) és az *1.34* számkonstans.

3.9. Operátorok

A kifejezésekben szereplő változókat, konstansokat és függvényhívásokat összekapcsoló műveleti jeleket operátoroknak nevezzük.

Operátorok az aritmetikai műveleti jelek (+,-,/,* stb), a logikai műveleti jelek (and, or, stb) és a relációk (<,>,<=, stb.). Az operátorokkal és a műveletek közötti precedenciával későbbiekben részletesen foglalkozunk.

```
a + b*c/sqr(x)+1.34
```

A fenti aritmetikai kifejezés operátorai: a +. -. *, / artimetikai műveleti jelek.

3.10. Kifejezések

A kifejezések, mint láttuk az operátorok és operandusok sorozatából állnak. Megkülönböztetünk aritmetikai és logikai kifejezéseket.
Aritmetikai kifejezés:

a+3.2*b
x+sin(y)

Az aritmetikai kifejezés kiértékelésének eredménye egy számérték.

Logikai kifejezés:

```
a>b
x<2 and y>3
```

A logikai kifejezés kiértékelésének eredménye logikai érték.

3.11. Megjegyzések

A program utasításai között bárhol tetszőleges hosszúságú magyarázatot, megjegyzéseket helyezhetünk el. Kétféle módon írhatjuk a megjegyzést:

- kapcsos zárójelek között:

```
{ a megjegyzés elsõ módja }
```

- kerekzárójelezés és csillagok között:

```
(* a megjegyzés második módja *)
```

A megjegyzéseket egyszeresen egymásba ágyazhatjuk, ha a különböző lezáró karaktereket használjuk:

```
{ ez a jó (* megjegyzés megadása *) }

{ ez így { nincs jól megadva } }

(* ez sincs (* jól megadva *) *)
```

A Pascal programban szóközt, a sorvéget és a megjegyzéseket elválasztóként használjuk. Az azonosítók, számok és speciális szimbólumok belseje kivételével bárhol tetszőleges számú elválasztó állhat.

A programban a megfelelő helyeken elhelyezett megjegyzések növelik a program olvashatóságát és kényelmessé teszik a program későbbi időpontban történő javítását és módosítását. A megjegyzések jó kiegészítője lehet a programdokumentációnak.

Ellenőrző kérdések:

1. Milyen karakterkészletet használ a Turbo Pascal nyelv?
2. Mekkora helyet foglal el egy karakter a memóriában ?
3. Mekkora egy *byte*-ba beírható maximális szám!
4. Hány elemet tartalmaz az ASCII karakterkészlet?
5. Mi az értéke az alábbi *byte*-oknak 10-es számrendszerben:

   0110 1010
   1101 0110
   0011 0101
   1001 1001

6. Adjuk meg az alábbi *byte*-ok hexadecimális értékét:

   0101 1100
   1110 1011
   1001 1100
   0110 1110

7. Mik azok a foglalt szavak?
8. A foglalt szavak érzékenyek-e a kis- és a nagybetű használatára?
9. Hány utasítást lehet írni egy sorban?
10. Mivel kell lezárni az utasítást?
11. A program végét mivel jelezzük?
12. Mit nevezünk azonosítónak?
13. Válasszuk ki az alábbi azonosítók közül a hibásan megadottakat és mely azonosítók azonosítják ugyanazt a változót:

```
sor, Sor, s-or, SOR, s1r, s/p, t_ir, z*text, ab#1, end,
1a1, sor, SOr, k.1, SoR, m-2, soR, a(1,
```

14. Sorolja fel a valós típusú számok részeit?

15. Mi a szerepe az E és a e betűknek az alábbi számkostansokban. Írja át a számokat tizedestört alakúvá!

    1.02E+2
    -3.24e-3
    2.5E-4
    4.5606e4

16. Milyen jellel kezdjük a hexadecimális számokat?
17. Milyen jelek között kell megadni a karaktersorozatot?
18. Mi a '''' string tartalom?
19. Mit jelent karaktersorozatban a #13 ?
20. Mivel azonosíthatjuk a címkéket?
21. Mire szolgálnak a címkék?
22. Mit jelent operandus fogalma?
23. Mit jelent az operátor?
24. Mi a kifejezés?
25. Milyen kifejezéseket ismer?
26. Milyen típusú az alábbi kifejezés?
    a+b*2+c
27. Milyen típusú az alábbi kifejezés?
    i < 2
28. Mi a megjegyzés írásának szabályai?
29. Hová helyezhető el megjegyzés?
30. Hová nem szabad megjegyzést elhelyezni?
31. Hogy lehet két megjegyzést egymásba ágyazni?
32. Mi a megjegyzés jelentősége?
33. Adjuk meg az alábbi megjegyzést kétfajta módon:
    Az adatok beolvasasa.

4. A TURBO PASCAL PROGRAM SZERKEZETE

A Pascal program három fő részből áll:

programfej
definíciós ill. deklarációs rész és a
programtörzs.

A kulcsszavak felhasználásával bemutatjuk a Turbo Pascal program szerkezetét:

```
{ programfej }
{ globális hatású fordítási direktívák }
    program programnév;

{ deklarációs rész }
    { lokális hatású fordítási direktívák }
    uses  { a használt unit könyvtárak felsorolása };
    label { címkék deklarációja };
    const { konstansok deklarálása };
    type  { típusok deklarálása };
    var   { változók deklarálása };

    procedure eljárásnév( paraméterek);
    { deklarációk }
    begin
        { az eljárás törzsének utasításai }
    end;

    function függvénynév(paraméterek);
    { deklarációk }
    begin
        { A függvény törzsének utasításai }
        függvénynév:= utasítás;   { a függvény eredménye }
    end;
```

```
{ a főprogram blokkja }
    begin
        { a főprogram törzsének utasításai }
    end.
```

A fentiek alapján láthatjuk, hogy a Pascal program feje a **program** kulcsszóval kezdődik, amelyet a program neve követ. A program nevének képzésére vonatkozó szabályok teljes mértékben megegyeznek az azonosítóknál ismertettettel.
Például:

```
program proba;

program Grafikai_probak;
```

A legegyszerűbb Pascal programban a deklarációs rész hiányozhat, azonban a program törzse azonban nem hiányozhat.

A programnak lehetnek paraméterei, azonban ezeknek Turbo Pascal-ban nincs jelentősége, mivel nincs kapcsolatuk a tényleges paraméterátadással, csak magunknak mintegy emlékeztetőül szolgál.
Például:

```
program Filekezelo(Olvas_file, Ir_file);

program(input, output);
```

A szabványos Pascal-ban az *input* a billentyűzetet és az *output* a képernyőt jelöli - a Turbo Pascal-ban ezeket nem szükséges megadni.

Egyes fordítási direktívákra a megfelelő fejezetekben hivatkozunk és az összes fordítási direktívát az F4. függelékben részletesen ismertetjük.

A deklarációs részben mindent fel kell felsorolni, amit a program használ. A külső modulokra (**unit**-okra) való hivatkozást mindig a deklarációs rész elején kell elhelyezni. A **uses** kulcsszó után adjuk meg ezeket a szabványos illetve a saját modulok neveit. A modulok alkalmazása nélkül csak 64 Kbyte méretű programot tudunk létrehozni. Ha viszont modulokat használunk, akkor csak a memória mérete, illetve az operációs rendszer korlátozza a

lefordított program méretét, amely maximálisan 640 Kbyte lehet. A modulok készítésével külön fejezetben foglalkozunk.

A deklarációban soroljuk fel a főprogram által használt címkéket (**label**) és konstansokat (**const**). A Pascal-ban a szabványos típusokon kívűl saját típusokat is létrehozhatunk (**type**). Az itt deklarált változók globálisak a teljes programra nézve (főmodul). A változók típusait (**integer, real,** stb.) külön fejezetben ismertetjük.

A deklarációs részhez tartoznak a felhasználó által készített alprogramok: eljárások és a függvények. Felépítésük hasonló a főprograméhoz, mivel szintén fejrészből, deklarációból és a végrehajtható utasításokból, tehát blokkból állnak. Az alprogramoknál a deklarációs rész az alprogramra vonatkozik, az itt deklarált változók a lokális változók, értékükhöz csak az alprogram fér hozzá. Ezek az alprogramok csak akkor működnek, ha a megfelelő paraméterekkel aktíváljuk őket.

A deklaráció ott zárul le, ahol a főprogram blokkja vagy az ún. törzse következik. A programnak ez nagyon lényeges része, amely **begin** és **end** között a végrehajtható utasításokat tartalmaz.

Ellenőrző kérdések:

1. Hány részből áll a Pascal program ?
2. Melyik rész hiányozhat egy egyszerű programnál?
3. A program feje milyen kulcsszóval kezdődik?
4. A programnak lehetnek-e paraméterei?
5. A külső modulokat milyen kulcsszó után kell deklarálni?
6. Hogyan növelhető a program mérete a 64Kbyte-os határ fölé?
7. Az eljárások és függvények leírása melyik részhez tartozik?
8. Mit tartalmaz a programtörzs?
9. A programtörzset milyen kulcsszavak között kell megadni?

PICDIC

Angol-Magyar komputeres szótár

szavakkal, képekkel, teszttel és játékkal!

A **PICDIC** 200 aprólékosan kidolgozott számítógépes grafikát és 5000 brit és amerikai angol, valamint 5000 magyar szót tartalmaz. (A szoftvernek németmagyar és olasz-magyar változata is van!)
A képek témakörök szerint vannak csoportosítva. Ezekben találhatjuk meg a mindennapi életben leggyakrabban előforduló szókincset.

Példaképpen álljon itt egy kép a **PICDIC**-ből:

In a Restaurant-Étteremben

A PICDIC egyik funkciója, hogy a képen megjelölt pontokra rámutathatunk.
Ekkor kinyílik egy ablak, és az ablakban megjelenik az objektum magyar és angol neve, valamint az angol szó kiejtése.
Lehetőség van arra, hogy csak az angol, csak a magyar szót, vagy csak a kiejtést lássuk.
82 témakör közül választhatunk. Lapozgathatunk a téma-körök, valamint a hozzájuk tartozó képek között.
Mód van a megadott angol, illetve magyar szavakhoz tartozó képelemek visszakeresésére is.
Az anyagot **Szendrő Borbála**, az I LOVE WORDS című könyv szerzője, a Budapesti Közgazdaságtudományi Egyetem Angol Tanszékének adjunktusa állította össze, felhasználva többek között angliai és amerikai kutatásait.

Kiknek ajánljuk szoftverünket ?

Kezdőknek, haladóknak, felnőtteknek és gyerekeknek egyaránt, vagyis mindazoknak, kik a szótanulás izzadságos munkáját szeretnék sokkal hatékonyabbá és sokkal könnyebbé tenni a számítógép felhasználásával.

Mi kell ahhoz, hogy ezt a szoftvert használhassa ?

Egy IBM PC számítógép VGA, EGA vagy Hercules monitorral, winchesterrel. Egér jó, ha van.

!!! A szoftver 2,5 Mbyte-nyi helyen elfér !!!

Érdeklődni és megrendelni az alábbi címen lehet:

PROFI-SZOFT BT.
6500 Baja,Kölcsey u.112.
TEL: 79-25845 napközben
79-25983 este
Borsódi Donát

FIGYELEM!
A szoftvernek német-magyar és olasz-magyar változata is létezik!

5. ADATTÍPUSOK ÉS DEKLARÁCIÓK

Ebben a fejezetben részletesen tárgyaljuk a változók deklarálását, foglalkozunk az felhasználói típusok létrehozásával, a struktúrált adattípusokkal, valamint a konstansok megadásával. Példákon keresztül mutatjuk be használatukat, a fejezet tartalmának a begyakorlására ajánljuk az ellenőrző kérdések áttanulmányozását.

A programok az adatokat *változókban* tárolják. A változók egy vagy több *byte*-ot foglalnak le a memóriából, ez a tárterület tartalmazza a változók *aktuális értékét*. A program futása során a változók változtathatják az értéküket. Ha megvizsgáljuk egy változó tartalmát, akkor *pillanatnyi értékről* beszélünk. Minden változót el kell neveznünk az azonosítók szabályai szerint, ezek az ún. *változóazonosítók*.

Minden változó a tulajdonságától függően veheti fel értékét. A változó tulajdonságát *típusnak* nevezzük. A Pascalban a **var** kulcsszó után minden változót deklarálnunk kell és meg kell jelölnünk a típusát. A deklarációban egy változó csak egy típusnál szerepelhet. Ha egy típushoz több változó tartozik, akkor azokat vesszővel elválaszta kell felsorolni, majd a kettőspont után a típus következik és az utasítást pontosvessző zárja.

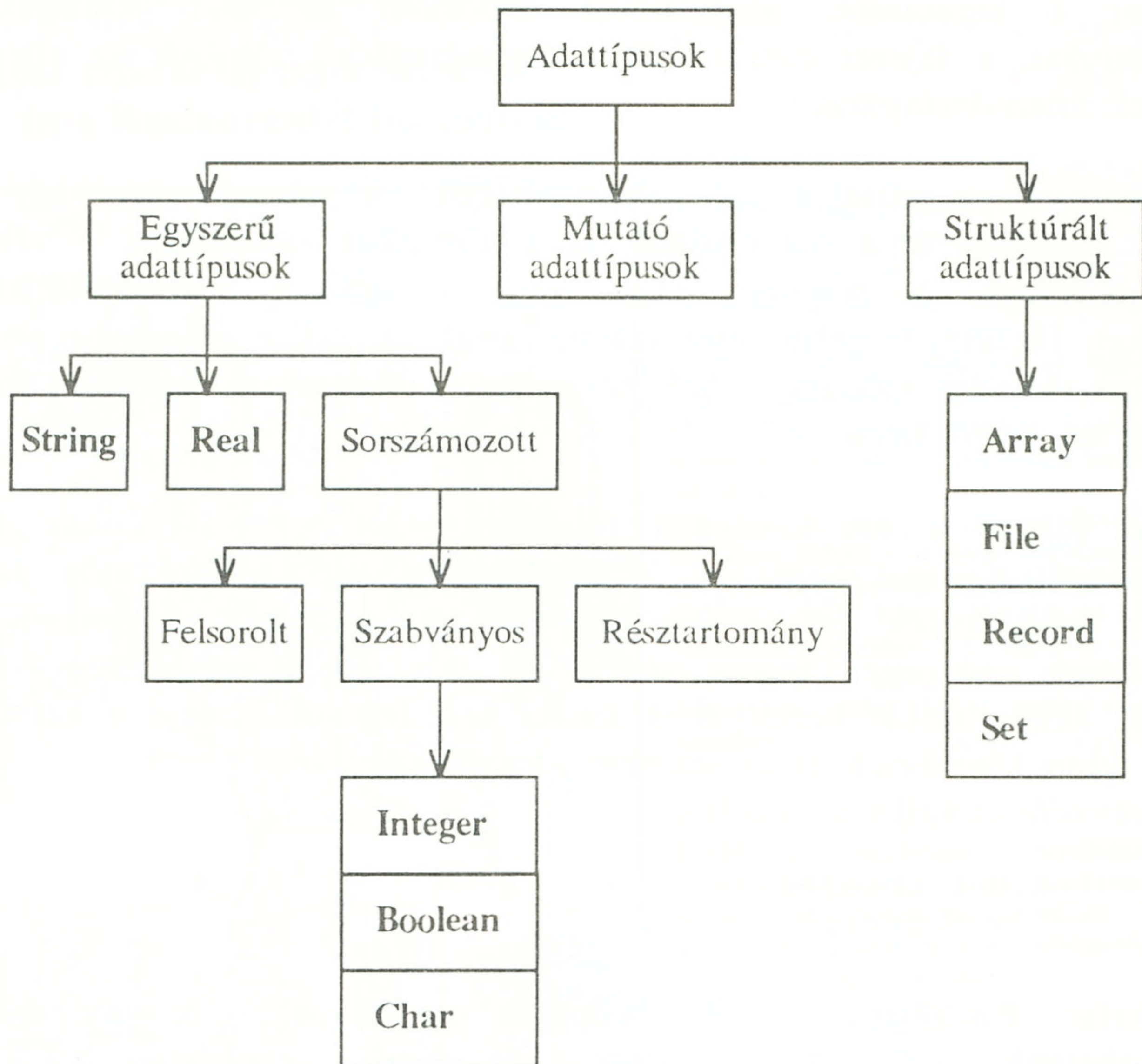
```
var
     a,b : integer;
       x : real;
```

A Turbo Pascalban vannak beépített adattípusok, de magunk is létrehozhatunk különféle típusokat. A Pascal nyelvben használatos adattípusokat az 5.1. ábra szemlélteti.

Az adattípusok tárgyalását az egyszerű adattípusokkal kezdjük. Először ismerkedjünk meg a numerikus értékeket, majd a karaktereket és a logikai értékeket tároló adattípusokkal.

5.1. Egyszerű adattípusok

Mint már említettük, az adattípus megmutatja, hogy egy változó milyen értékeket vehet fel, vagyis kijelöli az értékkészletét. Először tekintsük át a nyelvhez tartozó, beépített egyszerű (*standard*) típusokat.



5.1. ábra A Pascal nyelv adattípusai

5.1.1. Numerikus információ tárolása

Először azokkal az egyszerű adattípusokkal foglalkozunk, amelyek numerikus adatokat, számokat tárolnak. Numerikus adatokat tároló változókkal aritmetikai műveleteket végezhetünk.

5.1.1.1. Egész típusok

A Turbo Pascal-ban ötfajta előredefiniált egész típus van: **shortint**, **integer**, **longint**, **byte** és **word**. Ezek mind egész számokat tárolnak, de értékkészletük a helyfoglalásuk miatt különböző. Az 5.1. táblázatban összefoglalva tekintsük át az adott típushoz tartozó értékkészletet és a helyfoglalást:

típus	értékkészlet	helyfoglalás
shortint	-128 .. 127	előjeles 8 bit =1 byte
integer	-32768 .. 32767	előjeles 16 bit =2 byte
longint	-2147483648 .. 2147483647	előjeles 32 bit =4 byte
byte	0 .. 255	előjel nélküli 8 bit =1 byte
word	0 .. 65535	előjel nélküli 16 bit =2 byte

5.1. táblázat

A leggyakrabban az **integer** típust használjuk, természetesen amennyiben a -32768 és 32767 korlát megfelel a számábrázolásra. A programunk tervezésénél gondosan kell megválasztani a változók típusát, mivel nem megfelelő típus esetén a program helytelen eredményt adhat. A Turbo Pascal a legnagyobb lehetséges **integer** típusú szám tárolására fenntart egy MAXINT konstans, melynek értéke 32767 és a **longint** típus számára pedig MAXLONGINT konstanst, melynek értéke 2147483647.
A példában különféle egész típusú változók deklarálását láthatjuk:

```
var
        i,j       : shortint;
        flag      : longint;
        ch        : byte;
        xx        : word;
        n         : integer;
```

5.1.1.2. Valós típusok

A Turbo Pascalban a valós számok tárolására 5 különböző típust definiáltak, ezek a **real, single, double, extended** és a **comp**. A **real** használható minden esetben. A másik négyhez 8087 mód használata (8087, 80287 ill. 80387 géptípustól függően) vagy annak emulálása ({$N+}) (szoftver úton való kiváltása) szükséges.
A 5.2. táblázat foglalja össze a valós típusokat, a különböző valós típusokhoz tartozó értékkészletet abszolút értékben (ugyanez vonatkozik a negatív számokra, de ellenkező előjellel), az értékes jegyek számát és a helyfoglalásukat:

típus	értékkészlet	értékes jegyek	helyfoglalás
real	$2.9x10^{-39}$ .. $1.7x10^{38}$	11 - 12	6 byte
single	$1.5x10^{-45}$ .. $3.4x10^{38}$	7 - 8	4 byte
double	$5.0x10^{-324}$ .. $1.7x10^{308}$	15 - 16	8 byte
extended	$3.4x10^{-4932}$.. $1.1x10^{4932}$	19 - 20	10 byte
comp	$-2^{-63}+1$ .. $2^{63}-1$	19 - 20	8 byte

5.2. táblázat

A valós típusok közül leggyakran a **real** típust használjuk.

```
var
     osszeg, szumma: real;
```

5.1.2. Logikai informácó tárolása

A logikai értékek tárolására csak a logikai típusú változók alkalmasak, amelyekkel logikai műveleteket végezhetünk.

5.1.2.1. Logikai típus

A **boolean** kulcsszó után deklaráljuk a logikai típusú változókat, amelyek kétféle értéket vehetnek fel: *true* (igaz) és *false* (hamis). A logikai változó helyfoglalása egy **byte**.

```
var
    b1,b2 : boolean;
```

5.1.3. Szöveges információ tárolása

Kétféle típus használható szöveges információk tárolására. Tárolhatunk csak egy karaktert, vagy több karaktert, tehát karaktersorozatot. A szöveges információt az ASCII karaktersorozat elemei alkotják. A szöveges információkkal sztring műveleteket végezhetünk.

5.1.3.1. Karakter típus

A **char** a karakter típus egyetlen karakter tárolására alkalmas, amely egy byte-ot foglal le a memóriában. A karakter konstanst aposztrófok közé zárjuk, pl. 'a' , '0', '*'. Az aposztrófok közé zárt 0 a 0 karaktert jelenti, míg a 0 az egész 0 konstanst jelöli. A karakterek az ASCII kódokat tartalmazhatják, amelyek rendezett halmazt alkotnak. A karakterek sorszámozott típusúak.

```
var
    c1,c2,kod: char;
```

5.1.3.2. String típus

A **string** típus alkalmas karaktersorozat tárolására. Korábban már említettük, hogy egy karakter egy byte-ot foglal le. A **string** típusú változó karaktereinek darabszámára is csak egy byte van fenntartva. A byte-ba írható maximális szám 255, tehát ez szabja meg azt a korlátot, hogy a **string** típusú változó 255-nél több karaktert nem tartalmazhat. A **string** nem sorszámozott típus.

```
var
    nev: string;
```

Ebben a példában a *nev* maximálisan 255 karaktert tartalmazhat. Ha nem szükséges 255 karakter, használhatunk kevesebbet is, akkor indexben (szögletes zárójelek között) kell megadni a szükséges darabszámot.

```
var
    keresztnev    : string[10];
    vezeteknev    : string[20];
    kod           : string[3];
    betu          : string[1];
```

A sztringben tárolt karaktersorozat karaktereihez is hozzférhetünk, használva a sztring-indexet oly módon, hogy a sztring azonosító mellett szögletes zárójelben megadjuk a vizsgálandó karakter sorszámát. A **string** típusú változók tartalmát karakterenként a sztring-index segítségével könnyen tudjuk vizsgálni.

5.1.4. Sorszámozott típus

Az előzőekben megismerkedtünk néhány egyszerű adattípussal. Az 5.1. ábrán láthatjuk, hogy az egyszerű adattípusokat tovább osztályozhatjuk sorszámozott ill. nem sorszámozott típusokra.
Sorszámozott típusok amelyekkel már foglalkoztunk:

integer, shortint, longint, byte, word, boolean, char

Ide tartozik még a

felsorolt, résztartomány,

amelyeket még nem ismertettünk.
Nem sorszámozott típusok pedig a

real, string

ezeket a típusokat már megismertük.

Ismerkedjünk meg most három függvénnyel, amelyek csak a sorszámozott típusok vizsgálatára alkalmasak, ezek:

ord, succ, pred

Mint már említettük, az ASCII karakterek egy halmazt alkotnak, amelyben mindegyik ASCII karakterhez tartozik egy sorszám és ezért ezen a halmazon könnyen mozoghatunk.

Az ***ord*** függvény paramétere sorszámozott típus lehet, például karakter típusú változó, karakter konstans, egész szám stb., eredményként a karakter sorszámát vagy a paraméterének típusdeklarációbeli sorszámát adja vissza.
A ***succ*** függvény bemenő paramétere sorszámozott típus, eredményként a következő elemet adja.

A ***pred*** függvény bemenő paramétere sorszámozott típus, eredményként az előző elemet adja.

A *false* és a *true* is sorszámozott. A *false* sorszáma 0, a *true* sorszáma 1. Tehát a *false* < *true*;
Ekkor

ord(*false*)	eredménye	0
ord(*true*)	eredménye	1
succ(*false*)	eredménye	*true*
pred(*true*)	eredménye	*false*
succ('c')	eredménye	'd'
pred('b')	eredménye	'a'
succ(7)	eredménye	8
pred(10)	eredménye	9

5.1.4.1. Felsorolt típus

Lehetőség van arra, hogy magunk is létrehozhassunk sorszámozott típusokat. Ezt az ún. felsorolt típust a következőképpen hozzuk létre: a **var** kulcsszó után megadjuk az azonosítót, majd a kettőspont után kerek zárójelben vesszővel elválasztva soroljuk fel az egyedi azonosítókat. A felsorolás egy

nagyság szerinti sorrendet is jelent, ahol az első elem sorszáma 0. Az egyedi azonosítókat nem szabad '' (aposztrófok) közé tenni, mivel ezek nem sztringeket jelentenek.

```
var
    nyelvek=(angol, nemet, spanyol, olasz);
```

A fenti felsorolt típusokra is nézzük meg az *ord*, *succ* és a *pred* függvény eredményét :

ord(angol)	értéke 0
ord(spanyol)	értéke 2
succ(nemet)	értéke spanyol
pred(olasz)	értéke spanyol.

5.1.4.2. Résztartomány típus

Résztartomány típust bármely korábban definiált, sorszámozott típus tetszőleges részsorozatával megadhatunk. A definiálás a részsorozat azonosítójával kezdődik, az a kettőspont egyenlőségjel után a részsorozat, mint intervallum alsó és felső határát kell megadni két ponttal elválasztva, majd pontosvessző zárja az utasítást.

```
var
    index  : 1..100;
    kisbetu: 'a'..'z';
```

A index az egész számoknak, a kisbetű pedig a karakterek sorozatának egy résztartományát jelenti. Az *ord* függvényt itt is alkalmazhatjuk.

Ha futás közben ellenőrízni akarjuk, hogy nem léptük-e át a résztartomány határát, akkor a programot a {$R+} fordító direktíval kell fordítani, mert alapértelmezésben ez a direktíva nem aktív. Ezt csak akkor ajánlatos használni, míg a program helyes működését ellenőrízzük, mert ez a vizsgálat lassítja a program futását.

5.1.5. Mutató típus

A mutató típus szintén elemi adattípus. A mutató típust a **pointer** a kulcsszavával, másrészt pedig *^típus* alakban adhatunk meg. A mutató típus memória címet vesz fel, helyfoglalása 4 byte. Az első két byte az offszetcímet, a második két byte a szegmenscímet tartalmazza. Deklarációja a következő:

```
var
    mem, hely : pointer;
    iptr      : ^integer;
```

5.2. Struktúrált típusok

A struktúrált (másképpen összetett) típusok a tömb, a rekord, a halmaz és a file adattípusok.

5.2.1. Tömb típus

A tömb meghatározott számú, azonos típusú elemből álló összetett adattípus. A tömb memóriabeli helyfoglalását az elemszám és az elem típusának megfelelő elemméret szorzatával számíthatjuk ki. Tíz elemű **integer** (2 byte) típusú tömb 10x2 azaz 20 byte-ot foglal le a memóriában.
A tömb megadásának a szintaxisa a tömb azonosítójával kezdődik, majd a : következik, azután az **array** kulcsszót követő szögletes zárójelben a tömb alsó és felső indexhatárát, mint résztartományt adjuk meg. Az ezt követő **of** kulcsszó utáni adattípus fogja meghatározni, hogy a tömb elemei milyen adatokat fognak tartalmazni. A típus lehet elemi típus, összetett sőt felhasználó által definiált típus is.

```
var
    y: array[1..10] of integer;
```

Az *y* tömb 10 darab egész típusú elemet tartalmaz. A tömb egyes elemeire tömbváltozó nevével és a deklarált határok közé eső indexszel hivatkozunk:

```
y[1], y[2],y[3] .. y[10]
```

Az index lehet sorszámozott típusú változó, konstans vagy kifejezés:

`i:=2; j:=3;` az i változó értéke 2, a j változóé 3, akkor
`y[i]` az y tömb második elemét jelenti, mert i tartalma 2
`y[i+j]` mivel $i+j$ értéke 5, az y tömb ötödik elemét jelenti
`y[4]` az y tömb 4. elemét jelenti.

A tömb ha egydimenziós, akkor vektornak, ha kétdimenziós, mátrixnak nevezzük. Deklaráljuk egy valós számokat tartalmazó mátrixot:

```
var
    x: array[1..2,1..3] of real;
```

Az x kétdimenziós tömbnek 2 sora és 3 oszlopa van:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix}$$

A kétdimenziós tömb első indexe a sor index, a második indexe az oszlop index. A második sor harmadik oszlopának elemét a következőképpen választjuk ki:

`x[2,3]`

vagy `j:=2; k:=2;` akkor az `x[j,k+1]` ugyancsak az `x[2,3]` elemet választja ki.

A tömbök elemei a memóriában az indexük alapján helyezkednek el olymódon, hogy az indexük jobbról balra növekednek. Kétdimenziós tömb esetén először az oszlop index, majd a sor index növekszik (sorfolytonos tárolás). A példában deklarált x vektornál

```
x[1,1], x[1,2], x[1,3], x[2,1], x[2,2], x[2,3]
```

a memóriabeli sorrend.

Nézzünk meg egy háromdimenziós deklarációt:

```
var
    y: array[1..2,1..2,1..3] of real;
```

Az *y* tömb memóriabeli sorrendje

```
y[1,1,1], y[1,1,2], y[1,1,3],
y[1,2,1], y[1,2,2], y[1,2,3],
y[2,1,1], y[2,1,2], y[2,1,3],
y[2,2,1], y[2,2,2], y[2,2,3].
```

A Turbo Pascalban a tömb dimenzionálásának egyetlen korlátja az, hogy bármely Pascal adat maximális mérete kisebb kell legyen 64 Kbyte-nál. Az indexhatár 0 és negatív is lehet. Az indexeknek az érvényes számok tartományába kell esniük. Az index alúl- illetve túlcsordulásának figyelésére az {$R+} fordítási direktíva szolgál, az alapértelmezés szerint ez a direktíva nem aktív.

5.2.2. Rekord típus

A rekord a legrugalmasabb Pascal adatszerkezet, mivel benne tetszőleges számú különböző tulajdonságú (típusú) rész szerpelhet. A rekord típusdeklarációjában **record** kulcsszó után fel kell sorolni komponenseinek típusát és ezt a felsorolást **end** kulcsszóval kell lezárni. Ezeket a komponenseket a rekord mezőinek nevezzük.

A következőképpen írhatunk le egy dátumot:

```
var
     datum : record
               ev   : 1000..2000;
               honap: 1..12;
               nap  : 1..31;
               megjegyzes: string;
             end;
```

A *datum* rekordnak 4 mezeje van. Az *év*, a *hónap* és a *nap* résztartomány típusú, az adatokat csak a megadott határok között fogadja el. A dátumhoz megjegyzést fűzhetünk a *megjegyzes* mezőben, amely **string** típusú.
A rekord elemeire úgy hivatkozunk, hogy a rekord változó neve után ponttal elválasztva írjuk az elem azonosítóját. Adjunk értéket a *datum* rekordnak:

```
datum.ev          := 1974;
datum.honap       := 3;
datum.nap         := 13.
datum.megjegyzes := 'Zoli szuletesnapja';
```

A variálható rekordnak nevezzük az a rekordot, amelynek mezőadataihoz valamilyen tulajdonságtól függően más és más adatok tartoznak.

Példaként nézzük meg, hogy egy könyvtárban a könyveket milyen adatok megadása mellett tarthatjuk nyilván. A könyveknek egy vagy több szerzője lehet. A könyv címe mellett tárolhatjuk a kiadójának nevét, a kiadás évét, az oldalak számát és a leltáriszámot. A könyv tulajdonságát határozza meg az a tény, hogy kölcsönözhető-e. A könyv csak akkor kölcsönözhető, ha a raktárban van. Ilyen esetben a raktárban való helyét tartjuk nyilván. Ha a könyvet kikölcsönözték, akkor a kikölcsönző személy adatait rendeljük a könyv adataihoz. (A **case** utasítást lásd a 8.2.2.2. részfejezetben.)

```
var
  konyv : record
             szerzok   : sztring;
             cim       : sztring[50];
             kiado     : sztring[30];
             ev        : integer;
             oldalak   : 1..1000;
             raktariszam: 1..10000;
             case kiveheto : boolean of
              true:
                  (hely : record
                           emelet  : 1..4;
                           sor     : 1..15;
                           polc    : 1..50;
                            end);
             false:
                 ( olvaso : record
                           nev     : string[30];
                           lakcim  : string[40];
                           telefon : string[15];
                           lejar   : datum;
                         end);
           end;
         end;
```

5.2.3. Halmaz típus

A halmazelmélet a matematika külön tudományága, bizonyos feladatok legtermészetesebben halmazok használatával oldhatók meg. Ennek támogatására szolgál a Pascal *set* (halmaz) típusa.

A halmaz deklarálása:

```
var
     halmaz_változó : set of alaptipus;
```

formában történik. A Turbo Pascalban a halmaz maximum 256 elemet tartalmazhat, az alaptípus csak felsorolt (sorszámozott) típusú lehet.

```
var
     abc    : set of 'a'..'z';
     szamok : set of 0..1000;
```

a fenti deklarációkat használva az *abc* halmaz felveheti az alábbi értékeket

`abc:=[ ];`	üres halmaz
`abc:=['a'..'z'];`	teljes halmaz
`abc:=['b','d'];`	a *b* és a *d* betűt tartalmazza

A halmazokkal külön részfejezetben újra foglalkozunk.

5.2.4. File típus

A **file** azonos típusú komponensekből álló adatszerkezet, amely nagymennyiségű adat tárolását teszi lehetővé a háttértárolón. A **file** komponensei lehetnek egyszerű és struktúrált típusok, de leggyakrabban a **record** típust használjuk.
A **file** változó deklarálása:

```
var
     adat: file of integer;
```

A **file** kezeléssel részletesen a 11. fejezet foglalkozunk.

5.3. Felhasználói típus létrehozása (type)

Az előbbi részfejezetekben láttuk, hogy minden változónak a deklarációban meg kellett nevezni a típusát. A Turbo Pascal a típusdefiníciós részben a **type** kulcsszó után saját típusok létrehozását is támogatja. A típusdeklarációt a következő módon kell megadni:

type
újtipus = létezőtípus;

Először a létrehozandó új típus nevét, illetve azonosítóját adjuk meg, majd az = (egyenlőségjel) után különféle létező típust, akár korábban definiált saját típust is meg lehet adni.

```
type
     egesz = integer;
     valos = real;
     st10  = string[10];
var
     i,j  : egesz;
     a,b  : valos;
     nev  : st10;
```

A későbbiekben inkább a strukturált típusokra fogjuk alkalmazni típusalkotást, mivel az eljárások és függvények a struktúrált típusokat paraméterként csak ily módon fogadják el.

5.3.1. A type használata felsorolt típus esetén

A felsorolt típus megadása és ezen típusoknak a deklarációban való felhasználása az alábbiak szerint történik:

```
type
   tantargy=(magyar, fizika, matematika,tortenelem);
   nyelvek =(angol, nemet, spanyol, olasz, kinai);
   szinek  =(piros,kek,zold,sarga,lila,fekete,feher);
var
     orarend : tantargy;
     nyelvora: nyelvek;
     ruha    :  szinek;
```

A fenti felsorolt típusokra is használhatjuk az *ord*, *succ* és a *pred* függvényeket. Más szabványos függvény használata ezeknél a típusoknál nem megengedett.
Ugyanakkor az a szabály is teljesen egyértelmű, hogy az *angol*, *nemet*, *spanyol* és az *olasz* értékeket csak *nyelvek* típusú változó veheti fel, mint például a deklarációban látható *nyelvora*.

5.3.2. A type használata résztartomány típus esetén

Résztartomány típust valamely létező, sorszámozott típus bármely rész-intervallumából állíthatjuk elő. A definiálás a résztartomány típus azonosítójával kezdődik. Az egyenlőségjel után az intervallum alsó és felső határát kell megadni két ponttal elválasztva, majd pontosvessző zárja az utasítást.

```
type
    szinek    =(piros,kek,zold,sarga,lila,fekete,feher);
    kisbetu   = 'a'..'z';
    szam      = '0'..'9';
    szinresz  = kek..lila;
    korlat    = 10..100;
    index     = 1..1000;

var
    minta: szinek;
    i,j  : index;
    alap : szinresz;
```

A *szinresz* típus változó a résztartományát a korábban definiált felsorolt típusú *szinek* alapján jelöljük ki. Így az alap csak a *kék, zöld, sárga ,lila* színeket veheti fel értékként.

5.3.3. A type használata tömb esetén

Ha a tömböt felhasználói típusként akarjuk deklarálni, akkor a felhasználói típusú tömböt először a **type** kulcsszó mellett a változó deklarációhoz hasonlóan adjuk meg, csak a : helyett = jelet kell tennünk és ezután már a **var** kulcsszó után használható a kívánt deklaráció.

Nézzük meg az alábbi tömbtípusok deklarációban való felhasználását:

```
type
     szinek    =(piros, kek, sarga, feher);
     nev       = string[20];
     vektor    = array[1..100] of real;
     matrix    = array[1..10,1..10] of integer;
     adat      = array[1..50] of real;
     osztaly   = array[1..32] of nev;
     szintabla= array[1..8,1..8] of szinek;
     abc       = array['a'..'z'] of char;
     xx        = array[-1..4, 0..2] of integer;
var
     a,b       : vektor;
     x,y       : matrix;
     w         : szintabla;
     konyv     : abc;
     tarol     : xx;
```

5.3.4. A type használata record esetén

Nézzünk példát rekordtípus létrehozására:

```
type
     datum = record
               ev   : integer;
               honap: (jan,febr,marc,apr,maj jun,jul,
                       aug,szept,okt,nov,dec);
               nap  : 1..31;
               megjegyzes: string;
             end;
     szemely= record
               vezetek_nev: string[20];
               kereszt_nev: string[10];
               szul_datum : datum;
               allapota   : (hajadon,notlen,ferjezett,
                             nos,ozvegy);
               lakcim     : string;
              end;

var
     mai_nap, holnap: datum;
     munka_csoport  : szemely;
```

5.3.5. A type használata halmaz esetén

A halmaztípus deklarálása:

type
halmaz_tipus = set of alaptipus;

formában történik.
Például:

```
type
    napok  = (hetfo,kedd,szerda,csutortok,
              pentek,szombat,vasarnap);
    het    = set of napok;
var
    munkanap, pihenonap,teljes_het : het;
```

5.4. Konstansok deklarálása

A konstansok deklarációja a **const** kulcsszó után történik úgy, hogy az egyenlőségjel bal oldalán megadjuk az azonosítót, a jobb oldalán pedig a hozzárendelni kívánt értéket. A fordítóprogram az azonosító típusát a jobb oldalon álló konstansból állapítja meg.

Numerikus konstansok:

```
const
    jelzo    = -1;
    ev       = 1991;
    norm     = 25.6;
    max      = 1.5e6;
    min      = 1e-5;
    hexa     = $00ff;
```

Szöveges konstansok:

```
const
    fejlec        ='Az egyenlet megoldasa';
    ciml          ='Feladat:';
    betu          ='w';
    ures_sztring  ='';
```

A konstansok annyiszor szerepelnek a programban, ahányszor használjuk azokat. Ezzel szemben a tipusos konstansok csak egyszer jelennek meg a lefordított programban.

5.4.1. Típusos konstansok

Típusos konstansok megadási módja: típusos konstans azonosítója kettőspontttal zárva, majd a típus azonosítója következik és az egyenlőségjel után az értéket kell megadni, végül az utasítást pontosvessző zárja. A típusos konstansok paraméterként használhatók, azonban nem használhatók más konstansok és típusok deklarációjában.
A típusos konstansok nemcsak egyszerű típusúak lehetnek, hanem összetett típusúak, így tömb, halmaz és rekord típusú is. A típusos konstans nem lehet file típusú.
Mutató lehet pl.:

```
    const
        xp: pointer = pointer($FFFFFFFF);
```

Tömb típusú típusos konstans

Egydimenziós tömb esetén az értékeket zárójelek között vesszővel kell felsorolni.

```
  const
     adatok: array[1..4] of integer =(5,1,3,7);
     fejlec1:array[1..8] of char
            =('M','e','g','o','l','d','a','s');
     fejlec2: array[1..8] of char = 'eredmeny';
```

Példaként nézzük meg az alábbi 3 dimenziós deklarációt és értékadást:

```
var
    a: array[1..2, 1..2, 1..3] of integer;
begin
    a[1,1,1]:=1;  a[1,1,2]:=2;  a[1,1,3]:=3;
    a[1,2,1]:=4;  a[1,2,2]:=5;  a[1,2,3]:=6;
    a[2,1,1]:=7;  a[2,1,2]:=8;  a[2,1,3]:=9;
    a[2,2,1]:=10; a[2,2,2]:=11; a[2,2,3]:=12;
```

Ha többdimenziós konstansként adnánk meg az előző feldatatot, akkor az értékeket a tömböknél a dimenziónak jobbról balra növekvő sorrendjének megfelelően kell megadni.

```
const
    a:array[1..2, 1..2, 1..3] of integer=
    (((1,2,3),(4,5,6)),((7,8,9),(10,11,12)));
```

Többdimenziós konstansok értékadását mutatja be a TOMB_IND.PAS program.

Halmaz típusú típusos konstans

A halmaz elemeit résztartományként, vagy az elemek felsorolásával lehet megadni:

```
    const
        Szam10    : set of char = ['0'..'9'];
        ABC       : set of char = ['a'..'z','A'..'Z'];
        Mgh       : set of char = ['a','e','i','o','u'];
```

Rekord típusú tipusos konstans

Először a rekord típusát hozzuk létre, majd a **const** után a rekord konstanst úgy adjuk meg, hogy a mezőkonstansok a típusdefiníció sorrendjében kapjanak értéket.
Például dátum esetén:

```
    type
        datum = record
                    ev : integer;
                    honap : 1..12;
                    nap : 1..31;
                    megjegyzes: string;
                 end;

    const
        mai_nap : datum = (ev: 1992; honap: 7; nap : 29;
                                     megjegyzes: 'evfordulo');
```

5.5. Tipuskonverzió

A Turbo Pascalban a {$R-} direktíva megadása esetén adott a lehetőség, hogy különféle típuskonverziókat hajtsunk végre. Ehhez a kívánt típus neve után zárójelben kell megadni a konvertálandó kifejezést. Például:

```
chr(65)
integer('A')
```

A TIPKONV.PAS programban karakter típust egésszé, egészt logikai értékké, karaktert pointerré, pointert valós típussá, valamint pointert sztringre mutató pointerré alakítottuk.

```
(*  tipkonv.pas  *)
{$R-}
program tipuskonverzio;
uses crt;
var
     i : integer;
     w : word;
     c : char;
     b : boolean;
     p : pointer;
     r : real;
     s : string;
begin
     clrscr;
     c:='b';
     writeln('c = ',c);
     (* karakter konvertalasa egessze *)
     i:=integer(c);
     writeln('i = ',i);
     (* egesz konvertalasa boolean-ra *)
     b:=boolean(i);
     writeln('b = ',b);
     (* pointer erteke real valtozoba *)
     p:= @c;
     r:=real(p^);
     writeln('r = ',r:10);
     (* pointer ertek sztring-ge alakitva *)
     s:= string(p^);
     s[0]:=chr(1);
     writeln('s = ',s);
end.
```

A futtatás eredménye:

```
c = b
i = 98
b = TRUE
r = 8.094E-10
s = b
```

Ellenőrző kérdések:

1. Mit nevezünk változóknak?
2. Mi a változóazonosító?
3. Mi a kulcsszava a változók deklarálásának?
4. Melyek az elemi adattípusok?
5. Sorolja fel a numerikus információ tárolására szolgáló elemi adatípusokat?
6. Mekkora az értékkészlete az **integer** típusnak?
7. Mely egész típusú változók nem tartalmazhatnak negatív számokat?
8. Mikor érdemes **longint** típusúnak deklarálni egy egész típusú változót?
9. Mire használjuk a **byte** típusú változót?
10. Milyen különbség van a **shortint** és a **byte** között?
11. Hány fajta valós típusú változót használhatunk?
12. Melyik valós típus használható minden esetben?
13. Milyen nagyságrendű értéket tárolhat a **real** típusú változó ?
14. Mennyi az értékes jegyek száma a **real** típusú változók esetén?
15. Hány **byte** szükséges egy **real** típusú változó tárolásához?
16. Mikor kell használnunk **double** típusú változót?
17. Milyen legyen a deklaráció az alábbi változók esetében, ha
    - v tartománya 0 és 125,
    - m tartománya -20000 és 20000,
    - i tartománya -200000 és 100000,
    - k tartománya 0 és 65000,
    - x lebegőpontos számokat tárol és tartománya a 10^{-39} és 10^{38} közé esik,
    - betu az ABC egy - egy betűit tartalmazza,
    - nev 25 db karakterből álló nevet tárol,
    - cim 32 db karakterből áló nevet tárol.
18. Logikai értékek tárolására milyen típusú változók alkalmasak?

19. Milyen kulcsszóval deklaráljuk a logikai típusú változókat?
20. Milyen értéket vehetnek fel a logikai típusú változók?
21. Milyen lehetőségek vannak a szöveges információ tárolásásra?
22. A karaktertípus mennyi helyet foglal le a memóriából?
23. A karaktersorozat tárolására milyen típusú változó alkalmas?
24. Egy **string** típusú változóban maximálisan mennyi karaktert tárolhatunk?
25. Ha egy *nev* nevű változóban csak 20 karaktert akarunk tárolni, akkor hogyan kell megadni a deklarációt?
26. Hogyan kell megadni a deklarációt, ha a *kar* változó csak egy karaktert tárol?
27. Milyen kulcsszó szükséges a típusdefinícióhoz?
28. Mit jelent a sorszámozott típus?
29. Mely típusok nem tartoznak a sorszámozott típusok közé?
30. Mit ad vissza az ***ord*** függvény?
31. Mire szolgál az *succ* függvény?
32. Mi lesz az értéke:
    ```
    succ(9); succ('b');
    ```
33. Mire szolgál a ***pred*** függvény?
34. Mi lesz az értéke:
    ```
    pred(9), pred('b')
    ```
35. Mennyi a *false* sorszáma?
36. Mennyi a *true* sorszáma?
37. Mi lesz az értéke:
    ```
    suc(false); pred(true); ord(false); ord(true);
    ```
38. Mi a szintaktikája a felsorolt típusú változó deklarálásának?
39. Hozzunk létre egy felsorolt típust, azonosítója legyen
    osztalyzat
    elemei legyenek: *elegtelen*, *elegseges*, *kozepes*,*jo*,*jeles*,*kitűnő*
    Deklaráljunk az *osztalyzat* típusú *erdemjegy* változót.
40. Legyen adott az alábbi típus és változó deklaráció, mi lesz az eredménye a függvényhívásoknak:

```
type
    alak=(szikar, sovany, karcsu, telt, kover);
var
    szemely: alak;
```

```
begin
     szemely:=(szikar,sovany,karcsu,telt,kover);
     succ(sovany);
     ord(szikar);
     pred(telt);
     ord(kover);
```

41. Hogyan kell definiálni a résztartomány típust?
42. Az alábbi programrészben milyen értékeket vehet fel az *i* változó:

```
type
     hatar=1..100;
var
     i: hatar;

begin
```

43. A tömb milyen típusú elemekből áll?
44. Hogyan számítjuk ki a tömb helyfoglalását?
45. Mi a tömb megadás szintaxisa **type** és **var** kulcsszó esetén?
46. Legyen az *adat* 100 **real** típusú elemet tartalmazó tömbtípus, majd az *x* nevű tömböt deklaráljuk *adat* típusúnak?
47. Mit jelent az alábbi programrészlet

```
type
     vektor    =array[1..50] of integer;
     tomb      =array[1..5,1..5] of real;
var
     a,b  : vektor;
     x    : tomb;
```

Mennyi a helyfoglalása az *a* és az *x* tömbnek?

48. Mi a tulajdonsága a **record** típusnak?
49. Hogyan kell megadni egy **record** típust?
50. Hogy hivatkozunk a **record** mezőire?
51. Adjuk kezdőértéket a *teglalap* típusú rekordnak, úgy hogy a *szelesseg* mezeje legyen 15, a *magassag* mezeje vegye fel a 8 értéket. A **begin** után írjuk fel az értékadó utasításokat.

```
type
     teglalap = record
                  szelesseg : integer;
                  magassag  : integer;
                 end;
var
     a: teglalap;
begin
```

52. Milyen kulcsszó után lehet a konstansokat deklarálni?
53. Mi a szintaktikája a konstans megadásának?
54. Deklaráljuk az alábbi változókat konstansként

minimum	értéke	-10
maximum	értéke	1200
cim	értéke	'Megoldas'
karakter	értéke	'%'

55. A *w* valós tömbnek típusos konstansként adjunk értéket, ha elemeinek száma 5, és az alábbi értékeket vegye fel

w[1]	értéke	1.3
w[2]	értéke	0.5
w[3]	értéke	100.3
w[4]	értéke	-12.56
w[5]	értéke	-0.0017

6. PASCAL KIFEJEZÉSEK

A kifejezések operátorok és operandusok sorozatából áll. Az operátoroknak nevezzük a különféle műveleti jeleket, amelyek összekapcsolják a kifejezésben szereplő változókat, konstansokat és függvényhívásokat, tehát az ún. operandusokat.

6.1. Operátorok

Pascal-ban kétfajta operátor használható.

1. Az operátort, amely két opendus között szerepel (ún. *binary operator*), két operandusú operátornak nevezzük:

operadus1 *operator* **operandus2**

Például:	a + b	két változó összege
	x * 2	változó és egy konstans szorzata

2. Az operátort, amely egy operandus előtt szerepel (ún. *unary operator*), egy operandusú operátornak nevezzük:

operator **operandus**

Például: -a előjelváltás

Azért, hogy a bonyolultabb kifejezések kiértékelése is egyértelmű legyen, meg kell ismerkednünk az elsőbbségi (*precedencia*) szabállyal, amely pontosan meghatározza a kifejezésekben a műveletek kiértékelési sorrendjét.

A operátorok végrehajtási sorrendjét a 6.1 táblázat foglalja össze, ahol négy szintű elsőbbséget különböztetünk meg. Ebben a táblázatban az összes operátort felsoroltuk.

Operátorok	Precedencia szintje	Operátor típusa
@, not	első	unary
*,/,div,mod,and,shl,shr	második	multiplikatív
+,-,or,xor	harmadik	additiv
=,<>,<,>,<=,>=,in	negyedik	reláció

6.1. táblázat

6.2. Elsőbbségi szabály

A kifejezések kiértékelésénél három elsőbbségi (precedencia) szabályt kell figyelembe venni. Nézzük meg általánosan ezt a három elsőbbségi szabályt:

1. Ha két különböző precedenciájú operátor között van az operandus, akkor a magasabb precedenciájú operátorhoz tartozik, tehát először az a művelet kerül végrehajtásra.

2. Ha két azonos precedenciájú operátor között van az operandus, akkor a tőle balra álló operátorhoz tartozik és az a művelet kerül végrehajtásra.

3. Ha a művelet zárójelezve van, akkor a zárójelben lévő műveletek kiértékelésének van elsőbbsége.

6.3. Operátorok csoportosítása

Az operátorokat különböző típusú műveletek szerint csoportosíthatjuk:

- aritmetikai operátorok,
- bitenkénti logikai operátorok,
- boolean operátorok,
- relációk,
- @ (pointer) operátor.
- sztring összekapcsolás operátora,
- halmaz műveleti operátorok

6.3.1. Aritmetikai operátorok

Az aritmetikai operátorokat aritmetikai kifejezésben használjuk. Az operandusok lehetnek egész és valós típusúak. Vizsgáljuk meg a különböző típusú operandusokkal végzett műveletek eredményének a típusát is.

A kétoperandusú aritmetikai operátorokat példákkal bemutatva az alábbi táblázatban foglaltuk össze:

Operátor	Művelet	Egészekre	Valósakra
+	összeadás	8 + 3 = 11	8.0 + 3 = 11.0
-	kivonás	8 - 3 = 5	8 - 3.0 = 5.0
*	szorzás	8 * 3 = 24	8.0 * 3 = 24.0
/	osztás	nincs értelmezve	8.0 / 3 = 2.66
div	egész osztás	8 div 3 = 2	nincs értelmezve
mod	maradék képzés	8 mod 3 = 2	nincs értelmezve

Csoportosítsuk az aritmetikai operátorokat az elsőbbségi szabály szerint:

Elsődleges:	*, / ,div, mod	multiplikatív (szorzás, osztás)
Másodlagos:	+ -	additív (összeadás, kivonás)

Vizsgáljuk meg az egészekre és a valósakra a különböző műveletek eredményének a típusát. Legyen általánosan a két operandus *a* és *b*.

Az *a+b*, *a-b*, *a*c* műveleteknél, ha az *a* és a *b* egész típusú, akkor az eredmény is egész típusú lesz.

3+4 eredmény 7

Ha az *a* és a *b* közül valamelyik valós, akkor az eredmény mindig valós lesz.

3.0+4 eredmény 7.0
3+4.0 eredmény 7.0

Az *a/b* műveletnél az eredmény mindig valós, még akkor is ha a két operandus egész típusú.

6.0 / 3 eredmény 2.0
6 / 3 eredmény 2.0

A **div** az egészek osztása. A *i* **div** *j* műveletben az *i* és a *j* csak egész típusú lehet és az eredmény is egész típusú lesz. Hiba történik, ha a *j* tartalma zérus.

5 **div** 2 eredmény 2

A *i* **mod** *j* maradék képzés csak egész típusokra van értelmezve. Hiba történik, ha a *j* tartalma zérus. A **mod** segítségével könnyen vizsgálható az oszthatóság. Ha az *i* **mod** *j* eredménye 0, akkor ez azt jelenti, hogy a *j* maradék nélkül megvan az *i*-ben, tehát osztható *j*-vel.

10 **mod** 3 eredmény 1
12 **mod** 2 eredmény 0
12 **mod** 5 eredmény 2

A **div** és a **mod** valós operandusokra nincs értelmezve.

Egyoperandusú aritmetikai operátorok:

Operátor	Művelet
+	pozitív előjel
-	előjelváltás

Az pozitív előjel és előjelváltás eredménye egész típusú változó esetén egész típusú lesz, valós típus esetében pedig valós. A *+a* esetében az *a* tartalma változatlanul, *-a* esetében az tartalma ellenkező előjellel kerül felhasználásra. Ha az *a* tartalma -3 volt, akkor *-a* esetén +3 értékkel számolunk.

a:=3;
b:=2 + a; b tartalma 2 + (+3) = 2 + 3 = 5 lesz
c:=2 - a; c tartalma 2 - (+3) = 2 - 3 = -1 lesz

Vizsgáljuk meg a precedencia szabályt aritmetikai kifejezéseken keresztül

1. Két különböző precedenciájú *operátor* között lévő **operandus** a magasabb precedenciájú *operátorhoz* tartozik:

   $a+b*c$

   a b jobb oldalán *, bal oldalán + műveleti jel van, egyértelmű, hogy a b a szorzáshoz tartozik, mert a szorzásnak magasabb a precedenciája.
   A műveletek végrehajtási sorrendje
   1. $b*c$
   2. az eredményhez az a hozzáadódik.

2. Két azonos precedenciájú *operator* között lévő **operandus** a tőle balra álló *operátorhoz* tartozik.

   $b+c/d*e$

   a d jobb oldalán a * (szorzás), a bal oldalán / (osztás) műveleti jel van, tehát azonos precedenciájú operátorok vannak, ezért a d a tőle balra álló / operátorhoz tartozik.
   A műveletek végrehajtási sorrendje:

   1. c/d, először az osztás történik meg, majd
   2. az eredménynek az e-vel való szorzása következik, mert a szorzás magasabb precedenciájú, mint az összeadás, tehát az e felkerült a számlálóba
   3. majd az eredményhez hozzáadódik a b.

   Képletté visszaírva a Pascal kifejezést:

   $$b + \frac{c}{d} \cdot e$$

3. Ha a művelet zárójelezve van, akkor a zárójelben lévő műveletek kiértékelésének van elsőbbsége. Tehát, ha el akarunk térni a precedenciától, akkor zárójeleznünk kell.

Írjuk át Pascal kifejezéssé az alábbi képletet:

$$b + \frac{c}{d \cdot e}$$

a képlet szerint a *c* változót *d*e* szorzattal kell osztani. Ezt kétféleképpen írhatjuk át

1. *b+c/(d*e)*
   A zárójel miatt először a *d*e* szorzat kerül először kiértékelésre, majd ezután következik a *c*-nek az eredménnyel való osztása, végül a *b*-nek a hozzáadása.
2. Ugyanazt az eredményt kapjuk, ha egyenként osztunk: *b+c/d/e*
   Az azonos precedencia miatt először a *c*-nek *d*-vel való osztása után az eredmény *e*-vel való osztása következik, végül az eredményhez hozzáadódik a *b*.

Írjuk át a Pascal kifejezéssé az alábbi képletet:

$$b + \frac{c + d}{e - g}$$

Ha a számlálóban megadott összeget akarunk egy különbséggel osztani, akkor mindenképp zárójeleket kell használni:

b+(c+d)/(e-g)

Írjuk át Pascal kifejezéssé az alábbi képletet:

$$b + \frac{c + d}{e \cdot g}$$

Ennél a képlet átírásnál is kell zárójelet alkalmazni. Az átírás két lehetséges formája:

*b+(c+d)/(e*g)*
b+(c+d)/e/g

Gyakorlat:

1. Irjuk át az alábbi képletet Pascal kifejezéssé:

a. $x = \frac{y-2}{a \cdot y} + 4 \cdot \frac{a}{y}$

x:=(y-2)/(a*y)+4*a/y;

b. $w = \frac{\frac{x}{y} + \frac{y}{x}}{z - 5 \cdot p}$

```
w:=(x/y + y/x)/(z-5*p);
```

c. $d = \frac{a+b}{b+c} \sqrt{\sin^2 x + 2 \cdot \cos x^3}$

```
d:= (a + b)/(b + c)*sqrt(sqr(sin(x)) + 2*cos(x*x*x));
```

d. $x = -b + \sqrt{b^2 + 4 \cdot a \cdot c}$

```
x:= -b + sqrt(b*b + 4*a*c);
```

e. $w = 3a + 4^{x+1}$

w:=3*a + exp((x+1)*ln(4));

f. $z = x + \log_2 y$

z:= x + ln(y)/ln(2);

2. A Pascal kifejezést írjuk vissza képletté

`e:= a*b/c*d;` $e = \frac{a \cdot b}{c} \cdot d$

`e:= a*b*c/d;` $e = \frac{a \cdot b \cdot c}{d}$

`e:=a/d*b*c;` $e = \frac{a \cdot b \cdot c}{d}$

`e:= a*b/c/d;` $e = \frac{a \cdot b}{c \cdot d}$

`e:=a*b/(c*d);` $e = \frac{a \cdot b}{c \cdot d}$

`e:= a/b/c*d;` $e = \frac{a \cdot d}{b \cdot c}$

`e:= a/d*b/c;` $e = \frac{a \cdot b}{d \cdot c}$

3. Adjuk meg az alábbi kifejezések típusát

Kifejezés	típus	eredmény
7 - 4	egész	3
3.0*9	valós	27.0
17 mod 7	egész	3
9/3	valós	3.0
29 div 5	egész	5
12.4/2	valós	6.2
5.2 div 5	hibás	csak egész operandusokra értelmezett
6 mod 3.0	hibás	csak egész operandusokra értelmezett

6.3.2. Bitenkénti logikai operátorok

A bitenkénti logikai műveletek egész típusú operandusokra értelmezettek és az eredmény szintén egész típusú.

Operátor	Művelet
not	bitek szerinti negáció (tagadás)
and	aritmetikai és
or	aritmetikai vagy
xor	aritmetikai kizáró vagy
shl	eltolás balra (shift left)
shr	eltolás jobbra (shift right)

Példa a műveletekre:

not művelet: a **bitek szerinti negáció** az operandus minden bitjét negáltja, ami azt jelenti, hogy ahol 1 volt ott 0 lesz.

not 54 = 201

Gépi ábrázolásban:

```
    54  = 0011 0110
not 54  = 1100 1001 eredménye 128+64+8+1 = 201 (-55)
```

and művelet: az **aritmetikai és** kapcsolat a két operandust összehasonlítja, ahol mindkét helyen 1 volt, ott 1 lesz, különben 0 lesz az eredményben.

54 and 20 = 20

Gépi ábrázolásban:

```
       54 =  0011 0110
       20 =  0001 0100
54 and 20 =  0001 0100 = 16+4 = 20
```

or művelet: az **aritmetikai vagy** kapcsolat eredményeként ott lesz 1-es, ahol valamelyik operandusban 1 volt, csak ott lesz 0, ahol mind a kettőben 0 volt.

38 or 104 = 110

Gépi ábrázolásban:

```
       38  =  0010 0110
      104  =  0110 1000
38 or 104  =  0110 1110 = 64+32+8+4+2 = 110
```

xor művelet: **aritmetikai kizáró vagy** esetében ott kapunk 1-et, ahol a két operandusban 0 1 vagy 1 0 volt és 0-át kapunk 0 0 vagy 1 1 esetében. Ez azt jelenti, hogy az eltérések helyén 1, azonos értékeknél pedig 0 lesz az eredmény.

38 xor 104 = 78

Gépi ábrázolásban:

```
       38  =  0010 0110
      104  =  0110 1000
38 xor 104=  0100 1110 = 64+8+4+2 = 78
```

shl művelet: a megadott számú bittel az operandus bitjeit eltolja balra.

34 shl 2= 136

Gépi ábrázolásban:

```
       34  =  0010 0010
  34 shl 2 =  1000 1000 = 128+8 = 136
```

shr művelet: megadott számú bittel az operandus bitjeit eltolja jobbra.

36 shr 2 = 9

Gépi ábrázolásban:

```
     36 =   0010 0100
36 shr 2 =  0000 1001 = 8+1 = 9
```

Futtassuk le a BIT.PAS programot a példaként megadott számokkal.

```
program bit;
var
   a,b,c:integer;
begin
    writeln('Bitenkenti logikai muveletek');
    writeln(' not a');
    write(' a = '); readln(a);
    writeln('  not ',a:4,'    = ',not a);
    writeln(' a and b ');
    write(' a = '); readln(a);
    write(' b = '); readln(b);
    writeln(a:4,' and ',b:4,' = ',a and b);
    writeln(' a or b ');
    write(' a = '); readln(a);
    write(' b = '); readln(b);
    writeln(a:4,' or  ',b:4,' = ',a or b);
    writeln(' a xor b');
    write(' a = '); readln(a);
    write(' b = '); readln(b);
    writeln(a:4,' xor ',b:4,' = ',a xor b);
    writeln(' a shl b');
    write(' a = '); readln(a);
    write(' b = '); readln(b);
    writeln(a:4,' shl ',b:4,' = ',a shl b);
    write(' a shr b ');
    write(' a = '); readln(a);
    write(' b = '); readln(b);
    writeln(a:4,' shr ',b:4,' = ',a shr b);
end.
```

6.3.3. Boolean operátorok

A **boolean** típusú változók a boolean operátorokkal **boolean** eredményt hoznak létre.

A **boolean** operátorokat az alábbi táblázat foglalja össze:

Operátorok	**Műveletek**
not	logikai negálás
and	logikai és (logikai szorzás = multiplikativ művelet)
or	logikai vagy (logikai összeadás = additív művelet)
xor	logikai kizáró vagy (additív művelet)

not A A negálás azt jelenti, hogy a **not** operátor mögött megadott **boolean** változó tartalma az ellenkezőjére változik, ha igaz volt, hamis, ha hamis volt igaz lesz.

A	not A
TRUE	FALSE
FALSE	TRUE

A **and** B Az **and** csak akkor ad igaz eredményt, ha a két **boolean** változó az A és a B tartalma igaz volt. Hamis lesz az eredmény, ha a két változó közül legalább az egyik hamis.

A	B	A and B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

A **or** B Az **or** (logikai vagy) akkor ad igaz eredményt, ha a két **boolean** változó közül legalább az egyik tartalma igaz, csak akkor lesz hamis, ha mindkét boolean változó tartalma hamis.

A	B	A or B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

A **xor** B Az **xor** (logikai kizáró vagy) akkor ad igaz eredményt, ha a két **boolean** operandus tartalma különböző és akkor ad hamis eredményt, ha a két operandus tartalma megegyezik, tehát mindkettő igaz, vagy mindkettő hamis.

A	B	A xor B
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

A precedencia szabály a logikai kifejezésekre is vonatkozik.

Értékeljük ki az alábbi logikai kifejezéseket, ha a logikai változók az alábbi értéket tartalmazzák:

```
A :=false;
B :=true;
C :=true;
D :=false;
```

1. *A and B or C*
   a kifejezés kiértékelésének a sorrendjét a kapcsos zárójel mutatja:

A and B or C

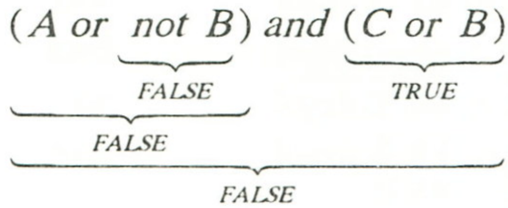
(*A and B* → FALSE; *A and B or C* → TRUE)

2. *A or B and C or D*

A or B and C or D

(*B and C* → TRUE; *A or B and C* → TRUE; *A or B and C or D* → TRUE)

3. *(A or not B) and C or B*



4. *(A or B) and (C or D)*

Határozzuk meg a kiértékelés sorrendjét, számítsuk ki a kifejezés értékét és hasonlítsuk össze a bool.pas program futási eredményével.

Futtassuk le a BOOL.PAS programot a példában megadottakkal, valamint más adatokkal is és értékeljük ki az eredményt. A BOOL.PAS program bemutatja, hogyan lehet **boolean** értéket beolvasni.

```
program bool;
var
      A,B,C,D : boolean;
            s : string;
  R1,R2,R3,R4 : boolean;
begin
   writeln('Boolean muveletek');
   writeln;
   write('A = '); readln(s);
   if s='true' then A:=true else A:=false;
   write('B = '); readln(s);
   if s='true' then B:=true else B:=false;
   write('C = '); readln(s);
   if s='true' then C:=true else B:=false;
   write('D = '); readln(s);
   if s='true' then D:=true else D:=false;

   R1:=A and B or C;
   R2:=A or B and C or D;
   R3:=(A or not B) and (C or B);
   R4:=(A or B) and (C or B);
   writeln('A and B or C = ',R1);
```

```
    writeln('A or B and C or B = ',R2);
    writeln('(A or not B) and (C or B) = ',R3);
    writeln('(A or B) and (C or D) = ',R4);
end.
```

6.3.4. Relációs (összehasonlító) operátorok

A reláció operátorok különféle egymással kompatibilis típusú kifejezések (aritmetikai, sztring stb.) összehasonításására szolgálnak. Az összehasonlító eredmény igaz, ha a reláció teljesül, különben hamis.

A relációs operátorok:

=	egyenlő
<>	nem egyenlő
<	kisebb
>	nagyobb
<=	kisebb vagy egyenlő
>=	nagyobb vagy egyenlő

Példák relációkra:
A változók pillanatnyi tartalma

a:=1.2; i:=2; j:=3; k:=4; x:=5.3; y:=-1.2; eps:=0.005;

(i+j) = 5	értéke **true**, mivel	5 = 5
(a+1)*k > x	értéke **true** , mivel	8.8 > 5.3
abs(x-y) < 1.0e-6	értéke **false**, mivel	6.5 > 1.0E-6
(x-eps) <= a	értéke **false**, mivel	4.695 > 1.2
i <> j	értéke **true**, mivel	2 <> 3

Futtassuk le a REALACIO.PAS programot a példában megadottakkal, valamint más értékekkel is és értékeljük ki az eredményt.

```
program relacio;
var
  i,j,k    : integer;
  a,x,y,eps: real;
```

```
begin
   write('i   = '); readln(i);
   write('j   = '); readln(j);
   write('k   = '); readln(k);
   write('a   = '); readln(a);
   write('x   = '); readln(x);
   write('y   = '); readln(y);
   write('eps = '); readln(eps);
   write(' (i+j)=5           ');
   if (i+j)=5 then writeln(' true ')
              else writeln(' false ');
   write(' (a+1)*k > x       ');
   if (a+1)*k > x then  writeln(' true ')
              else writeln(' false ');
   write(' abs(x-y) <1.0e-6 ');
   if abs(x-y) <1.0e-6  then  writeln(' true ')
              else writeln(' false ');
   write(' (x-eps) <= a      ');
   if (x-eps) <= a then  writeln(' true ')
              else writeln(' false ');
   write(' i<>j              ');
   if i<>j then  writeln(' true ') else writeln(' false ');
 end.
```

6.3.5. Pointer operátorok

A pointer operátor a @ és a ^. Ezek egyoperandusú operátorok, amelyek elsődleges prioritással rendelkeznek:

@a
p^

A @ operátor bármilyen típusú operandus címét jelenti. A példában a @ az *a* változó címét jelenti.
A ^ jelet pointer változó esetén lehet használni. A példában a *p^* jelenti a pointer által mutatott objektumot.

6.3.6. Sztring összekapcsolás operátora

A sztringeknél csak a + műveletet alkalmazhatjuk. Ezen sztring operátor neve: az összefűzés. Az összefűzött sztring hossza sem haladhatja meg a 255 karaktert.

'Feladat' + ' megoldasa' --> 'Feladat megoldasa'

Természetesen sztring típusú változókat is összefűzhetünk, ezt mutatja be a FUZ.PAS program:

```
program fuz;
var
        s1,s2,s3: string;
begin
        s1:='Ma';
        s2:='hideg';
        s3:=s1+' '+s2+' van.';
        writeln(s3);
end.
```

Az s3 tartalma:

```
Ma hideg van.
```

6.3.7. Halmaz műveleti operátorok

A halmazokkal is végezhetünk műveleteket. A kifejezésben az operandusoknak halmazoknak kell lenni, az eredmény is halmaz típusú lesz.

A halmazokra az alábbi kétoperandusú operátorok használhatók:

+	halmazok egyesítése (uniója)
*	halmazok metszete
-	halmazok különbsége

A halmazműveletek precedenciája:
elsődleges: *
másodlagos: +, -

A halmazok metszete: X*Y

Az X és Y halmaz metszetének (közös része) eredménye olyan halmaz, melynek elemei az X-nek és a Y halmaznak is elemei.

Például:

X halmaz elemei: a,b,c,d
Y halmaz elemei: a,c,e,f

akkor az X*Y eredménye [a,c] halmaz, mert ezek az elemek mind a két halmazban megtalálhatók, ez az ún. közös része a két halmaznak

A halmazok egyesítése: X+Y

Az X és Y halmaz egyesítésének eredménye a két halmaz elemeit tatalmazza. A közös elemek csak egyszer szerepelnek.

X+Y eredménye az [a,b,c,d,e,f] halmaz.

A halmazok különbsége: X-Y

Az X és Y halmaz különbsége olyan halmaz, amelynek minden eleme az első halmaznak és nem eleme a második halmaznak.

X-Y eredménye [b,d] halmaz.

Értékeljük ki az alábbi halmazkifejezést, ha:

AH halmaz elemei: a,b,c,d
BH halmaz elemei: a,c,e,f
CH halmaz elemi : a,g,h
DH halmaz elemei: b,f

*AH+BH*CH-DH*

A kiértékelés sorrendje a precedencia szabály figyelembevételével

a. *BH*CH*, eredmény halmaz [a]
b. az eredmény halmazhoz hozzáadva *AH*, az eredmény [a,b,c,d]
c. az eredmény halmazból levonva *DH*, az eredmény halmaz [a,c,d]

Futtassuk le a HALMAZ1.PAS progamot, amely az előző gyakorlatban kiértékelt halmazkifejezés tartalmát kiírja a képernyőre.

```
program halmaz1;
type
    abc=(a,b,c,d,e,f,g,h);
    hh= set of abc;
var
   AH,BH,CH,DH,RH: hh;
   i:abc;
begin
     AH:=[a,b,c,d];
     BH:=[a,c,e,f];
     CH:=[a,g,h];
     DH:=[b,f];
     RH:=AH+BH*CH-DH;
   write('Az eredmeny halmaz elemei: ');
  for i:=a to h do
  begin
    if i in rh then begin
                          case i of
                             a: write('a');
                             b: write('b');
                             c: write('c');
                             d: write('d');
                             e: write('e');
                             f: write('f');
                             g: write('g');
                             h: write('h');
                           end;
                      end;
  end;
  writeln;
end.
```

A futtatás eredménye:

```
Az eredmeny halmaz elemei: acd
```

6.3.7.1. Halmazokra vonatkozó vizsgálatok

Két halmazról megállapíthatjuk, hogy egyenlőek, ill. nem egyenlőek, vagy részhalmaza egyik a másiknak.

A halmazokra az alábbi relációk használhatók:

=	egyenlő,
<>	nem egyenlő,
<=	részhalmaz { a jel bal oldalán álló halmaz a jobb oldalon álló halmaz részhalmaza. }
>=	részhalmaz { a jobb oldalon álló halmaz a bal oldalon álló halmaz részhalmaza. }

A halmazra vonatkozó 'eleme?' vizsgálat:

ha (*halmazelem* **in** *halmaz*) **igaz, akkor** *{ benne van a halmazban }*
különben *{ nincs benne a halmazban}*

in az eredmény **boolean** típusú, akkor igaz, ha az **in** kulcsszó bal oldalán álló halmazelem a jobb oldalon álló halmaznak eleme.

Érdemes megjegyezni, hogy a halmazműveletek általában gyorsak, és a forráskódot is tömörebbé teszik.

A HALMAZ2.PAS programban tanulmányozzuk a halmaztípus használatát, a halmazokon végzett műveleteket és értékeljük ki az eredményt.

```
program halmaz2;

type
  napok=(hetfo,kedd,szerda,csutortok,
         pentek,szombat,vasarnap);
  het  = set of napok;

var
  munkanap,piheno_napok, teljes_het: het;
  rendel, nem_rendel,szabad_vagyok :het;

begin
     teljes_het:=[hetfo..vasarnap];
```

```
     szabad_vagyok:=[szerda];
     piheno_napok:=[szombat,vasarnap];

     if teljes_het >= piheno_napok then
       writeln('A teljes_het tartalmazza a piheno napokat');

     munkanap:=teljes_het-piheno_napok;

     if munkanap <= teljes_het then
           writeln('A munkanap reszhalmaza a teljes_hetnek');

     rendel:=[hetfo,szerda,pentek];
     nem_rendel:=munkanap-rendel;

     if csutortok in nem_rendel then writeln('Szabad nap');
     if szerda in rendel then writeln('Rendeles van');
     if rendel*szabad_vagyok <>[] then
                 writeln('Rendelesre mehetek');
end.
```

A program futási eredménye:

```
A teljes_het tartalmazza a piheno napokat
A munkanap reszhalmaza a teljes_hetnek
Szabad nap
Rendeles van
Rendelesre mehetek
```

6.4. Standard (szabványos) függvények használata

Az aritmetikai kifejezésekben függvényhívások is szerepelhetnek. Ismerkedjünk a Turbo Pascal könyvtárával, a SYSTEM unit-tal, amely minden programból elérhető anélkül, hogy erről külön intézkednénk. Tekintsük át a modul szabványos eljárásait és függvényeit funkció szerinti csoportosításban.

Az F1. függelék a függvények használatának teljes leírását adja.

Matematikai függvények

Abs(*x*) megadja az argumetum abszolút értékét
ArcTan(*x*) megadja az argumentum arkusztangensének főértékét,

***Cos*(*x*)** megadja a radiánban megadott argumentum koszinuszát,
***Exp*(*x*)** megadja az argumentum exponenciálisát,
***Frac*(*x*)** valós típusú kifejezés törtrészét adja vissza,
***Int*(*x*)** valós típusú kifejezés egész részét adja vissza valós típusként,
***Ln*(*x*)** megadja az argumentum természetes alapú logaritmusát,
Odd(*x*) ha az egész típusú kifejezés páratlan *true*, ha páros *false* értéket ad,
Pi Pi értékét adja meg,
Random véletlenszámot ad vissza,
ha nem adunk határt, akkor 0<= *x* <1 valós értéket,
***Random*(*n*)** ha adunk határt (n). akkor 0<= *x* <n egész érték között lesz a véletlenszám,
Randomize a véletlenszám generátornak véletlen kezdőértéket ad,
***Round*(*x*)** valós típusú kifejezést a legközelebbi egészre kerekíti,
***Sin*(*x*)** megadja a radiában megadott argumentum szinuszát,
***Sqr*(*x*)** megadja az argumentum négyzetét,
***Sqrt*(*x*)** megadja az argumentum négyzetgyökét,
***Tunc*(*x*)** valós típusú kifejezés nulla felé kerekített egész részét adja vissza (levágja a tizedestört részt).

Megszámlálható típusokra használható függvények

***Dec*(*x*)** csökkenti a változó értékét 1-el, (x:=x-1),
***Dec*(*x*,*n*)** csökkenti a változó értékét *n*-el, (x:=x-n),
Inc(*x*) növeli a változó értékét 1-el, (x:=x+1);
Inc(*x*,*n*) növeli a változó értékét *n*-el, (x:=x+n),
Ord(*x*) visszatér a sorszámozott kifejezés sorszámával,
***Pred*(*x*)** a sorszámozott típus előző sorszámát adja vissza,
***Succ*(*x*)** a sorszámozott típus következő sorszámát adja vissza.
***Chr*(*x*)** visszatér az *x* kódnak (sorszámnak) megfelelő karakterrel,

Sztring kezelést segítő függvények

Concat(*s1*,*s2*..) karakterláncokat fűz össze,
Copy(*s*,*index*,*count*) az *s* karakterlánc, az *index*-ben megadott sorszámú karakterétől *count* darab karaktert másol,

Delete(*s*,*index*,*count*)	az *s* karakterláncnak az index-ben megadott pozíciójától count darabot töröl,
Insert(*source*,*s*,*index*)	source karakterláncot az *s* karakterlánc *index*-ben megadott pozíciójától beszúrja,
Length(*s*)	a karakterlánc hosszát adja meg,
Pos(*substr*,*s*)	az *s* karakterláncban keresi a *subsrt*-ben megadott részláncot,
Str(*x*,*s*)	*x* egész típusú kifejezés értékét konvertálja az s sztringbe (karakterlánccá alakítja),
Str(*x*:*w*:*d*:*s*)	*x* valós típusú kifejezést *w* mezőszélességgel és *d* tizedek számával konvertálja az s sztringbe,
UpCase(*ch*)	ch kisbetű karaktert nagybetűvé alakítja,
Val(*s*,*v*,*code*)	a *s* karakterláncot konvertálja a *v* típusának megfelelő numerikus értékké. Ha a sztringben illegális karakter van, a *code* változó a hibás karakter első pozícióját tartalmazza, különben értéke 0 lesz.

A függvények vizsgálatára futtassuk le a FUGGV.PAS programot, amely bemutatja a függvények hívásait és az eredményt a képernyőre írja:

```
program fuggvenyek;
{ fuggvenyek vizsgalata }
var  x:longint;
     y:integer;
     z,code:integer;
     a,b:real;

begin
    x:=12; y:=10;
    writeln(' abs    :  'abs(12.56):7:2,
                         abs(12.56):7:2);
    writeln(' arctan:  ',arctan(pi):6:3);
    writeln(' fok    :  ',arctan(pi)*180/pi:6:3);
    writeln(' pi:       ',pi);
    writeln(' ord    :  ',ord('a'));
    writeln(' char   :  ',char(ord('a')+1));
    writeln(' cos    :  ',cos(pi));
    dec(x,1);
    writeln(' dec    :  ',x);
    dec(x,-2);
    writeln(' dec    :  ',x);
    dec(x,3);
```

```
    writeln(' dec    :  ',x);
    writeln(' exp    :  ',exp(1.0));
    writeln(' frac   :  ',frac(123.456):7:3);
    writeln(' frac   :  ',frac(-123.456):7:3);
    inc(y);
    writeln(' inc    :  ',y);
    inc(y,5);
    writeln(' inc    :  ',y);
    inc(y,-2);
    writeln(' inc    :  ',y);
    writeln(' int    :  ',int(123.456):7:1);
    writeln(' int    :  ',int(-123.456):7:1);
    writeln(' ln     :  ',ln(2.7182818285));
    writeln(' mod    :  ',16 mod 3);
    writeln(' odd    :  ',odd(13));
    writeln(' odd    :  ',odd(12));
    writeln(' ord    :  ',ord('0'));
    writeln(' pred   :  ',pred('b'));
    writeln(' succ   :  ',succ('b'));
    writeln(' succ   :  ',succ(FALSE));
    writeln(' pred   :  ',pred(TRUE));
    writeln(' pred   :  ',pred(FALSE));
    writeln(' succ   :  ',succ(TRUE));
    randomize;
    writeln(' random:   ',random(100));
    writeln(' random:   ',random);
    writeln(' random:   ',random(100));
    writeln(' round :   ',round(124.5));
    writeln(' round :   ',round(-124.3));
    writeln(' round :   ',round(-124.5));
    writeln(' sin    :  ',sin(pi));
    writeln(' sqr    :  ',sqr(2.0):5:1);
    writeln(' sqrt   :  ',sqrt(4.0):5:1);
    writeln(' trunc :   ',trunc(12.56):5);
    writeln(' trunc :   ',trunc(-12.12):5);
    val('123',z,code);
    writeln(' val    :  ',z,' ',code);
    val('123',a,code);
    writeln(' val    :  ',a:6:1,' ',code);
end.
```

A program futtási eredménye:

```
abs    :    12.56  12.56
arctan :   1.263
fok    :  72.343
pi     :   3.1415926536E+00
```

```
ord    :  97
char   :  b
cos    :  -9.9999999999E-01
dec    :  11
dec    :  13
dec    :  10
exp    :   2.7182818285E+00
frac   :     0.456
frac   :    -0.456
inc    :  11
inc    :  16
int    :  14
int    :     123.0
int    :    -123.0
mod    :   1.0000000000E+00
mod    :  1
odd    :  TRUE
ord    :  FALSE
pred   :  48
succ   :  a
suc    :  c
pred   :  TRUE
pred   :  FALSE
suc    :  TRUE
random:  67
random:   4.396538254E-01
random:  11
round :  125
round :  -124
round :  -125
sin    :   0.0000000000E+00
sqr    :   4.0
sqrt   :   2.0
trunc :    12
trunc :   -12
val    :  123 0
val    :   123.0 0
```

Gyakorlatok

1.Irjuk át az alábbi matematikai képleteket Pascal kifejezéssé

a. $$z = \frac{x + 2}{x - 1} - 3 \cdot a + 1$$

b. $$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

c. $$w = \frac{2 \cdot \sqrt{a^2 + b^2}}{a - b}$$

d. $$Y = \frac{a \cdot b + c \cdot d}{b \cdot c}$$

e. $$z = \frac{\frac{x}{y} + \frac{w}{x} \cdot m - 1}{\frac{a}{b} + 2 \cdot c}$$

f. $$z = x^3 + \frac{y^2 + 3}{\frac{x}{y} + \frac{y}{x}} \cdot \frac{2}{3}$$

g. $$w = u \cdot \left(x \cdot y + \frac{a}{b} \right) \cdot e^{\cos(x \cdot y + a)}$$

h. $$c = \frac{a \cdot b}{c \cdot d} - \sqrt{1 + \cos 2x}$$

i.

$$x = \frac{3 \cdot z^4 + y^2 + \sqrt{3,2 \cdot a \cdot b + c^2}}{\frac{z}{y} + \frac{y}{a \cdot b} + \sqrt{\sin x^2 + 1}}$$

j. $w = x + \log_4 (2 \cdot x + 3 \cdot x^2)$

k. $z = x^4 + 2 \cdot \log_2 x + 4 \cdot 3^{5 \cdot x}$

2. A Pascal kifejezést írjuk vissza képletté

```
a.  w:=(a+b)/x/y+2*x/y;
b.  z:=x*y/a*b+i*(k+1)/2;
c.  s:=sqt(x)*x+y/x*2;
d.  y:= sin(x)+cos(2.0*k*x)+exp(-1.0/x/x);
e.  w:=(sqrt(x)*x+2)/(y/x+2*x/y/z+1.2);
f.  z:=sqrt(1.+sqrt(sin(x))*sin(x)+exp(-a*x);
g.  z2:=(a*sin(2.*n*x)+b*cos(2*n*x))/(sqrt(a*x)+2*b*x*y+sqrt(c*y));
h.  w:=exp(5*ln(x))+1.0/y/y;
i.  y:=3*y-exp(6*ln(x*x+2));
j.  w:=ln(a+2)/ln(10)+4.5*b;
```

3. Adja meg a következő aritmetikai kifejezések értékét

```
a.  round(7.81) + 11 mod 3
b.  trunc(8.9) + 12 div 5
c,  6*4 div 3*5
d,  (6*4) div (3*5)
e   6* (4 div 3)*5
```

4. Adja meg a relációk értékét

a. 8 div 3 = 1
b. (3 > 2) and (4.2 < 1.5)
c. i:=5; j:=-1; m:=4; n:=2;
((i+1) <= 7) or (j <> m*n)

5. Adja meg a bitenkénti műveletek értékét

a. not 32
b. 42 and 18
c. 29 or 33
d. 29 xor 33
e. 45 shl 2
f. 45 shr 3

5. Értékelje ki az alábbi logikai feltételeket

A:=true; B:=false: C:=true; D:=false;

a. A and B or C and D
b. A and (B or C) and D
c. A or B and C or D
d. (A or B) and (C or D)

6. Mit látunk a képernyőn?

```
program mi_lesz_a_kepernyon;
var
     a1,a2,a3: string;
begin
     a1:= 'lesz';
     a2:= 'eredemeny';
     a3:= 'Jo '+a1+' az '+a2+' ?';
     writeln(a3);
end.
```

Az eredményünket ellenőrízhetjük a MI_LESZ.PAS program futtatásával.

7. Mi lesz a tartalma a HALMAZ3.PAS programban a

teljes_halmaz, ketto_harom_oszt és a ketto_negy_oszt

halmaz változóknak?

```
program halmaz3;
type
     szamok = 1..10;
     sz10   = set of szamok;
var
     paratlan,paros,ketto_oszthato,harom_oszthato,
     negy_oszthato,ketto_harom_oszt,
     ketto_negy_oszt, teljes_halmaz: sz10;
begin
     paratlan:=[1,3,5,7,9];
     paros   :=[2,4,6,8,10];
     ketto_oszthato:= [2,4,6,8,10];
     harom_oszthato:= [3,6,9];
     negy_oszathato:= [4,8];

     teljes_halmaz:= paratlan+paros;
     ketto_harom_oszt:= ketto_oszthato*harom_oszthato;
     ketto_negy_oszt := negy_oszthato*ketto_oszthato;

end.
```

Ha kiértékeltük a halmaz3 programot, akkor ellenőrzésképpen futtassuk le a HALMAZ3I.PAS programot, amely kiírja a kérdéses halmazváltozók tartalmát.

8. Értékeljük ki a HALMAZ4.PAS programot, mit látunk a képernyőn?

```
program halmaz4;
type
    teljes_abc = (a,b,c,d,e,f,g,h,i,j,k,l,m,
                  n,o,p,q,r,s,t,u,v,w,x,y,z);
    abc_halmaz = set of teljes_abc;
var
    abc : abc_halmaz;
    mgh : abc_halmaz;
    msh : abc_halmaz;
```

```
begin

abc:=[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,g,r,s,t,u,v,w,x,y,z];

mgh:= [a,e,i,o,u];
msh:= abc - mgh;

     if u in msh then writeln('u  benne van a halmazban')
                 else writeln('u  nincs benne a halmazban');
     if d in msh then writeln('d  benne van a halmazban')
                 else writeln('d  nincs benne a halmazban');
end.
```

Mit tartalmaz az *msh* halmaz?

7. ALAPVETŐ I/O MŰVELETEK

7.1. Irás képernyőre: Write és Writeln eljárások

A Pascal programozásban a *standard output* (szabványos kimenet) periféria a képernyő. A képernyőre két eljárással irhatunk:

Write(paraméterek);
Writeln(paraméterek);

A paramétereket vesszővel elválasztva kell felsorolni.
A paraméterek lehetnek:

- szöveg '' aposztrofok között,
- változónevek (csak egyező típusúak),
- konstansok,
- aritmetikai kifejezések,
- logikai kifejezések,
- függvényhívás.

A ***Write*** és a ***Writeln*** eljárások között az alábbi különbség van:

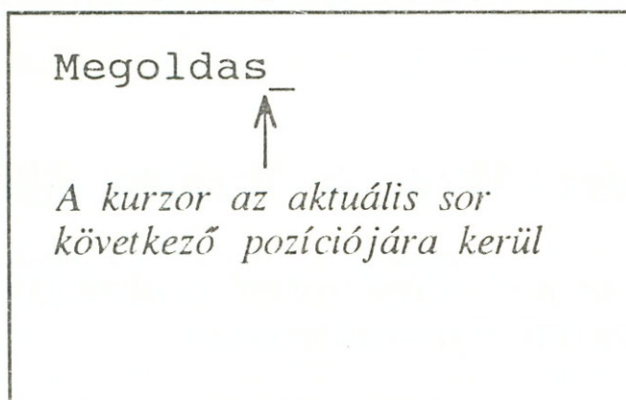
- a ***write*** eljárás a kurzor által mutatott pozícióra ír, és a kurzort a kiírás utáni pozícióban hagyja.

- a ***writeln*** eljárás is a kurzor által mutatott pozíciótól ír, azonban a paramétereknek a képenyőre való kiírása után a következő sor elejére állítja a kurzort.

- a paraméter nélküli ***writeln;*** eljárással üres sort írhatunk a képernyőre.

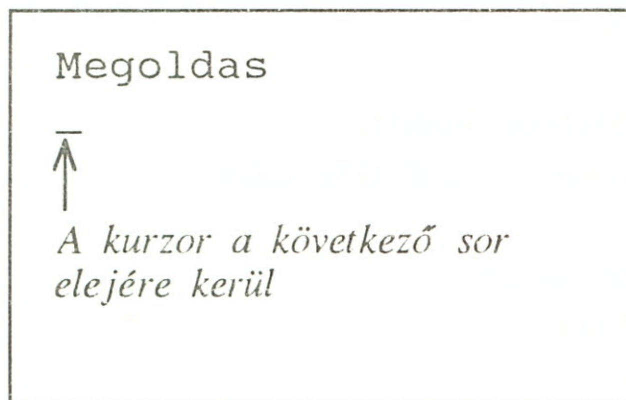
A 7.1. ábra szemléletesen bemutatja a különbséget a két eljárás között.

```
Write('Megoldas');
```



a.

```
Writeln('Megoldas');
```



b.

7.1. ábra Különbség a. *Write* eljárás és a b. *Writeln* eljárás között

7.1.1. Szöveg kiírása a képernyőre

A ***write*** és a ***writelen*** eljárásokban a szöveg konstanst '' között kell megadni.

A mintaprogramokban az eredmény kiírása előtt a képernyő előző tartalmát letöröljük, hogy a program többszöri futtatása esetén az eredmények üres képernyőn jelenjenek meg. A képernyő törlésére szolgál a ***clrscr*** paraméter nélküli eljárás, amely a ***crt*** **unit**-ban található. A ***crt*** könyvtárat a **uses** kulcsszó után kell meghivatkozni.

Futtassuk le az IR1.PAS programot:

```
program ir1;
uses crt;
begin
    clrscr;
    writeln('Szoveg irasa kepernyore');
    writeln('Ez a program fejlece');
    writeln;
    write('Az ');
    write('elso ');
    write('sor. ');
    writeln;
    write('A masodik ');
    writeln('sor.');
    writeln('Harmadik sor.'#13'Negyedik sor');
    writeln(#7#7'Ketszer csenget');
end.
```

A program eredménye:

```
Szoveg irasa kepernyore
Ez a program fejlece

Az elso sor.
A masodik sor.
Harmadik sor.
Negyedik sor.
Ketszer csenget
```

A példában is látható, hogy a ***writeln*** eljárás a sort zárja és a kurzort a következő sor elejére állítja. A ***write*** segítségével kiírt információ a kurzorral jelzett sorba kerül. Mivel a ***write*** nem zárja le sort, a következő ***write*** az előzővel azonos sorba ír. Gyakran szükséges a programban, hogy a megkezdett sorban írhassuk tovább a program által később kiszámolt eredményt.

A program utolsó ***writeln*** eljárása két darab #7 csengő (bell) kódot tartalmaz, ha a programot lefuttatjuk kétszer fog csengetni és a képernyőn a 'Ketszer csenget' szöveg jelenik meg új sorban.

7.1.2. Egész típusú eredmények kiiratása

A Turbo Pascalban a változók kiírására létezik egy ún. *standard* formátum. Az egész számok, egész típusú változók valamint egész típusú kifejezések értékét szorosan egymás mellé írja a ***write*** és a ***writeln*** eljárás, csak a negatív előjelet írja ki, a pozitív előjel helyén üres helyet sem hagy ki. Természetesen ilyen kiírásnál az eredmény kiértékelhetetlen, ilyenkor a ***writeln***(i,j,k); eljárás hívás helyett ***writeln***(i,' ',j,' ',k); hívást alkalmazhatjuk.

Kiiratás mezőszélességgel

write(*parameter: n*);
writeln(*parameter: n*);

ahol *n* egész szám, a mezőszélesség

A ***write*** vagy ***writelen*** eljárásban a paraméter után kettőaponttal (:) elválasztva megadott egész számot *mezőszélességnek* nevezünk. A mező-szélesség határozza meg, hogy a számkiírás mekkora helyet foglaljon le a képernyőn.

Ha a mezőszélesség nagyobb, mint a szám jegyeinek száma, hozzáadva még az előjelét, akkor természetesen megfelelő számú üres hely keletkezik a szám előtt, mert a mező jobb széléhez igazítva történik a kiírás.
Például írjuk ki az alábbi konstansokat:

```
     writeln(-100,22);
-10022
     writeln(-100,' ',22);
-100 22
     writeln(-100:6,22:4);
   -100  22
```

A -100 kiírásánál a szám előtt 3, a 22 kiírásánál a szám előtt 2 üres hely keletkezik.
Megjegyezzük, hogy ha olyan mezőszélességet adunk, amelybe nem fér el a szám, akkor a Turbo Pascal nem ad hibajelzést, hanem áttér a standard kiírási formátumra.

Értékeljük ki, majd futtassuk le az IR2.PAS programot:

```
program ir2;
uses crt;
var
   i,j,k: integer;
begin
   clrscr;
   i:=12; j:=-12; k:=+100;
   writeln('Standard kiiratas);
   writeln(i,j,k);
   writeln('Eredmeny szoveges kiiratasa');
   writeln('i=',i,' j=',j,' k=',k);
   writeln('Kiiratas mezoszelesseggel');
   writeln(i:4,j:5,k:6);
   writeln('i=',i:3,' j=',j:4,' k=',k:4);
   writeln('Helytelen mezoszelesseg megadasa');
   writeln(i:1,j:2,k:3);
   writeln('Kifejezes kiiratasa');
   writeln('k+i=',k+i);
   writeln('Konstansok kiiratasa');
   writeln(12,-100);
   writeln(12:4,-100:5);
end.
```

A program eredménye:

```
Standard kiiratas
12-12100
Eredmeny szoveges kiiratasa
i=12 j=-12 k=100
Kiiratas mezoszelesseggel
  12  -12   100
i= 12 j= -12 k= 100
Helytelen mezoszelesseg megadasa
12-12100
Kifejezes kiiratasa
k+i=112
Konstansok kiiratasa
12-100
  12 -100
```

7.1.3. Valós eredmények kiiratása

A valós, tehát lebegőpontos számok kiiratásának is létezik standard formátuma. A valós szám mindig egy egészre normálva, Turbo Pascal 5.0, 5.5 változatnál 10 tizedesjegyre kerül kiírásra úgy, hogy a 10 hatvány helyett E és a kitevő két helyiértéken kerül kiírásra, pl.

```
     0.001      1.0000000000E-03
    -1.2       -1.2000000000E+00
    11.3        1.1300000000E+01
 92560.43       9.2560430000E+04
```

Turbo Pascal 6.0 verzióban a tizedek száma 15 és a kitevő 4 jegyen kerül ábrázolásra:

```
-1.123456789012345E-0001
 2.123456789012345E+0000
```

A valós számok kiírásánál is adhatunk a mezőszélességet. A mezőszélesség kiszámításánál a következőket kell figyelembe venni: 3 hely szükséges a szám előjele, az egészrész és a tizedespont miatt, a exponens számára a verziótól függően 4 vagy 6 hely szükséges(E, előjel és a kitevő). Tehát a mezőszélességnek verziótól függően nagyobbnak kell lenni 7-nél vagy 10-nél. Ha kevesebbet adunk meg, akkor hibajelzés nélkül lebegőpontosan egy egész és egy tized értéket fog írni normált alakban a Turbo Pascal.

Fixpontos kiiratási formátum:

> ***write****(parameter: m:n)*;
> ***writeln****(parameter:m:n)*;

ahol *m* egész szám, mezőszélesség,
n egész szám, a tizedek száma,

Van még egy kiíratási forma, ahol a valós számot fixpontosan irathatjuk ki, tehát az egész és a tizedes részeivel az exponens rész nélkül. Ebben az esetben a mezőszélességet újabb : (kettőspont) követi, mely után megadott egész szám a tizedek számát fogja meghatározni. A tizedek száma a mezőszélesség részét képezi, tehát beleértendő a mezőszélességbe.

Értékeljük ki, majd futtassuk le az IR3.PAS programot:

```
program ir3;
uses crt;
var
   a,b,c,d,e: real;
begin
 clrscr;
 a:=12.56; b:=-12.56; c:=0.5; d:=1.0e-3;
 e:=1.63e+3;
 writeln('Standard kiiratas');
 write(a); write(b); writeln;
 writeln(c,d);
 writeln(e);
 writeln('Kiiratas mezoszelesseggel normalt alakban')
 writeln('a=',a:9,' b=',b:10);
 writeln('Helytelen mezoszelesseg megadasa');
 writeln('a=',a:4,' b=',b:4);
 writeln('Kiiratas fixpontos alakban');
 writeln('a=',a:9:2,' b=',b:10:3);
 writeln('c=',a:2:1,' b=',d:10:0);
 writeln('Konstansok kiiratasa');
 writeln(123.456,123.456:10,' ',-123.456:10:3,' ',
         0,001:7:3);
 writeln(123.456:9);
 writeln(123.456:9:4);
 writeln('Kifejezesek');
 writeln('a+c=',a+c:8:2);
 writeln(' sqr(2)=',sqr(2),' sqr(2.0)=',sqr(2.0));
end.
```

A program eredménye:

```
Standard kiiratas
 1.2560000000E+01-1.2560000000E+01
 5.0000000000E-01 1.0000000000E-03
 1.6300000000E+03
Kiiratas mezoszelesseggel normalt alakban
a= 1.26E+01 b=-1.256E+01

Helytelen mezoszelesseg megadasa
a= 1.3e+01 b=-1.3E+01
Kiiratas fixpontos alakban
a=    12.56 b=   -12.560
a=12.6 b=         -13
Konstansok kiiratasa
 1.2345600000E+02 1.2345E+02   -123.456  0.001
```

```
Kifejezesek
a+c=   13.06
sqr(2)=4 sqr(2.0)= 4.0000000000E+00
```

7.1.4. Boolean típusú eredmények kiiratása

A **boolean** konstansok, változók kifejezések kiiratása hasonlóan történik, mint az egész számoknál, mivel a standard kiiratás pontosan annyi karakterre vonatkozik, mint maga a *true* vagy a *false* konstans, tehát a kiírási felsorolásnál az eredmény szorosan egymás mellett keletkezik. Itt is megadhatunk mezőszélességet, ekkor a *true* vagy a *false* jobbra igazítva íródik ki.

Példaként, nézzük meg az IR4.PAS programot, amely demonstrálja a **boolean** típusú kiiratásokat:

```
program ir4;
uses crt;
var
   b1,b2: boolean;
begin
   clrscr;
   writeln('Boolean valtozok kiiratasa');
   b1:=true; b2:=false;
   writeln('Standard kiiratas');
   writeln(b1,b2);
   writeln('Mezoszelesseg megadasa');
   writeln(b1:6,b2:6);
   writeln('Helytelen mezoszelesseg megadasa');
   write(b1:3); write(b2:4);
   writeln;
   writeln('Logikai kifejezesek kiiratasa');
   writeln('2 <3 : ',2 < 3,' b1 and b2:',b1 and b2);
   writeln('Logikai konstansok kiiratasa');
   writeln(true,' ',false);
end.
```

Értékeljük ki a program eredményét:

```
Boolean valtozok kiiratasa
Standard kiiratas
TRUEFALSE
```

```
Mezoszelesseg megadasa
  TRUE FALSE
Helytelen mezoszelesseg megadasa
TRUEFALSE
Logikai kifejezesek kiiratasa
2 < 3 : TRUE b1 and b2:FALSE
Logikai konstansok kiiratasa
TRUE FALSE
```

Foglaljuk össze a különböző típusú változók mezőszélességgel való kiiratását (KIIR.PAS):

```
program kiir;
uses crt;
begin
   clrscr;
     writeln('#':1);
     writeln('#':3);

     writeln(1992:1);
     writeln(1992:4);
     writeln(1992:5);
     writeln(1992:7);
     writeln(-1992:1);
     writeln(-1992:3);
     writeln(-1992:5);
     writeln(-1992:7);
     writeln('Nyomj Enter-t!');
     readln;
   clrscr;
     writeln('hello':1);
     writeln('hello':3);
     writeln('hello':5);
     writeln('hello':7);

     writeln(true:1);
     writeln(true:3);
     writeln(true:5);
     writeln(true:7);
     writeln(false:1);
     writeln(false:3);
     writeln(false:5);
     writeln(false:7);
     writeln('Nyomj Enter-t!');
     readln;
   clrscr;
```

```
     writeln(123.456:1:1);
     writeln(123.456:1:3);
     writeln(123.456:1:4);
     writeln(123.456:5:1);
     writeln(123.456:6:1);
     writeln(123.456:7:1);

     writeln(-123.456:1:1);
     writeln(-123.456:1:3);
     writeln(-123.456:1:4);
     writeln(-123.456:5:1);
     writeln(-123.456:6:1);
     writeln(-123.456:7:1);

     writeln(567.8:1);
     writeln(567.8:8);
     writeln(567.8:9);
     writeln(567.8:10);
     writeln(567.8:11);

     writeln(-567.8:1);
     writeln(-567.8:8);
     writeln(-567.8:9);
     writeln(-567.8:10);
     writeln(-567.8:11);

     writeln('Nyomj Enter-t!');
     readln;

end.
```

char :

w	`write('#':w);`
1	`#`
3	`  #`

integer :

w	`write(1992:w);`	`write(-1992:w);`
1	`1992`	`-1992`
4	`1992`	`-1992`
5	` 1992`	`-1992`
7	`   1992`	`  -1992`

strings :

w	`write('hello':w);`
1	`h`
3	`hel`
5	`hello`
7	`  hello`

boolean :

w	`write(true:w);`	`write(false:w);`
1	`true`	`false`
3	`true`	`false`
5	` true`	`false`
7	`   true`	`  false`

real : w	f	`write(123.456:w:f);`	`write(-123.456:w:f);`
1	1	`123.5`	`-123.5`
1	3	`123.456`	`-123.456`
1	4	`123.4560`	`-123.4560`
5	1	`123.5`	`-123.5`
6	1	` 123.5`	`-123.5`
7	1	`  123.5`	` -123.5`

w	`write(567.8:w);`	`write(-567.8:w);`
1	` 5.7E+02`	`-5.7E+02`
8	` 5.7E+02`	`-5.7E+02`
9	` 5.68E+02`	`-5.68E+02`
10	` 5.678E+02`	`-5.678E+02`
11	` 5.6780E+02`	`-5.6780E+02`

7.2. Olvasás billentyűzetről: Read, Readln eljárások

A Turbo Pascalban a *standard input* (szabványos bemeneti) periféria a billentyűzet. A ***read*** és a ***readln*** eljárások felfüggesztik a program futását és a klaviatúráról az ENTER billentyű lenyomásáig olvassák az adatokat.

> ***read***(*paraméterek*);
> ***readln***(*paraméterek*);

A paraméterek:

- csak változók lehetnek, kivételt képeznek a **boolean** változók, mivel azok értékét nem lehet beolvasni billentyűzetről.

Először vizsgáljuk meg a ***read*** eljárás működését. A ***read*** eljárás felfüggeszti a program futását és várja az adatok billentyűzeten való begépelését befejezve az ENTER leütésével. A ***read*** utasítás a begépelt adatokból sorra értéket ad a paramétereinek. Ha kevesebb adatot adtunk meg, mint amennyi a ***read*** paraméterlistáján volt, akkor további adatmegadásra vár az eljárás. Ha viszont több adatot gépeltünk be, mint amennyi szükséges volt a ***read*** eljárásnak, akkor a következő ***read*** eljárás a további adatokat átadja a paramétereinek.

A numerikus adatokat legalább egy üres hely (space) vagy az ENTER választja el egymástól.

```
program io1;
var
   szam,szam1 : integer;
begin
  write('Kerek egy egesz szamot: ');
  read(szam);
  writeln('A beolvasott elso szam   : ',szam);
  write('Kerek egy egesz szamot : ');
  read(szam1);
  writeln('A beolvasott masodik szam: ', szam1);
end.
```

Futtassuk le az IO1.PAS programot az alábbi adatokkal:

```
Kerek egy egesz szamot: 2 3
A beolvasott elso szam  : 2
Kerek egy egesz szamot: A beolvasott masodik szam: 3
```

A fenti futtatásnál a 23 helyett véletlenül 2 3-at adtunk meg ENTER-rel terminálva, amely két külön adatnak bizonyul, így a ***read*** eljárás a *szam* változóba betöltötte a 2-es értéket. Ellenőrzésképpen vissza is írattuk a képernyőre a '*A beolvasott elso szam :* ' szöveg mellett és valóban a 2 jelenik meg. Mivel a következő eljáráshívás szintén ***read*** és mivel a ***read*** tovább tud olvasni az input pufferből és így a 3-as számot beolvasva nem függeszti fel a program futását.

A '*Kerek egy egesz szamot:* 'szöveg után nem vár adatvitelre, ezért ebben a sorban jelenik meg az ellenőrző szöveg '*A beolvasott masodik szam:* ' mellett a 3, amelyet tévesen adtunk meg. Futtassuk le a programot az adatok helyes megadásával is.

Az IO1.PAS programot módosítsuk úgy, hogy a ***read*** eljárás helyett ***readln*** eljárást hívást alkalmazzunk. Futtassuk le az IO2.PAS programot.

```
program io2;
var
   szam,szam1 : integer;
begin
  write('Kerek egy egesz szamot: ');
  readln(szam);
  writeln('A beolvasott elso szam   : ',szam);
  writeln('Kerek egy egesz szamot : ');
  readln(szam1);
  writeln('A beolvasott masodik szam: ', szam1);
end.
```

Futtassuk le a programot az alábbi adatokkal:

```
Kerek egy egesz szamot:  3 5
A beolvasott elso szam   :  3
Kerek egy egesz szamot : 4
A beolvasott masodik szam: 4
```

Kiértékelve az eredményt láthatjuk, hogy a 3 5 ENTER került be az input pufferba. A ***readln*** eljárás egy sort olvas be, egyetlen paraméterének ad értéket. Tehát a *szam* változó felveszi a 3-as értéket, amely a *A beolvasott elso szam : '* szöveg mellett ellenőrzésképpen visszaírásra kerül.

A következő ***readln*** nem folytatja az olvasást az előzőleg begépelt adatsorból, hanem új begépelésre vár, ami az input pufferba maradt az az adat elveszik a beolvasás számára. 4 és ENTER begépelése után a ***readln*** a szám1 változónak a 4-es értéket adja, amelyet újra kiíratunk.
Megjegyezzük, hogy a paraméter nélküli ***readln*** eljárással szintén egy sort olvasunk, az adatok elvesznek, mivel nem kerülnek tárolásra, ezzel a módszerrel sort tudunk kihagyni a beolvasandó adatok közül.

Olvassunk karaktereket először ***read*** utasítással:

```
program io3;
uses crt;
var
   c1,c2,c3,c4: char;
begin
   clrscr;
   writeln("Karakterek beolvasasa: ');
   read(c1); read(c2); read(c3); read(c4);
   writeln('A beolvasott karakterek:');
   writeln('c1=',c1,'c2=',c2,'c3=',c3,c4=',c4);
end.
```

Az IO3.PAS program futtatási eredménye:

```
Karakterek beolvasasa: abcdef
A beolvasott karakterek:
c1=a c2=b c3=c c4=d
```

Láthatjuk, hogy az *abcdef* karaktersorozatból a ***read*** eljárás folyamatosan olvas és csak annyi karaktert vesz ki az input pufferból, amennyi paraméternek kell értéket adni.

Az IO4.PAS programban a karaktereket ***readln*** eljárással olvassuk be:

```
program io4;
uses crt;
var
   c1,c2,c3,c4: char;
begin
   clrscr;
   writeln('Karakterek beolvasas');
   readln(c1); readln(c2); readln(c3); readln(c4);
   writeln('A beolvasott karakterek');
   writeln('c1=',c1,'c2=',c2,'c3=',c3,'c4=',c4);
end.
```

A program futási eredménye:

```
Karakterek beolvasasa:
abcd
efg
hij
kl
A beolvasott karakterek
c1=a c2=e c3=h c4=k
```

Láthatjuk, hogy hiába gépeltük be az *abcd* karaktersorozatot, hogy a négy karakter változónak adjunk értéket, mivel a *readln* eljárás paraméterlistáján csak egyetlen karakter változónak kell, hogy értéket adjon, így az új sorig beolvasott adatsorozatból az első karaktert adta értékül a *c1* változónak. A következő ***readln*** már nem olvas az előző sorból, ezért újabb karakter bevitelre vár. Látható a '*A beolvasott karakterek*' fejléc alatt visszairatott karakterváltozók tartalmából, hogy a ***readln*** mindig az új sorban megadott első karakter értékét vette át.

Most módosítsuk újra a programot úgy, hogy a beolvasandó 4 karakter változóját egyetlen ***readln*** paraméterlistáján adjuk meg. Futtassuk le az IO5.PAS programot.

```
program io5;
uses crt;
var
   c1,c2,c3,c4: char;
begin
   clrscr;
```

```
    writeln('Karakterek beolvasas');
    readln(c1,c2,c3,c4);
    writeln('A beolvasott karakterek');
    writeln('c1=',c1,'c2=',c2,'c3=',c3,'c4=',c4);
end.
```

A program futtatási eredménye:

```
Karakterek beolvasasa:
abcd
A beolvasott karakterek
c1=a c2=b c3=c c4=d
```

Nézzük meg miben tér el a program futtatási ereménye az előzőtől. Az *abcd* ENTER begépelése után a ***readln*** eljárás a paraméterlistáján lévő négy karakterváltozónak mindegyikének tudott értéket adni az input pufferből.

Nézzük meg, ha másképpen adtuk volna meg az adatokat:

```
Karakterek beolvasasa:
ab
c
d
A beolvasott karakterek
c1=a c2=b c3=c c4=d
```

Ennél az adatmegadásnál látható, hogy a ***readln*** a sorvégéig olvasva csak két változójának, a *c1* és *c2* változónak tudott értéket adni, mivel a paraméterlistáján van még két változó, így újra beolvasásra vár. Ezután leütöttük a *c* karaktert, a ***readln*** megint új sorig olvasva a *c3* változót töltötte fel, de hiányzik a *c4* változó adata, így újraolvasásra vár.

Futtassuk le az IO6.PAS programot az alábbi adatokkal, értékeljük ki, hogy mit kapunk eredményül:

```
5.1  6.7  3  4
8.1 5 6
8 9
```

```
program io6;
var
   x,y: real;
   n,m: integer;
begin
   writeln('Numerikus adatok beolvasasa');
   readln(x); readln(y); read(n); read(m);
   writeln('A beolvasott adatok');
   writeln('x=',x,' y=',y,' n=',n,' m=',m);
end.
```

A program futtatási eredménye:

```
Numerikus adatok beolvasasa
5.1  6.7  3  4
8.1 5 6
8 9
A beolvasott adatok
x=5.1 y=8.1 n=8 m=9
```

A program eredményének magyarázata:

A ***readln***(x) az első sort olvasta be és az *x* az első értéket, az 5.1-et vette fel, a többi adat a sorból elveszett. A következő ***readln***(y) újra új sort olvasott és az *y* a második sor első adatát, a 8.1 értéket vette fel. Itt is elvesztek a további adatok a sorból. A harmadik sorból a 8 és a 9 adatokat a két ***read*** eljárás olvasta be és így kapott értéket az *n* és az *m* változó.
A program beolvasó sorait a következőképpen módosítjuk:

```
     readln(x,y,n,m);
          vagy
     read(x,y,n,m);
```

Az IO7.PAS program:

```
program io7;
uses crt;
var
   x,y: real;
   n,m: integer;
begin
   clrscr;
   writeln('Numerikus adatok beolvasasa');
   readln(x,y,n,m);
```

```
   writeln('A beolvasott adatok');
   writeln('x=',x,' y=',y,' n=',n,' m=',m);
end.
```

akkor az adatmegadás a következő lehet:

```
5.1 8.1 5 6
```

A program futtatási eredménye:

```
Numerikus adatok beolvasasa
5.1 8.1 5 6
A beolvasott adatok
x=  5.10 y=  8.10 n=8 m=9
```

vagy akár minden szám lehet külön sorban is

```
5.1
8.1
5
6
```

A program futási eredménye:

```
Numerikus adatok beolvasasa
5.1
8.1
5
6
A beolvasott adatok
x=  5.10 y=  8.10 n=8 m=9
```

Az első adatmegadásnál két valós és két egész számot adtunk meg egy üres hellyel elválasztva egy sorban ENTER-rel terminálva. A ***readln*** eljárás paraméterlistáján *x,y* két valós és *n,m* két egész típusú paraméternek kell értéket adni. Az input pufferbe bekerülő számadatok pontosan megfelelnek a paraméterek számának és típusaiknak, így az eljárás sorban értéket tud adni a paramétereinek az egyetlen megadott adatsorból.

A másik adatmegadásnál a számokat ENTER leütésével adtunk meg, tehát minden adat új sorban van. A ***read*** illetve a ***readln*** ennél az adatmegadásnál azonosan működik. Mindkét eljárásnak a paraméter listáján négy változónak kell értéket adni, mivel csak egy adatot adtunk meg, így egyedül az *x*

változó veheti fel az értékét. Az eljárások ezért további adatmegadásra várnak és a külön sorban megadott értékeket sorra átadják a paramétereiknek.

A példaprogramok nem figyelmeztették a felhasználót, hogy milyen és mennyi adatot kell megadni a programnak, ezek csak abból a célból készültek, hogy megmagyarázzák a ***read*** és a ***readln*** eljárások működése közötti különbséget. Természetesen a programoknak párbeszédes formában kell működniök, hiszen egy üres képernyő és villogó kurzor mellett, mégha magunk is írtuk a programot, az adatok megadási sorrendjét elfelejthetjük. A Pascal beolvassa az egészként megadott valós számot, de ha egész számot vár és tizedes pontot talál a beolvasott számban, akkor hibajelzést ad.

Példaként nézzük meg egy programrészletet, amely párbeszédes formában olvassa be a különböző típusú adatokat:

```
program io8;
uses crt;
var
   nev : string;
   ev  : integer;
   suly: real;
   m   : real;
begin
   clrscr;
   write('Szemely neve   : '); readln(nev);
   write('Szuletesi eve : '); readln(ev);
   write('Testsulya [kg]: '); readln(suly);
   write('Magassaga [cm]: '); readln(m);
   writeln('A beolvasott adatok');
end.
```

Gyakorlatok

1. Feladatként egészítsük ki az IO8.PAS programot, hogy a beolvasott adatokat írja vissza a képernyőre (IO8M.PAS) .

2. Az IO9.PAS programot futtassuk le az alábbi adatokkal:

```
15  200
1.003
g
adatok
```

Mit fogunk látni a képernyőn?

```
program io9;
uses crt;
var
   szoveg: string;
   i,j   : integer;
   x     : real;
   kar   : char;
begin
   clrscr;
   write(' i j  =');
   readln(i,j);
   write('   x  =');
   readln(x);
   write(' kar  =');
   readln(kar);
   write('szoveg=');
   readln(szoveg);
   writeln;
   writeln('A beolvasott adatok:');
   writeln('i:',i,' j : ',j);
   writeln('x     :',x);
   writeln('kar   : ',kar);
  writeln('szoveg:', szoveg);
end.
```

3. Írjunk programot az alábbi adatok beolvasására. Az adatok számára megfelelő típusú változókat deklaráljunk, a beolvasott adatokat írjuk vissza a képernyőre:

```
0 100
 3
1.2  3.4  5.7
q
adat
```

4. Értékeljük ki az alábbi programot, ellenőrizzük az FG2.PAS program erdményével:

```
program fg2;
begin
  writeln(' 1.= ',chr(succ(ord('e'))+3):4);
  writeln(' 2.= ',chr(ord(pred('c'))-1):4);
  writeln(' 3.= ',21 mod (sqr(9))+5*2:4);
  writeln(' 4.= ',('b'>'c') and (8<9) or (succ(false)):4);
  writeln(' 5.= ',('a' < succ('a')) and odd(13):4);
  writeln(' 6.= ',12 mod 77:4);
  writeln(' 7.= ',77 mod 12:4);
  writeln(' 8.= ',odd(33):4);
  writeln(' 9.= ',odd(12):5);
end.
```

5. Értékeljük ki az alábbi programot, mit ír ki a képernyőre, majd ellenőrizzük le a MITIR.PAS program futási eredményével.

```
program mitir;
begin
  writeln(2*6 div 4*3:
           48 mod 3-(50-(50 div 4)*4)+6);
end.
```

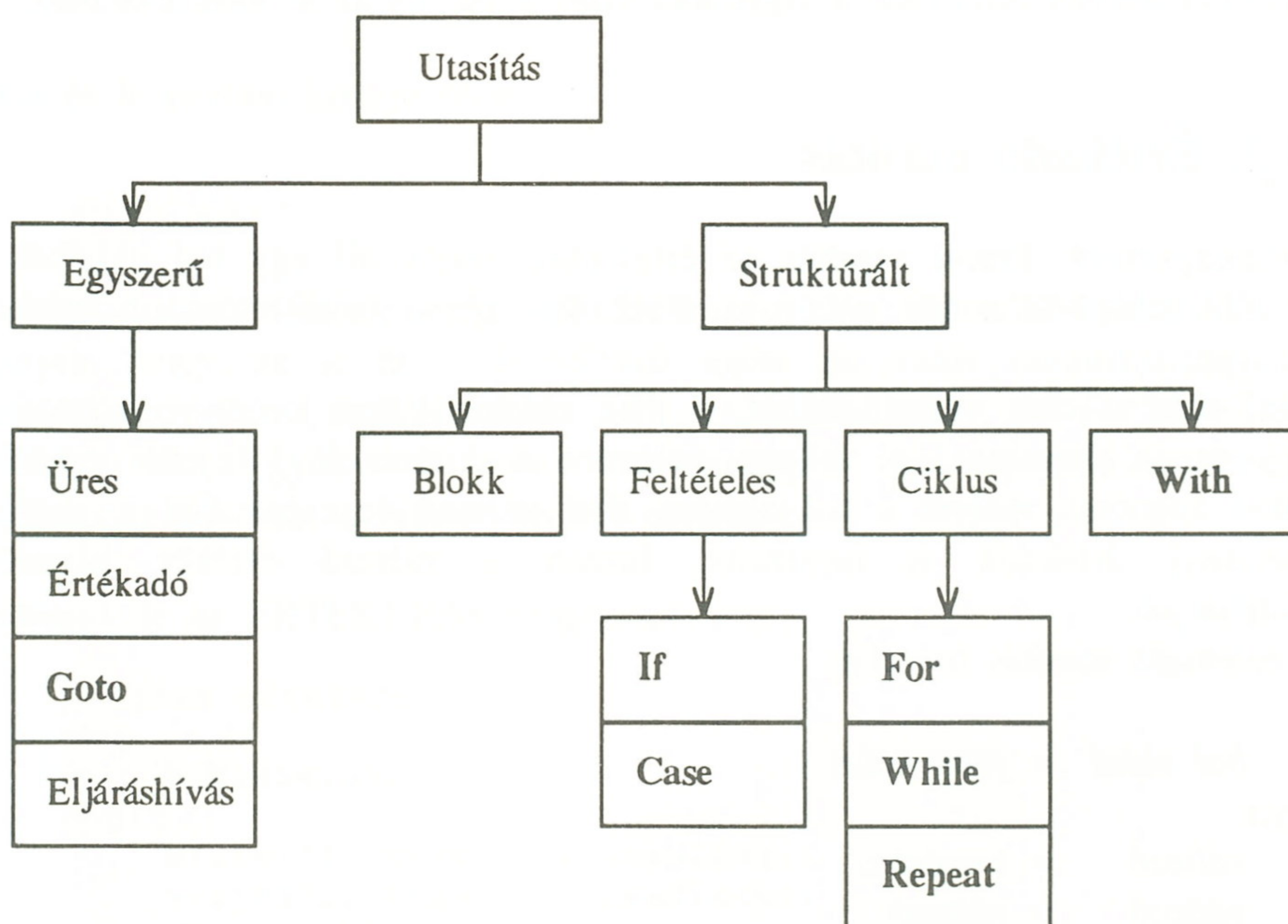
6. Először értékeljük ki a programot, azután ellenőrizzük le az IR5.PAS program futási eredményével.

```
(* ir5.pas *)
program ir5;
var
   sz    : string;
   i,j   : integer;
   a,b,c : real;
   ch    : char;
   b1,b2 : boolean;
begin
   sz:='Megoldas'; ch:='X'; a:= 3.1; b:=2.42; i:=5; j:=-1;
   writeln(sz);
   writeln; write(ch); writeln(' -> Y');
   writeln(a,b); write(a:8,' ',b:10); writeln;
   writeln('a+b=',a+b:7:2);
   b1:=i < j; b2:= b1 or (a > b);
   write(b1, ' ',b2); writeln(' ',b1 and b2);
   writeln('i*j=',i*j);
end.
```

8. A PASCAL NYELV UTASÍTÁSAI

Az utasításokat csoportosíthatjuk egyszerű és struktúrált utasításokra. Az egyszerű utasítások az üres, az értékadó, a **goto** utasítások és az eljáráshívás.

A struktúrált utasításokhoz soroljuk a blokk, a feltételes (**if**, **case**), a ciklus (**for**, **while** és **repeat-until**) és a **with** utasításokat. Az utasítások ilyen csoportosítását a 8.1 ábra mutatja.



8.1 ábra Az utasítások csoportosítása

8.1. Egyszerű utasítások

8.1.1. Üres utasítás

Az utasítás neve is jelzi, hogy semmilyen tevékenységet nem jelent. Akkor használjuk, ha valahol szintaktikailag utasításnak kell lennie, pl. **case** utasításban egy függvényhívás hiányzik és a függvényt még nem írtuk meg, akkor a ; (pontosvessző) ezt helyettesíti.

Többször okoz gondot a feleslegesen kitett utasítást lezáró pontosvesző, erre a megfelelő helyen felhívjuk a figyelmet (lásd a **for** és az **if** utasításoknál).

8.1.2. Értékadó utasítás

A legalapvetőbb Pascal utasítás az értékadás, amely áll egy bal oldalból és egy jobb oldalból, a két oldalt az értékadás jele a := (kettőspont egyenlő) köti össze.
A bal oldalon csak változó állhat, a jobb oldalon állhat konstans, változó és kifejezés. A konstans, a változó valamint a kiértékelt kifejezés egyetlen értéket képvisel, melyet a bal oldalon lévő változó vesz fel. Ezt az értéket pillanatnyi értéknek is nevezzük, hiszen a változó értékét bármikor felülírhatjuk.
Az értékadó utasítás formája:

bal oldal := jobb oldal;

vagyis

változó := konstans;
változó := változó;
változó := kifejezés;

Például:

```
x:=3.5;            { konstans , az x pillanatnyi tartalma 3.5 }
y:=w;              { változó }
z:=x*y+sqr(x);     { kifejezés }
x:=x+1;            { az x pillanatnyi tartalma 4.5-re változott }
```

8.1.2.1. Aritmetikai értékadás

Először foglalkozzunk az aritmetikai értékadással. Az aritmetikai értékadásnál a változó numerikus értéket vesz fel. Egy változó, kivéve a **boolean** változót perifériáról (klaviatúráról, lemez file-ból) való beolvasással is kaphat értéket.

Gyakorlat: Legyen a feladat két szám számtani és mértani közepének kiszámítása.
Az *a* és *b* számtani középértéke:

$$szk = \frac{a + b}{2}$$

Az *a* és *b* mértani középértéke:

$$mk = \sqrt{a \cdot b}$$

A feladat megoldásához meg kell határoznunk a változók típusát. Ha azt akarjuk, hogy az *a* és a *b* változó egész és valós számokat egyaránt tartalmazzon, akkor **real** típusként kell deklarálnunk. Az *szk* és *mk* változók az osztás és a gyökvonás miatt mindenképpen **real** típusúak lesznek. A program az adatokat a billentyűzetről olvassa be.

Futtassuk le az ERTEK1.PAS programot:

```
program ertekadas1;
var
     a,b,szk,mk: real;
begin
     write('1. szam: '); readln(a);
     write('2. szam: '); readln(b);
     szk:=(a+b)/2;
     mk:=sqrt(a*b);
     writeln('Szamtani kozepe: ',szk:6:1);
     writeln('Mertani kozepe : ',mk:6:1);
end.
```

A program futási képernyője:

```
1. szam: 9
2. szam: 4
Szamtani kozepe:     6.5
Mertani kozepe :     6.0
```

A program tervezésénél felhívjuk a figyelmet arra, hogy a változók típusának a meghatározásánál át kell gondolni, hogy milyen értékeket vehetnek fel a változók és a műveletek következtében az értékekhatárok hogyan alakulnak a program futása során. Hiszen a változó típusától függ, hogy a benne tárolt számérték milyen nagyságrendű lehet.

Például, ha a *v* változó típusa **integer**, akkor az alábbi programrészlet hibátlanul működik:

```
var
    v: integer;
begin
    v:= 500;
end.
```

De hibás az alábbi programrészlet:

```
var
    v: integer;
begin
    v:= 42589;
end.
```

mert az **integer** 2 byte-ban tárolja a számértéket és ezért a *v* változó csak -32768 és 32767 között vehet fel értéket.

Ha tudjuk, hogy a változó csak pozitív értéket fog tárolni és értéke nem lesz nagyobb a futás során 65535-nél, akkor helyesen járunk el a programrészlet módosításával:

```
var
    v: word;
begin
    v:= 42589;
end.
```

Ha azonban előfordulhat az, hogy negatív lesz a *v* tartalma és a szám nagysága meghaladja a 2 **byte** ábrázolási képességét, akkor módosítani kell a változat deklarációját:

```
var
        v: longint;
begin
        v:=-42589;
end.
```

Nézzük meg az alábbi programrészletet:

```
var
        v1,v2: word;
        ered1: longint;
begin
        v1:=42589;
        v2:=37242;
        eredm1:=v1+v2;
end.
```

Láthatjuk, hogy a *v1* és *v2* változók ugyan alkalmasak a számkonstansok tárolására, de az összegük már nem fér el a **word** típusú változóban, ezért az *eredm1* csak **longint** lehet.

Gyakorlat: Két változó tartalmának felcserélése

A programok írása során gyakran szükség van két változó értékének felcserélésére. Ennek leggyakoribb példája a számok nagyság szerinti sorba-rendezése, hiszen ez igazából a számok felcserélésének a sorozata. Legyen a két felcserélendő változó neve *x* és *y*. A felcserélő utasítás nincs a Pascal-ban, így csak értékadással tudjuk a cserét megvalósítani. Ha például a felcserélést az

```
        x:=y;
```

utasítással kezdjük, akkor mind a két változó értéke meg fog egyezni, mégpedig mindkét változó értéke *y* értéke lesz, az *x* eredeti értéke pedig elvész. Az előbbiek értelmében a

```
        y:=x;
```

utasítást már felesleges is kiadni, hiszen nem történik változás.
Nézzük meg ezt konkrét számokkal. Legyen *x* értéke 5, *y* értéke pedig 3. Az első utasítás hatására (`x:=y`) mind az *x*, mind az *y* értéke 3 lesz, ezután

az `y:=x;` utasítás az *y* értékét 3-ról 3-ra változtatja. Ez így tehát nem oldja meg a problémánkat. Gondoljuk végig, hogy hol a hiba!

A hiba az értékadás definíciójában rejlik, mivel a bal oldali változó felveszi a jobb oldalon álló konstans, változó vagy kifejezés értékét és a korábbi tartalma elvész.

A megoldás tehát az, hogy a bal oldalon lévő változó tartalmát a felülírás előtt ki kell menteni. Ehhez egy harmadik változóra, ún. munkarekeszre van szükség. Legyen a segéd változónk *id*. Az *id* változóba mentsük ki az *x* tartalmát, utána már felülírhatjuk az *y* tartalmával, majd ezután az *y* felveszi az *id* által megőrzött *x* értéket.

Nézzük meg a programban, hogyan változnak a változók tartalma. A változók értékét a program melletti táblázat mutatja olyan módon, hogy a táblázatban az adott sorban lévő utasítás végrehajtása utáni értékek vannak. A táblázatban lévő "-" jel jelenti, hogy a változónak még nincs értéke.

```
program csere;
var
    x,y,id : integer;                     x    y    id
begin
    x:=5;                                 5    -    -
    y:=3;                                 5    3    -
    writeln('x : ',x,'  y : ',y);
    id:=x;                                5    3    5
    x:=y;                                 3    3    5
    y:=id;                                3    5    5
    writeln('A csere utan  x: ',x,' y: ',y);
end.
```

A CSERE.PAS program eredménye:

```
x : 5  y : 3
A csere utan x: 3 y: 5
```

Ezzel a három utasítással fel tudtuk cserélni a két változó értékét, de ez segédváltozó használata nélkül nem sikerült volna.

8.1.2.2. Logikai értékadás

Boolean típusú változók *true* (igaz) vagy *false* (hamis) logikai értéket vehetnek fel. A logikai értékadásnál a bal oldalon **boolean** típusú változó áll és a jobb oldalon pedig **boolean** konstans, **boolean** változó vagy **boolean** kifejezés állhat, amely reláció is lehet, mert annak az eredménye szintén **boolean** értéket ad.

Mint már említettük, a **boolean** változók perifériáról, tehát a **read** és **readln** utasításokkal nem kaphatnak értéket.
Futtassuk le a LOG1.PAS programot:

```
program logikai_ertekadas;
var
                    i,j   : integer;
    b1,b2,b3,b4,b5,b6,b7,b8  : boolean;
begin
    i :=  2;
    j :=  5;
    b1:= true;
    b2:= false;
    b3:= i < j;
    b4:= b1 and b2;
    b5:= b2 or b1 and b3;
    b6:= b1;
    b7:= (i =3) or (j > 3);
    b8:= (b1 or b2) and (j+2 < i*3);
    writeln('b3: ',b3,' b4: ',b4,' b5: ',b5);
    writeln('b6: ',b6,' b7: ',b7,' b8: ',b8);
end.
```

A futási eredmény a következő:

```
b3: TRUE b4: FALSE b5: TRUE
b6: TRUE b7: TRUE b8: FALSE
```

Értékeljük ki az eredményt:

A *b3* értéke **true**, mert *2< 5* reláció igaz.
A *b4* értéke **false**, mert az **and** művelet bal oldalán lévő *b2* változó **false**.

A *b5* értéke **true**, mert a *b1* **and** *b3* igaz, mivel a jobb és a bal oldalon lévő logikai változó igaz és az **or** művelet jobb oldali eredménye igaz, így a végeredmény is igaz.
A *b6* értéke **true**, mert a *b1* tartalma **true** volt.
A *b7* értéke **true**, mert az **or** művelet jobb oldalán a j>3 reláció teljesült.
A *b8* értéke **false**, mert az **and** művelet bal oldalán a reláció értéke nem teljesült.

8.1.3. Goto utasítás

A **goto** utasítás feltétel nélküli vezérlésátadást valósít meg. Gyakorlatilag a Pascal-ban szinte mindig el lehet kerülni a **goto** utasítás használatát. A gyakorlott Pascal programozók nem is használják. Akkor szép egy Pascal program felépítése, ha nem tartalmaz **goto** utasítást, amely a programot áttekinthetetlenné teszi.
A **goto** hatásköre az adott blokkon belül érvényes, blokkba beugrani nem szabad, de blokkból kiugrani lehet. Nem lehet kiugrani függvényből vagy eljárásból.

```
program cimke;
label ide, ki;
begin  { A blokk }
     ...
     begin { B blokk }
          ...
          goto ide;
          ...
ide:      ...
     goto ki;
      ...
     end;
ki:     ...
end.
```

8.1.4. Eljáráshívás

Az eljárást a nevével és az aktuális paramétereivel hívjuk. A 9. fejezet foglalkozik részletesen az eljárások készítésével és aktiválásával.
Például az *olvas* eljárás a háromszög oldalainak (*a*,*b*,*c*) ad értéket, a *kerulet* eljárás pedig az adott oldalakból (*a*,*b*,*c*) kiszámítja a háromszög kerületét (*ker*).
Az eljárások hívása a következő:

```
olvas(a,b,c);
kerulet(a,b,c,ker);
```

8.2. Struktúrtált utasítások

8.2.1. A blokk utasítás

A **begin** és **end** kulcsszavak közötti utasítások sorozatát összetett utasításnak, blokknak nevezzük. Azon Pascal utasítások esetén, amelyek csak egy utasítást tudnak végrehajtani, fontos az egynél több utasítás összefogása egyetlen utasítássá a **begin** és **end** használatával.

8.2.2. Feltételes utasítások

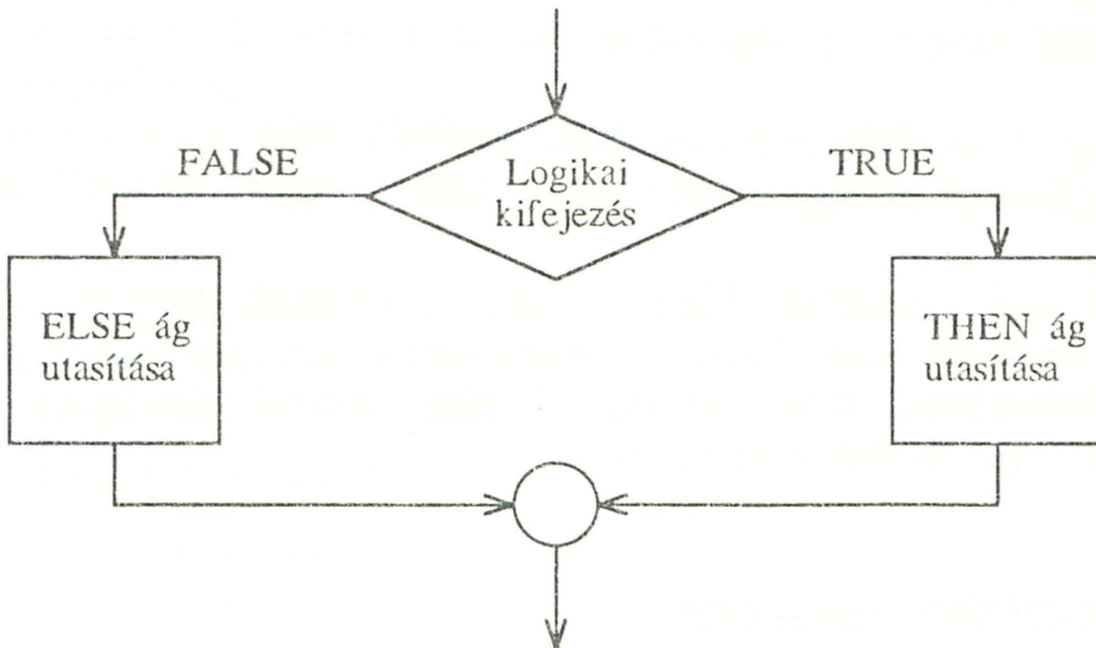
A feltételes utasítások segítségével a programokban lehetőség van bizonyos feltételek teljesülése esetén egy utasítás vagy akár egy programrészlet végrehajtására. A Pascal-ban ilyenfajta műveletek elvégzésére szolgál az **if** utasítás.

8.2.2.1. If utasítás

Az **if** utasítás kétirányú, feltételes elágazást hajt végre. Az **if** utasítás általános formája:

if *logikai kifejezés* **then** *utasítás1* **else** *utasítás2;*

Az **if** után *logikai változó* vagy *logikai kifejezés* állhat, amelynek az teljesülésétől függ, hogy melyik utasítás kerül végrehajtásra. Ha a feltétel igaz (*true*), akkor a **then** utáni *utasítás1* fog végrehajtódni, ha a feltétel hamis (*false*), akkor az **else** utáni *utasítás2* kerül végrehajtásra. Az **if** szerkezetet ; zárja, az **else** elé viszont *nem szabad* pontosvesszőt tenni.



8.2. ábra Az **if then ... else** utasítás

Ha akár a **then** vagy akár az **else** után egynél több utasítást akarunk elhelyezni, akkor az utasításokat utasítás zárójelben, tehát **begin** és **end** közé kell tenni. Így a feltételes utasítás a **then** ill. az **else** után egyetlen blokként fogja azokat végrehajtani.

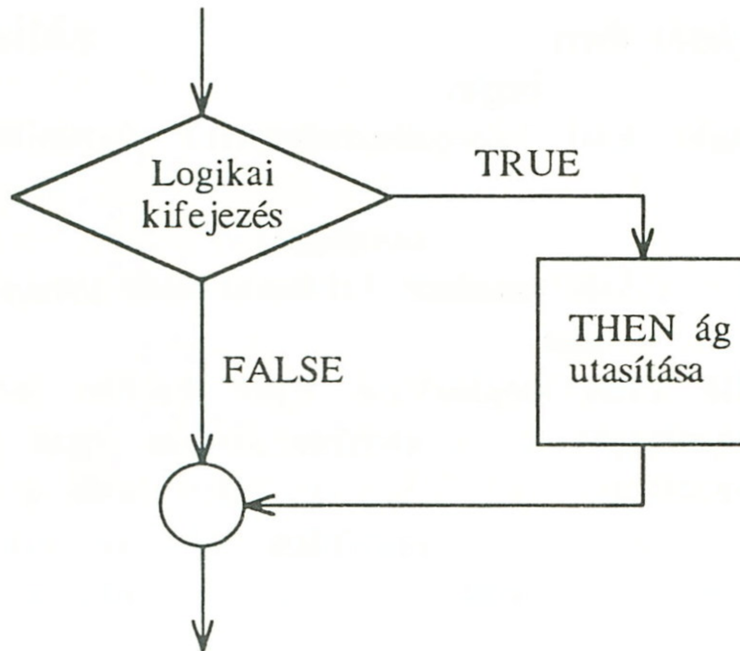
```
if logikai kifejezés then
              begin
                  utasítás1;
                  ...
                  utasításn;
              end
          else
              begin
                  utasítás1;
                  ...
                  utasításn;
              end;
```

A feltételes utasítás egyszerűbb formája az, amikor a feltételtől függően egyszerűen csak végre kívánjuk hajtani vagy át akarjuk ugrani a megadott utasítást, vagy utasítás blokkot. Ez esetben az **if** formája:

```
if logikai kifejezés then utasítás1;
kovetkezo utasítás;
```

vagy

```
if logikai kifejezés then
              begin
                  utasítás1;
                  ...
                  utasításn;
              end;
következő utasítás;
```



8.3. ábra **If then** utasítás

Nézzük meg az If utasítás használatát, futtassuk le a FELT1.PAS programot különböző adatokkal.

```
program feltetel1;
var
    x,y : integer;
begin
    write('x : '); readln(x);
    write('y : '); readln(y);
    if (x < y) then
                writeln('x : ',x,' a kisebb ertek')
            else
                writeln('y : ',y,' a kisebb ertek');
    if (x < y) and (y <>10) then
                        begin
                          s:= x + y;
                          writeln(' x+ y : ',s);
                        end;
    if (x > y) or (x-y < 0) then
                begin
                 inc(x); dec(y);
                 writeln('x novelt erteke: 'x);
                 writeln('y csokkentett erteke  : ',y);
               end;
    if (x mod y = 0) then
```

```
                           writeln(x,' oszthato 'y)
                         else
                           writeln(x,' nem oszthato ',y);
    end.
```

A program eredményei
Első futtatás:

```
    x : 4
    y : 8
    x : 4 a kisebb ertek
     x+y : 12
    x novelt erteke      : 5
    y csokkentett erteke: 7
    5 nem oszthato 7
```

Második futtatás:

```
    x : 7
    y : 3
    y : 3 a kisebb ertek
    x novelt erteke      : 8
    y csokkentett erteke: 2
    8 oszthato 2
```

Az **if** utasításokat egymásba is lehet ágyazni, ezt mutatja be a FELT2.PAS program:

```
    program feltetel2;
    var
         t : real;
    begin
         writeln('A viz halmazallapotanak vizsgalata');
         write('homerseklet : '); readln(t);
         if (t > 0) then
                    begin
                      if (t >= 100) then writeln('goz ')
                                   else writeln('viz ');
                    end
                   else writeln('jeg');
    end.
```

A program futtatási eredményei:

```
A viz halmazallapotanak vizsgalata
homerseklet : -1
jeg

A viz halmazallapotanak vizsgalata
homerseklet : 12
viz
```

Az **if .. then .. else .. if** utasítás használatára nézzük meg a FELT3.PAS programot:

```
program feltetel3;
var
    pont: 0 .. 100;
begin
    writeln('0 es 100 kozotti pontok osztalyzata');
    write('Az elert potszam: '); readln(pont);
    write('A dolgozat osztalyzata: ');
    if pont= 0 then writeln('Ervenytelen')
                else
                if pont<50 then writeln('elegtelen')
                else
                if pont<60 then writeln('elegseges')
                else
                if pont<70 then writeln('kozepes')
                else
                if pont<89 then writeln('jo')
                else writeln('jeles');
end.
```

A program futtatási eredményei:

```
0 es 100 kozotti pontok osztalyzata
Az elert pontszam: 92
A dolgozat osztalyzata: jeles

0 es 100 kozotti pontok osztalyzata
Az elert pontszam: 88
A dolgozat osztalyzata: jó
```

8.2.2.2. Case utasítás

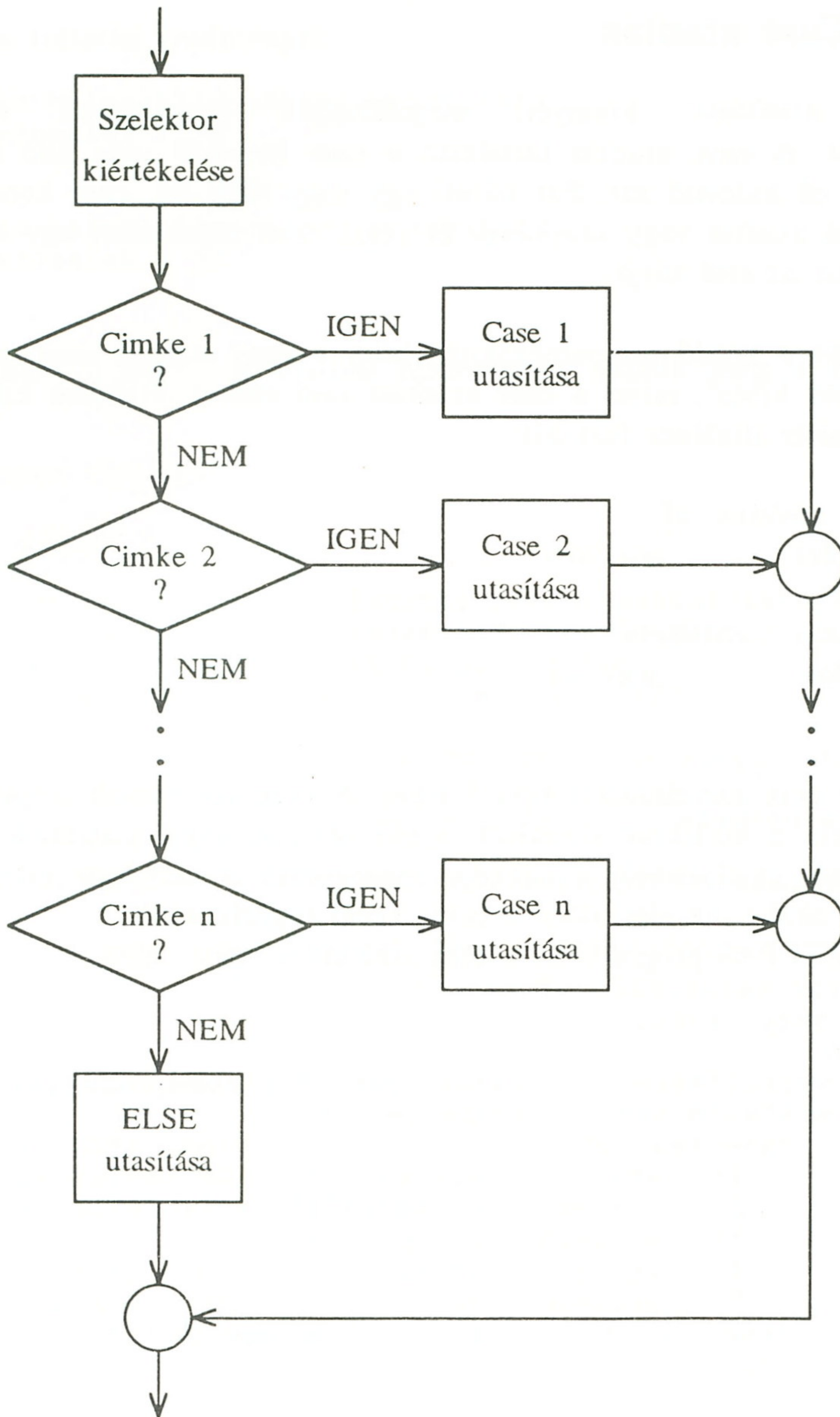
A **case** utasítással könnyen megoldhatjuk programunk többirányú elágaztatását. A **case** utasítás tartalmaz a **case** kulcsszó után álló szelektort, amelyet az **of** kulcsszó zár. Ezt követi egy vagy több ún. **case** konstans és a hozzátartozó utasítás vagy utasítások és végül nem kötelezően egy **else** ág. A **case** utasítást az **end** zárja.

Gyakori hiba a kezdő programozóknál, hogy a **case** utasítás miatt a program gyakran "**end** hibás", mivel a **case** utasítást záró **end**-et általában kifelejtik.
A **case** utasítás általános formája:

> **case** *szelektor* **of**
> *cimke1:* *utasítás1;*
> ...
> *cimken:* *utasításn;*
> **else:** *utsításm;*
> **end**;

A *szelektor* csak sorszámozott típusú lehet. A **case** constansok típusának meg kell egyeznie a szelektor típusával. Azon utasítás vagy utasítások kerülnek végrehajtásra, amelyeknek a cimkéje megegyezik a szelektor értékével, ha nincs ilyen címke, az **else** utáni utasítás kerül végrehajtásra.
A VALASZT1.PAS programban a **case** szelektora egész szám.

```
program valaszt1;
var  jegy: integer;
begin
     write('Kerem az erdemjegyet: '); readln(jegy);
     write('A vizsga eredmenye: ');
       case jegy of
          1:  writeln(' elegtelen');
          2:  writeln(' elgseges')'
          3:  writeln(' kozepes');
          4:  writeln(' jo');
          5:  writeln(' jeles');
       else  writeln(' Hibas erdemjegy);
      end;
end.
```



8.4. ábra A **case** utasítás folyamatábrája

A VALASZT2.PAS programban a **case** szelektora karakter.

```
program valaszt2;
var
    kod  : char;
    aru  : string;
begin
    writeln('Az aru minositese kituno(k), ',
    'jo(j),megfelelo(m)');
    write('Az aru megnevezese: '); readln(aru);
    write('A minosites kodja : '); readln(kod);
    writeln;
    write(string,' minositese: ');
    case kod of
       'K','k':   writeln('kituno');
       'J','j' :  writeln('jo');
       'M','m':   writeln('megfelelt');
    else
        writelen('hibas adat');
    end;
end;
```

A VALASZT3.PAS programban a **case** szelektora résztartomány típusú.

```
program valaszt3;
var
    pont: 0 .. 100;
begin
    writeln('0 - 100 kozotti pontok osztalyzata');
    write('Az elert pontszam: '); readln(pont);
    write('A dolgozat osztalyzata: ');
    case pont of
       90..100 : writeln(' jeles');
       70.. 89 : writeln(' jo');
       60.. 69 : writeln(' Kozepes');
       50.. 59 : writeln(' elegseges');
       0..  49 : writeln(' elegtelen');
      else     writeln(' Hibas adat!');
    end;
end.
```

8.2.3. Ciklusutasítások

A programban gyakran van szükség utasítások ismételt végrehajtására. Az ilyen feladatok megoldására szolgálnak a ciklusutasítások. A Pascalban három ciklusutasítás használható. Ha ismerjük az ismétlések számát, akkor a **for** utasítást használjuk, máskülönben valamilyen feltétel vezéli a ciklust a **while** és a **repeat until** utasításokban.

8.2.3.1. For utasítás

A **for** ciklusutasítást akkor használhatjuk, amikor pontosan ismerjük az ismétlések darabszámát, így a ciklusmagot egy meghatározott számszor végrehajtjuk. A **for** ciklust kétféleképpen lehet használni, az ún. növekvő ciklus esetén a ciklus változója adott kezdőértéktől egyesével növeszik a **to** kulcsszó használatakor és csökkenő ciklus esetén a ciklus változója egyesével csökken **downto** kulcsszó esetén az adott végértékig és mindannyiszor végrehajtja a ciklusmagot. A **for** ciklusutasítás csak egyetlen utasítás végrehajtására vontakozik. Példaként a **for** utasítás alakja növekvő ciklus esetén :

for *ciklusváltozó:=kezőérték* **to** *végérték* **do**
ciklusmag (egy utasítás);

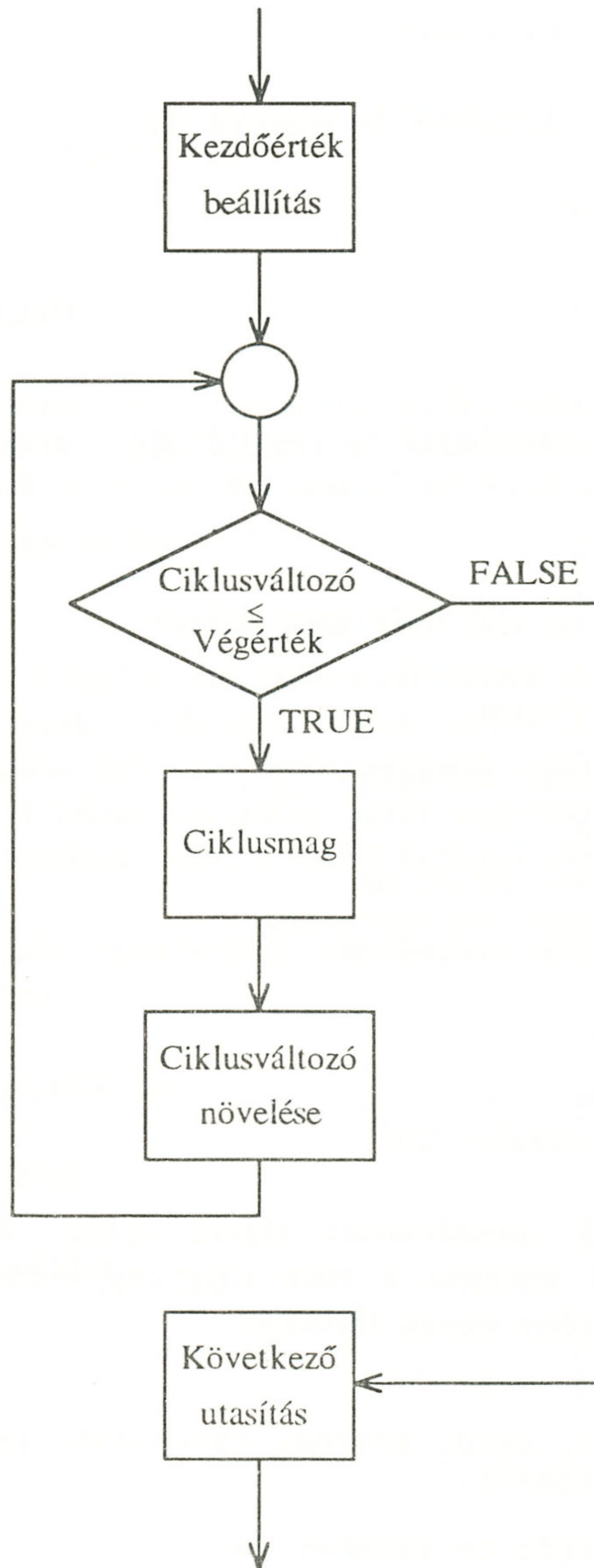
A **for** utasítás alakja csökkenő ciklus esetén :

for *ciklusváltozó:=kezőérték* **downto** *végérték* **do**
ciklusmag (egy utasítás);

A növekvő cilklus esetén
kezdőérték < *végérték*

a csökkenő ciklus esetén
kezdőérték > *végérték*

feltétel esetén indul el a ciklusmag végrehajtása, különben a ciklusmag egyszer sem lesz végrehajtva.
Gyakori hiba a ciklus végértéke után a **do** kulcsszó megadásának elfelejtése.



8.5. ábra A **for** utasítás folyamatábrája.

Ha egynél több utasítást akarunk ciklusban végrehajtani, akkor **begin end** utasítás zárójelet kell alkalmazni:

> **for** *ciklusváltozó:=kezőérték* **to** *végérték* **do**
> **begin**
> *utasítás1;*
> ...
> *utasításn;*
> **end**;

A **for** *ciklusváltozó:=kezőérték* **to** *végérték* **do** ; utasítás; tipikus esete az üres utasításnak, mert a ciklus üresen fut le, és a **do** utáni utasítás csak egyszer hajtódik végre.

Példaként adjuk össze az első húsz egész számot.

```
program for_ciklus;
var
    szam,osszeg: integer;
begin
    osszeg:=0;
    for szam:=1 to 20 do
        osszeg:=osszeg+szam;
    writeln('Az osszeg for ciklussal: ',osszeg);
end.
```

A futtatás eredménye:

```
Az osszeg for ciklussal: 210
```

A ciklusváltozó csak sorszámozott típusú lehet. Általános esetben a ciklusváltozó növekvő léptetése a *succ* függvény szerint, míg a csökkenő léptetése a ***pred*** függvény szerint történik.

```
var
    a: (hetfo, kedd, szerda, csutortok, pentek, szombat,
       vasarnap);
begin
    for a:=hetfo to szombat do
      ...
```

```
type
    betu='a'.. 'z';
var
    i: betu;
begin
    for i:='a' to 'z' do
     ...
```

8.2.3.2. While utasítás

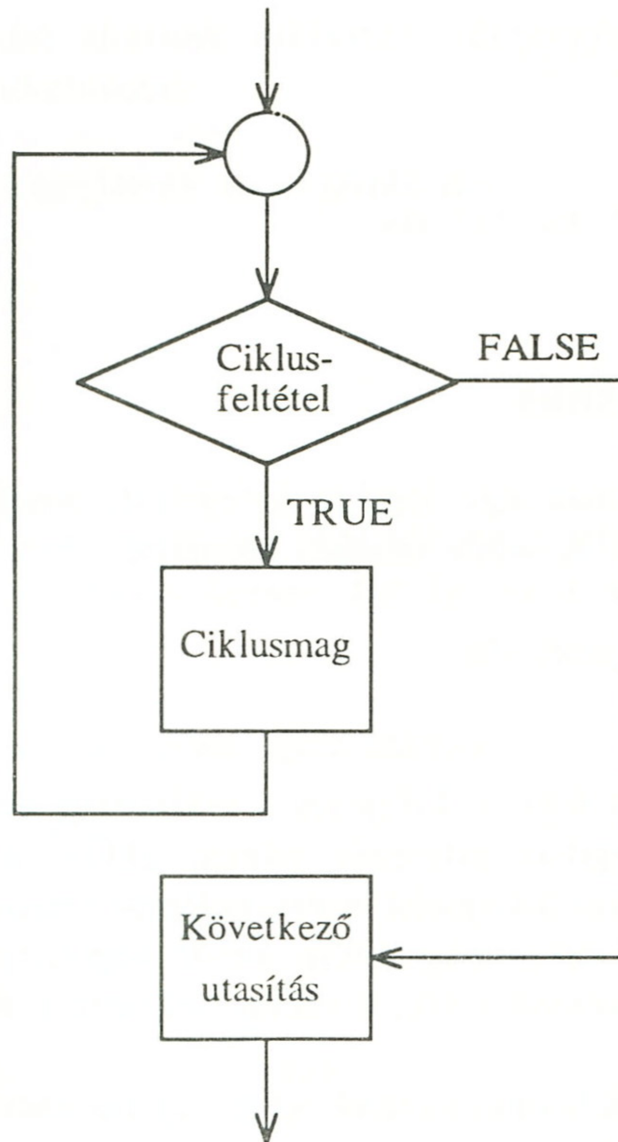
A **while** utasítás tartalmaz egy logikai kifejezést, amely vezérli az ismételt utasítások végrehajtását. A **while** utasítás formája:

> **while** *logikai kifejezés* **do**
> *utasítás1*;

Ennél a ciklusfajtánál a logikai kifejezés a ciklusmag végrehajtása előtt kerül kiértékelésre. Ha a logikai kifejezés hamis, akkor a ciklus egyszer sem hajtódik végre. A logikai kifejezést ezért belépési feltételnek is hívjuk. Az egyetlen utsításból álló ciklusmag addig kerül végrehajtásra, amíg a logikai kifejezés igaz, amint hamissá válik, a ciklus befejezi a működését.

Egynél több utasítás ciklusban történő végrehajtása esetén **begin-end** utasítás zárójelet kell alkalmazni

> **while** *logikai kifejezés* **do**
> **begin**
> *utasítás1;*
> ...
> *utasításn*
> **end**;



8.6. ábra A **while** ciklus folyamatábrája

Adjukösssze az első húsz egész számot.

```
program while_ciklus;
     var
          szam, osszeg: integer;
     begin
          szam:=1;
          osszeg:=0;
          while szam < 21 do
            begin
               osszeg:=osszeg+ szam;
               szam:=szam+1;
            end.
     writeln('Az osszeg while ciklussal:", osszeg);
     end.
```

A futtatás eredménye:
```
Az osszeg while ciklussal: 210
```

8.2.3.3. Repeat utasítás

A **repeat** utasítás tartalmaz egy logikai kifejezést, amely a ciklus végrehajtását vezérli. A **repeat-until** között megadott utasítások ismételten hajtódnak végre. Egyszer mindeképpen végrehajtódik a ciklus, mivel a logikai kifejezés csak az egyszeri végrehajtás után kerül kiértékelésre. A ciklusmag addig hajtódik végre, amíg a logikai kifejezés hamis, és akkor fejeződik be, amikor a logikai kifejezés igazzá válik. Ebben az esetben a logikai kifejezést a kilépési feltételnek hívjuk.
A **repeat** utasítás általános alakja:

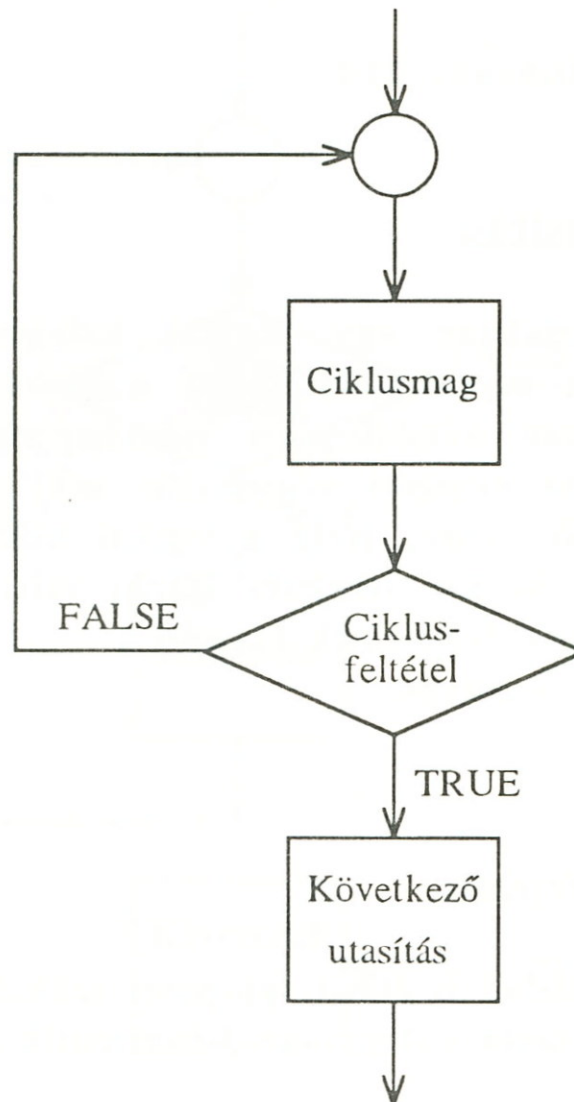
repeat
ciklusmag;
until *logikai kifejezés;*

A ciklusmag több utasításból is állhat, itt nincs szükség a **begin-end** utasítás zárójelre, mert a **repeat-until** kulcsszavak helyettesítik azt.

Adjukösssze az első húsz egész számot.

```
program repeat_ciklus;
var
     szam, osszeg: integer;
begin
     szam:=1; osszeg:=0;
     repeat
          osszeg:=osszeg+ szam;
          szam:=szam+1;
     until szam szam = 21;
     writeln('Az osszeg repeat ciklussal:", osszeg);
end.
```

A futtatás eredménye:
```
Az osszeg repeat ciklussal: 210
```



8.7. ábra A **repeat-until** ciklus folyamatábrája

8.2.4. With utasítás

A **with** utasítással egyszerűbben adhatunk rekordnak értéket, mivel a rekord mezeire, mint egyedülálló változókra hivatkozhatunk. A **with** utasítás általános formája:

with *rekord_változó* **do** *utasítás*;

Az *utasítás* tetszőleges egyszerű vagy struktúrált utasítás lehet.

Annyival rövidebb egy rekord megadási módja, hogy a **with** után megadott *rekord_változó* nevét és a pontot a **do** után nem kell újra kiírni, hanem elegendő csak a mezőneveit megadni.

Az WITH_PR.PAS program bemutatja a **with** használatát:

```
program with_pr;
type
    datum = record
              ev    : integer;
              honap : 1..12;
              nap   : 1..31;
              megjegyzes: string;
           end;
var
     szul_nap, mai_nap, holnap: datum;
begin
     with szul_nap do
     begin
         ev    := 1974;
         honap := 3;
         nap   := 13;
         megjegyzes := 'szuletesnap';
    end; { with }
    writeln('Szuletesi datum: ',szul_nap.ev,' ',
             szul_nap.honap,' ',szul_nap.nap);
    writeln('Mai nap datuma ');
     write('Ev    : '); readln(mai_nap.ev);
    write('Honap( 1..12): '); readln(mai_nap.honap);
    write('nap (1..31): '); readln(mai_nap.nap);
    write('megjegyzes: '); readln(mai_nap.megjegyzes);

    holnap.ev := mai_nap.ev;
    holnap.honap:= mai_nap.honap;
    holnap.nap:= 1;
    write('Holnapi datum: ',holnap.ev,' ',
           holnap.honap,' ',holnap.nap);
end.
```

Két azonos típusú rekord közötti értékadás lehet mezőnként:

```
holnap.ev := mai_nap.ev;
holnap.honap := mai_nap.honap;
```

Lehet egyszerre az összes mezőnek adni értéket:

```
holnap := mai_nap;
```

Bonyolultabb, úgynevezett *variálható* rekordokat is létrehozhatunk a **case** utasítás beépítésével.

with kulcsszó után több rekordváltozót is felsorolhatunk, mindaddig, amíg ez nem vezet a kétértelműségre.
A VARI_REK.PAS a variálható rekordra mutat példát:

```
program varians_rekord;
type
    sikidom=(teglalap, negyzet, kor);
    alakzat = record
                x,y : real;
                case fajta: sikidom of
                  teglalap: (szelesseg, magassag : real);
                  negyzet:  (oldal : real);
                  kor:      (sugar: real);
               end;

var
   g: alakzat;
   tipus: sikidom;
   sz,m,l,r,ter: real;

begin
   writeln('Az alakzat tipusok ');
   for tipus:=teglalap to kor do
    begin
     g.fajta:=tipus;
     with g do
      case fajta of
           teglalap : begin
                        writeln;
                        writeln('TEGLALAP');
                        write('A teglalap szelessege: ');
                        readln(sz);
                        write('A teglalap magassaga : ');
                        readln(m);
                        szelesseg:=sz; magassag:=m;
                      end;

           negyzet  : begin
                        writeln;
```

```
                        writeln('NEGYZET');
                        write('A negyzet oldala  : ');
                        readln(l);
                        oldal:=l;
                      end;
        kor         : begin
                        writeln;
                        writeln('KOR');
                        write('A kor sugara  : '); readln(r);
                        sugar:=r;
                      end;
        end;
    with g do
      case fajta of
        teglalap: begin
                    ter:=szelesseg*magassag;
                    writeln('A teglalap terulete : ',ter:10);
                  end;
        negyzet: begin
                    ter:=oldal*oldal;
                    writeln('A negyzet terulete: ',ter:10);
                  end;
        kor:     begin
                    ter:=r*r*pi;
                    writeln('A kor terulete: ', ter:10);
                 end;
      end;
    end;
end.
```

A program futási eredménye:

```
Az alakzat tipusa:

TEGLALAP
A teglalap szelessege: 3
A teglalap magassaga : 4
A teglalap terulete  : 1.200E+01

NEGYZET
A negyzet oldala  : 5
A negyzet terulete: 2.500E+01

KOR
A kor sugara  : 6
A kor terulete: 1.131E+02
```

Ellenőrző kérdések:

1. Mit nevezünk összetett utasításnak?
2. Miért szükséges egynél több utasítás össszefogás a **begin** **end** utasításokkal?
3. Mire szolgálnak a feltételes utasítások?
4. Hány fajta feltételes utasítás van?
5. Mi az **if** utasítás általános alakja?
6. Az **if** utasításban hová nem szabad pontosvesszőt tenni?
7. Ha a **then** vagy az **else** ágban egy utasításnál több utasítást szeretnénk végrehajtani, mit kell tenni?
8. Melyik utasítással oldhatjuk meg a többirányú elágaztatást?
9. Mit jelent a **case** utasításban az **else** kulcsszó használata?
10. Utasítások ismételt végrehajtását mivel oldhatjuk meg?
11. Mikor alkalmazhatjuk a **for** utasítást?
12. Milyennek kell lenni a feltételnek a **while** típusú ciklusnál?
13. A feltételnek mire kell változnia ahhoz, hogy a **while** a működését befejezze?
14. A **repeat** ciklusnál hol történik a feltétel vizsgálata?
15. Milyennek kell lennie a feltételnek, hogy a **repeat** ciklus működjön?
16. A feltételnek mire kell változnia ahhoz, hogy a **repeat** ciklus a működését befejezze?

Feladatok

1. Olvassa be a téglatest adatait és számítsa ki a téglatest felszínét és térfogatát. (TEGLALAP.PAS)

2. Számítsa ki az első n négyzetszám összegét. Az n értéke konstansként legyen 100. (NEGYZET.PAS)

3. n darab egész számot egy tömbbe olvasson be és vizsgálja meg hány eleme negatív, nulla és pozitív. (CIKLUS6.PAS)

4. n darab egész számot egy tömbbe olvasson be, számolja meg hány páros és páratlan eleme van a tömbnek és hány eleme osztható hárommal. (CIKLUS7.PAS)

5. Max. 30 egész adat közül számolja meg mennyi eleme esik 1..10, 11..100, 101..1000 közé és >1000 fölé. (CASE1.PAS)

6. Max. 30 tanuló nevének és megtakarított pénzének nyilvántartására írjon programot, az adatokat táblázat formájában írja vissza, valamint számítsa ki osztálypénz összegét. (CIKLUS51.PAS)

7. Készítsen egy egyszerű kalkulátor programot, amely az adatokat művelet szám1 szám2 alakban várja. A programból e betűvel lehessen kilépni. (CASEPLD.PAS)

8. Olvasson be max. 4 jegyű számot és írja ki fordítva. (FORDIT.PAS)

9. Irjon egy programot, amely 0 és 100 között generál egy véletlenszámot. A program párbeszédes formában kérdezzen a számra, és segítségül jelezze, hogy a kapott szám kevés vagy sok a kitalálandó számhoz képest. A lépéseket számlálja, amelyet a szám kitalálásakor írja vissza. (JATEK.PAS)

10. n darab kockadobásból számolja meg, mennyi 1,2,3,4,5 és 6 volt a dobás értéke. (KOCKA.PAS)

11. mm-ben megadott távolságot számítson át m, cm és mm-re. (MM_M.PAS)

12. Másodpercben megadott időt számítson át óra, perc, másodpercre. (ORA.PAS)

13. Írjon egy öröknaptár programot, amely egy dátumról megmondja, hogy melyik napra esik. (OROKNAP.PAS)

14. A 'Ma szep ido van' szövegben a üres helyet * karakterrel helyettesítse. (POSPR.PAS)

15. Irjon programot, amely egy dátumról megmondja, hogy szökőév-e. (SZOKO.PAS)

16. 1-től kétezerig írja fel a római számokat. (ROMAI.PAS)

17. Irjunk programot, amely beolvas egy mondatot.
Például: Ma szep ido van.
Visszaírja: Mava szevep ividovo vavan.
Tehát a magánhangzó esetében v betűvel megismétli a magánhangzót:
i -> ivi, o -> ovo, e-> eve, a-> ava, u-> uvu
(VAVEVI.PAS)

18. Irjon programot, amely az alábbi elrendezésű számhalmazt jeleníti meg. Maximálisan 19 sorban. A példa 4 sorra vonatkozik. (KUPASC1.PAS)

```
         1
      2     2
   3     3     3
4     4     4     4
```

19. Irjon programot, amely az alábbi elrendezésű számhalmazt jeleníti meg. Maximálisan 20 sorban. A példa 4 sorra vonatkozik. (KUPASC2.PAS)

```
1 2 3 4
1 2 3
1 2
1
```

20. Irjon ötös lottó kitöltéséhez generáló programot. A lottószámokat bármely billentyű lenyomására jelenítse meg a program, kilépés az ESC billentyűvel történjen. (LOTTO.PAS)

21. Személyi adatok tárolására készítsen rekordot. Irjon programot, amely ilyen típusú elemekből álló tömböt feltölt, illetve adott elemét listázza. (REKORD2.PAS).

22. Értékelje ki a REKORD1.PAS programot, amely adott szempontok alapján válogatást végez a rekordtömb elemei között.

9. ELJÁRÁSOK ÉS FÜGGVÉNYEK

A mintafeladatokban láttuk, hogy a Pascal program a végrehajtható utasításokat a program törzsében tartalmazza.

Mielőtt rátérnénk az eljárások és függvények tárgyalására oldjunk meg egy egyszerű feladatot, amelyen bemutatjuk az eljárások és a függvények készítését.

Legyen a feladat a háromszög kerületének és területének kiszámítása, ha ismert a háromszög három oldala.

A program a következő részfeladatok végrehajtását tartalmazza:

- olvassa be a háromszög három oldalának adatait,
  A beolvasó programrész gondoskodjon arról, hogy ne fogadjon el olyan adatot, amellyel nem lehet háromszöget szerkeszteni. A háromszög megszerkeszthetőségének feltétele: a háromszög bármely két oldalának összege nagyobb kell legyen, mint a harmadik oldal. Ha a feltétel nem teljesül, akkor azt a program jelezze és újra kérje az adatokat.
- számítsa ki a háromszög kerületét,
- számítsa ki a háromszög területét,
- írja ki az eredményeket.

A program megírásához szükséges algoritmus a következő:
1. Egy a,b,c oldalú háromszög kerülete

k= a+b+c

2. Ha ismerjük a háromszög három oldalát, akkor a háromszög területetét legegyszerűbben a Heron képlettel határozhatjuk meg:

Definiáljuk a félkerületet:

$$s = \frac{a + b + c}{2}$$

akkor a háromszög területe a Heron képlettel:

$$t = \sqrt{s(s-a)(s-b)(s-c)}$$

A következő lépés, megtervezni a program szükséges változóit.

a_oldal	A oldal,
b_oldal	B oldal,
c_oldal	C oldal adatainak tárolásárára
h_kerulet	kerület,
h_terulet	terület eredményének tárolására
s	a fél kerület tárolására
f	az adatellenőrzéshez szükséges logikai változó számára.

Mivel egy a háromszög oldalainak hossza nem csak egész szám lehet, így a változókat **real** típussal deklaráljuk.

A HAROMSZ.PAS program listája:

```
program haromszog;
(* Globális deklaráció *)
var
   a_oldal,b_oldal,c_oldal,
   h_terulet, h_kerulet     : real;
                           f: boolean;
                           s: real;
begin
(* A program törzse *)

(* Az adatok beolvasása *)
       writeln('A haromszog oldalainak megadasa');
       repeat
          write('A oldal: '); readln(a_oldal);
          write('B oldal: '); readln(b_oldal);
          write('C oldal: '); readln(c_oldal);

(* Szerkeszthetõség ellenõrzése *)
          if(a_oldal+b_oldal>c_oldal) and
            (a_oldal+c_oldal>b_oldal) and
            (b_oldal+c_oldal>a_oldal)
                  then f:=true
```

```
                    else
                     begin
                        f:=false;
                        writeln('Hibas adatok');
                    end;
      until f;
(* A háromszög kerületének számítása *)
      h_kerulet:=a_oldal+b_oldal+c_oldal;

(* A félkerület számitása *)
      s:=(a_oldal+b_oldal+c_oldal)/2;

(* A háromszög területének számítása *)
      h_terulet:=sqrt(s*(s-a_oldal)*(s-b_oldal)*
                     (s-c_oldal));

(* Elsõ eredmény kiiratása *)
      writeln;
      writeln('A haromszog adatai');
      writeln('A oldal :',a_oldal:10:2);
      writeln('B oldal :',b_oldal:10:2);
      writeln('C oldal :',c_oldal:10:2);
      writeln;
      writeln('A haromszog kerulete : ',h_kerulet:10:2);

(* Második eredmény kiiratása *)
      writeln;
      writeln('A haromszog adatai');
      writeln('A oldal :',a_oldal:10:2);
      writeln('B oldal :',b_oldal:10:2);
      writeln('C oldal :',c_oldal:10:2);
      writeln;
      writeln('A haromszog terulete : ',h_terulet:10:2);
end.
```

Sokszor szükséges, hogy adott műveleteket többször végrehajtsunk a program különböző helyein. Például ugyanazon adatokat többször szükséges beolvasni, eredményeket kiírni (lásd a fenti mintaprogramot), részfeladatokat leíró utasítássorozatokat többször kell végrehajtani stb. Az utasításcsoportok többszöri felesleges ismétlését helyettesíti az alprogram.

Az alprogram egyszerűen az utasításoknak egy blokkja, amely azért készül, hogy a programozó kivánalmainak megfelelően többször legyen végrehajtva.

Az alprogramnak szintén van neve és a főprogramon vagy más alprogramon belül lehet hívni, hogy végrehajtsa az utasításait.

A Pascal program szerkezete áttekinthetőbb, rövidebb és struktúráltabb, ha a logikailag összetartozó utasításokat alprogramokba foglaljuk. Az Pascal nyelvben a **procedure** kulcsszó, amely az alprogram fejét vezeti be, hasonlóan a **program** kulcsszóhoz, eljárásnak nevezzük.

Már megismerkedtünk a szabványos függvényekkel, mint például az ***sqr*** a négyzetre emelés, az ***sqrt*** a gyökvonás, stb. Lehetőség van saját tervezésű függvényt készítésére is. Erre szolgál a **function** kulcsszó, amellyel a szabványos függvényekhez hasonlóan felhasználható függvényeket készíthetünk.

Másik nagy előnye a függvényeknek és az eljárásoknak, hogy a programban többször, különféle paraméterekkel aktiválhatjuk, ezáltal sok munkát takarítunk meg, mivel nem kell ugyanazokat a programrészeket feleslegesen, kis változtatással többször leírni.

A Pascal-ban tehát kétfajta alprogram van: az eljárás és a függvény. A két alprogram közötti fő különbség a hívás módjában jelentkezik. Az alprogramokat a program deklarációs részében kell megadni (lásd a 4. fejezetet).

Az alprogram utasításainak végrehajtásához szükséges bemenő adatokat, valamint azt, hogy az eredmény hol keletkezzen, pontosan meg kell adni. Ezeket az információkat az ún. paraméterlistán adjuk meg. A paraméterlistán keresztül küldünk és kapunk adatokat, mivel a paramétereken keresztül bonyolódik le az adatcsere.

Nagyon fontos, hogy megértsük a paraméterek megadási módját, hiszen a paraméterek biztosítják az adatok cseréjének lehetőségét az alprogram (eljárás, függvény) és az őt hívó külső program között.

Az alprogram definíciójában a paramétereket *formális*, az alprogramot hívó utasításban szereplő paramétereket *aktuális* paramétereknek nevezzük. Fontos szabály, hogy a formális és az aktuális paramétereknek darabszámban, típusban és a megadási sorrendben meg kell egyezniük.

A paraméterek átadásának kétféle módja van:

- érték (értékparaméter) és
- cím szerint (változóparaméter).

Érték szerinti paraméterek

Az érték szerinti paraméterátadásnál a formális paraméterek számára a memóriában a Pascal mindig helyet foglal. Ezt talán úgy lehet könnyen megmagyarázni, hogy amikor hívjuk az eljárást, akkor az érték szerint átadott aktuális paraméterek értéke átmásolódik a megfelelő formális paraméterbe.
Az alprogram ezekkel az átmásolt értékekkel számol, mintegy védelmet biztosít a felhasználónak arra nézve, hogy az ilyen módon megadott aktuális paramétereinek tartalma ne íródjon felül.

Ha mégis ezek közül a paraméterek közül az alprogram törzsében felülírás történik, tehát a paraméter az értékadó utasítás bal oldalán is szerepel, akkor ez a felülírás nem rontja el az aktuális paraméter tartalmát, hiszen az alprogram által más címen lefoglalt formális paraméterben történt a változás.

Egyébként szokás az érték szerinti paramétereket az alprogram bemenő paramétereinek is nevezni, hiszen ezekben adjuk meg azokat az értékeket, amelyekkel az alprogramnak számolnia kell, tehát az alprogram törzsében lévő utasítások ezeknek az adatoknak az ismeretében új értékeket állítanak elő, végeznek műveletet.

Az értékparaméterek helyén a híváskor megadhatunk konstanst, változót, ill. kifejezést is. Ha kifejezést adtunk meg, akkor az először kiértékelődik, majd annak az értéke kerül a formális paraméterbe.

Az alprogram a megadott adatokból új értékeket képezhet, ezek fogják alkotni az alprogram "eredményeit". Az ilyen értékeket nevezzük az alprogram kimenő paramétereinek.

A paraméterlista megadásánál az alábbi szabályokat kell figyelembe venni:

- a paramétereket kerek zárójelbe kell tenni,
- az azonos típusú paramétereket felsorolva vesszővel lehet elválasztani,
- az utolsó paraméter után kettőspontot téve a paraméterek típusát kell megadni. A típus csak egyszerű típus, vagy felhasználói típus (Type) lehet.
- ha különböző típusú paraméterek szerepelnek a listán, akkor a különböző típusú paraméterek közé pontosvesszőt kell tenni.

Legyen a példában az alábbi ötféle típusú értékszerinti paraméter:

```
(a,b,c: real; i:integer; b:boolean; c:char; sz:string)
```

Ha egy alprogramnak csak bemenő paraméterei vannak, ahogy az előző példa mutatta, akkor semmilyen eredmény nem kerül ki az alprogramból. Ilyen esetben lehet, hogy a számítás eredményeit csak kiírja az alprogram.

Cím szerinti paraméterek

A cím szerinti paraméterátadásnál az aktuális és a formális paraméterek az memóriában ugyanazon a címen helyezkednek el, ilyen esetben az alprogram a paraméter címét veszi át. Az alprogram törzsben ezek a paraméterek alkotják a kimenő paramétereket, hiszen az értékadó utasítás bal oldalán valójában felveszik a jobb oldali kifejezés értékét. A paraméterlistán a cím szerinti paramétereket az érték szerinti paraméterektől a paraméterek felsorolása előtt megadott **var** kulcsszó különbözteti meg.

A cím szerinti aktuális paraméter csak változó lehet, és nem lehet sem konstans, sem kifejezés. Ha különböző típusú kimenő paramétere van az alprogramnak, akkor ***mindegyik típus elé ki kell tenni*** a **var** kulcsszót.

```
(c,b,d:real; k: integer;
var x,y: real; var n:integer);
```

Összefoglalva az elmondottakat:

A *bemenő paraméterek*, amelyek érték szerint kerülnek átadásra az alprogramnak, kötelezően kell, hogy rendelkezzenek a típusúknak megfelelő értékkel, hiszen ezek a paraméterek az értékadó utasítás jobb oldalán szerepelnek.

A *kimenő paraméterek*, melyeket a **var** kulcsszóval jelöltünk, nem kell, hogy az alprogram hívása előtt rendelkezzenek kezdőértékkel, hiszen éppen az a cél, hogy az alprogram törzsében lévő utasításokkal ezeknek a paramétereknek adjunk értéket.

Példa

```
    (a,b : real; var ered: real)
```

az alprogram törzsében szerepelhet az alábbi értékadás:

```
    ered:= a+b;
```

így az alprogram eredményként bejövő *a* és *b* paraméterek értékének összegét szolgáltatja az *ered* változóban.

Idáig olyan példákat használtunk a paraméterlistán, amelyek egyszerű típusúak voltak : **integer**, **real**, **char**, **boolean**. Ha az alprogram paramétere összetett típusú pl. **array** (tömb), **set** (halmaz), **record** (rekord), **file**, akkor azt csak előredefiniált típusnévvel adhatjuk meg.

Nézzünk egy példát arra, hogyan lehet tömb patamétert megadni az alprogram paraméterlistáján. Ha az alprogram egyik paramétere egy 20 elemű egész típusú több, akkor a tömb számára először létre kell hozni egy *tomb* típust, ezután az alprogram paraméterlistáján már szerepelhet *tomb* típusú paraméter.

```
type
    tomb= array[1..20] of integer;
    ....

    (x,y : tomb; n:integer; var s: real);
```

Tekintsük azt az esetet, amikor bemenő paraméter egy nagyméretű tömb, így ebben az esetben is készül erről egy másolat a megfelelő formális

paraméterbe. Ha azonban a programozó nem tartja szükségesnek ezt a védelmi módszert és inkább kevesebb tárterületet kíván használni, akkor ezt a bemenő paramétert teheti is **var** listára, ily módon az cím szerint kerül átadásra. Így a programozónak kell ügyelnie arra, hogy a tömbből csak olvasson az eljárás a törzsében, vagyis csak az értékadó utasítás jobb oldalán használja azt.

9.1. Függvények

Már megismerkedtünk és használtunk a szabványos függvényeket (***Sqr, Sqrt, Sin, Cos, Abs stb.***) Pascal programokban.
Például:

```
w:= a+b*sin(x)+sqr(2+b);
```

Láthatjuk, hogy a függvényeket az értékadó utasítás jobb oldalán hívtuk.

Feltételezzük, hogy szeretnénk köbre emelő függvényt használni, ez viszont nincs a szabványos függvények között a Pascalban.
Ha

$$Cube = x^3$$

matematikai képletet átírjuk át Pascal kifejezéssé,

```
Cube:= x*x*x;
```

akkor ezt az utasítást beírhatjuk azokra a helyekre, ahol szükséges. Sokkal egyszerűbb lenne azonban az, ha az x négyzetreemelését elvégző *sqr(x)* függvényhez hasonlóan írni egy *Cube(x)* függvényt. Erre van lehetőség, létrehozhatunk egy saját *Cube* függvényt, amelyet felhasználó által definiált függvénynek nevezünk.
A függvény egy olyan alprogram, amely aktiválásakor a nevével ad vissza értéket a hívó programnak. A *Cube* függvénynek az lesz a feladata, hogy az x értéknek meghatározza a köbét.

Ki is írathatjuk a képernyőre

```
writeln(Cube(x));
```

Nézzük meg a függvény általános leírását:

> **function** ***függvény_neve***(*paraméterlista*): *függvény_típusa*:
> *lokális deklarációs rész*
> {**label, const, type, var, procedure, function** }
> **begin**
>   *a függvény törzse (végrehajtható utasítások)*
>   ***függvény_neve***:= .....;
> **end;**

Láthatjuk, hogy a felhasználó által definiált függvény hasonlít a Pascal programhoz. Megtaláljuk benne a deklarációt és a törzsben a végrehajtható utasításokat.

A függvény fejléce

A függvény fejléce a **function** kulcsszóval kezdődik, ezután következik a függvény azonosítója (a neve), a formális paraméterlistája és a függvény típusa.

A függvény azonosítója

A függvény azonosítóját a Pascal azonosítóképzés szabályai szerint kell kialakítani. A példánkban a függvény neve a *Cube*, és a hívásakor egy értéket köbre kell emelnie. A függvény nevére vonatkozik a függvény típusának a megadása, hiszen a típusnak megfelelő eredmény kerül bele.

A paraméterlista

A függvény paraméterlistája tartalmazza azokat a paramétereket, amelyeket át kell vennie a hívó programtól. A paraméternek nemcsak nevet kell adni, hanem a típusát is meg kell határozni. A függvénynél a paramétereket általában név szerint adjuk át, hiszen ebben adjuk meg az értékekeket, amivel a függvény számol. Bonyolultabb esetben a cím szerinti paraméterátadás is használható, például hibajelzés céljából.

Nézzünk néhány paraméterlistát:

```
(x,y : integer)
(a,b: real; i:integer)
(ch: char; n: integer; sz:string)
(s: string; v: integer; var hibakod: integer);
```

A függvény típusa

A függvény fejlécében meg kell adni a függvény típusát is. Ez magától érthetődő, hiszen a függvény általában egyetlen értéket állít elő és ezt a függvény nevével adja vissza, a definiált típusnak megfelelően.

A *Cube* köbreemelő függvénynek egyetlen bemenő paramétere lesz, amelyben megadjuk azt az értéket, amelyet köbre kell emelni. A formális paraméter neve legyen *x*, típusa **real**, a függvény **real** típusú eredményt fog szolgáltatni. Nézzük meg a függvény teljes fejlécét:

```
function Cube(x: real): real;
```

A függvény lokális deklarációi

A függvény fejléce alatt a függvény számára szükséges deklarációk következnek, amelyeket csak a függvény használ. A példánkban a *Cube* függvény nem használ lokális változókat.

A függvény törzse

Végül eljutottunk a függvény törzséhez, amely tartalmazza azokat a műveleteket, amelyeket a függvénynek végre kell hajtani ahhoz, hogy megadja a visszatérési értékét a hívó programnak. Ezeket az utasításokat **begin** és **end** közé kell tenni és pontosvesszővel lezárni.
A *Cube* függvény törzsében szerepelnie kell a

```
Cube:=x*x*x;
```

utasításnak.

Az alábbi két dolgot nem tehetjük meg:

1. A függvény neve függvény törzsén belül nem szerepelhet az értékadás jobb oldalán, például

   ```
   Cube:= Cube * x;
   ```

   Kivéve azt az esetet, ha szándékosan rekurzívan kívánjuk hívni. A rekurzív hívással külön részfejezetben foglalkozunk.

2. A függvény azonosító neve nem szerepelhet az értékadó utasítás bal oldalán a függvény törzsén kívül

   ```
   Cube:=x*x*x;
   ```

   Ez megengedett a függvényen belül, de tilos a függvényen kívül.

A köbreemelő függvényünk teljes listája:

```
function Cube(x : integer) : integer;
begin
    Cube:= x*x*x;
end;
```

Példaként nézzük meg a *Cube* függvény különböző hívásait, ez azért lehetséges, mert a paraméterét érték szerint veszi át:

`m:= Cube(3);`	értékadó utasítás jobb oldalán, paramétere számkonstans.
`writeln(Cube(3));`	**writeln** eljárásban
`y:=3;`	
`m:=Cube(y);`	paramétere változó
`y:=3;`	
`w:=2+Cube(y+1)*2;`	paramétere aritmetikai kifejezés
`if Cube(y)>27 then`	feltételes utasításban is szerepelhet.

Az FUGGVENY.PAS program tartalmazza a *Cube* függvény hívási módjait:

```
program fuggveny;
var
   m,y,w : real;
   function Cube(x : real) : real;
   begin
        Cube := x*x*x;
   end;
begin
     m:=Cube(3);
     writeln('m: ',m:6:1);
     writeln('Cube(3): ',Cube(3):6:1);
     y:=3; w:=Cube(3);
     writeln('w: ',w:6:1);
     y:=3;
     w:=2 + Cube(y)*2;
     writeln('w: ',w:6:1);
     if Cube(y) > 27 then writeln('Cube(y) > 27')
                     else writeln('Cube(y) <= 27');
end.
```

Az program futási eredménye:

```
m:     27.0
Cube(3):     27.0
w:    27.0
w:     56.0
Cube(y): <= 27
```

9.2. Eljárások

Az alprogramok másik fajtája az eljárás. Az eljárás eltérően a függvénytől a nevén keresztül nem ad vissza értéket. Az eljárás felépítésének általános formája:

> **procedure** *eljárásnév(formális paraméterek listája)*;
> *Az eljárás lokális deklarációi*
> {**const, label, type, var, procedure, function** }
> **begin**
>   *az eljárás törzse (utasítások)*
> **end**;

Most nézzük meg részletesen az eljárás felépítését:

Az eljárás fejléce

Az eljárás kulcsszava a **procedure**, ezzel kell kezdeni az alprogramot. Az eljárásnak nevet kell adni, a névnek betűvel kell kezdődnie, a második karaktertől betű és szám is előfordulhat benne, valamint tartalmazhat _ (aláhúzás) karaktert is.

A formális paraméterek listája

Kerek zárójelben kell felsorolnunk az eljárás paramétereit, majd ; zárja az eljárás fejlécét.
Írhatunk olyan eljárást, amelynek nincsenek paraméterei, ilyen eljárások általában valamit kiírnak, például:

```
procedure fejlec;
begin
     writeln('Az egyenletrendszer megoldasa');
end;
```

A *gyok_kiir* eljárás átveszi a két paraméterének megfelelő aktuális értékeket, a képernyőre kiírja, azonban semmilyen értéket nem ad vissza:

```
procedure gyok_kiir(x1,x2 : real);
begin
     writeln('Az egyenlet valos gyokei');
     writeln('x1 : ',x1);
     writeln('x2 : ',x2);
end;
```

A tömb a paraméterlistán csak előredefiniált típusként szerepelhet. Nézzünk példát egy *rendez* eljárásra, ahol a *be_tomb* a rendezendő adatokat, a *ki_tomb* eredményként a rendezett adatokat tartalmazza, tehát a *ki_tomb*-nek a **var** kell listán kell szerepelnie:

```
procedure rendez(n: integer; be_tomb:tomb;
                       var ki_tomb:tomb);
```

Ha a *be_tomb* nagy méretű, a program kevesebb helyet foglal a memóriában, ha a *be_tomb* is cím szerinti paraméter lesz:

```
procedure rendez(n: integer; var be_tomb,ki_tomb:tomb);
```

Irhatunk olyan *rendez* eljárást, hogy a *be_tomb*-ben keletkezzen a rendezett adat, vagyis a bemenő adat felülírásra kerül - ezt megengedi a cím szerinti paraméterátadás:

```
procedure rendez(n: integer; var be_tomb:tomb);
```

Lokális deklaráció

Az eljárás fejléce után következik az eljárás törzsben használt, tehát az eljárásra nézve lokális konstansok, változók deklarációja. A lokális változók az eljárás aktiválásakor kapnak értéket, de értékük kívülről hozzáférhetetlen, amire nincs is szükség. Ezek a változók az algoritmus beprogramozásához szükséges munkatárakat jelentik. Soha nem szabad olyan eljárást írni, amelyben például a ciklus változója a főprogramban deklarált globális változó. Gondoljuk meg milyen bonyodalom származik abból, ha az eljárást a főprogramból ciklusban hívjuk, ahol a ciklus változó *i*, és a ciklus *i* változóját a hívott eljárás módosítja, mert az eljárásban nem deklaráltunk egy lokális *i* változót.

Nem általános az az eljárás, amelynek nincs paramétere. Az ilyen eljárások használatának az oka talán az, hogy a programozó nem értette meg a paraméterek átadási módjait és a főprogramban deklarált globális változókat használja az eljárása, olvas a globális változókból, illetve ír a globális változókba. Ez a módszer azért sem jó, mivel az eljárásnak nincsenek paraméterei, nem lehet az eljárást dinamikusan más paraméterekkel aktíválni. Kerüljük ezt a módszert.

Az eljárás törzse

Amikor már elneveztük a bemenő és kimenő paramétereket, megállapítottuk a típusukat, valamint az eljárás lokális változóit, következik az eljárás algortimusának Pascal utasításokra való átírása. Bonyolultabb algoritmushoz blokkdiagramot is készíthetünk, ahol ellenőrízhetjük a működési elvet, jobban

át tudjuk gondolni a végrehajtási sorrendet. A blokkdiagram azért is jó, mert valóban bonyolultabb algoritmus esetén mint dokumentáció is megmarad, amely szükséges az ejárás esetleges további javításához, bővítéséhez. Könnyebb bizonyos idő után emlékezni az eljárás készítésével kapcsolatos gondolatmenetekre.
Ne felejtsük el, hogy az eljárás törzsét **begin** és **end**; utasítások közé kell zárni.

Példaprogram az eljárás készítésére

Példaként dolgozzuk át a fejezet elején ismertetett HAROMSZ.PAS programot úgy, hogy az adatok beolvasását, a háromszög kerületének ill. területének számítását, valamint az eredmény megjelenítését foglaljuk eljárásokba. A főprogram feladata legyen az eljárások hívásának bemutatása.

a. A háromszög kerületének kiszámítása eljárással

Az eljárás neve legyen:

haromszog_kerulet

Tervezzük meg az eljárás paraméterlistáját és egyben döntsük el a paraméterek típusát. Ahhoz, hogy a változók ne csak egész számokat, hanem valós számokat is tudjanak tárolni, a paraméterek **real** típusúaknak kell lenni. Azok a paraméterek, amelyekből az eljárás számol, a bemenő paraméterek. Az a paraméter, amely az eredményt adja vissza, a kimenő paramétert:

bemenő paraméter az érték szerinti paraméter, a háromszög 3 oldala:
a, b, c : **real**;

kimenő paraméter a cím szerinti paraméter a háromszög kerülete:
var *ker* : **real**;

Az *a,b,c* változók lesznek az érték szerint átadott paraméterek, hiszen ekkor ezeket egyaránt megadhatjuk konstanssal és kifejezéssel is. Az eredmény a *ker* változóban keletkezik, amelyet cím szerinti paraméterként kell megadni.

Az eljárás törzsében a háromszög kerületének kiszámítását jelentő algoritmust kell Pascal kifejezéssé átírni és az eljárás törzsét **begin** és **end** közé zárni. A eljárás utasításai:

```
procedure haromszog_kerulet(a,b,c : real; var ker:real);
begin
    ker:= a+b+c;
end;
```

b. A háromszög területének kiszámítása eljárással

Az eljárás neve legyen:

haromszog_terulet

Az eljárás paraméterei:

A bemenő paraméterei azonosak a *haromszog_kerulet* eljáráshoz, a háromszög 3 oldala : *a, b, c* : **real**;
A kimenő paraméter pedig a háromszög területe : **var** *ter* : **real**;

A *ter* változót cím szerint adjuk át az eljárásnak, mert ebben keletkezik az eredmény, a háromszög területe. Az algoritmus programozásánál beláthatjuk, hogy érdemes lefoglalni egy *s* lokális változót, amelyben egyszer kiszámítjuk a háromszög félkerületét, és a terület kiszámítását végző kifejezésben inkább négyszer használjuk a tartalmát, és nem kell a félkerületet mindig újra kiszámítanunk.
Az eljárás listája:

```
procedure haromszog_terulet(a,b,c:real; var ter:real);
  var
      s: real;
begin
    s:=(a+b+c)/2;
    ter:=sqrt(s*(s-a)*(s-b)*(s-c));
end;
```

Ha az eljárás működését ellenőrizni akarjuk, meg kell írnunk hozzá egy főrprogramot, amely az eljárást hívja.

Az eljárások gyakorlására írjunk még két eljárást, az egyik az adatokat olvassa be, a másik az eredményt írja ki.

d. Az adatokat beolvasó eljárás

Az eljárás feladata legyen az, hogy a háromszög három oldalának adataira kérdezzen rá és eredményként adja vissza a háromszög oldalaira beolvasott adatokat. Ennek az eljárásnak csak kimenő paraméterei lesznek, ezért a paramétereket **var** listára kell tenni, így az eljárásban beolvasott értékek valóban beíródnak a paraméterekbe.

Az eljárás neve: *olvas* .
Kimenő paraméterek: *a,b,c* .
Az *a,b,c* paraméterek **real** típusúak és **var** listán kell lenniük.

Az egyszerű *olvas* eljárás, amely ellenőrzés nélkül olvassa az adatokat:

```
procedure olvas(var a,b,c: real);
begin
     write('A haromszog oldalainak megadasa);
     write('a oldal: '); readln(a);
     write('b oldal: '); readln(b);
     write('c oldal: '); readln(c); writeln;
end;
```

d. Az eredményeket dinamikusan kiíró eljárás

Tervezzük meg az eredményt kiíró eljárást is. Az eljárásnak legyen az a feladata, hogy írja ki a képernyőre a háromszög oldalainak az adatait, egy **string** paraméterben megadott szöveget, valamint egy **real** típusú változó tartalmát. Így ezzel a kiíró eljárással például nemcsak a háromszög területének értékét, hanem a kerület értékét is kiirathatjuk. Mennél ügyesebben szervezzük meg egy eljárás paramétereit, annál sokrétűbb feladatok végrehajtására lesznek alkalmasak.

Az eljárás neve: *kiir*
Bemenő paraméterei: *a,b,c, ered* **real** típusú
szoveg **string** típusú

```
procedure kiir(a,b,c,ered: real; szoveg:string);
begin
     writeln('A haromszog adatai');
     write('a oldal : ',a);
     write('b oldal : ',b);
     write('c oldal : ',c);
     writeln;
     write(szoveg,' ',ered);
     writeln('Nyomj Enter-t !');
     readln;
end;
```

Az eredmény kiírása után a program az *Enter* leütésére vár.

A főprogram készítése

A programnak a neve legyen *haromszog_szamitasok*;
A programban az eljárásokat aktivizáló aktuális változóknak kell helyet foglalni, amelyek szükségesek ahhoz, a beolvasott adatokat illetve az eredményt tárolják:

a_oldal, b_oldal, c_oldal	a háromszög oldalainak adatait tartalmazzák, (**real** típusúak)
h_terulet	a háromszög területének eredményét fogja tárolni, **real** típusú
h_kerulet	a háromszög kerületének eredményét fogja tárolni, **real** típusú
fejlec	az eredmény kiírásához biztosítja a szöveget, **string** típusú

A ELJARAS1.PAS file a teljes programot tartalmazza:

```
program haromszog;
uses crt;
var     a_oldal,b_oldal,c_oldal,
        h_terulet,h_kerulet: real;
                     fejlec: string;
procedure olvas(var a,b,c: real);
begin
     write('A haromszog oldalainak megadasa);
     write('a oldal: '); readln(a);
     write('b oldal: '); readln(b);
     write('c oldal: '); readln(c); writeln;
end;
```

```
procedure kiir(a,b,c,ered: real; szoveg:string);
begin
     writeln('A haromszog adatai');
     write('a oldal : ',a:10:2);
     write('b oldal : ',b:10:2);
     write('c oldal : ',c:10:2); writeln;
     write(szoveg,' ',ered:10:2);
     writeln('Nyomj Enter-t !'); readln;
end;
procedure haromszog_kerulet(a,b,c : real;
                          var ker:real);
begin
     ker:= a+b+c;
end;
procedure  haromszog_terulet(a,b,c:real;  var  ter:real):
var
       s: real;
begin
     s:=(a+b+c)/2.;
     ter:=sqrt(s*(s-a)*(s-b)*(s-c));
end;
(* foprogram blokkja *)
begin clrscr; (* a kepernyo torlese *)
      (* a haromszog adatainak beolvasasa *)
     olvas(a_oldal,b_oldal,c_oldal);

      (* a haromszog teruletenek szamitasa *)
      writeln('Az eljaras hivasa: valtozokkal ');
     haromszog_terulet(a_oldal,b_oldal,c_oldal,
                         h_terulet);
     kiir(a_oldal,b_oldal,c_oldal,
     'A haromszog terulete: ',h_terulet);

      (* eljarast  hivasa konstanssal *)
      writeln('Az eljaras hivasa: konstasokkal');
     haromszog_terulet(3,4,5,h_terulet);
     kiir(3,4,5,'A haromszog terulete: ',h_terulet);

      (* eljaras hivasa kifejezessel *)
     writeln('Az eljaras hivasa: kifejezesekkel ');
     haromszog_terulet(a_oldal+1,2*b_oldal,c_oldal,
                          h_terulet);
     kiir(a_oldal+1,2*b_oldal,c_oldal,
             'A haromszog terulete:',h_terulet);

     (* A haromszog keruletenek szamitasa *)
     haromszog_kerulet(a_oldal,b_oldal,c_oldal,
```

```
                        h_kerulet);

        kiir(a_oldal,b_oldal,c_oldal,
             'A haromszog kerulete:',h_kerulet);
    end.
```

Az ELJARAS.PAS program futási eredménye: Mivel az eredmény hosszabb, mint egy képernyő, ezért egy paraméter nélküli ***readln*** eljárással felfüggesztjük a program futását a kiértékelés számára és *Nyomj Enter-t !* szöveg jelzi, hogy a program az *Enter* leütésére vár.

```
A haromszog oldalainak megadasa
a oldal: 3
b oldal: 4
c oldal: 5

Az aljaras hivasa: valtozokkal
A haromszog adatai
a oldal:        3.00
b oldal:        4.00
c oldal:        5.00

A haromszog terulete:       6.00
Nyomj Enter-t

Az eljaras hivasa: konstansokkal
A haromszog adatai
a oldal:        3.00
b oldal:        4.00
c oldal:        5.00

A haromszog terulete:       6.00
Nyomj Enter-t

Az eljaras hivasa: kifejezesekkel
A haromszog adatai
a oldal:        3.00
b oldal:        4.00
c oldal:        5.00
A haromszog terulete:       6.00
Nyomj Enter-t

A haromszog adatai
a oldal:        4.00
b oldal:        8.00
c oldal:        5.00
```

```
A haromszog terulete:      8.18
Nyomj Enter-t

A haromszog adatai
a oldal:       3.00
b oldal:       4.00
c oldal:       5.00

A haromszog kerulete:      12.00
Nyomj Enter-t
```

Módosítsuk először a beolvasó eljárást úgy, hogy ellenőrizze az adatokat a háromszög megszerkeszthetősége szempontjából. Három oldalból csak akkor szerkeszthető meg a háromszög, ha páronként két oldal összege nagyobb, mint a harmadik oldal.
Tehát az alábbi feltételeknek kell teljesülni:

$(a+b) > c$ és
$(a+c) > b$ és
$(b+c) > a$

Ha ezek a feltételek nem teljesülnek adjon az *olvaso* eljárás figyelmeztetést a felhasználónak és várja az új adatok megadását.
A módosított *olvas* eljárás;

```
procedure olvas(var a,b,c: real);
var f : boolean;
begin
    write('A haromszog oldalainak megadasa);
    repeat
     write('a oldal: '); readln(a);
     write('b oldal: '); readln(b);
     write('c oldal: '); readln(c);
          if (a+b>a)and (a+c>b) and (b+c>a) then f:=true
                        else
                      begin
                           f:=false;
                           writeln('Hibas adatok');
                      end;
    until f;
end;
```

Az ELJARAS2.PAS program tartalmazza ezt az olvasó eljárást, amely azt figyeli, hogy az adatok eleget tesznek-e a háromszög megszerkeszthetőségi feltételeknek. Az ELJARAS3.PAS a beolvasó eljárás olyan továbbfejlesztett változatát tartalmazza, amely nem fogad el hibásan megadott adatot.

Csak összehasonlítás szempontjából írjuk át a *haromszog_terulet* eljárást függvénynek.

A függvény neve legyen: *fg_haromszog_terulet*
Bemenő paraméterei : *a,b,c* **real** típusúak.
Az eredmény a függvény nevében, az *fg_haromszog_terulet* -ben keletkezik.
A függvény a következő:

```
function fg_haromszog_terulet(a,b,c:real): real;
 var
     s: real;
begin
     s:=(a+b+c)/2.;
     fg_haromszog_terulet:=sqrt(s*(s-a)*(s-b)*(s-c));
end;
```

Ha ezt a függvényt beillesztjük az előzőekben ismertetett főprogram deklarációs részébe, az *fg_ter* változót **real** típusként deklaráljuk és a kiegészítjük a főprogramot az alábbi utasításokkal:

```
olvas(a_oldal,b_oldal,c_oldal);
fg_ter:=fg_haromszog_terulet(a_oldal,b_oldal,c_oldal);

kiir(a_oldal,b_oldal,c_oldal,
'A haromszog terulete fuggvennyel: ',fg_ter);
```

A módosított programot (ELJARASF.PAS) lefuttatva az eljárásnak és a függvénynek ugyanazt az eredményt kell adnia.

9.3. Típusdefiníció használata a paraméterlistán

Mint már említettük az eljárások és függvények formális paraméterlistáján csak olyan változó lehet paraméter, amelynek külön típusneve van. Ilyen például az **integer**, **real**, **word**, **char**, **string**, **boolean** stb., de nem

használhatjuk az **array**, **set**, **file**, **rekord** változókat közvetlenül, ezekre típust kell definiálni, a **type** deklaráció segítségével.
Ugyancsak felhasználói típust kell létrehoznunk a **string**[*n*] típusú paraméterek deklarálásához.

```
type
    tomb     = array[1..10][1..10] of real;
    betuk    = set of [a..z];
    tfile    = file of integer;
    komplex  = record
                 x,y: real;
               end;
    st12     = string[12];
var
    st12:string[12]
    a   : tomb;
    f   : tfile;
    abc : betuk;
    w,y : komplex;
    fnev: st12;
    procedure (x:tomb; olv: tfile; fn: st12);
    ..
    begin
    ...
    end;
```

9.4. Eljárás paramétere: függvény

Az eljárás paraméterlistáján függvény is szerepelhet. Ennek megvalósításához az alábbiakat kell tennünk:

- az eljárást a {$F+} fordítási opcióval kell fordítani,
- meg kell adni a függvény típusát, például

```
type
    fuggv= function(x:real):real;
```

- meg kell adni a felhasználói függvényt, amelynek bemenő paraméterei típusban és darabszámban, valamint kimenő érték típusában megegyeznek.

Legyen a feladat egy táblázatkiíró eljárás írása, amely a paraméterként megkapott függvényt adott intervallumban tabellázza, fejlécként a függvény képletét is kiírja.
Először olyan típust kell létrehoznunk, amely megegyezik a tábellázni kívánt függvény paramétereivel és a függvény típusával. Mivel a két tabellázandó függvényünk

```
function tablazat1(x:real):real;
function tablazat2(x:real);real;
```

ezért az alábbi típust hozzuk létre:

```
type
    fuggveny =function(x: real): real;
```

A a *kiir* eljárás paraméterei:

x1,x2	a táblázat kezdő és végértéke
dx	a táblázat lépésköze
f	a tábellázandó függvény
szoveg	a fejlécként kiirandó szöveg

A *kiir* eljárás feje:

```
procedure kiir(x1,x2,dx:real; f:fuggveny;
                szoveg: string);
```

A FUGG_PAR.PAS program listája:

```
{$F+}
(*  eljaras parametere fuggveny *)
(*  file neve: fugg_par         *)
program fuggveny_parameter;
type fuggveny   = function(x:real): real;
var
    x1,x2,dx : real;
    fejlec1,fejlec2: string;
    function tablazat1(x: real): real;
    begin
       tablazat1:= 3*sin(x*x);
    end;
```

```
    function tablazat2(x: real): real;
    begin
       tablazat2:=2.5 * x * x/cos(x);
    end;
    procedure kiir(x1,x2,dx:real; f:fuggveny;
                        szoveg: string);
    var  x:real;
    begin
       x:=x1;
       writeln('   x        ',szoveg);
       while x<x2 do
       begin
           writeln(x:6:2,f(x):12:4);
           x:=x+dx;
       end;
    end;
    begin
      x1:=0.1; dx:=0.2; x2:=1.1;
      fejlec1:='3*sin(x)';
      fejlec2:='2.5*x*x/cox(x)';
      writeln('Tablazatkszites fuggvenyparameterrel');
      writeln;
      writeln('   Elso tablazat ');
      kiir(x1,x2,dx,tablazat1,fejlec1);
      writeln;
      writeln('   Masodik tablazat ');
      kiir(x1,x2,dx,tablazat2,fejlec2);
   end.
```

A program eredménye:

```
Tablazatkeszites fuggvenyparameterrel
   Elso tablazat
   x        3*sin(x*x)
  0.20        0.1200
  0.40        0.4780
  0.60        1.0568
  0.80        1.7916
  1.00        2.5244
   Masodik tablazat
   x       2.5*x*x/cos(x)
  0.20        0.1020
  0.40        0.4343
  0.60        1.0905
  0.80        2.2965
  1.00        4.6270
```

9.5. Forward deklaráció - előre hivatkozás

A **forward** deklarációt akkor használjuk, ha a függvényt vagy az eljárást előbb használjuk, mint ahogy deklaráljuk. Ilyenkor az eljárásfejet a **forward** kulcsszóval deklaráljuk. Példa a **forward** deklarációra:

```
procedure olvas(var a,b :real); forward;
procedure szamol(var x,y: real);
var
      aa,bb :real;
begin
      olvas(aa,bb);
      x:= srq(aa);
      y:= sqrt(bb);
end;
procedure olvas(var a,b,:real);
begin
      write(' a = '); readln(a);
      write(' b = '); readln(b);
end;
```

A **forward** deklarációban használt függvény, vagy eljárás teljes deklarációjában elegendő a **function** illetve **procedure** kulcsszó után a függvény nevét megadni:

```
procedure olvas;
begin
      write(' a = '); readln(a);
      write(' b = '); readln(b);
end;
```

9.6 Rekurzív alprogramok

A matematikában lehetőség van bizonyos adatok és műveletek rekurzív definiálására. Ne feledjük el azonban, hogy minden rekurzív problémának létezik iteraktív megoldása, amely általában sokkal nehezebben programozható, de hatékonysága miatt mégsem szabad megfeledkezni róla!

Klasszikus példaként tekintsük először a Fibonacci néven is ismert Leonardo de Pisa nyúl feladatát:

"Hány nyúlpárunk lesz 3,4,5,...,n hónap múlva, ha egy nyúlpár kéthónapos kortól kezdve havonta egy-egy új párt hoz világra. Feltéve, hogy az új párok is ezen törvény alapján szaporodnak, és mind életben maradnak."

A megoldást a Fibonacci számok sora tartalmazza (ha a 0 kezdőelemet figyelmen kivül hagyjuk):

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

A sort n-dik elemének meghatározására az alábbi rekurziós szabály szolgál:

$$a_0 = 0$$
$$a_1 = 1$$
$$a_n = a_{n-1} + a_{n-2}, \qquad n: 2, 3, 4, \ldots$$

A rekurziós szabály felhasználásával, elegáns megoldásként kihasználhatjuk azt, hogy a Pascal nyelben egy eljárás önmagát is meghívhatja:

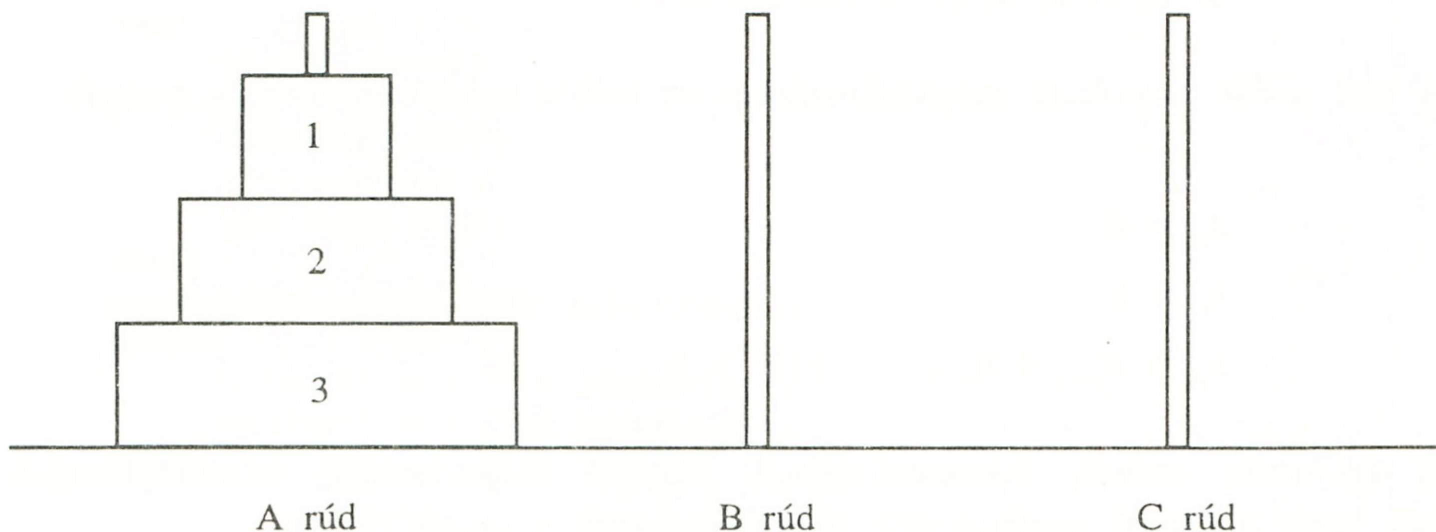
```
function fibr(n : integer) : longint;
      begin
        if n <= 1 then fibr := n
                   else fibr := fibr(n-1) + fibr(n-2);
      end;
```

A rekurzív megoldás általában rövidebb és áttekinthetőbb mint az iteraktív megoldás, azonban a számítási idő jelentős növekedése miatt, mégis az iteraktív megoldás használatát javasoljuk:

```
function fib(n : integer) : longint;
   var f0,f1,f2:longint;
      begin
        f0:=0; f1:=1; f2:=n;
        while (n>1) do
          begin
            f2:=f0+f1; f0:=f1;  f1:=f2;
            dec(n);
          end;
        fib:=f2;
      end;
```

A következő feladat megoldása a rekurzió alkalmazása nélkül igen bonyolult. A megoldandó probléma a Hanoi tornyai nevet viseli.

Adott három rúd: A, B és C. Az A rúdon induláskor N darab különböző átmérőjű lyukas korong helyezkedik el, nagyságuk szerinti csökkenő sorrendben:



A feladat a korongok átrakása a C rúdra, az alábbi szabályok figyelembevételével:

- a B rúd közbenső tárolásra használható,
- minden lépésben csak egy korong mozgatható,
- minden korong csak nálánál nagyobb átmérőjű korongra helyezhető.

(A feladat egy legendán alapul, amely szerint Hanoi közelében található kolostorban 64 aranykorongból álló tornyot raknak át a szerzetesek, a fenti szabályok betartásával, minden nap egyetlen korongot mozgatva. A legenda szerint akkor lesz vége a világnak, ha az átrakást befejezik.)

A feladat megoldását az alábbi példaprogram tartalmazza:

```
program Hanoi_tornyai;
      var
         korongszam:integer;

      procedure Hanoi(n:integer;honnan,mivel,hova: char);

         procedure honnan_hova_mozgatas;
         begin
           write('Tedd a korongot a(z) ',#9);
           write(honnan,#9);
           write(' rúdról a(z) ',#9);
           write(hova,#9,' rúdra.');
           writeln;
         end  { honnan_hova_mozgatas };
         begin
           if n=1 then honnan_hova_mozgatas
           else
             begin
               hanoi(N-1, honnan, hova, mivel);
               honnan_hova_mozgatas;
               hanoi(N-1, mivel, honnan, hova)
             end
         end;   { Hanoi }

      begin
         writeln;
         write('A korongok száma : ');
         readln(korongszam);
         writeln;
         hanoi(korongszam, 'A', 'B', 'C')
      end.
```

Három korong megadása esetén a program által javasolt megoldás:

```
A korongok száma : 3

Tedd a korongot a(z)    A     rúdról a(z)   C      rúdra.
Tedd a korongot a(z)    A     rúdról a(z)   B      rúdra.
Tedd a korongot a(z)    C     rúdról a(z)   B      rúdra.
Tedd a korongot a(z)    A     rúdról a(z)   C      rúdra.
Tedd a korongot a(z)    B     rúdról a(z)   A      rúdra.
Tedd a korongot a(z)    B     rúdról a(z)   C      rúdra.
Tedd a korongot a(z)    A     rúdról a(z)   C      rúdra.
```

9.6.1 A rekurzív alprogramok csoportosítása

A rekurzív eljárások és függvények általában az alábbiak szerint csoportosíthatók.

- **Önrekurzió**:

Ekkor a rekurzív eljárás vagy függvény közvetlenül hívja önmagát. A fenti két példában ezt a megoldást használtuk. A rekurzív működés megértéséhez tekintsük a faktoriális számítást, amely rekurzív megoldással szintén hosszasan működik, így ebben az esetben is az iteratív (ciklusos) megoldás javasolt:

A faktoriális rekurzív definíciója:

$$n! = \begin{cases} 1, & \text{ha } n=0 \\ n*(n-1)!, & \text{ha } n>0 \end{cases}$$

A rekurzív definíció felhasználásával 4! kiszámításának lépései:

```
4!= 4*3!
      3*2!
        2*1!
          1*0!
            1 = 4*3*2*1*1=24
```

A fenti számítási menetet megvalósító Pascal függvény:

```
function factr(n:integer):longint;
      begin
        if n=0 then factr:=1
                else factr:=n*factr(n-1);
      end;
```

A teljesség kedvéért közöljük a hatékonyabb interaktív megoldást is:

```
function fact(n:integer):longint;
 var
    f:longint;
 begin
    f:=1;
    while (n>0) do
     begin
       f:=f*n;
       dec(n);
     end;
   fact:=f;
 end;
```

- **Kölcsönös rekurzió**

Kölcsönös rekurzióról akkor beszélünk, ha egy alprogram egy általa meghívott másik alprogramból hívódik meg újra. Az ilyen megoldáshoz szükséges a **forward** deklaráció használata:

```
procedure b(j:integer); forward;
      procedure a(i:integer);
       ...
       begin
         ...
         b(i);
         ...
       end;

      procedure b;
       ...
       begin
         ...
         a(j);
         ...
       end;
```

9.7. Globális és lokális változók, az azonosítók érvényességi köre

A HAROMSZ.PAS programban is láttuk, hogy a program deklarációjában lévő változók:
var *a_oldal, b_oldal, c_oldal, h_terulet, h_kerulet:* **real;**

az egész programtörzsre vonatkoztak. Ezek a változók a program törzsére néve *globális változók*.
A *haromszog_terulet* eljárás paraméterlistáján az *a,b,c,ter* változók *formális* paraméterek. Az eljáráson belül deklarált *s* változó *lokális*, mert csak az eljárás törzsén belül érhető el.

```
procedure haromszog_terulet(a,b, c: real; var ter: real);
var
    s: real;
begin
    s:= ...
end;
```

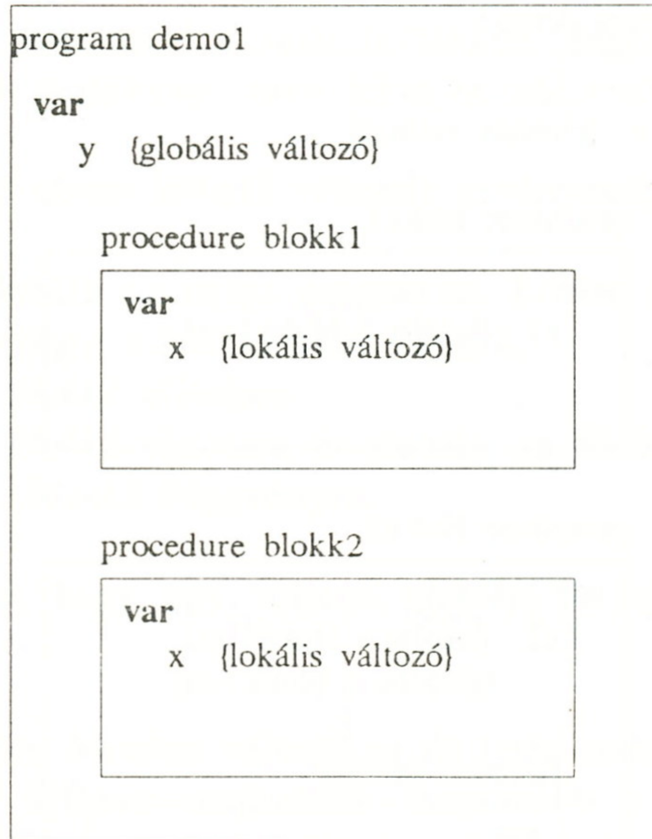
A főprogramban az *olvas* eljárás hívása a globális aktuális paraméterekkel történik, amelyekre helyet foglaltunk a memóriában:

```
olvas(a_oldal, b_oldal, c_oldal);
```

Ezekkel az értéket kapott változókkal hívjuk tovább a *haromszog_terulet* eljárást:

```
h_terulet(a_oldal,b_oldal,c_oldal,h_terulet);
```

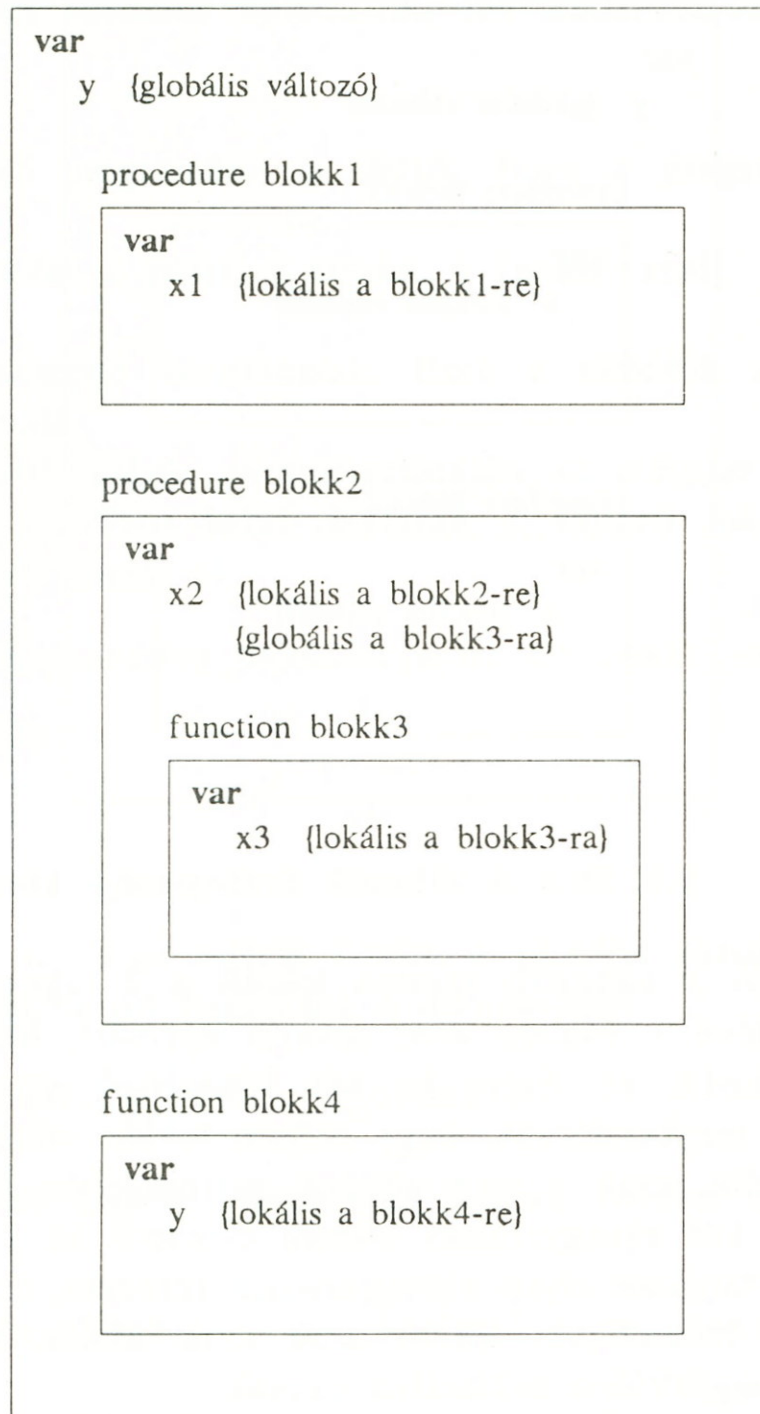
Tekintsük meg a 9.1. ábrát, amelyben a főprogram két alprogramot tartalmaz.
A főprogram deklarációjában lévő *y* változó globális a teljes programra néve, így az alprogramokra is. A *Blokk1* alprogram deklarációjában szereplő *x* változó lokális az 1. alprogramon belül.



9.1. ábra A változók érvényességi köre

A *Blokk2* alprogram *x* változója szintén lokális a 2. alprogramon belül. A két alprogramban lévő *x* változó nem zavarja egymást, hiszen teljesen más tárterületen helyezkedik el. Ezért ha két különböző személy alprogramot fejleszt, nem kell megbeszélniük, hogy milyen lokális változókat használnak az alprogramon belül, csak a paraméterek sorrendjében és típusában kell megállapodniuk. A két alprogramban viszont elérhető az *y* változó tartalma is. Egyébként nem tanácsos olyan alprogramokat tervezni, amelyek különböző globális változókat használnak, mivel azok csak akkor működnek, ha a globális változóik megfelelően deklarálva vannak.

A 9.2. ábrán látható főprogram több alprogramot tartalmaz, sőt alprogramon belül újabb alprogram is van. Értékeljük ki itt is a változók érvényességi körét.



9.2. ábra Bonyolultabb blokkszerkezet

A főprogram **var** deklarációjában szereplő *y* változó globális az összes alprogramra, kivéve a Blokk4 függvényt, mivel lokális változójának neve megegyezik a főprogram *y* változójának a nevével. A két *y* változó más memóriacímre kerül, függetlenek egymástól. Egyébként az eljárásokban ill.

függvényekben lévő deklaráció a saját törzsükre (**begin ... end**) vonatkozik, tehát lokálisak, ahhoz máshonnan nem lehet hozzáférni.

Foglaljuk össze a 9.2. ábrán látható változók érvényességi körét:

- *y* változó globális az egész programra, kivéve a *Blokk4* függvényt, ahol az *y* (egy másik) lokális változó,
- *x1* lokális a *Blokk1* eljárásra
- *x2* lokális a *Blokk2* eljárásra és globális az *Blokk3* függvényre,
- *x3* lokális az *Blokk3* függvényre.

A példából is látható, hogy egy változó globális és lokális lehet, ez relatív fogalom.

A változók élettartama: Minden változó az őt tartalmazó blokkba belépve jön létre, és a blokkból kilépve megszünik. Legtovább a főblokkban deklarált változók élnek, amelyek a program teljes futása alatt elérhetők (statikus változók).

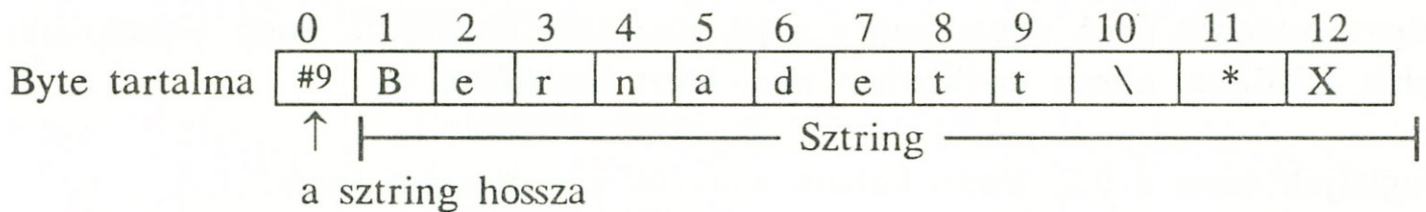
9.8. Sztringek használata Turbo Pascal-ban

A **char** adattípus egyetlen byte-on tárolja a karaktert. A sztring karakter típusú elemekből épül fel. A sztring adattípus szöveges információt tárol. A szöveg hossza 255 karakterig terjedhet.

Például:

```
var
     nev : string[12];

begin
     nev:='Bernadett';
end;
```

Ebben a példában a *nev* sztring változó 13 byte-ot foglal le a memóriából, mivel a 0. byte-ja a sztring hosszát tartalmazza:



A sztring hosszát az *ord(nev[0])* kifejezés adja meg. A példában a sztringbe írt szöveg hossza 9 karakter, binárisan 0000 1001. Mivel a 12 karakterre deklarált sztringbe csak 9 karaktert helyeztünk el, az utolsó három byte véletlenszerű értékeket tartalmaz.

9.8.1. Szabványos eljárások és függvények a sztringek kezelésére

Chr függvény

A *Chr* függvény **integer** paramétere egész számot fogad és annak megfelelő sorszámú karaktert ad vissza. Például az ASCII kód 65 az *A* karaktert jelöli, akkor a *Chr*(65) az *A* karaktert adja vissza. Írjuk ki a képernyőre az Ok! szöveget.

```
(* chrtest.pas *)
program chrtest;
uses crt;
var
        s: string[20];
begin
        clrscr;
        s:= 'Ok' + chr(19);
        writeln(s);
end.
```

A program eredménye:

Ok!

Uppcase függvény

Az Uppcase karakter szintű függvény, amely egyetlen kisbetű karaktert fogad 'a' .. 'z' -ig és visszatér a nagybetűs megfelelőjével. Ha az Uppcase paramétere nem kisbetű, akkor változatlanul adja vissza a karaktert.

```
(* upcaset.pas *)
program upcaset;
uses crt;
var
        s: string;
        i: integer;
begin
        clrscr;
        s:='abcdefgHIJkl';
        writeln('sztring      : ',s);
        for i:=1 to length(s) do
               s[i]:=Upcase(s[i]);
        writeln('Upcase utan: ',s);
end.
```

A program eredménye:

```
sztring     : abcdefgHIJkl
Upcase utan: ABCDEFGHIJKL
```

Concat függvény

A sztrigek összefűzésének két változata is létezik a Turbo Pascal-ban. Használhatjuk a *concat* függvényt, vagy a + (plus) operátort.
A szabványos *concat* függvénnyel bármilyen számú sztring paramétert összefűzhetünk, de az összefűzött sztring hossza sem lehet 255 karakternél nagyobb.

```
(* sztring1.pas *)
program sztring1;
uses crt;
var
        s1,s2,s3,s4: string;
begin
        s1:='Ez pelda ';
        s2:='a concat';
        s3:=concat(s1,s2,' fuggvenyhivasra');
        writeln('concat    : ',s3);
```

```
        s4:='Ez pelda' + ' a sztringek ' + 'osszeadasara.';
        writeln('+ muvelet: ',s4);
end.
```

A program eredménye:

```
concat   : Ez pelda a concat fuggvenyhivasra
+ muvelet: Ez pelda a sztringek osszeadasara
```

Ha a *concat* függvénnyel, vagy a + operátorral összefűzött sztring hossza túlnő a sztring maximális hosszánál (255 karakter), akkor az eredménysztring a 255. karakter után elveszik.

Copy függvény

A ***copy*** függvénnyel részsztringet másolhatunk ki egy nagyobb sztringből. A ***copy*** függvénynek három paramétere van. Az első paraméter a sztring, amelyből másolni akarunk, a második a karakter sorszáma, ahonnan másolni akarunk és a harmadik paraméter, pedig megmondja, hogy mennyi karakter kerüljön másolásra.

```
(* sztring2.pas *)
program sztring3;
uses crt;
var
        s1,s2: string;
begin
        clrscr;
        s1:='Elemer';
        writeln('Keresztnev        : ',s1);
        s2:=copy(s1,1,3);
        writeln('Copy hivasa utan: ',s2);
        s2:=s2+'k';
        writeln('Uj nev            : ',s2);
end.
```

A program eredménye:

```
Keresztnev       : Elemer
Copy hivasa utan: Ele
Uj nev           : Elek
```

Delete eljárás

A **Delete** eljárás sztringből karaktereket töröl. Az eljárás első paramétere a sztring, ahonnan törölni kell, a második paramétere az első törlendő karakter helyét jelenti és a harmadik paraméterben adjuk meg a törlendő karakterek számát.

```
(* sztring3.pas *)
program sztring2;
uses crt;
var
     s1,s2: string;
begin
     clrscr;
     s1:='Katoka';
     writeln('Keresztnev         : ',s1);
     Delete(s1,4,3);
     writeln('Delete hivas utan: ',s1);
     s2:=concat(s1+'inka');
     writeln('Modositott nev     : ',s2);
     end.
```

A program eredménye:

```
Keresztnev         : Katoka
Delete hivas uran : Kat
Modositott nev     : Katinka
```

Insert eljárás

Az *insert* eljárással sztringbe részsztringet szúrhatunk be. Az eljárás első paramétere a beszúrandó részsztring, a második paramétere a módosítandó sztring és a harmadik paramétere határozza meg, hogy a módosítás hányadik karaktertől kezdődjön.

```
(* sztring4.pas *)
program string4;
uses crt;
var
     s1,s2: string;
begin
     clrscr;
     s1:='Ma az ido';
```

```
        writeln(s1);
        insert(' szep',s1,3);
        writeln('insert utan: ',s1);
end.
```

A program eredménye:

```
Ma az ido
insert utan : Ma szep az ido
```

Length függvény

A **length** függvény a paraméterében megadott sztring hosszát adja vissza. A példaprogramban a **length** függvénnyel meghatározzuk a sztring végét és **insert** hívásával bővítjük a sztring tartalmát.

```
(* sztring5.pas *)
program string5;
uses crt;
var
        s1,s2: string;
        k    : integer;
begin
        clrscr;
        s1:='Ma az ido';
        writeln(s1);
        insert(' szep',s1,3);
        writeln('insert utan   : ',s1);
        s2:=', de hideg van.';
        k:=length(s1);
        insert(s2,s1,k+1);
        writeln('Vegere toldva: ',s1);
end.
```

A program eredménye:

```
Ma az ido
Insert után    : Ma szep az ido
Vegere toldva: Ma szep az ido, de hideg van.
```

Pos függvény

A *pos* függvény segítségével sztringben részsztringet lehet keresni. A *pos* első paramétere a részsztring, a második paramétere az a sztring, amelyben keresünk és a függvény visszatérési értéke lehet

- 0, ha a rész sztringet nem találta,
- pozitív szám azt jelenti, hogy a rész sztringet megtalálta a keresett sztringben, és az értéke a megtalálás helyének a pozíciója.

```
(* postest.pas *)
program postest;
uses crt;
var
     a:string[80];
begin
     clrscr;
     a:='Szovegben xx karaktersorozat keresese';
     writeln('Az ''xx'' pozicioja: ',pos('xx',a));
     writeln('Az ''XX'' pozicioja: ',pos('XX',a));
     writeln;
end.
```

A program eredménye:

```
Az 'xx' pozicioja : 11
Az 'XX' pozicioja : 0
```

Str eljárás

Az ***str*** eljárásnak két paramétere van. Az első paramétere szám (**integer** vagy **real**), amelyet a második paraméterben megadott sztringbe karaktersorozattá alakít át. A egész szám paraméternél használhatjuk a mezőszélességet, valamint valós szám esetén a tizedek számára vonatkozó megadási módot is. Ha valós számnál 0 mezőszélességet adunk meg, akkor a a szám egy egész és egy tizedes normált alakban jelenik meg.

```
program strtest;
uses crt;
var
     i: integer;
     a: real;
     s: string[20];
begin
     clrscr;
     i:=15;
```

```
        str(i,s);
        writeln(s);
        str(i:4,s);
        writeln(s);
        a:=4.5;
        str(a,s);
        writeln(s);
        a:=0.05;
        str(a:0,s);
        writeln(s);
        str(a:10:2,s);
        writeln(s);
    end.
```

A program eredménye:

```
15
  15
 4.5000000000E+00
 5.0E-02
      0.05
```

Val eljárás

A Val eljárásnak három paramétere van: sztring, amelyet számmá kell átalakítani, numerikus változó (**integer** vagy **real**), amely a számértéket fogadja és egy egész típusú változó a hiba jelzésére. Ha a hibaváltozó értéke 0, akkor a konverzió hibátlan volt, egyébként az elsőnek előforduló hiba helyére mutat a sztringben.

```
(* valtest.pas *)
program valtest;
uses crt;
var
     s:string;
     x:real;
     j:integer;
  hiba:integer;
begin
     clrscr;
     s:='1.53e-2';
     val(s,x,hiba);
     if hiba=0 then writeln(s,' jo szam: ',x)
           else writeln(s,' valos szam ',hiba,
```

```
                                  '. helyen hibas!');
      s:='1.5.3e-2';
      val(s,x,hiba);
      if hiba=0 then writeln(s,' jo szam: ',x)
            else writeln(s,' valos szam ',hiba,
                                  '. helyen hibas!');
      writeln;
      s:='102';
      val(s,j,hiba);
      if hiba=0 then writeln(s,' jo szam: ',j)
            else writeln(s,' egesz szam ',hiba,
                                  '. helyen hibas!');
      s:='10.2';
      val(s,j,hiba);
      if hiba=0 then writeln(s,' jo szam: ',j)
            else writeln(s,' egesz szam ',hiba,
                                  '. helyen hibas!');
      end.
```

A program eredménye:

```
1.53e-2 jo szam: 1.5300000000E-2
1.5.3e-2 valos szam 4. helyen hibas!

102 jo szam: 102
10.2 egesz szam 3. helyen hibas!
```

Mint láttuk, a sztringek tárolása tömbben történik. Habár a sztringek kezelésére sok könyvtári függvény használható, sok esetben mégis egyszerűbb és gyorsabb, ha a tömbben való tárolást kihasználva a tömb elemein végezzük el a kívánt műveletet. Példaként írjunk eljárást, amely adott sztringben minden angol betűt nagybetűvé alakít.
Az első eljárás (*strup1*) tömb elemein végzi el az átalakítást, míg az *strup2* sztringkezelő függvényeket és eljárásokat használ.

```
procedure strup1(var s:string);
var
     i: integer;
begin
     for i:=1 to ord(s[0]) do
       s[i]:=upcase(s[i]);
end;
```

```
procedure strup2(var s:string);
var
     i: integer;
     c: string[1];
begin
     for i:=1 to length(s) do
     begin
        c:= copy(s,i,1);
        c[1]:=upcase(c[1]);
        delete(s,i,1);
        insert(c,s,1);
     end;
end;
```

Gyakorlat

1. Sztring hosszát változtassuk meg úgy, hogy a 0-dik elemét írjuk át *chr* függvénnyel, majd írjuk ki a sztring tartalmát. Mit tapasztalunk?

```
(* sztring6.pas *)
program sztring6;
uses crt;
var
     s: string;

begin
     clrscr;
     s:='abcdefg';
     writeln(s,' sztring hossza: ',length(s));
     s[0]:= chr(4);
     writeln(s,'    sztring hossza: ',length(s));
end.
```

A SZTRING6.PAS program eredménye:

```
abcdefg sztring hossza: 7
abcd    sztring hossza: 4
```

2. Szövegben való keresést és helyettesítést mutatja be a SZTRING7.PAS program. Értékeljük ki az eredményt!

```
(* sztring7.pas *)
program sztring7;
uses crt;
```

```
var
     Szoveg        : string;
     Keres,helyez: string[20];
                  i: integer;
begin
     clrscr;
     Szoveg:='Ma Kati hianyzott az orarol.';
     Keres:='Kati';
     Helyez:='Janos';
     writeln(Szoveg);
     i:=Pos(Keres,Szoveg);
     Delete(Szoveg,i,length(Keres));
     Insert(Helyez,Szoveg,i);
     writeln(Szoveg);
end.
```

A program eredménye:

```
Ma Kati hianyzott az orarol.
Ma Janos hianyzott az orarol.
```

3. Szövegben # karaktert helyettesítsünk klaviatúráról beolvasott névvel.

```
(* nevins.pas *)
program nevins;
uses crt;
type
       str255=string;
var
     uzenet1,uzenet2,uzenet3: string;
     nev: string[20];

     function uzenet_ir(s,nev:string):string;
     var
          i:integer;
     begin
          i:=Pos('#',s);
          if i>0 then
              begin
                      Delete(s,i,1);
                      Insert(nev,s,i);
                    end;
          uzenet_ir:=s;
     end;
begin
     uzenet1:=' Hello, #!';
```

```
        uzenet2:=' Az uzenet valtozatlan.';
        uzenet3:=' Ez az uzenet, #, megvaltozott.';
        clrscr;
        write('Adja meg a nevet: '); readln(nev);
        writeln(uzenet_ir(uzenet1,nev));
        writeln(uzenet_ir(uzenet2,nev));
        writeln(uzenet_ir(uzenet3,nev));
        writeln;
   end.
```

A NEVINS.PAS program eredménye:

```
   Adja meg a nevet: Kati
   Hello Kati!
   Az uzenet valtozatlan.
   Ez az uzenet, Kati, megvaltozott.
```

4. Sztringben az számok mögötti üres helyek törlése után a sztringet a ***val*** függvénnyel alakítsuk át számmmá.

```
   (* spacedel.pas *)
   program spacedel;
   uses crt;
   var
        s      : string;
        i,code : integer;
        procedure BlankDel(var s: string);
        begin
             while (s[length(s)] = ' ') do
                  Delete(s,length(s),i);
        end;
   begin
        clrscr;
        s:='          33      ';
        writeln('sztring          : <',s,'>');
        BlankDel(s);
        writeln('BlankDel utan : <',s,'>');
        Val(s,i,code);
        writeln('Ertek : ',i);
   end.
```

A SPACEDEL.PAS program eredménye:

```
   sztring       : <          33      >
   BlankDel utan : <          33>
   Ertek : 33
```

Ellenőrző kérdések:

1. Mire szolgál az alprogram készítése?
2. Hány fajta alprogram létezik a Pascal-ban?
3. Mit nevezünk formális és aktuális paraméternek?
4. Mit értünk érték szerinti paraméterátadás alatt?
5. Mit értünk cím szerinti paraméterátadás alatt?
6. Ismertesse a függvény általános felépítését?
7. Hol látszik, hogy a függvény milyen típusú értéket ad vissza?
8. A függvény hívási módjai?
9. Ismerttesse az eljárás általános felépítését?
10. Mi a különbség a függvény és az eljárás hívása között?
11. Mi kell tennünk ahhoz, hogy egy függvény eljárás paramétere lehessen?
12. Mit jelent a **forward** kulcsszó és mikor kell használni?
13. Mit jelent a rekurzió?
14. Hogyan készítünk rekurzív alprogramot?
15. Mikor érdemes rekurziót használni?
16. Mit jelent a változók érvényességi köre?
17. Mit jelent a globlis és lokális változó fogalma?
18. Hogyan tárolja a Pascal a sztringet a memóriában?
19. Mit jelent a sztring hossza és hogyan lehet lekérdezni?
20. Milyen függvények és eljárások szolgálnak a sztringek kezelésére?

Feladatok:

1. Készítsünk eljárást téglalap kerületének és területének kiszámítására. Oldjuk meg a feladatot függvénnyel is és főprogramban különféle paraméterekkel aktíváljuk azt.
   (TEGLALAP.PAS)

2. Készítsük programot a Pithagoras tétel alkalmazására. A derékszögű háromszög befogójának ill. átfogójának számítására.
   (PITAG1.PAS, PITAG2.PAS)

3. Irjon programot a Celsius és a Fahrenheit fokok oda-/visszaalakítására.
   (FC_CONV.PAS)

4. Írjon programot, amely az alábbi n tagú sor összegét számítja ki:
   h= 1+ 1/2 + 1/3 + ... + 1/n
   (H.PAS)

5. Írjon eljárást, amely egy adott mondatot szavaira bont és a szavakat sztring tömbben adja vissza.
   (MONDAT.PAS)

6. Írjon eljárást, amely egy nxn-es tömb elemit a főátlóra tükrözi (transzponálja), illetve egy másikat, amely az eredmény megjeleníti.
   (MATRIX.PAS)

7. Írjon függvényt, amely adott karakterből adott hosszúságú sztringet állít elő.
   (REPCH.PAS)

8. Írjon rekurzív függvényt a legnagyobb közös osztó meghatározására.
   (LKOPRO.PAS)

9. Írjon eljárást, amely az adott számot törzstényezőire bontja.
   (TORZS.PAS)

10. Olvasson be egy n elemű tömböt. Írjon eljárást és függvényt, amellyel kiszámítja az n elemű tömb átlagát.
    (ATLAG.PAS)

11. Írjon rekurzív és nem rekurzív eljárást, amely 1-től n-ig kiszámítja az egész számok összegét.
    (SUM.PAS)

12. A 14. fejezet áttanulmányozása után írja meg a Hanoi tornyai probléma grafikus megoldását.
    (GRHANOI.PAS)

10. MODULOK HASZNÁLATA A TURBO PASCAL-BAN

A szabványos Pascal nyelv semmilyen lehetőséget nem tartalmaz arra, hogy nagyobb program esetén a program egyes részeit külön lefordítva állítsuk elő a futtatható kódot. Sőt az objektumok felhasználás előtti kötelező deklarációja miatt, programkönyvtárat csak forrás nyelven lehet létrehozni, és azt a programba beépíteni (lásd a {$I file-név} direktiva használatát).

A Turbo Pascal nyelvet azonban ellátták a MODULA-2 és az ADA nyelvekből ismert modulkezelési mechanizmussal. A modul - önállóan lefordítható programegység (**unit**), amely jól definiált kapcsolódási felületen (**interface**) keresztül kapcsolódik a program más részeihez. Alapvető elv a moduláris programozás során, hogy a modulok belső része (**implementation**) rejtett a külvilág számára. A Turbo Pascal modulokat **unit**-oknak nevezzük.

A modulok bevezetésével lehetővé vált programkönyvtárak kidolgozása, és a moduláris programozás megvalósítása Turbo Pascal nyelven. Nagy méretű programok fejlesztése során a modulok használata feltétlenül szükséges, hisz a fordító a főprogram kódméretét 64 Kbyte-ban (egy szegmens) korlátozza. A **unit**-okra nézve ez a korlátozás szintén érvényes, azonban a főprogram és a modulok együttes kódméretének nagysága csak a memória méretétől függ.

10.1. A Turbo Pascal modulok felépítése

A **unit**-ok felépítése jellegében emlékeztet a főprogram felépítésére, azonban jelentősen el is tér attól:

```
unit modulnév;            { a modul fejléce, nem hagyható el }

interface                 { a kapcsolódási felület rész }
  uses  ...
  const ...
  type  ...
  var   ...
  eljárások és függvények fejlécei
```

implementation { *az implementációs rész* }
uses ...

label ...
const ...
type ...
var ...
eljárások és függvények teljes deklarációja

[[**begin** { *az inicializációs rész* }
Pascal utasítások]]

end.

A **unit**-ok három jól elkülöníthető részből állnak, amelyek közül az inicializációs rész elhagyható. A továbbiakban részletesen ismertetjük az egyes részek használatával kapcsolatos szabályokat és lehetőségeket.

10.1.1. A modulok fejléce - a modulok közötti kapcsolat

A modulok fejléce a **unit** foglalt szóból és az azt követő modulnévből áll - sohasem hagyható el. A modulnévnek (ellentétben a programnévvel) meg kell egyeznie a modult tartalmazó file nevével. Például a

```
unit Global;
```

modult a GLOBAL.PAS file-ban kell elhelyezni.

A modul nevét használjuk más modulokkal illetve a főprogrammal való kapcsolat kialakítására:

uses *modulok listája*;

ahol a **uses** (használja) foglalt szót a felhasználni kívánt modulok neveinek vesszővel tagolt listája követi.

Például:

```
uses CRT, Graph, Global;
```

Az **interface** részben deklarált Pascal objektumokra a főprogramból megismert módon hivatkozhatunk. Ha a felhasznált modulok közül többen azonos névvel fordul elő valamely eljárás vagy függvény, akkor a hivatkozást egyértelművé tehetjük a modulnév megadásával. Például a *Line* eljárás, amely a Graph **unit**-ban található, az alábbi alakban is hívható:

```
Graph.Line(10,10,200,200);
```

10.1.2. Az interface rész

A kapcsolódási felület részt az **interface** foglalt szó nyitja meg. Ebben a részben található a **unit** globális objektumainak (típusok, konstansok, változók és alprogramok) deklarációja, amelyek más modulokból is elérhetők. A globális eljárásoknak és függvényeknek csak a fejléce adható meg az **interface** részen belül:

```
unit Cmplx;
 interface
 type
   complex=record
             re, im : real;
           end;

 Procedure AddC(x,y : complex; var z : complex);
 Procedure MulC(x,y : complex; var z : complex);
 Procedure WriteC(x:complex);
```

Ha ezek után a főprogramban megadjuk a

```
uses Cmplx;
```

programsort, akkor a főprogram számára elérhetővé válik a *complex* típus és az *AddC* , a *MulC* a *WriteC* eljárás.

Meg kell jegyeznünk, hogy az **interface** részben definiált típusos konstansokat és változókat szintén a program adatszegmensében tárolja a fordító (ennek maximális hossza 65520 byte). A **const, type, var, procedure** és **function** deklarációk száma és sorrendje tetszőleges, azonban **forward** deklaració használata nem megengedett. Az **interface** részben szintén hivatkozhatunk más modulokra (**uses**), abban az esetben, ha az ott definiált típusokat kivánjuk felhasználni.

10.1.3. Az implementation rész

A modul implementációs része egyrészt az **interface** részben megadott globális eljárások és függvények teljes definícióját, másrészt pedig a **unit** lokális deklarációit (címke, konstans, típus, változó, eljárás és függvény) tartalmazza. A globális alprogramok fejlécét az **implementation** részben kétféleképpen is megadhatjuk:

- vagy használjuk az **interface** részben megadott teljes formát,

```
Procedure AddC(x,y : complex; var z : complex);
      begin
      end;
```

- vagy a formális paraméterek (és a függvények esetén a visszatérési érték típusának jelölése) nélkül adjuk meg a fejlécet.

```
Procedure AddC;
```

Egészítsük ki az előző részben közölt **interface** részt teljes modullá:

```
unit Cmplx;

 interface
 type
   complex=record
             re, im : real;
           end;

 Procedure AddC(x,y : complex; var z : complex);
 Procedure MulC(x,y : complex; var z : complex);
```

```
        Procedure WriteC(x:complex);

        implementation
        Procedure AddC;
        begin
           z.re :=  x.re + y.re;
           z.im :=  x.im + y.im;
        end;

        Procedure MulC;
        begin
           z.re := x.re * y.re - x.im * y.im;
           z.im := x.re * y.im + x.im * y.re;
        end;

        Procedure WriteC;
        begin
          write(x.re:3:1,'+',x.im:3:1,'i');
        end;
        end.
```

A fenti programot a CMPLX.PAS file-ban tárolva, a komplex számokkal való összeadás (*Addc*) , szorzás (*Mulc*) elvégzésére valamint a komplex szám kiírására alkalmas **unit**-ot kapunk. A modul felhasználását az alábbi főprogram mutatja be (COMPLEX.PAS):

```
    program Komplex_pelda;

        uses cmplx;

        const c:complex=(re:1;im:2);

        var x,y,z:complex;

        begin

          x:=c;
          y:=c;
          Addc(x,y,z);
          writeln;
          Writec(x);
          write('  +  ');
          WriteC(y);
          write('  =  ');
          WriteC(z);
```

```
  Mulc(x,y,z);
  writeln;
  Writec(x);
  write(' * ');
  WriteC(y);
  write(' = ');
  WriteC(z);
end.
```

Az implementációs részben deklarált típusos konstansok és az alprogramokon kívül definiált változók szintén a program adatszegmensében helyezkednek el, habár elérhetőségük az implementációs részre korlátozódik.

10.1.4. A inicializációs rész

Az inicializációs rész a modul működéséhez szükséges tevékenységek elvégzésére szolgál (például kezdőértékek beállítása). A program futtatása során a **unit** inicializációs része egyetlen egyszer hajtódik végre, mégpedig ott, ahol a **uses** utasításban a modul neve először előfordul. Ez a rész el is hagyható, ekkor az implementációs részt zárja az **end**. utasítás. Nézzünk egy példát, amelyben az inicializációs részben egy szöveg file-t nyitunk meg:

```
Unit FileText;

Interface
Procedure Print(s:string);
Procedure Endprint;

Implementation
var
     f      : text;
const
     fname  = 'out.txt';
Procedure Print;
begin
  writeln(f,s);
end;
Procedure Endprint;
begin
  flush(f);
  close(f);
end;
```

```
{ -- Az inicializációs rész kezdete -- }
begin
  assign(f,fname);
  rewrite(f);
{ -- Az inicializációs rész vége -- }
end.
```

10.2. A modulok használatát bemutató példaprogram

Hozzunk létre egy olyan **unit**-ot, amely segíti a szöveges képernyőn az ablaktechnika megvalósítását. A KERETU.PAS file-ban található modul globális eljárásai adott terület törlését illetve keretezését végzi el:

```
unit keretu;
{ a KERETU.PAS file-ban kell megírni! }

{-------------------- interface -----------------}
interface
   var keretszin, torolszin: byte;

   { Adott képernyõterület keretezése }
   procedure keret(x1,y1,x2,y2,tipus,kszin:byte);

   { Adott képernyõterület törlése }
   procedure torol(x1,y1,x2,y2,szin:byte);

{------------------ implementation---------------}
implementation
uses crt;
function ellenor(a1,a2,b1,b2:byte):boolean;
{ A helyes területkijelölés ellenõrzése
  - (a1,a2) : a bal felsõ sarok koordinátái
  - (b1,b2) : a jobb alsó sarok koordinátái
}
var ok:boolean;
begin
  ok:=true;
  if (a1>=b1) or (a2>=b2) then ok:=false;
  if not((a1 in [1..80])
     and (a2 in [1..25])) then ok:=false;
  if (b1<1)  or (b1>80)
     and (b2<1) or (b2>25) then ok:=false;
  ellenor:=ok;
end;
```

```
procedure torol(x1,y1,x2,y2,szin:byte);
{
  - (x1,y1) : a bal felsõ sarok koordinátái
  - (x2,y2) : a jobb alsó sarok koordinátái
  - szin    : a terület színe
}
var sor:string[80];
    i:integer;
begin
  if ellenor(x1,y1,x2,y2) then
     begin
       window(x1,y1,x2,y2);
       textbackground(szin);
       clrscr;
       window(1,1,80,25);
    end;
end;

procedure keret(x1,y1,x2,y2,tipus,kszin:byte);
{
  - (x1,y1) : a bal felsõ sarok koordinátái
  - (x2,y2) : a jobb alsó sarok koordinátái
  - tipus   : vonaltípus (1 - egyszeres,
                          2 - dupla ,
                          3 vastag)
  - kszin   : a keret színe
}
var i:byte;
    r:string[9];
begin
  if ellenor(x1,y1,x2,y2) then
     begin
       case tipus of
          1: r:=#218+#191+#192+#217+#196+#179+#196;
          2: r:=#201+#187+#200+#188+#205+#186+#205;
          3: r:=#219+#219+#219+#219+#223+#219+#220;
       end;
       textcolor(kszin);
       gotoxy(x1,y1); write(r[1]);
       for i:=x1 to x2-2 do  write(r[5]);
       write(r[2]);
       for i:=1 to ((y2-1)-y1) do
         begin
           gotoxy(x1,y1+i); write(r[6]);
           gotoxy(x2,y1+i); write(r[6]);
         end;
       gotoxy(x1,y2); write(r[3]);
```

```
            for i:=x1 to (x2-2) do  write(r[7]);
            write(r[4]);
          end;
      end;
      {----------------- unit init ---------------}
      begin
        keretszin:=15; { feher }
        torolszin:=0 ; { fekete }
        textbackground(torolszin);
        clrscr;
        textcolor(keretszin);
      end.
```

A **unit** működését bemutató főprogram különböző színű területeket keretez be, különböző kerettípus felhasználásával.

A főprogramot a **KERETPLD.PAS** file tartalmazza:

```
program keretezes_pelda;
 { a KERETPLD.PAS file tartalmazza ! }

 uses crt,keretu;
 var i:integer;

 begin

 torol(1,1,80,25,4);
 keret(2,1,79,25,1,keretszin);

 for i:=1 to 3 do
 begin
   torol(10+i*2,1+i*2,70-i*2,24-i*3,i);
   keret(10+i*2,1+i*2,70-i*2,24-i*3,i,15-i);
 end;
 end.
```

10.3. Szabványos modulok

A Turbo Pascal rendszer nyolc szabványos modullal rendelkezik, amelyek sok hasznos típust, konstanst, eljárást és függvény tartalmaznak. Ezek a modulok: SYSTEM, DOS, CRT, PRINTER, GRAPH, OVERLAY, TURBO3 és a GRAPH3. A GRAPH, TURBO3 és a GRAPH3 modulok különálló file-okban helyezkednek el (.TPU), míg a többi modult a TURBO.TPL (Turbo Pascal

Library) könyvtár tartalmazza. A SYSTEM **unit** automatikusan hozzákapcsolódik minden Turbo Pascal programhoz, míg minden más **unit** eléréséhez a **uses** hivatkozás használata szükséges.

Az alábbiakban röviden jellemezzük a szabványos modulokat:

A **CRT** modul elsősorban a képernyő szöveges üzemmódját támogató eljárásokat és függvényeket tartalmaz. Ezek segítségével lehetőség van a kurzor pozícionálására, a használt színek beállítására és ablakok létrehozására. Ezen kívül a modul tartalmaz néhány eljárást, amelyek segítségével a billentyűzet speciális módon érhető el, és lehetőség adódik hanggenerálásra.

A **DOS** modulban azok az eljárások és függvények vannak összegyűjtve, amelyek segítségével hatékonyabban kihasználhatjuk az **MS-DOS** operációs rendszer lehetőségeit.

A **GRAPH unit** a képernyő grafikus üzemmódját támogató típusok, konstansok eljárások és függvények gazdag készletét tartalmazza. Ezek segítségével tetszőleges grafikus kép előállítható illetve a programozható karakterkészlet felhasználásával szövegek is megjeleníthetők. A **GRAPH unit** rutinjai a megfelelő BGI-vezérlő felhasználásával tetszőleges grafikus kártya esetén használhatók. A Turbo Pascal rendszer a CGA, a Hercules, az EGA, a VGA, az MCGA, az IBM8514, az ATT400 és a PC3270 grafikus adapterekhez tartalmaz BGI vezérlőt. Más kártya esetén a vezérlőt külön kell beszerezni.

Az **OVERLAY** modul használata akkor szükséges, amikor nagy programok fejlesztése esetén az átlapolási (**overlay**) technika használata válik szükségessé. Ezzel a technikával a rendelkezére álló szabad memória méretét (550-620 Kbyte) meghaladó méretű programok is előállíthatók.

A **PRINTER** modul leegyszerűsíti a szöveges információk nyomtatását. A modulban definiált a TEXT típusú LST file, amely a PRN logikai egységgel áll kapcsolatban. Az eredménynek a nyomtatón való megjelenítése ezáltal egyszerűen, az alábbi példában bemutatott módon végezhető el:

```
uses printer;
begin
  writeln(LST,'Turbo Pascal 5.0, 5.5, 6.0');
end.
```

A **SYSTEM unit** tartalmazza a szabványos Pascal nyelv eljárásait és függvényeit illetve azokat a Turbo Pascal rutinokat (***inc***, ***dec***, ***getdir*** ...), amelyek nem kerültek más modulba. Mint ahogy azt már említettük, a SYSTEM modul minden program fordításakor felhasználásra kerül függetlenül attól, hogy a **uses** utasításban hivatkoztunk rá, vagy sem.

A **TURBO3** és a **GRAPH3** modulokat a Turbo Pascal 3.0-ás verziójával való forrásnyelvű kompatibilitás miatt tartalmazzák az újabb verziójú rendszerek. Az itt definiált rutinok használata nem ajánlott, hiszen az új könyvtárakban bőséges lehetőség kinálkozik minden tevékenység elvégzésére.

(A Turbo Pascal 6.0 a Turbo Vision lehetőségek használatához még további szabványos modulokat tartalmaz.)

Ellenőrző kérdések:

1. Hasonlítsa össze a Turbo Pascal főprogramjának és **unit**-jainak szerkezetét !
2. Ismertesse a modulok felépítését!
3. Milyen deklarációkat lehet elhelyezni az **interface** és az **implementation** részekben?
4. Hogyan lehet a programokból modulokra hivatkozni?
5. Melyek a Turbo Pascal szabványos moduljai?
6. Milyen elvek érvényesülnek a moduláris programozás során?
7. Hogyan lehet a Pascal program kódja nagyobb, mint 64 Kbyte?
8. Mit tartalmaz a TURBO.TPL file?
9. Mely szabványos **unit**-ok helyezkednek el különálló file-ban?

Feladatok:

1. A lemezen tallálható PASPROG.PAS file térbeli vektorok kezelését mutatja be a programok felhasználásával. Bontsa szét ezt a programot PUNIT.PAS és PPROG.PAS modulokra.

2. A kijelölt helyeken egészítse ki az alábbi programot, majd írjon hozzá működését bemutató főprogramot!

```
____ peldau;
____________
  const
          n=10;
  _______
          arrt=array[-n..n] of char;
  var
          ac: arrt;

  function makesrt(a:arrt; n:integer):string;

________________

  var
          i:integer;
  __________    _________

        var
          k:integer;
          s:string;
        begin
          s:='';
          ___ k:=-n to n do  s:=s+a[k];
          makestr:=s;
        end;
  ___________

     for i:=-n __  n do  ac[i]:=#32;
  ___.
```

11. FILE-KEZELÉS

A Pascal programunkban használt adatok (változók, típusos konstansok) a számítógép memóriájában helyezkednek el. A memória mérete azonban erősen korlátozott, nem beszélve arról, hogy a számítógép kikapcsolása után az adatok egyszerűen elvesznek. Hogyan lehet a memória méretét jóval meghaladó mennyiségű adatot kezelni, úgy hogy azok a számítógép kikapcsolása után is megmaradjanak? A választ a file (adatállomány) bevezetése adja. A file kifejezés rendszerint valamely háttértárolón (merevlemez, mágnesszalag) tárolt adatok összességét jelöli.

A Turbo Pascal-ban a file fogalma valamelyest eltér a szabványos Pascal-ban használttól:

Eszköz (*device*) file-ok:
a számítógépben található I/O eszközök (mint például a billentyűzet, a képernyő, a nyomtatók és a kommunikációs csatornák közvetlen elérését teszik lehetővé.

Lemez (*disk*) file-ok:
amelyek a számítógép lemezegységein tárolt egymással összefüggő adatok együttesét jelölik. A lemez file-kat, tartalmuk alapján szöveges (*text*), típusos és típus nélküli file-ok csoportjára osztjuk.

11.1. Turbo Pascal szöveges file-ok

A szöveges file olyan sorokból épül fel, amelyek karaktereket tartalmaznak, és a sort a kocsivissza/soremelés (CR/LF) vezérlőkarakterek zárják. A CR/LF karakterek (ASCII kódjuk 13 és 10), a sorok végét jelzik a file-ban. A szöveges file-okat tetszőleges szövegszerkesztővel létrehozhatjuk illetve módosíthatjuk.

A 11.1. ábra a szöveges file-ok szerkezetét szemlélteti.

1. sor:	Ez egy példa a text file sorának felépítésére	CR	LF
2. sor:	Mindegyik sort a CR/LF zárja	CR	LF
3. sor:	A következő sor egy üres sor	CR	LF
4. sor:	CR	LF	
5. sor:	A sorok számokat is tartalmazhatnak:	CR	LF
6. sor:	50 3.1415926 1.45E-2	CR	LF

11.1. ábra
Példa a szöveges file-ok szerkezetére

A TEXT file-ok az elérésük és használatuk szempontjából a szekvenciális file-okhoz tartoznak. Az egyes sorokat mindig a legelső sortól kezdve egyesével olvashatjuk illetve írhatjuk. Kivételt képez a hozzáírás művelete, mivel ekkor az írás a már meglévő file utolsó sora után kezdődik. A szöveges file-t vagy csak írhatjuk, vagy csak olvashatjuk, tehát a file használata előtt el kell döntenünk, hogy mit szeretnénk csinálni.

A file-kezelés lépései programozási nyelvtől függetlenek. A file-t a felhasználása előtt meg kell "nyitni", mivel csak így férhetünk hozzá a benne tárolt adatokhoz. A megnyitás után következik a file elérése - adatok írása a file-ba vagy adatok olvasása a file-ból. Munkánk végeztével a file-t le kell "zárni", hogy annak adatai háttértáron megőrzésre kerüljenek. A 11.2 ábrán egy olyan Turbo Pascal programot látható, amely a file-kezeléshez szükséges lépéseket tartalmazza szöveges file esetén.

```
PROGRAM text_file_kezeles;
TYPE
   ptype = array[1..4096] of char;
VAR
   F     : text;          {a file tipusa TEXT,  a file-változó F}
   puffer: ptype;         {4 Kbyte-os file-puffer}
   v1, v2: integer;

BEGIN
   ASSIGN(F,'szov.txt');{a Pascal file-változó és a DOS file}
                         {összerendelése}

   SETTEXTBUF(F,puffer);{I/O puffer hozzárendelése
                          a file-hoz}

   RESET(F);        { olvasás }
   REWRITE(F);      { a file megnyitása  -> létrehozás és írás}
   APPEND(F);       { hozzáírás }

   { A file-kezelés lépései }

   { Ha reset-tel nyitottuk meg a file-t: }

     READ(F, v1,v2);       { olvasás a file-ból }
     READLN(F,v1,v2);      { teljes sor olvasása a file-ból }

     IF EOF(F)  then       { a file végének tesztelése };

     IF EOLN(F) then       { a sor végének tesztelése  };

     IF SEEKEOF(F)  then { a file végének tesztelése az
                           elválasztó karakterek átlépésével };

     IF SEEKEOLN(F) then { a sor végének tesztelése az
                           elválasztó karakterek átlépésével };

   { Ha rewrite-tal vagy append-del nyitottuk meg a file-t: }

     WRITE(F,v1,v2+26);  { írás a file-ba }
     WRITELN(F,v1,v2+26);{ teljes sor írása a file-ba }
     FLUSH(F);            { az output puffer kiírása a file-ba}

   { Bármely megnyitási mód esetén a file lezárása:}
   CLOSE(F);
END.
```

11.2. ábra
Példa a szöveges file-t feldolgozó Pascal program szerkezetére

11.1.1. A szöveg file azonosítása

Ahhoz, hogy a programunkból szöveg file-t használjunk, deklarálnunk kell egy ún. file-változót, amely egyértelműen azonosítja a file-t a programon belül. A deklarációban a változót **TEXT** típusúnak kell megadni.

var
file_valtozo:**text**;

A következő lépésben meg kell mondanunk, hogy az adott file-változóhoz melyik külső (operációs rendszer szintjén található) file-t rendeljük hozzá. Ezt az összerendelést mindig a file megnyitása előtt kell elvégeznünk:

Assing (*file_valtozo*, *file_nev*) ;

A *file_nev* paraméter sztring formájában tartalmazza a felhasználni kívánt file nevét (maximálisan 79 karakter hosszon). Nézzünk néhány lehetséges példát a file nevének megadására:

'C:\PASCAL\PELDA.TXT' - megadás a teljes elérési útvonal kijelölésével.

'SZOVEG.SZG' - a file az aktuális könyvtárban található.

Az ***Assign*** rutin paraméterezésének további lehetőségeivel a fejezet végén ismerkedünk meg.

11.1.2. A szöveg file megnyitása

Mint ahogy az a 11.2. ábrán is látható, a **TEXT** típusú file-okat háromféleképpen lehet megnyitni.

A

reset *(file_valtozo)* ;

eljárás létező szöveg file-t nyit meg, csak olvasásra. A megnyitás után az *eof* függvény *true* értékkel jelzi, ha a file üres. A ***reset*** a file-ban az aktuális pozíciót a file elejére állítja (ez a művelet már megnyitott file-on is elvégezhető.).

A

rewrite *(file_valtozo)* ;

eljárás új szöveg file-t hoz létre, vagy már meglévő szöveg file-t újraír (így annak teljes tartalma elvész!). Az így megnyitott file-okon csak az írás művelete használható (csak írható file). A ***rewrite*** az aktuális pozíciót a file elejére állítja.

Az

append *(file_valtozo)* ;

eljárás létező szöveg file-t nyit meg, csak hozzáírásra. Az ***append*** a file-ban az aktuális pozíciót a file végére állítja.

Mind a három megnyitási mód a gyorsabb file-kezelés érdekében egy-egy 128 byte-os pufferterületet rendel a file-hoz. Ha a file-műveleteket tovább szeretnénk gyorsítani, akkor ezt a területet átdefiniálhatjuk saját nagyobb pufferre. Az átdefiniálást a ***settextbuf*** eljárás hívásával a file megnyitása előtt kell elvégezni. (Ajánlott a pufferméretet 2 hatványának választani: 256, 512, 1024 ,....)

A ***settextbuf*** eljárás kétféleképpen paraméterezhető:

settextbuf*(file_valtozo,puffer);*

vagy

settextbuf*(file_valtozo,puffer, puffermeret);*

ahol a *puffer* általában valamilyen tömbváltozó, és a *pufferméret* a *puffer* mérete (*sizeof*(*puffer*)), vagy annál kisebb szám. Az első esetben a rendszer állítja be a puffer méretét, a második esetben pedig a híváskor adjuk meg.

Ha a file nyitásakor valamilyen hiba lép fel (például nem létező file-t nyitunk meg olvasásra), akkor a programunk '*Runtime error..*' üzenettel leáll. A Turbo Pascal rendszer lehetővé teszi hogy az input/output (I/O) műveleteknél fellépő hibákat programon belül dolgozzuk fel. A kritikus programrészletet **{$I-}** és **{$I+}** direktívák közé helyezve, az ***ioresult*** függvény visszaadott értékéből következtethetünk a hiba megjelenésére.

Az alábbi kis program felhasználva az elmondottakat, bekér egy file-nevet, és ellenőrzi, hogy az adott file létezik, vagy sem.

```
program io_check;

var f  : text;
    fn : string[80];

begin
    write('Kerem a file nevét: ');
    readln(fn);

    assign(f,fn);
    {$I-}
    reset(f);
    if ioresult<>0 then
                    begin
                     writeln('A file nem elérhetõ!');
                     halt(1);
                    end;
    {$I+}
    writeln('A file létezik.');
    close(f);
end.
```

Felhívjuk a figyelmet arra, hogy az ***ioresult*** függvény értékéből mindig csak a legutolsó I/O művelet lefolyására következhetünk. A függvény hívása törli a belső állapotjelző értékét, így ha többször szeretnénk ezt lekérdezni, saját egész típusú változóban kell eltárolnunk az első hívásakor visszaadott értéket. Az ***ioresult*** 0 értékkel tér vissza, ha a legutolsó I/O művelet sikeres volt, és valamilyen *run-time* hiba kódjával, ha nem.

11.1.3. A szöveg file I/O műveletei

Ha a szöveg file-t írásra nyitottuk meg (***rewrite/append***), akkor az adatok file-ba írását a már jól ismert ***write*** és ***writeln*** eljárások módosított változataival oldhatjuk meg. Minkét eljárásnál az első paraméterként a file-változót kell megadnunk:

write(*file_valtozo,kifejezéslista*);

illetve

writeln(*file_valtozo,kifejezéslista*);

A különbség a két eljárás között, hogy a ***writeln*** eljárás az írási művelet végeztével sorvége (CR/LF) jelet is ír a file-ba.

Ha a **Text** típusú file-t olvasásra nyitottuk meg (***reset***), akkor szintén ismert eljárásokat használhatunk az olvasási művelet elvégzésére:

read(*file_valtozo, változólista*);

illetve

readln(*file_valtozo, változólista*);

A ***readln*** eljárás ellentétben a ***read*** eljárással, a szöveg file-ból mindig teljes sort dolgoz fel, így az aktuális pozició a file-ban az olvasás után a következő sor eleje lesz.

A ***write***, ***writeln***, ***read*** és ***readln*** eljárások használatára vonatkozó szabályok (például formátum használata, milyen típusú adat írható olvasható segítségükkel,...) megegyeznek a 7. fejezetben ismertetettel.

Ha a file-t olvasásra nyitottuk meg lehetőségünk van bizonyos ellenőrzések beépítésére, amelyek szükségesek ahhoz, hogy a programunk helyesen dolgozza fel a szöveg file-t.

a. A file vége elérésének ellenőrzése:

Az *eof(file_valtozo)* függvény **boolean** típusú értéket ad vissza, amely jelzi, hogy az aktuális file-pozíció a file utólsó eleme után helyezkedik el (*true*), vagy sem *(false)*. Szöveg file-ok esetén a file-vége esemény két esetben is bekövetkezhet:
- a file-ban ún. *end-of-file* jelzőt találtunk (ASCII 26), vagy
- elértük a file fizikai végét.

A *seekeof(file_valtozo)* függvény hasonlóan az *eof* függvényhez a file végét jelzi *true* értékkel, abban az esetben is, ha az aktuális file-pozíció és a file-vége között csak szóköz, tabulátor vagy sorvége karakterek helyezkednek el.

b. A sorok végének érzékelése

A sorvége elérésének ellenőrzésére szintén két függvény használható. Az *eoln(file_valtozo)* függvény true értékkel tér vissza, ha az aktuális file-pozícióban ún. *end-of-line* (sorvége) jelző (CR/LF) áll. A *seekeoln(file_valtozo)* függvény szintén jelzi a sor végét, méghozzá akkor is, ha az aktuális file-pozíció és a sor vége között csak szóköz és tabulátor karakterek helyezkednek el.

A függvények ismeretében a file-olvasó programunk fő ciklusa az alábbiak szerint alakítható ki:

```
while not eof(f) do
  begin
    readln(f,sor);
    .....
  end;
```

Ha a file biztosan tartalmaz sorokat (nem üres), akkor használhatjuk a **repeat until** ciklust is:

```
repeat
  readln(f,sor);
  ...
until eof(f);
```

11.1.4. Szöveg file lezárása

A **text** típusú file lezárása a

close(*file_valtozo*);

eljárás hívásával történik, függetlenül attól, hogy milyen módon nyitottuk meg a file-t.

Bizonyos esetekben, mint például ha a ***write*** / ***writeln*** eljárások helyett saját rutinokkal írunk a file-ba, külön kell gondoskodnunk a output puffer file-ba írásáról. Erre a célra használható a

flush(*file_változo*);

eljárás. Meg kell jegyeznünk, hogy a ***write***, ***writeln*** és a ***close*** eljárások automatikusan ürítik az output puffert.

Az fenti lépések bemutatására tekintsük a szöveg file tartalmának képernyőn való megjelenítését végző programot.

```
program type0;
     var
         tf      : text;
         fn,sor  : string;
         puffer  : array[1..5120] of char;   { 5K puffer }

     begin
         write('Kérem a file nevét: ');
         readln(fn);
         assign(tf,fn);
         settextbuf(tf,puffer);

         {$I-}
         reset(tf);
         if ioresult<>0 then
                     begin
                 writeln('A(z)',fn,
                           'nevû file nem olvasható!');
                    exit;
                end;
         {$I+}
```

```
        while not eof(tf) do
          begin
            readln(tf,sor);
            writeln(sor);
          end;
        close(tf);
    end.
```

11.1.5. A szabványos szöveg file-ok: input és output

A szabványos input és output szöveg file-ok használatát a 7. fejezetben részletesen ismertettük. Az alábbi megjegyzések figyelembevételével ezen file-okat még sokoldalúbban tudjuk felhasználni.

1. A szabványos I/O műveletek esetén az alábbi rövidítések használatosak a Pascal nyelvben:

write(*output*, ...);	***write***(...);
writeln(*output*, ...);	***writeln***(...);
read(*input*, ...);	***read***(...);
readln(*input*, ...);	***readln***(...);
eof(*input*)	***eof***
eoln(*input*)	***eoln***

2. A szabványos bevitelre szintén használható az I/O hibaellenőrzés. Az alábbi példában csak numerikus input-ot fogadunk el:

```
program num_inp;
    var
        x:real;
    begin
        repeat
           writeln('Kérek egy számot: ');
           {$i-}
           readln(x)
           {$i+}
        until ioresult=0;
        writeln('A beírt szám négyzete: ',sqr(x):10:2);
    end.
```

3. A szabványos input és output átirányítható a program neve után megadott < és > jelekkel.

```
program stdio_1;
      var ch:char;
      begin
        repeat
          read(ch);
          write(ch);
        until eof;
      end.
```

Ha a fenti programot .EXE file-ba fordítjuk, akkor DOS készenléti jelnél (*prompt*) az alábbi műveleteket végezhetjük el:

```
c:\pascal>stdio_1
```

A program a begépelt karaktereket *<enter>* lenyomása után visszaírja a képernyőre. A programból a *CTRL-Z* billentyűkombinációval lehet kilépni.

```
c:\pascal>stdio_1 <proba.txt
```

A program a proba.txt file tartalmát jeleníti meg a képernyőn.

```
c:\pascal>stdio_1 >proba.txt
```

A program a begépelt karaktereket *<enter>* lenyomása után beírja a proba.txt file-ba. A programból a *CTRL-Z* billentyűkombinációval lehet kilépni.

```
c:\pascal>stdio_1 <proba.txt >proba2.txt
```

A program a proba.txt file-t átmásolja a proba2.txt file-ba.

4. Ha a **CRT** unit-ot használjuk, akkor a szabványos I/O műveletek nem a DOS felhasználásával mennek végbe, ily módon az átírányítás sem működik. Ebben az esetben az input és output file-ok alábbi formában történő újranyitásával az átírányítás biztosítható:

```
        assign(output,'');
        assign(input,'');
        rewrite(output);
        reset(input);
```

A 3. pontban közölt program CRT-s változata:

```
program stdio_2;
        uses crt;
        var ch:char;
        begin
           assign(input,'');
           assign(output,'');
           reset(input);
           rewrite(output);
           repeat
             read(ch);
             write(ch);
           until eof;
           repeat
           until keypressed;
        end.
```

A következő példa azt az esetet mutatja be, amikor a szabványos input és output műveletek a CRT unit felhasználásával mennek végbe (nem irányítható át). Lehetőség van azonban arra, hagy saját text típusú file-okhoz rendeljük a DOS szabványos be- és kimenetét, az *assign* eljárásban megadott '' file-névvel. A program a megadott szöveg file-t 23 soronként írja ki.

```
program stdio_3;
      uses crt;
      const
          max=23;
      var
          sor    : string[80];
          cnt    : integer;
          ch     : char;
          inp,out: text;

      begin
         assign(inp,'');
         assign(out,'');
         reset(inp);
         rewrite(out);
```

```
        cnt:=0;
        repeat
          readln(inp,sor);
          inc(cnt);
          writeln(out,sor);
          if cnt mod max=0 then
                     begin
                       writeln('Tovább bármely
                                billentyûvel...');
                       repeat until keypressed;
                       ch:=readkey;
                     end;
        until eof(inp);
     end.
```

5. Érdekes lehetőséget biztosít a **CRT** unit-ban definiált ***AssignCrt*** eljárás használata. Az eljárás paraméterként megadott szöveg file-t a konzolhoz rendeli. ***Reset***-tel megnyitva közvetlenül a billentyűzetről tudunk olvasni adatokat, ***rewrite***-val megnyitva pedig közvetlenül a képernyőre írhatunk.

   Az stdio_3 programot alakítsuk át az ***AssignCrt*** rutin felhasználásával:

```
     program stdio_4;
     uses crt;
     const
         max=23;
     var
         sor   : string[80];
         cnt   : integer;
         ch    : char;
         crtout: text;

     begin
        assign(input,'');
        assign(output,'');
        assigncrt(crtout);
        reset(input);
        rewrite(output);
        rewrite(crtout);
        cnt:=0;
        repeat
          readln(sor);
          inc(cnt);
          writeln(sor);
          if cnt mod max=0 then
```

```
                    begin
                     writeln(crtout,'Tovabb barmely
                        billentyvel...');
                     repeat until keypressed;
                     ch:=readkey;
                    end;
      until eof;
   end.
```

11.1.6. Példaprogramok text típusú file-ok használatára

a. A app_txt program a file1.txt file tartalmát hozzámásolja a file2.txt file-hoz. Külön felhívnánk a figyelmet, hogy a programban minden I/O művelet eredményét ellenőriztük.

```
program app_txt;

    type
       Tpuffer       = array[1..4*1024] of char;

    var inf,outf     : text;
       ok            : boolean;
       sor           : string[80];
       inp_puffer,
       out_puffer    : tpuffer;

    begin
       {$I-}
       assign(outf,'file2.txt');
       settextbuf(outf,out_puffer);

       append(outf);
       writeln(outf);
       ok:=ioresult=0;
       if ok then  { ha sikeres volt az outf file
                     megnyitása }
             begin
                assign(inf,'file1.txt');
                settextbuf(inf,inp_puffer);
                reset(inf);
                ok:=ioresult=0;
             end;
```

```
          if ok then  { ha sikeres volt az inf file
                   megnyitása }
               begin
             { amíg nics vége az inf file-nak, és sikeres
                          a kiírás az outf file-ba }
                  while (not eof(inf)) and ok do
                      begin
                        readln(inf,sor);
                        ok:=ioresult=0;
                        if ok then  { ha sikeres volt a
                                  sor beolvasása }
                              begin
                                writeln(outf,sor);
                                ok:=ioresult=0;
                              end;
                      end;
               end;
          close(inf);
          close(outf);
          {$I+}
      end.
```

b. Az alábbi program matrix.txt file-ban megadott maximálisan 10x10-es mátrix elemeit olvassa fel. A sorok és az oszlopok számát a program állapítja meg a file felépítéséből.

A matrix.txt file lehetséges felépítése:

```
4  6  17  5
7  8  19  8
8  9  10  9
```

A beolvasást elvégző program:

```
program matrix;
  const
      maxn =10;
  type
      mtype=array[1..maxn,1..maxn] of integer;
  var
       t   : mtype;
       m,n : integer;
       fn  : string;

  {Az mt tömb feltöltése az fnév nevû file-ból.}
```

```
procedure read_matrix(var mt:mtype; fnev:string);
var
     mf  : text;
     i,j : integer;
begin
     assign(mf,fnev);
     {$I-} reset(mf);
     if ioresult<>0 then
                      begin
                        writeln('File-hiba.');
                        halt;
                      end;
     {$i+} i:=0;
     while not seekeof(mf) do
       begin
           j:=0;
           inc(i);
           while not seekeoln(mf) do
              begin
                inc(j);
                read(mf,mt[i,j]);
              end;
           readln(mf);
       end;
     m:=i;  { a sorok száma }
     n:=j;  { az oszlopok száma }
     close(mf);
end;
{ A tömb tartalmának kiírása mátrixos formában a
  képernyõre}
procedure list_matrix(mt:mtype);
var
     i,j : integer;
begin
     for i:=1 to m do
       begin
         for j:=1 to n do
             write(mt[i,j]:5);
         writeln;
       end;
end;
begin
  fn:='matrix.txt';
  read_matrix(t,fn);
  list_matrix(t);
end.
```

c. A Turbo Pascal nyelven megírt programokat lehet a DOS parancssorból is paraméterezni. A ***ParamCount*** word típusú függvény megadja, hogy hány paraméter található a program indítási sorában. Az egyes paramétereket a ***ParamStr*** string típusú függvény szolgáltatja, a paraméter sorszámának megadása után.

Az alábbi program kiírja a saját paramétereit:

```
program param;
var i:integer;
begin
  for i:=0 to paramcount do
     writeln( 'Paraméter ',i,'   ',paramstr( i ) );
end.
```

MS-DOS 3.0 illetve újabb verziók esetén a 0-ás indexű paraméter a program nevét tartalmazza teljes elérési útvonallal.

Az alábbi program parancssorban megadott szöveg file-ban adott sztring előfordulásait keresi meg.

```
program stkereso;

var
    tf   : text;
    sor  : string;
    ssz  : integer;
begin
if paramcount < 2 then
              begin
                writeln;
                writeln('A program használata:');
                writeln('STKERESO file sztring');
                halt;
              end;

assign(tf,paramstr(1));
{$I-}
reset(tf);
if ioresult<>0 then
              begin
                writeln('Hibás file.');
                halt;
              end;
```

```
{$I+}
ssz:=0;
while not eof(tf) do
  begin
    inc(ssz);
    readln(tf,sor);
    if pos(paramstr(2),sor)<>0  then
                                 writeln(ssz,'. sorban');
  end;

close(tf);
end.
```

11.2. Típusos file-ok

A típusos file-ok olyan adathalmazok, amelyek jól definiált, azonos típusú összetevőkre, elemekre bonthatók. Az elemek típusa a file és objektum típus kivételével tetszőleges lehet. A file-ban tárolt elemek a felírás sorrendjében 0-tól kezdve egyesével sorszámozódnak. A file feldolgozásánál ezen sorszám felhasználásával pozícionálhatunk az állomány valamely elemére. A típusos file-ok szerkezetét a 11.3. ábra szemlélteti

0. elem
1. elem
2. elem
.
.
.
n. elem

11.3. ábra
Példa a típusos file szerkezetére

A file-kezeléses szóhasználatban általában a rekord szót használják a file elemeire. A Pascal nyelvben a rekord szó azonban a felhasználói adattípust (**record**) jelöli.

```
PROGRAM tipusos_file;

 TYPE
      tipus=RECORD                  { A file típusa }
             a:integer;
             s:string[30];
           END;
 VAR
      f   : FILE OF tipus;              {  A file-változó }
      a,b : tipus;
        p : longint;

 BEGIN
     { A Pascal - DOS kapcsolat megadása}
     ASSIGN(f,'file-név');

     { A file megnyitása }

     { Létezõ file megnyitása írásra / olvasásra }
        RESET(f);

          {vagy}

    { Uj file megnyitása írásra / olvasásra  (létrehozás) }
        REWRITE(f);

     { File-mûveletek }

       WRITE(f,a,b);     { Irás a file-ba }
       READ(f,a);        { Olvasás a file-ból }
       SEEK(f,2);        { Pocionálás a file-ban }
        TRUNCATE(f);     { A file csonkítása }

     { A file-lal végzett mûveleteket segítõ
       függvények }

      p:=FILEPOS(f);     {Az aktuális file-pozíció }
      p:=FILESIZE(f);    {A file-ban található elemek száma }
        IF EOF(f) THEN;  {A file-végének érzékelése }

     { A file lezárása }
     CLOSE(f);
 END.
```

11.4. ábra
Példa a típusus file-t feldolgozó
Pascal program szerkezetére

A típusos file-on végzett műveletek esetén, a művelet egysége az elem. A file mérete szintén a benne található elemek számát jelöli. Hogy hány byte-ot foglal el a file a lemezen, azt egyszerűen kiszámíthatjuk:

a file mérete * *sizeof*(*elem*).

A típusos file-ok feldolgozása *szekvenciálisan* illetve - a pozícionálás műveletét használva - *direkt* módon egyaránt lehetséges.

A típusos file-ok kezelése a szöveges file-okhoz hasonlóan végezhető el. Vannak azonban lényeges eltérések, amelyeket a továbbiakban ismertetünk. A típusos file-ok esetén felhasználható Turbo Pascal eljárásokat és függvényeket a 11.4. ábrán látható programban foglaltuk össze.

11.2.1. Típusos file deklarálása és megnyitása

A típusos file változójának deklarálása az alábbi formában történik:

> **var**
> *file_valtozo* : **file of típus**;

ahol a típus tetszőleges Pascal típus lehet, kivéve a file és az object típusokat.

Nézzünk néhány helyes deklarációt különböző típusok felhasználásával:

```
type
    str15  = string[15];
    arr100 = array[1..100] of word;
    telrec = record
                nev : string[30];
                cim : string[40];
                tel : string[12];
              end;
var
  { egész számokat tartalmazó file }
    numfile : file of integer;

  { karaktereket tartalmazó file }
    chrfile : file of char;

  {15 karakter hosszú sztringeket tartalmazó file }
    strfile : file of str15;
```

```
{ 100 elemû szavas tömböket tartalmazó file }
  arrfile : file of arr100;

{ telrec típusú rekordokat tartalmazó file }
  telfile : file of telrec;
```

A programban a file-kezelés első lépéseként a Pascal *file_valtozo*-t, hozzá kell rendelnünk valamely külső (DOS) file-hoz:

assign*(file_valtozo,DOS_File_nev)*;

Ezek után következhet a file megnyitása. Ha létező file-t szeretnénk használni, akkor a megnyitást a ***reset*** eljárás hívásával végezhetjük el:

reset*(file_valtozo)*;

Ha azonban új file-t szeretnénk létrehozni, vagy létező file-t felülírni, akkor a

rewrite*(file_valtozo)*;

eljárást kell meghívnunk.

Mindkét megnyitási mód után, az aktuális file-pozíció a file elejét jelöli. Lényeges eltérés a szöveg file-okhoz képest, hogy megnyitása után a típusos file egyaránt írható és olvasható.

A file-kezelés során fellépő I/O hibák kezelése típusos file esetén is egyszerűen elvégezhető a **{$I-}** és **{$I+}** direktívák felhasználásával:

```
{$I-}
reset(f)
if ioresult<>0 then
                begin
                  writeln('A file nem létezik!');
                  exit;
                end.
{$I+}
{ Létezõ file sikeres megnyitása: }
```

11.2.2. File-műveletek

Megnyitott típusos file esetén az adatok kiírására a ***write***, míg az olvasására a ***read*** eljárásokat használhatjuk.
A

read(*file_valtozo, változólista*);

eljárás az aktuális pozíciótól kezdve olvassa a file elemeit a megadott változókba. A művelet végrehajtása után az aktuális pozíció a utoljára beolvasott elem után helyezkedik el. A file utolsó elemének beolvasása után az

eof(*file_valtozo*)

függvény *true* értékkel jelzi a file-végének elérését.
A

write(*file_valtozo, változólista*);

eljárás az aktuális file-pozíciótól kezdve a változólistában megadott, a file típusával megegyező típusú változók tartalmát a file-ba másolja. Az aktuális file-pozíció az utoljára kiírt elem után helyezkedik el. Ha az írási művelet megkezdésekor a az aktuális pozíció a file végén volt, akkor a ***write*** eljárás a file-t kibővíti a megadott elemekkel.

Ha file-kezelés során nem használjuk a ***seek*** eljárás által biztosított pozícionálási lehetőséget, akkor a ***read*** és ***write*** műveletekkel a file-t szekvenciális módon érjük el. A ***seek*** eljárás segítségével azonban a közvetlenül (*direct , random access*) elérhetjük a file tetszőleges elemét.
A

seek(*file_valtozo, longint_index*);

hívás után a file aktuális pozíciója a *longint_index* paraméterben megadott pozíció lesz. Ha a file vége után pozícionálunk, akkor az *eof* függvény *true* értéket ad vissza. Ilyen pozíció esetén az olvasás művelete *'Disk read error'* hibát eredményez, míg az írás hatására a rendszer a file-t felépíti (kiterjeszti) a megadott pozícióig.

Mielőtt tovább mennénk, ismerkedjünk meg a pozícionálásnál is jól felhasználható ***filesize*** és ***filepos*** függvényekkel.
A

filesize(*file_valtozo*)

függvény **longint** típusú értékben adja meg a file-ban található elemek számát. Ha a file üres, a visszaadott érték 0.
A

filepos(*file_valtozo*)

függvény szintén **longint** típusú értékben az aktuális file-pozíció indexét adja meg. A file-ban az elemek sorszámozása 0-tól kezdődik. A file végének elérésekor a ***filesize*** és a ***filepos*** függvények ugyanazt az értéket szolgáltatják.

Nézzünk néhány különleges pozícionálást a fenti függvények felhasználásával:

Pozícionálás a file utolsó eleme utáni pozícióra:

seek(*file_valtozo*, ***filesize***(*file_valtozo*));

Visszalépés az előző pozícióra:

seek(*file_valtozo*, ***filepos***(*file_valtozo*)-*1*);

A

truncate(*file_valtozo*);

hívás hatására az aktuális file-pozíción lesz a file vége. Igy az aktuális pozíciótól kezdve a file végéig "lecsonkítjuk" a file-t. A levágott rész tartalma elvész.

11.2.3 A file lezárása

A típusos file-okat szintén a close eljárás hívásával kell lezárni:

close(*file_valtozo*)

11.2.4. Példák típusos file-ok használatára

a. Az első példában a tomb.dat nevű file-t feltöltjük *n* darab **longint** értékkel, majd a file tartalmát egyetlen ***read*** utasítással felolvassuk egy *n* elemű **longint** tömbbe (*t*).

```
program tomb;
  uses crt;

  const
       n     = 10;
  type
       ttomb = array[1..n] of longint;

  var
       f1    : file of longint;
       f2    : file of ttomb;
       l     : longint;
       t     : ttomb;
       i     : integer;

  begin
     clrscr;
     randomize;
     { A f1 file feltöltése n db számmal }
     assign(f1,'tomb.dat');
     rewrite(f1);

     for i:=1 to n do
       begin
          l:=random(maxint)*1000;
          write(f1,l);
          writeln(i:2,'. ',l);
       end;
     close(f1);
```

```
        { A file felolvasása a t tömbbe }
        assign(f2,'tomb.dat');
        reset(f2);
        read(f2,t);
        close(f2);

        {A felolvasott tömb kiírása a képernyõre }
        writeln;
        for i:=1 to n do
           writeln('t[',i:2,']=',t[i]);

     end.
```

b. Ha a programunkból nagy tömböt szeretnénk használni, akkor elképzelhető, hogy a dinamikus tárterület sem elegendő annak tárolására. Ekkor a tömb tárolását és elérését típusos file segítségével is megoldhatjuk, ún. virtuális tömb létrehozásával.

A példánkban egy 100x200-as real tömböt használunk (120000 byte).

```
program filetomb;
     uses crt;

     const m=100;
           n=200;
     var
        f     : file of real;
        i,j,k : integer;

     { A virtuális tömb megnyitása }
     procedure file_tomb_open;
     begin
        assign(f,'virtomb.dat');
        rewrite(f);
     end;

     { A tömb [s,o] elemének írása }
     procedure fwrite(s,o:integer; elem:real);
     var
        poz : longint;
     begin
        poz:=(s-1)*longint(n)+(o-1);
        seek(f,poz);
        write(f,elem);
     end;
     { A tömb [s,o] elemének olvasása }
```

```
function fread(s,o:integer): real;
var
   elem : real;
   poz  : longint;
begin
   poz:=(s-1)*longint(n)+(o-1);
   seek(f,poz);
   read(f,elem);
   fread:=elem;
end;

begin
  writeln('A file megnyitása');
  file_tomb_open;

  writeln('Pozícionálás az utolsó elemre.');
  fwrite(m,n,pi);

  writeln('A file tömb feltöltése.');
  for i:=1 to m do
      for j:=1 to n do
          fwrite(i,j,i*1000.0+j);

  writeln('Elemek véletlenszerû kiolvasása.');
  for k:=1 to 20 do
    begin
      i:=succ(random(m-1));
      j:=succ(random(n-1));
      writeln(i:5,' ',j:5,' ',fread(i,j):10:0);
    end;
  writeln;
  writeln(1:5,' ',1:5,' ',fread(1,1):10:0);

  writeln(m:5,' ',n:5,' ',fread(m,n):10:0);
  close(f);
  erase(f);
end.
```

c. Az utolsó példában a file-kezelő programok általános szerkezetét láthatjuk. Egyszerű menüvel vezérelve valósítottuk meg a file-ok kezelésének alapműveleteit (létrehozás, írás, keresés). A program telefonkönyvet kezel.

```
program telefonkonyv;
uses crt;
const
   telbook:string='telefon.dat';

type
   telefon=record
             nev : string[25];
             cim : string[30];
             tel : string[12];
           end;

var
    telfile: file of telefon;
    ch     : char;

{----------------------------------------------}
{  A régi telefonkönyvet megnyitó vagy az újat }
{             létrehozó eljárás.               }
{----------------------------------------------}
procedure opentel;
var
    tknev  : string;
begin
  gotoxy(20,13);
  write('Kérem a telefonkönyv file nevét: ');
  readln(tknev);
  if tknev<>'' then telbook:=tknev;
  clrscr;
  assign(telfile,telbook);
  {$I-}
  reset(telfile);
  if ioresult<>0 then
                begin
                  writeln('Uj telefonkönyv: ',
                           telbook);
                  rewrite(telfile);
                  end
                else
                  writeln('Régi telefonkönyv: ',
                           telbook);
end;

{----------------------------------------------}
{ Az s1 keresése az s2-ben, a kis- és nagybetûk}
{         megkülönböztetése nélkül             }
{----------------------------------------------}
```

```
function hasonlit(s1,s2:string):boolean;
var i:byte;
begin
  for i:=1 to length(s1) do
      s1[i]:=Upcase(s1[i]);
  for i:=1 to length(s2) do
      s2[i]:=Upcase(s2[i]);
  hasonlit:=pos(s1,s2)<>0;
end;

{-----------------------------------------------}
{ Adatok bevitele a meglévõ elemek átlépésével }
{-----------------------------------------------}
procedure felvitel;
var
    telrec : telefon;
begin
  opentel;
  { Pozicionálás a file végére }
  seek(telfile,filesize(telfile));

  with telrec do
  begin
     gotoxy(10,6);  write('Név          :  ');
                    readln(nev);
     gotoxy(10,8);  write('Lakcím       :  ');
                    readln(cim);
     gotoxy(10,10); write('Telefonszám :  ');
                    readln(tel);
  end;

  write(telfile,telrec);
  close(telfile);
end;

{-----------------------------------------------}
{  Soros keresés a definiált maszkok alapján   }
{-----------------------------------------------}
procedure kereses;
var
    keres,telrec : telefon;
    van          : array [1..3] of boolean;
begin
  opentel;
```

```
if eof(telfile) then { üres file }
           begin
             close(telfile);
             write(#7);
             exit;
           end;
gotoxy(10,4);
write('Kérem a keresendõ szövegrészleteket:');
gotoxy(10,14);
write('Tovább bármely billentyûvel');
window(1,2,80,12);

with keres do
begin
  gotoxy(10,6);  write('Név          :  ');
                 readln(nev);
  gotoxy(10,8);  write('Lakcím       :  ');
                 readln(cim);
  gotoxy(10,10); write('Telefonszám :  ');
                 readln(tel);
end;
seek(telfile,0);
while not eof(telfile) do
   begin
     read(telfile,telrec);
     van[1]:=hasonlit(keres.nev,telrec.nev);
     van[2]:=hasonlit(keres.cim,telrec.cim);
     van[3]:=hasonlit(keres.tel,telrec.tel);
     if van[1] and van[2] and van[3] then
        begin
          clrscr;
          with telrec do
          begin
               gotoxy(10,6);
               write('Név          :  ');
               write(nev);
               gotoxy(10,8);
               write('Lakcím       :  ');
               write(cim);
               gotoxy(10,10);
               write('Telefonszám :  ');
               write(tel);
            end;
            repeat until keypressed;
            ch:=readkey;
        end;
   end;
```

```
      window(1,1,80,25);
      close(telfile);
    end;

    {------------------------------------------------------}
    {    Véletlenszerû pozícionálás a file-ban              }
    {------------------------------------------------------}
    procedure lapozgatas;
    var
        telrec : telefon;
        poz    : longint;
        rsz    : longint;
    begin
      opentel;
      if eof(telfile) then  { üres file }
                 begin
                   close(telfile);
                   write(#7);
                   exit;
                 end;

      gotoxy(10,14);
  write('Tovább bármely billentyûvel - Kilépés: ESC');

      window(1,2,80,12);
      rsz:=filesize(telfile);
      randomize;

      while true do
         begin
           poz:=random(rsz);
           seek(telfile,poz);
           read(telfile,telrec);
           clrscr;
           with telrec do
           begin
              gotoxy(10,6);  write('Név          :  ');
                             write(nev);
              gotoxy(10,8);  write('Lakcím       :  ');
                             write(cim);
              gotoxy(10,10); write('Telefonszám :  ');
                             write(tel);
           end;
           repeat until keypressed;
           ch:=readkey;

           { Kilépés az eljárásból az ESC billentyûvel}
```

```
            if ch=#27 then
                        begin
                           window(1,1,80,25);
                           close(telfile);
                           exit;
                        end;
          end;
     end;
     begin
        {---------------------------------------------}
        {       A program vezérlését végzõ menü       }
        {---------------------------------------------}
        while true do
         begin
            clrscr;
            gotoxy(30,2);  writeln('Válasszon:');
            gotoxy(20,4);
              writeln('1: Uj telefonszámok felvitele');
            gotoxy(20,6);
              writeln('2:
              Keresés a telefonkönyvben');
            gotoxy(20,8);
         writeln('3: Lapozásgatás a telefonkönyvben');
            gotoxy(20,10);
              writeln('4: Kilépés a programból');

            repeat until keypressed;
            ch:=readkey;

            case ch of
              '1':  felvitel;
              '2':  kereses;
              '3':  lapozgatas;
              '4':  exit;
              else
                   begin
                     gotoxy(25,13);
                     writeln('Hibás választás!');
                     delay(500);
                   end;
            end;
        end;
     end.
```

11.3. Típus nélküli file-ok

A típus nélküli file-ok használata a Turbo Pascal-ban nagy segítséget jelent ismeretlen szerkezetű file-ok feldolgozásában.

```
PROGRAM tipus_nelkuli_file;
    CONST
       bs   = 512;              { Blokkméret }
       pm   = 4;                { A puffer mérete blokkokban }
    VAR
         f   : FILE;            { A file-változó }
         p   : longint;         { File-pozíció }
         puf : array[1..pm*bs] of byte;  { Puffer }
         nok : word;            { Sikeres blokkok száma }
    BEGIN
    { A Pascal - DOS kapcsolat megadása}
          ASSIGN(f,'file-név');
    { A file megnyitása }
    { Létező file megnyitása írásra / olvasásra }
            RESET(f);  { vagy }  RESET(f,bs);
              {vagy}
    { Uj file megnyitása írásra / olvasásra (létrehozás) }
            REWRITE(f); { vagy } REWRITE(f,bs);

    { File-mûveletek }
            BLOCKWRITE(f,puf,pm); { Irás a file-ba }
              {vagy}
            BLOCKWRITE(f,puf,pm,nok);
            BLOCKREAD(f,puf,pm);  { Olvasás a file-ból }
              {vagy}
            BLOCKREAD(f,puf,pm,nok);
            SEEK(f,p);            { Pozícionálás a file-ban }
            TRUNCATE(f);          { A file csonkítása }
    { A file-lal végzett mûveleteket segítõ függvények }
            p:=FILEPOS(f);        { Az aktuális file-pozíció }
            p:=FILESIZE(f);       { A file-ban található
                                    elemek száma }
            IF EOF(f) THEN ;      {A file-végének érzékelése }
    { A file lezárása }
          CLOSE(f);
    END.
```

11.5. ábra
Példa a típus nélküli file-t feldolgozó
Pascal program szerkezetére

Amíg a szöveg file-ok sorelválasztó karaktereket tartalmaznak, a típusos file-ok adott típusú adatelemekből állnak, addig a típus nélküli file-ból tetszőleges adatok olvashatók illetve írhatók a file-ba. Mivel az adatforgalom a file és program között mindenféle ellenőrzés nélkül megy végbe, gyors file-kezelés valósítható meg. A típus nélküli file-ok kezelésének lépéseit a 11.5. ábrán látható programban foglaltuk össze.

Mint ahogy az ábrán is látható, a típusos file-ok esetén használt eljárások és függvények többsége a típus nélküli file-okhoz is használhatók. Az alábbiakban a két file-kezelés közötti különbségekre hívjuk fel a figyelmet.

A típus nélküli file-okkal végzett műveletekben egyetlen lépésben elérhető egység a blokk. A blokk mérete alapértelmezés szerint 128 byte, de ez a file nyitásakor a ***reset*** és a ***rewrite*** eljárások második paraméterével módosítható. (A legtöbb számítógépen a leghatékonyabb blokkméret 512 vagy ennek többszöröse.)

Ennek megfelelően a ***seek*** eljárás blokkonkénti pozícionálást hajt végre, illetve a ***filepos*** függvény az aktuális blokk sorszámával tér vissza.

A ***filesize*** függvény pedig a

file fizikai mérete (byte) **div** *blokkméret*

kifejezés értékével tér vissza. Ha a file-hossza nem osztható maradék nélkül a blokkmérettel, akkor a fenmaradó byte-ok nem érhetők el. Ezért, ha olyan programot kívánunk írni, amely tetszőleges file-t fel tud dolgozni, akkor a blokkhosszat 1-nek kell választanunk.

A típus nélküli file-ok írása és olvasása szintén blokkokban történik. A blockread eljárást használjuk a file-ból való olvasásra, míg ***blockwrite*** eljárás a file írására szolgál:

```
procedure BlockRead( var file_valtozo : file;
                     var puffer       : tetszőleges típus;
                         cnt          : Word [;
                     var res          : Word ]
                   );
```

```
procedure BlockWrite( var file_valtozo : file;
                      var buffer       : tetszőleges típus;
                          cnt          : Word [ ;
                      var res          : Word ]
                    );
```

A két eljárás paraméterezése teljesen megegyezik. Meg kell adnunk a file-t azonosító file-változót, az adatátvitelhez használt puffert és az egy lépésben olvasni illetve írni kívánt blokkok számát (*cnt*). A negyedik opcionális változóparaméterben (*res*) adja vissza a rendszer a ténylegesen beolvasott illetve kiírt blokkok számát.

A blokkos I/O műveletek során a rendszer semmilyen ellenőrzést sem végez arra vonatkozólag, hogy puffer mérete és a *cnt*blokkhossz* adat összhangban van-e egymással. Erről a program írása során kell gondoskodnunk. Egyetlen korlátja azonban mégis van a fenti eljárások használatának, nevezetesen az, hogy a *cnt*blokkhossz* érték nem lehet nagyobb 65535-nál (64K).

A fejezetben ismertetett lehetőségek összefoglalásaként tekintsük a *mycopy* programot, amelyet parancssorból paraméterezve használhatunk:

MYCOPY *forrás_file cél_file*

A program a DOS COPY parancsához hasonlóan a forrás_file állomány tartalmát átmásolja a cél_file állományba. A művelethez egy 16 Kbyte-os puffert használtunk, amely mérete tovább növelhető ha azt dinamikusan, a heap területen hozzuk létre.

```
program mycopy;

type puf=array[1..32*512] of byte;  { 16 Kbyte puffer }

      var
        fromf, tof            : file;
        n_olvasott, n_kiirt : word;
        puffer                : puf;
```

```
procedure halterr(msg:string);
begin
 writeln(#7,'Futási hiba :  ',msg);
 writeln(#7,'a program futása véget ért.');
 halt;
end;

begin
{ A paraméterezés tesztelése }

if paramcount<>2 then
   halterr(#13+#10+#13+#10+
        'Használat: MYCOPY forrás_file'+
            ' cél_file'+#13+#10);

  { A forrás file megnyitása}
  assign(fromf, paramstr(1));
  {$I-}
  reset(fromf, 1);
  {$I+}
  if ioresult<>0 then
     halterr('nem elérhetõ file: '+paramstr(1));

  { A cél file megnyitása }
  assign(tof, paramstr(2));
  {$I-}
  rewrite(tof, 1);
  {$I+}
  if ioresult<>0 then
     halterr('nem elérhetõ file: '+paramstr(2));

  { Másolás }
  writeln(filesize(fromf),
         ' byte másolása folyamatban ...');

  repeat
     blockread(fromf,puffer,sizeof(puf),n_olvasott);
     blockwrite(tof,puffer,n_olvasott,n_kiirt);
  until (n_olvasott = 0) or (n_kiirt <>n_olvasott);

  close(fromf);
  close(tof);
end.
```

11.4. Eszközök (device) használta

A Turbo Pascal és a DOS operációs rendszer a külső hardvert (billentyűzet, képernyő, nyomtató) eszközökként kezeli. Programozói szempontból az eszköz file, amelyet ugyanazon eljárásokkal és függvényekkel érhetünk el, mint a lemez file-okat.

A Turbo Pascal két csoportba osztja az eszközöket:

- DOS eszközök,
- szöveg file eszközök.

Jó példa a szöveg file eszköz megvalósítására a szabványos CRT unit, amely használata gyorsabb és kényelmesebb lehetőségeket biztosít a képernyő és a billentyűzet elérésére, mint a DOS CON eszköze. A fejezetben a DOS eszközök használatát foglaljuk össze.

Mint ahogy azt a fejezet bevezetőjében említettük a file-változót vagy lemez file-hoz, vagy pedig valamilyen eszközhöz rendelhetjük hozzá. Az összerendelés az *assign* rutin hívásával történik, ahol a file-név helyén a 11.6. ábrán összefoglalt előredefiniált eszközneveket kell megadni.

A DOS eszközök kezelésére általában a szöveg file-okat használunk, de bizonyos esetekben a típus nélküli file hatékonyabb megoldást biztosít.

A NUL eszköz egyszerűen elnyeli a rá írt adatokat, és file-véget jelez, ha olvassuk. Általában akkor használjuk, amikor a program mindenképpen vár valamilyen file-nevet, de nem akarunk semmilyen file-t létrehozni.

File-név	Szabványos eszköz	Input vagy Output
'AUX'	Auxiliary (a COM1 szinonim neve)	I/O
'COM1', 'COM2'	Soros kommunikációs portok	I/O
'CON'	Konzol (I - billenyűzet, O - képernyő)	I/O
'LPT1', 'LPT2', 'LPT3'	Nyomtató portok	O
'PRN'	Nyomtatók (LPT1 szinonim neve)	O
'NUL'	Nulla eszköz	I/O

11.6. ábra
A DOS eszközök neve

Az eszközök felhasználását a CON_PRN.PAS program szemlélteti. Ha a programot file-név paraméter nélkül indítjuk, akkor az megkérdezi a kiírandó file nevét is, különben pedig csak azt, hogy a nyomtatót (p), vagy a konzolt (c) kívánjuk használni.

```
program con_prn;

var
    file_nev   : string;
    hova       : char;
    infile     : text;
    outfile    : text;
    egy_sor    : string;
```

```
begin
    if (paramcount > 1) then
        begin
        writeln( 'Használat: con_prn <file_nev>' );
            exit;
        end
    else
        if (paramcount = 1) then
           file_nev := paramstr( 1 )
        else
            begin
                write( 'A kiírandó file neve: ');
                readln( file_nev );
            end;
    if file_nev='' then exit;

    repeat
        write( 'Kiírás nyomtatóra (''p'')'+
               ' vagy képernyõre (''c''):  ' );
        readln( hova );
        hova := upcase( hova );
    until (hova = 'P') or (hova  = 'C');

    if (hova = 'P') then
        assign( outfile, 'prn' )
    else
        assign( outfile, 'con' );

    assign( infile, file_nev );
    reset( infile );
    rewrite( outfile );
    while not eof( infile ) do
        begin
            readln( infile, egy_sor );
            writeln( outfile, egy_sor );
        end;

    close( infile );
    close( outfile );

end.
```

11.5. File-ok törlése, átnevezése

A file törlésére az ***erase*** eljárást, az átnevezésére pedig a ***rename*** eljárást használhatjuk. Mindkét esetben a műveletben résztvevő file-t file-változóval azonosítjuk, amelyet az ***assign*** segítségével rendelünk a file-hoz:

assign(*file_valtozo,file_nev*);

erase(*file_valtozo*); { törlés }

rename(*file_valtozo,uj_nev*); { átnevezés }

Az átnevezés során az *uj_nev* sztring tartalmazza az új file-nevet.

A műveletek sikerességét a {$I-} kapcsoló megadása után, az ***ioresult*** függvény értékéből tudhatjuk meg (0 - sikeres). Nyitott állományra sem az ***erase***, sem a ***rename*** eljárást nem szabad használni.

11.6. Könyvtárak kezelése

A könyvtárak kezelésére szolgáló DOS parancsoknak léteznek Turbo Pascal megfelelőjük:

Művelet	DOS	Turbo Pascal
könyvtár váltása	CD/CHDIR temp	chdir('temp');
könyvtár létrehozása	MD/MKDIR c:\pas	mkdir('c:\pas');
könyvtár törlése	RD/RMDIR c:\dos	rmdir('c:\dos');

Az eljárások hívásakor a szokásos I/O hibakezelést használhatjuk.

A

getdir(*drv*,*dirnev*)

eljárás segítségével megtudhatjuk a *drv* meghajtón az akuális könyvtár nevét a *dirnev* változóparaméterben. A *drv* **byte** típusú paraméter definiálja a meghajtót:

0 - aktuális meghajtó
1 - A:
2 - B:
3 - C:
....

Ha a megadott meghajtó nem létezik akkor az eljárás '\' könyvtárnevet ad vissza. (Nincs I/O hibafigyelés!)

Az eljárások felhasználását az alábbi program szemlélteti:

```
program dir_muv;

procedure write_act_dir;
var
    dn:string;
begin
    getdir(0,dn);
    writeln(dn);
end;

procedure errchk(es:string);
begin
  if ioresult<>0 then
                begin
         writeln('Hibás directory mûvelet - ',es);
                  halt;
                end;
end;

begin
  {$I-}
  write_act_dir;
  mkdir('tempy');
  errchk('A directory már létezik.');
  chdir('tempy');
```

```
        errchk('Nincs ilyen directory');
        write_act_dir;

        chdir('..');
        errchk('Nincs ilyen directory');
        write_act_dir;

        rmdir('tempy');
        errchk('A directory nem létezik vagy nem üres.');
        {$I+}
      end.
```

Ha a program indításakor az aktuális könyvtár a C:\PASCAL, akkor a program kimenete:

```
C:\PASCAL
C:\PASCAL\TEMPY
C:\PASCAL
```

Ellenőrző kérdések:

1. Hasonlítsa össze a szöveges és a típusos file-t az alábbi szempontok alapján:

   - elérési lehetőségek,
   - az elemek jellemzői,
   - megnyitási módok,
   - pozícionálás a file-ban.

2. Ismertesse a file-kezelés általános lépéseit!
3. Ismertesse a ***reset*** eljárás működését mind a három file-típus esetében?
4. Hogyan lehet a I/O hibákat a programon belül lekezelni?
5. Definiáljon rekordot, amely a következő adatokat tartalmazza: név, születési év, hónap, nap, hely, fizetés, adósság, lakcím. Deklaráljon ehhez a típushoz file-változót közvetlen elérésű állományhoz!
6. Csopostosítsa a következő eljárásokat és függvényeket:
   seek, append, assign, close, write, readln, rewrite, filepos, filesize, blockread, eoln, eof, reset
   csak text file   csak típusos file   mindkettő   egyik sem

Feladatok:

1. Írjon programot, amely az

```
type
     rt = record
          tipus : string[12];
          ear   : real;
          db    : integer;
          memkb : integer;
        end;
```

   rekordokat tartalmazó file-t létrehozza, illetve amely kiírja annak tartalmát, miközben az *ear* mezőket összegzi.
   (RTCREAL.PAS, RTREAD.PAS)

2. Írjon **boolean** típusú függvényt, amely egy adott nevű file-ról megmondja, hogy létezik-e (*true*=igen).
   (FEXIST.PAS)

3. Egy szöveges file tetszőleges számú **real** adatot tartalmaz soraiban. Írjon programot, amely a szöveges file tartalmát **real** típusú file-ba másolja!
   (TEXTREAL.PAS)

4. Írjon programot, amely adott nevű szöveges file-t lapokra tördelve másik szöveg file-ba másol! A program kérje be a file-ok neveit, sorok számát a lapon, oldalsorszám kezdetét és a fejléc szövegét.
   (TORDEL.PAS)

12. A TURBO PASCAL MEMÓRIAHASZNÁLATA

12.1. A mutató típus - dinamikus változók

Az eddig használt egyszerű és struktúrált típussal rendelkező változók közös tulajdonsága, hogy rájuk a deklarálás során megadott névvel hivatkozhatunk. Az így definiált változók mérete a program fordításakor dől el, és a program futása során nem módosítható. A Pascal nyelvben ezeket a változókat *statikus változóknak* nevezzük.

A hatékony memóriafelhasználás érdekében szükség van arra, hogy bizonyos memóriaterülettel a programozó saját maga gazdálkodhasson - ha szükség van terület kijelölésére valamely változónk számára, azt onnan elvégezhessük, illetve ha már nincs szükségünk a változóra, a lefoglalt területet felszabadíthassuk. Ilyen módon megvalósíthatjuk a memória dinamikus felhasználását. A *dinamikus változók* számára rendelkezésre álló területet a Turbo Pascal-ban *halomterületnek* (*heap*) nevezzük. Ezen terület mérete jóval meghaladja program egyéb adatterületeinek méretét. A kérdés most már csak az, hogy hogyan férhetünk hozzá a *heap* területhez?

Mint ahogy azt az előző fejezetekben láttuk, a

```
var A:integer;
```

deklaráció hatására a fordítóprogram lefoglal a memóriában egy 2 byte hosszúságú területet az *A* változó számára. Az *A* változóra a nevének felhasználásával hivatkozhatunk:

```
A:=22;
A:=A+10;
```

Ahhoz, hogy a memóriafoglalás fenti lépéseit a program futása során végezhessük el, meg kell ismerkednünk egy új változótípus, a mutató (*pointer*) fogalmával. A mutató típusú változó olyan speciális változó (4 byte hosszú), amely memóriacímet tartalmaz. A magyar nyelvű szakirodalom a *mutató* és a *pointer* szavakat egyaránt használja.

A mutató deklarálása:

```
var ap : ^integer;
```

A ^ jel után tetszőleges szabványos vagy felhasználó által definiált típus állhat. Az *ap* mutató a deklarálás után semmilyen meghatározott címet sem tartalmaz, tehát "sehova sem mutat".

Turbo Pascal-ban a mutatónak többféleképpen lehet értéket adni. Most azonban tekintsük csak a legegyszerűbb szabványos megoldást. A programon belüli helyfoglalás elvégzésére a ***new*** eljárás használható, amely a lefoglalt terület címét a paraméterként átadott mutatóba tölti:

new(ap);

Ha nincs elegendő hely a halomterületen, a programunk 203-as futás közbeni hibajelzéssel leáll: "*Heap overflow error*". A hibajelzés elkerülhető, ha a ***new*** aktiválása előtt ellenőrizzük, hogy van-e elegendő szabad terület a *heap*-en. Hasonlítsuk össze a ***maxavail*** függvény által visszaadott maximális szabad blokk méretét az **integer** változó tárolásához szükséges mérettel (`sizeof(integer)`). Ekkor az előző programsor a következőkkel helyettesíthető:

```
if maxavail<sizeof(integer) then halt
                            else new(ap);
```

Megjegyezzük, hogy a ***memavail*** függvény segítségével a halomterületen található szabad területek összmérete is lekérdezhető.
A Pascal-ban létezik egy előredefiniált **nil** konstans, amelyet a *null* pointer jelölésére használjuk, így megengedett az `ap:=nil` értékadás is.

Ha a memóriafoglalás sikeres volt, következik az *ap* "által mutatott" terület felhasználása:

```
ap^:=22;
ap^:=ap^+10;
```

Jól látható az *A* változó és az *ap^* hivatkozás közötti analógia. A pointer neve után álló ^ jel hatására nem a mutatóra, hanem az általa kijelölt memóriaterületre hivatkozhatunk.

Ha már nincs szükségünk az *ap^* egész típusú változóra, akkor a ***dispose*** eljárás hívásával törölhetjük azt:

```
dispose(ap);
```

A ***dispose*** hívás utáni, illetve a ***new*** hívás előtti *ap^* hivatkozás programhibához vezethet. A Pascal nyelven fontos tudnunk azt, hogy a pointernek van típusa. Ez a típus azt jelöli ki, hogy a dinamikus helyfoglalás során mekkora területet kell a mutatóhoz hozzárendelni.
A dinamikus helyfoglalás előnye természetesen akkor tűnik ki igazán, ha nem néhány byte-os, hanem több kbyte-os dinamikus változót (például tömb), használunk.

Az elmondottak jobb megértése érdekében nézzünk néhány rövid példát a dinamikus változók felhasználására.

12.1.1. Tömb a halomterületen

A *tombt* típusú tömböt, melynek mérete 60000 byte, a *tptr* segítségével helyezzük el a *heap*-en. A tömb ebben az esetben a *tptr^* lesz, míg az elemekre a *tptr^[i]* kifejezéssel hivatkozhatunk. Felhívjuk a figyelmet, hogy struktúrált típusra mutató pointer deklarációjában a struktúrált típust felhasználó által definiált típusnévvel kell azonosítanunk (**type**).

```
program ptl;
type
      tombt    = array[1..10000] of real; {60000 byte!}
      tombptr = ^tombt;

var
      tptr : tombptr;
         i : integer;

begin
```

```
{ helyfoglalás a tömb számára}

new(tptr);

{ a tömb feltöltése }

for i:=1 to 10000 do
    tptr^[i]:=i*pi;
{a véletlenszerû i-dik tömbelem kiíratása}

randomize;
i:=random(10000)+1;
writeln(i:10,tptr^[i]:16:5,tptr^[i]/i:10:5);

{a tömb által lefoglalt terület felszabadítása}

dispose(tptr);

end.
```

12.1.2. Mutatótömb használata

Ha az előző példában szereplő *tombt* típusú tömbből többet (amennyi elfér a memóriában) szeretnénk használni, érdemes az egyedi tömbmutatók helyett a mutatótömböt bevezetni. Ezáltal az i-dik tömbre a *ptrtomb[i]^*, míg az j-dik tömb elemeire a *ptrtomb[i]^[j]* kifejezésekkel hivatkozhatunk. Külön kiemelést érdemel a 203-as hiba elkerülését biztosító foglalási lépéssorozat.

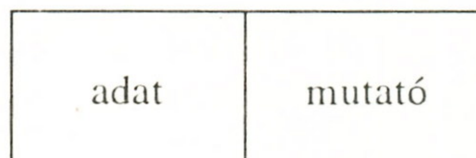
```
program pt2;

const
      n=10;
type
      tombt   = array[1..10000] of real; {60000 byte!}

var
      ptrtomb : array[1..n] of ^tombt;
      maxi,i,j: integer;
begin

  { helyfoglalás a tömbök számára, a 203-as hiba kivédésével}
  maxi:=0;
```

```
  while maxavail>=sizeof(tombt) do
    begin
      inc(maxi);
      new(ptrtomb[maxi]);
      writeln(maxi,'. tömb foglalása kész');
    end;

  { a tömbök feltöltése 0.0 értékkel}
  for i:=1 to maxi do
    begin
      for j:=1 to 10000 do
        ptrtomb[i]^[j]:=0.0;
      writeln(i,'. tömb feltöltése kész');
    end;

  {a tömbök által lefoglalt terület felszabadítása}
  for i:=1 to maxi do
    begin
      dispose(ptrtomb[i]);
      writeln(i,'. tömb felszabadítása kész');
    end;
end.
```

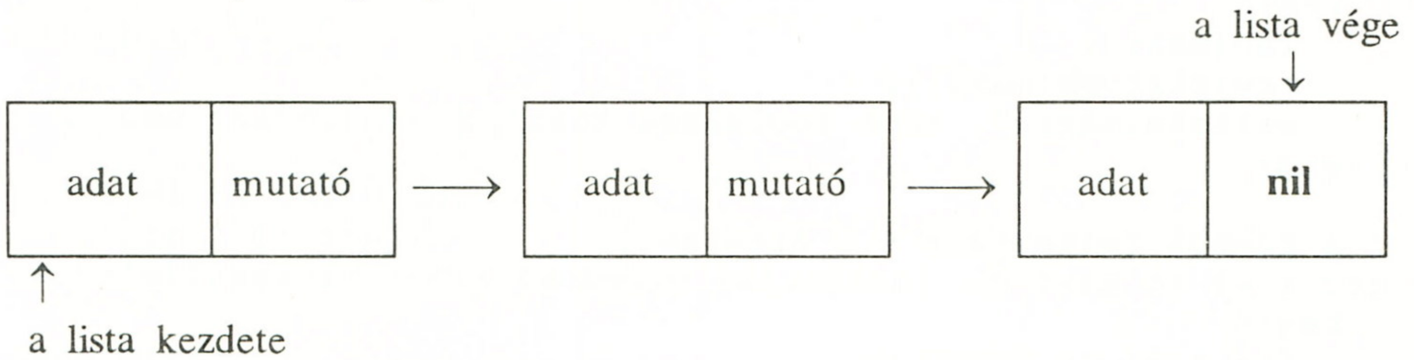
12.1.3. A lista tárolási szerkezet

A dinamikus adatterületek optimálisan a lista stuktúra segítségével kezelhetők. A lista struktúra Pascal nyelven történő megvalósításához a rekord típus használható. Az így keletkező lista tulajdonképpen egy rekordláncnak tekinthető.

A legegyszerűbb struktúrájú lista (*lineáris lista*) alapján megnézzük a listák kezelésének alapvető lépéseit. A lista elemei a listaelem tartalmaz adatokat és mutatót a következő listaelemre:



A lista felépítése pedig a következő



Az alábbi példában tetszőleges hosszúságú (max 64K/5) sztring kezelését valósítjuk meg lineáris lista felhasználásával. A listelemet a *karakter* típusú rekordban tároljuk.
A

```
var strptr:mutato;
```

deklaráció felhasználásával az *strptr^* jelöli a *karakter* típusú rekordot (*new(strptr)*; után). Mint ismeretes a rekord mezőire hivatkozhatunk a . (pont) jellel:

```
strptr^.ch        :='A';
strptr^.kovetkezo:=nil;
```

vagy pedig a **with** utasítás segítségével:

```
with strptr^ do
  begin
    ch       :='A';
    kovetkezo:=nil;
  end;
```

E rövid bevezető után tekintsük meg magát a programot:

```
program pt3;
uses crt;
 type
   mutato=^karakter;
   karakter=record
              ch:char;
              kovetkezo:mutato;
            end;
```

```
var
  str_ptr,             {pointer a krakterlánc kezdetére }
  char_ptr:mutato;     {pointer a lánc valamely eleméhez }
  mem0:longint;

procedure char_read(var list_ptr:mutato);

{    feladat:egy karakter beolvasása a
     billentyûzetrõl és hozzáfûzése
     a lánchoz                          }
{parameter: a hozzáfûzött karakter mutatója }

var k:char;
begin
  k:=readkey;
  if ord(k) in [32..255]
  then begin
         write(k);
         new(list_ptr);
         list_ptr^.ch:=k;
         list_ptr^.kovetkezo:=nil;
       end
  else list_ptr:=nil;
end;

procedure str_read(var list_ptr:mutato);

{feladat   : karaktersorozat beolvasása a
             billentyûzetrõl      }
{parameter : a karaktersorozat
             mutatója             }

var tmp_ptr:mutato;
begin
  char_read(list_ptr);    { az elsõ karakter bevitele }
  tmp_ptr:=list_ptr;
  while tmp_ptr<>nil do
  begin
    char_read(tmp_ptr^.kovetkezo);
    tmp_ptr:=tmp_ptr^.kovetkezo;
  end;
end;

procedure char_print(list_ptr:mutato);

{feladat: karakterlánc megjelenítese
          a képernyõn              }
```

```
{parameter: mutató a megjelenítendõ
            karakterlánc kezdetére   }

begin
  writeln;
  while list_ptr<>nil do
  begin
    write(list_ptr^.ch);
    list_ptr:=list_ptr^.kovetkezo;
  end;
  writeln;
end;

function str_len(list_ptr:mutato):integer;

{feladat:  karakterlánc hosszának
           lekérdezése               }
{parameter: mutató a karakterlánc
           kezdetére                 }

var h:integer;
begin
  h:=0;
  while list_ptr<>nil do
  begin
    inc(h);
    list_ptr:=list_ptr^.kovetkezo;
  end;
  str_len:=h;
end;

procedure str_free(list_ptr:mutato);

{feladat: karakterlánc megszüntetése  }
{parameter:mutató a felszabadítandó karakterlánc
          kezdetére  }

var str_ptr:mutato;
begin
  while list_ptr<>nil do
  begin
    str_ptr:=list_ptr^.kovetkezo;
    dispose(list_ptr);
    list_ptr:=str_ptr;
  end;
end;
begin
```

```
    clrscr;
    mem0:=memavail;        { a kiindulási heap-méret }

    write('Kérek egy karaktersorozatot: ');
    str_read(str_ptr); {a sztring beolvasása és tárolása }

    char_print(str_ptr);{ a karakterlánc kiírása }

    writeln('A sztring hossza            : ',
        str_len(str_ptr),' byte');

    writeln('A lefoglalt terület         : ',mem0-memavail,
          ' byte');

    str_free(str_ptr); { a sztring törlése a memóriából }

  end.
```

12.1.4. Saját verem kialakítása

Az alábbi példában 2..32 alapú számrendszerbe alakítunk át decimális számokat. Az algoritmus a maradékos osztás elvén alapul. Tegyük fel, szeretnénk az 1206 számot átalakítani 16-os számrendszerbe, a maradékos osztás felhasználásával. Az algoritmus menete az alábbi ábrán látható:

1206	16
75	6
4	11
0	4

A maradékokat fordított sorrendben összeolvasva kapjuk az új számot: 4B6.

A megoldásban a verem szerkezetet a keletkező maradékok dinamikus tárolására és fordított sorrendben történő kiolvasására használjuk.

```
program pt4;

{ A feladat megoldásához a saját verem (stack) használata
ajánlott}

type mutato = ^elem;
     elem   = record
                szamjegy: byte;
                elozo   : mutato;
              end;

var  alap,i: integer;
     szam  : longint;
     verem, seged:mutato;
     var jegyek:array[0..31] of char;

begin
   { a lehetséges számjegyeket tartalmazó tömb feltöltése}

   for i:=0  to  9 do jegyek[i]:=chr(48+i);
   for i:=10 to 31 do jegyek[i]:=chr(55+i);

   write('Kérem a számrendszer alapját:',#9#9);
   readln(alap);
   if not (alap in [2..32]) then halt;

   write('Az átváltandó szám:', #9#9#9);
   readln(szam);

   {  A maradékok elhelyezése a veremben,  }
   {  a helyfoglalás elvégzésével          }

   verem:=nil;
   while szam<>0 do
   begin
       new(seged);
       seged^.szamjegy:= szam mod alap;
       seged^.elozo    := verem;
       verem := seged;
       szam := szam div alap;
   end;

   {  A szám kiírása a veremből történõ,   }
   {  visszaolvasással                     }

   write('Az adott számrendszerbeli alak:',#9#9);
   seged := verem;
```

```
   while seged<>nil do
   begin
      write(jegyek[seged^.szamjegy]:1);
      seged := seged^.elozo;
   end;
   writeln(#13#13);

   {  A verem által lefoglalt memória-      }
   {  terület felszabadítása                }

   seged:= verem;
   while seged<>nil do
   begin
     verem := seged^.elozo;
     dispose(seged);
     seged:= verem;
   end;
end.
```

12.2. További lehetőségek a memória elérésére

Az 12.1. részben ismertetett szabványos megoldásokon felül a Turbo Pascal egy sor új lehetőséget tartalmaz a számítógép memóriájának elérésére. Ahhoz azonban, hogy élni tudjunk ezekkel a lehetőségekkel mélyebben meg kell ismerkednünk a memória felépítésével és címzésével.

12.2.1. Amit a 8086 mikroprocesszorról tudni kell

Ahhoz, hogy teljes mértékben kezünkben tarthassuk a Pascal programok memóriahasználatát, nézzük meg, hogyan is címzi a memóriát az IBM PC mikroprocesszora. A 8086 mikroprocesszor 16 bites (16 bitnyi információt fogad, illetve küld ki egy lépésben) - ennek megfelelően minden belső regisztere 16 bites.

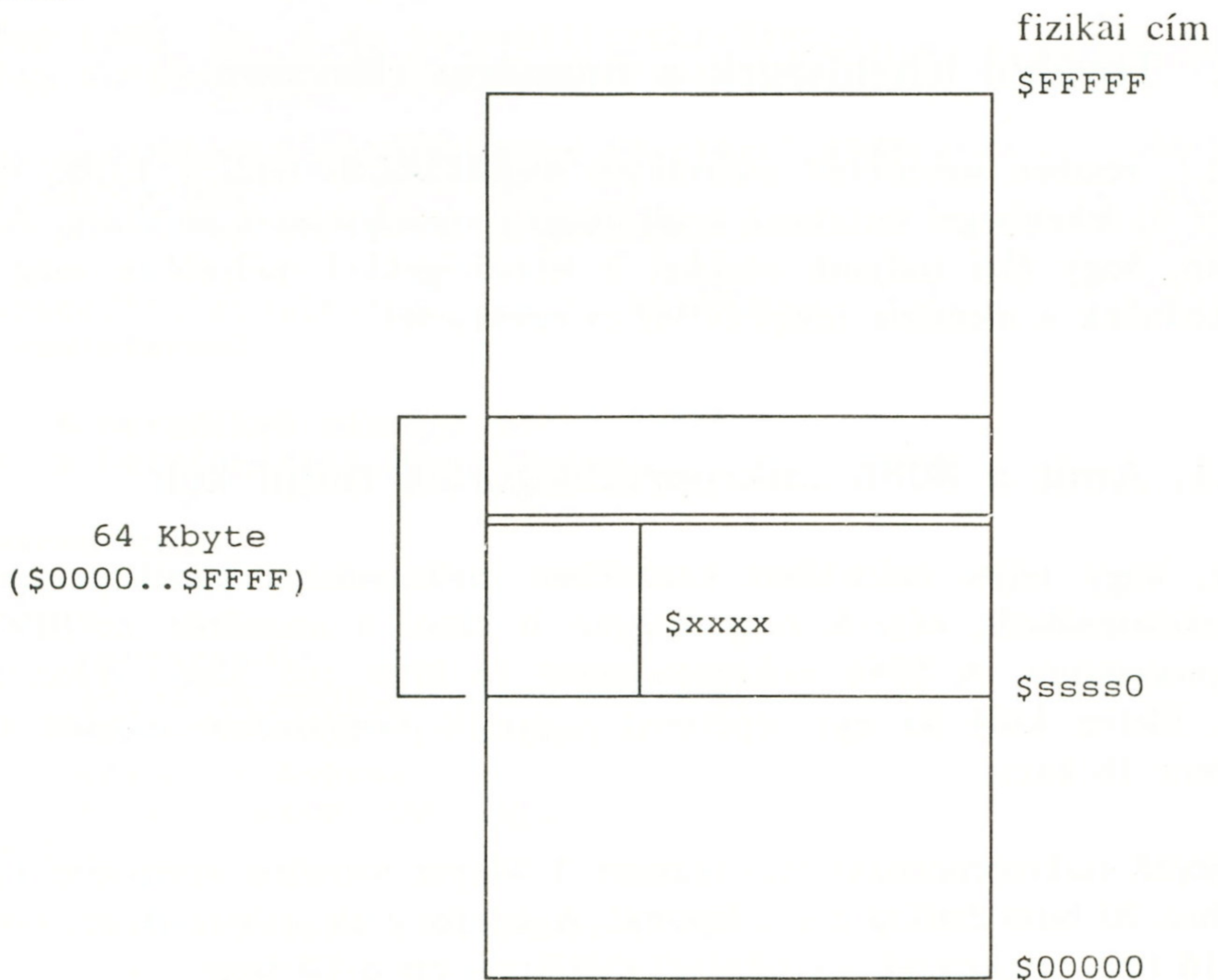
A 8086/88 mikroprocesszor maximálisan 1 Mbyte memória címzésére képes, amelyhez 20 bites fizikai címet használ. A kérdés ezek után csak az, hogyan lehet 16 bites regiszterek segítségével előállítani ezt a 20 bitet.
A megoldást a memória szegmentált címzése adja, amely a 12.1. ábrán látható.

Ha megnézzük a fizikai címeket, akkor azt tapasztaljuk, hogy minden 16-dik cím olyan, hogy az alsó 4 bitje 0 ($ssss0), így ezen címek azonosításához elegendő a felső 16 bitet használni ($ssss). Ezzel a 16 bites értékkel (*szegmenscím*) egy ún. szegmenst jelölünk ki a memóriában. Ezek után a kívánt byte címe egyszerűen megadható a szegmens kezdetétől mért távolsággal, amely szintén 16 bites érték ($xxxx - *offszetcím*). Mivel az offszet 16 bites, egy szegmens maximális mérete 64 Kbyte, amely korlátozás lépten nyomon kísért a Pascal programunk fejlesztése során.

A szegmes- és offszetcím segítségével a memória tetszőleges területe megcímezhető a

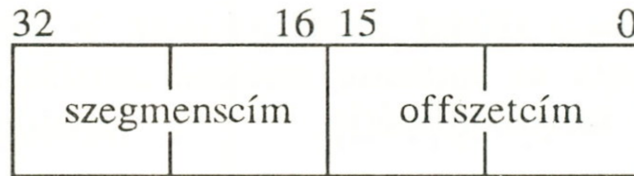
$ssss : $xxxx *(szegmenscím : offszetcím)*

alakban. Például a színes szöveges képernyő pufferterülete a $b800:0 címen kezdődik.



12.1. ábra A memória szegmentált címzése

A Turbo Pascal a címeket 4 byte hosszú területen tárolja:



A 8088/86 mikroprocesszor egyszerre maximálisan négy szegmenst képes megcímezni az ún. *szegmensregiszterek* felhasználásával:

CS - a kód (futó program) szegmense
DS - az adatszegmens
SS - a *stack* szegmense
ES - másodlagos adatszegmens (általában munkaterület)

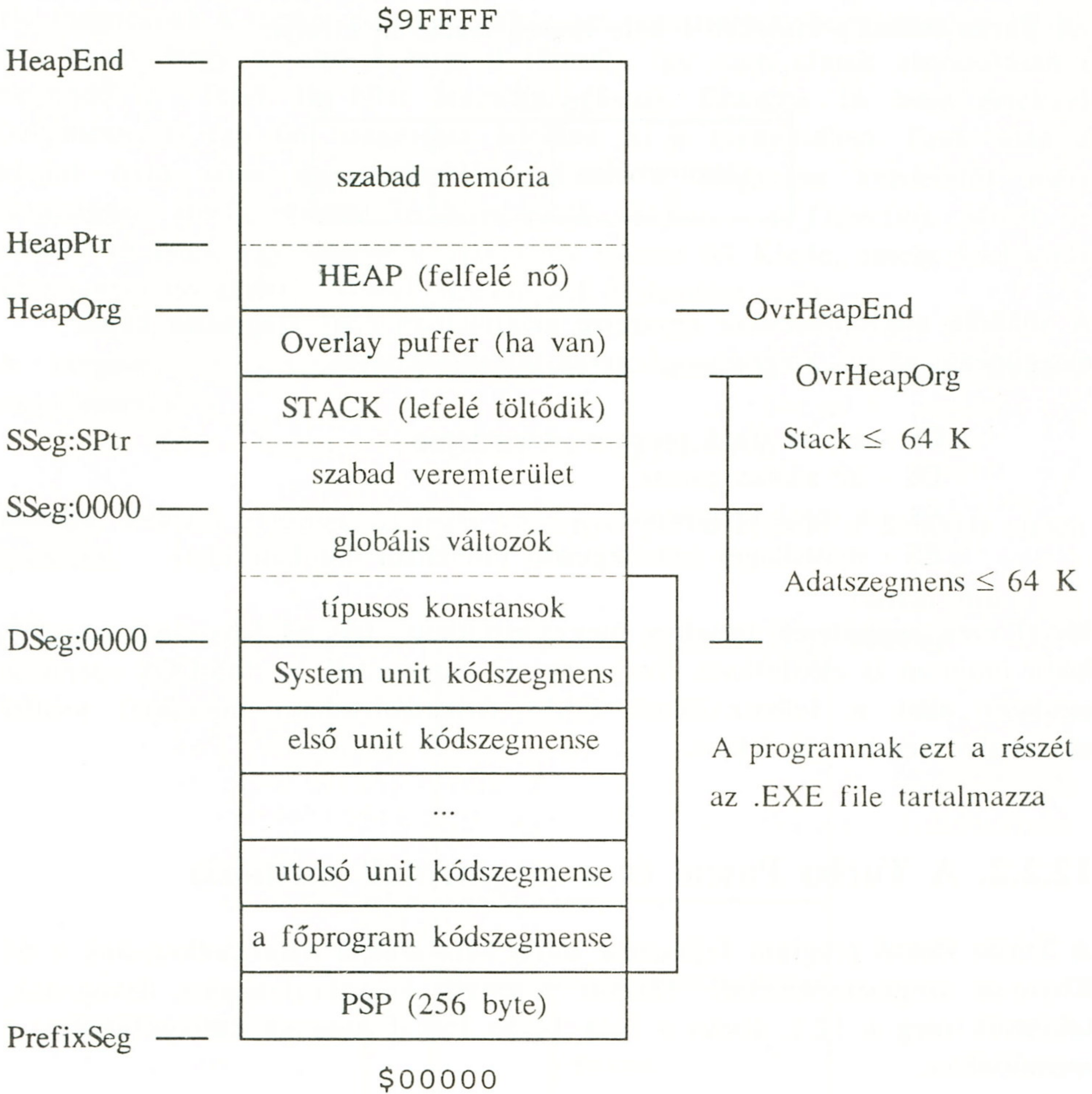
Mivel ezen regiszterek tartalma megváltoztatható, így 64 Kbyte-nál nagyobb kódú program is előállítható. Turbo Pascal programból, az MS-DOS operációs rendszer alatt a felhasználható RAM (írható/olvasható memória) terület maximális mérete 640 Kbyte.

12.2.2. A Turbo Pascal és a szegmentált memória

A Turbo Pascal program fejlesztése során nem szabad megfeledkeznünk a 64 Kbyte-os szegmensméretről. Mielőtt bizonyos következtetéseket levonnánk, tekintsük meg a 12.2. ábrán a futó Turbo Pascal program elhelyezkedését a memóriában.

Mire kell figyelnünk a program írása során?

- Mint ahogy az ábrán látható a Turbo Pascal kódrésze több (legalább két), szegmensből áll. Minden egyes szegmens maximális mérete 64 Kbyte lehet. Ha tehát 64 Kbyte-nál nagyobb kódú programot kell írnunk, ezt csak akkor tehetjük meg, ha a programot részekre (modulokra - **unit**-okra) bontjuk.



12.2. ábra A futó Turbo Pascal program elhelyezkedése a memóriában

- Nagyon súlyos korlát az adatszegmens 64 Kbyte-os határa. A program által használt statikus adatok (globális változók és típusos konstansok) által használt memória nem haladhatja meg a 64 Kbyte-ot. Ezt a kötöttséget csak a dinamikus változók bevezetésével lehet feloldani.

- A verem (*stack*) maximális mérete szintén 64 Kbyte, amely érték az eljárásokban és függvényekben használt lokális változók mennyiségét korlátozza. Ne feledjük el azonban, hogy lokális mutató segítségével szintén használhatunk *heap*-területet, amelyet azonban az alprogramból való kilépés előtt fel kell szabadítanunk!

- Az ábráról nem olvasható le, de tudnunk kell, hogy a Turbo Pascal-ban tetszőleges adatstruktúra (tömb, rekord) maximális mérete szintén 64 Kbyte.

A $M globális fordítási direktíva megadásával beállítható a program *stack* és *heap* területe. A $M alapértelmezés szerinti formájában

{$M 16384,0,655360}

a program 16 Kbyte verem- illetve maximális (a 640 Kbyte-os határig) halomterülettel rendelkezik. A direktíva általános alakja:

{$M *stack-méret, minimális heap-méret, maximális heap-méret*}
(heapmin) (heapmax)

A paraméterek lehetséges étékei:

paraméter	Minimum	Maximum
stack-méret	*1024*	*65520*
heapmin	*0*	*655360*
heapmax	*heapmin*	*655360*

A *heapmin* azt a minimális halomterület méretet definiálja, amely megléte esetén indul csak el a programunk. A *heapmax* a futás során felhasználható maximális *heap*-területet méretet adja meg.

12.2.3. Speciális lehetőségek a memória elérésére

12.2.3.1 A Mark és a Release eljárások használata

A ***mark*** eljárás segítségével valamely mutatóban feljegyezhetjük a heap állapotát. Ez a feljegyzett állapot a ***release*** eljárás hívásával egyszerűen visszaállítható. Másképp fogalmazva, a ***release*** hívással az összes ***mark*** után lefoglalt memóriablokk felszabadításra kerül.

```
program mark_release;

type  arr=array[1..20000] of byte;
var   ap,bp,cp,mp : ^arr;


begin
   writeln;
   writeln('A New elõtt         ',memavail);
   New(ap);
   writeln('A Mark elõtt        ',memavail);
   Mark(mp);
   New(bp);
   New(cp);
   Writeln('A Release elõtt     ',memavail);
   Release(mp);
   Writeln('A Release után      ',memavail);
   Dispose(ap);
   Writeln('A Dispose után      ',memavail);
end.
```

A program az alábbi eredményt adja, ahonnan jól leolvasható a *heap* kezelése a ***mark*** és ***release*** eljárások használatával:

```
A New elõtt       599232
A Mark elõtt      579232
A Release elõtt   539232
A Release után    579232
A Dispose után    599232
```

12.2.3.2. A Getmem és a Freemem eljárások használata

A Turbo Pascal lehetőséget biztosít **pointer** típusú általános mutató deklarálására. Az így létrehozott mutató nem rendelkezik konkrét típussal, ezért a *New* eljárást nem használhatjuk a memóriafoglalás elvégzésére. Ebben az esetben a dinamikus memóriahasználatot a *Getmem* és a *Freemem* eljárások hívásával végezhetjük el:

Getmem **(var *p*:pointer; *size*:word);**

illetve

Freemem **(var *p*:pointer; *size*:word);**

A megadható maximális blokkméret 65521 (64K - $F) byte. Ha a foglalásnál nem áll rendelkezésre elegendő szabad *heap*-terület, akkor a program 203-as hibával leáll.
A ***Getmem*** és ***Freemem*** eljárások használhatók típussal rendelkező mutatók esetén is, ahol a szükséges méretet a **sizeof**(típus) hívás szolgáltatja.

```
program getmem_freemem;
const n=10000;
type arr= array[1..n] of longint;
var p  : pointer;
    q  : ^arr;
    i  : integer;
begin
   { A helyfoglalások elvégzése }
   getmem(p,n*sizeof(longint));
   getmem(q,sizeof(arr));

   { A q^ tömb feltöltése }
   for i:=1 to n do q^[i]:=i;
   writeln(q^[n div 2]);

   { A q^ tömb tartalmának átmásolása a p^
    területre típuskonverzió felhasználásával.}
   arr(p^):=q^;

   { A q^ tömb nullázása }
   for i:=1 to n do q^[i]:=0;
   writeln(q^[n div 2]);
```

```
        { A q^ tömb eredeti tartalmának visszamásolása a
          p^ területrõl a move  eljárás felhasználásával.}
        move(p^,q^,sizeof(arr));
        writeln(q^[n div 2]);

       { A lefoglalt területek felszabadítása.}
        freemem(p,n*sizeof(longint));
        freemem(q,sizeof(arr));
      end.
```

A **pointer** típusú mutatót legtöbbször dinamikus pufferterületek kijelölésére használjuk ahol a kívánt méret a program futása során kerül meghatározásra, mint ahogy ezt a grafikus ablakterület kezelése szemlélteti.

```
program grafikus_puffer;

 uses  graph;
 var
   gd, gm : Integer;
   p      : pointer;
   meret  : word;

 begin
   { A grafika bekapcsolása ellenõrzésse. }
   gd := detect; initgraph(gd, gm, 'c:\tp55');
   if graphresult <> grok then halt(1);

   { A képernyõ törlése. }
   setfillstyle(solidfill,white);
   bar(0, 0, getmaxx, getmaxy);

   { A (100,50) - (300,200) terület kezelése.}

   meret:= magesize(100,50,300,200);
          { A szükséges méret.  }
   getmem(p, meret);
    { Helyfoglalás.         }
   getimage(100,100,300,250,P^);
          { A terület lementése.}

   setcolor(black);
          { Rajzolás. }
   ellipse( 200,175,0,360,100,75);
   readln;

   putimage(100,102, P^, NormalPut);
```

```
                  { A lementett terület }
          readln;
            { visszatöltése.        }
          freemem(p,meret);
            { Felszabadítás.        }

        { A grafika kikapcsolása. }
          closegraph;
          restorecrtmode;
        end.
```

12.2.3.3. A mutatókról bővebben

Mint ahogy láttuk, a Turbo Pascal-ban egyaránt definiálhatunk típusos és konkrét típus nélküli mutatókat:

```
var   ip : ^byte;      { típusos }
      p  : pointer;    { típus nélküli }
```

Mindkét esetben értelmezettek az alábbi műveletek:

- Értékadás:

Értékadás a mutató által definiált objektumnak:

```
ip^:=4;
```

Értékadás magának a mutatónak:

```
 p :=nil;
ip:=p;
```

- Összehasonlítás:

Azonos típusú mutatók esetén használhatók az = és a <> műveletek. A **nil** konstansmutató és a **pointer** típussal deklarált mutatók bármel más mutatóval összehasonlíthatók.

```
if (ip=p) and (ip<>nil) then halt;
```

- A mutató típusa Turbo Pascal típuskonverzióval megváltoztatható. Ily módon **pointer** mutatókhoz is rendelhetünk típust.

a., Az *arr* típusú mutató által kijelölt területre **longint**-ként hivatkozunk:

```
type
    arr=array[1..4] of byte;
 var
    ap:^arr;
 begin
   new(ap);
   longint(ap^):=$12345678;
   {A fenti értékadás megfelel a }
   {ap^[1]:=$78;ap^[2]:=$56;
    ap^[3]:=$34;ap^[4]:=$12;}
   { értékadások sorozatának }
   dispose(ap);
 end.
```

b., A **pointer** mutatót számára lefoglalt blokkot a *tomb* típus segítségével **integer** egységekben érjük el:

```
const
    n=1000;
 type
    tomb=array[1..n] of integer;
 var
    p:pointer;
    i:integer;
 begin
    getmem(p,n*sizeof(integer));
    for i:=1 to n do
      tomb(p^)[i]:=i;
   freemem(p,n*sizeof(integer));
 end.
```

- A mutatók megadhatók eljárás és függvény paraméterként, vagy függvény visszatérési értékként egyaránt:

```
program pointer_pelda;
 var ip:^integer;

 { Ellenõrzött helyfoglalás }
 function fnew(meret:word):pointer;
 var p:pointer;
 begin
   if meret>maxavail then fnew:=nil
                     else
                       begin
                         getmem(p,meret);
                         fnew:=p;
                       end;
 end;

 procedure fdispose(p:pointer; meret:word);
 begin
   freemem(p,meret);
 end;

 begin
  ip:=fnew(sizeof(integer));
  if ip=nil then halt(1);
  {...}
  fdispose(ip,sizeof(integer));
 end.
```

- Az ***Addr*** függvény és a @ operátor

Az ***Addr*** függvény illetve az ennek megfelelő @ operátor segítségével bármely Pascal objektum (függvény, változó, ...) címe lekérdezhető, és tetszőleges típusú mutatóhoz hozzárendelhető.

```
var
   ip : ^integer;
   i : integer;

 begin
   i :=5;
   ip:=@i; { vagy ip:=Addr(i); }
   ip^:=i+6;
   writeln(i);
   { A program által kiírt eredmény: 11 }
 end.
```

- Tetszőleges Pascal objektum címének szegmens és offszet része egyaránt lekérdezhető a ***Seg*** és az ***Ofs*** függvények hívásával. A MEMORIA.PAS program a futó program néhány jellemző memóriaadatát jeleníti meg:

```
program memoria;

 var
    x               : Integer;
    off, segm       : Word;
    ds, cs, ss, sp  : Word;
    p               : ^Integer;

{ max.32 bites egész szám(w) konvertálása pos darab }
{ jegyet tartalmazó hexadecimális sztringgé         }
 function HexStr(w : longint; pos:byte):string;
   const hex: array [0..$F] of Char
              ='0123456789ABCDEF';
   var    r : record
                case byte of
                  0: ( w:longint);
                  1: ( a:array[1..4] of byte);
              end;
        res : string[10];
   begin
     res:='';
     repeat
        r.w:=w;
        res:=hex[r.a[1] and $0f]+res;
        w:=w shr 4;
        dec(pos);
     until pos<1;
      hexstr:='$'+res;
   end;

 begin
    x    := 30000;
    off  := Ofs(x);
    segm := Seg(x);

    Writeln( 'Az x szegmenscíme: ',hexstr(segm,4));
    writeln( '     offszetcíme : ',hexstr(off,4) );
    p := Ptr( segm, off );

    Writeln('A ',hexstr(longint(p),8),'
            ' címen tárolt egész: ',p^ );
    cs := CSeg;
```

```
        ds := DSeg;
        Writeln;
        Writeln( 'Kódszegmens   CS:  ',hexstr(cs,4));
        Writeln( 'Adatszegmens  DS:  ',hexstr(ds,4));
        ss := SSeg;
        sp := SPtr;
        Writeln( 'Veremszegmens SS:  ',hexstr(ss,4));
        Writeln( 'Veremmutató   SP:  ',hexstr(sp,4));
        Writeln;
    end.
```

A program futásának eredménye az alábbiakban látható:

```
Az x szegmenscíme: $099F
      offszetcíme: $004E
 A $099F004E címen tárolt egész: 30000

 Kódszegmens   CS:  $08BD
 Adatszegmens  DS:  $099F
 Veremszegmens SS:  $09CA
 Veremmutató   SP:  $3EFE
```

Külön felhívnánk a figyelmet a *hexstr* konverziós rutinra, amely a $0..$FFFFFFFF tartományon képes egész számokat hexadecimális sztringgé konvertálni.

Ha egy memóriacímet ismerünk szegmens:offszet alakban, akkor erre a címre rámutathatunk egy pointerrel amelyet a ***Ptr*** függvény szolgáltat. A példában a billentyűzetpuffer törlésére mutatunk megoldást:

```
procedure clearkbd;
 var
   headp,tailp:^word;
 begin
   headp:=ptr(0,$41a);
   tailp:=ptr(0,$41c);
   headp^:=$1e;
   tailp^:=$1e;
 end;
```

12.2.3.4. Az absolute deklaráció

Az előző alfejezetekben a memória elérését mutatók felhasználásával mutattuk be. Az abszolút deklaráció felhasználásával tetszőleges Pascal változót a memória tetszőleges címére helyezhetünk. A deklarációt az alábbi két alakban használhatjuk:

var *azonsoító* : *típus* **absolute** *szegmenscím: offszetcím*;

vagy

var *azonosító* : *típus* **absolute** *változó*;

Az első alak alkalmazása esetén a deklarált változó a memória *szegmenscím:offszetcím* címén kerül elhelyezésre, míg a második esetben már meglevő változóra definiáljuk rá az új változót.
Nézzünk egy-egy példát a két alak felhasználására:

a., A színes szöveges képernyőpuffert a *screen* tömb képernyőterületre való rádefiniálásával érjük el (a 128-nál nagyobb kódú karaktereket pontra cseréljük, illetve az attribútum byte villogás bitjét invertáljuk):

```
type elem   =record
                ch:char;
                at:byte;
              end;
     scrtype=array[1..25,1..80] of elem;

var  screen :scrtype absolute $b800:0000;
     s,o    :byte;

begin
   for s:=1 to 25 do
     for o:=1 to 80 do
       begin
         iford(screen[s,o].ch)>128 then
              screen[s,o].ch:='.';
         screen[s,o].at:=screen[s,o].at xor $80;
       end;
end.
```

b., A második alak lehetővé teszi különböző típusú változók konvertálását. A példában a **pointer** <--> **longint** konverziót használjuk, a színes szöveges képernyő jobb felső sarkában villogó A betű megjelenítésére:

```
var l:longint;
    p:pointer absolute l;
begin
   l:=$b800009E;
   char(p^):='A';
   inc(l);
   byte(p^):=$8f;
end.
```

12.2.3.5. A memória és a portok közvetlen elérése

A Turbo Pascal rendelkezik öt előredefiniált tömbbel: ***Mem***, ***MemW***, ***MemL***, ***Port***, ***PortW***.

Mem[*szegmens:offszet*]
MemW[*szegmens:offszet*]
MemL[*szegmens:offszet*]

A ***Memx*** tömbök segítségével tetszőleges memóriacímen található byte (***Mem***), szó (***MemW***) illetve duplaszó (***MemL***) típusú adatelem elérhető. A tömbök indexei eltérnek a szokásos indexektől mivel *szegmenscím:offszetcím* alakúak. Példaként tekintsük a billentyűzetpuffer törlését, amely egyszerűen elvégezhető a ***MemW*** tömb felhasználásával:

```
procedure clearkbd;
begin
  memw[0:$41a]:=$1e;
  memw[0:$41c]:=$1e;
end;

{CAPSLOCK billentyû bekapcsolása
 a $40:$17 címentalálható }
{byte 6.bitjének 1-beállitásával. }
Mem[$40:$17] := Mem[$40:$17] OR $40;
```

```
{ Az éjfél óta eltelt idõ (1/18 mp) le
  kérdezése longint típusú változóba }
ticks := MemL[$40:$6C];

{ $2F attributumú, $01 kódú karakter el
  helyezése a képernyõ bal felsõ arkán }
MemW[$B800:2] := $2F21;
```

Port [*portszám*]
PortW [*portszám*]

A ***Port*** és a ***PortW*** tömbök segítségével a perifériák közvetlenül programozhatók illetve vezérelhetők. Ebben az esetben az index **word** típusú és lehetséges értékei IBM PC számítógépen 2..1023.
Nem szabad elfelednünk, hogy a portok közvetlen használata potenciális veszély rejt magában, ezért mindig körültekintően kell eljárnunk:

```
{ byte olvasása a $64 portról }
 writeln(Port[$64]);

{ word írása a $B6 portra:ez a mûvelet párhuzamosan }
{ irja a $B6 és a $B7 portokat rendre $22 illetve $11 }
{ értékekkel.                                          }
 PortW[$B6] := $1122;
```

12.2.3.6. Típus nélküli paraméterek használata

Nagyon hasznos lehetősége a Turbo Pascal-nak a típus nélküli paraméterek használata az eljárások és a függvények paraméterlistájában. Ha a változó-paraméter típusát elhagyjuk, akkor azt típus nélküli paraméterként kezeli a rendszer. Általában akkor használjuk ezt a lehetőséget, amikor a paraméterek típusa nem lényeges, például memória blokkok másolásakor (***blockread***, ***blockwrite***, ***move***), vagy más memóriablokkra vonatkozó műveletnél (***fillchar***).
Nézzünk néhány példát a típus nélküli paraméterek felhasználására:

a., A *swapvar* eljárás tetszőleges típusú változók értékének felcserélésére használható:

```
procedure swapvar(var a,b; size:longint);
type
   tp = array [1..maxint] of byte;
var
   v1 : tp absolute a;
   v2 : tp absolute b;
    i : longint;
   tmp: byte;
begin
   for i:=1 to size do
     begin
       tmp    := v1[i];
       v1[i]  := v2[i];
       v2[i]  := tmp;
     end;
end;
```

b., Tetszőleges méretű egész elemeket tartalmazó négyzetes mátrixok összegzése. A $R dírektívával lokálisan letiltjuk az indexhatár ellenőrzését.

```
procedure SumMatr(var a,b,c; n:integer);
var
   x : array[1..1] of integer absolute a;
   y : array[1..1] of integer absolute b;
   z : array[1..1] of integer absolute c;
   i,j : integer;
begin
{$R-}
    for i:=0 to n-1 do
      for j:=1 to n do
          z[i*n+j]:=x[i*n+j]+y[i*n+j];
{$R+}
end;
```

c., A ***move*** és a ***fillchar*** rutinok hiányzó társa, a memóriaterületek összehasonlítását végző ***compare*** eljárás az alábbiakban látható. A hivatkozáshoz a típuskonverziót használtuk:

```
function compare(var source, dest; size:word):boolean;
type
    barr = array[0..maxint] of byte;
var
  n:integer;
```

```
    begin
    n:=0;
    while(n<size) and (barr(dest)[n]=barr(source)[n]) do
         inc(n);
    compare :=  n=size;
  end;
```

A Turbo Pascal több olyan eljárást és függvényt tartalmaz, amelyek tetszőleges típusú változóparaméterrel hívhatók. Ezek közül csak a fejezetünkhöz szorosan kapcsolódó ***move*** és ***fillchar*** eljárások használatát tekintjük át. A két eljárás deklarációja az alábbi fejléceket tartalmazza:

procedure *move*(**var** *source, dest*; *count*:**word**);

procedure ***fillchar***(**var** *dest; count*:**word**; *ch*:**char**);

A ***move*** eljárás a *source* címmel azonosított memóriaterületről *count* darab byte-ot átmozgat a *dest* címmel megadott területre. A mozgatás átfedésben levő területek esetén is helyesen működik.

A ***fillchar*** eljárás a *dest* címtől kezdve *count* darab byte-ot *ch* karakterrel tölt fel. Jól használható ez az eljárás például tömbök nullázására.

A fenti eljárások használatát bemutató programban felhívjuk a figyelmet a *source* és a *dest* paraméterek különböző megadási lehetőségeire.

A program a színes szöveges képernyőn végez műveleteket:

```
program move_fillchar;
 type
    scrt    = array[1..25,1..80] of word;
 var sp     : ^scrt;
     screen : scrt absolute $b800:0;
     i      : integer;
 begin
   new(sp);

   {A szöveges képernyő lementése az sp^ területre.}
   move(mem[$b800:0],sp^,sizeof(sp^));
   readln;

   { A szöveges képernyő törlése.}
```

```
    fillchar(screen,sizeof(scrt),0);
    readln;

    { A lementett képernyõ visszatöltése.}
    move(sp^,screen,sizeof(scrt));
    readln;

    dispose(sp);
  end.
```

Ellenőrző kérdések:

1. Mit tartalmaz a mutató típusú változó?
2. Milyen adatokat tárol a Turbo Pascal az adatszegmensben, a *stack*-en és a *heap*-en?
3. Ismertesse a dinamikus tárkezelés lényegét!
4. Mi a különbség a ***new/dispose*** ill. a ***GetMem/FreeMem*** eljárások használata között?
5. Tekintsük az alábbi deklarációt

```
type
     trec = record
          ido : longint;
          adat: real;
        end;

     trp= ^trec;

var
     r  : trec;
     rp : trp;
     rpp: ^trp;
```

   Mit tartalmaznak az *r*, az *rp* és az *rpp* változók?

6. Hogyan lehet közvetlen memóriát címezni Turbo Pascal programból?

7. Hol használunk típus nélküli paramétereket?

Feladatok:

1. Írjon programot, amely 10 elemű valós tömböt használ dinamikusan és az elemeket feltölti az első tíz egész szám négyzetgyökével.

```
type
     aptr= ___________ ;          { 10 elemu valos tomb }

var
     a: ___________;              { mutato a tombhoz }
     j: integer;
begin
     new(___);                    { helyfoglalas a tomb szamara}
     for j:=1 to 10 do
         _____:=sqrt(j);          ( ertekadas a tombelemeknek}
     for j:=1 to 10 do
         writeln(_____ );         { a tombelemek kiirasa  }

     dispose(___);                { a tomb felszabaditasa )

end.
```

(TPTR.PAS)

2. Értékelje ki a PAROS.PAS programot, amely két nevet olvas és a párokat láncba fűzi. Az adatok bevitelének végét *Enter* jelzi. Sematikusan rajzolja fel a program által létrehozott adatszerkezetet!

3. Definiáljon **absolute** tömböt a színes, szöveges képernyőre. Írjon modult (**unit**-ot) , amely képes a képernyő tartalmát file-ba menteni, illetve onnan betölteni!
(KEPFILE.PAS)

13. A CRT UNIT HASZNÁLATA

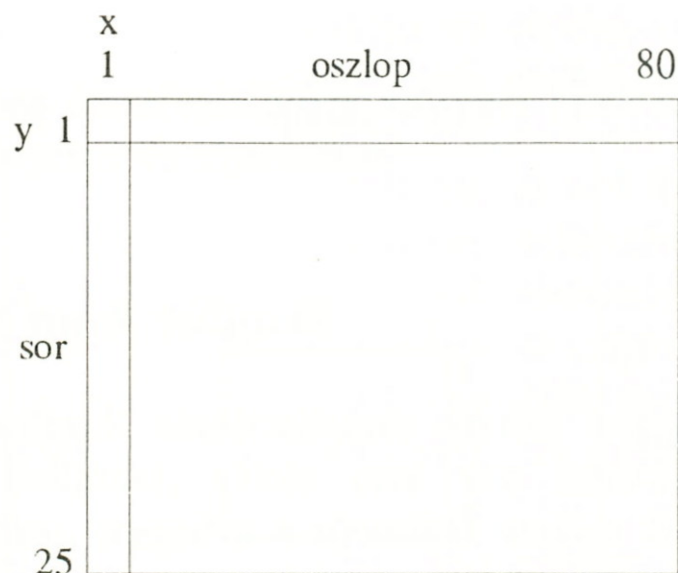
13.1. A szöveges üzemmód

Az számítógép a képernyőt alapértelmezés szerint szöveges módban használja. A képernyőablak (*Window*) négyzethálóhoz hasonlóan sorokból és oszlopokból áll. Alapértelmezés szerinti ablak 25 sort és egy sorban 80 oszlopot tartalmaz. Ez azt jelenti, hogy 80 karaktert írhatunk egy sorba, a 80 feletti karakterek átkerülnek a következő sorba. Ha viszont 25 sornál többet írunk, akkor a 25. sor után amennyi sor megjelenik a képernyő alján, annyi sor tünik el a képernyő tetejéről. A képernyőből kilépő sorokat nem lehet később megtekinteni, mondhatjuk úgy is, hogy az információ kiszaladt a képernyőből (görgetés - scroll). A program írójának kell gondoskodnia arról, hogy a program eredménye a képernyőn kiértékelhető legyen. Ahhoz, hogy az eredmény színesen, és irányítottan adott oszlopba és sorba kerüljön, használnunk kell a *Crt* unit eljárásait és függvényeit.

A *Crt* unit ***Window*** eljárása a szöveges képernyőn aktív ablakot definiál. Az eljárás szintaxisa:

Window(*x1,y1,x2,y2*);

ahol az *x1,y1,x2,y2* **byte** típusú paraméterek.



13.1. ábra Window(1,1,80,25)

Az *x1,y1* az ablak bal felső sarkának, az *x2,y2* pedig a jobb alsó sarkának a koordinátái. Ha az alapértelmezést használjuk, akkor nem kell az eljárást hívnunk, mert `Window(1,1,80,25)` a teljes képernyőablakot jelenti. Az 13.1. ábrán látható, hogy a képernyő x irányú koordinátái az oszlopokat, az y irányú koordinátái a sorokat jelentik.

Az alapértelmezés szerinti ablakméret 80x25 mellett használható a 40x25, illetve EGA monitor esetén 80/40x43, VGA monitor esetén 80/40x50 is.
Például:

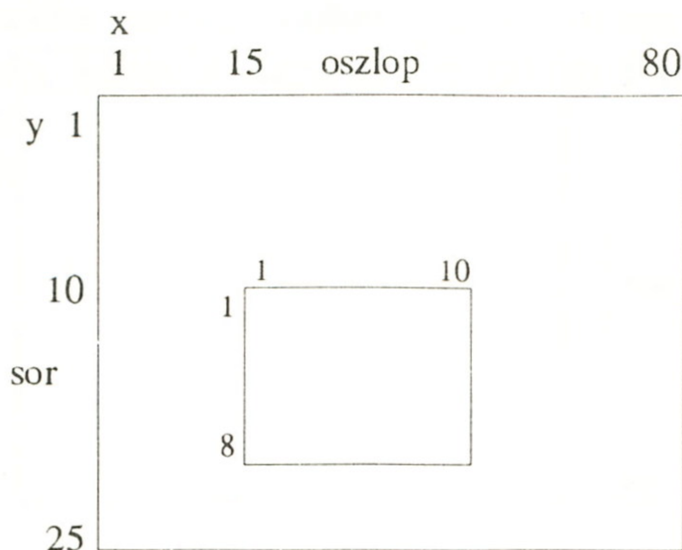
```
Window(1,1,80,25);        Window(1,1,40,25);
Window(1,1,80,43);        Window(1,1,40,43);
```

VGA monitornál: `Window(1,1,80,50); Window(1,1,40,50);` is lehet.

Ablakban ablakot is létrehozhatunk. A ***Window*** eljárás koordinátáit abszolút koordinátákkal kell megadni a fő ablak (1,1) koordinátájához képest.
Példaként hozzunk létre 10 oszlopot és 8 sort tartalmazó ablakot a fő ablak (15,10) koordinátapontjában. Az ablak bal felső koordinátái a beültetési koordináták, a jobb alsó kordinátái pedig a beültetési koordinátákhoz hozzáadva az ablak szélességét és magasságát, az eljárás hívása a következő:

```
Window(15,10,25,18);
```

Az 13.2. ábra mutatja a (15,10) pontban létrejött 10,8 méretű ablakot. Az aktív ablak bal felső sarokpontja (1,1).



13.2. ábra Window(15,10,25,18)

A szöveges módban a *Crt* unit függvényeivel és eljárásaival állíthatjuk a háttér és a karakterek színét. Az adott pozícióra írhatunk. Megszólaltathatjuk a belső hangszórót, adott időre felfüggeszthetjük a program futását. A képernyőn bárhol definiálhatunk ablakot, sort írhatunk, törölhetünk és szúrhatunk be. Lekérdezhetjük a kurzor pozícióját. A programon belül a grafikus módot szöveges móddal oda/vissza lehet váltani (lásd a 14. fejezetet). A *Crt* unit-ot akkor érjük el, ha a programban a ***uses*** kulcsszó mellett definiáljuk:

```
program crt_probak;
{ crt könyvtár hivatkozása }
uses crt;
```

Először tekintsük át a szöveges módva vonatkozó információkat, a használható eljárások és függvények rövid leírását. Az F2 függelékben találjuk meg a teljes leírást.

A 13.3. részfejezetben ismertetett mintaprogramok mutatják be a *Crt* unit alkalmazását.

13.2. Szöveges mód változói, konstansai, függvényei és eljárásai

Ebben a részfejezetben összefoglaljuk a szöveges módban használt változókat, konstansokat, függvényeket és eljárásokat.

13.2.1. Szöveges mód változói

Az szöveges mód változói alapértelmezés szerint kapnak értéket, ha ettől eltérőt szeretnénk beállítani, akkor erre van lehetőség. Az alábbiakban ismertetjük a változókat, megadva a típusúkat majd a funkciójukat is.

A ***Crt* unit** változói:

var

```
    CheckBreak     : Boolean;
    CheckEOF       : Boolean;
    DirectVideo    : Boolean;
    CheckSnow      : Boolean;
    LastMode       : Word;
    TextAttr       : Byte;
    WindMin        : Word;
    WindMax        : Word;
```

A ***CheckBreak*** logikai változó szabályozza a program CTRL Break-kel való megszakítását. Ha a változó értéke *true*, a program megszakítható, *false* esetén nem. Az alapértelmezés a *true*.

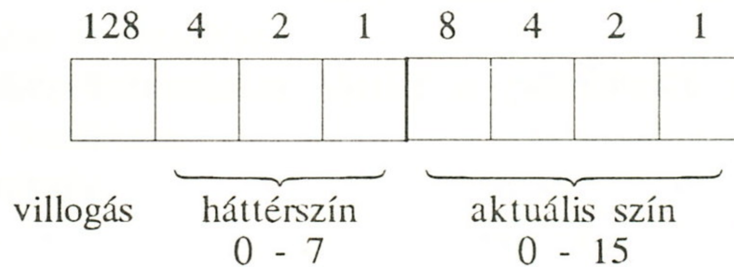
A ***CheckEOF*** logikai változó engedélyezi vagy nem engedélyezi *CTRL-z* határása az *end-of-file* (file-vége) jel írását. Ha a ***CheckEOF*** logikai változó *true*, akkor a képernyőhöz rendelt file-ból való olvasáskor *CTRL-z* leütésére file vége esemény generálódik, ha a változó értéke *false*, akkor nem. Az alapértelmezés *false*.

A ***CheckSnow*** változó az ún. havazás jelenséget szabályozza a CGA monitoroknál. Ez a havazás nem jelentkezik a monokróm vagy EGA, VGA monitorok esetében. Alapértelmezés szerint *true*-ra van állítva. Ha nem CGA monitoron futtatunk a ***TextMode*** állítása után érdemes a ***CheckSnow*** változót *false*-ra állítani, mert jelentősen gyorsabb outputot eredményez. A havazás nem hatásos, ha a ***DirectVideo*** értéke **false**.

A ***DirectVideo*** változó *true* értéke biztosítja a ***write*** és ***writeln*** eljárásoknak a képernyőhöz való közvetlen hozzáférését oly módon, hogy az output közvetlenül a képernyőmemóriában történik. A képernyőkezelés így gyorsabb, mint a **BIOS**-on keresztül. A ***DirectVideo*** változó alapértelmezés szerint *true*-ra van állítva. *False*-ra állítjuk akkor, ha a BIOS-on keresztül akarjuk a karakterek megjelenítését, az értékadást csak a ***TextMode*** hívása után lehet végrehajtani.

A ***LastMode*** változó tartalmazza az utóljára kiválasztott szöveges módot. A rendszer program indulásakor az aktív videomódot kódját tölti be a ***LastMode*** változóba.

A *TextAttr* tárolja az aktuálisan kiválasztott szövegszínt, amelyet a *TextColor* és a *TextBackground* állított. Ezt a változót közvetlenül is beállíthatjuk.
A **byte** tartalma a következő



A *WindMin* és *WinMax* változók az aktuális ablak képernyőkoordinátáit tárolják. Az értéküket akkor veszik fel, amikor a *Window* eljárást hívtuk. A *WinMin* változó tartalmazza az ablak bal felső, a *WinMax* tartalmazza a jobb alsó sarok koordinátáit. Az X-koordináta az alsó byte-ban, az Y-koordináta a felső byte-ban kerül tárolásra.

13.2.2. Szöveges mód konstansai

A *TextMode* eljárás paramétere az alábbi konstansokat fogadja:

```
const
    BW40     = 0;     { 40x25  fekete-fehér }
    CO40     = 1;     { 40x25  színes       }
    BW80     = 2;     { 80x25  fekete-fehér }
    CO80     = 3;     { 80x25  színes       }
    Mono     = 7;     { 80x25  monkróm MDA vagy Hercules
                               monitoron    }
    Font8x8  = 256;   { 40x43, 40x50 sor EGA és VGA
                               monitorokon  }
```

A *TextMode* aktiválásával állíthatunk fekete/fehér ill, színes módot a megfelelő ablakmérettel.

Néhány példa a hívásra:

```
TextMode(0);        fekete-fehér, 40x25
TextMode(BW80);     fekete-fehér, 80x25
TextMode(CO40);     színes, 40x25
```

A szöveg színét a ***TextColor,*** a háttér színét a ***TextBackGround*** eljárás állítja.

A színkonstansok:

```
const
        Black           = 0;    { fekete          }
        Blue            = 1;    { kék             }
        Green           = 2;    { zöld            }
        Cyan            = 3;    { türkiz          }
        Red             = 4;    { piros           }
        Magenta         = 5;    { lila            }
        Brown           = 6;    { barna           }
        LightGray       = 7;    { világosszürke   }
        DarkGray        = 8;    { sötétszürke     }
        LightBlue       = 9;    { világoskék      }
        LightGreen      = 10;   { világoszöld     }
        LightCyan       = 11;   { világos türkiz  }
        LightRed        = 12;   { világospiros    }
        LightMagenta    = 13;   { világoslila     }
        Yellow          = 14;   { sárga           }
        White           = 15;   { fehér           }
        Blink           = 128;  { villog          }
```

A színekből a sszöveg írására mindegyiket (0-14) használhatjuk. A háttérre azonban csak 0-tól 7-ig használhatunk színeket. Legyen a háttér színe türkíz, a szöveg színe kék. A eljárások számkonstanssal vagy színkonstanssal is hívhatók:

```
TextBackGround(3); vagy TextBackGround(Cyan);
TextColor(1); vagy TextColor(Blue);
```

13.2.3. Szöveges mód eljárásai és függvényei

A szöveges mód függvényeit és eljárásait öt csoportba sorolhatjuk:

- szöveg kiírása és kezelése,
- ablak és mód vezérlése,
- tulajdonság beállítása,
- állapotlekérdezés,
- hangkeltés.

SZÖVEG KIÍRÁSA ÉS KEZELÉSE

Szöveg írás és olvasása

AssignCrt	szöveg file-t rendel a CRT eszközhöz (képernyőhöz),
Write	írás a képernyőre,
Read	olvasás billentyűzetről,
ReadKey	egy karakter beolvasására vár.
KeyPressed	jelzi, ha van lenyomott billentyű.

Szöveg és kurzor mozgatása a képernyőn

ClrScr	törli a szöveges képernyő ablakot,
ClrEol	törli a sort a kurzor pozíciójától,
DelLine	törli a kurzort tartalmazó sort,
GotoXY	pozícionálja a kurzort,
InsLine	beszúr egy üres sort azon sor alá, ahol a kurzor áll,

ABLAK ÉS MÓD VEZÉRLÉSE

TextMode	beállítja a képernyőt a kívánt szöveges módba,
Window	definiál egy képernyőablakot.

TULAJDONSÁGOK BEÁLLÍTÁSA

A szöveges módban a tulajdonságok beállítását az alábbi eljárások végzik:

Az előtér (írás) és a háttér színének beállítása

TextColor	beállítja az előtér színét, ez lesz a karakter színe,
TextBackground	beállítja a háttér színét.

Intenzitás beállítása

HighVideo	magas fényű intenzitás beállítása,
LowVideo	alacsony fényű intenzitás beállítása,
NormVideo	az eredeti, normál intenzitás beállítása.

ÁLLAPOTLEKÉRDEZÉS

WhereX	megadja a kurzor helyzetének x koordinátáját,
WhereY	megadja a kurzor helyzetének y koordinátáját.

HANGKELTÉS

Sound	adott Hertz frekvenciájú hangot generál a belső hangszórón,
NoSound	kikapcsolja a hangszórót.

13.3. A szöveges mód programozása

13.3.1. Mintaprogramok a Crt unit használatára

A CRT1.PAS program ablakot definiál, amelyben a szöveg kiírása mellett jelzi, hogy a szöveg utolsó karaktere mely oszlopban és sorban keletkezik. A ***write*** és ***writeln*** eljárással kerül a szöveg kiírásra. A kuzor pozíciójának koordinátái lekérdezhetőek, az oszlopot a *WhereX* függvény, sort a ***WhereY*** adja vissza. A ***Delay*** eljárásban másodpercben megadott érték felfüggeszti a program futását. Általában azért teszünk a programban ilyen jellegű

képernyő kimerevítést, hogy a szemünk észrevegye a változást. A változás a gép gyorsasága miatt olyan gyorsan következik be, hogy a képet éppen csak egy villanásra láthatjuk.
Ezt a késleltető módszert a többi mintaprogramban is alkalmazzuk.

A CRT2.PAS program ***CrlEol*** eljárás használatát mutatja be, amely a kurzor pozíciójától töröl a sor végéig.

A CRT3.PAS programban a ***DelLine*** eljárással a kurzort tartalmazó sort töröljük és az ***InsLine*** segítségével sort szúrunk be.

A CRT4.PAS a ***KeyPressed*** függvényt használja, amely billentyűleütést figyel. A képernyőn addig fut a 'Barmilyen billentyu leutesere var' szöveg kiírása, amíg valamilyen billentyűt le nem ütünk.

A DALLAM.PAS a belső hangszóró megszólaltatását mutatja be. Egy hang megszólaltatása három eljárás hívást jelent.

```
Sound(440);     { a 440 Hz A hang megszólaltatása }
Delay(250);     { a hang kitartása 250 ms-ig      }
Nosund;         { a hangszóró kikapcsolása        }
```

A VILLOG.PAS program több ablakot nyit, az ablaknak más színű hátteret rendel. Értékeljük ki az alábbi programrészletet:

```
(* Teljes ablak hattere *)
TextBackGround(LightGray);
ClrScr;
(* elso ablak *)
Window(10,10,20,15);
TextBackGround(Cyan);
ClrScr;
TextColor(Red);
Writeln('1. Sor ');
Writeln('2. Sor ');
Writeln('3. Sor ');
Delay(1000);
GotoXY(1,1);
TextColor(Yellow+Blink);
Write('felulir');
TextColor(Black);
```

```
Window(30,23,50,24);
Write('Nyomj Enter-t!'); readln;
```

Az alapételmezés szerinti 80x25 méretű ablak hátterét világosszürkére színezi a **ClrScr** eljárás hívása. Ez a függvény a képernyőt törli, és az attribútum byte-ot a beállított színekkel tölt fel. A fő ablak (10,10) koordinátapontjában létrehozunk egy ablakot, amely 10 oszlopot és 5 sort tartalmaz. A háttér színét türkízre állítja be a *TextBackGround* és a **ClrScr** eljárás végzi el az aktív ablak kiszínezését. Az írás színét a *TextColor* pirosra állítja be. A kurzor az aktív ablak bal felső sarkában villog, amely az (1,1) koordinátapontot jelenti. A `Writeln('1. Sor ');` piros színű kiírása ezért az aktív ablak első sorába kerül, majd a kurzor a következő sor elejére áll. Ugyanígy íródik ki a *2. Sor* ill. a *3. Sor* az ablakban. 1000 ms késleltetés után a kurzor a `GotoXY(1,1)` hatására az ablak első pozíciójára áll, az írás színe villogó sárgára vált:

```
SetColor(Yello+Blink);
```

hívás hatására. A `Write('felulir');` végrehajtásakor az *1.Sor* szöveg valójában felülíródik. Majd a program feketére váltja az írás színét és egy új ablakban kiírja a *Nyomj Enter-t* figyelmeztető szöveget. Egy paraméternélküli ***Readln;*** utasítás felfüggeszti a program futását, míg meg nem nyomjuk meg az ENTER billentyűt.

A program az alábbi utasításokkal fejeződik be:

```
(* az ablak vissaállítása *)
TextBackGround(black);
Window(1,1,80,25);
ClrScr;
TextColor(LightGray);
```

Ezek az utasítások gondoskodnak a fekete háttérről, az ablak visszaállításáról, valamint az írás színéről, különben visszatérve a DOS rendszerbe, a háttér és az írás színe megmarad.

SCREENT.PAS program bemutatja, hogyan lehet a képernyőt két részre osztani.

13.3.2. Adat beolvasása és ellenőrzése

A CRT_PR.PAS program adat beolvasásán és ellenőrzésén keresztül mutatja be a szöveges mód eljárásainak és függvényeinek használatát.

A program az alábbi feladatot hajtja végre:

- fejlécet ír ki,
- kéri az adat beolvasását,
- megvizsgálja, hogy az adat valóban **real** típusú-e,
- ha a beolvasott adat hibás, akkor a beolvasott adat mellett piros villogó *Hibás adat!* és sárga színű *Nyomj Space-t* üzenet jelenik meg.
  Ilyenkor a Space billentyű leütése után az adatot újra beolvashatjuk.
- Ha a megadott számadat formailag jó, lehetőség van a javítására, ha az *Akarja javitani (i/n):* kérdés mellett i betűt ütünk, egyébként n-et. A válasz után az üzenetsor törlődik és a beolvasás elölről kezdődik az újbóli adat ellenőrzésével.

A program képernyője a következőképpen néz ki:

```
Adatbeiras billentyuzetrol

Kerem az adatot:  s    Hibas adat: Space

Akarja javitani (i/n):
```

A program megtervezése
A program változói:

a **real**, a jó adatot tartalmazza,
sz **string**, először sztringbe olvasunk az adat ellenőrzése miatt,
code **integer**, a ***val*** eljárás ebbe a változóba jelzi, hogy az adat megfelel-e a kért adattípusnak.
x,y **byte**, a beolvasott adat melletti koordinátákat tartalmazza,
ch **char**, ebbe a változóba olvassa be a ***ReadKey*** függvény az *i* vagy az *n* válaszkaraktert.

A program listájának magyarázata:

Eltérve az alapértelmezéstől a színes 40 oszlopos szöveges módot választjuk ki:

```
TextMode(C40);
```

Ahhoz, hogy a teljes képernyő színe kék legyen, először kiválasztjuk a `TextBackgorund(blue)` eljárással a háttérszínt, majd töröljük a képernyőt a `ClrScr` eljárással, ezután veszi fel a háttér a kivánt színt.

A `TextColor(Magenta)` az írás színére a lilát választja ki. A `GotoXY(2,5)` a kuzort a második oszlop, ötödik sorába helyezi át és a

```
write('Adatbeolvasas billentyuzetrol')
```

szöveg a kurzor pozíciójától lila színnel jelenik meg.

Két egymásbaágyazott `repeat until` ciklust alkalmazunk az adat vizsgálatára. A `2. repeat until` ciklusból addig nem lépünk ki, amíg a ***val*** eljárás *code* változója nulla nem lesz, vagyis csak akkor, ha a számadat megfelelő formátumú. Az első `repeat until` ciklus csak akkor tér vissza, ha az adatbeolvasáskor kapott adatot javítani akarjuk és *i* választ adtunk, különben a program befejezi a működését.

Fehér színnel a 2. oszlop 10. sorában kerül kiírásra az

Adat:

szöveg és mögötte feltétel nélkül töröljük a sort (*ClrEol*). Sárga színre állítjuk a begépelendő adatok színét. Az adatot a ***readln*** eljárással olvassuk be és a ***val*** eljárással az *sz* sztringet az *a* **real** típusú változóba alakítjuk át. Ha a *code* változó tartalma nem lesz zérus, akkor az adat hibás. Lekérdezzük kurzor *x* és *y* koordinátáit a ***WhereX*** és **WhereY** függvényekkel.

A hibajelzés a következőképpen néz ki:

Az adatbeolvasás sorában, de tőle négy karakterrel pirosan villogva kerül kiírása

```
Hibas adat!
```

Egy üres hellyel távolabb pedig sárga színnel jelenik meg a

```
Nyomj Space-t
```

szöveg és mindezt 200 ms ideig 100 Hz-es hangjelzés kísér.

Ha a *SPACE* billentyűt leütjük, a teljes sor törlődik és a `repeat until` ciklus addig ismétlődik, amíg a *code* nulla nem lesz, ami a hibátlan adat bevitelét jelenti.

Két sorral lejjebb zöld színnel kerül kiírásra

```
Akarja javitani (i/n):
```

A ***ReadKey*** a *ch* változóba olvassa be a leütött karaktert. Ezt a karaktert az ***UpCase*** függvény nagybetűvé alakítja, így elég az *I* és az *N* betűt vizsgálni. Az *i* válasz hatására az adatbeolvasás és az ellenőrzés elölről kezdődik, *n* válasz esetén a program befejezi a működését.

A CRT_PR.PAS program listája

```
program crt_pr;
uses crt;
var
     a  : real;
    sz  : string;
    code: integer;
    x,y : byte;
    ch  : char;
begin
  TextMode(C40);
  TextBackground(blue);
  clrscr;
  TextColor(Magenta);
  GotoXY(2,5);
  write('Adat beolvasasa billentyuzetrol');
  repeat
    repeat
      TextColor(White);
      GotoXY(2,10);
      write('Kerem az adatot: '); ClrEol;
      TextColor(Yellow);
      readln(sz);
```

```
            val(sz,a,code);
             x:=WhereX; y:=WhereY;
            if code <>0 then
                        begin
                           gotoXY(x+4,y);
                           TextColor(Red+Blink);
                           write('Hibas adat!');
                           TextColor(Yellow);
                           write(' Nyomj Space-t');
                           sound(100);Delay(200);
                           NoSound;
                           repeat until Keypressed;
                           gotoXY(2,10); DelLine;
                         end;
          until code=0;
          repeat
            gotoXy(2,12);
            TextColor(Green);
            write('Akarja javitani (i/n): '); ch:=ReadKey;
            ch:=UpCase(ch);
          until (ch='I') or (ch='N');
            gotoXY(2,12); delline;
        until ch='N';
end.
```

A CRT_PR.PAS program továbbfejlesztett változatát használjuk fel a 14. fejezetben bemutatott példában.

13.3.3. Menükezelés

A szöveges üzemmód használatára javasoljuk áttanulmányozni az alábbi egyszerű, de könnyen változtatható menükészítő programot.
A MENU.PAS programhoz tartozik a UTIL.PAS file is, amely unit formában tartalmazza a menükezelés eljárásait:

CursorOn	a kurzor megjelenítése,
CursorOff	a kurzor eltüntetése,
MenuWin	menüablak rajzoló eljárás.

A *MenuWin* eljárás hívása:

MenuWin(X,Y, Border, MenuCol, Item, ItemNum,Title, Choice)

A paraméterei:

X,Y	az ablak bal felső sarkának koordinátái,
Border	keret típusa, mely a következő értékeket veheti fel:

NoLine	nincs keret,
ThinLine	vékony keret,
FatLine	vastag keret,
DoubleLine	dupla vonalú keret,
DoubleTop	dupla a tetején, vékony az oldalán,
DoubleSide	dupla az oldalán, vékony a tetején.

MenuCol *MenuColors* típusú rekord, amely a keret és a menüpontok színeit tartalmazza:

FgBorder	menükeret színe,
BgBorder	menükeret háttere,
FgNormal	menüpont karakterének színe,
BgNormal	menüpont karakterének háttere,
FgSelect	kiválasztott menüpont karakterének színe,
BgSelect	kiválasztott menüpont karakterének háttérszíne

Items	*ItemType* típusú tömb, mely a menüpontok szövegét tartalmazza
ItemNum	a menüpontok száma
Title	Str80 típusú sztring, mely a menü felirata
Choice	a kiválasztott menüpont sorszámát tartalmazza, ha nulla a visszatérési érték, akkor a menüből ESC billentyűvel léphet ki.

A MENU.PAS program bemutatja a menükészítés módját, az alábbi öt menütételt jeleníti meg színesen:

```
Olvas
Szamol
Rajzol
Ment adatot
Vege
```

Ellenőrző kérdések:

1. Mire szolgál a *Crt* unit?
2. Mennyi az oszlopok és a sorok száma az alapértelmezés szerinti ablakban?
3. Melyik eljárással hozhatunk létre ablakot?
4. Hol van az aktív ablak (1,1) koordinátapontja?
5. Hangjelzés keltésére milyen eljárásokat kell aktiválni?
6. Melyik eljárással helyezzük át a kurzor pozícióját?
7. Melyik eljárás állítja a háttér színét?
8. Melyik eljárással változtathatjuk az írás színét?
9. Melyik eljárás törli a képernyőt?
10. Milyen eljárásokat kell aktíválnunk, hogy a
    ```
    Window(5,5,20,25);
    ```
    ablak háttere kék legyen?
11. Melyik eljárással lehet a kurzort tartalmazó sort törölni?
12. Melyik eljárással lehet a kurzor pozíciójától a sor végéig törölni?
13. Aktíváljuk a megfelelő eljárást, hogy az írás színe villogó piros legyen!
14. Milyen eljárások hívásával állítjuk vissza az alapállapotot, mielőtt visszatérünk DOS rendszerbe?

14. GRAPH UNIT HASZNÁLATA

14.1. Grafikus mód

Turbo Pascalban írt programok az IBM PC képernyőjét kétfajta módban használhatják

- szöveges módban, vagy
- grafikus módban,

azonban ezt a kétfajta módot csak felváltva lehet működtetni. A szöveges móddal már megismerkedtünk, most nézzük meg részletesebben a grafikus mód használatát is.

14.2. Graph unit

A *Graph* unit tartalmazza a grafikus eljárásokat és függvényeket, amely nagymértékben segítik a felhasználók grafikus programjainak elkészítését.

Az eljárások között van pont, vonal, kör, körív, ellipszis, ellipszisív, téglalap, poligon és téglatest rajzoló, de vannak olyan rutinok, melynek segítségével különböző színnel és mintával befesthetők az alakzatok. A rajzolásra felhasznált vonal vastagsága, színe és mintája is változtatható.

Többfajta karakterkészlet áll rendelkezésre: a kirajzolandó szöveg mérete változtatható és a helyzete is beállítható.

A grafikus képernyőnek is van kurzora (*current pointer* = CP, aktuális mutató vagy pointer, amely egy képpontra mutat) hasonló a szöveges mód kurzorához, amely villog a képernyőn, ez azonban nem látszik. A grafikus kurzort is tudjuk mozgatni a képernyőn, ívek rajzolásánál lekérdezhetjük a koordinátáit. Csak akkor tudunk igazán tervezni rajzot, ha ismerjük ennek a grafikus kurzornak a helyét a képernyőn.

Definiálhatunk ablakokat a képernyőn. Az ablak paramétereit beállíthatjuk úgy, hogy az ablakból kieső rajzok ne jelenjenek meg, ilyenkor vágás történik. A vágás nem vonatkozik az aktuális grafikus kurzorra.

Az IBM PC többfajta grafikus hardverrel rendelkezhet, a Turbo Pascal grafikus programcsomagja támogatja ezeket a változatokat.

14.3. Grafikus vezérlők típusa

Először tekintsük át a leggyakrabban használt grafikus vezérlő fajtákat, és csoportosítsuk ezeket a színkezelésük és felbontóképességük szerint.
Színkezelésük szerint megkülönböztetünk:
- monochrom (egyszínű, pl. papírfehér, zöld vagy narancssárga),
- színes monitorokat.

A felbontóképesség azt jelenti, hogy hány pontot tud kirajzolni vízszintes (oszlop) és függőleges (sor) irányban. Ez a két szám együtt adja a felbontást. A következő táblázat bemutatja a Turbo Pascal által támogatott vezérlők felbontását és a maximálisan használható színeinek számát.

	Vezérlő típusa	Felbontás	Színek száma
Hercules		720x348	2
CGA	(Color Graphics Adapter)	640x200	2
		320x200	4
EGA	(Enhanced Graphics Adapter)	640x350	16
VGA	(Video Graphics Adapter)	640x480	16
IBM8514		640x480	256
PC3270		720x350	2

A monochrom grafikus képernyő a legtöbb feladat igényeit kielégíti, mivel a szoftverek nagy része monochrom monitor esetén is működik. A felbontása elég jó, a szemet kevésbé veszi igénybe és az ára is kedvező. Ez a képernyő a színeket nem tudja kezelni, mivel összesen két színe van, ebből az egyik a háttér szín. Fekete/fehér (black&white) esetén az egyik szín a fekete a

háttér és a rajzolás színe a fehér vagy fordítva. A karakterek intenzitását tudjuk változtatni, amivel a megkülöböztetéseket programozhatjuk.

A CGA képernyőnek kicsi a felbontása, hosszabb ideig kellemetlen rajta dolgozni. Újabban már nem is kapható. A CGA 320x200 felbontásban a 16 szín közül egyszerre csak meghatározott 4 színt lehet grafikus üzemmódban használni. Az EGA monitor 640x350 felbontásban 64 színnel rendelkezik, ebből 16 színt tud egyidejűleg megjeleníteni. A VGA monitor 640x480-as felbontásban 256 színnel rendelkezik, azonban közvetlenül csak 16 színt tudunk programozni.

A Turbo Pascal grafikus rendszere két részből tevődik össze. A felhasználói felületet, amely hardver független, a GRAPH.TPU modul definíciói biztosítják. A grafikus vezérlő típusától függő részt a .BGI (*Borland Graphics Interface*) file-ok biztosítják.

A Pascal grafikus programokhoz különféle file-ok szükségesek. Egyik legfontosabb file, amely az adott grafikus kártyához tartozó meghajtó szoftver, kiterjesztése .BGI. A Pascal a következő grafikus hardvereket támogatja:

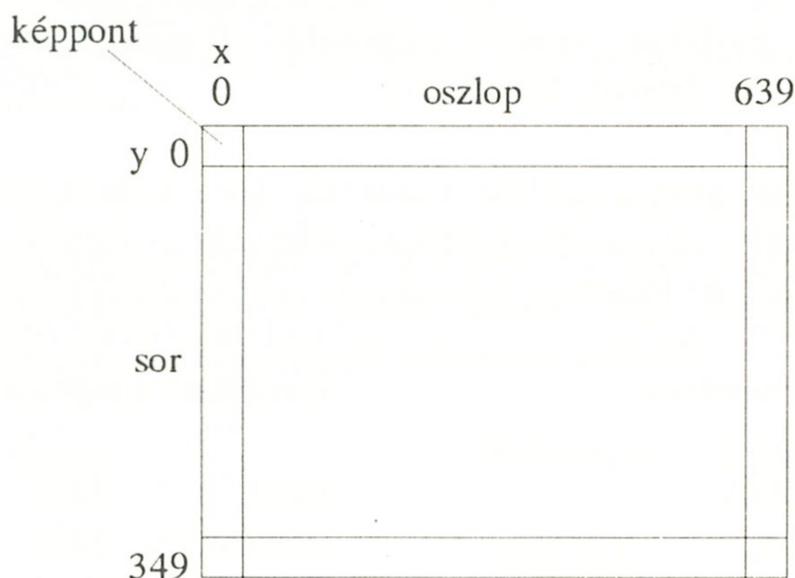
Grafikus hardver	Grafikus meghajtó program	
CGA, MCGA	CGA	.BGI
EGA, VGA	EGAVGA	.BGI
Hercules egyszínű	HERC	.BGI
IBM 3270	PC3270	.BGI
IBM 8514	IBM8514	.BGI
AT&T 6300	ATT	.BGI

A grafikus program fordításához a forrásprogram mellett szükséges, hogy a fordító program elérje a

- standard unit-okat, a TURBO.TPL file-t,
- a grafikus eljárásokat tartalmazó GRAPH.TPU file-t,
- a grafikus meghajtó szoftvert, a grafikus kártyának megfelelő .BGI file-t,
- karakterkészlet használata esetén .CHR kiterjesztésű file-kat.

A .BGI és a .CHR file-oknak a grafikus programot tartalmazó alkönyvtárban kell lenniük, különben a program a futás közben hibajelzést ad. A FERDEHAJ.PAS példaprogram bemutatja ezeknek a file-oknak a programba való beszerkesztését.

A rajzok készítéséhez meg kell ismerkednünk a képernyő koordinátarendszerével. Bármely grafikus módot választjuk ki, a (0,0) koordinátapont a képernyő bal felső sarkában van. Az x koordinátaértékek (oszlopok) jobbra, az y koordinátaértékek (sorok) lefelé növekednek. A sorokban és az oszlopokban a pontok száma függ a grafikus kártyától (felbontásától), ezek értéke a programból lekérdezhető a *GetMaxX* és a *GetMaxY* függvényekkel. Nézzük meg az EGA koordinátarendszerét 640x350 pontos felbontás esetén.



14.1. ábra Az EGA monitor EGAHi felbontása

Egy (x,y) képernyőpont (*pixel*) helye a képernyőn az x-edik oszlopot és az y-adik sort jelenti.
A grafikus eljárásokban a szögeket fokokban kell megadni. A szögek az óramutató járásával ellenkező irányban növekednek. A 0 fok 3 órának, 90 fok 12 órának és a 180 fok 9 órának felel meg.

14.4. A grafikus könyvtár eljárásainak és függvényeinek csoportosítása

A grafikus eljárásokat és függvényeket az alábbi hét csoportba sorolhatjuk:

- grafikus rendszert vezérlő,
- rajzolás és festés,
- képernyő és képernyő ablak kezelés,
- szöveg kiírása képernyőre,
- szín beállítása,
- hibakezelés,
- állapot lekérdezés.

GRAFIKUS RENDSZERT VEZÉRLŐ

CloseGraph	lezárja a grafikus rendszert,
DetectGraph	ellenőrzi a harvert, eldönti, hogy milyen grafikus rendszert használjunk, ajánl egy grafikus módot,
GraphDefaults	az alapértelmezés szerint beállítja a grafikus változókat,
GetGraphMode	visszatér az aktuális grafikus móddal,
GetModeRange	visszatér a specifikált meghajtónak a legalacsonyabb és legmagasabb módjával,
InitGraph	inicializálja a grafikus rendszert, a hardvert grafikus módba helyezi,
RestoreCrtMode	visszatölti az eredeti, a grafikus üzemmód előtti képernyőmódot,
SetGraphBufSize	a belső grafikus puffer méretét határozza meg poligon rajzoláshoz,
SetGraphMode	beállítja a grafikus módot, törli a képernyőt és újratölti az összes adatot az alapértelmezés szerint.

A grafikus programban az *InitGraph* eljárás tölti be a grafikus meghajtót és a rendszert grafikus módban helyezi. A grafikus módot *CloseGraph* eljárás hívásával kell lezárni, mielőtt a program befejezi a futását.

RAJZOLÁS ÉS FESTÉS

Rajzolás

Arc	körívet rajzol,
Circle	kört rajzol,
DrawPoly	poligon körvonalát rajzolja,
Ellipse	ellipszis ívet rajzol,
GetArcCoords	megadja a körív vagy ellipszis ív utolsó hívásának koordináta értékeivel,
GetAspectRatio	visszatér a grafikus képernyő maximális méretével, amelyből az oldalarány számítható,
GetLineSettings	visszatér az aktuális vonal stílusával, mintájával és vastagságával,
Line	egyenest rajzol (*x1,y1*) és (*x2,y2*) pontok között,
LineRel	egyenest rajzol az aktuális grafikus kurzor pozíciótól az adott relatív távolságban lévő pontba,
LineTo	egyenest rajzol az aktuális grafikus kurzor pozíciótól az adott (*x,y*) pontba,
MoveRel	mozgatja a grafikus kurzort egy relatív távolsággal,
MoveTo	mozgatja a grafikus kurzort az (*x,y*) pontba,
Rectangle	téglalapot rajzol,
SetAspectRatio	változtatja a beépített arány faktort,
SetLineStyle	beállítja az aktuális vonal vastagságát és stílusát.

Festés (kitöltés: Fill)

Bar	rajzol és befest egy téglalapot,
Bar3D	rajzol és befest egy téglatestet,
FillEllipse	rajzol és befest egy ellipszist,
FillPoly	rajzol és befest egy poligont,
GetFillPattern	visszatér a felhasználó által definiált mintával,
GetFillSettings	visszatér az aktuális festő mintával és színével,
PieSlice	rajzol és befest egy körcikket,
Sector	rajzol és befest egy ellipszis cikket,
SetFillPattern	kiválasztja a felhasználó által definiált festő mintát,
SetFillStyle	kiválasztja a festő mintát és a színt.

KÉPERNYŐ ÉS KÉPERNYPABLAK KEZELÉS

Képernyő kezelése

ClearDevice	törli az aktív képernyőt (az aktív lapot),
SetActivePage	kijelöli az aktív lapot a grafikus outputra,
SetVisualPage	láthatóvá teszi az adott grafikus lapot,

Képernyőablak kezelése

ClearViewPort	törli az aktuális képernyő ablakot,
GetViewSettings	visszaadja a képernyőablak infomrációit,
SetViewPort	kijelöli az aktuális képenyő ablakot grafikus outputra,

Kép kezelése

GetImage	a megadott képmező bittérképét elmenti egy pufferba,
ImageSize	megadja a tárolni kívánt téglalap alakú tartomány byte-jainak számával,
PutImage	a korábban tárolt képmező bittérképét a képernyőre helyezi.

Képpont kezelése

GetPixel	visszatér az (x,y) képpont színével,
PutPixel	egy pontot rajzol az (x,y) pontba.

SZÖVEGKIÍRÁS KÉPERNYŐRE GRAFIKUS MÓDBAN

GetTextSettings	visszatér az aktuális karakterkészlet típusával, irányával, méretével és helyzetével,
OutText	az aktuális pozíciónál szöveget ír,
OutTextXY	a megadott pozíciónál szöveget ír,
SetTextJustify	szöveg helyzetének beállítása az *OutText* és *OutTextXY* eljárások számára,

SetTextStyle	beállítja az aktuális karakterkészlet, a kiírásának irányát és a karakterek méretét,
SetUserCharSize	beállítja a karakter szélesség és magasság faktorát,
TextHeight	megadja a szöveg képpontokban mért magasságát,
TextWidth	megadja a szöveg képpontokban mért szélességét.

SZÍN BEÁLLÍTÁS

Szín információ vétele

GetBkColor	visszatér az aktuális háttér színével,
GetColor	visszatér az aktuális rajzolási színnel,
GetDefaultPalette	visszatér a paletta struktúrával,
GetMaxColor	visszatér a maximálisan használható szín értékével az aktuális grafikus módban,
GetPaletteSize	visszatér a paletta színeinek számával.

Beállít egy vagy több színt

SetAllPalette	változtatja a paletta színeit a megadott színekkel,
SetBkColor	beállítja az akutális háttérszínt,
SetColor	beállítja az aktuális rajzolási színt,
SetPalette	egy paletta színt változtat.

HIBAKEZELÉS

GrapghErrorMsg	visszatér a hibakónak megfelelő üzenettel,
GraphResult	az utoljára végrehajtott grafikus művelet hibakódját adja vissza.

ÁLLAPOT LEKÉRDEZÉS

GetArcoords	visszatér az utoljára rajzolt ív vagy ellipszis ív koordinátáinak értékével,
GetAspectRatio	visszatér a grafikus képernyő oldalarány értékeivel,
GetBkColor	visszatér az aktuális háttérszínnel,

GetColor	visszatér az akutális rajz színével,
GetDriverName	visszatér az aktuális grafikus meghajtó nevével,
GettFillPattern	visszatér a felhasználó által definiált festő mintával,
GetFillSettings	visszatér az aktuális festő mintával és színével,
GetGraphMode	visszatér az aktuális grafikus móddal,
GetLineSettings	visszatér az aktuális vonal típusával, mintájával és vastagságával,
GetMaxColor	visszatér a maximálisan használható színek számával,
GetMaxMode	visszatér az aktuális megható maximális grafikus módjainak számával,
GetMaxX	visszatér a maximális pontok számával x irányban,
GetMaxY	visszatér a maximális pontok számával y irányban,
GetModeName	visszatér az aktuális mód méretével,
GetModeRange	visszatér az adott meghajtó módhatáraival,
GetPalette	visszatér a palettával és méretével,
GetPixel	visszatér az (x,y) képpont színével,
GetTextSettings	visszatér az aktuális karakterkészlettel, irányával, méretével és helyzetével,
SetViewSettings	információt ad az aktuális képernyő ablakról,
GetX	visszatér az aktuális pozíció x koordináta értékével,
GetY	visszatér az aktuális pozíció y koordinátaértékével.

Grafikus mód konstansai, változói és rekordjai

Grafikus vezérlők

```
Detect   = 0;
CGA      = 1;
MCGA     = 2;
EGA      = 3;
EGA64    = 4;
EGAMono  = 5;
IBM8514  = 6;
HercMono = 7;
ATT400   = 8;
VGA      = 9;
PC3270   =10;
```

Grafikus módok

```
CGA meghajtó grafikus módjai:
     CGAC0       = 0;      { 320x200 }
     CGAC1       = 1;      { 320x200 }
     CGAC2       = 2;      { 320x200 }
     CGAC3       = 3;      { 320x200 }
     CGAHi       = 4;      { 640x200 }

MCGA meghajtó grafikus módjai:
     MCGAC0        = 0; { 320x200 }
     MCGAC1        = 1; { 320x200 }
     MCGAC2        = 2; { 320x200 }
     MCGAC3        = 3; { 320x200 }
     MCGAMed       = 4; { 640x200 }
     MCGAHi        = 5; { 640x480 }

EGA meghajtó grafikus módjai:
     EGALo         = 0; { 640x200 }
     EGAHi         = 1; { 640x350 }

EGA64 meghajtó grafikus módjai:
     EGA64Lo       = 0; { 640x200 }
     EGA64Hi       = 1; { 640x350 }
     EGAMonoHi     = 3; { 640x350 }

HercMono meghajtó grafikus módja:
     HercMonoHi    = 0; { 720x348 }

ATT400 meghajtó grafikus módjai:
     ATT400C0      = 0; { 320x200 }
     ATT400C1      = 1; { 320x200 }
     ATT400C2      = 2; { 320x200 }
     ATT400C3      = 3; { 320x200 ]
     ATT400Med     = 4; { 640x200 }
     ATT400Hi      = 5; { 640x400 }
```

VGA meghajtó grafikus módjai:

```
VGALo        = 0; { 640x200 }
VGAMed       = 1; { 640x350 }
VGAHi        = 2; { 640x480 }
```

PC3270 meghajtó grafikus módjai:

```
PC3270Hi     = 0; { 720x350 }
```

IBM8514 meghajtó grafikus módjai:

```
IBM8514Lo    = 0; { 640x480 }
IBM8514Hi    = 1; { 1024x768 }
```

Színkonstansok

```
Black        = 0;
Blue         = 1;
Green        = 2;
Cyan         = 3;
Red          = 4;
Magenta      = 5;
Brown        = 6;
LightGray    = 7;
DarkGray     = 8;
LightBlue    = 9;
LightGreen   = 10;
LightCyan    = 11;
LightRed     = 12;
LightMagenta = 13;
Yellow       = 14;
White        = 15;
```

Színek a 8514

```
EGABlack     = 0;
EGABluse     = 1;
EGAGreen     = 2;
EGACyan      = 3;
EGARed       = 4;
EGAMagenta   = 5;
```

```
EGABrown        = 20;
EGALightgray    = 7;
EGADarkgray     = 56;
EGALightblue    = 57;
EGALightgreen   = 58;
EGALightcyan    = 69;
EGALightred     = 60;
EGALightmagenta= 61;
EGAYellow       = 62;
EGAWhite        = 63;
```

Vonal stílusok és szélességek:

```
SolidLn         = 0;
DottedLn        = 1;
CenterLn        = 2;
DashedLn        = 3;
UserBitLn       = 4;
NormWidth       = 1;
ThickWidth      = 3;
```

Karakterkészletek és irány:

```
DeafultFont     = 0;
TriplexFont     = 1;
SmallFont       = 2;
SanSerifFont    = 3;
GothicFont      = 4;
HorizDir        = 0; { balról jobbra }
VertDir         = 1; { alulról felfelé ]
UserCharSize    = 0;
```

Vágó konstansok:

```
ClipOn      = true;
ClipOff     = false;
```

Bar3D konstansok:

```
TopOn       = true;
TopOff      = false;
```

Töltő minták:

```
EmptyFill      = 0;   { nincs minta, kitöltés háttér színű }
SolidFill      = 1;   { egyenletes, halvány tónus }
LineFill       = 2;   { vízszintes vonalas minta }
LtSlashFill    = 3;   { dőlt '/' vonalas minta }
SlashFill      = 4;   { dőlt '/' vastag vonalas minta }
BkSlashFill    = 5;   { dőlt '\' vastag vonalas minta }
LtBkSlashFill  = 6;   { dőlt '\' vonalas minta }
HatchFill      = 7;   { kockás minta }
XHatchFill     = 8;   { dőlt kockás minta }
InterLeaveFill = 9;   { súrún pontozott minta }
WideDotFill    = 10;  { ritkán pontozott minta }
CloseDotFill   = 11;  { közepesen pontozott minta }
UserFill       = 12;  { a felhasználó által definiált minta, amit
                        a SetFillPattern eljárásnál adott meg }
```

PutImage bit operátorai:

```
NormalPut  = 0; { MOV }
CopyPut    = 0; { MOV }
XORPut     = 1; { XOR }
OrPut      = 2; { OR  }
AndPut     = 3; { AND }
NotPut     = 4; { NOT }
```

Vízszintes és függőleges elrendezés a SetTestJustification eljárás számára:

```
CenterText  = 1;
LeftText    = 0;
RightText   = 2;
BottomText  = 0;
TopText     = 2;
```

Konstansok és rekordok:

```
cont
    MaxColor = 15;
type
    PaletteType = record
        Size : Byte;
        Colors: array[0..MaxColors] of Shortint;
    end;
```

```
    LineSettingsType = record
         LineStyle     : Word;
         Pattern   : Word;
         Thickness : Word;
    end;

    TextSettingsType = record
         Font      : Word;
         Direction : Word;
         CharSize  : Word;
         Horiz          : Word;
         Vert      : Word;
    end;

    FillSettingsType = record
         Pattern   : Word;
         Color          : Word;
    end;

    FillPatternType = array[ 1..8 ] of Byte;

    PointType = record
         X,Y : integer;
    end;

    ViewPortType = record
         x1,y1,x2,y2   : integer;
         Clip      : Boolean;
    end;

    ArcCoordType = record
         X,Y,
         Xstart,Ystart,
         Xend,Yend : integer;
    end;
```

Változók:

```
var
    GraphGetMemPtr  : pointer;
    GraphFreeMemPtr : integer;
```

14.5. Grafikus programok készítése

14.5.1. Színkezelés különböző vezérlők esetén

Pascalban a színek kiválasztásánál névvel vagy sorszámmal hivatkozhatunk. Közvetlenül 16 szín programozható (0-15) vezérlőtöl és a kiválasztott módtól függően.
Példaként bemutatunk néhány színt. A szinekre névvel, vagy számmal is hivartkozhatunk:

```
SetTextColor(blue); vagy SetTextColor(2);
```

14.5.2. Jelentősebb mód konstansok

A grafikus programok készítéséhez a grafikus meghajtókhoz tartozó mód típusokra és a hozzájuk tartozó adatokra.
A jelentősebb grafikus mód konstansokat az alábbi táblázat fogalja össze:

Grafikus meghajtó	Érték	Konstans neve	Érték	Oszlopxsor	Paletta	Lap
HercMono	7	HercmonoHi	0	720x348	2 szín	2
CGA	1	CGAC0	0	320x200	C0	1
		CGAC1	1	320x200	C1	1
		CGAC2	2	320x200	C2	1
		CGAC3	3	320x200	C3	1
		CGAHi	4	640x200	2 szín	1
EGA	3	EGALo	0	640x200	4 szín	1
		EGAHi	1	640x350	16 szín	1
VGA	9	VGALo	0	640x200	16 szín	2
		VGAMed	1	640x350	16 szín	2
		VGAHi	2	640x480	16 szín	1

14.5.3. Grafikus program felépítésének vázlata

Ahhoz, hogy rajzolni tudjunk az **uses** mellett definiálni kell a *Graph* **unit**-ot. Deklarálni kell három egész típusú változót

Gdriver	grafikus vezérlő,
Gmode	grafikus mód,
Hibakód	a grafikus hiba kódját tartalmazza.

Ha a `Gdriver:=Detect;` ez azt jelenti, hogy a ***GraphInit*** eljárásra bízzuk, hogy állapítsa meg a meghajtó típusát és a *Gmode* változóba a típusnak megfelelő legnagyobb felbontásnak megfelelő módot ajánlja. Az ***InitGraph*** harmadik paramétere, hogy a grafikus vezérlő program (.BGI) hol található.

Három eset lehetséges:

1. '' üres sztring megadása esetén abban az alkönyvtárban van a megfelelő .BGI, ahol a program van.

2. 'C:\tp\bgi\' az úvonal pontos nevét adjuk meg,

Két megoldásra mutatunk példákat a GRAFIKA1 ill. a GRAFIKA2 programokkal.
Ha a grafikus megnyitás sikeres volt, megkezdődhet a rajzolás. Ahhoz, hogy a rajz a képernyőn maradjon javasolt a ***readln***; aktiválása, amely az ENTER leütéséig kimerevíti a képernyőt. A grafikát a ***CloseGraph*** eljárással zárjuk le, mielőtt a program a futását befejezné.

```
program grafika1;
uses Graph;
var
    Gdriver,Gmode,Hibakod: integer;
begin
    Gdriver:=Detect;
    GraphInit(Gdriver,Gmode,'');
    Hibakod:=GraphResult;
    if Hibakod<>GrOk then
```

```
                           begin
                             writeln('Grafikus hiba: ',
                                    GraphErrorMsg(Hibakod);
                             Writeln('Program exit...');
                             Halt;
                            end

                          else
                            begin
                             {  rajzolás  }
                            end;
          readln;
          CloseGraph;
     end.
```

A GRAFIKA2.PAS program sztringből veszi a .BGI file elérési útvonalát, ha ez nem megfelelő a program rákérdez:

```
     program grafika2;
     uses Graph;
     var
          Gdriver,Gmode,Hibakod: integer;
          BGIUtvonal: string;
     begin
          BGIUtvonal:='C:\BGI\';
          Gdriver:=Detect;
          GraphInit(Gdriver,Gmode,BGIUtvonal);
          Hibakod:=GraphResult;
          if Hibakod<>GrOk then
                begin
                  writeln('Grafikus hiba: ',
                          GraphErrorMsg(Hibakod);

                  if(Hibakod = grFileNotFound then
                    begin
                     write('Adja meg a BGI teljes utvonalat',
                      ' vagy Ctrl-Break program exit');
                     readln(BGIUtvonal);
                     writeln;
                    end
                  else Halt(1);
                end;
               {  rajzolás  }
               readln;
               CloseGraph;
     end.
```

Nézzünk néhány példát különböző típusú monitorok valamely grafikus módjának a megnyitására:

a. Hercules monitor esetén

A grafikus üzemmód megnyitása:

```
GraphDriver:=HercMono; GraphMode:=HercMonoHi;
InitGraph(GraphDriver, GraphMode,'');
```

A Hercules monitornak nincs más üzemmódja. A 720x348 felbontás azt jelenti, hogy 0-719 sorszámúak az oszlopok száma és 0-347 sorszámúak a sorok száma. Két szín használható, amelyből egy a háttérszín. A lapok száma kettő, lehetőség van megjelenített lap mellett egy másik lapra is rajzolni és a lapokat váltani lehet, amely animációs rajzprogramnál meggyorsítja a képek mozgatását a képernyőn.

b. CGA monitor esetén

CGA monitor esetén 5 fajta üzemmód van. Négy üzemmódban a 320x200 durva grafikában négy színt használhatunk, a 640x200 finom grafikában pedig csak két színt. Csak egy grafikus lapja van.

Nézzük meg részletesebben a CGA durva grafika színkiválasztását. A négy szín közül egy a háttérszín, amely a 16 szín közül bármely lehet. Négy paletta közül választhatjuk a további 3 színt a rajzoláshoz. A mód kiválasztásával aktiváljuk a palettát.

Grafikus mód		**színek sorszáma**		
neve	**száma**			
		1	2	3
CGAC0	0	világoszöld	világospiros	sárga
CGAC1	1	világos türkiz	világoslila	fehér
CGAC2	2	zöld	piros	barna
CGAC3	3	türkiz	lila	világosszürke
CGAHi	4			

Például válasszuk ki a CGA monitor 0-ás palettáját:

```
GraphDriver:=CGA; GraphMode:=CGAC0;
InitGraph(GraphDriver, GraphMode,'');
```

Ennél a palettánál a világoszöld, világospiros és a sárga használható.

Válasszuk háttérszínnek a kéket és váltogassuk a rajzolás színét:

```
GraphDriver:=CGA; GraphMode:=CGAC1;
InitGraph(GraphDriver, GraphMode,'');

SetBkColor(Blue);   háttérszín kék
SetColor(1);        rajzolás színe :  világos türkíz,
SetColor(2);                          világoslila,
SetColor(3);                          fehér.
```

Válasszuk ki a CGA monitor nagyfelbontású módját:

```
GraphDriver:=CGA; GraphMode:=CGAHi;
InitGraph(GraphDriver, GraphMode,'');
```

c. EGA monitor esetén

EGA monitor esetén a hardver 64 színt tud kezelni, azonban közvetlenül 16 színt használhatunk egyidejűleg.
Válasszuk ki az EGA monitor nagyfelbontású módját:

```
GraphDriver:=EGA; GraphMode:=EGAHi;
InitGraph(GraphDriver, GraphMode,'');
```

Rajzoljuk kék színnel:

```
SetColor(Blue);   vagy
SetColor(1);
```

d. VGA monitor esetén

VGA monitor 256 színből közvetlenül csak 16 szín. A VGALo és a VGAMed mód kiválasztása esetén 2 grafikus lapot programozhatunk.

14.5.4. Grafikus üzemmód hibajelzései

A ***GraphResult*** függvény adja vissza a GRAPH **unit** belső hibajelzéseit, mely mindig az utoljára végrehajtott grafikus művelet állapotáról ad információt.

A *GraphResult* eljárásnak az alábbi hiba kód konstansai léteznek:

```
const
     grOk                = 0;    { Nincs hiba. }
     grNoInitGraph       = -1;   { BGI nincs installálva
                                   InitGraph hívásával }
     grNotDetected       = -2;   { Nem érzékelt grafikus
                                   hardvert. }
     grFileNotFound      = -3;   { Az egység meghajtó .BGI file
                                   hiányzik. }
     grInvalidDriver     = -4;   { Érvénytelen a meghajtó
                                   .BGI file }
     grNoLoadMem         = -5;   { Kevés a memória a meghajtó
                                   betöltéséhez. }
     grNoScanMem         = -6;   { Kevés a memória. }
     grNoFloodMem        = -7;   { Kevés a memória. }
     grFontNotFound      = -8;   { Karakterkészlet file-t nem
                                   találja. }
     grNoFontMem         = -9;   { Kevés a memória a karakter-
                                   készlet betöltéséhez. }
     grInvalidMode       = -10;  { Érvénytelen a grafikus mód
                                   a kiválasztott meghajtó szá-
                                   ra. }
     grError             = -11;  { Grafikus hiba. }
     grIOError           = -12;  { Grafikus I/O hiba. }
     grInvalidFont       = -13;  { Érvénytelen karakterkészlet
                                   file. }
     grInvalidFontNum    = -14;  { Érvénytelen karakterkészlet
                                   szám. }
     grInvalidDeviceNum  = -15;  { Érvénytelen a grafikus eszköz
                                   száma }
```

14.6. Grafikus mintafeladatok

14.6.1. Szöveg kiírása grafikus módban

MINTA1.PAS program minden monitoron fut, fekete/fehér rajzot készít. Ez az egyszerű rajzprogram egy maximális méretű téglalapot rajzol és a téglalap közepére 'Grafikus feladat' szöveget írja ki. Tervezzük meg a programot:

A program neve legyen rajz_minta. A **uses** kulcsszó mellett meg kell adni a *Graph* grafikus könyvtár nevét, így tudunk hozzáférni a grafikus eljárásokhoz és függvényekhez.

Három egész típusú változóra lesz szükségünk:

GraphDriver	a grafikus meghajtó
GraphMode	a grafikus mód és a
ErrorCode	a hibaüzenet számára.

Ha olyan grafikus programot tervezünk, amely automatikusan meghatározza a vezérlő típusát és a legnagyobb felbontásnak megfelelő módot ajánlja fel, akkor az grafikus üzemmód megnyitásánál az ***InitGraph*** eljárást a ***GraphDriver*** paraméterben megadott ***Detect*** konstanssal kell hívni. Ennek hatására a *GraphDriver* a megfelelő meghajtót és a *GraphMode* pedig az aktuális meghajtónak megfelelő legnagyobb felbontást jelentő módot fogja tartalmazni.
A ***GraphResult*** függvénnyel ellenőrizhetjük, hogy az ***InitGraph*** eljárás aktiválásával sikerült-e áttérni a grafikus üzemmódra. Ha az *ErrorCode* változóban visszaadott érték megegyezik a *grOk* konstanssal, akkor sikeres volt a grafikus üzemmód inicializálása, ellenkező esetben a ***GraphErrorMsg*** eljárást az *ErrorCode* változóval aktiválva szöveges formában is kiírathatjuk a hiba okát és a program futása megszakad.

Hibátlan inicializálás esetén rajzoljunk egy maximális méretű téglalapot a ***Rectangle*** eljárással. A téglalap bal felső sarka a (0,0) pont és a jobb alsó sarka pedig a grafikus kártyától függően a maximális oszlop, ill. sorok száma legyen, ezeket a ***GetMaxX*** és a ***GetMaxY*** függvényekkel kérdezhetjük le.

A képernyőn a szöveg helyzetét a *SetTextJustify* eljárással állíthatjuk be. Ha a *SetTestJustify* mindkét paraméterénél a *CenterText* konstanst adjuk meg, akkor a szöveg középre kerül. Ezután definiálnunk kell a karakterkészlet típusát, a kiirandó szöveg irányát, valamint a betűk méretét. Jelen példánkban használjuk a normál karakterkészletet, a kiírás iránya legyen vízszintes és a betűk nagysága 3-szoros. A *SetTextStyle* eljárást a *DefaultFont*, a *Horizdir* és 3 paraméterekkel kell aktíválnunk.

Ha a szöveget adott koordinátapontnál akarjuk megjeleníteni, akkor az *OutTextXY* eljárást használjuk. Az *OutTextXY* eljárás első két paramétere az *x,y* koordinátapont, ahol a szöveg megjelenik, mivel a *SetTextJustify* eljárással a szöveget középre centírozzuk, így itt a képernyő közepét kell megadni. Képernyő közepét különböző típusú monitorok esetén dinamikusan a *GetMaxX* és a *GetMaxY* függvények hívásával tudjuk kiszámítani. Az így megkapott maximális x, ill y irányú méretet egészosztással (**div**) a képernyő közepére tesszük. Az *OutTextXY* eljárás harmadik paramétere a kiirandó szöveg, jelenleg a programban aposztrofok között sztring konstansként adjuk meg a kiirandó szöveget.

Gondoskodnunk kell arról, hogy a felrajzolt kép ne tünjön el azonnal, hanem addig lássuk, amíg akarjuk. Ezt elérhetjük egy ***Readln*** paraméternélküli hívásával, ennek hatására a kép kimerevedik a képernyőn addig, míg *Enter* billentyűt nem nyomtunk meg.
A grafikus programot a *CloseGraph* eljárás hívásával kell lezárni, amely visszaállítja azt a módot, amely a grafikus üzemmód előtt volt, jelen esetben az eredeti normál video módot.

A *rajz_minta1* rajzprogram listája a következő:

```
(* rajz1.pas *)
program rajz_minta1;
uses Graph;
var
     GraphDriver, GraphMode, ErrorCode : integer;
begin
     GraphDriver := Detect;
     InitGraph(GraphDriver, GraphMode,'');
     ErrorCode := GraphResult;
     if ErrorCode <> grOk then
```

```
         begin
            writeln('Grafikus hiba: ',
                     GraphErrorMsg(ErrorCode));
              writeln('Program exit! ');
              Halt(1);
         end;
       Rectangle(0, 0, GetMaxX, GetMaxY);
       SetTextJustify(CenterText, CenterText);
       SetTextStyle(DefaultFont, HorizDir, 3);
       OutTextXY(getMaxX div 2, GetMaxY div 2,
                'Grafikus feladat ');
       Readln;
       CloseGraph;
end.
```

Ha a programot lefuttatjuk olyan PC-ken, amelyeknek Hercules, CGA, EGA vagy VGA monitort csatlakozik, akkor a program mindegyik gépen a legnagyobb felbontást választva két színt használva rajzolja a keretet és írja ki a szöveget a képernyő közepére.

RAJZ2.PAS program

Módosítsuk a programunkat úgy, hogy a különböző típusú monitorokat kiválasztva színes jelenjen meg a háttér, a téglalap és a szöveg, kivéve Hercules monitor esetében.

```
(* rajz2.pas *)
program rajz_minta2;
uses Graph;
var
     GraphDriver, GraphMode, ErrorCode: integer;
begin
     GraphDriver := Detect;
     case GraphDriver of
     1: { CGA driver }
        GraphMode := CGAC1;

     3: { EGA driver }
        GraphMode:= EGALo;

     7: { Hercules driver }
        GraphMode := HercMonoHi;
     8: { VGA driver }
        GraphMode := VGALo;
     end;
```

```
          InitGraph(GraphDriver, GraphMode,'');
          ErrorCode := GraphResult;
          if ErrorCode <> grOk then
             begin
                  writeln('Grafikus hiba: ',
                           GraphErrorMsg(ErrorCode));
                  writeln('Program exit! ');
                  Halt(1);
             end;
          SetBkColor(White);
          SetColor(2);
          Rectangle(0, 0, GetMaxX, GetMaxY);
               SetTextJustify(CenterText, CenterText);
               SetTextStyle(DefaultFont, HorizDir,3);
               SetColor(1);
               OutTextXY(GetMaxX div 2, GetMaxY div 2,
                  'Grafika');
               Readln;
               CloseGraph;
     end.
```

A G_SZOVEG.PAS program a képernyő közepén jeleníti meg a szöveget. Az összes karakterkészlet be van építve, csak egy érvényes belőle. A futtatáskor változtathatjuk a szöveg nagyságát és a karakterkészletet is.

A meghajtó nevének kiiratása

A MEGHAJTO.PAS program kiírja a meghajtó nevét a képernyőre. A .BGI file-oknak az aktuális könyvtárban kell lenniük,

```
(*  meghajto.pas *)
program meghajto;
  uses Graph;
  { GetDriverName procedure }

var Gd,Gm,ErrorCode: integer;
  procedure grstart;
  begin
         Gd:=Detect;
         Initgraph(Gd,Gm,'');
         ErrorCode := GraphResult;
         if ErrorCode<>GrOk then
                begin
                writeln('Grafikus hiba: '
```

```
                    ,GraphErrorMsg(ErrorCode));
                  writeln('A program befejezi a futasat. ');
                  Halt(1);
                  end;
      end;
  begin
  grstart;
    Outtext('Meghajto neve: '+ GetDriverName);
    Readln;
  CloseGraph;
  end.
```

14.6.2. Szöveges és grafikus mód váltása

GRVALTAS.PAS program mutatja be a grafikából szöveges módba, majd újra grafikába való váltást.

```
(* GRValtas.pas *)
(* Grafika es szoveges mod valtasa *)
program GValtas;
  uses Graph,Crt;
  { GetGraphMode procedure }
  var Gd,Gm:integer;
      Mode: Integer;
      Procedure grstart;
      begin
        Gd:=Detect;
         Initgraph(Gd,Gm,'');
         if GraphResult<>GrOk then Halt(1);
      end;
  begin
  grstart;
  Rectangle(20,20,290,100);
  OutTextxy(25,30,'<RETURN> kilepunk a grafikabol.');
  Readln;

  RestoreCRTMode;
  GotoXY(10,10);
  Write(' Szoveges modban vagyunk!');
  GotoXY(10,12);
  write('<RETURN> hatasara visszaterunk grafikus modba.');
  Readln;
```

```
SetGraphMode(GetGraphMode);
Rectangle(20,20,290,100);
OutTextXY(25,30,'Grafikus modban vagyunk');
OutTextXY(25,30+TextHeight('H'),
          '<RETURN> lezarjuk a grafikat');
Readln;
CloseGraph;
end.
```

A RAJZ_T.PAS program a CRT_PRG.PAS program továbbfejlesztett változata. A 40 oszlopos szöveges módban az olvas eljárás felhasználásával olvassuk be a téglalap két oldalának adatait és a bal felső koordinátáinak értékét, mindegyik adat ellenőrízve van, és javításra is van lehetőség.

A *DetectGraph* eljárás automatikusan megvizsgálja a képernyő típusát és annak megfelelően állítjuk be a grafikus módot és a rajzolás két színét.

14.6.3. CGA.BGI és a LITT.CHR programba fordítása

A FERDEHAJ.PAS program a ferdehajítást szimulálja. A szöveges módban 40 karakterre állítja be a képernyőt. Párbeszédes üzemmódban kérdezi meg a dobás távolságát, melyet el kell találni, majd a dobás szögét és sebességét. Az adatokat ellenőrzi.

A grafikához a CGA meghajtót és a CGAC1 módot választja ki, ezáltal a zöld, piros és a sárga palettát használja.

A grafikus mód nyitásánál meg kell adni azt annak az alkönyvtárnak nevét, ahol program megtalálja a grafikát meghajtó .BGI file-okat, vagy a programunkkal együtt mindig be kell másolni az adott alkönyvtárba és mindig a programmal együtt kell léteznie.

Lehetőség van a .BGI file-oknak a programba való befordítására. A jelen program a grafikus mód váltásánál nem keresi a CGA.BGI file-t, mert a meghajtót a programba fordítottuk az alábbi módon:

1. A BINOBJ segédprogram segítségével object formátumra alakítjuk a megfelelő .BGI file-t, pl. a CGA.BGI, akkor a következő parancsot kell kiadni:

```
binobj cga.bgi cga.obj cga
```

Hatására a CGA.OBJ nevű file-ban előáll az object formátumú grafikus meghajtó, melyet programunkban CGA eljárásnéven deklaráltunk.

2. A program elején {$L} fordítási direktívával beszerkesztjük a CGA.OBJ nevű file-t, majd az **external** kulcsszóval külső eljárásnak deklaráljuk CGA néven.

```
{$L CGA.OBJ}             { az object formatumu CGA }
procedure CGA; EXTERNAL;  { meghajto beszerkesztese }
```

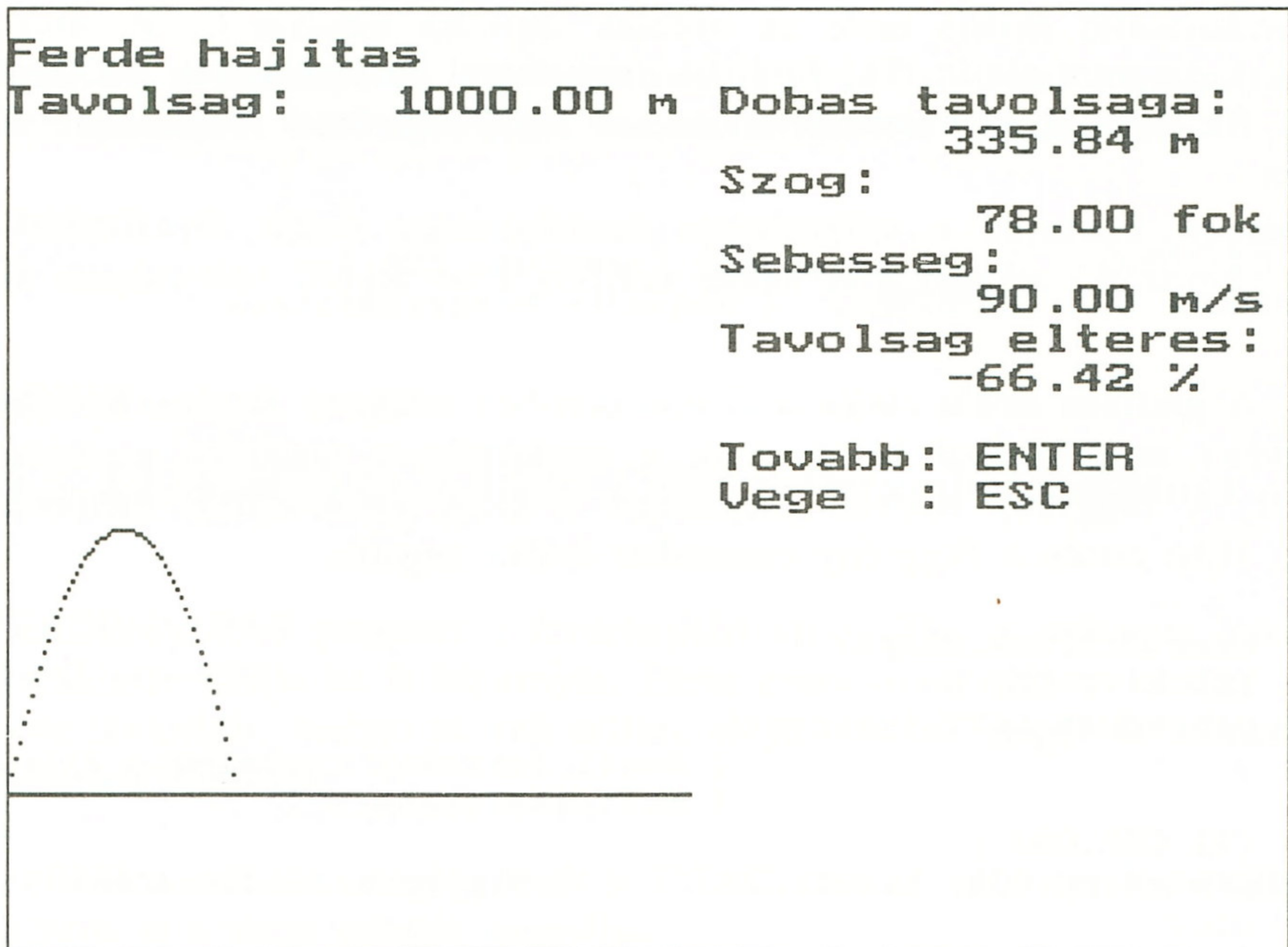
3. A grafikus nyitás elején a *Graph* unit-ban található ***RegisterBGIDriver*** függvény meghívásával bejegyezzük a rendszerbe a meghajtó memóriabeli helyét. A függvény bemenő paramétere a CGA nevű eljárás memóriabeli címe. Hiba esetén a függvény visszaadott értéke negatív.

```
uses Crt,Dos,Graph;
{$L LITT.OBJ }
procedure LITT; External;
                        { Small fontokat tartalmazo file }
                        { beszerkesztesere }
{$L CGA.OBJ }
procedure CGA; External;     { meghajto beszerkesztese }
....
{ grafika megnyitasa elotti ellenorzes }
gd:= CGA; gm:=CGAC0;

if RegisterBGIDriver(@CGA) < 0 then      { CGA meghajto }
 begin
    writeln('Nem sikerult beepiteni a CGA file-t ');
    halt(1);
  end;
if RegisterBGIFont(@LITT) < 0 then
   { Small font }
 begin
    writeln('Nem sikerult beepiteni a LITT file-t ');
```

```
        halt(2);
    end;
        initgraph(gd,gm,'');
```

A program a SMALL fontkészletet használja a grafikában. A LITT.CHR file-t az CGA.BGI-hez hasonló módon kell beszerkeszteni.

A program grafikus képernyője



14.6.4. Grafikus kurzor mozgatása

A CUR_PROG.PAS program bemutatja, hogyan lehet grafikus kurzort mozgatni a képernyőn.

A program CGA, EGA és VGA képernyőt tud kezelni. Kirajzolja a képernyőre az aktuális dátumot, a napot valamint az időt is kijelzi.

A program eljárásai:

EGA_color	EGA és VGA képernyőre a rajzolás színeit állítja be,
CGA_color	CGA képernyőre a rajzolás szineit állítja be,
graph_ini	automatikusan kiválasztja a grafikus meghajtót,
frame_ini	a rajzkeret adatait és a kurzor x irányú mozgatásának nagyságát állítja,
frame	megrajzolja a keretet a fejlécekkel, aktiválja a rajzol eljárást,
adat_ki	a keret jobb szélén függőlegesen írja vissza a kurzor alatti adat értékét,
rajzol	adatokat generál és rajzol,
draw_cursor	mozgatja a kurzort és leolvassa az ábrázolt adatokat a kurzor jobbra mozgatásánál szaggatott vonallal, balra mozgatásánál telt vonallal jelenik meg és az adat kiíratási színe is változik.

A CURSOR.PAS unit grafikusan mozgatható kurzor rutinokat tartalmaz. A kurzor hossza, színe és stílusa (teljes vagy szaggatott vonalú) is beállítható. A unit két eljárást tartalmaz:

putcursor
delcursor

A *putcursor* eljárás kihelyezi a kurzort a képernyőre, a paraméterei:

p	pointer, kimenő paraméter, amely rámutat arra a memória címre, ahol a kurzor képe van tárolva,	
h	a kurzor x koordinátája,	
v	a kurzor y koordinátája,	
hl	a kurzor magasságának mérete, szélessége 5-5 raszterre van állítva,	
vonal	1	a kurzor teljes vonal,
	0	a kurzor szaggatott vonal,
color	a kurzor színe.	

A *getcursor* eljárás törli a kurzort a képernyőről, a paraméterei:

- *p* pointer, a kurzor memóriabeli címére mutat,
- *h* a kurzor x koordinátája,
- *v* a kurzor y koordinátája,
- *hl* a kurzor magassága.

14.6.5. Alakzat mozgatása

Az URHAJO.PAS program inkább egy repülő csészealjhoz hasonló tárgyat mozgat. A EGA vagy VGA monitorral rendelkező gépen fut. Az animációhoz szükséges lapozási technikát mutatja be. A *Page* eljárása váltogatja a lapokat.

```
procedure Page;
begin
     setvisualpage(Active);
     Active := 1 - Active;
     setactivepage(Active);
     cleardevice;
end;
```

PALETTA.PAS program a színskálát körcikkein mutatja be. ENTER leütésére a kör forogni kezd, ha megáll újra ENTER leütésére fejezi be a program futását.

14.6.6. Képernyő torzításának kiküszöbölése

ASPECT_P.PAS program EGA és VGA monitoron fut, bemutatja a négyzet és a beleírható kör torzításának számítását.

A TORZIT.PAS program ellipszisből kiindulva addig növeli a torzítási arányt, míg kör nem lesz a rajzból.

14.6.7. Alakzatok rajzolása

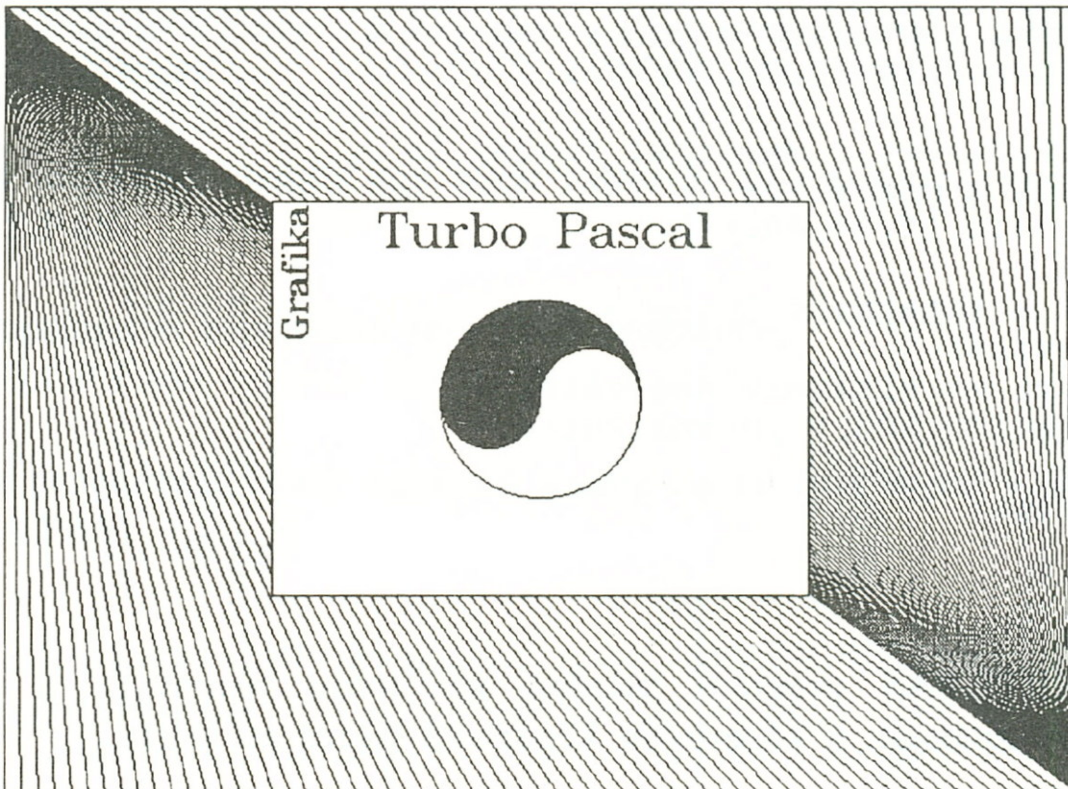
A RAJZ1.PAS program EGA, VGA monitoron jeleníti meg az összes rajzolható alakzatot, vonalat, görbeívet, alakzat festését is demonstrálja.

A RAJZ2.PAS program EGA,VGA monitoron bemutatja a betűkészleteket. A szöveg nagyságának és irányának programozásával.

A RAJZ3.PAS program EGA monitoron a véletlenszám generátorral képzett színkódot, véletleszszám generátorral megadott helyre teszi ki a ***PutPixel*** eljárás.

A PALETTAD.PAS program EGA monitoron bemutatja a paletta átdefiniálását.

A GPLD.PAS program EGA, VGA monitoron egy szép grafikus képet hoz létre. 2500 ms ideig kimerevíti a képet, majd a grafikus módot lezárva, visszatér szöveges módba.



14.6.8. Kép kivágása és újrahelyezése

A grafikus képernyő egy részének a memóriába való mentése, majd újra megjelenítése demonstrálja a KIVAG.PAS program.

```
(* Kivag.pas *)
program kivag;
  uses Graph;
  { ImageSize function }
  var      Gd,Gm: integer;
               P: pointer;
            Size: word;
      Procedure grstart;
      begin
         Gd:=Detect;
         Initgraph(Gd,Gm,'');
         if GraphResult<>GrOk then Halt(1);
      end;
  begin
  grstart;
  { A teljes kepernyot kifesti }
  Bar(0,0,GetMaxX,GetMaxy);

  { Lekeri a kivagando negyzet meretet }
  Size:=ImageSize(10,20,30,40);

  GetMem(P,Size); { memoriat foglal le }
  { Elteszi a memoriaba }
  GetImage(10,20,30,40,P^);
  Readln;
  { torli a kepernyot }
  ClearDevice;
  { Kiteszi az eltarolt negyzetet }
  PutImage(100,100,P^,NormalPut);
  Readln;
  CloseGraph;
  end.
```

A V_FEKETE.PAS program minden monitoron bemutatja a ***getimage***, ***putimage*** használatát, a képernyőn képezhető összes művelettel.

A V_SZINES.PAS program EGA, VGA monitoron színesen mutatja be a **getimage** és a ***putimage*** használatát.

Ellenőrző kérdések:

1. Milyen módokban használhatják a Pascal programok a PC képernyőjét?
2. Melyik unit tartalmazza a grafikus eljárásokat?
3. Mire szolgál a .BGI file?
4. Mi a kiterjesztése a karakterkészletnek?
5. Milyen eljárással kell a grafikus módot megnyitni és mivel zárni?
6. Milyen módszerrel lehet a programba befordítani a grafikus meghajtót?
7. Melyek a koordinátái a grafikus képernyő bal felső sarkának?
8. Mivel lehet a grafikus képernyőt kimerevíteni, hogy a felrajzolt ábra a képernyőn maradjon?

Feladatok:

1. Írjon programot, amely grafikus módban kirajzolja az olimpiai karikákat.
   (OLIMPIA.PAS)

2. Írjon programot, amely a képernyőn adott osztásban négyzethálót rajzol.
   (RASTER.PAS)

3. Írjon programot, amely egy házat rajzol ki és a falakat kifesti.
   (HAZ.PAS)

4. Oszlopdiagram rajzolására tervezzen programot.
   (OSZLOP.PAS)

5. Írjon programot, amely egyszerű analóg-digitál órát rajzol a képernyőre.
   (ORA.PAS)

Használja a Scriptum Kft.

szótárprogramjait

DOS, NOVELL és WINDOWS

környezetben!

Kétnyelvű szótárak:

— rezidens, grafikus megvalósítású számítógépes könyv,
— kifejezések, szóhasználati példamondatok,
— fonetika, nyelvtani információk.

Közös kiadásban az Akadémiai Kiadóval:

Angol-Magyar,
Magyar-Angol

(42 000, ill. 55 000 szó és kifejezés.)

Felhasználói (szak)szótár:

— többnyelvű bővíthető, módosítható,
— együttműködik a kétnyelvű szótárakkal,
— grafikus, rezidens megvalósítás.

Modulonkénti ár: 4000-6000 Ft.
Kedvező hálózati árak!

Forgalmazók

Scriptum Kft.
6771 Szeged
Mályva u. 34.
Tel./Fax: (62) 355-722
Levél:
6771 Szeged-Szőreg, Pf.: 2.

ComputerBooks Kft.
Budapest, XII., Tartsay V. u. 12.
Telefon: 175-1564
Fax: 175-3591
Levél:
1253 Budapest, Pf.: 71.

15. MINTAFELADATOK

15.1. Másodfokú egyenlet megoldása

Az együtthatók ismeretében oldjunk meg egy másodfokú egyenletet. A program vizsgálja meg, hogy az egyenletnek van-e megoldása, egy megoldása van-e, illetve a gyökei valósak vagy komplexek.

A feladat matematikai modellje

Az a,b és c együtthatók ismeretében a másodfokú egyenlet alakja

$$a \cdot x^2 + b \cdot x + c = 0$$

A programban az alábbi vizsgálatokat kell elvégezni

1. Ha az a,b,c értéke zérus, akkor az egyenletnek bármelyik szám a megoldása.
2. Ha a,b zérus és c nem zérus, akkor az egyenlet ellentmondásos, nincs megoldás.
3. Ha az a zérus, de a b és c nem zérus, akkor az egyenlet elsőfokú

$$x = \frac{-c}{b}$$

4. Ha a,b és c nem zérus, akkor kiszámítjuk a diszkrimináns értékét:

$$d = b^2 - 4 \cdot a \cdot c$$

4.a. Ha a d előjele negatív, az egyenletnek komplex megoldása van
A gyök valós része az xv a képzetes része xk

$$xv = \frac{-b}{2 \cdot a} \qquad xk = \frac{\sqrt{|d|}}{2 \cdot a}$$

akkor a két komplex gyök

$$x_1 = xv + xk \cdot i$$

$$x_2 = xv - xk \cdot i$$

4.b. Ha a diszkrimináns értéke zérus, akkor az egyenletnek csak egy megoldása van

$$x = \frac{-b}{2 \cdot a}$$

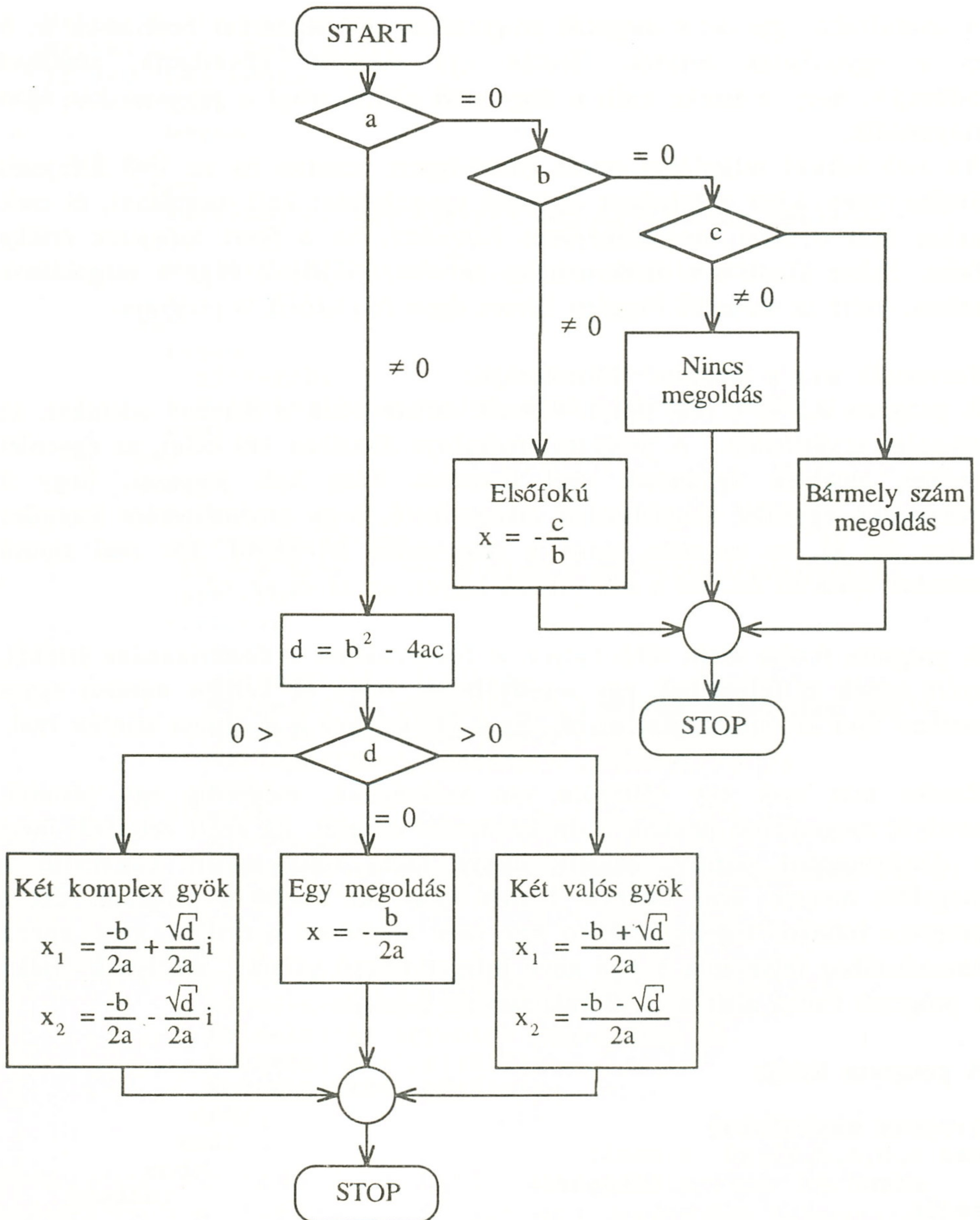
4.c. Ha a diszkrimináns értéke pozitív, akkor az egyenletnek két valós gyöke van

$$x_1 = \frac{-b + \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

A program felépítése

A másodfokú egyenlet alogitmusának a felépítését a 15.1. ábrán látható blokkdiagram mutatja a legszemléletesebben.



15.1. ábra Másodfokú egyenlet megoldásának blokkdiagramja

A másodfokú egyenletet megoldó programban első lépésként beolvassuk ***a***, ***b*** és ***c*** együtthatók értékét. Ezután egy vizsgálat következik, amellyel eldöntjük, hogy ***a*** értéke nulla-e vagy attól eltérő, majd a program két ágon folytatódik.
Az a=0 feltétel teljesülése esetén (másképpen mondva ha az a=0 kifejezés értéke **true**) akkor a feladatot elsőfokú egyenletként kell megoldani, és csak akkor kell a fenti megoldóképletet használni, ha a fenti kifejezés értéke **false**. Ekkor azonban a diszkrimináns (b^2*-4ac)* előjelétől függ a megoldások száma, ezért az előjeltől függően három ágon folytatódik a program.

Tervezzük meg a program változólistáját:
A program *a,b* és *c* **real** típusú változói tartalmazzák a bemenő adatokat, az egyenlet együtthatóit. A program eredménye általában két szám, az egyenlet gyökei, amelyek ugyancsak **real** típusúak. Meg kell jegyezni, hogy a másodfokú egyenlet megoldásakor elképzelhető, hogy eredményként komplex szám jön ki, ez azonban, mint a programból követhető, két **real** típusú számból építhető fel. Ez a két változó legyen az *x1* és az *x2*.

A program futása során több helyen is felhasználjuk a diszkrimináns értékét, ezért ennek is felveszünk egy segédváltozót, hogy ne kelljen minden egyes esetben újra és újra kiszámolnunk. Ez a változó lesz a *d*, típusa szintén **real**.

Ezután már csak egy változóra van szükségünk, mégpedig egy olyanra, amelyik megmutatja nekünk a diszkrimináns előjelét. Ez azért célszerű, mert a diszkrimináns pozitív, negatív illetve nulla volta esetén különböző a megoldás menete, azaz összesen három ágon fut tovább a program. Ezt a program írásakor legelegánsabban egy **case** utasítással oldhatjuk meg, ennek használatához felvesszük a *sign* nevű **integer** típusú változót, amelynek értéke a program futása alatt a -1, 0 valamint az 1 lehet.

A program listája

```
program masodfoku;
var a,b,c,d,x1,x2  : real;
    sign           : integer;
begin
   Writeln('   Másodfokú egyenlet megoldása');
   Writeln('Ax^2+Bx+C=0 egyenlet együtthatóinak megadása ');
   Write('A='); ReadLn(a);
```

```
   Write('B='); ReadLn(b);
   Write('C='); ReadLn(c);
   if a=0 then begin
    if b=0 then
           begin
     if c=0
         then
           WriteLn('Az egyenletnek bármelyik szám megoldása')
         else
           WriteLn('Ellentmondás, nincs megoldás');
           end
         else
           begin
             x1:=-c/b;
             WriteLn('Az egyenlet elsôfokú, x=',x1:8:4);
           end;
         end
         else
          begin
            d:=sqr(b)-4*a*c;
            if d=0 then sign:=0 else sign:=round(d/abs(d));
           case sign of
           -1:begin
              x1:=-b/2/a;
              x2:=sqrt(abs(d))/2/a;
              Writeln('Az egyenletnek komplex megoldása van');
              Writeln('X1=',x1:8:4,'+',x2:8:4,'*i');
              Writeln('X2=',x1:8:4,'-',x2:8:4,'*i');
              end;
           0:begin
              x1:=-b/2/a;
              WriteLn('Az egyenletnek csak egy megoldása van');
              Writeln('X=',x1:8:4);
             end;
           1:begin
              x1:=(-b+sqrt(d))/2/a;
              x2:=(-b-sqrt(d))/2/a;
              WriteLn('Az egyenlet megoldásai');
              WriteLn('X1=',x1:8:4);
              WriteLn('X2=',x2:8:4);
             end;
            end;
          end;
     ReadLn;
end.
```

A programban két fontos dolgot követhetünk nyomon. Az egyik a **begin** - **end** utasítászárójelek használata, a másik pedig a többágú elágazás, a **case** utasítás. Az utasítászárójelek szerepe az, hogy több utasítást egybefogjon, és például az **if** utasításban lehetővé tegye több utasítás ugyanazon feltételtől függő végrehajtását.

A program futtatási eredménye

1. Futási eredmény

```
   Másodfok egyenlet megoldása
Ax^+Bx+C=0 egyenlet együtthatóinak megadása
A=1
B=2
C=-15
Az egyenlet megoldásai
X1=  3.0000
X2= -5.0000
```

2. Futási eredmény:

```
   Másodfokú egyenlet megoldása
Ax^+Bx+C=0 egyenlet együtthatóinak megadása
A=1
B=1
C=2
Az egyenletnek komplex megoldása van
X1= -0.5000+  1.3229*i
X2= -0.5000-  1.3229*i
```

3. Futási eredmény:

```
   Másodfokú egyenlet megoldása
Ax^+Bx+C=0 egyenlet együtthatóinak megadása
A=0
B=1
C=2
Az egyenlet elsõfokú, x= -2.0000
```

A másodfokú egyenlet megoldására készíthetünk olyan programot, amely feladatmegoldást eljárásokkal oldja meg. A MASOD.PAS program eljárással működő változata a MASOD_FG.PAS program.

A MASOD_F2.PAS programban teljesen különválasztottuk a feladat megoldásától az eremény kiíratását. A MASOD eljárás a *jelez* formális paraméterében jelzi a megoldás típusát. Az *eredmeny_kiir* eljárás kiírja az eredményt.

```
(* masod_f2.pas *)
program masodfoku;
var
    a1,b1,c1,
    x1,x2,x1i,x2i : real;
                j : integer;

   procedure olvas(var a,b,c :real);
   begin
     Writeln('Másodfokú egyenlet megoldása');
     Writeln('Ax^2+Bx+C=0 egyenlet együtthatóinak megadása');
     Write('A='); ReadLn(a);
     Write('B='); ReadLn(b);
     Write('C='); ReadLn(c);
   end;

   procedure eredmeny_kiir(x1,x2:real; jelez:integer);
   begin
      case jelez of
      1: WriteLn('Az egyenletnek bármelyik szám megoldása');
      2: WriteLn('Ellentmondás, nincs megoldás');
      3: WriteLn('Az egyenlet elsõfokú, x=',x1:8:4);
      4: begin
            Writeln('Az egyenletnek komplex megoldása van');
            Writeln('X1=',x1:8:4,'+',x2:8:4,'*i');
            Writeln('X2=',x1:8:4,'-',x2:8:4,'*i');
         end;

      5: begin
            WriteLn('Az egyenletnek csak egy megoldása van');
            Writeln('X=',x1:8:4);
         end;
      6: begin
            WriteLn('Az egyenlet megoldásai');
            WriteLn('X1=',x1:8:4);
            WriteLn('X2=',x2:8:4);
         end;
     end;
   end;
```

```
   procedure masod(a,b,c: real; var x1,x2:real;
                   var jelez: integer);
   var
      d  : real;
     sign: integer;
   begin
     if a=0 then begin
                  if b=0 then begin
                                 if c=0 then jelez:=1
                                        else jelez:=2;
                                end
                         else begin
                                 x1:=-c/b;
                                 jelez:=3;
                                end;
                 end
            else begin
                  d:=sqr(b)-4*a*c;
                  if d=0 then sign:=0
                         else sign:=round(d/abs(d));
                  case sign of
                      -1 : begin
                              x1:=-b/2/a;
                              x2:=sqrt(abs(d))/2/a;
                              jelez:=4;
                             end;
                       0 : begin
                              x1:=-b/2/a;
                              jelez:=5;
                             end;
                       1 : begin
                              x1:=(-b+sqrt(d))/2/a;
                              x2:=(-b-sqrt(d))/2/a;
                              jelez:=6;
                             end;
                  end;
                 end;
     end;
begin
     olvas(a1,b1,c1);
     masod(a1,b1,c1,x1,x2,j);
     eredmeny_kiir(x1,x2,j);
end.
```

15.2. Adatok rendezése

Rendezésnek nevezzük azt a folyamatot, amikor egy halmaz elemeit valamilyen szabály szerint sorba (növekvő vagy csökkenő) rendezzük. A rendezés meggyorsítja az elemek későbbi keresését, mivel egy rendezett halmazon való keresés jóval gyorsabban megy végbe. A rendezés kiemelkedően fontos tevékenység az adatfeldogozásban során. Sokfajta rendező algoritmus van, mindegyik ugyanazt a feladatot végzi, mégis van közöttük eltérés.

A rendező módszereket két csoportba sorolhatjuk: tömbök rendezése és (soros) file-ok rendezése. Gyakran a két csoportot nevezik még belső ill. külső rendezésnek, mivel a tömbök rendezése a számítógép gyors, belső tárában (memóriájában) zajlik, míg a file-ok a lassúbb, de tágasabb külső tárolón (winchester-en) megy végbe.

A tömbök rendezése

A legfontosabb követelmény a tömbök rendezőmódszereitől, hogy a rendelkezésre álló tárat gazdaságosan használják ki. Az ismertetett módszerek az adatok átrendezését a saját helyükön hajtják végre. A közvetlen módszerek igen alkalmasak arra, hogy megértsük a legfontosabb rendezőelveket, mivel a programjaik könnyen érthetőek, viszonylag rövidebbek, tehát kevesebb művelettel működnek, kevesebb helyet foglalnak le a memóriában. A bonyolultabb algoritmusok kevesebb lépést tartalmaznak, de ezek a lépések részleteikben jóval bonyolultabbak. Megjegyezzük, hogy a közvetlen módszerek elég kis n-re gyorsabbak, míg a nagyobb n-re már nem érdemes őket használni.

Az ismertetésre kerülő módszerek az eredeti helyükön rendezik át az adatokat növekvő sorrendre. Az alkalmazott módszereket az alábbi csoportba sorolhatjuk:

1. rendezés cserével
2. rendezés beszúrással
3. rendezés kiválasztással

15.2.1. Rendezés cserével

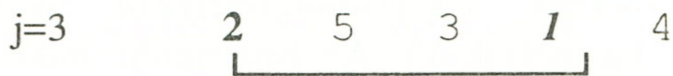
Nézzünk egy példát egész típsusú tömb adatainak rendezésére adatok cseréjével:
Legyen az adatok : *n*
Az elv a következő:
Két egymásba ágyazott ciklussal vizsgáljuk az elemeket úgy, hogy a külső i ciklus egytől *n-1* -ig, a belső *j* ciklus pedig *i+1* -től *n*-ig vizsgálja az elemeket. A rögzített *i*-edik elemet hasonlítjuk össze a mögötte lévő elemekkel és cseréljük, ha a mögötte lévő elem kisebb.
A rögzített elem **vastag** kiemeléssel, az összehasonlítandó elem pedig ***vastag, dőlt*** kiemeléssel szerepel a példában.

A tömb indexe: 1 2 3 4 5

i=1

j=2 **5** 2 ***3*** 1 4

j=3 **2** 5 3 ***1*** 4

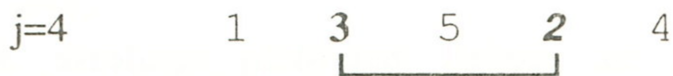


j=4 **1** 5 3 2 4 nincs csere

i=2

j=3 1 **5** ***3*** 2 4

j=4 1 **3** 5 ***2*** 4



j=5 1 **2** 5 3 ***4*** nincs csere

i=3

j=4	1	2	**5**	***3***	4	
j=5	1	2	**3**	5	***4***	nincs csere

i=4

j=5	1	2	3	**5**	***4***

Rendezve	1	2	3	4	5

A RENDEZN.PAS program mutatja be az elemek cseréjét.

```
(*  rendezn.pas *)
program rendez_integer_adatot;
{ Egesz tipusu adatok rendezese }
(* deklaracio *)
var x         : array[1..50] of integer;
    n,i,j,s   : integer;
(*  fo program  *)
begin
(* adatok beolvasasa *)
   writeln;
   writeln('Egesz tipusu adatok rendezese novekvo sorrendben');
   writeln;
   repeat
      write('Az adatok szama: '); readln(n);
      if n>50 then writeln('Hibas adat: max 50! ');
   until n <= 50;
    for i:=1 to n do
    begin
      write('x[',i:2,']= '); readln(x[i]);
    end;

 (* adatok rendezese *)
   for i:=1 to n-1 do
    for j:=i+1 to n do
```

```
      (* csokkeno sorrend: x[j]>x[i] javitassal *)
       if x[j]<x[i] then
           begin
                s:=x[i];
                x[i]:=x[j];
                x[j]:=s;
           end;

  (* rendezett adatok kiirasa *)
    writeln;
    writeln('Rendezett adatok');

    writeln('Az adatok szama: ',n:3);
    writeln;
    for i:=1 to n do
    writeln('x[',i:2,']= ',x[i]);

end.
```

A feladatot oldjuk meg eljárás használatával: RENDEZ_I.PAS egész típusú adatok rendezésére alkalmas.

```
(*  rendez_i.pas *)
program rendez_integer_adatot;
{ Egesz tipusu adatok rendezese }
uses crt;
const n=50;
type vekt=array[1..n] of integer;
var y:vekt;
    k:integer;

procedure rend(var x:vekt;m:integer);
var i,j,s:integer;
begin
   for i:=1 to m-1 do
    for j:=i+1 to m do
      if x[j]<x[i] then
          begin
               s:=x[i];
               x[i]:=x[j];
               x[j]:=s;
          end;
end;
```

```
procedure olvas(var x:vekt;var m:integer);
var i:integer;
begin
   writeln;
   writeln('Egesz tipusu adatok rendezese novekvo sorrendben');
   writeln;
   repeat
    write('Az adatok szama: '); readln(m);
    if m>n then writeln('     Hibas adat: max 50 !');
   until m<=n;
    for i:=1 to m do
    begin
      write('x[',i:2,']= '); readln(x[i]);
    end;
end;

procedure kiir(x:vekt;m:integer);
var i:integer;
begin
writeln;
writeln('Rendezett adatok');

    writeln('Az adatok szama: ',m:3);
    writeln;
    for i:=1 to m do
    writeln('x[',i:2,']= ',x[i]);
end;

begin
  clrscr; (* kepernyo torlese *)
  olvas(y,k);
  rend(y,k);
  kiir(y,k);
end.
```

15.2.2. Sztring típusú adatok rendezése

Sztring típusú adatok rendezése hasonlóan történik a numerikus adatok rendezéséhez, természetesen az ékezetes karakterek nem kerülnek abc sorrendbe. A rendezés csak ékezet nélküli karakterláncra működik helyesen.
A NEVREND.PAS program maximálisan 50 darab 30 karakteres nevet rendez.

```
(* nevrend.pas *)
program nevrendn;
{ sztringet tartalmazo tomb abc sorrendbe torteno rendezese }
type
  str=string[30];
var
  a:array[1..50] of str;
  i,j,n:integer;
  g:str;
  k:boolean;
begin
  writeln('Max. 50  nev rendezese');
  writeln;
  repeat
     write('Nevek szama: '); readln(n);
     if n>50 then writeln('Hibas adat: max 50 !');
  until n <= 50;
  writeln;
  for i:=1 to n do
    begin
      write(i,'. nev : ');
      readln(a[i]);
    end;
  writeln;
  for i:=1 to n-1 do
    for j:=i to n do
      if a[i]>a[j] then
           begin
             g:=a[i];
             a[i]:=a[j];
             a[j]:=g;
           end;
  for i:=1 to n do
    writeln(i,'. nev : ',a[i]);
end.
```

15.2.3. Különféle rendező algoritmusok

Egyszerű menüvel vezérelt programban kiválaszthatunk rendezést közvetlen beszúrással, valamint buborék módszerrel is.

Rendezés közvetlen beszúrással

A rendezés közvetlen beszúrásnál az algoritmus úgy működik, hogy minden lépésben, 2-től egyesével m darab adat esetén m-ig kiemeljük a csökkenő sorozat szerinti első elemet és beszúrjuk a megfelelő helyre.

Alap	5	***2***	3	1	4
1. lépés	2	5	***3***	1	4
2. lépés	2	3	5	***1***	4
3. lépés	1	2	3	5	***4***
4. lépés	1	2	3	4	5

Buborék rendezés

Ha a tömböt egy víztartályhoz hasonlítjuk és a rendezendő tételeknek a buborékok súlyát feleltetjük meg, akkor minden lépés egy buborék felemelkesését eredményezi a súlyának megfelelő szintre.
Ennél az algoritmusnál előfordul, hogy már nincs szükség cserére, mert az adatok már rendezve voltak, mégsem áll le a rendezési művelet. Ezen úgy javíthatunk, hogy minden menetben feljegyezzük, hogy történt-e csere a menet alatt. Az első olyan menetnél, ahol nem történt csere, az algoritmus befejezheti a munkáját. Sőt tovább javítható az algoritmus úgy, hogy megjegyezzük az utolsó csere indexét. Ha ez az index k, akkor ez azt jelenti, hogy a k alatti adatok már jó sorrenben állnak.

Alap	5	2	3	***1***	4
1. lépés	1	**5**	***2***	3	4
2. lépés	1	2	5	***3***	4
3. lépés	1	2	3	5	***4***
4. lépés	1	2	3	4	5

A RENDEZM.PAS mutatja be rendező algoritmusok pascal programját.

```
(* rendezm.pas *)
(* Egesz tipusu adatok rendezese menuvel *)
(* tobb modszerrel *)
program rendezes_menuvel;
uses crt;
const db =100;
type
       itype=array[1..db] of integer;
var
           y: itype;
       n,jel: integer;

procedure olvas(var x:itype; var m:integer);
var i:integer;
begin
   writeln;
   writeln('Rendezendo (integer tipusu) adatok');
 repeat
   write('Az adatok szama : '); readln(m);
   if m>db then writeln('Az adatok max. szama : ',db);
 until m<=db;
   for i:=1 to m do
   begin
      write(' [',i:2,']= '); readln(x[i]);
   end;
end;

procedure kiir(x:itype; m,sorszam:integer);
var i:integer;
begin
   writeln;
   writeln('Rendezett adatok');
```

```
   write('Rendezes modszere: ');
   case sorszam of
     1: writeln('csere');
     2: writeln('kozvetlen csere');
     3: writeln('buborek');
   end;
   writeln('Az adatok szama : ',m);
   for i:=1 to m do
   begin
      writeln(' [',i:2,']= ',x[i]);
   end;
   writeln;
end;
procedure rend(var x:itype; m:integer);
var i,j,s:integer;
begin
   for i:=1 to m-1 do
    for j:=i+1 to m do
      if x[j]<x[i] then
          begin
               s:=x[i];
               x[i]:=x[j];
               x[j]:=s;
          end;
end;

procedure beszur(var x:itype; m:integer);
var i,j,k:integer;
    buf  :integer;
begin
    for i:=2 to m do
    begin
       buf:=x[i];
       j  :=i-1;
       while (j>0) and (buf<x[j]) do
       begin
          x[j+1]:= x[j];
               j := j-1;
       end;
        x[j+1]:= buf;
    end;
end;

procedure buborek(var x:itype; m:integer);
var i,j :integer;
    buf : integer;
begin
```

```
    for i:=2 to m do
    begin
       for j:=m downto i do
         if x[j-1]>x[j] then
         begin
            buf   :=x[j-1];
            x[j-1]:=x[j];
            x[j]  :=buf;
        end;
    end;
end;

procedure menu(var sorszam:integer);
var
        s  : string;
        kod: integer;
begin
  repeat
     clrscr;  (* kepernyo torlese *)
     writeln;
     writeln('       RENDEZES ');
     writeln;
     writeln(' 1.  Csere ');
     writeln(' 2.  Kozvetlen csere ');
     writeln(' 3.  Buborek modszer ');
     writeln(' 4.  Kilepes         ');
     writeln;
     write('A modszer sorszama: '); readln(s);
     val(s,sorszam,kod);
     if (kod<>0) or (sorszam<=0) or (sorszam>4)
               then
                 begin
                   writeln('Hibas menusorszam');
                   writeln('      Nyomj Enter-t');
                   readln;
                 end;
  until (kod=0) and (sorszam>0) and (sorszam<=4);
end;
(* rendezes foprogramja *)
begin
     menu(jel);
     clrscr;
     case jel of
       1: begin
             olvas(y,n);
             rend(y,n);
          end;
```

```
        2: begin
              olvas(y,n);
               kozvetlen(y,n);
           end;
        3: begin
               olvas(y,n);
               buborek(y,n);
           end;
        4: exit;
     end;
     kiir(y,n,jel);
end.
```

A program eredménye:

```
     RENDEZES

1.   Csere
2.   Közvetlen beszuras
3.   Buborek modszer
4.   Kilepes

A modszer sorszama: 1

Rendezendo (integer tipusu) adatok
Az adatok szama: 5
 [ 1]= 5
 [ 2]= 2
 [ 3]= 3
 [ 4]= 1
 [ 5]= 4

Rendezett adatok
Rendezes módszere: csere
Az adatok szama: 5
 [ 1]= 1
 [ 2]= 2
 [ 3]= 3
 [ 4]= 4
 [ 5]= 5
```

Rendezés quick (gyors) módszerrel

A REKORD.PAS bemutatja rekord típusú adatok rendezését *quick* módszerrel. A *qs* eljárás *rekurzív* módon saját magát hívja. A rekurzió ilyen felhasználása igen hatékony eszköz az algoritmusokban. Az *iteratív* megoldás lényege az, hogy a felosztások végrehajtási igényét feljegyezzük egy várakozó listára. Minden lépésben két újabb felosztási feladat keletkezik. Csak az egyiket lehet azonnal elkezdeni a soronkövetkező iterációban, a másikat erre a várakozó listára helyezzük. Legyen a következő rekord:

```
  cim = record
          nev    : string[25];
          utca   : string[20];
          varos  : string[20];
          orszag: string[2];
          kod    : string[5];
       end;
```

amely tartalmazza egy személy nevét lakcímét, irányító szám, ország, város és utca adatokkal.

```
(* rekord.pas *)
program rekord_rendezes;
{ rekord rendezes quick modszerrel }

const db=50;
type
      cim = record
          nev    : string[25];
          utca   : string[20];
          varos  : string[20];
          orszag: string[2];
          kod    : string[5];
       end;
     cim_t = array[1..db] of cim;

var   lakos : cim_t;
      r     : cim;
      n     : integer;

procedure olvas_rec(var x:cim_t; var m:integer);
var i:integer;
begin
   writeln('Rendezendo adatok');
 repeat
   write('Az adatok szama : '); readln(m);
   if m>db then writeln('Az adatok max. szama : ',db);
```

```
 until m<=db;
   for i:=1 to m do
   begin
      write(' Nev     (max. 25 kar.)  = '); readln(x[i].nev);
      write(' Utca    (max. 20 kar.)  = '); readln(x[i].utca);
      write(' Varos   (max. 20 kar.)  = '); readln(x[i].varos);
      write(' Orszag  (max. 2 kar. )  = '); readln(x[i].orszag);
      write(' Kod     (max. 5.kar. )  = '); readln(x[i].kod);
      writeln;
   end;
end;

procedure kiir_rec(x:cim_t; m:integer);
var i:integer;
begin
   writeln('Rendezett adatok ');
   writeln('Az adatok szama ; ',m);
   for i:=1 to m do
   begin
      writeln(' Nev    : ',x[i].nev);
      writeln(' Utca   : ',x[i].utca);
      writeln(' Varos  : ',x[i].varos);
      writeln(' Orszag : ',x[i].orszag);
      writeln(' Kod    : ',x[i].kod);
      writeln;
   end;
   writeln;
end;

procedure qs_rekord(var x:cim_t; m:integer);
   procedure qs(l,r:integer; var xx:cim_t);
   var  i,j       :integer;
        buf1,buf2:cim;
   begin
      i:=l; j:=r;
      buf1:=xx[(l+r) div 2];
      repeat
        while xx[i].nev < buf1.nev  do i:=i+1;
        while buf1.nev  < xx[j].nev do j:=j-1;
        if i<=j then
        begin
           buf2 :=xx[i];
           xx[i]:=xx[j];
           xx[j]:=buf2;
           i:=i+1; j:=j-1;
        end;
     until i>j;
```

```
        if l<j then qs(l,j,xx);
        if l<r then qs(i,r,xx)
     end;
begin
        qs(1,m,x);
end;
begin
 writeln('REKORD ADATAINAK RENDEZESE');
  (* rekord adatok rendezese  *)
    olvas_rec(lakos,n);
    qs_rekord(lakos,n);
    kiir_rec(lakos,n);
end.
```

A RENDEZ2.PAS a közvetlen beszúrássos rendezést teszteli egész adatokkal.

A RENDEZ3.PAS a buborék módszerrel történő rendezést teszteli.
A SORT_RF.PAS rekord adatokat rendez file-ban *quick* módszerrel. Az adatokat a megadott néven diszk file-ra viszi fel a *kiir_rec* eljárás, a rendezett adatokat írja vissza az *olvas_rec_rend* eljárás, a vizsgálandó adatok kiírására szolgál az *olvas_rec_vizsg* eljárás. A közvetlen elérésű diszk file adatit quick módszerrel rendezi a *qs_rekord* eljárás, amely használja a keres nevű eljárást.

A TERM_F.PAS a természetes összefésülés algoritmusának programját mutatje be.

A KERESES.PAS programban a rendezetlen tömbben a szekvenciális keresés a tömb első elemétől kezdődik és addig tart, míg vagy megtalálta a keresendő elemet vagy a tömb a végére ért. Az eljárás használható rendezetlen és rendezett halmazokon is.

Ha az adathalmaz rendezett, akkor a bináris keresés felhasználásával gyorsabb megoldást kapunk.

16. A TURBO PASCAL SPECIÁLIS LEHETŐSÉGEI

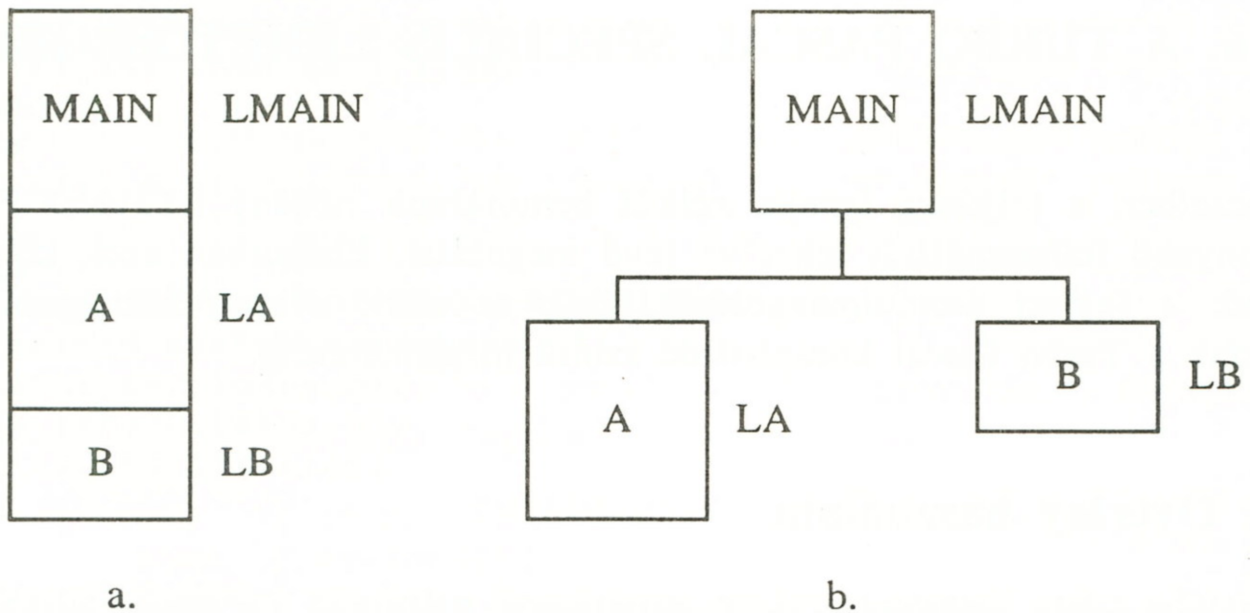
A fejezetben a teljesség igénye nélkül bemutatunk néhány a Turbo Pascal hatékonyabb felhasználását lehetővé tevő megoldást. Elsősorban azok számára ajánljuk a fejezet áttanulmányozását, akik az előző részek feldolgozásával eljutottak a Turbo Pascal középhaladó szintű megismeréséig.

16.1. Overlay használata

Mint ahogy már említettük a Turbo Pascal modulok kódjának maximális mérete 64 kbyte, azonban a felhasznált modulok száma nem korlátozott. Ily módon elképzelhető olyan program előállítása, amely kódja már nem fér be a memóriába. Hasonló problémával találkozunk akkor is, ha nagyobb méretű programból sok adatot kell feldolgoznunk.

Ezen problémák áthidalására a Turbo Pascal az *overlay* (átlapolás) technikáját tartalmazza. Az *overlay* az operatív memória használatának olyan módja, amely során a memóriának ugyanazon részét *(overlay puffer)* más-más modulok kódjai foglalják el. Az *overlay* modulok a lemezen kerülnek tárolásra, és a program igényének megfelelően onnan töltődnek be.

A 16.1. ábra szemlélteti a különbséget az *overlay* megoldást felhasználó és az *overlay* nélküli programok szerkezete között. Tegyük fel, hogy a programunk három modulból áll: *MAIN* (a főprogram), *A* és *B*. Legyen az egyes modulok kódjának hossza *LMAIN*, *LA* és *LB* *(LA > LB)*. Igy az *overlay* nélküli program kódja *LMAIN+LA+LB* memóriát foglal le, míg az *overlay* megoldás használatával a program kódjának memóriaigénye *LMAIN+LA*-ra csökken.



16.1. ábra
Az *overlay* nélküli (a) és az *overlay*-t használó (b) program szerkezete.

A program indításakor valamelyik modul azonnal betöltődik a memóriába a (fordító határozza meg), például az A. Ha valamely B-ben tárolt rutinra van szükség akkor a B betöltődésével az A részben felülíródik. Majd, ha újra A-beli rutinra hivatkozunk, akkor az A visszatöltődik.

A Turbo Pascal-ban a fenti séma szerinti működés bonyolultabban valósul meg. A ciklikus szervezésű *overlay puffert* az *overlay* manager kezeli, amely egyszerre több *overlay* modult is képes a memóriában tartani. A Turbo Pascal 5.5-ös verziójától kezdve az *overlay* manager az *overlay* rutinok hívási gyakorisága alapján dönt arról, hogy a puffer betelte után melyik modult törölje.

Az *overlay* modulok lemezről való betöltés elég időigényes, ezért a Turbo Pascal támogatja az *expanded* memória (EMS) felhasználását az *overlay* file tárolására, ahonnan a modulok betöltése sokkal gyorsabban elvégezhető.

Az *overlay* file (.OVR) egyszerű másolással hozzákapcsolható az .EXE file-hoz a:

```
COPY /B PROGRAM.EXE + PROGRAM.OVR
```

DOS parancs alkalmazásával. Ebben az esetben a programot a *{$D-,L-}* kapcsolókkal ajánlott lefordítani.

Az *overlay* rendszer működéséhez az OVERLAY szabványos modul szükséges. A *unit*-ban található nagy számú konstans, változó, eljárás és függvény ismertetését a függelék tartalmazza.

Tekintsük át az *overlay* program kialakításához szükséges lépéseket:

1. Első lépésben el kell különítenünk a főprogramot és az *overlay* modulok tartalmát. Érdemes az egyes részeket úgy összeállítani, hogy minimális mennyiségű töltögetésre legyen szükség. Meg kell jegyeznünk, hogy *overlay* modul nem tartalmazhat megszakítást kezelő rutint (**interrupt**).

2. A főprogram elején a {$O *modulnév*} fordítási direktíva felhasználásával fel kell sorolnunk az *overlay* modulokat, a *uses* parancsban pedig meg kell adnunk az *overlay* nevű modult is:

   ```
   Program Main;
   uses DOS,Overlay, UnitA, UnitB;
   {$O DOS}
   {$O UnitA}
   {$O UnitB}
   ```

   A szabványos modulok közül a DOS **unit** használható egyedül mint *overlay* modul.

3. A főprogram és minden *overlay* modul elején a {$F+,O+} fordítási direktívákat kell elhelyeznünk.

4. Végezetül az *overlay* rendszer inicializálását az

   ```
   OvrInit(overlay_file_neve);
   ```

   eljárás hívásával kell elvégeznünk. Ha az *overlay* modulok egyike sem tartalmaz inicializációs rész, akkor ezt a hívást általában a főprogramból

hajtjuk végre. Ellenkező esetben azonban a második példánkban bemutatott megoldást kell használnunk, ahol az *OvrInit* eljárás hívása az *OvrStart* modulból történik. (A főprogram a **uses** sorában az *overlay* modulok neve előtt kell megadnunk ezt a modult.)
Az EMS memória használata az *OvrInit* hívását követő *OvrInitEMS* hívással kérhető. Ajánlott minden, az OVERLAY szabványos unit-ban definiált, eljárás hívása után a *OvrResult* változó tesztelésével meggyőződnünk a művelet eredményességéről.

Az alábbiakban egy rövid példa felhasználásával mutatjuk be az *overlay*-es programszerkezet kialakításához szükséges lépéseket:

```
{ O_MAIN.PAS }
 Program Overlay_demo;
 {$F+,O+}
   uses overlay,o_unita,o_unitb;
   {$O o_unita}
   {$O o_unitb}
 begin
    OvrInit('O_main.ovr');
    ProcA;
 end.


 {O_UNITA.PAS}
 unit o_unita;
 {$F+,O+}
 interface
   uses O_UnitB;
   procedure proca;
 implementation
   procedure proca;
     const st='Procedure';
     begin
       writeln(st,' A');
       procb(st);
     end;
 end.
```

```
{O_UNITB.PAS}
unit o_unitb;
{$F+,O+}
interface
  procedure procb(s:string);
implementation
  procedure procb;
    begin
      writeln(s,' B');
    end;
end.
```

A második példánk inicializációs részt tartalmazó *overlay* modulokból álló program ajánlott felépítését mutatja be:

```
{OVR_1.PAS}
unit ovr_1;
{$O+,F+}
interface
   procedure msg1;
implementation
  var
    s:string;
  procedure msg1;
    begin
      writeln(s);
    end;
begin
  s := '1. üzenet';
end.

{OVR_2.PAS}
unit ovr_2;
{$O+,F+}
interface
   procedure msg2;
implementation
  var
    s:string;
  procedure msg2;
    begin
      writeln(s);
    end;
begin
  s := '2. üzenet';
end.
```

```
{OVRSTART.PAS}
unit ovrstart;
interface
  uses overlay;
implementation
var
  ovrhiba:boolean;
  ovrhstr:string;
begin
  ovrinit('ovrtest.ovr');
  ovrhiba:=true;
  case ovrresult of
    ovrerror   : ovrhstr:='Definiálatlan hiba';
    ovrnotfound:ovrhstr:=
                 'Az overlay file nem található';
    ovrnomemory: ovrhstr:='Nincs elég memória';
    ovrIOerror :ovrhstr:=
                 'Az overlay file olvasási hiba';
    ovrnoEMSdriver:ovrhstr:=
                 'Az EMS driver nincs betöltve';
    ovrnoEMSmemory: ovrhstr:=
                 'Az EMS emória mérete kicsi';
    else
     ovrhiba:=false;
  end;
  if ovrhiba then
     begin
       writeln('OVR Hiba: ',ovrhstr);
       halt;
     end;
end.

{OVRTEST.PAS}
program ovrtest;
{$F+,O+}
  uses overlay, ovrstart, ovr_1, ovr_2;
  {$O ovr_1}
  {$O ovr_2}
begin
  msg1;
  msg2;
end.
```

16.2. Rendszerhívások

Az IBM PC számítógépek rendszerprogramja alapvetően két részből tevődik össze. A perifériák vezérlését végző **BIOS** (Basic I/O System) programból, és az erre ráépülő operációs rendszerből, például az **MS-DOS.**

A Turbo Pascal lehetőségeket biztosít a programozó számára, hogy a rendszerprogram bizonyos rutinjait (megszakítások - **interrupt**) közvetlenül meghívja. Maximálisan 256 ilyen rutin érhető el, amelyek belépési pontja (címe) a memória első 1 kbyte-jában került tárolásra, 4 byte-os elemeket tartalmazó tömbben.
Ezek a megszakítások közvetlenül nem hívhatók, mivel a paraméterezésükhöz a processzor regisztereit kell használnunk.

A DOS szabványos modul felhasználásával ezek a megszakítások elérhetők. A **unit**-ban a ***Registers*** nevű struktúra lefedi a processzor regisztereinek egy részét:

```
type
  Registers = record
    case Integer of
      0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
      1: (AL,AH,BL,BH,CL,CH,DL,DH        : Byte);
  end;
```

Registers típusú változó felhasználásával paraméterezhetjük a rendszerhívásokat, illetve ebben kapjuk vissza bizonyos szoftveres megszakítások eredményét.

A 256 különböző megszakítás közül tetszőleges aktivizálható az

intr*(intno:****byte;***var *regs:****registers);***

rutin segítségével, ahol az *intno* paraméterben a megszakítás rutin indexét kell megadni.

A $21 indexű megszakítás az operációs rendszer funkcióit tartalmazza (DOS funkciók), ezér ennek hívására külön rutint definiáltak:

*msdos(*var *regs:registers);*

(Ez a hívás ekvivalens a `intr($21,regs);` hívással)

Ahhoz, hogy tudjunk ilyen megoldásokat használni, rendelkeznünk kell a rendszerhívásokat tartalmazó leírással, amely alapján a paraméterezés egyszerűen elvégezhető.

Azok a megszakítások, amelyek a ***Registers*** struktúrában nem közölt regiszteren keresztül paraméterezhetők, ily módon nem érhető el.

Az alábbi példában a *Kursor* eljárással a $10-s BIOS megszakítást aktivizálva a szövegkurzor kikapcsolható vagy bekapcsolható. A *Verzio* eljárás pedig a DOS funkciók közül a $30-ast használja a DOS verziójának lekérdezésére.

```
program intr_dos;
 uses crt,dos;
 const ki=false;
       be=true;

 { Szövegkurzor ki- /bekapcsolása}
 procedure kurzor(ki_be:boolean);
 { ki_be: TRUE  - bekapcsolás
          FALSE - kikapcsolás }

 const
    cur: word=$0607;

 var
    r  : registers;

 begin
   case ki_be of

     false : { kikapcsolás }
             begin
```

```
                     with r do
                      begin
                        {a kurzoralak lekérdezése}
                        ah:=3;
                        bh:=0;
                        intr($10,r);
                        cur:=cx;
                        {a kurzoralak állítása}
                        ah:=1;
                        cx:=$2000;
                        intr($10,r);
                      end;
                   end;
           true   : { bekapcsolás }
                   begin
                     {a kurzoralak állítása}
                     r.ah:=1;
                     r.cx:=cur;
                     intr($10,r);
                   end;
         end;
       end;

       {DOS verziószán kiírása}
       procedure dosver;
         var
           r:registers;
       begin
           r.ah:=$30;
           MsDos(r);
           writeln('Az operációs rendszer verziószáma: ',
                            r.al, '.', r.ah);
       end;
       begin
         clrscr;
         dosver;
         gotoxy(10,10);
         kurzor(ki);
         write('Hol a kurzor? ');
         delay(2000);
         gotoxy(50,20);
         kurzor(be);
         write('Ihol ni!');
         delay(2000);
       end.
```

16.3. Megszakítási rutinok készítése - TSR programok

Az előző részben azt néztük meg, hogyan lehet rendszermegszakításokat aktívizálni Turbo Pascal programból. Most azonban a másik oldalról közelítjük meg a kérdést, nevezetesen, hogyan tudunk megszakítási (**interrupt**) rutint írni Turbo Pascal nyelven.

A Turbo Pascal támogatja az ún. **interrupt** rutin írását az alábbi struktúrával:

```
{$F+}
procedure azonosito [( Flags, CS, IP, AX, BX, CX, DX,
                 SI, DI, DS, ES, BP : Word )]; interrupt;
{$F-}
  begin
    .
    .
  end;
```

Az **interrupt** eljárást mindig távoli (*far*) hívási konvencióval kell definiálni ({$F+}). A paramétersort, vagy annak egy részét akkor kell használni, ha a rutinunk paramétereket fogad regiszterekben, illetve értéket ad vissza, szintén regiszterekben.

Az **interrupt** rutinunk csak az *Intr* hívással illetve a rendszer által aktívizálható. Ehhez azonban a rutinunk címét el kell helyezni a megszakítások táblázatában. A DOS unit-ban találunk eljárásokat ezen lépés elvégzésére:
a

SetIntVec(IntNo,Vector);

rutin a pointer típusú *Vector* értéket a megszakítástábla *IntNo*-adik indexű elemébe másolja. Sok esetben az átírás előtt szükséges az ott tárolt címet elmenteni. A mentést a

GetIntVec(Intno,Vector);

hívással végezhetjük el, ahol a címet a *Vector* pointer típusú változóban kapjuk vissza.

Az **interrupt** rutinok használatának három esetét példaprogramokkal ismertetjük.

1. Példa paraméterezett interrupt rutin készítésére. A példában a *my_it* nevű eljárás az AX és BX regisztereket bemenő, míg a CX és DX regisztereket kimenő paraméterként használja.

```
program it_pld;
 uses crt,dos;
 const
    it = $fd;

 {$F+}
 procedure my_it(AX,BX,CX,DX,SI,DI,DS,ES,BP:word);
           interrupt;
 begin
   writeln('my_it: AX=',ax:5);
   writeln('my_it: BX=',bx:5);
   CX:=6496;
   DX:=2308;
 end;
 {$F-}

 var
    r   : registers;
    pit : pointer;
 begin
    getintvec(it,pit);
    setintvec(it,@my_it);

    with r do
      begin
        ax:=6512;
        bx:=294;
      end;
    intr(it,r);

    writeln('main : CX=',r.cx:5);
    writeln('main : DX=',r.dx:5);

    setintvec(it,pit);
 end.
```

2. A megszakítások egy része nem hívható az *Intr* eljárás segítségével. Ezeket mindig valamely hardver eszköz kérelmére aktívizálja a processzor (*hardver interrupt*). Nagyon fontos megjegyeznünk, hogy *hardver interrupt* rutinból semmilyen DOS művelet sem használható, mivel a DOS egyprogramos operációs rendszer. Ha alap I/O műveleteket kívánunk elvégezni, akkor a CRT unit rutinjai gond nélkül használhatók.

A következő példában a $1C, az ún. felhasználói timer megszakítást definiáljuk át saját céljainkra. Ha a program nem fut végig, akkor az első példában definiált módon (a ***setintvec*** rutin segítségével) a régi **interrupt** rutin címét nem tudjuk visszaírni, így a számítógépünk "lefagy". Hogy lehet ezt a kellemetlen jelenséget kiküszöbölni?

A System unit lehetőséget biztosít ún. saját kilépési eljárás definiálására, az ***exitproc*** nevű globális mutató felülírásával. Az ***exitproc*** pointerben tárolt rutin meg fog hívódni, ha a program valamilyen futás közbeni hiba miatt (*run-time error*) megáll, vagy egyszerűen akkor, ha a program véget ér. A rutinon belül az ***exitcode*** és ***erroraddr*** globális változókból lekérdezhető a hiba kódja illetve a hiba helyének címe a programban. Az ***erroraddr*** **nil** értékkel való törlésével letiltjuk a Turbo Pascal rendszer saját hibaüzenetének kiírását.
Ezen rövid gondolatmenet után tekintsük magát a programot.

```
program timer;
 uses
     crt, dos;
 const
     timerit  = $1c;
     maxtime  = 18;
 var
     exitsave : pointer;
     oldvect  : pointer;
     lefutas  : integer;
     key      : char;
```

```
{Az átdefiniált kilépési eljárás}
{$f+}
procedure exit_timer; {$f-}
begin
    if (exitcode<>0) or (erroraddr<>nil) then
         begin
            writeln('hiba történt: ',exitcode);
            meml[0:4*timerit]:=longint(oldvect);
            port[$20]:=$20;
         end
       else
         writeln('normál kilépés.');

    erroraddr:=nil;
    exitproc := exitsave;
end;

{A timer megszakítás rutin egy számlót dekrementál}
procedure newtimer; interrupt;
begin
    if( lefutas > 0 ) then
        lefutas := lefutas - 1;
end;

begin
    {Saját kilépési eljárás aktivizálása}
    exitsave:=exitproc;
    exitproc:=@exit_timer;

    getintvec( timerit, oldvect );
    lefutas := maxtime;
    setintvec( timerit, @newtimer);

    while (keypressed = false) do
        { Törölte a számlálót a megszakítás?}
        if ( lefutas = 0 ) then
            begin
               writeln(#247, '1 mp. eltelt');
               lefutas := maxtime;
            end;

    key := readkey;
    setintvec( timerit, oldvect);
end.
```

3. Harmadik példánk ún. *"Terminate and Stay Resident"* TSR program készítését mutatja be. Ha a főprogramból a

   Keep(*kilepesi_kod*);

   hívással lépünk ki, akkor a programunk a kilépés után is bent marad a memóriában. A program a különböző billentyűk lenyomásának hatására különböző hangot ad. A működés a CTRL-ESC billentyűkkel vezérelhető. Meg kell jegyeznünk, hogy a TSR programok memóriából való kitörlés nem egyszerű feladat, amelynek tárgyalása meghaladja a könyvünk célkitűzését.

```
{$M $800,0,0 }   { 2K stack, nincs heap }
 program tsr_pld;
 uses Crt, Dos;

 var
   oldit_09    : Procedure;
   sc          : byte;
   on_off      : boolean;
   al,ah       : byte;

 {Inline eljárás - a PUSHF gépikódú utasítás}
 procedure PUSHF; inline ($9C);

 { A új megszakítást kezelõ rutin }
 {$F+}
 procedure newit_09; interrupt;
 begin
   { A scan kód beolvasása }
   sc:=port[$60];
   if sc=1 then
     begin
      if (mem[0:$417] and $04)<>0 then
        {A CTRL-ESC kombináció hatására ki-/bekapcsol}
          begin
              on_off:= not on_off;

              {A billentyûzet vezérlõ programozása}
              al:=port[$61];
              ah:=al or $80;
              port[$61]:=ah;
              port[$61]:=al;
```

```
                {A megszakítás vezérlõ programozása}
                port[$20]:=$20;
                exit;
            end;
      end;

     {Ha be van kapcsolva és lenyomtak billenyût}
     if on_off and (sc<$80) then
      begin
       {Hanggenerálás}
       sound( Abs(880 + 60 * sc)) ;
       Delay(10);
       Nosound;
       end;

     { A régi megszakítás rutin hívása }
     PUSHF;
     oldit_09;
   end;
   {$F-}

   begin
     on_off:=false;

     {A régi belépési pont lekérdezése}
     GetIntVec($9,@oldit_09);

     {A IT-cím saját területre állítása}
     SetIntVec($9,Addr(newit_09));

     {TSR -  rezidens kilépés a programból }
     Keep(0);
   end.
```

16.4. A DOS unit file-kezelési lehetőségei

A 11. fejezetben ismertetett file-kezelést az alaprendszer (SYSTEM **unit**) tartalmazza. A DOS **unit** alprogramok sorával fedi le az operációs rendszer file-ok kezelésére szolgáló funkcióit. A megfelelő rutinok részletes leírását a függelék tartalmazza, itt csak egy rövid ismertetést adunk róluk:

- A DOS idő és dátum kezelése:

  GetDate, GetFTime, Gettime, PackTime, Setdate, SetFTime, SetTime, UnpackTime.

- A lemezegység teljes méretének és szabad helyének lekérdezése:

  DiskFree, DiskSize

- File-ok feldolgozása:

  FindFirst, FindNext, FSplit, FExpand, FSearch,

- File-attribútumok lekérdezése és állítása:

  GetFAttr, SetFAttr.

- A rutinok többsége a DOS **unit** ***doserror*** nevű globális változójában ad információt az adott művelet sikerességéről.

A DIRTREE.PAS program a fenti funkciókat felhasználva, parancssorban megadott könyvtár adott file-jait keresi. A megtalált file-ok esetén kijelzi azok attribútumát. A program érdekessége, hogy a könyvtár fastruktúrájának bejárását rekurzív eljárással (*keres*) oldottuk meg.

```
program dirtree;
 uses Crt,Dos;

 const ch:char=#32;

 {A file-attribútumokat kiíró rutin}
 procedure writeattr(fn:string);
   const
     attrstr:string[6]='RHSVDA';
   var
     fp  : file;
     i   : integer;
     as  : string[7];
     attr: word;
```

```
begin
     assign(fp,fn);
     getfattr(fp,attr);
     as:='[';
     for i:=1 to 6 do
if (attr and (1 shl (i-1)))<>0 then as:=as+attrstr[i]
                              else as:=as+'.';
     write(as,']');
end;

{A file ellenõrzése az adott könyvtárban}
procedure checkfile(fname:string);
  var dirinfo: Searchrec;
begin

    FindFirst(fname, Anyfile, DirInfo);
    while DosError = 0 do
    begin
      if dirinfo.name[1]<>'.' then
         begin
            Write('.....',DirInfo.Name);
            gotoxy(30,wherey);
            writeattr(dirinfo.name);
         end;
      writeln;
      FindNext(DirInfo);
    end;
end;

{A teljes könyvtár bejárását végzõ
 rekurzív eljárás }
procedure keres(dir,fname:string);
  var
    DirInfo: SearchRec;
    s1:string;
  begin
    if keypressed then ch:=readkey;
    if ch=#27 then exit;
    getdir(0,s1);
    {$I-}
    chdir(dir);
    {$I+}
    if ioresult<>0 then
        begin
          writeln('Hibás directory: ',dir);
          exit;
        end;
```

```
        writeln(dir);
        checkfile(fname);
        FindFirst('*.*', Anyfile,DirInfo);
        while DosError = 0 do
        begin
          if dirinfo.name[1]<>'.' then
           begin
             {$I-}
             chdir(dirinfo.name);
             {$I+}
             if ioresult=0 then
                begin
                if dir[ord(dir[0])]='\' then dec(dir[0]);
                  chdir('..');
                  keres(dir+'\'+DirInfo.Name,fname);
                end;

           end;
          FindNext(DirInfo);
        end;
      chdir(s1);
    end;

{ A fõprogram }
begin
assign(output,'');
rewrite(output);
if paramcount<2 then
     begin
       writeln;
       writeln('Használat: DIRTREE dir file');
       writeln;
       exit;
     end
     else
       keres(paramstr(1),paramstr(2));
end.
```

16.5 Programok indítása

A DOS **unit**-ban definiált *Exec* eljárás tetszőleges program paraméterezett indítását teszi lehetővé. Mivel minden Turbo Pascal program néhány megszakításvektort automatikusan átdefiniál, az *Exec* rutin hívása előtt ezeket a vektorokat vissza kell állítani, majd a hívás után újra át kell

definiálni. A megszakítási vektorok cseréjére a ***SwapVectors*** eljárás használható.

Nem szabad megfeledkeznünk arról, hogy a hívott program futásához is memória szükséges, ezért a {$M} kapcsoló megfelelő megadása elengedhetetlen az ***Exec*** használatához.

Az EXECPGM.PAS program a futtatandó program teljes nevének és a paramétersorának bekérése után megpróbálja azt elindítani. A probálkozás után végzett hibakezeléssel informál bennünket a futás eredményességéről. A DOS unit a ***doserror*** globális változóban közli a fellépő hiba okát.

```
program execpgm;
 {$M 10000,0,0}
 uses  dos;

 var
     command_line  : string;
     program_path  : string;
     key           : char;

 begin
     writeln('DOS belsõ parncsok végrehajtása:');
     writeln('   programnév   : ''c:\command.com''');
     writeln('   argumentumok: ''/c <dos parancs>''');

     repeat
         write( 'A végrehajtandó program: ' );
         readln( program_path );
         write( 'A program paraméterei    : ' );
         readln( command_line );
         writeln('A ',program_path,
                 ' program indítása...' );

         { A program indítása }
         swapvectors;
         exec(program_path, command_line );
         swapvectors;
```

```
            writeln('... kilépés.');
            if doserror <> 0 then {hiba volt?}
              writeln('dos hiba #', doserror)
            else
              writeln('sikeres futtatás. ',
                     'a child process kilépési kódja = ',
                      dosexitcode);
      writeln;
      write('Akar más parancsot végrehajtani (i/n) ? ' );
            readln( key );
      until upcase( key ) <> 'i';
    end.
```

17. JÁTÉKPROGRAMOK KÉSZÍTÉSE

Ebben a fejezetben játéprogramok készítésével foglalkozunk. Minden programnál először ismertetjük a játékszabályt, tehát tulajdonképpen a program feladatát, majd ismertetjük a program algoritmusát.

17.1. Számkirakó játékprogram

Játékszabályok

A játékot egy 4 x 4 mezőből álló táblán játszuk. A tábla mezői közül egy üres, a többin pedig 1 és 15 közé eső számok találhatók. A játék kezdetén a program kirajzolja a táblát, olyan módon, hogy a számok növekvő sorrendben követik egymást. Rövid várakozás után a program elkezdi összekeverni a számokat, és ezt egy billentyű leütéséig teszi. A játék ezután kezdődik, ugyanis a táblán ismét ki kell rakni az eredeti sorrendet, mégpedig úgy, hogy mindig az üres mezőt cseréljük ki a négy szomszédos mező valamelyikével. A csere végrehajtásához a kurzormozgató billentyűket használjuk. A játék végén kiíródik a lépések száma valamint a számok rendberakásával eltöltött idő. A játék az *Esc* billentyű leütésével bármikor megszakítható.

A játék haszálata

A játék kezdetén a program elkezdi összekeverni a számokat. Ez bármelyik billentyű leütésével megszakítható. A rendberakáshoz a kurzormozgató billentyűket használjuk, ezek leütésekor felcserélődik az üres mező és a kurzormozgató billentyű által kijelölt szomszédja. A játék az "Esc" billentyű leütésével a keverés után bármikor megszakítható.

A program felépítése

A programnak az alábbi főbb lépéseket kell végrehajtania:

a./ A tábla felrajzolása.
b./ A számok összekeverése.

c./ Az üres mező felcserélése a kijelölt szomszéddal.
d./ A tábla vizsgálata
e./ A c./ és a d./ pontok szükség szerinti ismétlése

A program az alábbi eljárásokat és függvényeket használja:

A tábla felrajzolásához szükséges eljárások

A *keret* eljárás

A szöveges üzemmódú képernyőn a *keret* eljárás az adott bal felső sarokpontból a tábla keretét rajzolja meg, amely adott számú és méretű mezőből áll. Az eljárás bemenő paramétere a bal felső sarok koordinátái, egy mező mérete és a mezők száma.

A *kitolt* eljárás

A *kitolt* eljárás a tábla mezőit szinezi és tölti ki az 1-től 15-ig terjedő számokkal. Az eljárásnak nincsenek paraméterei.

A számok összekeverése

A *csere* eljárás

A számok összekeverését a program úgy végzi, hogy ciklikusan ismételve véletlenszerűen kiválasztott szomszédjával cseréli fel az üres mezőt. Ehhez a részhez tartozik a *csere* eljárás, amely a megadott iránytól függően kicseréli az üres mezőt a megfelelő szomszédjával Az eljárás bemenő paramétere egy *byte* típusú változó, amelyik megmutatja, hogy melyik szomszédos mezővel legyen felcserélve az üres mező.

Az üres mező kicserélése a szomszéddal

Ez a rész is használja a *csere* eljárást, valamint az *irány* függvényt. A függvény visszatérési értéke a leütött billentyűtől függően egy 1 és 4 közé eső szám. A függvény tartalmaz egy ***halt*** utasítást is, amelyik az *Esc*

billentyű leütése esetén megállítja a programot. Csak a kurzormozgató billentyűk és az *Esc* billentyű leütését figyeli a függvény.

A tábla vizsgálata

Ehhez a részhez csak a *vege* függvény tartozik. Ennek visszatérési értéke *true*, ha a táblán a számok sorrendben vannak, különben *false*.

A játék befejezése

A játék befejezésekor a kiértékelést az *eredmeny* függvény végzi el. A függvény bemenő paramétere a rendberakáshoz szükséges lépések száma valamint az idő. Ezeket az értékeket egy táblára írja ki és felajánlja egy új játék lehetőségét. A függvény visszatérési értéke új játék esetén **true**, a játék befejezése esetén **false**.

A főprogram felépítése

A program először az alkalmazott képernyőtől függően beállítja a szineket (Hercules képernyőn ugyanis más szinkiosztás kell mint a többin). Ezután felrajzolja, kitölti majd a ***Random*** beépített függvény segítségével összekeveri a táblát.
A következő lépésben beállítja a kezdőértékeket (lépésszám, kezdőidő).

Ezután ciklikusan ismétli a következőket:
A játékos által leütött billentyű szerinti két mezőt felcseréli, és megvizsgálja a tábla állapotát. Ha a táblán sorrendben vannak a számok, akkor befejezi a ciklust.

A játék végén kiírja az eredményt és felajánlja az új játékot.

A program főbb változói

A program viszonylag kevés változót használ, lokális változói segédváltozók szerepét töltik be, a függvények és az eljárások paraméterei azok leírásánál szerepeltek.

Főbb globális változói a következők:

tabla	**record** típusú tömb, amely tartalmazza a tábla *i*-dik sorának *j*-dik oszlopában levő mező szinét és számát.
ures	**byte** típusú kételemű tömb, amelyik az üres mező helyét mutatja meg.
t0, *jido*	a játék kezdésének időpontja és a játékkal a *t0* időpont óta eltöltött idő.

17.2. Memória játék

Játékszabályok

A játékot két személy játszhatja, 56 kis kártyával, amely kártyák midegyikén egy szó szerepel, az 56 kártyán összesen 28 szó van, így minden egyes szó pontosan két kártyán található meg. A játék kezdetekor ezt az 56 kártyát kirakjuk, mégpedig úgy, hogy a rajta levő szavak nem látszanak. Ezután az első játékos megnéz két kártyát. Ha a rajtuk levő szavak megegyeznek, akkor az illető nyert egy pontot és ismét ő nézhet meg egy párt, ellenkező esetben viszont az ellenfél következik. A játék akkor ér véget, ha az egyik játékos eléri a 20 pontot.

A játékprogram kialakítása olyan, hogy egy pár eltalálása után azokat a kártyákat, amelyeket még senki nem nézett meg, a számítógép ismét öszekeveri, így ha valaki eltalált egy párt, utána a pár helyén nem párt talál.

A játék használata

A kártyák megnézéséhez bal felső sarokban megjelenő villogó csillagot kell a kiválasztott kártya fölé vinni, és ott megnyomni az "Enter"-t. A villogó csillag ekkor folyamatosan világítani fog, és megjelenik a jobb alsó sarokban a másik csillag. Ezzel a csillaggal kijelölhetjük a másik kártyát, és a két kártyát egyszerre fogja a számítógép megmutatni. A villogó csillagot a kurzormozgató billentyűkkel lehet mozgatni. Ha a második csillagot az első

fölé visszük és ott ütjük le az "Enter"-t akkor az első kártya kijelölését töröljük.

A program felépítése

A programnak az alábbi főbb lépéseket kell végrehajtania:

a./ A kártyák felrajzolása
b./ A kártyán levő szavak kitöltése
c./ A kiválasztott kártyák megmutatása
d./ Pár esetén a még nem látott kártyák összekeverése
e./ A c./ és d./ lépések ismétlése, míg az egyik játékos a 100 pontot el nem éri.

A program főbb eljárásai és függvényei

a./ A kártyák felrajzolását a *számkirakó* programnál ismertetett *keret* eljárás végzi.

b./ A kártyákon levő szavak kitöltésére szolgál a *kitolt* eljárás paraméter nélküli eljárás. Az itt alkalmazott algoritmus leírása eléggé nehézkes, de mint a program listájából látható, meglehetősen tömör, jól programozható módszert eredményez.

A módszer lényege a következő:

Egy tömbváltozóban megjegyezzük valamennyi kártya sorát és oszlopát. Ezt ezentúl a kártyák indexének fogjuk nevezni. A tömbváltozó mellett felveszünk még egy segédváltozót is, amelyik azt mutatja meg, hogy még hány kártyát nem töltöttünk ki. Ezt a változót mutatónak nevezzük. A tömb kiinduló értékei ennek alapján az (1,1), (1,2), (1,3), ... ,(7,7), (7,8) A zárójelben szereplő első szám a kártya oszlopát, a második a sorát jelöli. A mutató kezdeti értéke 7*8 azaz 56. Ezután véletlenszerűen kiválasztjuk a tömb valamelyik elemét, és az itt szereplő értékek szerinti kártyára ráírjuk az első szót. A következő lépésben, azért, hogy ezt a kártyát a továbbiakban már ne bántsuk, a tömbnek ezt az elemét kicseréljük az utolsóra, a mutató értékét pedig egyel csökkentjük. Ezek szerint pl. ha a harmadik elemet

választottuk ki, akkor a csere után a tömb és a mutató értéke a következők szerint alakul:

tömb: (1,1), (1,2), (7,8), (1,4), ... ,(7,7), (1,3)
mutató: 55

A következő lépésben már csak a mutató által kijelölt számú indexből választunk, és a mutató által kijelölt indexű elemmel cserélünk. Ezt a módszert 56 esetben megismételve mindegyik kártyánkat ki tudjuk tölteni.

c./ ehhez a részhez két lényeges eljárás tartozik, az egyik a *kijelol*, a másik a *visz*.
Az első eljárás kimenő paramétere a két kijelölt koordinátapár. Az eljárás hatására egy villogó "*" karakter jelenik meg a képen, amelynek a helyzetét a kurzormozgató billentyűkkel lehet változtatni. A kártya kijelölésére az *Enter* billentyű szolgál ezután a kijelölt kártyát ez folyamatosan világító "*" mutatja. A mozgatásra és kijelölésre a *számkirakó* programban is alkalmazott *irany* függvény, az *Enter* billentyű figyelésével kiegészített változata szolgál. A kijelölő eljárás felépítése olyan, hogy lehetséges az először kijelölt kártya kijelölt állapotát törölni, azzal hogy a villogó "*"-ot ismét erre a kártyára visszük és újból megnyomjuk az *Enter*-t

A második a *visz* eljárás. Ennek bemenő paraméterei a kijelölt kártyák koordinátái, szerepe pedig, hogy egy-egy megfelelő méretű ablakban megjelenítse a kijelölt kártyán levő szavakat.

d./ ehhez a ponthoz egy függvény és egy eljárás tartozik. A *vizsgal* függvény, amelynek bemenő paraméterei a kijelölt kártyák koordinátái, visszatérési értéke akkor igaz (*true*), ha a két kártya azonos volt, különben *false*. A függvény meghívja az ismertetett *kever* eljárást is. Ez az eljárás *kitolt* eljárásnál leírtakhoz hasonlóan összekeveri a kártyákon levő szövegeket. A lényeges különbség nem a keverés módjában van, hanem abban, hogy a *kitolt* eljárásnál a sorban egymás után beírt szavakat kellett véletlenszerűen összekevert koordinátájú helyekre beírni, itt pedig az adott helyen levő kártyákon levő szöveget kell összekeverni.

e./ ehhez a részhez nem tartozik külön eljárás illetve függvény.

A főprogram felépítése

A program a **számkirakó** játékhoz hasonlóan először beállítja a színkiosztást. Ezután kitölti a kártyákat, majd felrajzolja azokat, majd ciklikusan megvizsgálja a kijelölt kártyákat, számolja az eredményt és újrarajzolja a kártyákat. (A teljes képernyő újrarajzolása lényegesen egyszerűbb, mint a kártyák kiemeléshez hasonló levétele.)

A program főbb változói

A program három lényeges változót használ, ezek a következők:

szabad: a kártyák koordinátáit tartalmazó kétdimenziós **byte** típusú tömb

szabnr: a még nem megnézett kártyák számát mutató **integer** típusú szám

kartya: **record** típusú tömb. A **record** elemei a *szov* nevű **string**, amelyik tartalmazza a kártyán szereplő szöveget, valamint a *mut* nevű **integer** szám, amelyik megmutatja, hogy az adott kártyához a *szabad* nevű tömb melyik eleme tartozik.

17.3. Türelem tüske játékprogram

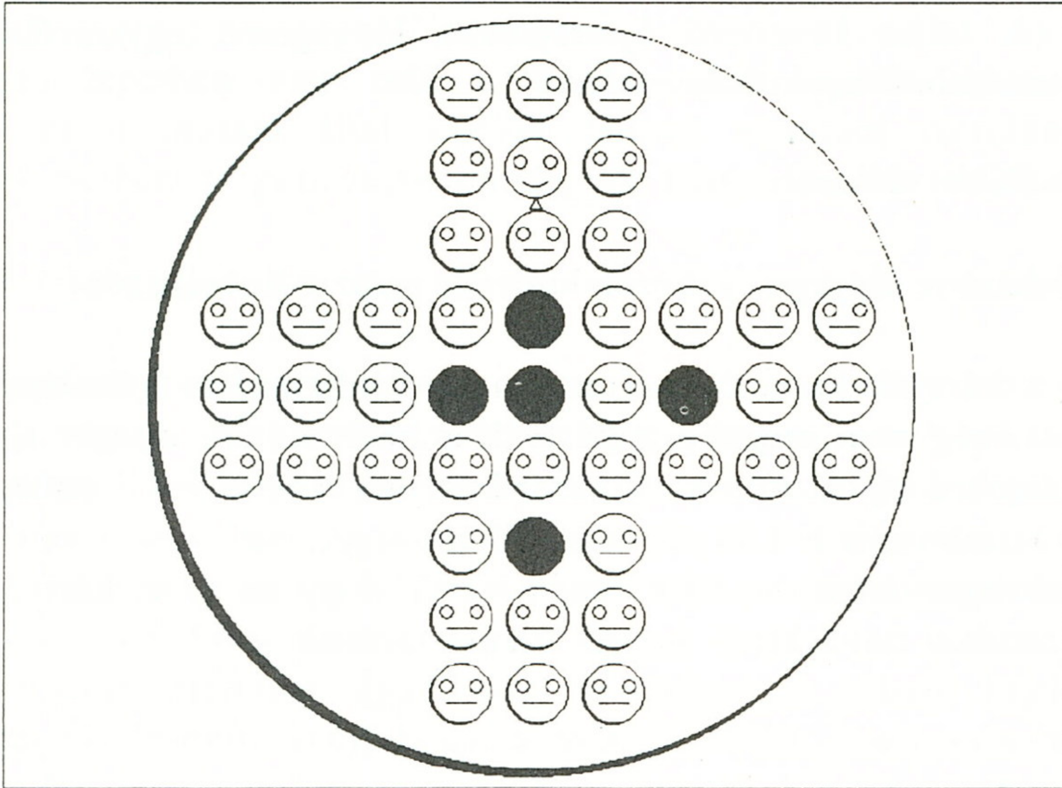
Játékszabályok

A játékot egy személy játszhatja egy összesen 45 mezőből álló táblán. A mezők 3 x 3-as csoportokban, kereszt alakban helyezkednek el. A táblán a játék kezdetekor 44 bábu (tüske) van, úgy, hogy a középső mező üresen marad. A játék során valamelyik bábuval átugorhatjuk a bábu négy szomszédjának valamelyikét, amennyiben az átugorni kívánt bábu másik oldalán üres mező van. Az így átugrott bábut a tábláról le kell venni. A játék célja az, hogy a táblán minél kevesebb bábu maradjon.

A játék használata

A játék kezdetekor a középső hely alatt egy kis nyíl látható. Ezt a nyilat a kurzormozgató billentyűkkel a kiválasztott bábu alá kell vinni majd az "Enter" leütésével a bábut kijelölni. Ezután ugyanígy ki kell jelölni azt a

mezőt amelyikre a bábuval ugrani akarunk. Ha az ugrásra kijelölt bábura ismét rávisszük a nyilat, és ismét megnyomjuk az "Enter"-t akkor a bábu kijelölése törlődik. A játék az "Esc" billentyű leütésével bármikor megszakítható.



A program felépítése

A programnak az alábbi főbb lépéseket kell végrehajtania:

a./ A tábla és a bábuk felrajzolása
b./ A bábuk ugrásainak kezdő- és végpontjainak a kijelölése
c./ A bábuk mozgatása illetve levétele
d./ A játék befejezése

A fenti pontok főbb eljárásai illetve függvényei:

a./ Ennek a résznek három fontos eljárása van, a *Tablainit* a *Kirajzol* és a *Pofa*.

A *Tablainit* paraméter nélküli eljárás adja meg a kezdőértéket a bábuk táblán való elhelyezkedését mutató *tabla* nevű, tömb típusú változónak. Ez a

változó az egyszerűbb kezelhetőség érdekében egy 9 x 9 elemű tömb (**array** [1..9,1..9] of **byte**), amelynek a négy sarkában levő 4 db 3x3-as részt nem használjuk. A játék során ennek a tömbnek a mezői a következő értékeket vehetik fel:

0 - üres mező
1 - bábu alaphelyzetben
2 - ugrásra kijelölt bábu
3 - átugrott bábu(ezt az értéket a mező csak ideiglenesen veszi fel, kb 0.5 másodperc után ez az érték automatikusan 0 lesz
255 - a játékban nem szereplő mezők

A *Tablainit* a táblát a kezdeti alakzatnak megfelelő módon 0, 1 illetve 255 értékekkel tölti fel. A *Pofa* nevű háromparaméteres eljárás a *Graph* unit eljárásainak segítségével - a program futásának kezdetekor beállított, képernyőtípustól függő méretben - felrajzolja a bábukat. Az eljárás első paramétere szabja meg a bábu képét, (ez a paraméter a program futása során a *tabla* nevű tömb megfelelő értéke lesz) az eljárás második és harmadik paramétere pedig a bábu pozícióját. A *Kirajzol* eljárás ugyancsak a *Graph* unit eljárásaival valamint a *Pofa* eljárás meghívásával kirajzolja a táblát.

b./ A bábuk ugrásainak kezdő illetve végpontját kijelölő *Kijelol* eljárás majdnem teljesen megegyezik a *Memory* játékban ismertetett, ugyanilyen nevű eljárással, a különbség csupán annyi, hogy az itteni eljárás grafikus képernyőt használ, és így az *Irany* nevű függvényen kívül a *Nyilki* és *Nyille* eljárásokat is használja, a kijelölt mező megjelölésére, illetve ennek a jelölésnek a törlésére. Ez az eljárás figyeli még azt is, hogy csak olyan mezőt lehessen kijelölni elsőként ahol van bábu, másodiknak pedig ahol nincs.

c./ A bábuk mozgatásának alapvető eljárása *Mozgat*. Ennek bemenő paramétere tömb típusú, és a kezdő illetve a végpont koordinátáit tartalmazza. Az eljárás átírja a *tabla* nevű tömbváltozó megfelelő elemeit, majd először a *Töröl* nevű háromparaméteres eljárás segítségével letörli a képernyőn a változtatni kívánt mezőket, majd a *tabla* nevű tömbváltozó megfelelő elemei alapján visszarajzolja azokat a *Pofa* nevű eljárás segítségével.

d./ A játék befejezését a *Vege* nevű **boolean** típusú függvény jelzi. A függvénynek a visszatérési értékén kívül van egy kimenő paramétere, amelyik azt mutatja, hogy még hány bábu van a táblán. A *Vege* függvény visszatérési értéke *true*, ha már nincs a táblán átugorható bábu. A függvény ezt két - majdnem teljesen egyforma - ciklusban ellenőrzi le. Az ellenőrzés annak alapján történik, hogy ha a *tabla* nevű változó valamelyik eleme 1 és ugyanakkor a két szomszédos elem összege is 1, akkor az a bábu átugorható. A két említett ciklusból az egyik a vízszintes, a másik pedig a függőleges szomszédokat ellenőrzi. A függvény ezenkívül egy kettős ciklus segítségével megszámolja a táblán levő bábukat a kimenő paraméter számára.

Ha a *Vege* függvény *true* értékkel tér vissza, akkor a *Befejezes* nevű egyparaméteres függvény kiírja a végeredményt, és felajánlja az új játékot. A függvény bemenő paramétere a bábuk száma, visszatérési értéke pedig *true*, ha ismét akarunk játszani.

A főprogram felépítése

A főprogram felépítése igen egyszerű. A program egy inicializációs résszel kezdődik. Ennek első tagja a *GrInit* amelyben a monitor típusától függően beállítjuk a grafikus képernyőt, valamint a *GrOszt* nevű változót, amelyik a rajzolás során a különböző rajzelemek méretének kiszámításához kell.

Ezután a *TablaInit* nevű eljárással beállítjuk a *tabla* nevű változó kezdőértékeit. Ez az inicializációs rész vége, majd mindaddig, míg a *vege* függvény *true* értékkel nem tér vissza, ciklikusan ismételjük a *Kijelöl* és a *Mozgat* nevű eljárásokat.
A *Vege* függvény *true* értéke esetén meghívjuk a *Befejezes* nevű függvényt, ennek *true* értéke esetén visszatérünk a *Tablainit* nevű eljáráshoz, *false* értéke esetén pedig kilépünk a programból.

A program főbb változói

tabla :**array** [1..9,1..9] of **byte**, a táblán levő bábuk állapotát tartalmazó tömb

par :**array** 1..3,1..2] of **byte**, a kijelölt mezők koordinátáit tartalmazó tömb

gd,gm	:**integer**, a monitorra jellemző értékek.
x0,y0,GrOszt	:**integer**, a képernyőre való rajzoláshoz szükséges segédértékek.

17.4. Rex játékprogram

Játékszabályok

A játék szabályai némiképp eltérnek a valódi szabályoktól. Itt a játékot két személy játszhatja, hagyományos rexasztalon, öt golyóval. A rexasztal olyan asztal, amelyen meghatározott helyeken összesen nyolc lyuk van, amelyekhez egy-egy érték tartozik. A játékban szereplő öt golyó közül egy piros, a többi pedig fehér. A játék lényege, hogy egy adott helyre kitett golyót úgy lökjünk meg, hogy az az asztalon levő golyók valamelyikének nekiütközzön, és ezután valamelyik golyó valamelyik lyukba beleessen. Minden lyukba esett golyó annyi pontot ér, amennyi a lyukhoz tartozó érték, illetve ha a piros golyó esik a lyukba akkor a lyukhoz tartozó érték kétszeresét.

A játék során mindaddig ugyanaz a játékos következik, ameddig az általa meglökött golyó hatására valamelyik golyó ütközés után lyukba esik. A sorozat alatt szerzett pontok összeadódnak. Ha a játékos által meglökött golyó nekiütközik ugyan egy másiknak, de egyik golyó sem esik lyukba, akkor a játékos sorozata érvényesen ért véget, a sorozat alatt szerzett pontokat az addigi pontjaihoz adhatja. Ha azonban a meglökött golyó nem ütközik neki egy golyónak sem, akkor a játékos sorozata érvénytelen, a sorozat alatt szerzett pontok elvesznek. Akár érvényesen, akár érvénytelenül ér véget a sorozat, a másik játékos következik.

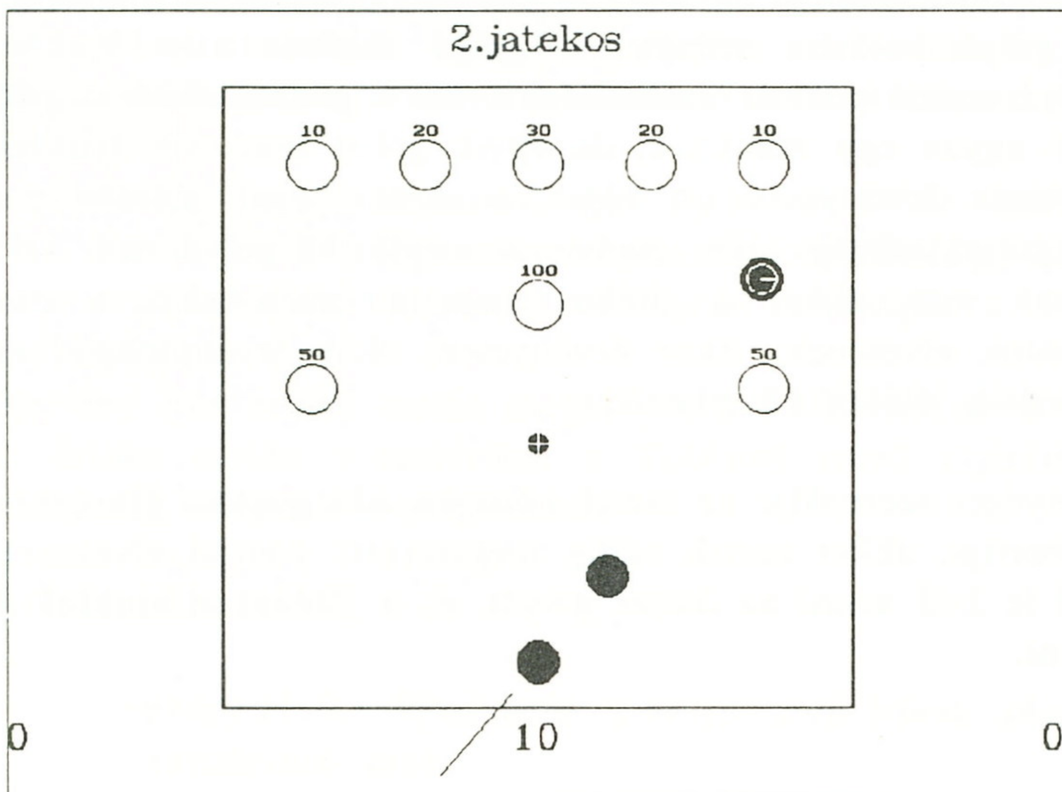
A játék lényeges szereplője az asztal közepén álló gomba. Ha ezt valamelyik játékos felborítja, akkor annak addig megszerzett pontjai elvesznek, továbbá az asztalról le kell venni az összes golyót, és a játékot a kiinduló helyzetből kell folytatni.

A játék menete

A játék kezdetén a számítógép két golyót tesz ki az asztal adott két pontjára, ebből a lejjebb lévőt löki meg a kezdő játékos. A játék során a számítógép mindaddig egy-egy újabb golyót rak ki az asztal adott pontjára, ameddig vagy az asztalon nem lesz az összes golyó, vagy el nem fogy az asztalról valamennyi. Az első esetben a számítógép kiválasztja az asztal alsó széléhez legközelebb lévő golyót, és ezt rakja a kiinduló pontba, a második esetben pedig a játék kezdetével megegyező módon jár el.
A játékot az a játékos nyeri, aki hamarabb éri el az 1000 pontot.

A játék használata:

A golyót automatikusan rakja a számítógép az asztalra. A lökés irányát (amelyet a dákó iránya mutat) a vízszintes kurzormozgató billentyűkkel, a golyó kezdősebességét pedig (amelyet a dákó golyótól való távolsága jelez) a függőleges kurzormozgató billentyűkkel lehet beállítani. A beállítás után a golyót az "Enter" leütésével lökjük el. A játék az "Esc" billentyű leütésével bármikor megszakítható.



17.2. ábra Rex asztal

A program felépítése

A programnak az alábbi főbb lépéseket kell végrehajtania:

a./ Az asztal felrajzolása
b./ A golyók kirakása
c./ A golyók meglökése és további mozgatása
d./ Az ütközések és a golyók lyukba való beesésének figyelése, valamint az ütközések hatásának kiszámítása.
e./ A lökések és a sorozatok érvényességének figyelése és a pontok számolása

A program főbb eljárásai és függvényei

a./ Az asztal grafikus képernyőn való felrajzolását végzi el az *Asztalrajz* nevű, paraméter nélküli eljárás. Az eljárás a képernyőn használható képpontok számától függően elhelyezi a képernyőn az asztalt, valamint a *Lukrajz* nevű eljáráson belüli eljárással kirajzolja a lyukakat. Ez utóbbi eljárás rendeli hozzá az egyes lyukakhoz a hozzájuk tartozó értékeket is. A *Lukrajz* nevű eljárás bemenő paraméterei rendre a lyuk sorszáma, vízszintes és függőleges koordinátája és a lyukhoz tartozó érték. (A program során a golyókkal és lyukakkal kapcsolatos koordináták végig relatív koordináták lesznek, amelyek a golyók és a lyukak középpontjának valamint az asztal bal felső sarkának képpontokban mért távolságát mutatják meg.)

b./ A golyók kirakása során érdemes előre gondolni azok mozgatására is. Ez utóbbi miatt a golyók kirakásánál a következőket fogjuk végrehajtani:

- helyet foglalunk a *heap*-ben azon pointeres változók számára, amelyekbe a golyók képének megfelelő képernyőterületet, illetve a golyók által eltakart képernyőterületet fogjuk kimenteni.
- felrajzolunk egy fehér golyót a képernyő bal felső sarkába.
- azt a képernyőterületet, amelyiken a golyó van, kimentjük a négy fehér golyó számára fenntartott pointeres változóba.
- ugyanerre a helyre felrajzoljuk a piros golyót, és a képernyőterületet a piros golyó számára fenntartott pointeres változóba mentjük.
- letöröljük a képernyő sarkát.

Az eddigiek végrehajtására szolgál a *Golyorajz* nevű paraméter nélküli eljárás. A következő lépés a golyó megjelenítése az asztalon. Ezt a *Kitesz* nevű háromparaméteres eljárás hajtja végre. Az eljárás paraméterei rendre a golyó sorszáma, vízszintes és függőleges koordinátája. Az eljárás először beírja a golyóhoz rendelt változóba a két koordinátát,valamint *true*-ra állítja azt a golyóhoz rendelt **boolean** típusú változót, amelyik megmutatja, hogy a golyó látszik-e a képernyőn. Ezután a golyó által eltakart terület számára fenntartott pointeres változóba mentjük az asztalnak azt a részét, ahová a golyót rajzolni fogjuk, megjegyezve ezt a két koordinátát. Ezután az asztal képére rámásolja az *i*-dik golyó képét tartalmazó képernyőterületet. A játék során (azaz nem a játék kezdetekor) kirakott golyók esetén még néhány dolgot figyelnünk kell:

- van-e az asztalon golyó, mert ha nincs akkor kettőt kell kitenni
- van-e kitehető golyó, mert ha nincs akkor a legalsót le kell venni az asztalról
- nincs-e ott golyó, ahova a számítógép tenni akar, mert akkor az új golyót arrébb kell rakni.

Ezeket a feltételeket az *Ujgolyo* nevű egyparaméteres eljárás vizsgálja meg, melynek kimenő paramétere annak a golyónak a sorszáma, amelyet meg kell majd lökni. Az eljárás először megvizsgálja, hogy van-e az asztalon golyó (ekkor a *vangolyo* nevű ***boolean*** tipusú változó értéke *true*), illetve, hogyha van kitehető golyó, akkor ennek mennyi a sorszáma (ha nincs kitehető golyó, akkor a sorszám értéke 0). ezután, ha nincs kitehető golyó, akkor leveszi a legalul levő golyót, ha pedig nincs az asztalon golyó akkor kitesz egyet a megfelelő helyre. Ezután, figyelve azt, hogy van-e elegendő hely, kitesz egy golyót.

c./ A golyók meglökéséhez az egyparaméteres *Lok*, valamint az általa meghívott háromparaméteres *Dako* nevű eljárás tartozik. A *Dako* nevű eljárás bemenő paramétere a golyó sorszáma, kimenő paramétere pedig egy sebesség és egy irány. Az eljárás először megrajzolja a dákót. Ezt a következőképp hajtja végre:

- a dákó golyótól való távolságától valamint állásától függően kijelöli azt a grafikus ablakot, amelybe a dákót rajzoljuk.
- meghatározza az ablak elmentéséhez szükséges pointeres változó méretét, és lefoglalja azt a *heap*-ben

- kimenti az említett grafikus ablakot
- kirajzolja a dákót
- leolvassa a klaviatúráról a leütött billentyűt
- visszamenti a grafikus ablakot és felszabadítja a *heap*-ben lefoglalt területet.
- a leütött billentyűtől függően megváltoztatja a dákó helyzetét és előről kezdi az eljárást vagy kilép az eljárásból.

A *Lok* nevű eljárás - amelynek paramétere a golyó sorszáma - meghívja a *Dako* nevű eljárást, majd az az által szolgáltatott értékeket beírja a golyóhoz tartozó record típusú változó megfelelő mezőibe (sebesség, irány).

A golyók mozgatásához három eljárást használunk. A *helyzet* nevű paraméter nélküli eljárás számítja ki a golyók helyzetét *dt* idő múlva. A másik két eljárás szolgál a golyók képernyőn történő elmozdítására. Ez ugyanis úgy történik, hogy az asztalról levesszük az *i*-dik golyót, azaz az *i*-dik golyó által eltakart képernyőterületet visszamentjük a megjegyzett helyre. Erre a célra szolgál a *Levesz* nevű eljárás, amelynek paramétere a golyó sorszáma. Ugyanez az eljárás *false*-ra állítja azt a - golyóhoz tartozó, **boolean** típusú - változót amelyik megmutatja, hogy a golyó látszik-e a képernyőn. A golyók elmozdításához használt másik eljárás a már ismertetett *Kitesz*.

d./ Két golyó összeütközését az *utkozik1* nevű függvény figyeli. Ennek két bemenő paramétere a két golyó sorszáma, visszatérési értéke pedig *true* ha a két golyó középpontjának távolsága kisebb mint a golyók sugarának kétszerese. Ha két golyó összeütközik, akkor ki kell számítani az ütközés utáni sebességet. Ezt a *sebesseg* nevű kétparaméteres eljárás végzi el, melynek bemenő paraméterei a golyók sorszámai. A sebességek kiszámítása a következő összefüggések alapján történik:

$$\underline{v}_j=\underline{v}_j+\underline{dv}$$
$$\underline{v}_i=\underline{v}_i-\underline{dv}$$

ahol

$$\underline{dv}=\underline{v}_i(\sin(\alpha_i)\sin(ß)+\cos(\alpha_i)\cos(ß))-\underline{v}_j(\sin(\alpha_j)\sin(ß)+\cos(\alpha_j)\cos(ß))$$

α : a golyók sebességének iránya és a függőleges közötti szög

ß : a két golyó középpontján átmenő egyenes és a függőleges közötti szög.

Azt hogy egy golyó egy lyukba beleesik-e a *beesik* nevű függvény figyeli. Ennek is két bemenő paramétere van, egy golyó sorszám és egy lyuk

sorszám. A függvény megvizsgálja, hogy a golyó középpontja a lyuk fölé esik-e. Ha igen akkor kiszámítja a golyó gyorsulását, és megvizsgálja, hogy a golyó beleesik-e a lyukba (elképzelhető, hogy a golyó a lyuk szélén csak irányt változtat, de nem esik bele).

e./ A lökések érvényességét és a pontok számolását közvetlenül a főprogram végzi. Külön eljárás a pontszámok kiírására van, ennek neve *Felirat*, bemenő paramétere két **integer** típusú szám, amelyik az aktuális játékos sorszámát, illetve a sorozatban elért pontszámot adja át, valamint a gomba fellökését figyelő paraméter nélküli *gomba* nevű **boolean** típusú függvény.

A főprogram felépítése

A program kezdetén található egy inicializációs rész, amelyik a változók kezdőértékeit állítja be, valamint inicializálja a grafikus képernyőt. A grafikus képernyő inicializálása előtt kéri be a program a *BGI* file-ok elérési útvonalát. Szintén az inicializációs részhez tartozónak lehet tekinteni az *Asztalrajz* a *Golyorajz* illetve az első *Felirat* nevű eljárásokat valamint az első két golyó kitételét. A tényleges program a *Lok* eljárással kezdődik. Innen szükséges az ütközések és a lyukba való beesések figyelése. A lökések érvényességét két **boolean** típusú változó segítségével figyeljük. A lökés után mindkét változó alapértelmezésben *false* lesz. Az *erv1* változó értéke akkor lesz *true*, ha két golyó összeütközik, az erv2 pedig akkor, ha egy golyó beleesik valamelyik lyukba. Ezután hajtódik végre a *gomba* függvény, amelynek *true* visszatérése esetén valamennyi golyó értéke üres (lásd a változók leírásánál), az *erv1* és *erv2* értéke *false* a *szumma* értéke pedig -1 lesz, és nem hajtódik végre a következő lépés a golyók új helyzetének kiszámítása (*helyzet* eljárás) és a golyók képernyőn történő áthelyezése. Az előbbi lépések ismétlődnek mindaddig míg valamelyik golyó sebessége nagyobb nullánál. Ha minden golyó megállt, akkor a képernyő felfrissítése úgy történik, hogy újra meghívjuk az *Asztalrajz* eljárást és kirajzoljuk az összes golyót. Erre azért van szükség, mert a golyók összeütközésekor két golyó között átfedés lehet, ami a képernyőn nem odaillő pöttyöket eredményezhet. Ezután következik az *erv1* és *erv2* értéke alapján a játékos esetleges megváltoztatása, érvényes sorozat esetén a pontszámok felírása majd a következő lökés. A játék végén pedig következik az eredmény kiírása, és a grafikus képernyő lezárása.

A program főbb változói

golyo a golyók helyzetét, sebességét és gyorsulását és az utóbbi kettő irányát leíró **record** típusú tömb.
l*yuk* a lyukak helyzetét és értékét leíró **record** típusú tömb
ures az asztalon nem lévő golyók értékét mutató record (a golyók vízszintes és függőleges koordinátája -10, sebessége, gyorsulása, valamint az ezekhez tartozó szög értéke 0
pasztal az egyes golyók által eltakart területek elmentésére szolgáló pointer típusú tömb.
pgolyo Az egyes golyók képének elmentésére szolgáló pointer típusú tömb.
x0,y0,
xmax,ymax az asztal koordinátáit képpontokban megadó értékek.
r,rl a golyó illetve a lyuk sugarát képpontokban megadó értékek.
asztx,aszty a golyók által takart területek visszamentési koordinátái.

17.5. Játékok keretprogramja

A program célja

A program lehetőséget biztosít arra, hogy a leírt játékprogramok illetve a játékprogramok szabályai egy menüből lehívhatók legyenek. A menüben a játékok kiválasztása a ↓ illetve az ↑ billentyűkkel vagypedig a játék nevének kezdőbetűjével történik, azt pedig, hogy a keretprogram a játékot indítsa el, vagypedig a játékszabályokat írja ki, a ← illetve a → billentyűkkel lehet eldönteni. A választás után az *Enter* billentyű hatására végzi el a program a kiválasztott feladatot.

A program főbb lépései

a./ a menüablak kirajzolása
b./ a menüpont kiválasztása
c./ a választás alapján a játék lefuttatása vagy a szabályok kiírása.

Az egyes pontokhoz tartozó főbb eljárások és függvények

a./ A menüablak kirajzolását - figyelembevéve, hogy a teljes program során csak ez látható a monitoron - a főprogramban végezzük el.
b./ ennek a résznek az alapja a jelöl nevű egyparaméteres **boolean** típusú függvény. Az eljárás paramétere kimenő paraméter, értéke azt mutatja meg, hogy melyik menüpontot, azaz melyik játékot választottuk ki. A kiválasztáshoz a függvény figyeli a kurzormozgató billentyűket, továbbá az *Esc* billentyűt, amelynek hatására a program futása megszakad, valamint az *Enter* billentyűt, amelynek leütése esetén a függvény visszatérési értéke *true* lesz.
c./ ehhez a részhez, az a./ ponthoz hasonló okok miatt, ismét nem tartozik eljárás, a játékok lefuttatása illetve a szabályol kiírása a főprogramból történik.

A főprogram leírása

Az előzőek alapján a főprogram leírása is meglehetősen rövid, a főprogram beállítja a színeket, megrajzolja a menüablakot, meghívja a választáshoz szükséges függvényt majd ennek alapján végrehajtja a kijelölt műveletet.

A program főbb változói

szov	: az egyes menüpontok szövegét tartalmazó **string** típusú tömb
jatek	: **boolean** változó annak figyelésére, hogy a játékot akarjuk-e lejátszani vagy a szabályokat kiiratni
st	: **string** típusú változó, a meghivandó file (*.exe* vagy *.hlp*) nevét tartalmazza
hlpfile	: **text** típus, a szabályok szövegét tartalmazó file megnyitásához.

F1. A TURBO PASCAL SZABVÁNYOS ELJÁRÁSAI ÉS FÜGGVÉNYEI

F1.1. Matematikai függvények

Abs függvény
Visszatér az argumentum abszolút értékével.

Abs(x) : azonos x típusával;
x egész vagy valós típusú kifejezés.

ArcTan függvény
Visszatér az *arctg* főértékével.

ArcTan(x: **real**) : **real**;
x valós típusú kifejezés.

Az x valós típusú kifejezés arkusztangens főértékét adja vissza radiánban.

Cos függvény
Visszatér az argumentum koszinusz értékével.

Cos(x: **real**) : **real**;
x valós típusú kifejezés.

Az x értékét radiánban kell megadni.

Exp függvény
Visszatér az argumentum exponenciális értékével.

Exp(x: **real**) : **real**;
x valós típusú kifejezés.

Frac függvény

Visszatér az argumentum tizedes részével.

Frac(*x*: **real**) : **real**;

x valós típusú kifejezés.

Az *x* valós típusú kifejezés tizedes részét adja vissza:

```
Frac(x):=x-Int(x).
```

Int függvény

Visszatér az argumentum egész részével.

Int(*x*: **real**) : **real**;

x valós típusú kifejezés

Az *x* valós típusú kifejezés egész részét adja vissza:

```
Int(x):=x-Frac(x).
```

Ln függvény

Visszatér az argumentum természetes alapú logaritmusával.

Ln(*x*: **real**) : **real**;

x valós típusú kifejezés.

Odd függvény

Vizsgálja az argumentum páratlanságát.

Odd(*x*: **longint**) : **boolean**;

x egész típusú kifejezés.

Igaz értéket ad vissza ha az *x* páratlan, és hamisat, ha páros.

Pi függvény

Pi értékét adja vissza.

Pi : **real**;

Pi értékét adja vissza: 3.1415926535897932385. Az érték pontossága a hardver ($N+) vagy szoftver ($N-) számítási módtól függ.

Random függvény

Visszatér egy véletlenszámmal.

> ***Random*** [(*range*: **word**)] : *range* függvénye;
> *range*   word típusú kifejezés.

Ha a *range* nincs megadva, akkor valós típusú 0 <= x < 1, egyébként word típusú 0 <= x < *range* véletlenszámot generál. Ha a *range* értéke 0, akkor a függvény értéke is 0 lesz. A véletlenszám generátornak kezdő értéket lehet adni a *Randomize* eljárással, vagy a *RandSeed* értékadásával.

Randomize eljárás

A beépített véletlenszám generátornak kezdőértéket ad.

> ***Randomize***;

A rendszer órája alapján számítja a véletlenszám generátor kezdőértékét.

Round függvény

Valós típusú kifejezést a legközelebbi egészre kerekíti.

> ***Round***(*x*: **real**) : **longint**;
> *x*   valós típusú kifejezés.

Az *x* valós kifejezést a legközelebbi egészre kerekíti. Ha az *x* kerekített értéke nem esik bele a Longint típus értékkészletébe, futási hiba lép fel. Ha a tizedesrész 0.5-nél nagyobb, a nagyobb abszolút értékűt egészre kerekíti.

Sin függvény

Visszatér az argumentum szinusz értékével.

> ***Sin***(*x*: **real**) : **real**;
> *x*   valós típusú kifejezés.

Az *x* értékét radiánban kell megadni.

Sqr függvény
Visszatér az argumentum négyzetével.

Sqr(*x*) : azonos *x* típusával;
x egész vagy valós típusú kifejezés.

Sqrt függvény
Visszatér az argumentum négyzetgyökével.

Sqrt(*x*: **real**) : **real**;
x valós típusú kifejezés.

Trunc függvény
Visszatér az argumentum egész részével.

Trunc(*x*: **real**) : **longint**;
x valós típusú kifejezés.

Az *x* valós típusú kifejezés nulla felé kerekített egész részét adja vissza. Ha a kerekített érték nem esik bele a **Longint** típus értékkészletébe, akkor futási hibát okoz.

F1.2 Függvények a megszámlálható típusokra

Chr függvény
Visszatér a sorszámnak megfelelő karakterrel.

Chr(*x*: **byte**) : **char**;
x egész típusú kifejezés.

Eredményként az ASCII kódnak megfelelő karaktert adja vissza.

Dec eljárás
Csökkenti a változó értékét.

> ***Dec***(**var** *x* [; *n*: **longint**]);
> *x* sorszámozott típusú változó,
> *n* egész típusú kifejezés.

A *Dec* eljárás az *x* értékét csökkenti eggyel, illetve *n* értékével, ha meg van adva.A *Dec(x)* megfelel x:=x-1 illetve *Dec(x,n)* pedig x:=x-n utasításnak.

Inc eljárás
Növeli az argumentum értékét.

> ***Inc***(**var** *x* [; *n*: **longint**]);
> *x* sorszámozott típusú változó,
> *n* növekmény.

A *Inc* eljárás az *x* értékét növeli eggyel, illetve *n* értékével, ha meg van adva. A *Inc(x)* megfelel x:=x+1 illetve *Inc(x,n)* pedig x:=x+n utasításnak.

Pred függvény
A sorszámozott típus előző sorszámát adja vissza.

> ***Pred***(*x*);
> *x* sorszámozott típus.

A *Pred* függvény az *x* sorszámozott (egész, karakter, logikai) típusnak előző sorszámát adja vissza.

Succ függvény
A sorszámozott típus következő sorszámát adja vissza.

> ***Succ***(*x*);
> *x* sorszámozott típus.

A *Succ* függvény az *x* sorszámozott (egész, karakter, logikai) típusnak következő sorszámát adja vissza.

F1.3. Sztring kezelések

Concat függvény
Karakterláncokat fűz össze.

Concat(*s1* [, *s2*, ..., *sn*]: **string**) : **string**;
s1,... sztring típusú.

Minden paraméter sztring típusú kifejezés. Eredmény az összefűzött sztring. Ha az eredmény hosszabb 255 karakternél, akkor a 255. karakter utániakat levágja. A függvény helyettesíthető a + művelettel.

Copy függvény
Részkarakterlánccal tér vissza.

Copy(*s*: **string**; *index*: **integer**; *count*: **integer**) : **string**;

s	sztring, ahonnan másolunk,
index	karakter sorszáma, amelytől másolunk,
count	a másolandó karakterek darabszáma.

Az *s* karakterláncnak *index* pozíciójától *count* darabot másol és eredményként a részláncot adja vissza. Ha az *index* nagyobb, mint a sztring hossza, az eredmény üres sztring lesz. Ha a *count* nagyobb, mint amennyi karakter van az *s* sztringben az index pozíciótól, akkor csak a hátralévő maradékot adja vissza.

Delete eljárás
Részlánc törlése.

Delete(**var** *s*: **string**; *index*: **integer**; *count*: **integer**);

s	sztring típusú változó, amelyből törlünk,
index	karakter pozíció, amelytől törlünk,
count	a törölni kívánt karakterek száma.

Az *s* sztringnek *index* pozíciójától *count* darabot törlünk. Ha az *index* nagyobb, mint a sztring hossza, nem lesz törlés. Ha a *count* nagyobb, mint az

index pozíciótól a maradék karakterek száma, akkor csak a maradék karakterek törlődnek.

Insert eljárás
Részlánc beszúrása.

Insert(*source*: **string**; **var** *s*: **string**; *index*: **integer**);
source a beszúrandó részlánc,
s az eredmény karakterlánc,
index a karakter pozíció sorszáma, ahonnan beszúrás történik.

A *source* karakterláncot az *s* karakterlánc *index* pozíciójától beszúrja. Ha az eredmény sztring hossza nagyobb 255 karakternél, a 255. karakter utániakat levágja.

Length függvény
Visszatér a karakterlánc hosszával.

Length(*s*: **string**) : **integer**;
s karakterlánc.

Ord függvény
Visszatér a sorszámozott típusú kifejezés sorszámával.

Ord(*x*) : **longint**;
x sorszámozott típusú kifejezés.

Pos függvény
Karakterláncban részláncot keres.

Pos(*substr*, *s*: **string**) : **byte**;
substr részlánc,
s vizsgálandó karakterlánc.

A *Pos* függvény az *s* karakterláncban keresi a *substr* részláncot, eredményként az *s* karakterláncban megtalált *substr* részlánc első karakterének pozícióját adja, ha nem találja a részláncot, akkor 0-val tér vissza.

Str eljárás
Numerikus értéket karakterlánccá alakít.

Str(*x* [: *width* [: *decimals*]]; **var** *s*: **string**);

x	egész vagy valós típusú kifejezés,
width	mezőszélesség,
decimals	tizedes jegyek száma,
s	eredmény sztring.

Az *x* egész vagy valós típusú kifejezés értékét konvertálja *width* mezőszélességgel és *decimals* tizedesek számával az *s* sztringbe.

UpCase függvény
A karaktert nagybetűvé alakítja.

UpCase(*ch*: **char**) : **char**;
ch kisbetű karakter.

Az *ch* kisbetű karaktert nagybetűvé alakítja. Ha a karakter nincs az 'a'-'z' interallumban, akkor a függvényhívás hatástalan.

Val eljárás
Karakterláncot numerikus értékké konvertálja.

Val(*s*: **string**; **var** *v*; **var** *code*: **integer**);

s	a konvertálandó karakterlánc,
v	egész vagy real típusú változó,
code	a hibás első karakter pozíciója.

A *Val* függvény az *s* karakterláncot konvertálja a *v* típusának megfelelő numerikus értékké. Ha a sztringben illegális karakter van, a *code* változó tartalmazza a hibás első karakter pozícióját, különben értéke 0 lesz.

F1.4. Byte és regiszter műveletek

CSeg függvény
Visszatér a CS regiszter aktuális értékével.

Seg : word;

DSeg függvény
Visszatér a DS regiszter aktuális értékével.

Seg : word;

Hi függvény
Visszatér az argumentum felső byte-jával.

Hi(*x*) : byte;
x egész vagy word típusú kifejezés.

Lo függvény
Visszatér az argumentum alsó byte-jával.

Lo(*x*) : byte;
x egész vagy word típusú kifejezés.

Ofs függvény
Az adott objektum eltolási (offset) címét adja vissza.

Ofs(*x*) : word;
x bármilyen típusú változó, eljárás vagy függvény azonosító.

Seg függvény
Visszatér az adott objektum szegmens címével.

Seg(*x*) : word;
x bármilyen típusú változó, eljárás vagy függvény azonosító.

SPtr függvény
Visszatér az SP regiszter aktuális értékével.

SPtr : word;

SSeg függvény
Visszatér az SS regiszter aktuális értékével.

SSeg : word;

Swap függvény
Felcseréli az argumentum felső és alsó byte-ját.

Swap(*x*) : azonos x típusával;
x egész vagy word típusú kifejezés.

F1.5. Könyvtárak kezelése

ChDir eljárás
Átváltja az aktuális tartalomjegyzéket.

ChDir(*s*: **string**);
s sztring típusú kifejezés.

Az aktuális tartalomjegyzéket átváltja az *s* sztringben megadott tartalomjegyzékre. Az *s* tartalmazhat meghajtót is. A DOS az itt megadott útvonalat és meghajtót tekinti majd aktuálisnak a program lefutása után.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

DiskFree függvény **Dos**

Visszatér a specifikált diszk meghajtó szabad byte-jának számával.

DiskFree(*Drive*: **byte**) : **longint**;

Drive	a meghajtó száma, 0 az aktuális, 1 az A, 2 a B stb.

DiskSize függvény **Dos**

Visszatér a specifikált diszk meghajtó teljes méretével byte-okban.

DiskSize(*Drive*: **byte**) : **longint**;

Drive	a meghajtó száma, 0 az aktuális, 1 az A, 2 a B stb.

FindFirst eljárás **Dos**

Keresi a specifikált vagy az aktuális tartalomjegyzékben az első adott nevű és tulajdonságú bejegyzést.

FindFirst(*Path*: **string**; *Attr*: **word**; **var** *s*: **Searchrec**);

Path	sztring típusú kifejezés (tartalomjegyzék maszk, pl. *.*),
Attr	file tulajdonság,
s	*Searchrec* típusú rekord tartalmazza a hívás eredményét.

A *Path* paraméterben meghatározott normál és *Attr* tulajdonságú katalógus-bejegyzések közül kikeresi az elsőt és elhelyezi az *s* rekordban.

A DOS **Unit**-ban vannak definiálva az alábbi konstansok:

```
const
      Readonly   = $01;   { csak olvasható }
      Hidden     = $02;   { rejtett file }
      SysFile    = $04;   { rendszer file }
      VolumeID   = $08;   { lemez neve }
      Directory  = $10;   { tartalomjegyzék bejegyzés }
      Archive    = $20;   { archivált file }
      AnyFile    = $3F;   { bármilyen típusú file }
```

A *SearchRec* típus:

```
type
        Searchrec = record
            Fill:   array[1..21] of byte;
            Attr:   byte;
            Time:   longint;
            Size:   longint;
            Name: string[12];
        end;
```

FindNext eljárás **Dos**

Az előzőleg *FindFirst* eljárással definiált katalógus-bejegyzések közül a következőt adja vissza.

FindNext(var *s*: ***Searchrec***);

s *Searchrec* típusú rekordban keletkezik az eredmény.

FSearch függvény **Dos**

Egy fizikai állományt keres aktuális lemezegység aktuális katalógusában.

FSearch(*Path*: ***PathStr***; *DirList*: **string**) : ***PathStr***;

Path *PathStr* típusú kifejezés,

DirList sztring típusú kifejezés.

Keresi a *Path* paraméterben megadott fizikai állományt az aktuális lemezegység katalógusában, majd a *DirList* paraméterben definiált katalógusokban. Ha megtalálta, akkor a teljes állomány-specifikációt adja vissza, különben a függvény értéke üres karakterlánc lesz. A dinamikus keresés céljából a *DirList* paramétert a *GetEnv('PATH')* hívásával helyettesítjük.

FSplit eljárás **Dos**

A paraméterben megadott állomány specifikációt komponenseire bontja.

FSplit(*Path*: ***PathStr***; var *Dir*: ***DirStr***;
var *Name*: ***NameStr***; var *Ext*: ***ExtStr***);

Path állomány specifikáció,

Dir meghajtó és az útvonal,

Name file neve,

Ext kiterjesztés a ponttal együtt.

A DOS Unit-ban az alábbi típusok vannak definiálva:

```
type
        PathStr    = string[79];
        DirStr     = string[67];
        NameStr    = string[8];
        ExtStr     = string[4];
```

GetDir eljárás
Visszatér a megadott meghajtó aktuális tartalomjegyzékével.

GetDir(*d*: **byte**; **var** *s*: **string**);

d egész típusú kifejezés a meghajtó, 0 az aktuális, 1 az A, 2 a B stb.

s sztring típusú változó, tartalmazza eredményül az aktuális útvonalat.

A *d* paraméterben definiált meghajtón *s*-be adja az aktuális útvonalat.
A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

MkDir eljárás
Tartalomjegyzéket hoz létre.

MkDir(*s*: **string**);

s sztring típusú kifejezés.

Létrehozza az *s* paraméterben megadott tartalomjegyzéket, az *s* tartalmazhat meghajtót is. Az eljárás hívása megfelel a DOS MD parancsának.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

RmDir eljárás
A tartalomjegyzék törlése.

RmDir(*s*: **string**);

s sztring típusú kifejezés.

Törli az s paraméterben megadott tartalomjegyzéket, az *s* tartalmazhat meghajtót is. Az eljárás hívása megfelel a DOS RM parancsának.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

F1.6. File kezelések

Append eljárás
Létező file-t nyit meg adatok file végéhez való hozzáírására.

Append(var *f*: **text**);
f **text** típusú file változó.

Az *f* text típusú file változó, amelyet korábban *Assign* utasításban használtunk külső file létrehozására.

Az *Append* létező file-t nyit meg a megadott file névvel és az *f* azonosítóval. Hibajelzést kapunk, ha az adott néven a file nem létezik. Ha az *f* file nincs lezárva, akkor először megtörténik a file lezárása, majd újramegnyitása. A file mutató a file végére mutat és a file csak írásra használható.
A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a nyitási művelet sikeres volt, különben a hibakódot adja vissza.

Assign eljárás
Külső file nevéhez file változót rendel.

Assign(var *f*; *name*: **string**);
f bármilyen típusú file változó,
name sztring típusú kifejezés, a fizikai file-nevet tartalmazza.

Minden további művelet az *f*-el azonosított adott nevű file-on történik. Addig létezik, míg egy másik *Assign* felül nem írja.

A file név lehet üres sztring, vagy akár tartalmazhat meghajtót és több al-tartalomjegyzéket, amely \ (backslash) jellel van elválasztva.

meghajtó:\direktori\...\direktori\file név

Pl: c:\munka\kod\dat.dat

A file név maximális hossza 79 karakter lehet. Ha a file név üres sztring, akkor a *Reset(f)* hívásakor a szabványos input file ill. *Rewrite(f)* hívásakor a szabványos output file lesz.

BlockRead eljárás
Egy változóba egy vagy több blokkot olvas.

***BlockRead*(var *f*: file; var *buf*; *count*: word**
[, var *result*: word]);

f	típus nélküli file változó,
buf	bármilyen változó,
count	word típusú kifejezés,
result	word típusú változó.

Az *f* típus nélküli állományból *count* rekordot olvas és elhelyezi a *buf* változó memóriacímétől kezdődően. A *result* változóban kapjuk meg azt az értéket, amelyet az eljárás valójában beolvasott. A *result* megadása nem kötelező, ha azonban nem tudta a megadott számú rekordot beolvasni I/O hibajelzést kapunk. 128 byte a rekordméret, ha az *f* állomány megnyitásakor nem adtuk meg. Maximálisan 65535 byte lehet a *count* * rekordméret.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

BlockWrite eljárás
Egy vagy több blokkot ír ki egy változóba.

BlockWrite(var *f*: **file**; var *buf*; *count*: **word**
[, var *result*: **word**]);

f	típus nélküli file változó,
buf	bármilyen típusú változó,
count	word típusú kifejezés,
result	word típusú változó.

Az *f* típus nélküli állományba *count* rekordot ír ki a *buf* változó címétől. A result változóban kapjuk meg azt az értéket, amelyet az eljárás valójában kiírt. A *result* megadása nem kötelező, ha azonban nem tudta a megadott számú rekordot kiírni I/O hibajelzést kapunk. 128 byte a rekordméret, ha az *f* állomány megnyitásakor nem adtuk meg. Maximálisan 65535 byte lehet a *count* * rekordméret.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Close eljárás
Nyitott file-t zár le.

Close(var *f*);
f bármilyen típusú file változó

Reset, Rewrite vagy *Append* utasítással megnyitott file-t zár le.

A {$I-} direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Eof függvény (text típusú file)
Visszatér a text típusú file end-of-file státuszával.

Eof [(var *f*: **text**)] : **boolean**;
f *text* típusú file változó

Ha az *f*-et elhagyjuk a függvény a standard Input állományt vizsgálja. Az *Eof(f)* értéke igaz, ha a file mutató a file végére mutat, vagy a file üres, egyébként hamis.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Eof függvény (típusos vagy típus nélküli file)
Visszatér a típusos vagy típus nélküli file end-of-file státuszával.

Eof(**var** *f*) : **boolean**;
f file változó.

Az *Eof(f)* értéke igaz, ha a file mutató az file végére mutat, vagy a file üres, egyébként hamis.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Eoln függvény
Visszatér a file *end-of-file* státuszával.

Eoln(**var** *f*: **text**) : **boolean**;
f *text* típusú file változó.

Ha az f-et elhagyjuk a függvény a standard Input állományt vizsgálja. Az *Eoln(f)* értéke igaz, ha a file mutató a sor végére mutat, vagy az *Eof(f)* igaz, egyébként hamis. Nyitott állományra nem szabad használni.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

FilePos függvény
Visszatér az aktuális file pozíciójával.

FilePos(**var** *f*) : **longint**;
f file változó.

Ha a file mutató a file elejére mutat, a *FilePos(f)* 0-val, ha a file végére mutat, akkor a file méretével *FileSize(f)* értékével tér vissza.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

FileSize függvény
A file méretét adja vissza.

> ***FileSize***(*f*) : **longint**;
> *f* file változó.

A *FileSize(f)* visszatér a file-ben lévő komponensek számával, ha a file üres, a visszatérési érték 0 lesz. Csak nyitott file esetén kérdezzük meg a file méretét és ne használjuk az utasítást ha a file text típusú.

Flush eljárás
Text típusú file pufferjének kiírása az outputra.

> ***Flush***(var *f*: **text**);
> *f* text típusú file változó.

Rewrite vagy *Append* utasítással outputra megnyitott file esetén a *Flush* utasítás az átmeneti puffer tartalmát kiírja. Olvasásra megnyitott állományok esetén az utasításnak nincs hatása.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

IOResult függvény
Az utoljára végrehajtott Input/Output művelet hibakódjával tér vissza.

> ***IOResult*** : **word**;

A {$I-} fordító direktívával történő fordítás esetén, amikor a periféria műveletek inaktívak, a függvény a System egység *InOutRes* változójának

értékét veszi fel, ami egy DOS I/O hibakód, majd törli a belső változó értékét. Hiba esetén a belső változó értéke nem nulla, ebben az esetben az I/O művelet nem kerül végrehajtásra.

Read eljárás (text file)
Egy vagy több értéket olvas szöveg állományból.

> ***Read***([*var f*: **text**;] *v1* [, *v2*, ..., *vn*]);
> *f* **text** file változó.

Ha az *f* hiányzik, akkor a standard Input állományból olvas. A változók lehetnek egész, valós, karakteres és sztring típusúak. Az eljárás az aktuális pozíciótól kezdő értéke(ke)t olvassa a paraméter típusától függően, majd elhelyezi a felsorolt változó(k)ban. A file mutató az utolsó beolvasott karakter mögé áll. Numerikus változók olvasásakor a TAB, szóköz vagy sorvégjel (CR/LF) karaktereket nem veszi figyelembe. Sztring esetén a sorvég jelig olvas, ha a változó rövidebb a szöveg csonkulni fog.

Read eljárás (típusos file)
Változóba file komponenst olvas.

> ***Read***(*f*, *v1* [, *v2*, ..., *vn*]);
> *f* bármilyen típusú file változó, kivéve a text típust.

A változók típusának meg kell egyeznie az állományban lévő adatok típusával. Csak nyitott file-ból lehet olvasni. A file mutató minden olvasás után automatikusan a következő komponensre lép, ha az utolsó komponens után áll, olvasás esetén I/O hibát kapunk.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Readln eljárás
Végrehajt egy *Read* utasítást és a file következő sorára pozíciónál.

> ***Readln***([var *f*: **text**;] *v1* [, *v2*, ..., *vn*]);

A *Readln* egy kiterjesztett *Read* utasítás, amely csak szöveges állományokra használható. A változó(k)ba történő beolvasás után a *Readln* a file mutatót, illetve a kurzort a következő sor elejére állítja. Ha az *f* hiányzik, a standard Input állományból olvas és az olvasás után a puffer törlődik.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Rename eljárás
File-t átnevez.

> ***Rename***(**var** *f*: *newname*: **string**);
> *f* bármilyen típusú file változó,
> *newname* sztring típusú kifejezés.

Az *f*-hez rendelt fizikai állományt átnevezi a *newname* változóban megadott névre. Nyitott állományra nem szabad használni. Megfelel a DOS REN parancsának.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Reset eljárás
Létező file-t nyit meg.

> ***Reset***(**var** *f* [; **file**, *recsize*: **word**])
> *f* bármilyen típusú file,
> *recsize* opcionális **word** típusú kifejezés, csak típus nélküli file esetén kell megadni.

Létező file-t nyit meg, azzal a névvel amelyet *Assign* utasítással az f-hez rendeltünk. Ha az adott nevű file nem létezik, hibajelzést kapunk, ha a file már nyitva volt, akkor először lezárja, majd újranyitja - ebben az esetben az előző file elveszik - a file mutató a file elejére mutat.
Assign(f,'') esetén a file neve üres sztring, ilyenkor a standard Input állomány lesz megnyitva.

Ha az *f* szöveg állományhoz van rendelve, a **Reset** utasítás csak olvasásra nyitja meg az állományt, ilyenkor az *Eof(f)* igaz, az állomány üres. Típus nélküli állomány esetén a *recsize* változó adja meg a rekord méretét. Ha a *recsize* változó hiányzik a paraméterlistáról, a rekord mérete 128 byte lesz.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Rewrite eljárás
Létrehoz és megnyit egy új file-t.

> *Rewrite*(**var** *f* [: **file**; *recsize* : **word**]);
>
> *f* — bármilyen típusú file,
>
> *recsize* — opcionális word típusú kifejezés, csak típus nélküli file esetén kell megadni.

A *Rewrite* utasítás új file-hoz létre azzal a névvel, amelyet az *Assign* utasítással az *f*-hez rendeltünk. Ha a file már létezett, törli és a helyén újat hoz létre. Ha nyitva volt, lezárja és újra létrehozza. A file mutató a file elejére mutat.

Ha a file neve üres sztring, akkor a *Rewrite* a standard Output file-t fogja megnyitni.

Tipus nélküli állomány esetén a *recsize* változó adja meg a rekord méretét. Ha a *recsize* változó hiányzik a paraméterlistáról, a rekord mérete 128 byte lesz.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Seek eljárás

A file mutatót a megadott komponensre állítja.

> *Seek*(**var** *f*: *n*: **longint**);
> *f* file változó, text típusú nem lehet,
> *n* longint típusú kifejezés.

Az aktuális file mutatót az *n* változóban megadott számú komponensre állítja. 0 esetén a file első komponensére, *FileSize(f)* esetén az utolsó komponensére fog mutatni a file mutató.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

SeekEof függvény

Visszatér a file *end-of-file* (file vége) státuszával.

> *SeekEof* [(**var** *f*: **text**)] : **boolean**;
> *f* text típusú file változó.

A függvény a szöveg állományban a következő tabulátort, szóközt és sorvégjelet átugorja és értéke igaz lesz, ha a mutató a file végére került. Ha az *f* hiányzik, akkor a standard Input állományról van szó.
Az utasítás csak nyitott file-ra adható ki.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

SeekEoln függvény

Visszatér a file *end-of-line* (sor vége) státuszával.

> *SeekEoln* [(**var** *f*: **text**)] : **boolean**;
> *f* **text** típusú file változó.

A függvény a szöveg állományban a következő tabulátort, szóközt és sorvégjelet átugorja és értéke igaz lesz, ha a mutató a sor végére került. Ha az *f* hiányzik, akkor a standard Input állományról van szó.
Az utasítás csak nyitott file-ra adható ki.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

SetTextBuf eljárás
Szöveg állományhoz I/O puffert rendel.

SetTextBuf(**var** *f*: **text**; **var** *buf* [; *size*: **word**])

f	text típusú file változó,
buf	bármilyen típusú változó,
size	word típusú opcionális változó.

Minden text típusú file változónak van egy belső 128 byte-os puffere. Ez a méret az I/O műveleteknél általában megfelelő. Azonban text file-ok másolásánál, konvertálásánál érdemesebb nagyobb pufferrel dolgozni.

Az eljárás a belső puffert kicseréli *buf* változóban megadott méretűre, ha a *size* változó is adott, akkor az abban megadott méretűre.

A *SetTextBuf* utasítást **Assign** utasítás után és a *Reset*, *Rewrite* vagy *Append* utasítások előtt kell aktiválni.

Truncate eljárás
Az aktuális file pozíciótól a file végéig törli a file-t.

Truncate(**var** *f*);

f	bármilyen típusú file változó.

Az *f* bármilyen típusú file aktuális pozíciójától törli a komponenseket a file végéig.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Write eljárás (text file)

Egy vagy több értéket ír **text** file-ba.

> ***Write***([var *f*: **text**;] *v1* [, *v2*, ..., *vn*]);
> *f* text típusú file változó,
> *v1*, ... a kiírandó paraméterek.

Az eljárás az *f* szöveg állományba az aktuális pozícótól kezdve kiírja a *v1* [*v2*, ...] kifejezés(ek) értékét. A kifejezések típusa lehet karakter, egész, valós, sztring és logikai. A file mutató az utolsó kiírt karakter mögé áll. Ha az *f* hiányzik, akkor a standard Output állományba ír.
A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Write eljárás (típusos file)

Egy változót ír ki típusos állományba.

> ***Write***(*f*, *v1* [, *v2*, ..., *vn*])
> *f* file változó.
> *v1*, ... változó típusa megegyezik a file komponens típusával.

Az *f* típusos állományba kiírja a *v1* [, *v2*, ...] változó(k) értékét. A file mutató minden írás után automatikusan a következő komponensre lép. Ha file mutató az utolsó komponens után áll, a file bővül.

A {$I-} fordító direktívával történő fordítás esetén az *IOResult* változó 0 lesz, ha a művelet sikeres volt, különben a hibakódot adja vissza.

Writeln eljárás

Write végrehajtása után egy sorvége jelet ír a file-ba.

> ***Writeln***([var *f*: **text**;] *v1* [, *v2*, ..., *vn*]);

Teljesen azonos a *Write* eljárással, melynek végrehajtása után sorvége jelet ír a file-ba.

F1.7. DOS és rendszer paraméterek kezelése

DosExitCode függvény **Dos**

Alprogram visszatérési kódját adja vissza.

DosExitCode : **word**;

Az Exec eljárással hívott program visszatérési kódját adja vissza. Az alsó byte a program által küldött kódot tartalmazza - a *Halt* vagy *Keep* eljárásnak adott paramétert -, míg a felső byte értéke:

0	ha az alprogram hiba nélkül lefutott,
1	ha *Ctrl C* billentyűkkel szakították meg,
2	futási hibával állt meg, vagy
3	a *Keep* eljárás hívása miatt fejeződött be futása.

DosVersion függvény **Dos**
Visszatér a DOS verzió számával.

DosVersion : **word**;

Az alsó byte a fő-, a felső byte az alverziószámot tartalmazza.

EnvCount függvény **Dos**
Visszatér a DOS környezeti változók számával.

EnvCount : **integer**;

Az *EnvCount* függvény az *EnvStr* sztring tömb elemeinek számát adja, amely tartalmazza a DOS környezetet.

EnvStr függvény **Dos**
A DOS környezeti sztring aktuális értékét adja.

EnvStr(*Index*: **integer**) : **string**;
Index egész típusú változó, a DOS környezeti változó aktuális eleme.

Az első DOS környezeti változó értéke 1. Ha az *Index* értéke kisebb egynél, vagy nagyobb az *EnvCount* függvény által visszaadottnál, akkor a függvény értéke üres sztring lesz.

Exec eljárás Dos

Végrehajt egy programot egy specifikált parancs sorral.

Exec(*Path*, *CmdLine*: **string**);

Path sztring típusú kifejezés tartalmazza a program nevét, és esetleg az elérési utat

CmdLine sztring típusú kifejzés pedig a parancs sort tartalmazza.

A heap maximális felső határát akkorára kell lecsökkenteni, hogy a futtatható program beférjen a memóriába. *DOSError* változó értéke 0, ha az eljárás eredményes volt. A hívott program visszatérési kódját a *DosExitCode* függvénnyel kérdezhetjük le. A COMMAND.COM futtatásakor a parancs sor első paraméterének *Control C* -nek kell lennie. Az *Exec* eljárás hívása előtt és után használjuk a *SwapVectors* eljárást.

Exit eljárás

Kilép az aktuális blokkból.

Exit

Az *Exit* eljárás hatására a vezérlés egy programszinttel feljebb kerül: kilép az aktuális függvényből, eljárásból illetve főprogramból. Az eljárás hívása hasonló egy **Goto** utasításhoz, amely a blokk **end** utasítására adja a vezérlést.

FExpand eljárás Dos

A file-evet kiegészíti a meghajtóval és a teljes útvonallal.

Expand(*Path*:**PathStr**) : **PathStr**;

Path a file-nevet tartalmazó *PathStr* (**string**[79]) típusú sztring

GetCBreak eljárás **Dos**

Visszatér a DOS *Ctrl-Break* vizsgálatának státuszával.

GetCBreak(**var** *Break*: **boolean**);

Break logikai változó.

Az eljárás a *Break* paraméterben adja vissza, hogy aktív-e a DOS *Ctrl-Break* ellenőrzése. Ha hamis az értéke, akkor a DOS csak periféria műveletek hívásakor fogadja el a *Ctrl-Break* által történő megszakítást, míg igaz esetén minden rendszer hívásakor.

GetDate eljárás **Dos**

Az operációs rendszer szerinti aktuális dátumot adja vissza.

GetDate(**var** *Year*, *Month*, *Day*, *DayofWeek*: **word**)

Year	az év dátuma: 1980-2099-ig,
Month	a hónap dátuma: 1 .. 12,
Day	nap dátuma: 1 .. 31,
DayofWeek	a hét napjainak sorszáma: 0 .. 6 (vasárnap sorszáma 0)

GetEnv függvény **Dos**

Visszatér az adott környezeti változó értékével.

GetEnv(*EnvVar*: **string**) : **string**;

EnvVar sztring típusú kifejezés.

A *GetEnv* függvény az *EnvVar* paraméterben meghatározott DOS környezeti változó értékét adja vissza. Ha nem létezik, üres karakterlánc lesz a visszatérési érték.

GetFAttr eljárás **Dos**

A file tulajdonságait adja vissza.

GetFAttr(**var** *f*; **var** *Attr*:**word**);

f bármilyen típusú file változó, amely *Assign* utasításban szerepelt, de nincs megnyitva.

Az *Attr* változóban kapott tulajdonságot and utasítással maszkolhatjuk, és az alábbi konstansokat használhatjuk:

```
const
    Readonly  = $01;    { csak olvasható }
    Hidden    = $02;    { rejtett file }
    SysFile   = $04;    { rendszer file }
    VolumeID  = $08;    { lemez neve }
    Directory = $10;    { tartalomjegyzék bejegyzés }
    Archive   = $20;    { archivált file }
    AnyFile   = $3F;    { bármilyen típusú file }
```

3-as hibakód keletkezik, ha érvénytelen az útvonal, 5 lesz a hibakód, ha a file nem érhető el.

GetFTime eljárás **Dos**

Visszatér az utolsó file írás dátumával és idejével.

GetTime(**var** *f*; **var** *Time*: *longint*);

f	bármilyen típusú file változó,
Time	az utolsó írás dátuma és ideje pakolt formában.

A dátumot és az időt az *UnpackTime* eljárással lehet szétválasztani.

GetIntVec eljárás **Dos**

Visszatér a specifikált vektorban a megszakítási rutin címével.

GetIntVec(*IntNo*: **byte**; **var** *Vector*: **pointer**);

IntNo	a megszakítási vektor száma (0 .. 255),
Vector	tartalmazza a megszakítási rutin címét.

GetTime eljárás **Dos**

Visszatér az operációs rendszer aktuális idejével.

GetTime(**var** *Hour*, *Minute*, *Second*, *Sec100*: **word**);

Hour	óra	: 0 .. 23,
Minute	perc	: 0 .. 59,
Second	másodperc	: 0 .. 59,
Sec100	század másodperc	: 0 .. 99.

GetVerify eljárás Dos

Visszatér a DOS *Verify* (ellenőrzés) jelzőjének aktuális állapotával.

GetVerify(var *Verify*: boolean);

Verify logikai kifejezés.

Ha a *Verify* változó igaz, akkor minden lemez írásakor külön ellenőrzés történik, egyébként nem.

Halt eljárás

Leállítja a program futását és visszaadja a vezérlést az operációs rendszernek.

Halt [(*ExitCode*: **word**)]

Exitcode word típusú kifejezés, a program vissztérési kódja.

Az *ExitCode* értéke bekerül a rendszer *ExitCode* változójába. A paraméter nélküli *Halt* hívása megfelel *Halt(0)* hívásnak.

Intr eljárás Dos

Szoftver megszakítást hajt végre.

Intr(*IntNo*: **byte**; **var** *Regs*: **Registers**);

IntNo byte típusú kifejezés, a szoftver megszakítás sorszáma, értéke : 0 .. 255 lehet.

Regs Registers típusú rekord.

A megszakítás hívása előtt a processzor AX, BX, CX, DX, BP, SI, DI, DS és a ES regisztereit a *Regs* rekordból tölti fel, visszatéréskor ezeket és a *Flags* regisztert is visszamenti a *Regs* rekordba.

Azon szoftver megszakítást, amely használja és módosítja az SP és SS regisztereket, ezzel az eljárással nem szabad meghívni.

```
type
     Registers = record
          case  integer of

0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags:word);
               1: (AL,AH,BL,BH,CL,CH,DL,DH: byte);
          end;
```

Keep eljárás **Dos**

Leállítja a program futását, de a program a memóriában marad.

Keep(*ExitCode*: **word**);

ExitCode **word** típusú kifejezés, a visszatérési kód.

A program visszatérési kódját a hívó programból *DosExitCode* függvénnyel kérdezhetjük le, batch állományban pedig az ERRORLEVEL segítségével.

MsDos eljárás **Dos**

Végrehajt egy DOS megszakítást.

MsDos(var *Regs*: **Registers**);

Regs **Registers** típusú rekord.

Az MsDos hívásának a hatása azonos az Intr eljárás IntNo = $21 paraméterrel történő meghívásával.

PackTime eljárás **Dos**

Átkonvertálja a *DataTime* rekordot 4 byte-ba.

PackTime(var *DT*: ***DataTime***; var *Time*: ***longint***);

DT *DataTime* típusú rekord,

Time **longint** típusú változó, ebben keletkezik az idő pakolt formában.

```
DataTime = record
           Year, Month, Day, Hour, Min, Sec: word;
           end;
```

ParamCount függvény
A program parancs sorában lévő paraméterek számát adja vissza.

ParamCount : **word**;

Ha a *ParamCount* kisebb, mint 1, a parancssorban nincs paraméter.

ParamStr függvény
A parancssor adott sorszámú paraméterét adja vissza.

ParamStr(*Index*: **word**) : **string**;

Index **word** típusú kifejezés, a parancssorban lévő paraméter sorszáma.

Ha az *Index* értéke 0 vagy nagyobb, mint a *ParamCount* függvény, akkor üres sztring a visszatérési érték. A *ParamStr(0)* értéke a futó programot tartalmazó katalógus teljes útvonalát és a program nevét adja.

RunError eljárás
Leállítja a program futását és futási hibát generál.

RunError [(*ErrorCode*: **word**)];

ErrorCode **word** típusú kifejezés, a hiba kódja.

0 lesz a kód, ha az *ErrorCode* hiányzik.

SetCBreak eljárás **Dos**
Beállítja a DOS-ban a Ctrl-Break ellenőrzésének állapotát.

SetCBreak(*Break*: **boolean**);

Break logikai típusú kifejezés.

Ha a *Break* értéke hamis, a DOS csak a periféria műveletek alatt figyeli a *Ctrl-Break* megnyomását, igaz érték esetén minden rendszer-híváskor.

SetDate eljárás **Dos**

Beállítja a operációs rendszer dátumát.

SetDate(*Year*, *Month*, *Day*: **word**);

Year	az év dátuma	: 1980-2099-ig,
Month	a hónap dátuma	: 1 .. 12,
Day	nap dátuma	: 1 .. 31.

SetFAttr eljárás **Dos**

Megváltoztatja a file tulajdonságait.

SetFAttr(**var** *f*; **var** *Attr*:**word**);

f bármilyen típusú file változó, amely *Assign* utasításban szerepelt, de nincs megnyitva.

Az *Attr* változóban a következő tulajdonságokat adhatjuk meg (Az and utasítással fűzhetünk össze többet):

```
const
      Readonly   = $01;   { csak olvasható }
      Hidden     = $02;   { rejtett file }
      SysFile    = $04;   { rendszer file }
      VolumeID   = $08;   { lemez neve }
      Directory  = $10;   { tartalomjegyzék bejegyzés }
      Archive    = $20;   { archivált file }
      AnyFile    = $3F;   { bármilyen típusú file }
```

3-as hibakód keletkezik, ha érvénytelen az útvonal, 5 lesz a hibakód, ha a file nem érhető el.

SetFTime eljárás **Dos**

Beállítja a file utolsó írásának a dátumát és idejét.

SetFTime(**var** *f*; *Time*: **longint**);

f	bármilyen típusú file változó,
Time	longint típusú változó, pakolt formában tartalmazza a dátumot és az időt.

A *SetFTime* eljárás hívásakor az állománynak nyitva kell lennie.

SetIntVec eljárás Dos

Beállítja a megadott megszakítási vektort egy megadott címre.

SetIntVec(*IntNo*: **byte**; *Vector*: **pointer**);

IntNo	byte típusú változó, a megszakítási vektor sorszámát tartalmazza (0 .. 255),
Vector	pointer típusú változó, tartalmazza a címet.

SetTime eljárás Dos

Beállítja az operációs rendszer idejét.

SetTime(*Hour*, *Minute*, *Second*, *Sec100*: **word**)

Hour	óra:	0 .. 23,
Minute	perc:	0 .. 59,
Second	másodperc:	0 .. 59,
Sec100	századmásodperc:	0 .. 99.

SetVerify eljárás Dos

Beállítja a DOS verify jelzőjének állapotát.

SetVerify(*Verify*: **boolean**);

Verify logikai típusú kifejezés.

Ha a *Verify* értéke igaz, minden disk műveletnél ellenőrzés történik.

SwapVectors eljárás Dos

Felcseréli a megszakítási vektorokat.

SwapVectors;

A *System* unitban lévő *SaveIntXX* mutatók tartalmát felcseréli a megszakítási vektorok tartalmával. Az *Exec* eljárás előtt és után kell használni.

UnpackTime eljárás **Dos**

Szétpakolja a 4 byte-os pakolt dátum és időt egy *DateTime* típusú rekordba.

UnpackTime(*Time*: **longint**; **var** *DT*: ***DateTime***);

F1.8. Overlay kezelés

OvrClearBuf eljárás **Overlay**

Törli az overlay puffer tartalmát.

OvrClearBuf;

Ha egy overlay egységről hívjuk az eljárást, akkor a puffer törlése után ismét betölti az egységet. Az *OvrClearBuf* használatát kizárólag speciális célra szánták, hogy ideiglenesen törölje az overlay puffer által elfoglalt memóriát.

OvrGetBuf függvény **Overlay**

Visszatér az overlay puffer aktuális méretével.

OvrGetBuf : **longint**;

Kezdetben az overlay puffer azt a legkisebb méretet foglalja le, amely a legnagyobb overlay szegmens számára elegendő. Az overlay program végrehajtásakor a puffer mérete automatikusan megállapításra kerül. Az overlay puffer mérete lehet nagyobb, mint 64 K, mivel az utasításokat és a legnagyobb *overlay* szegmens információit tartalmazza.

OvrGetRetry függvény **Overlay** **5.5**

Visszatér az overlay puffer próbálkozási területének aktuális méretével.

OvrGetRetry : longint;

Ez az érték az *OvrSetRetry* eljárás által utoljára adott érték.

OvrInit eljárás **Overlay**

Inicializálja az overlay-kezelőt és megnyitja az overlay file-t.

OvrInit(*FileName*: **string**);

FileName	sztring típusú kifejezés, overlay file neve;

Ha *FileName* paraméter nem tartalmaz meghajtót vagy útvonalat, az overlay-kezelő a file-t először az aktuális tartalomjegyzékben, majd az .EXE állományt tartalmazó katalógusban, végül a DOS PATH környezeti változójával meghatározott katalógusokban keresi.

Az *OvrResult* változó tartalmazza a hibajelzést.

ovrOK	sikeres a végrehajtás,
ovrError	az overlay file formátuma nem megfelelő, vagy a program nem overlay típusú,
ovrNotFound	jelenti, hogy az overlay file-t nem tudja betölteni.

208-as futási hibát kapunk, ha az overlay-kezelő nincs installálva.

OvrInitEMS eljárás **Overlay**

Betölti az overlay file-t az EMS memóriába.

OvrInitEMS;

Ha van elég hely az EMS memóriában, az *OvrInitEMS* eljárás betölti oda az overlay file-t, majd lezárja azt, és ezután a lemezről való betöltést a gyors memóriamásolás váltja fel. A programfutás végén az EMS memória automatikusan felszabadul.

Az *OvrResult* tartalmazza a hibajelzést.

ovrOK	nincs hiba,
ovrError	*OvrInit* hibás, vagy nincs aktiválva,
ovrIOError	I/O hiba történt az overlay file olvasása közben,
ovrNoEMSDriver	EMS meghajtót nem érzékelt,
ovrNoEMSMemory	nincs elég EMS memória.

Ha CONFIG.SYS file-ban EMS-en alapuló RAM diszk parancs van, akkor bizonyosodjon meg arról, hogy van még elegendő hely az overlay működésére.

OvrSetBuf eljárás Overlay

Beállítja az overlay puffer méretét.

OvrSetBuf(*BufSize*: **longint**);

BufSize **longint** típusú kifejezés, az overlay puffer mérete.

A *BufSize* paraméternek nagyobb vagy egyenlőnek kell lennie az overlay puffer kezdeti méreténél, és kisebb vagy egyenlő mint

```
MemAvail+OvrGetBuf.
```

Az eljárás hívásakor a heap-nek üresnek kell lennie, ezért szüntessük meg az összes dinamikus változót és zárjuk le a grafikus üzemmódot.

Ha az *OvrSetBuf* eljárással növelni kívánja az overlay puffer méretét, akkor használja egyidejűleg a {M fordítási direktívát a heap minimum méretének növelésére.

Hibajezések az *OvrResult* változóban:

ovrError	*OvrInit* hívása hibás, vagy nincs aktiválva, *BufSize* mérete túl kicsi, vagy a heap nem üres,
ovrNoMemory	nincs elég memória az overlay puffer méretének növelésére.

OvrSetRetry eljárás Overlay **5.5**

Beállítja az adott méretre az overlay puffer probálkozási területét.

OvrSetRetry(*Size*: **longint**);

Size **longint** típusú kifejezés, overlay puffer mérete.

Ha a próbálkozási terület elegendő egy újabb overlay egység betöltéséhez, akkor az overlay-kezelő nem törli a már betöltött szegmenst.

Az overlay puffer bármely szabad területe a próbálkozási területen van figyelembe véve. Az overlay-kezelő korábbi verziójának kompatibilitása

céljából alapértelmezésben a próbálkozási terület nulla, mely lehetetlenné teszi a próbálkozási/végehajtási mechanizmust. A probálkozási terület optimális méretének meghatározására nincs empirikus formula, azonban a tapasztalat szerint az overlay puffer méretének körülbelül 1/3-a az optimális adat.

F1.9. Pointerek kezelése

Addr függvény
Visszatér a specifikált objektum címével.

Addr(*x*) ; **pointer**;
x bármilyen változó, eljárás vagy függvény.

Megadja az *x* változóra, eljárásra vagy függvényre mutató pointert.
A függvényhívás helyettesíthető a @*x* operátorral.

Dispose eljárás
Felszabadítja a dinamikus változót.

Dispose(**var** *p*: **pointer** [; *destructor*])

p	bármilyen típusra mutató pointer,
destructor	destruktor azonosító.

A *Dispose* eljárás felszabadítja a *New* eljárással létrehozott *p* pointer heap-beli területét, ezután a *p* határozatlan lesz, a rá való hivatkozás futási hibát okozhat. Ha az eljárás hívásakor a *p* nem a heap-be mutat, futási hiba lép fel.

Turbo Pascal 5.5 verzióban a dinamikus objektumok megszüntetésére használható, ekkor második paraméterként az objektum destruktorának neve is megadható. Ebben az esetben a destruktor végrehajtódik a *Dispose* utasítás végrehajtása előtt.

FillChar eljárás
Feltölti a memóriát a változó kezdőcímétől egy megadott értékkel.

FillChar(**var** *x*; *count*: **word**; *value*);

x	bármilyen típusú változó,
count	word típusú kifejezés,
value	sorszámozott típusú kifejezés.

Az *x* változó kezdőcímétől feltölti a memóriát *count* darab *value* kifejezés értékével. Mivel nincs memóriaellenőrzés, óvatosan kell használni.

FreeMem eljárás
Felszabadít a heap-ből egy dinamikus változóhoz rendelt adott méretű területet.

FreeMem(**var** *p*: **pointer**; *size*: **word**);

p	bármilyen típusra mutató pointer, amely a *GetMem* eljárás hívásával kapott értéket,
size	word típusú kifejezés, a felszabadítandó terület byte-okban.

A *FreeMem* felszabadítja a heap-ből a *p* pointer mutatóhoz rendelt size méretű területet és ezáltal határozatlanná válik. A *p* pointerre való további hivatkozás futási hibát okozhat. Ha a size nem egyezik meg a *p* mutatóval mutatott terület méretével, a heap megsérülhet. Ha az eljárás hívásakor a *p* nem a heap-be mutat, futási hiba lép fel.

GetMem eljárás
Adott méretű területet foglal le a heap-ben és a területre mutató címet elhelyezi egy pointerben.

GetMem(**var** p: **pointer**; *size*: **word**);

p	a lefoglalt területre mutató bármilyen típusú pointer,
size	**word** típusú kifejezés, a lefoglalandó terület méretét tartalmazza **byte**-okban.

A *GetMem* eljárással dinamikusan foglalhatunk le size méretű területet a heap-ből. Az újonnan létrehozott változóra *p^* hivatkozhatunk. Ha nincs

elég összefüggő szabad terület a heap-ben, akkor hibajelzést kapunk. 65521 byte méretű az a legnagyobb terület, melyet lefoglalhatunk.

Mark eljárás
Egy pointer változóba másolódik a heap-mutató értéke.

Mark(var *p*: **pointer**);
p bármilyen típusú pointer.

A Mark eljárás a *p* pointerbe másolja a heap-mutató (*HeapPtr*) értékét, amelyet később a *Release* eljárással lehet visszaállítani.

MaxAvail függvény
Visszatér a legnagyobb összefüggő terület méretével, amelyet a heap-ben lehet egyidejűleg dinamikusan lefoglalni.

MaxAvail : **longint**;

A *MaxAvail* függvény byte-okban adja vissza a heap-ben lévő legnagyobb összefüggő szabad terület méretét.

MemAvail függvény
Visszatér a heap-ben lévő összes szabad terület méretével.

MemAvail : **longint**;

A *MemAvail* függvény byte-okban adja vissza a heap-ben lévő összes szabad terület méretét.

Move eljárás
Átmásol egy meghatározott nagyságú részt egyik változóból a másikba.

Move(var *source*, *dest*; *count*: **word**);
source, dest bármilyen típusú változó,
count word típusú kifejezés.

A *Move* eljárás *count* darab byte-ot átmásol a *source* változó kezdőcímétől a *dest* változóba, a *dest* változó első byte-jától kezdődően. A két változó esetleges átfedését az eljárás figyelembe veszi, és a másolást úgy hajtja végre. Nem végez viszont memória- és típusellenőrzést, ezért óvatosan használjuk.

New eljárás
Létrehoz a heap-ben egy dinamikus változót és a címét elhelyezi benne.

New(**var** *p*: **pointer** [; **constructor**]);
p bármilyen típusú pointer,
constructor konstruktor azonosító.

Az újonnan létrehozott változóra *p^* hivatkozhatunk. Ha nincs elég hely a heap-ben az új változó létrehozására, akkor hibajelzést kapunk.

Turbo Pascal 5.5 verzióban a dinamikus objektumok létrehozására használható, ekkor második paraméterként az objektum konstruktorának neve is megadható. Ebben az esetben a konstruktor végrehajtódik a *New* utasítás végrehajtása után.

Ptr függvény
Pointer típusú mutatóvá konvertálja a szegmens : eltolás címet.

Ptr(*Seg*, *Ofs*: **word**) : **pointer**;
Seg,*Ofs* word típusú kifejezések.

A *Ptr* függvény a *Seg* és az *Ofs* paraméterekben megadott címet mutató típusúvá konvertálja.

Release eljárás
Pointer értékének átmásolása a heap-mutatóba.

Release(**var** *p*: **pointer**);
p bármilyen típusú pointer.

A *Release* eljárás átmásolja a *Mark* eljárás által kapott *p* pointer értékét a heap-mutatóba (*rHeapPt*).

SizeOf függvény

Visszatér az argumentumnak byte-okban elfoglalt értékével.

SizeOf(*x*) : **word**;

x bármilyen változó, vagy típus azonosító.

A *SizeOf* függvény az *x* által elfoglalt byte-ok számát adja vissza. TP 5.5-ben VMT/vel rendelkező objektum típusú változó esetén a VMT mérete lesz a függvény értéke.

TypeOf függvény **5.5**

Visszatér az objektum típusú változóhoz tartozó VMT címével.

TypeOf(*x*) : **pointer**;

x olyan objektum típus, amely rendelkezik VMT/vel.

F2. A CRT UNIT ELJÁRÁSAI ÉS FÜGGVÉNYEI

AssignCrt eljárás
A képernyőhöz rendel egy szöveg file-t.

AssignCrt(**var** *f*: **text**);

Paraméter: *f* szöveg file neve.
Megjegyzés: A képernyőhöz való hozzárendelés utána a file-t meg kell nyitni és a **writeln** hatására a szöveg a képernyőn jelenik meg.

ClrEol eljárás
Törli az összes karaktert a sor végéig anélkül, hogy a kurzor elmozdulna.

ClrEol;

Megjegyzés: A törlés után a kurzor helyétől a sor a képernyő széléig háttérszínű lesz.

ClrScr eljárás
Törli a képernyőt és a kurzort a bal felső sarokba helyezi.

ClrScr;

Megjegyzés: Ha a háttér nem volt fekete, akkor a törlés után a háttér felveszi az előzőkben definiált háttér színét.

Delay eljárás
Adott számú ezredmásodpercig felfüggeszti a program futását.

Delay(*ms*: **word**);

Paraméter: *ms* a késleltetés ideje.

DelLine eljárás
Törli a kurzort tartalmazó sort.

DelLine;

Megjegyzés: A kurzort tartalmazó sor törlődik és az alatta lévő sorok egy sorral feljebb lépnek a képernyőn. Az utolsó sor üres lesz.

GotoXY eljárás
Pozícionálja a kurzort.

GotoXY(*x*,*y*: **byte**);

Paraméterek: *x*,*y* a kurzor új pozíciója (oszlop, sor).
Megjegyzés: Mozgatja a kurzort az aktuális ablaknak *x*-edik oszlopába és *y*-adik sorába. Az ablak bal felső sarokpontja (1,1).
Érvénytelen koordináták esetén a *GotoXY* hívás nem kerül végrehajtásra.

HighVideo eljárás
Kiválasztja a nagyintenzitású karaktereket.

HighVideo.

Megjegyzés: Egy **byte** változó tartja nyilván a karakterek video tulajdonságát.

InsLine eljárás
Beszúr egy üres sort a kurzor pozíciónál.

InsLine;

Megjegyzés: A sorbeszúrás következtében az utolsó sor kilép a képernyőből.

KeyPressed függvény
Igaz értékkel tér vissza, ha a klaviatúrán megnyomtunk egy billentyűt, különben értéke hamis.

KeyPressed;

Megjegyzés: A ***KeyPressed*** nem érzékeli a *Shift*, az *Alt*, a *NumLock*, stb. billentyűk megnyomását.

LowVideo eljárás
Beállítja az alacsony intenzitású karaktereket.

LowVideo;

Megjegyzés: A ***LowVideo*** törli a nagy intenzitású biteket.

NormVideo
Kiválasztja a karakterek eredeti intenzitását.

NormVideo;

Nosound eljárás
Kikapcsolja a belső hangszórót.

NoSound;

Megjegyzés: A ***Sound*** eljárással bekapcsolt hangszórót kikapcsolja.

TextBackground eljárás
Kiválasztja a háttér színét.

TextBackground(*Color*: **byte**);

Paraméter: *Color* háttér színe (0-7).

A háttérszín konstansai:

```
const
    Black     = 0;    { fekete          }
    Blue      = 1;    { kék             }
    Green     = 2;    { zöld            }
    Cyan      = 3;    { türkíz          }
    Red       = 4;    { piros           }
    Magenta   = 5;    { lila            }
    Brown     = 6;    { barna           }
    LightGray = 7;    { világos szürke  }
```

TextColor eljárás
Kiválasztja az előtér (az írás) színét.

TextColor(*Color*: **byte**);

Paraméter: *Color* előtérszíne (0-15).

Megjegyzés: A színkonstansok megnevezését lásd a ***SetAllPalette*** eljárás ismertetésénél. A karaktereket villogtatni is lehet, ha a szín értékéhez hozzáadunk 128-at, vagy felhasználjuk a ***Blink*** konstanst.
Például: ***TextColor***(*Green+Blink*);

TextMode eljárás
Szöveg üzemmódot választ ki.

TextMode(*Mode*: **word**);

Paraméter: *Mode* a szöveg üzemmód típusa.

Megjegyzés: A szöveg üzemmód konstansai:

```
const
    BW40  = 0; { 40x25 B/W üzemmód színes monitoron
               (B/W black and white = fekete és fehér }
    BW80  = 2; { 80x25 B/W üzemmód színes monitoron }
    Mono  = 7; { 80x25 monokróm (egyszínű) MDA-n
                  Hercules monitoron                 }
```

```
CO40      = 1;  { 40x25 színes üzemmód színes monitorom
                  (CO Color = színes)                   }
CO80      = 3;  { 80x25 színes üzemmód színes monitoron
C40       = CO40;   { 3.0 kompatibilitás }
C80       = CO80;   { 3.0 komaptibilitás }
Font8x8   = 256;    { 40x43, 40x50 sor EGA/VGA típusú
                      monitoron }
```

WhereX függvény
Visszatér a kurzor *x* koordinátaértékével, amely az aktuális ablakhoz képest relatív távolságot jelent.

WhereX;

Megjegyzés: A függvény visszatérési értéke **byte** típusú.

WhereY függvény
Visszatér a kurzor *y* koordinátaértékével, amely az aktuális ablakhoz képest relatív távolságot jelent.

WhereY;

Megjegyzés: a függvény visszatérési értéke ***byte*** típusú.

Window eljárás
A szöveges képernyőn ablakot definiál.

Window(*x1,y1,x2,y2*: **byte**);

Paraméter: *x1,y1* az ablak bal felső sarokpontja,
x2,y2 az ablak jobb alsó sarokpontja.

Megjegyzés: Az alapértelmezés szerint az ablakok 25 és 43-as soros, 80 és 40 oszlopos módokban:

```
Window(1,1,80,25);
Window(1,1,40,25);
Window(1,1,80,43);
Window(1,1,40,43);
```

A VGA képernyőnél a 43 sor helyett 50 sor van.

A Window eljárás aktiválása után a kurzor az aktív ablak bal felső (1,1) koordinátájú pontjába kerül. A képernyőkoordináták mindig az ablak bal felső sarkához képest relatív koordinátákat jelentik.

F3. A GRAPH UNIT ELJÁRÁSAI ÉS FÜGGVÉNYEI

Arc eljárás

x,y középpontú körívet rajzol kezdő és végszög között.

Arc(*x,y*: **integer**; *Stangle,EndAngle, Radius*: **word**);

Paraméterek:	*x,y,*	a körív középpontja
	StAngle	a kezdőszög (fokban)
	EndAngle	a végszög (fokban)
	Radius	a sugár

Megjegyzés: Ha a kezdőszögnek 0 fokot és a végszögnek 360 fokot adunk, akkor az eljárás teljes kört rajzol. A szögeket az óramutató járásával ellenkező irányban kell megadni, 0 fok 3 órának, 90 fok 12 órának felel meg. A körív színe az aktuális szín.

Bar eljárás

Téglalapot rajzol, aktuális színnel és mintával befesti.

Bar(*x1,y1,x2,y2*: **integer**);

Paraméterek:	*x1,y1*	a téglalap bal felső sarka,
	x2,y2	a téglalap jobb alsó sarka.

Megjegyzés: A ***SetFillStyle*** vagy a ***SetFillPattern*** által megadott mintával és színnel festi be a téglalapot.

Bar3d eljárás

Téglalapot rajzol és az aktuális színnel és mintával befesti.

Bar3d(*x1,y1,x2,y2*: ***integer***; *Depth* : ***word***; *Top:* **boolean**);

Paraméterek:	*x1,y1*	a téglalap bal felső sarka,
	x2,y2	a téglalap jobb alsó sarka,
	Depth	a téglalap mélysége,
	Top	**true** esetén a téglatest tetejére újabb téglatest

illeszthető,
false esetén újabb téglatest nem illeszthető.

Mejegyzés: A téglatest festéséhez felhasználja a ***SetFillStyle*** vagy a ***SetFillPattern*** által megadott mintát és színt. A téglatest körvonalához a vonaltípust és a színt a ***SetLineStyle*** és a ***SetColor*** határozza meg.
Az eljárás *Top* paraméteréhez felhasználhatjuk az alábbi konstansokat;

```
const
    TopOn  = True;
    TopOff = False;
```

Circle eljárás
x,y köréppontú kört rajzol.

Circle(x,y: ***integer***; *Radius*: **word**);

Paraméterek: A kör rajzolása a ***SetColor*** által megadott színnel történik.

Cleardevice eljárás
Törli a grafikus képernyőt és előkészíti az outputra.

ClearDevice;

Megjegyzés: A ***ClearDevice*** az aktuális pointert a (0,0) pozícióba mozgatja és törli a képernyőt, a háttér színét a ***SetColor*** határozza meg.

ClearViewPort eljárás
Törli az aktuális grafikus ablakot.

ClearViewPort:

Megjegyzés: A kitöltő színnek a háttérszínt állítja be, meghívja a ***Bar*** eljárást, és az aktuális pointert a (0,0) pozícióba mozgatja.

CloseGraph eljárás
Lezárja a grafikus üzemmódot.

CloseGraph;

Megjegyzés: A **CloseGraph** a grafikus üzemmód inicializálása előtti képernyőt jeleníti meg.

DetectGraph eljárás

Ellenőrzi a hardvert és meghatározza, hogy milyen grafikus meghajtót és módot lehet használni.

DetectGraph(var *GraphDriver*, *GraphMode*: **integer**);

Megjegyzés: Visszatér az érzékelt meghajtóval és móddal, ezekkel az értékekkel kell meghívni az ***InitGraph*** eljárást, amely a korrekt meghajtót tölti be. Ha nem érzékel grafikus meghajtót, akkor a *GraphDriver* paraméter és a *GraphResult* függvény -2 értékkel tér vissza.
A következő konstansokat definiálták:

```
const
      Detect    = 0;  { automatikus érzékelés }
      CGA       = 1;
      MCGA      = 2;
      EGA       = 3;
      EGA64     = 4;
      EGAMono   = 5;
      IBM8514   = 6;
      HercMono  = 7;
      ATT400    = 8;
      VGA       = 9;
      PC3270    = 10;
```

DrawPoly eljárás
A megadott vonaltípussal és színnel pontsorozatot köt össze.

DrawPoly(*NumPoints*: **word**; **var** *PolyPoints*);

Paraméterek:	*NumPoints*	a koordináták száma,
	PolyPoints	*PoinType* típusú tömb.

Megjegyzés: A koordinátapárokat a *PolyPoints* rekord *X* és *Y* változóiban kell tárolni. Egy n pontból álló zárt poligon esetén n+1 koordinátapárt kell megadni, ahol az első koordinátapárnak meg kell egyeznie az n+1-edik koordinátapárral.

PointType típus:

```
type
         PointType = record
                   X,Y : integer;
         end;
```

Ellipse eljárás
x,y középpontú ellipszis ivet rajzol kezdő és végszög között.

Ellipse(*x,y*: **integer**; *StAngle,EndAngle*: **word**; *XRadius,YRadius*: **word**);

Paraméterek:	*x,y*	középpont kezdő koordinátái,
	StAngle	kezdőszög (fokban),
	EndAngle	végszög (fokban),
	XRadius	vízszintes tengely,
	YRadius	függőleges tengely.

Megjegyzés: A szögek az óramutató járásával ellentétes irányúak. A 0 fok 3 órának, a 90 fok 12 órának felel meg.

FillEllipse eljárás
Rajzol és befest egy ellipszist.

FillEllipse(*x,y*: **integer**; *XRadius,YRadius*: **word**);

Paraméterek:	*x,y*	az ellipszis középpontja,
	XRadius	a vízszintes tengely,
	YRadius	a függőleges tengely.

Megjegyzés: Befest egy *x,y* középpontú, *XRadius* és *YRadius* vízszintes és függőleges tengelyú ellipszist a megadott festőszínnel és mintával, a háttér is megadott színű.

FillPoly eljárás

Rajzol és befest egy poligont.

FillPoly(*Numpoints*: **word**; **var** *PolyPoints*);

Paraméterek:	*Numpoints*	a poligon pontpárainak száma,
	PolyPoints	*PointType* típusú tömb.

Megjegyzés: A koordinátapárokat a *PolyPoints* rekord *x* és *y* változóiban kell tárolni. A ***SetFillStyle*** vagy a ***SetFillPattern*** által definiált mintával és színnel festi be a poligont, körvonalát pedig a ***SetLineSTyle*** határozza meg.

FloodFill eljárás

Aktuális mintával befesti az adott színű vonallal zárt területet.

FloodFill(*x,y*: **integer**; *Border*: **word**);

Paraméterek:	*x,y*	a zárt terület egy belső pontjának koordinátái,
	Border	szín.

GetArcCoords eljárás

Megadja az utoljára rajzolt ív (***Arc, Ellipse***) kezdő- és végkoordinátáinak értékét.

GetArcCoords(**var** *ArcCoords*: ***ArcCoordsType***);

Megjegyzés: Visszatér az *ArcCoordsType* típusú rekord értékeivel.

Az *ArcCoordsType* a következő:

```
type
    ArcCoordsType = record
                        X,Y:             integer;
                        Xstart,Ystart:   integer;
                        Xend,Yend:       integer;
                    end;
```

ahol	X,Y	középpont koordinátái,
	Xstart,Ystart	az ív kezdőpontjának koordinátái,
	Xend,Yend	az ív végpontjának koordinátái.

Ezeknek az adatoknak az ismeretében lehet pl. egy ellipszis ív végpontjából egy vonalat húzni.

GetAspectRatio eljárás

Visszatér a grafikus képernyő maximális méretével, amelyből (*Xasp:Yasp*) oldalarány képződik.

GetAspectRatio(**var** *Xasp,Yasp*: **word**);

Paraméter:	*Xasp*	a maximális x érték,
	Yasp	a maximális y érték.

Megjegyzés: Minden grafikus kártyának és az aktuális grafikus módnak van oldalaránya (ezt a maximum y osztva x-el kapjuk meg). Ez az arány szükséges a ***Circle, Arc, PieSlice*** és a ***Rectangle*** kerekítésénél.

GetBkColor függvény

Visszatér a háttér akutális színével.

GetBkColor;

Megjegyzés: Az eredmény **word** típusú. A háttér szín 0-15-ig változhat, ez függ a grafikus kártyától és az aktuális grafikus módtól.

GetColor függvény
Visszatér az előtér, tehát az utolsó sikeres *SetColor* hívás színének értékével.

GetColor:

Megjegyzés: Az eredmény **word** típusú. A rajzolás színe 0-15-ig változhat, ez függ a grafikus kártyától és az aktuális grafikus módtól.

GetDefaultPalette függvény
Visszatér a paletta (színskála) értékeivel.

GetDefaultPalette(var *Pal*: *PaletteType)*;

Megjegyzés: Az eredmény *PaletteType* típusú.
A *PaletteType* rekord a következő:

```
const
        MaxColors=15;

type
        PaletteTYpe = record
                          Size: byte;
                          Colors: array[0..MAxColor] of shortint;
                      end;
```

GetDriverName függvény
A grafikus kártya nevével tér vissza.

GetDriverName;

Megjegyzés: Az eredménye **string** típusú. *InitGraph* aktiválása után az aktív grafikus kártya nevével tér vissza.

GetFillPattern eljárás
Az előző ***SetFillPattern*** hívás értékeivel tér vissza,

GetFillPattern(**var** *FillPattern*: *FillPatternType*);

Megjegyzés: Ha nem volt ***SetFillPattern*** hívás, akkor a ***GetFillPattern*** tömbjébe $FF íródik be.

GetFillSettings eljárás
Visszatér az előző SetFillPattern hívásának festőmintájával és színével.

GetFillSettings(**var** *FillInfo*: *FillSettingsType*);

Megjegyzés: A ***FillSettingsType*** deklarálva van a *Graph* unit-ban:

```
type
      FillSettingsType = record
                              Pattern: word;
                              Color  : word;
                         end;
```

GetGraphMode függvény
Visszatér az aktuális grafikus móddal.

GetGraphMode;

Megjegyzés: Az eredmény **integer** típusú. A mód értéke 0-5-ig változhat, ez függ az aktuális grafikus kártyától.

GetImage eljárás
A megadott képmezőt elmenti egy bufferbe.

GetImage(*x1*,*y1*,*x2*,*y2*: **integer**; **var** *BitMap*);

Paraméterek:	*x1*,*y1*	a téglalap bal felső sarka,
	x2,*y2*	a téglalap jobb alsó sarka,
	BitMap	bufferre mutató pointer.

Megjegyzés: Az eljárás a képernyő megadott területét elmenti a *BitMap* bufferbe. A buffer első két szava a téglalap alakú tartomány szélességét és hosszúságát tartalmazza, ezért a buffer méretének 4-el nagyobbnak kell lennie.

GetLineSettings eljárás
Visszatér a vonal típusával, mintájával és vastagságával, amelyet a ***SetLineStyle*** állított be.

GetLineSettings(**var** *LineInfo*: *LineSettingsType*);

Megjegyzés: A *LineSettingsType* rekord az alábbi:

```
type
    LineSettingsType = record
                           LineStyle  : word;
                           Pattern    : word;
                           Thickness  : word;
                       end;
const { Vonaltipus }
    SolidLn    = 0;       { normál vonal            }
    DottedLn   = 1;       { pontozott vonal         }
    CenterLn   = 2;       { középvonal              }
    DashedLn   = 3;       { szaggatott vonal        }
    UserBitLn  = 4;       { felhasználó által
                            definiált vonal         }
    { vonalszélesség }
    NormWidth     = 1;    { normál szélesség        }
    ThickWidth    = 3;    { vastag vonal            }
```

GetMaxColor függvény
Visszatér a maximálisan használható színek számával.

GetMaxColor;

Megjegyzés: Az eredmény word típusú. Például 256 Kbyte Ega esetén a ***GetMaxColor*** visszatérési értéke 15, amely azt jelenti, hogy a ***SetColor*** 0 és 15 között használhatja a színeket. A CGA finom grafika és a Hercules

monochrom adaptere esetén a ***GetMaxColor*** 1 értékkel tér vissza, ebben az esetben a szín 0 és 1 lehet.

GetMaxMode függvény
Visszatér a használható maximális mód számával.

GetMaxMode;

GetMaxX függvény
Visszatér a maximálisan használható oszlopok számával (x irány).

GetMaxX;

Megjegyzés: Az eredmény ***integer*** típusú. Például CGA 320x200 mód esetén a ***GetMaxX*** 319 értékkel tér vissza.

GetMAxY függvény
Visszatér a maximálisan használható sorok számával (y irány).

GetMaxY;

Megjegyzés: Az eredmény **integer** típusú. Például CGA 320x200 mód esetén a ***GetMaxY*** 199 értékkel tér vissza.

GetModeName függvény
String-ben adja vissza a grafikus mód nevét.

GetModeName(*ModeNumber*: **word**);

Megjegyzés: Az eredmény **string** típusú.

GetModeRange eljárás
A legkisebb és a legnagyobb grafikus móddal tér vissza.

GetModeRange(*GraphDrive*: **integer**; **var** *Lomode,HiMode*: **integer**);

Megjegyzés: Érvénytelen ***GraphDriver*** megadása esetén a visszatérési érték -1 lesz.

GetPalette eljárás

Visszatér az aktuális paletta és mérete értékeivel.

GetPalette(**var** *Palette*: *PaletteType*);

Megjegyzés: Az eredmény ***PaletteType*** típusú rekord.

```
const
        MaxColor = 16;
type
        PaletteType = record
                        Size      : byte;
                        Colors    : array[0..MaxColors] of shortint;
                      end;
```

GetPaletteSize függvény

Visszatér a paletta színtábla méretével.

GetPaletteSize;

Megjegyzés: Az eredmény **word** típusú. A visszatérési érték megadja az aktuális grafikus módban használható paletták számát. Például színes üzemmódban használt EGA esetén ez az érték 16 lesz.

GetPixel függvény

Visszatér az (x,y) képpont színértékével.

GetPixel(*x*,*y*: **integer**);

Megjegyzés: A visszatérési érték **word** típusú.

GetTextSettings eljárás

Visszatér az aktuálisan használt karakterkészlet típusával, irányával, méretével és beállításával, melyet a ***SetTextStyle*** és a ***SetTextJustify*** beállítottak.

GetTextSettings(**var** *TextInfo*: *TextSettingsType*);

Megjegyzés: Az alábbi típusokat és konstansokat definiálták:

```
type
    TextSettingsType = record
                           Font      : word;
                           Direction : word;
                           CharSize  : word;
                           Horiz     : word;
                           Vert      : word;
                       end;
const
    DefaultFont    = 0;    { alapértelmezett              }
    TriplexFont    = 1;    { háromvonalas                 }
    SmallFont      = 2;    { kisméretű                    }
    SanSerifFont   = 3;    { egyszerű                     }
    GothicFont     = 4;    { gót                          }
    HorizDir       = 0;    { vizszintes, balról jobbra    }
    VertDir        = 1;    { függőleges, alulról felfelé  }
```

GetViewSettings eljárás

Visszatér a ***SetViewPort*** által aktuálisan beállított ablak adataival.

GetViewSettings(**var** *ViewPort*: *ViewPortType*);

Megjegyzés: A ***ViewPortType*** rekord az alábbi:

```
type
    ViewPortType = record
                       x1,y1,x2,y2 : integer;
                       Clip        : boolean;
                   end;
```

Az (x1,y1) és (x2,y2) koordinátapontok az aktív ablak méretét szolgáltatják, melyek az abszolut képernyő koordinátákban értendők. A ***Clip*** **boolean** változó értéke **true** (igaz), ha vágást kérünk, egyébként **false** (hamis). A vágás esetén a rajzoknak csak az ablakba eső része látható.

GetX függvény
Visszatér a kurzor aktuális *x* koordináta értékével.

GetX;

Megjegyzés: A függvény értéke **integer** típusú. Ha a képernyőn egy ablak van kijelölve, akkor az ahhoz viszonyított koordinátákat kapjuk vissza.

GetY függvény
Visszatér a kurzor aktuális *y* koordinátaértékével.

GetY;

Megjegyzés: A függvény értéke ***integer*** típusú.

GrapDefaults eljárás
Alapértelmezés szerint állítja vissza a grafikus üzemmódot.

GraphDefaults;

Megjegyzés: A kurzort a képernyő bal felső sarkába (0,0) helyezi és alapállapotba állítja a grafikus rendszer változóit.

GrapErrorMsg függvény
Visszatér az *ErrorCode*-nak megfelelő hibaüzenettel.

GraphErrorMsg(*ErrorCode*: **integer**);

Megjegyzés: A függvény értéke **string** típusú.

GraphResult függvény

Visszatér az utoljára végrehajtott grafikus művelet hibakódjával.

GraphResult;

ImageSize függvény

Visszatér a tárolni kívánt téglalap alakú grafikus képernyő **byte**-jainak számával.

ImageSize(*x1,y1,x2,y2*: **integer**);

Paraméterek:	*x1,y1*	a téglalap bal felső sarka,
	x2,y2	a téglalap jobb alsó sarka.

Megjegyzés: A függvény értéke két **word** típusú változó, amely megadja a tartomány szélességét és magasságát. 64Kbyte-nál kisebb tartomány tárolható, ellenkező esetben 0 lesz a visszatérési érték. A ***GraphResult*** hívásakor *GrError*=-11 hibakódot kapunk.

InitGraph eljárás

Beállítja a grafikus üzemmódot.

InitGraph(**var** *GraphDriver*: **integer**; **var** *GraphMode*: **integer**,
DriverPath: **string**);

Paraméterek:	*GraphDriver*	a grafikus kártya típusa,
	GraphMode	a grafikus mód,
	DriverPath	az akutális .BGI file-t tartalmazó könyvtár (directory) neve.

Megjegyzés: *GraphDriver* := ***Detect*** megadása esetén a grafikus kártya típusa és a grafikus mód automatikusan kerül kiválasztásra. Amennyiben az aktuális .BGI file abban a könyvtárban van, amelyben dolgozunk, a *DriverPath* paraméter értéke '' (üres string) lehet.

Line eljárás

Egyenest rajzol (x1,y1) és (x2,y2) pontok között.

Line(*x1,y1,x2,y2*: **integer**);

Paraméterek: *x1,y1* az egyenes kezdőpontja,
x2,y2 az egyenes végpontja.

Megjegyzés: Egyenest rajzol a ***SetLineStyle*** általa beállított vonaltípussal és vonalvastagsággal. A vonal színét a ***SetColor*** határozza meg.

LineRel eljárás

Egyenest rajzol az aktuális pointer pozíciótól adott *x,y* távolságban lévő ponthoz.

LineRel(*Dx,Dy*: **integer**);

Paraméterek: *Dx* távolság *x* irányban,
Dy távolság *y* irányban.

Megjegyzés: Egyenest rajzol a ***SetLineStyle*** által beállított vonaltípussal és vonalvastagsággal. A vonal színét a ***SetColor*** határozza meg.

LineTo eljárás

Egyenest rajzol az aktuális pointer pozíciótól az adott pontig.

Line(*x,y*: **integer**);

Paraméterek: Egyenest rajzol a ***SetLineStyle*** által beállított vonaltípussal és vonalvastagsággal. A vonal színét a ***SetColor*** határozza meg.

MoveRel eljárás

Mozgatja az aktuális pointert az akutális helyétől az adott távolsággal.

MoveRel(*Dx,Dy*: **integer**);

Paraméterek: *Dx* távolság *x* irányban,
Dy távolság *y* irányban.

MoveTo eljárás
Mozgatja az aktuális pointert az (*x*,*y*) pontba.

MoveTo(*x*,*y*: **integer**);

Paraméterek: *x*,*y* a pointer új helyzetének pontja.

Megjegyzés: Ha van ***viewport*** aktiválva, akkor az *x*,*y* értékek hozzáadódnak a képernyőablak bal felső sarokpontjának értékéhez és ez lesz a pointer új helyzetének pontja.

OutText eljárás
Az akutális pointer helyétől szöveget ír.

OutText(*TextString*: **string**);

Paraméter: *TextString* a kiirandó szöveg.

Megjegyzés: A korábban beállított betűtípussal, valamint a kijelölt irányban (vízszintes, függőleges) szöveget ír.

OutTextXY eljárás
Szöveget ír ki az adott (*x*,*y*) pontnál.

OutTextXY(*x*,*y*: ***integer***; *TextString*: **string**);

Paraméterek: *x*,*y* a szöveg kiírásának a helye,
TextString a kiirandó szöveg.

Megjegyzés: A korábban beállított betűtípussal, valamint a kijelölt irányban (vízszintes, függőleges) szöveget ír.

PieSlice eljárás
Rajzol és befest egy (*x*,*y*) középpontú körcikket egy kezdő- és végszög között.

PieSlice(*x*,*y*: ***integer***; *StAngle*,*EndAngle*,*Radius*: **word**);

PutImage eljárás
Tartomány (képmező) ráhelyezése a képernyőre.

PutImage(*x*,*y*: **integer; var** *BitMap*; *BitBlt*: **word**);

Paraméterek:	*x*,*y*	a képernyőn a téglalap tartomány bal felső sarokpontja,
	BitMap	típusnélküli paraméter, mely tartalmazza a téglalap tartomány szélességét, magasságát, és a kép bittérképét.
	BitBlt	bináris művelet a kihelyezendő tartomány pontja és a képernyő pontjai között.

Megjegyzés: *BitBlt* az alábbi bináris műveleteket veheti fel:

```
const
      CopyPut    = 0;      { MOV    }
      XORPut     = 1;      { XOR    }
      OrPut      = 2;      { OR     }
      AndPut     = 3;      { AND    }
      NotPut     = 4;      { NOT    }
```

PutPixel eljárás
Egy pontot rajzol az *x*,*y* pontban.

PutPixel(*x*,*y*: **integer**; *Pixel*: **word**);

Paraméterek:	*x*,*y*	pont koordinátája
	Pixel	a pont színe

Rectangle eljárás
Téglalapot rajzol.

Rectangle(*x1,y1,x2,y2*: ***integer***);

Paraméterek:	*x1,y1*	a téglalap bal felső sarka,
	x2,y2	a téglalap jobb alsó sarka.

Megjegyzés: A téglalapot az aktuális színnel és vonaltípussal rajzolja.

RestoreCrtMode eljárás
A grafikus üzemmód előtti képernyőmódot állítja vissza.

RestoreCtrMode;

Megjegyzés: A grafikus és a *text* üzemmódot váltogathatjuk a ***RestoreCrtMode*** és a ***SetGraphMode*** eljárások hívásával.

Sector eljárás
Rajzol és befest egy ellipszis cikket,

Sector(*x,y*: **integer**; *StAngle,EndAngle,XRadius,YRadius*: **word**);

Paraméterek:	*x,y*	középpont koordinátái,
	StAngle	kezdőszög,
	EndAngle	végszög,
	XRadius	vízszintes tengely;
	YRadius	függőleges tengely.

SetActivePage eljárás
Új lapot nyit meg a grafikus output számára.

SetActivePage(*Page*: **word**);

Paraméter:	*Page*	a lap száma.

Megjegyzés: Több lap használatát csak az EGA (256KByte), a VGA és a Hercules grafikus kártya teszi lehetővé. A ***SetVisualPage*** eljárás segítségével váltogatjuk a látható lapokat, ez segítséget nyújt az animáció számára.

SetAllPalette eljárás

Változtatja a paletta színeit.

SetAllPAlette(**var** *Palette*);

Paraméter: *Palette* típus nélküli paraméter.

Megjegyzés: A Palette első **byte**-ja a paletta hossza. A következő n **byte** szín fogja felülírni a paletta színeit. A szín tartománya -1 és 15 között változhat, -1 esetén a konstansok tartoznak a palettához:

```
const
        Black          = 0;      { fekete          }
        Blue           = 1;      { kék             }
        Green          = 2;      { zöld            }
        Cyan           = 3;      { türkiz          }
        Red            = 4;      { piros           }
        Magenta        = 5;      { lila            }
        Brown          = 6;      { barna           }
        LightBrown     = 7;      { világos barna   }
        DarkGray       = 8;      { sötét barna     }
        LightBlue      = 9;      { világos kék     }
        LightGreen     = 10;     { világos zöld    }
        LightCyan      = 11;     { világos türkiz  }
        LightRed       = 12;     { világos piros   }
        LightMagenta   = 13;     { világos lila    }
        Yellow         = 14;     { sárga           }
        White          = 15;     { fehér           }
        MaxColors      = 15;     { a színek maximális száma }
Type
        PaletteType =record
                        Size: byte;
                        Colors: array[0..MaxColors] of shortint;
                     end;
```

SetAspectRatio eljárás
Változtatja a beépített oldalarányt.

SetAspectRatio(*Xasp,Yasp*: **word**);

Paraméterek: *Xasp,Yasp* oldalarányok.

Megjegyzés: Amennyiben a beépített oldalarányok felhasználásával a kör rajza torzult, a hibát softver úton kiküszöbölhetjük, ha az oldalarányokat változtatjuk.

SetBkColor eljárás
Beállítja a háttér színét.

SetBkColor(*Color*: **word**);

Paraméter: *Color* indexe a színt tartalmazó *Colors* tömbnek. Az index 0-15 között változhat.

SetColor eljárás
Beállítja a rajzolás színét.

SetColor(*Color*: **word**);

Paraméter: *Color* indexe a színt tartalmazó *Colors* tömbnek. Az index 0-15 között változhat.

SetFillPattern eljárás
Kiválasztja a felhasználó által definiált mintát.

SetFillPattern(*Pattern*: *FillPatternType*; *Color*: **word**);

Paraméterek: *Pattern* 8 **byte** hosszú minta megadása hexadecimálisan.
Color szín.

Megjegyzés: A ***FillPatternType*** a következő:

```
type
    FillPatternType = array[1..8] of byte.
```

SetFillStyle eljárás

Beállítja a festőmintát és a színt.

SetFillStyle(*Pattern*: **word**; *Color*: **word**);

Paraméterek: *Pattern* minta típus,
Color szín.

Megjegyzés: A kiválasztott színnel és mintával lehet festeni a ***FillPoly***, ***Bar***, ***Bar3D*** és ***PieSlice*** rutinok hívásával poligont, téglalapot, téglatestet és körcikket.

Az alábbi konstansokat használhatjuk fel:

```
const
    EmptyFill      = 0;    { háttérszínnel fest              }
    SolidFill      = 1;    { egyenletes, gyenge tónus        }
    LineFill       = 2;    { ___ vízszintes vonalas minta    }
    LtSlashFill    = 3;    { dőlt /// vonalas minta          }
    SlashFill      = 4;    { dőlt /// vastag vonalas minta   }
    BkSlashFill    = 5;    { dőlt \\\ vastag vonalas minta   }
    LtBkSlashFill  = 6;    { dőlt \\\ vonalas minta          }
    HatchFill      = 7;    { kockás minta                    }
    XHatchFill     = 8;    { dőlt kockás minta               }
    InterleaveFill = 9;    { sűrűn pontozott                 }
    WideDotFill    = 10;   { ritkán pontozott                }
    CloseDotFill   = 11;   { közepesen pontozott             }
    USerFill       = 12;   { felhasználó által definiált     }
```

SetGraphBufSize eljárás

Felülírja a poligon kitöltéséhez használt belső buffer méretét.

SetGraphBufSize(*BufSize*: word);

Paraméter: *BufSize* a buffer mérete.

Megjegyzés: 4K a beépített buffer mérete, amely elég nagy ahhoz, hogy 650 töréspontú poligont befessen. Ha ennél több töréspontú poligont akarunk befesteni, akkor szükséges a buffer méretét növelni, hogy elkerüljük a buffer túlcsordulását.

SetGraphMode eljárás

Beállítja a grafikus módot és törli a képernyőt.

SetGraphMode(*Mode*: **integer**);

Paraméter: *Mode* a beépített grafikus kártya érvényes módja.

SetLineStyle eljárás

Beállítja a vonal típusát és vastagságát.

SetLineStyle(*LineStyle*: **word**; *Pattern*: **word**;
Thickness: **word**);

Paraméter: *LineStyle* vonaltípus,
Pattern vonaltípus minta,
Thickness vonalvastagság.

Megjegyzés: A következő konstansokat használhatjuk fel a vonalvastagsághoz:

```
const
    SolidLn    = 0;     { teljes vonal    }
    Dottedln   = 1;     { pontozott       }
    CenterLn   = 2;     { középvonal      }
    DashedLn   = 3;     { szaggatott      }
    UserBitLn  = 4;     { bitmintával megadott }
    NormWidth= 1;       { normál vastagságú vonal }
    ThickWidth= 3;      { vastag vonal    }
```

A ***LineStyle*** értéke a ***SolidLn*** és a ***UserBitLn*** (0-4) között változhat, a *Pattern* értéke 0, hacsak nem a *UserBitLn*-t adtuk meg. Ebben az esetben a felhasználó által megadott minta lesz a vonaltípus. 16 bites hexadecimális

szám lesz a *Pattern* értéke. Például $AAAA. A vonalszélesség normál (*NormWidth*) vagy vastag (*ThickWidth*) lehet.

SetPalette eljárás

Egy paletta színt változtat.

SetPalette(*Colornum*: **word**; *Color*: **word**);

Paraméterek:	*Colornum*	szín sorszáma a táblázatban,
	Color	szín.

Megjegyzés: Ha a *Colornum* 0 és a *Color Green*, akkor az első elem zöld lesz. A beépített színkonstansok sorszámát lásd a ***SetAllPalette*** eljárásnál.

SetTextJustify eljárás

Szöveg helyzetének beállítása az ***OutText*** és az ***OutTextXY*** eljárások számára.

SetTextJustify(*Horiz*,*Vert*: word);

Paraméterek:	*Horiz*	vízszintes beállítás,
	Vert	függőleges beállítás.

Megjegyzés: A következő konstansokat használhatjuk fel:

```
const
      { vízszintes beállítás }
      LeftText   = 0;      { balra    }
      CenterText = 1;      { középre  }
      RightText  = 2;      { jobbra   }

      { függőleges beállítás }
      BottomText= 0;       { aljára   }
      CenterText = 1;      { középre  }
      TopText    = 2;      { tetejére }
```

SetTextStyle eljárás

Beállítja az aktuális karakterkészlet típusát és méretét.

SetTextStyle(*Font*: **word**; *Direction*: **word**; *CharSize*: **word**);

Paraméterek:	*Font*	karakterkészlet neve,
	Direction	az írás iránya,
	CharSize	a karakterek mérete.

Megjegyzés: Az alábbi típusok és konstansok használhatók fel a paraméterek megadásához.

```
const
     DefaultFont     = 0;      { alapértelmezett }
     TriplexFont     = 1;      { háromvonalas    }
     SmallFont       = 2;      { kisbetű         }
     SanSerifFont    = 3;      { egyszerű        }
     GothicFont      = 4;      { gót betű        }

     HorizDir        = 0;      { balról jobbra   }
     VertDir         = 1;      { alulról felfelé }
```

SetUSerCharSize eljárás

A karakter szélességének és magasságának változtatása.

SetUserCharSize(*MultX*,*DivX*,*MultY*,*DivY*: **word**);

Paraméterek:	*MultX*,*DivX*	*MultX*:*DivX* arány szorzódik a normál szélességgel,
	MultY,*DivY*	*MultY*:*DivY* arány szorzódik a normál magassággal.

SetViewPort eljárás

Ablakot jelöl ki a grafikus outputon.

SetViewPort(*x1*,*y1*,*x2*,*y2*: **integer**; *Clip*: **boolean**);

Paraméterek:	*x1,y1*	az ablak bal felső sarka,
	x2,y2	az ablak jobb alsó sarka,
	Clip	*ClipOn* (**true**) esetén a kivágást bekapcsolja, *ClipOff* (**false**) esetén kikapcsolja.

Megjegyzés: Továbbiakban minden koordinátapont az adott ablakhoz lesz viszonyítva. A *Clip* paraméter határozza meg, hogy az ablakból kinyúló vonalak látszanak-e.

SetVisualPage eljárás

Láthatóvá teszi az adott grafikus lapot.

SetVisualPage(*Page*: **word**);

Paraméter: *Page* a lap száma.

Megjegyzés: Több lap használata csak EGA (256Kbyte), VGA és Hercules grafikus kártya esetén lehetséges.

SetWRiteMode eljárás

Beállítja az írásmódot a vonalrajzolás számára.

SetWriteMode(*WriteMode*: **integer**);

Paraméter: *WriteMode* kétfajta lehet, az alábbi konstansok közül választhatunk.

```
const
      CopyPut   = 0;     { MOV     }
      XORPut    = 1;     { XOR     }
```

Megjegyzés: A *CopyPut* a MOV assembler utasítást használja, a vonal felülírja a képernyőt. Az *XORPut* az XOR parancsot hajt végre a vonal pontjai és a képernyő pontjai között. Két egymásutáni következő XOR parancs a vonalat letörli és a képernyőn az eredeti kép marad meg.

TextHeight függvény
Megadja a szöveg képpontokban mért magasságát.

TextHeight(*TextString*: ***string***);

Paraméterek: *TextString*: szöveg.

Megjegyzés: A függvény visszatérési értéke **word** típusú.

TextWidth eljárás
Megadja a szöveg képpontokban mért szélességét.

TextWidth(*TextString*: **string**);

Paraméterek: *TextString*: szöveg.

Megjegyzés: A függvény visszatérési értéke **word** típusú.

F4. A TURBO PASCAL FORDÍTÓ DIREKTÍVÁI

A Turbo Pascal fordítási lehetőségeinek egy része direktívák felhasználásával vezérelhető. A fordító direktíva, mint speciális megjegyzés jelenik meg a Pascal programban, és minden olyan helyre elhelyezhető, ahol utasítás állhat. A fordító direktívában a megjegyzés nyitó jelét közvetlenül a $ jel követi, amelyet szintén szóköz nélkül követ a direktíva neve. A direktívák az alábbi három szempont alapján csoportosíthatók:

Kapcsoló direktívák. Ezek a direktívák a fordító bizonyos lehetőségeit kapcsolják be, ill. ki a direktíva nevét közvetlenül követő + ill. - jelek megadásával.

Paraméter direktívák. Ezen direktívák felhasználásával a fordító számára tudunk bizonyos paramétereket definiálni, mint pl. file-név, memóriaméret.

Feltételes direktívák. A feltételes direktívák az ún. feltételes fordítást vezérlik, vagyis a felhasználó által definiált feltételes szimbólumoktól függően dől el, hogy egyes programrészek lefordítódnak, vagy nem kerülnek fordításra.

A kapcsoló direktívák kivételével a direktíva nevét és a paramétert legalább egy szóköznek kell elválasztania. Nézzünk példákat a direktívák használatára:

```
{$B+}
{$R- a tartományellenõrzés kikapcsolása}
{$I TIPUSOK.PAS}
{$O Editor}
{$M 32000,10000,655360}
{$DEFINE Debug}
{$IFEF Debug}
{$ENDIF}
```

A fordítási direktívákat elhelyezhetjük a forrásprogramban - ez a javasolt megoldás, hisz ezen direktívák nagy része programfüggő. De megadhatjuk a direktívákat a Turbo Pascal integrált fejlesztői környezetén (TURBO.EXE) belül az *Options-Compiler* menüben interaktív módon, ill. a parancssor fordító

(TPC.EXE) számára */direktíva* alakban vagy a TPC.CFG konfigurációs file-ban.

F4.1 A kapcsoló direktívák

A kapcsoló direktívák lehetnek globálisak és lokálisak egyaránt. A globális direktíva érvényes a fordítás teljes menetében, míg a lokális direktíva csak a megjelölt programrészlet fordításakor fejti ki hatását. A globális direktívákat a program vagy **unit** deklarációs része előtt kell elhelyezni, míg a lokális direktívák a programban vagy **unit**-ban bárhol lehetnek.

Megjegyzésben vesszővel elválaszva több kapcsoló is használható:

{$B+,R-,S-}

$A - Adatok tárolása (5.0)

Alapértelmezés: {$A+}
Típus : globális

A Turbo Pascal változóinak és típusos konstansainak memóriába való elhelyezését szabályozza az **$A** kapcsoló. A 80x86 CPU-k adatelérése gyorsabb, ha az adatot szóhatárra igazítva tároljuk.
Kivétel ez alól a 8088μP , ahol a fordító nem veszi figyelembe ezt a kapcsolót

{$A+} állapot:

Ebben az állapotban minden 1 byte-nál hoszabb változó és típusos konstans szóhatárra (páros memóriacímre) igazítva kerül tárolásra. (Ha szükséges, a változók közé nem használt byte-okat helyez a fordító.) A {$A+} nincs hatással a rekordmezőknek és a tömbelemeknek a tárolására.

{$A-} állapot:

A {$A-} állapotban a változók és a típusos konstansok egyszerűen a következő szabad memóriacímtől kezdve tárolódnak.

$B - Logikai kifejezések kiértékelése (5.0)

Alapértelmezés: {$B-}
Típus : lokális

A kapcsoló felhasználásával megválaszthatjuk, hogy az **AND** és **OR** operátorokat tartalmazó kifejezésekből milyen módon generálódjék a futtatható programkód.

{$B+} állapot:

Bekapcsolt állapotban {$B+}, a logikai kifejezéseket teljes egészében kiértékeli a program.

{$B-} állapot:

A {$B-} állapotban a fordító csak a kifejezés azon részéből generál kódot, amelyik feltétlenül szükséges a kifejezés logikai értékének a meghatározásához. Ez azt jelenti, hogy a kiértékelés a teljes kiértékelés előtt befejeződik, ha az eredmény már ismert.

$D - Nyomkövetési információk (5.0)

Alapértelmezés: {$D+}
Típus : globális

A nyomkövetéshez szükséges (*debug*) információk generálását tudjuk a **$D** kapcsolóval vezérelni. Bekapcsolt {$D+} állapotban a fordítás során egy információs tábla épül fel, amely tartalmazza a hibakereséshez,

nyomkövetéshez ill. a hibaforrás file-beli helyének meghatározásához szükséges adatokat.

$E - 80x87 emuláció (5.0)

Alapértelmezés: {$E+}
Típus : globális

Az $E kapcsolóval előírhatjuk a fordító számára a 8087 emulátor *run-time* könyvtár beépítését a programunkba. Ez a könyvtár, ha nincs a gépünkben 80x87 aritmetikai segédprocesszor, képes emulálni azt. A **$E** kapcsolót általában a **$N** kapcsolóval együtt használjuk.

{$N+,E+} állapot:

Ebben az állapotban a Turbo Pascal a teljes emulátor könyvtárat beszerkeszti a programunkba. Az így keletkező .EXE file indításkor ellenőrzi, hogy van-e 80x87 segédprocesszor a gépben, ha van azt használja, de ha nincs akkor az emulátor rutinokat használja.

{$N+,E-} állapot:

Ebben az állapotban lényegesen kisebb lebegőpontos könyvtár szerkesztődik a programunkhoz. A program azonban csak 80x87 segédprocesszort tartalmazó gépeken futtatható.

Az **$E** kapcsoló nincs hatással a **unit**-ok fordítására, csak a program-fordításakor veszi figyelembe a fordító.

{$N-,E+} állapot:

Ha unit-okat és a programot is az {$N-} kapcsolóállás mellett fordítottuk le, akkor az emulátor könyvtár nem szükséges a program futtatásához, így az emulátor kapcsolót figyelmen kívül hagyja a fordító.

$F - FAR módú rutinhívások (5.0)

Alapértelmezés: {$F-}
Típus : lokális

{$F+} állapot:

Az {$F+} kapcsolót követő eljárások és függvények mindig far (távoli - szegmensek közötti) módon kerülnek meghívásra.
{$F-} állapot:

Az {$F-} állapotban a Turbo Pascal automatikusan kiválasztja a megfelelő hívási módot:

FAR (távoli), ha az eljárást vagy függvényt unit **interface** részében deklaráltuk.

NEAR (közeli), ha az eljárás vagy függvény nem egy unit **interface** részében került deklarálásra.

Overlay modulokat tartalmazó programok esetén ajánlott minden unit és a program elejére helyezni a {$F+} kapcsolót. Eljárás típusú változókat használó programoknál a változóhoz tartozó eljárásokat szintén a {$F+} kapcsolóval kell lefordítanunk.

$G - Kódgenerálás a 80286 CPU számára (6.0)

Alapértelmezés: {$G-}
Típus : lokális

{$G-} állapot:

Ebben az állapotban generált 8086-os utasítások bármely 80x86 mikroprocesszoron futtathatók.

{$G+} állapot:

A {$G+} állapotban a fordító olyan processzor utasításokat is generál, amelyeket csak 80286 és a későbbi mikroprocesszorok ismernek. Ezekkel az utasításokkal általában gyorsabb kód állítható elő.

$I - Az input/output műveletek ellenőrzése (5.0)

Alapértelmezés: {$I+}
Típus : lokális

A $I kapcsolóval vezérelhetjük az I/O eljárások meghívását követő ellenőrző kódrészlet beépítését a programunkba. Ha a program futása során I/O hiba lép fel, ennek lekezelését rábízhatjuk a fordító által generált kódra a {$I+} kapcsolóval, vagy a hibakezelést saját magunk is elvégezhetjük. Ha a {$I-} direktívát használjuk (az ellenőrzés kikapcsolása), akkor az ***IOResult*** függvény meghívásával kaphatunk információt a I/O műveletek sikerességéről.

$L - Lokális szimbólumok információi (5.0)

Alapértelmezés: {$L+}
Típus : globális

Bekapcsolt állapotban {$L+} engedélyezzük a lokális szimbólumokra vonatkozó nyomkövetési információk generálását, ill. felhasználását a nyomkövetés és hibakeresés során.

A **unit**-okban a lokális szimbólumok információit a .TPU file tartalmazza, megnövelve ezzel a file méretét. A {$L+} kapcsoló használata esetén a program fordításához szükséges memória mérete is megnő. A $L kapcsolót csak akkor veszi figyelembe a fordító, ha a {$D+} direktívát is megadtuk.

$N - Lebegőpontos kódgenerálás (5.0)

Alapértelmezés: {$N-}
Típus : globális

A $N kapcsoló segítségével a Turbo Pascal különböző lebegőpontos számítási modelljei közül választhatunk.

{$N-} állapot:

Az {$N-} állapotban a Turbo Pascal olyan kódot generál, amelyben a valós típusú számításokat run-time könyvtári rutinok végzik el. Ebben a módban csak a Real lebegőpontos típus használható.

{$N+} állapot:

Bekapcsolt {$N+} állapotban minden valós típusú számításhoz a 8087 aritmetikai társprocesszor számára generál kódokat a fordító. Ha nincs a gépünkben 80x87 processzor akkor emuláltatnunk kell annak működését a {$E+} kapcsoló megadásával.

$O Overlay kódgenerálás (5.0)

Alapértelmezés: {$O-}
Típus : globális

Overlay-struktúra kialakítását engedélyezhetjük {$O+}, vagy tilthatjuk {$O-}. Az {$O+} kapcsolót általában együtt használjuk a {$F+} kapcsolóval, hisz az overlay-kezelő mindig távoli (far) hívásokat használ.

$R - Értéktartomány ellenőrzése (5.0)

Alapértelmezés: {$R-}
Típus : lokális

Az értéktartomány-ellenőrzéséhez szükséges kódrészlet generálását engedélyezhetjük vagy tilthatjuk a fordító számára.

{$R+} állapot:

Bekapcsolt {$R+} állapotban végbemenő ellenőrzések:

- tömb és sztring indexhatárainak ellenőrzése,
- sorszámozott típusra vonatkozó értékadás esetén az értéktartomány ellenőrzése,
- virtuális metódus hívásakor a VMT meglétének ellenőrzése.

Ha az ellenőrzés hibát észlel, a program *run-time* hibaüzenettel leáll.

$S - Verem túlcsordulásának ellenőrzése (5.0)

Alapértelmezés: {$S+}
Típus : lokális

Engedélyezhetjük vagy tilthatjuk a verem túlcsordulásának ellenőrzésére szolgáló kódrészlet generálását

{$S+} állapot:

Bekapcsolt {$S+} állapotban a fordító minden eljárás és függvény elején kódot helyez el, amely ellenőrzi, hogy elegendő veremterület áll-e rendelkezésre a lokális változók és más ideiglenes tárterületek létrehozására. Ha nem elég a terület a stack-en, akkor rutinhíváskor a program futása *run-time* hibaüzenettel megszakad.

Az {$S-} állapotban, ha nincs elegendő hely a veremben, a rendszer összeomolhat.

$V - Sztring típusú változó paraméterek ellenőrzése (5.0)

Alapértelmezés: {$V+}
Típus : lokális

Bekapcsolt {$V+} állapotban a fordító ellenőrzi, hogy a sztring típusú formális és aktuális változó (**var**) paraméterek típusa megegyezik-e.

Kikapcsolt {$V-} állapotban tetszőleges hosszú sztring típusú aktuális paraméter átadható, még akkor is, ha a formális paraméter maximális hossza nem egyezik meg az aktuális paraméter hosszával.

$X - Kibővített Turbo Pascal szintaxis (6.0)

Alapértelmezés: {$X-}
Típus : globális

Az $X kapcsoló segítségével engedélyezhetjük, vagy tilthatjuk a fordító számára a kibővített Turbo Pascal szintaxis használatát.

Alaphelyzetben a kapcsoló kikapcsolt állapotban van, így a kibővített szintaxis használata fordítási hibához vezet.

{$X+} állapot:

Ebben az állapotban adott a lehetőség a függvények utasításként történő (eljárás-szerű) meghívására. Ekkor a függvények visszatérési értékét a rendszer nem dolgozza fel. A {$X+} direktíva nem érvényes a beépített függvényekre (függvényekre a SYSTEM unit-ban).

F4.2. Paraméter direktívák

A paraméter direktívák használatakor a direktíva nevét és a paraméterét legalább egy szóköznek kell elválasztania. Pl.:

{$I TYPES.INC}
{$O MOONUNIT.TPU}

{$I File-név} - Forrásállomány beépítése (5.0)

Típus : lokális

A fordító a megadott forrásállomány tartalmát beépíti a direktíva helyére. Az alapértelmezés szerinti kiterjesztés .PAS. Az állományok maximálisan 15 szint mélységig ágyazhatók egymásba.

Az *include* file nem képezheti utasításnak a részét, ezért a blokkutasítást (**begin** ... **end**) is teljes egészében tartalmaznia kell a file-nak.

{$L File-név} - Tárgykódú (object) file beszerkesztése (5.0)

Tipus : lokális

A direktíva utasítja a fordítót, hogy az adott nevű tárgykódú file-t összeszerkessze az éppen fordított programmal vagy unit-tal. Ezt a direktívát assembly nyelven megírt **external** alprogramok kódba való beépítésére használjuk. A megadott file ún. Intel relokálható tárgykódú file (object file, .OBJ file) kell legyen. Az alapértelmezés szerinti kiterjesztés .OBJ.

{$M veremméret, heapmin, heapmax} - A program memóriafoglalása (5.0)

Alapértelmezés: {$M 16384,0,655360}
Típus : globális

A direktíva program indításakor lezajló memóriafoglalást definiálja. Segítségével beállíthatjuk a verem méretét (1024..65520), a heap méretének alsó határát (0..655360) és a heap mértének felső határát (heapmin..655360). A direktíva unit fordításakor nem fejti ki hatását.

{$O Unit-név} - Overlay-struktúra kialakítása (5.0)

Típus : lokális

A direktíva hatására a fordító overlay-file-ba (.OVR) szervezi az így megadott unit-okat, amelyeket előzőleg a {$O+} kapcsoló direktívával fordítottunk le. A {$O unit-név} dírektívákat közvetlenül a program **uses** utasítása után kell elhelyeznünk. A direktíva nincs hatással unit-ok fordítására.

F4.3. Feltételes fordítás

A Turbo Pascal feltételes fordítási direktívái lehetővé teszik, hogy egy forrás file-ból a feltételes szimbólumok használatával különböző kódokat generálhassunk. Ehhez a Turbo Pascal rendelkezik néhány, az **if** utasításra hasonlító szerkezettel:

{$IFxxx} .I.. {$ENDIF}

vagy

{$IFxxx} .I.. {$ELSE} .H.. {ENDIF}

Az I. részben elhelyezkedő programsorokat akkor fordítja a fordító, ha a feltétel igaz, míg a H. részt hamis feltétel mellett dolgozza fel. Nézzünk néhány példát a feltételes fordítási szerkezetekre:

```
{Ha a Debug szimbólum definiált, kiirja az a változó
 értékét}
{$IFDEF Debug}
writeln('a=',a);
{$ENDIF}
```

```
{Ha van 8087 a rendszerben a real helyett double típust
használunk, ha nincs, minden valós típus real lesz. }
{$IFDEF CPU87}
{$N+}
type
real = double;
{$ELSE}
{$N-}
type
single = real;
double = real;
extended = real;
{$ENDIF}
```

F4.3.1 Feltételes szimbólumok

A feltételes fordítás vezérlése a feltételes szimbólumok felhasználásával történik. Feltételes szimbólumokat a következő módszerek valamelyikével definiálhatunk:

- használva a {**$DEFINE** szimbólum} és a {**$UNDEF** szimbólum} direktívákat,

- a parancssor fordítónál a **/D** kapcsolóval megadva,

- az IDE fordítónál az *Options/Compiler* menü *Conditional Defines* input sorába beírva.

A feltételes szimbólumokra is a Pascal azonosítókra használt megkötések érvényesek.

A Turbo Pascal az alábbi szabványos feltételes szimbólumokat definiálja:

VER50
VER55
VER60 - mindig definiált, jelzi a fordító verziószámát

MSDOS - mindig definiált, jelzi hogy az operációs rendszer **MS-DOS** vagy PC-DOC

CPU86 - mindig definiált, jelzi hogy a processzor 80x86
CPU87 - definiált, ha a fordítási időben van a gépben 80x87 társ-processzor

F4.3.2. Feltételes direktívák

{$DEFINE név}

Feltételes szimbólum definiálása megadott névvel. Ha az adott névvel már létezik szimbólum, nincs hatása a direktívának.

{$UNDEF név}

Előzőleg definiált szimbólumot definiálatlanná tesz. Ha a szimbólum még nem definiált, nincs hatása.

{$IFDEF név}

A direktíva után álló forrássorokat lefordítja a fordító, ha az adott nevű szimbólumot előzőleg definiáltuk.

{$IFNDEF név}

A direktíva után álló forrássorokat lefordítja a fordító, ha az adott nevű szimbólumot előzőleg még nem definiáltuk.

{$IFOPT kapcsoló}

Ha az adott kapcsoló a megadott állapotban van, a fordító elvégzi a direktívát követő sorok fordítását.

Például akkor lesz **real** típusból **extended,** ha az $N kapcsoló aktív:

```
{$IFOPT N+}
    type real=extended;
{$ENDIF}
```

{$ELSE}

Az **{$INDEF}**, **{$IFNDEF}**, **{$IFOPT}** és a **{$ENDIF}** között használható a hamis ág kijelölésére.

{$ENDIF}

Az utolsó {**$IF**xxx} feltételes direktívához tartozó programrészlet végét jelöli.

F5. ÖSSZEFOGALÓ TÁBLÁZATOK

F5.1. A Turbo Pascal futás közbeni hibaüzenetei

Ha a már lefordított program futása közben a számítógép valamilyen hibát észlel, illetve ha valami miatt a program valamelyik utasítása nem végrehajtható (pl egy *Reset* eljárással megnyitni kívánt file nincs a lemezen), akkor a program futása megszakad, és a képernyőn a következő üzenet látható:

```
Runtime error nnn at ssss:oooo
```

ahol *nnn* a futási hiba kódját mutató háromjegyű decimális szám, *ssss:oooo* pedig annak a programkódnak a címe hexadecimális alakban, ahol a hiba történt. A cím *szegmenscím:offsetcím* formátumú. A futási hibák egy része a fordító direktívák beállításával és az *IoResult* beépített függvény használatával feloldhatók (ezek a Dos, az I/O hibák és a kritikus hibák egy része).

Az alábbiak tartalmazzák az egyes kódokhoz tartozó hibaüzeneteket:

Dos hibák:

002 A file, amelyre hivatkoztunk, nem létezik (Append, Erase, Rename vagy Reset eljárások használatánál)
003 Az adott elérési útvonal vagy az adott könyvtár nem létezik
004 A megengedettnél több a megnyitott file. A megnyitott file-k száma legfeljebb 15 lehet. (Érdemes odafigyelni, ugyanis ha a programot a Turbo Pascal 6.0 keretből futtatjuk, akkor a 15 file-ból a keretrendszer hármat lefoglal)
005 A file-hoz nem tudunk hozzáférni. Ennek oka lehet, hogy a file írásvédett, olvasásra nem lehet megnyitni, betelt a katalógus stb...
006 A file kezelése helytelen
012 A file hozzáférési kódja helytelen (Append vagy Reset eljárások használatánál)

015 A GetDir utasításban megadott egységszám helytelen
016 Az RmDir eljárás nem tudja törölni az adott könyvtárat.
017 A Rename eljárás nem tudja átnevezni az adott file-t

I/O hibák

100 Hiba a lemez olvasásánál a Read eljárás használatakor
101 Hiba a lemez írásánál a Close, Flush, Write, WriteLn eljárás használatakor
102 Hiányzik a fizikai file hozzárendelés az Append, Erase, Rename, Reset eljárások előtt.
103 A file nincs megnyitva.
104 A file nincs olvasásra megnyitva
105 A file nincs írásra megnyitva
106 Helytelen numerikus formátum

Kritikus hibák

150 A lemez írásvédett
151 Ismeretlen unit
152 A meghajtó üzemképtelen
153 Ismeretlen utasítás
154 Adathiba
155 Helytelen hosszúságú struktúra
156 Helytelen a filemutató pozícionálása a lemezen
157 Ismeretlen típusú adathordozó
158 Az adott szektor nem található
159 A papír elfogyott a nyomtatóból
160 Periféria írási hiba
161 Periféria olvasási hiba
162 Hardware hiba

Súlyos hibák

200 Nullával való osztás
201 Index vagy értelmezési tartomány túlcsordulás
202 Stack túlcsordulás (betelt a Stack)

203 Heap túlcsordulás
204 Helytelen pointeres művelet Dispose vagy FreeMem eljárások használatánál
205 Túlcsordulás lebegőpontos műveleteknél
206 Alulcsordulás lebegőpontos műveleteknél. (Ez a hibaüzenet csak aritmetikai processzor használata esetén fordulhat elő)
207 Helytelen lebegőpontos művelet
208 Az overlay kezelő installálása nem történt meg az OvrInit eljárással
209 Overlay file olvasási hiba

F5.2. A funkcióbillentyűk visszatérési kódjai

3	Ctrl-@	66	F8	106	Alt-F3
15	Shift-Tab	67	F9	107	Alt-F4
16	Alt-Q	68	F10	108	Alt-F5
17	Alt-W	71	Home	109	Alt-F6
18	Alt-E	72	↑	110	Alt-F7
19	Alt-R	73	PgUp	111	Alt-F8
20	Alt-T	75	←	112	Alt-F9
21	Alt-Y	77	→	113	Alt-F10
22	Alt-U	79	End	114	Ctrl-PrintScreen
23	Alt-I	80	↓	115	Ctrl-←
24	Alt-O	81	PgDn	116	Ctrl-→
25	Alt-P	82	Ins	117	Ctrl-End
30	Alt-A	83	Del	118	Ctrl-PgDn
31	Alt-S	84	Shift-F1	119	Ctrl-Home
32	Alt-D	85	Shift-F2	120	Alt-1
33	Alt-F	86	Shift-F3	121	Alt-2
34	Alt-G	87	Shift-F4	122	Alt-3
35	Alt-H	88	Shift-F5	123	Alt-4
36	Alt-J	89	Shift-F6	124	Alt-5
37	Alt-K	90	Shift-F7	125	Alt-6
38	Alt-L	91	Shift-F8	126	Alt-7
44	Alt-Z	92	Shift-F9	127	Alt-8
45	Alt-X	93	Shift-F10	128	Alt-9
46	Alt-C	94	Ctrl-F1	129	Alt-0
47	Alt-V	95	Ctrl-F2	130	Alt--
48	Alt-B	96	Ctrl-F3	131	Alt-=
49	Alt-N	97	Ctrl-F4	132	Ctrl-PgUp
50	Alt-M	98	Ctrl-F5	133	F11
59	F1	99	Ctrl-F6	134	F12
60	F2	100	Ctrl-F7	135	Shift-F11
61	F3	101	Ctrl-F8	136	Shift-F12
62	F4	102	Ctrl-F9	137	Ctrl-F11
63	F5	103	Ctrl-F10	138	Ctrl-F12
64	F6	104	Alt-F1	139	Alt-F11
65	F7	105	Alt-F2	140	Alt-F12

F5.1. Táblázat: Funkcióbillentyűk visszatérési kódjai

A Turbo Pascalban bizonyos billentyűk illetve billentyűkombinációk két karakterrel térnek vissza. Ezeknél a billentyűknél az első karakter mindig 0.

Ez azt jelenti hogy ha a ***ReadKey*** függvény visszatérési értéke 0, akkor funkció billentyű lett leütve, és egy újabb ***ReadKey*** függvény hívásával kapjuk meg a kódját.

Az F5.1. táblázat tartalmazza a fent említett billentyűk illetve billentyű-kombinációk második karakterét.

Az alábbi program tetszőleges billentyűkódok lekérdezését valósítja meg:

```
program kodok;
uses crt;
var  ch:char;
begin
     repeat
          repeat until keypressed;
            ch:=readkey;
            if ch=#0 then ch:=readkey;
            writeln(ord(ch));
     until ch=#32;
end.
```

F5.3. IBM karakterkódok táblázata

A táblázatból hexadecimális formában lehet leolvasni a karakterkódokat. A bal oldali oszlopban található a kód első jegye, míg a felső sorból a második jegy olvasható le. Például a 'A' betű kódja $41.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
1	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
F	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	

IRODALOMJEGYZÉK

1. K. Jensen - N. Wirth
Programozási nyelv
Felhasználói kézikönyv és a nyelv formális leírása
Műszaki Könyvkiadó, Budapest, 1988.

2. Andrew C. Staugaard, JR.
Technical Pascal Using TURBO
Prentice Hall International, Inc., 1990.

3. Scott D. Palmer
Mastering Turbo Pascal 6
Sybex, 1991.

4. Stephen K. O'Brien
Turbo Pascal 5.5: The Complete Reference
Borland - Osborne/McGraw-Hill, 1990.

5. Doug Cooper - Michael Clancy
Oh! Pascal!
W.W. Norton & Company, Inc. 1985.

6. Gordon Erzsébet - Körvélyesi Gézáné - Sós István - Székely Zoltán
Pascal programozási nyelv
Számítástechnikai - Alkalmazási Vállalat, Budapest, 1982.

7. Benkő Tiborné - Benkő László - Kiss Zoltán - Tóth Bertalan
Objektum-Orientált Programozás Turbo Pascal 6.0-ban
Turbo Vision
ComputerBooks, Budapest, 1991.

8. Turbo Pascal Version 5.0, 5.5, 6.0
Programmer's Guide
Borland International Inc.

TÁRGYMUTATÓ

A
abs 399
absolute 282
adattípus 18, 51, 52
  egész 18, 53
  egyszerű 52
  előredefiniált megszámlálható
  felsorolt 57, 64
  file 63,
  halmaz 63, 67, 69
  karakter 55
  logikai 54, 55
  megszámlálható
  mutató 59, 68
  rekord 61, 66, 69
  résztartomány 58, 65
  sorszámozott 56
  struktúrált 59
  szöveges 55
  sztring 55
  tömb 59, 65, 68
  valós 18, 54
Addr 279, 435
algoritmus 7
append 221, 412
Assign 412
Arc 310, 447
ArcTan 399
aritmetikai műveleti jelek 77
  értékadás 129
ASCII karakterkészlet 35
Assign 219, 220, 237, 255
AssignCrt 229, 441
azonosító 39

B
Bar 310, 447
Bar3D 310, 447
bináris szám 35
bit 35
blokk 135
blokkread 248, 249, 413
blokkwrite 248, 250, 414
byte 35

C
Case 29, 30, 141
  szelektor 30
ChDir 408
CheckBrake 292
CheckEof 292
CheckSnow 292
Chr 192, 402
ciklusutasítás 32, 144
  változó 32
  mag
címke 42
Circle 310, 448
ClearDevice 311, 448
ClearViewPort 311, 448
Close 219, 225, 240, 414
CloseGraph 309, 449
ClrEol 295, 441
ClrScr 295, 441
Concat 193, 404
Cos 399
Copy 194, 404
CSeg 407
Crt unit 214, 292

D
Dec 403
deklarációs rész 15, 47
Delay 441
Delete 194, 404
DelLine 295, 442
DetectGraph 309, 449
device 252
DirectVideo 292
DiskFree 409

dispose 261, 435
DiskSize 409
dokumentálás 4
DosExitCode 423
DOS unit 214
DosVersion 423
DSeg 407
DrawPoly 310,450

E
Ellipse 310, 450
eljárás 168
  hívás 135
eljárás paraméterek 161
  aktuális 160
  érték szerinti 161
  bemenő 162
  cím szerinti 162
  formális 160
  kimenő 163
elsőbbségi szabály 76
EnvCount 423
EnvStr 423
eof 224, 226, 414
eoln 226, 415
erase 255
eszközök 252
értékadó utasítás 26, 128
  aritmetikai 129
  logikai 133
érvényességi kör 188
Exec 424
Exit 424
Exp 399

F
feltételes utasítás 28, 135
FExpand 424
file kezelés 217
  típusos file 234
  típus nélküli file 248
filepos 239, 248, 415
fillchar 285, 286, 436
filesize 239, 248, 416
FillEllipse 310, 450
FillPoly 310, 451
FindFirst 409
FindNext 410
FloodFill 451
flush 225, 416
foglalt szavak 37
folyamatábra 4
for 144
fordítóprogram 4
forward 182
Frac 400
freemem 275, 436
FSearch 410
FSplit 410
function 160, 164
függvény 164
  eljárás paramétere 179

G
GetArcCoords 310, 451
GetAspectRatio 310, 452
GetBCBreak 425
GetBkColor 312, 452
GetColor 312, 453
GetDate 425
GetDefaultPalette 312, 453
getdir 256, 411
GetDriverName 313, 453
GetEnv 425
GetFAttr 425
GetFillPattern 310, 454
GetFillSettings 310, 454
GetFTime 426
GetGraphMode 309, 454
GetIntVec 426
GetImage 311, 454
GetLineSettings 310, 455
GetMaxColor 312, 455
GetMaxMode 313, 456
GetMaxX 313, 456
GetMaxY 313, 456

getmem 275, 436
GetModeName 313, 456
GetModeRange 309, 456
GetPalette 313, 457
GetPixel 311, 457
GetTime 426
GetTextSettings 311, 458
GetPaletteSize 312, 457
GetVerify 427
GetViewSettings 311, 458
GetX 313, 459
GetY 313, 459
globális változók 188
GraphDefaults 309,459
Graph unit 214, 305
GraphErrorMsg 312, 459
Graph3 unit 215
GraphResult 312, 460
grError 324
grFileNotFound 324
grFontNotFound 324
grInvalidDeviceNum 324
grInvalidDriver 324
grInvalidFont 324
grInvalidFontNum 324
grInvalidMode 324
grIOError 324
grNoFontMem 324
grNoInit 324
grNoLoad 324
grNoScanMem 324
grNotDetected 324
grOk 324
goto 42,134
GotoXy 295, 442

H

halmaz vizsgálat 94
halomterület (heap) 259
Halt 427
hexadecimális szám 36
hexstr 281
Hi 407
HighVideo 442

I

I/O műveletek 105
if 28,136
ImageSize 460
implementation 206, 208
InitGraph 309, 460
Inc 403
Int 400
Intr 427
Insert 195, 405
InsLine 295, 442
interface 205, 207
Ioresult 222, 416

J

nyelv jelkészlete 37

K

Keep 428
KeyPressed 443
kifejezés 43
konstansok 67
  típusos 68

L

label 42
LastMode 292
Length 196, 405
Line 310, 461
LineRel 310, 461
LineTo 310, 461
lista tárolás 263
Ln 400
Lo 407
lokális változók 188
LowVideo 443

M

mark 274, 437
maxavail 260, 437
megjegyzés 44

Mem 283
memavail 260, 437
MemL 283
MemW 283
mikroprocesszor 269
MkDir 411
move 285, 286, 437
MoveRel 310, 461
MoveTo 310, 462
MsDos 428
mutató 259

N
new 260, 438
nil 277
NormVideo 443
NoSound 296, 443

O
Odd 400
Ofs 407
offsetcím 270
operandus 43
operátor 43, 75, 76
  aritmetikai operátorok 77
  bitenkénti operátorok 83
  boolean operátorok 85
  halmaz műveleti op. 91
  pointer operátorok 90
  realációs operátorok 89
  sztring összekapcsolás 91
Ord 405
OutText 311, 462
OutTextXY 311, 462
Overlay unit 214
OvrClearBuf 432
OvrGetBuf 432
OvrGetRetry 432
OvrInit 433
OvrInitEMS 433
OvrSetBuff 434
OvrSetRetry 434

Ö
összetett utasítás 33

P
packed kulcsszó 38
PackTime 428
ParamCount 233, 429
ParamStr 223, 429
Pi 400
PieSlice 310, 463
pointer 259
Pos 196, 405
Pred 403
Printer unit 214
precedencia szabály 76, 79
procedure 160
program 3
program kulcsszó 48
- fej 15, 47
  fordítása 16
  futtatása 16
  törzs 47
programozási nyelv 3
ptr 281, 438
PutImage 311, 463
PutPixel 311, 463

R
Random 401
Rndomize 401
read 116, 226, 238 , 417
readln 116, 226, 417
ReadKey 295
Rectangle 310, 464
rekurzió 183
  kölcsönös 187
release 274, 438
rename 255, 418
repeat until 34, 149
Reset 219, 229, 237, 248, 418
RestoreCrtMode 309, 464
Rewrite 219, 229, 237, 248, 418
RmDir 411

Round 401
RunError 429

S
SearchRec 410
Seg 407
Seek 239, 248, 420
SeekEof 219, 224, 420
SeekEoln 219, 224, 420
Sector 310, 464
SetActivePage 311, 464
SetAllPalette 312, 465
SetAspectRatio 310, 466
SetBkColor 312, 466
SetCBreak 429
SetColor 312, 466
SetDate 430
SetFAttr 430
SetFillPattern 310, 466
SetFillStyle 310, 467
SetFTime 430
SetGraphBufSize 309, 467
SetGraphMode 309, 468
SetIntVec 431
SetLineStyle 310, 468
SetPalette 312, 469
SetTextBuff 421
SetTextJustify 311, 469
SetTestStyle 311, 470
SetTime 431
SetUserCharSize 311, 470
SetViewPort 311, 470
SetVisualPage 311, 471
SetVerify 431
SettextBuf 219, 221
SetWriteMode 471
Sin 401
sizeof 260, 439
Sound 296
SPtr 408
SSeg 408
struktúrált utasítások 135
Str 197, 406
Succ 403
Sqr 402
ßqrt 402
System unit 215
Swap 408
SwapVectors 431

SZ
szabványos függvéneyk 95
számok 40
szegmenscím 270
szemantika 4
szematikai hiba 7
szintaktiaki hiba 4, 7
szintaxis 4
sztring 41, 191

T
TextAttr 293
TextBackGround 293, 296, 443
TextColor 293, 296, 444
TextHeight 311, 472
TextMode 293, 294, 444
TextWidth 312, 472
típusdefiníció használata
  paraméterlistán 179
típus konverzió 70
Turbo3 unit 215
Trunc 402
truncate 239, 248, 421
type 63
TypeOf 439

U
unit 48,205
UnpackTime 432
Uppcase 192, 406
utasítás 127
  egyszerű 128
  értékadó 128
utasítás diagram 4
uses 48

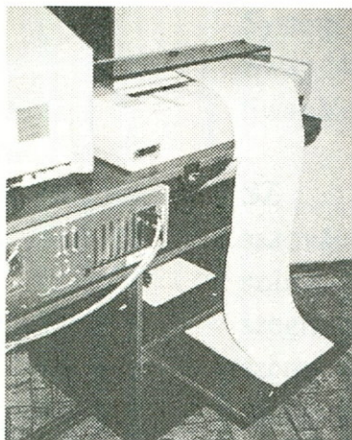
V

Val 198, 406
változó 14, 17, 51
  aktuális értéke 51
  azonosító 26
  pillanatnyi értéke 51
var kulcsszó 162
verem 267

W

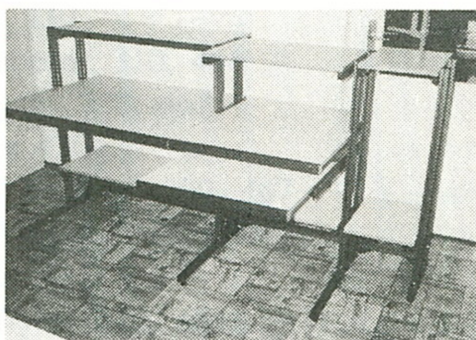
WhereX 296, 445
WhereY 296 , 445
Window 289, 290, 445
WindMax 293
WindMin 293
with 150
while 34, 147
write 106, 223, 422
writeln 106, 223, 422

SZÁMÍTÓGÉPASZTALOK ÉS TARTOZÉKOK



Modul-Alter típusú számítógépasztalok

Alter típusú számítógépasztalok
görgős szekrénnyel, lábtartóval



Alter és Modul-Alter márkanevű számítógépasztalok különféle méretben (60, 90, 100, 120, 135, 160, 180 cm) állítható magasságú, két- és háromszintes kivitelben készülnek.

A Modul típusból több asztal összekapcsolható.

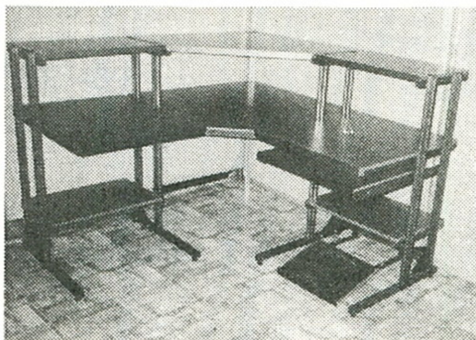
Oktatási célra ideális méretű, jó helykihasználású munkahelyet lehet kiépíteni. Helyigényük kicsi.

A vázszerkezet eloxált alumínium profil, a munkafelület különféle színű laminált forgácslap.

Görgős szekrény, fénymásolószekrény, nyomtatóállvány, toronytartó, olvasólap, lábtartó, kihúzható pótlap, kábeltartó egészíti ki a munkahelyet.

A gyártott típusok mellett egyéni kívánságok is kivitelezhetők.

Az asztalok összeszerelve vagy elemekben szállíthatók. Áruk kedvező.



Kérje részletes prospektusunkat, árajánlatunkat.
Áruválasztékunk központunkban megtekinthető.

SPIRALTER Kft.

1163 Budapest, Batsányi u. 6.
(Örs vezér tértől 5 perc, sashalmi HÉV-megállónál)
Tel+Fax: 271-04-05, Tel: 271-14-65

AGRO-PRINT Kft. Gyál, 92-211
Felelős vezető: Tóth Antal

Ára: 756 Ft