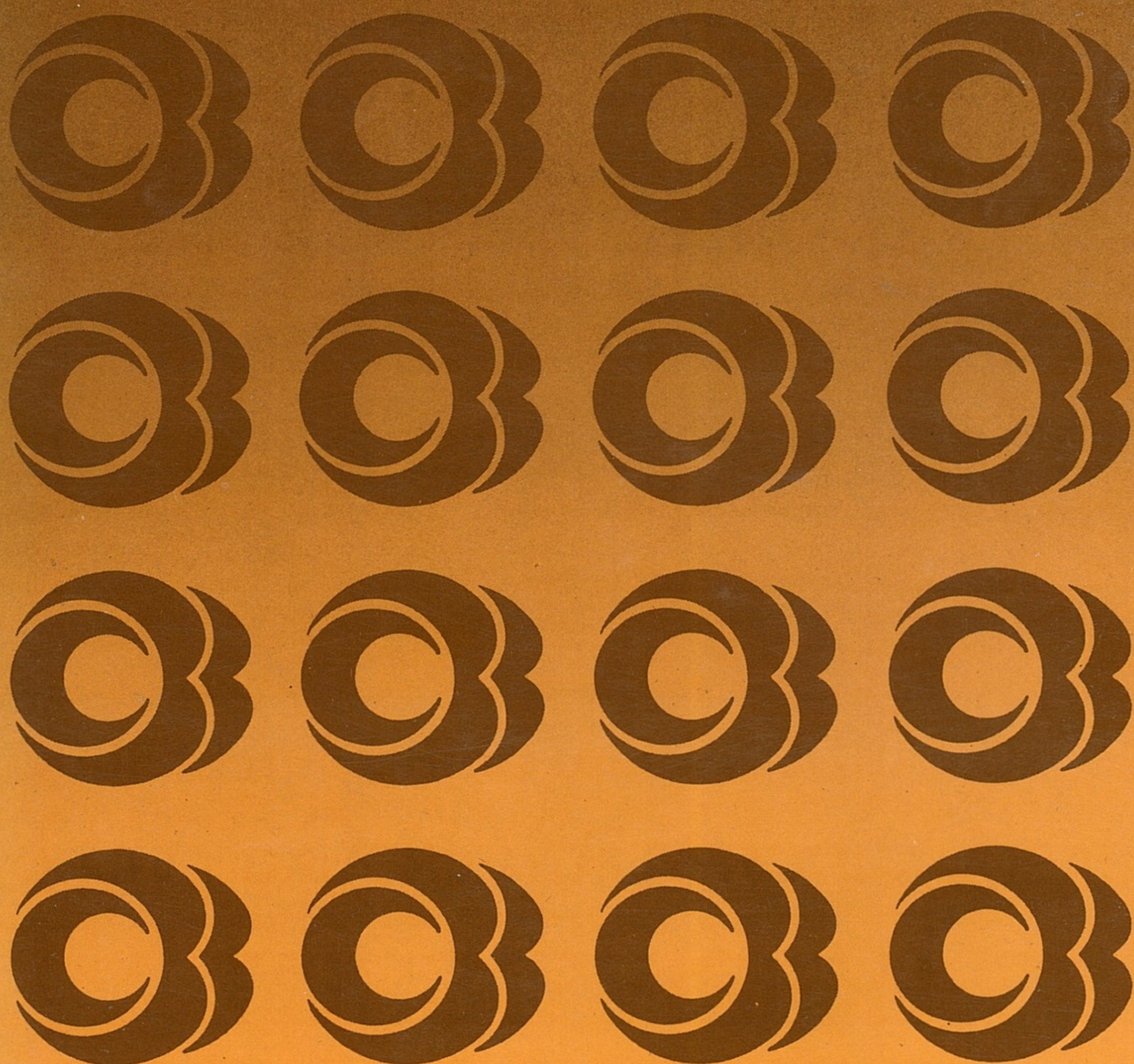
OBJEKTUM-ORIENTÁLT PROGRAMOZÁS

TURBO PASCAL 6.0-BAN

TURBO VISION

LEMEZMELLÉKLETTEL



ComputerBooks

Benkő Tiborné
Benkő László
Kiss Zoltán
Tóth Bertalan

OBJEKTUM-ORIENTÁLT PROGRAMOZÁS TURBO PASCAL 6.0-BAN

TURBO VISION

LEMEZMELLÉKLETTEL

Lektor: Poppe András

COMPUTERBOOKS
BUDAPEST

Köszönetnyilvánítás

Ezúton szeretnénk megköszönni Poppe Andrásnak a hasznos tanácsait, észrevételeit, melyeket figyelembe véve reméljük hasznos irodalmat adunk a Turbo Pascal 6.0 felhasználók kezébe.

A szerzők

TARTALOMJEGYZÉK

Bevezetés 9

I. rész Turbo Pascal 6.0

1. Turbo Pascal 6.0 bemutatása 11
1.1. Turbo Pascal 6.0 jellemzői 11
1.2. Hardver és szoftver igények 11
1.3. Installálás 12
1.4. Ismerkedés az IDE-vel 12
1.4.1. Az IDE felépítése és ablakai 13
1.4.2. Dialógus dobozok 15

2. Turbo Pascal 6.0 integrált fejlesztői környezet (IDE) 20
2.1. Az IDE kezelésének általános szabályai és a menürendszer bemutatása 20
2.1.1. Az ún. (*System*) menü: ≡ 21
2.1.2. A *File* menü 21
2.1.3. Az *Edit* (szerkesztés) menü 25
2.1.4. A *Search* (keresés) menü 26
2.1.5. A *Run* (futtatás) menü 29
2.1.6. A *Compile* (fordítás) menü 30
2.1.7. A *Debug* menü 30
2.1.8. Az *Option* (rendszerjellemzők beállítása) menü 33
2.1.9. *Window* menü 43
2.1.10. *Help* menü 44

3. Objektum-orientált programozás 46
3.1. Az objektum-orientált programozás alapfogalmai 46
3.2. Objektumok 46
3.2.1. Öröklés 47
3.3. Metódusok 50
3.3.1. Program és az adatok 51
3.3.2. A **private** kulcsszó 52
3.3.3. Metódusok definíciója 53
3.3.4. A metódus hatásköre és a Self paraméter 54
3.4. Egy objektum adatmezői és a metódusok formális paraméterei 56
3.5. Objektumok unit-ban 56
3.6. Egységbezárás (*encapsulation*) 59

3.7. Többrétűség (*polymorphism*) 60
3.7.1. Sokalakú objektumok 62
3.7.2. Örökölt statikus metódusok 63
3.7.3. Virtuális metódusok 63
3.7.4. Korai és késői kötés 64
3.7.5. Példa a késői kötésre 65
3.8. Objektum típus-kompatibilitás 65
3.9. Eljárás vagy metódus ? 66
3.10. Statikus vagy virtuális metódus 67
3.11. Dinamikus objektumok 68
3.11.1. Helyfoglalás és kezdeti értékadás a **new** eljárással 69
3.11.2. Dinamikus objektumok felszabadítása **dispose** eljárással 69
3.11.3. Konstruktor 70
3.11.4. Destruktor 71
3.12. Belső adatszerkezetek 72
3.12.1. Objektumok belső adatformátuma 72
3.12.2. Virtuális metódus tábla (VMT) 74
3.13. Összefoglaló az objektum-orientált programozás alapfogalmairól 75
3.14. Példa az objektum-orientált programozásra 77
3.15. Objektum-orientált programozáshoz kapcsolódó rutinok 87

4. A Turbo Pascal és az assembly nyelv kapcsolata 88
4.1. A beépített (inline) assembler 88
4.1.1. Az **asm** utasítás 88
4.1.1.1. Regiszterek használata 89
4.1.2. Az assembler utasítások felépítése 89
4.1.2.1. Cimkék 89
4.1.2.2. Prefixum műveleti kódok 90
4.1.2.3. Assembler utasítások 91
4.1.2.4. Assembler direktívák 92
4.1.2.5. Operandusok 93
4.1.3. Kifejezések 94
4.1.3.1. Kifejezések elemei 94
4.1.3.2. Kifejezések osztályai 98
4.1.3.3. Kifejezések típusai 99
4.1.3.4. Kifejezésekben használható operátorok 101
4.1.4. Assembler eljárások és függvények 102
4.2. Külső assembler kód beszerkesztése 103
4.2.1. A Turbo Assembler és a Turbo Pascal 104
4.3. Gépi kód beépítése a Pascal programba (inline) 105
4.3.1. Az Inline utasítás 105
4.3.2. Az Inline direktíva 106
4.4. A fejezet összefoglalása egy példa bemutatásával 107

5. Turbo Vision alapismeretek 112
5.1. Turbo Vision keretrendszer elemei 113
5.1.1. Látvány (*view*) 114
5.1.2. Esemény (*event*) 114
5.1.3. Néma objektumok 114
5.2. Programozás Turbo Vision felhasználásával 115
5.2.1. Alkalmazói objektum 119
5.2.2. A dialógus doboz objektum 120
5.2.3. A Tv_demo főprogram 121
5.2.3.1. Az *Init* metódus 121
5.2.3.2. A *Run* metódus 121
5.2.3.3. A *Done* metódus 122

6. Turbo Vision alkalmazói programok 125
6.1. Menütervezés 125
6.1.1. Ablak (*window*) nyitása 132
6.1.2. Ablak (*window*) inicializálása 133
6.1.3. Egyszerű megjelenítő 137
6.1.4. Pufferelt rajzolás 139
6.1.5. Kép görgetés előre és hátra 141
6.1.6. Többszörös *view* az ablakban 143
6.2. Dialógus doboz 145
6.2.1. Opció beállító dialógus doboz 146
6.2.2. Vezérlés fogadása 147
6.2.3. Input sor objektum 151
6.2.4. Statikus szöveg kiírása 155
6.2.5. Lista doboz 155
6.2.6. History 155
6.2.7. Standard dialógus dobozok 155
6.3. Objektum hierarchia 156
6.3.1. Objektumok típusai 157
6.3.2. Objektum - példányok létrehozása, származtatott típusok 158
6.3.3. Turbo Vision metódusai 159
6.3.4. Turbo Vision mezői 160
6.3.5. Primitív objektum típusok 161
6.3.6. *View* (látvány) 162
6.3.6.1. Csoportok 163
6.3.6.2. Terminal *view*-k 165
6.3.7. A Turbo Vision nem látható elemei 167

7. A *view* objektumok 171
7.1. Az egyszerű *view* objektumok 171
7.1.1. A *view* elhelyezése a képernyőn 172
7.1.2. A *view* megjelenítése a képernyőn 173
7.1.3. A *view* viselkedése 174
7.2. Összetett *view* objektumok 174
7.2.1. Csoportok és *subview*-k 175
7.2.2. Felvétel a csoportba 175
7.2.3. A csoportok megjelenítése a képernyőn 177
7.2.4. Kapcsolatok a *view*-k között 177
7.2.5. A *subview*-k és a *view* fák 179
7.2.6. Kiválasztott és fókuszált *view*-k 182
7.2.7. *Modal* view-k 183
7.2.8. Az alapértelmezés szerinti viselkedés módosítása 184
7.2.8.1. Az *Options* mező 184
7.2.8.2. A *GrowMode* mező 187
7.2.8.3. A *DragMode* mező 188
7.2.8.4. A *State* mező és a *SetState* metódus. 190
7.2.9. A *view* színének beállítása 193

8. Esemény-vezérelt programozás 197
8.1. Az események természete 198
8.1.1. Az események fajtái 198
8.1.2. Események és parancsok 200
8.2. Az események irányítása 200
8.2.1. Honnan érkeznek az események? 201
8.2.2. Hová irányítódnak az események? 201
8.2.3. Események tiltása 203
8.2.4. A fázis 204
8.3. A parancsok 206
8.3.1. Parancsok definiálása 206
8.3.2. Parancsok kötése 207
8.3.3. Parancsok engedélyezése és tiltása 207
8.4. Események kezelése 208
8.5. Az esemény rekord 209
8.5.1. Események törlése 210
8.5.2. Elhagyott események 210
8.6. Az események mechanizmusának módosítása 210
8.6.1. Események centralizált összegyűjtése 211
8.6.2. A *GetEvent* metódus átdefiniálása 212
8.6.3. Az üresjárati (*Idle*) idő felhasználása 213
8.7. *View*-k közötti kommunikáció 213
8.7.1. Közvetítők használata 214
8.7.2. *View*-k közötti üzenetek 214
8.7.3. Ki kezeli le a szórt (*broadcast*) eseményeket? 216
8.7.4. A *HandleEvent* metódus meghívása 217
8.7.5. Szövegösszefüggéstől függő HELP (*context sensitive*) 217

9. Turbo Vision nem látható elemei 218
9.1. *Stream*-ek 218
9.1.1. A *stream* sokrétűsége 218
9.1.2. A *stream* megnyitása 219
9.1.3. Írás és olvasás a *stream*-en 220
9.1.3.1. A *Put* eljárás 220
9.1.3.2. A *Get* eljárás 220
9.1.3.3. Hibakezelés 221
9.1.4. A *stream* lezárása 221
9.1.5. Az objektumok és *stream*-ek 221
9.1.5.1. Betöltés és tárolás 221
9.1.5.2. A regisztrálás 222
9.1.5.3. Az objektum azonosítása 223
9.1.5.4. Az automatikus mezők 223
9.1.6. Az *stream* működési mechanizmusa 224
9.1.6.1. A *Put* eljárás 224
9.1.6.2. Az olvasási folyamat........ 224
9.1.7. A kollekciók és a *stream* 224
9.1.7.1. Az objektumok definíciói 225
9.1.7.2. A regisztrációs rekordok 226
9.1.7.3. A regisztrálás 227
9.1.8. Hivatkozás *subview*-ra 228
9.1.9. Hivatkozás egyenrangú *view-kra* 229
9.1.10. A *desktop* tárolása és betöltése 229
9.1.11. Véletlen hozzáférés a *stream*-en 230
9.2. Erőforrások 231
9.3. Kollekciók 233
9.3.1. A kollekció (*collection*) objektum 234
9.3.2. A típusellenőrzés és a kollekciók 234
9.3.3. A nem objektum típusú elemek 235
9.3.4. Kollekció létrehozása 235
9.3.5. Iterációs metódusok (iterátorok) 236
9.3.6. Rendezett kollekciók 238
9.3.7. Sztring-kollekció 239
9.3.8. A kollekció sokrétűsége 242
9.3.9. Kapcsolat a memóriakezelővel 243

10. Turbo Vison kiegészítések 244
10.1. Megbízható programok írása 244
10.1.1. Biztonsági terület 244
10.1.2. Nem memóriakezelésből származó hibák 246
10.2. A Turbo Vision alkalmazói programok nyomkövetése 247
10.3. Bittérképek felhasználása 248
10.3.1. A bitműveletek összefoglalása 249
10.4. Turbo Vision alkalmazói programok *overlay* szervezése 249

F1. Turbo Vision unit-ok rövid referenciái 252

F2. Turbo Pascal 6.0 fordítási direktívák 278

F3. A 8086/80286 és a 8087/80287 processzorok utasításkészlete... 290

F4. A lemezmelléklet ismertetése 302

Irodalomjegyzék 310

BEVEZETÉS

Az első Turbo Pascal fordítók a 80-as évek első felében jelentek meg. Az első változatok mind a 8 bites 8080/Z80 mikroprocesszorú számítógépeken futó CP/M operációs rendszer, mind pedig a 8088/8086 mikroproceszzoros 16 bites rendszerekre kifejlesztett CP/M-86, illetve MS-DOS operációs rendszerek alatt voltak hozzáférhetők. A Turbo Pascal nagy népszerűséget vívott ki magának. Ez több tényezőnek tulajdonítható. Ezek egyike a rendkívül alacsony ár/teljesítmény viszony (price/performance ratio), a másik fontos tényező a rendkívül felhasználó-barát kezelő felület és a kitűnő programdokumentáció.

A Turbo Pascal rendszer 3.0-ás változata kitűnt az integrált fejlesztői környezetével (editálás-fordítás-futtatás egybeépítésével), az igen gyors fordítással és tömör és gyors célprogramok készítésével. A Turbo Pascal fordítók a szabványos Pascal majdnem minden elemét megvalósítják, ugyanakkor ki is bővítik a nyelvet úgy, hogy az IBM PC tulajdonságait minél jobban kiaknázhassuk (lásd pl. a grafikus lehetőségeket). A Borland cég a Pascal fordítóját folyamatosan fejlesztette, karbantartotta. A Turbo Pascal rendszerek népszerűségéhez a fentieken kívül még hozzájárultak a Borland cég által közzétett ún. tool-box-ok is. Ezek olyan forrásnyelvi szubrutincsomagok, amelyek egy adott feladat-típus megoldását teszik nagyon egyszerűvé az alkalmazói programok fejlesztői számára.

A Borland cég 1987-ben a Turbo Pascal 4.0 verziónál továbbfejlesztette a fejlesztői környezetet, bevezette az ún. unit-okat (ezzel támogatva a nagyobb, több forrás-, illetve tárgykódú modulból álló programok írását), teljesen új grafikus könyvtár-rendszert építettek be a fordítóba, amely nagy mértékben elősegíti a grafikus kártyáktól független grafikus alkalmazói programok készítését, és sok új, beépített eljárást bocsátottak a felhasználók rendelkezésre. A nyelvi elemek bővítése nagy mértékben elősegítette a szabványos Pascal bizonyos kötöttségeitől való "megszabadulást". Ez azonban kétélű fegyver. Egyrészt gyorsan lehet hatékony programokat írni IBM PC (és azzal kompatibilis) mikroszámítógépekre, másrészt viszont sokat veszthetünk a magasszintű programozási nyelvek nyújtotta hordozhatóságból. A megfelelő kompromisszum meghozatala természetesen mindig az adott felhasználói program jellegétől függ.

A következő mérföldkő a Turbo Pascal történetében az objektum-orientált programozás lehetőségének a megteremtése volt. Ez az 5.5-ös verziótól áll rendelkezésre. Az 5.5-ös változat 1989-ben jelent meg.

A Borland cég 1991-ben a 6.0 verzióban teljesen új integrált fejlesztői környezetet fejlesztett ki. Az új integrált fejlesztői környezetben lehetőség nyílik többszörösen átfedő ablakok kezelésére, egérhasználatra. Az új editor több file-t tud kezelni, az editálás 1 Mbyte-ig kiterjeszthető. A hibakeresési lehetőség is bővített változata segíti a felhasználót a programjai belövéséhez. Az objektum-orientált programozás egy célszerű új utasítással bővült. A *private* mezők és metódusok szerepelhetnek az objektum deklarációiban.

A programrendszerhez tartozik egy teljes *inline* assembler. Az objektum-orientált programozás elősegítés érdekében a Borland cég a Turbo Vision könyvtárat tette közzé, amellyel tulajdonképpen az integrált fejlesztői rendszerét fejlesztette ki. Ez a könyvtár lerövidíti a programfejlesztési időt, hiszen egy keretrendszert ad, amellyel modern, minden igényt kielégítő felhasználói felületek hozhatók létre az egyes alkalmazói programok számára.

A gyors fejlődést jelzi, hogy jelen könyv kiadása közben jelent meg a Turbo Pascal for Windows nevű termék, amely a Microsoft cég Windows rendszere alatt futó alkalmazói programok fejlesztését teszi lehetővé a már megszokott Turbo Pascal környezetben.

Jelen könyv feltételezi a Turbo Pascal ismeretét, ezért főleg az objektum-orientált Pascal programozással, az integrált fejesztői környezettel, és a Turbo Vision könyvtár felhasználásával foglalkozik, valamint kitér az assembler programrészletek Pascal programba való beültetésére is. Mivel célunk a 6.0-ás Turbo Pascal sajátosságainak, illetve a Turbo Vision keretrendszer megismertetése, programhordozhatósági kérdésekkel értelemszerűen nem foglakozunk.

Ha az olvasó úgy érzi, hogy pótolni valója van a Pascal alapokban, akkor azt javasoljuk, szerezzen be egy-két könyvet az igen bőséges magyar nyelvű Pascal irodalomból. Ehhez talán segítséget nyújt a könyvünk végén található irodalomjegyzék.

Könyvünk lemezmellékletet is tartalmaz. Ezen a lemezen a Turbo Pascal 6.0-ban és a Turbo Vision felhasználásával készült programok forrásnyelvi listái találhatók.

1. TURBO PASCAL 6.0 BEMUTATÁSA

1.1. Turbo Pascal 6.0 jellemzői

A Turbo Pascal 6.0 a világszerte ismert standard Pascal fordítóra épül. Teljes a kompatibilitás a Turbo Pascal korábbi verzióiban írt programokkal, azonban az új verzió az alábbiakat is magába foglalja:

- teljesen új integrált fejlesztői környezet Integrated Development Environment, a továbbiakban (IDE), amelyre az alábbiak jellemzők:
  - egymást többszörösen átfedő ablakok kezelése,
  - egérhasználat, menük és dialógus dobozok,
  - több file-t kezelő editor, editálás 1 Mbyte-ig,
  - bővített hibakeresési lehetőség,
  - munkaállományok és azok környezetének (*desktop*) teljes megőrzése és újratöltése.
- Egy objektum-orientált alkalmazói keretrendszer: a Turbo Vision,
- teljes inline assembler,
- private mezők és metódusok az objektum deklarációiban,
- kiterjesztett szintaktikai direktíva,
- a 80286-os processzor utasításkészletének megfelelő kódgenerálás
- tipizált konstansok cím referenciái,
- *far* és *near* eljárások,
- új *heap* kezelő,
- kiterjesztett, azonnali (online) help, ahonnan a könyvtári függvényekre és eljárásokra vonatkozó mintapéldák közvetlenül a forrásállományokba másolhatók.

1.2. Hardver és szoftver igények

A Turbo Pascal 6.0 programrendszer használatához szükséges egy IBM PC kompatibilis számítógép min. 640 Kbyte RAM-mal, 80 karakter oszlopos monitor, merevlemezes egység (hard disk) és legalább egy hajlékonylemez meghajtó (floppy drive). Az operációs rendszer legyen PC-DOS (MS-DOS) 2.0, vagy magasabb verziószámú változata. A teljes Turbo Pascal rendszer körülbelül 3 Mbyte helyet foglal el a merev lemezen. A Turbo Pascal támogatja a 640 Kbyte-on felüli memória (extended vagy expanded) használatát.

A Turbo Pascal fejlesztői környezete támogatja az egér (*mouse*) használatát is.

1.3. Installálás

A Turbo Pascal 6.0 rendszert az első gyári lemezen található INSTALL program segítségével kell installálni. Ez biztosítja azt, hogy minden, a rendszer megfelelő használatához szükséges állomány rendelkezésre álljon, és hogy a programrendszer egyes paramétereinek alapértéke az adott hardver konfigurációnak megfeleljen. További előnye az INSTALL program használatának az, hogy kialakítja a Turbo Pascal használatához optimális alkönyvtárrendszert a merev lemezen, az általunk megadott főkönyvtárban.

Installálás menete a következő:

- Helyezzük be az 1. számú installációs lemezt az A: jelű lemezegységbe.
- Adjuk ki a a DOS-nak az A: meghajtóról az INSTALL parancsot.
- Az INSTALL program egy menüvel kezdeti installációs paramétereket kínál fel (pl. könyvtárnevek, példaprogramokat kicsomagolja-e, stb.). Ezen opciók egy része az *ENTER* billentyű leütésével ki/bekapcsolható, más esetben egy-egy sztringet adhatunk meg (pl. könyvtárnevet).

Az installálást azzal fejezzük be, hogy az AUTOEXEC.BAT file-ban a SET paranccsal állítsuk be a PATH környezeti változóba a Turbo Pascal keresési útvonalát. Tehát, ha a C: merevlemez egységen a TP alkönyvtárba installáltuk a Pascal fordítót, akkor

```
SET PATH=C:\TP
```

paranccsal állíthatjuk be a Turbo Pascal megfelelő DOS keresési útját. Ezután a TURBO paranccsal bármely könyvtárból elindíthatjuk a Turbo Pascal-t.

A rendszer installálása után javasoljuk a felhasználóknak, hogy próbálják ki a TPTOUR programot. A TPTOUR végigvezeti a felhasználót a Turbo Pascal új integrált fejlesztői környezetén, bemutatja a fejlesztői környezet új elemeit (ablakhasználat, egérkezelés stb.).

1.4. Ismerkedés az IDE-vel

A Turbo Pascal megkönnyíti a programozási munkát, mert nincs szükség külön editorra, fordító programra, szerkesztőre és hibakeresőre, mert mindez a lehetőség be van építve, közvetlenül elérhető az IDE-ből. Az IDE részletes ismertetésével a 2. fejezet foglalkozik.

1.4.1. Az IDE felépítése és ablakai

Az IDE ablaka három látható elemből épül fel:

- a főmenü az ablak tetején,
- az ablak közepén a *desktop* (az íróasztal),
- az ablak alján a státuszsor.

A főmenü a menüparancsokkal közvetlenül elérhető, a kiválasztott menü magasabb fényerővel világít. A menüpontnál a három egymást követő pont (...) jelzi, hogy a menüponthoz dialógus doboz tartozik, a → pedig az almenüt jelzi. Ha a nincs jelzés, akkor a menüpont kiválasztása parancsként hajtódik végre.

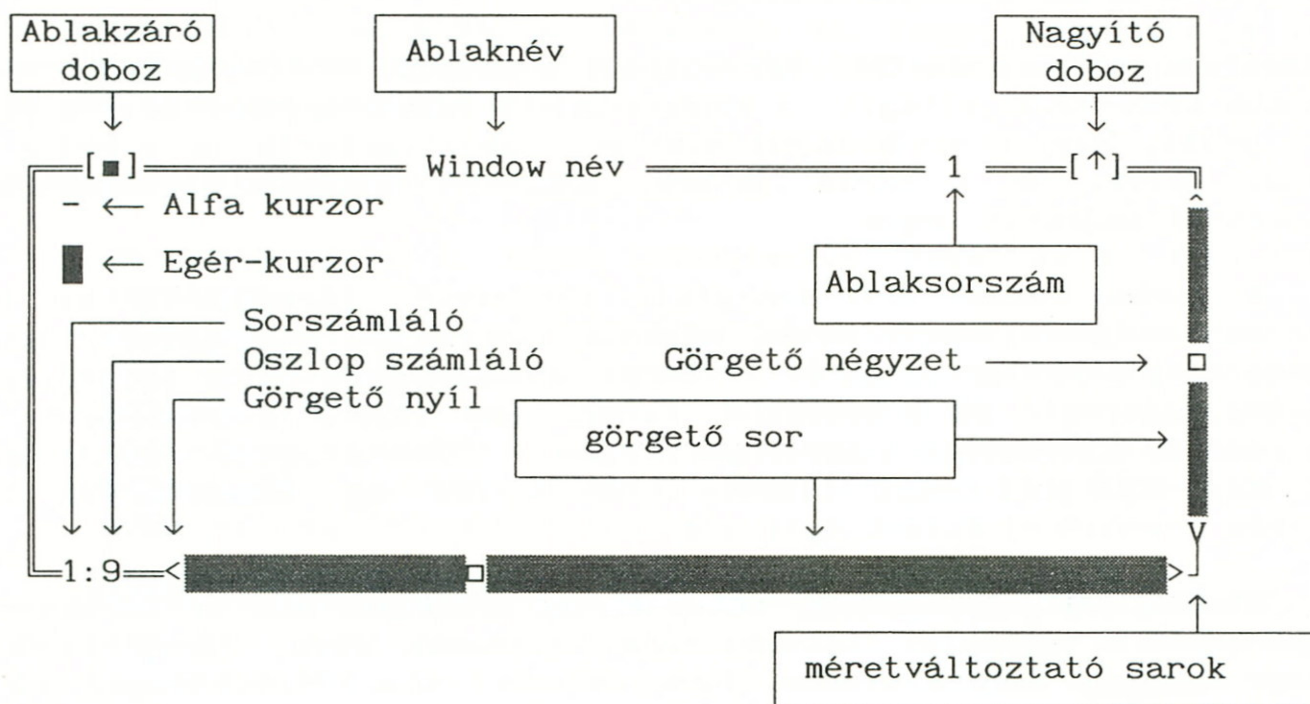
A Turbo Pascal 6.0 integrált fejlesztői környezete, az IDE ablakozós (window) technikával működik; az egyes programfunkciókhoz különböző ablakok tartoznak (1.1. ábra). Minden egyes ablak tetszőleges méretűre állítható és a képernyőn tetszőleges helyre áthelyezhető. Az első indítás alkalmával a képernyő közepén a TURBO program verziószáma, majd az *ENTER* billentyű leütése után a *IDE* egy **NONAME00.PAS** (00 sorszámú nevenincs) file-t nyit meg.

Minden ablaknak van egy neve, ez az *ablaknév*, ami az ablakkeret közepén, felül látható. Az *IDE* saját ablakain kívül (*Help window, Message window, Output window, Watch window,* stb.) tetszőleges számú szerkesztő (editáló) ablakot nyithatunk meg. Minden egyes nyitott forrásállományhoz egy önálló ablakot rendel az *IDE,* ezen ablakok közül az első 9 darab egy-egy *ablaksorszámot* kap. Az editáló ablakok nevei a szerkesztés alatt álló file nevével egyeznek meg. Az ablakok további tartozékainak csak akkor van jelentősége, ha egér (mouse) is installálva van a számítógépünkön. (Az *IDE* kényelmes kezeléséhez az egér használatát javasoljuk.)

A képernyőn a jól ismert *alfanumerikus kurzor* mellett, ha egeret is installáltunk a számítógépünkhöz, egy tömör téglalap is látható. Ezt *egér-kurzor*-nak nevezzük. Az egér-kurzor az egér mozgatásával tetszőleges pozícióba helyezhető. A kényelmes ablakkezelés az egér-kurzor és az egér baloldali nyomógombja segítségével lehetséges. A továbbiakban *kattintás*-nak vagy *klikkentés*-nek hívjuk azt, ha az egér-kurzorral a képernyő adott pozícióján állva, egyszer, rövid ideig megnyomjuk az egér baloldali gombját.

Minden ablak bal felső sarkában szögletes zárójelek közt láthatunk egy kis tömör négyszöget [■], ez az *ablakzáró doboz (close box).* Az egérrel az ablakzáró dobozon kattintva az adott ablakot bezárhatjuk. Mivel ekkor az ablak tartalma elvész, szerkesztő ablakok zárásánál az *IDE* figyelmeztet minket. Minden ablak jobb felső sarkában egy szögletes zárójelek közt álló nyilacska látható ([↑]). Ez az ún. *nagyító doboz* (*zoom box*). A [↑] alakú nagyító dobozon kattintva az aktuális ablak el fogja foglalni a képernyő teljes hasznosítható részét. Ha egy ablak a teljes, rendelkezésre álló képernyő területet elfoglalja, akkor a

nagyító dobozban egy kettős nyilacska található. Ez az automatikus kicsinyítés lehetőségére utal. A nagyító dobozon kívül a *méretváltoztató sarok* - minden ablak jobb alsó sarka - használható egy ablak "átszabására". Álljunk az egér-kurzorral erre a sarokra, majd a az egér baloldali gombját folyamatosan lenyomva tartva és az egeret le-fel, jobbra-balra mozgatva az aktív ablak mérete megváltoztatható.



1.1. ábra
Tipikus ablak

Az ablakok - a méretük megváltoztatásán túlmenően - természetesen a képernyő tetszőleges helyére áthelyezhetők. Álljunk az egérrel az áthelyezendő ablak keretének tetejére (pl. az ablak nevére), és az egér baloldali gombját folyamatosan lenyomva tartva az ablak a képernyőn követni fogja az egér mozgását. Egérrel nem rendelkező rendszereken az ablakkezelés a főmenü *Window* menüjének funkcióival lehetséges.

Az egyes ablakok részben, vagy teljesen fedhetik egymást. Egyik ablakból a másikba az egér segítségével könnyen átléphetünk: kattintsunk az egérrel az aktivizálandó ablakban. A különböző editáló ablakok az *ALT* billentyű és az ablaksorszám (1-től 9-ig) egyidejű lenyomásával a billentyűzetről is könnyen aktivizálhatók. Az *ALT-0* billentyűkombináció a *Window* főmenüpont *List* funkcióját aktivizálja. Ekkor egy ún. dialógus dobozban (a dialógus doboz fogalmát lásd később) felsorolt létező ablakok közül a kurzorral választhatunk.

Ha egy ablakot keskenyre zsugorítottunk, akkor a forrás-szövegnek (vagy a Turbo Pascal üzeneteinek, helpjeinek) csak egy kis része látszik egyszerre. Hogy mégis az ablak méretének megváltoztatása nélkül az aktuális szöveges állomány egyéb részeit megnézhessük, a szöveg

görgethető. Egér nélküli rendszer esetén a kurzormozgató nyilakkal (↓, ↑, →, ←), valamint a *PgDn*, *PgUp* billentyűkkel görgethetjük az aktív ablak tartalmát. Ha van egerünk, akkor a *görgető sorokban* (melyek az ablak alsó, illetve jobb szélső keretén találhatók) lévő *görgető nyilakkal*, illetve *görgető négyzetekkel* mozgathatjuk a szöveget (kattintással, illetve a baloldali gombot lenyomva tartva és az egeret mozgatva).

Az editáló ablakok alsó keretén további információ is található: az alfanumerikus kurzor aktuális poziciójára vonatkozó *sor* és *oszlop számláló.*

A képernyő tetején mindig látható a *főmenü* sora és a legalján az *IDE* aktuális állapotára vonatkozó *státuszsor*.

F1 Help	F2 Save	F3 Open	Alt-F9 Compile	F10 Menu

A státuszsorban kiemelve látható billentyűk ún. *hot key*-k, azaz leütésük hatása azonnal jelentkezik. (A hot key elnevezés a jól ismert hot line = forró drót, azaz állandóan rendelkezésre álló telefonvonal kifejezés mintájára született.) Az *F1*, melynek hatására a *Help* ablakban az aktuális szituációra vonatkozó help-szöveg jelenik meg (*context sensitive help*). Az *ESC* billentyű leütésével bármely művelet félbeszakítható. A főmenüben kiemelő színnel jelzett billentyűk az *ALT* billentyű egyidejű leütésével töltenek be *hot key* funkciót. A főmenü egyébként az *IDE* bármely helyéről az *F10* leütésével aktivizálható. Innen sorozatos kiválasztásokkal minden feladat hívható, mint pl. szövegszerkesztés, fordítás, különböző opciók beállítása, szimbolikus nyomkövetés stb. A főmenü egyes pontjai a képernyő felső részén találhatók, aktiválásuk a kurzor rápozícionálásával és az *ENTER* leütésével (vagy egy kattintással), vagy a kezdőbetű begépelésével történhet. (Ugyanez az elv érvényesül a további menük esetén is.) Egy főmenüpont kiválasztása után az adott funkcióhoz tartozó ún. *roll-in* menü jelenik meg a képernyőn.

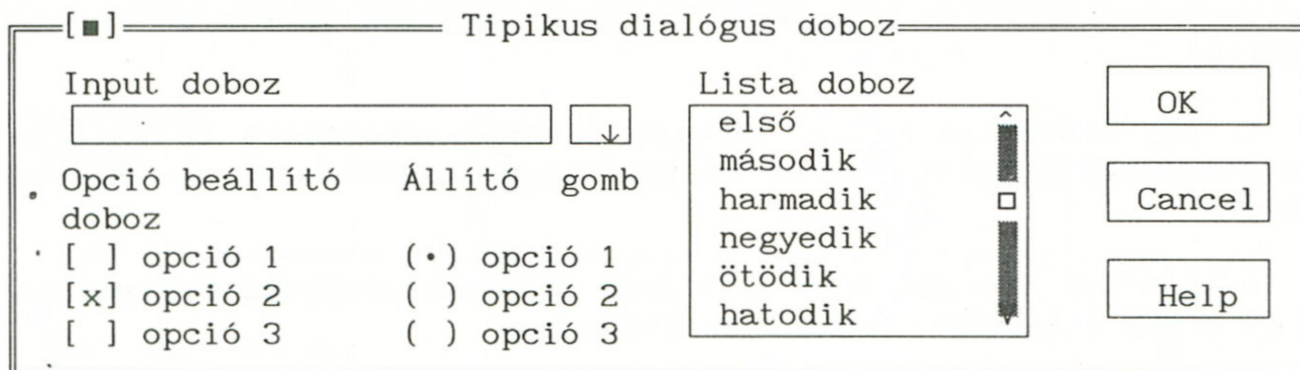
1.4.2. Dialógus dobozok

Ha a menü parancs után a ... jel van, ez azt jelenti, hogy a végrehajtáskor egy ún. dialógus doboz jelenik meg.

A dialógus doboz a különböző beállításokra 5 fajta vezérlési típust alkalmaz:

- opció kiválasztó doboz (*check box*),
- állító gomb (*radio button*),
- vezérlő gomb (*push button*),
- input doboz (*input box*),
- lista doboz(*list box*).

Példaként nézzünk meg egy dialógus dobozt az 5 fajta beállítás illusztrálására:



1.2. ábra
Tipikus dialógus doboz

A dialógus doboznak legalább három vezérlő gombja van: az OK, Cancel és a Help. Ha az OK-t választjuk akkor, ha az aktuális beállítás lesz érvényes, a Cancel esetében semmi sem történik, marad az eredeti beállítás. A Help a dialógus dobozról ad felvilágositást. Az *ESC* billentyű leütése megegyezik a Cancel gomb "megnyomásával" és a dialógus dobozt le is zárja. A dialógus dobozokban a alapértelmezés szerinti gomb az OK, így a *ENTER* leütése az OK választását jelenti.

Az egérrel a megfelelő helyen klikkentünk, klaviatúráról pedig a *TAB* billentyűvel járkálhatunk a menüpontokon előre és a *SHIFT TAB* leütésével hátrafelé. Akkor nyomjuk meg a magasabb intenzitással világító betűt, ha az éppen aktív.

Az opció beállító dobozban egyszerre többet is kiválaszthatunk az egyes lehetőségek közül. Amelyiket kiválasztottuk, ott egy *X* jelenik meg. Ez mutatja a bekapcsolt állapotot. Ahol nincs semmilyen jel, az opció kikapcsolt állapotban van. Egérrel kiválaszthatjuk az egyes opciókat a szövegen, vagy a [] dobozon klikkentve. Klaviatúráról a *TAB* billentyűvel az opció beállító doboz fejlécére lépve a kurzor billentyűvel mozoghatunk az opciók között és a *SPACE* billentyűvel választhatjuk ki az opciókat.

A csoportban az állító gombok közül csak egyet választhatunk ki egyszerre, a kiválasztás módja teljesen azonos az opció beállító dobozéhoz.

Az input dobozban szöveget gépelünk, érvényesek az alap editálási billentyűk, így a kurzor jobbra és balra, *Home*, *End*, *Ins*. Ha a szöveg nem fér a dobozba, akkor görgethető. A → vagy ← nyíl jelenti, ha a szöveg eleje és vége nem látszik az ablakban.

Ha az input dobozhoz tartozik egy lefelé mutató nyíl(↓), az azt jelenti, hogy az input dobozhoz tartozik egy ún. történeti (*history*) lista. A lefelé nyilat megnyomva megjelenik a lista, ahonnan a választás *ENTER*-rel történik. *ESC* billentyűvel választás nélkül léphetünk ki e listából.

A lista dobozból változó hosszúságú szöveget választhatunk ki. A lista doboz akkor aktív, ha a fejléce magasabb fényerővel világít, ezt elérhetjük a *TAB* billentyűvel, vagy egér-klikkentéssel. A listán a kurzor billentyűkkel mozoghatunk.

Az 1.1 táblázat foglalja össze a leggyakrabban használt ún. *hot key*-ket, a leütésük hatása azonnal jelentkezik, általuk pl. menüpontot vagy parancsot közvetlenül végrehajthatunk a Turbo Pascal-ban.

1.1 Táblázat

Általános hot key-k

Billentyű	Menüpont	Funkció
F1	**H**elp	Help információt jelenít meg egy ablakban.
F2	File|**S**ave	Elmenti az aktív szerkesztő ablakban lévő file-t.
F3	File|**O**pen	File megnyitáshoz egy dialógus dobozt jelenít meg.
F4	Run|**G**oto Cursor	Futtatja a programot addig a sorig, ahová a kurzort pozícionáltuk.
F5	Window|**Z**oom	Kinagyítja az aktív ablakot.
F6	Window|**N**ext	Lépked a nyitott ablakokon.
F7	Run|**T**race Into	Hibakeresés üzemmódban futtatja a programot, az eljárások utasításait is.
F8	Run|**S**tep Over	Hibakeresés üzemmódban futtatja a programot, de az eljárásokat egy lépésben hajtja végre.
F9	Compile|**M**ake	Fordítja és szerkeszti a programot.
F10		A főmenübe lép.

A menükre vonatkozó hot key-k

Billentyűk	Menüpont	Funkció
ALT SPACE	≡ menü	Kiválasztja a ≡ (System) menüt.
ALT-C	**C**ompile menü	Kiválasztja a Compile menüt.
ALT-D	**D**ebug menü	Kiválasztja a Debug menüt.
ALT-E	**E**dit menü	Kiválasztja az Edit menüt.
ALT-F	**F**ile menü	Kiválasztja a File menüt.
ALT-H	**H**elp menü	Kiválasztja a Help menüt.
ALT-O	**O**ption menü	Kiválasztja az Option menüt.
ALT-R	**R**un menü	Kiválasztja a Run menüt.
ALT-S	**S**earch menü	Kiválasztja a Search menüt.
ALT-W	**W**indow menü	Kiválasztja a Window menüt.
ALT-X	File|E**x**it	Kilép a DOS-ba a Turbo Pascal-ból.

Az editorra vonatkozó hot key-k

Billentyűk	Menüpont	Funkció
Ctrl-Del	Edit|**Clear**	Törli a kiválasztott szöveget az ablakból, nem teszi be a *Clipboard*-ba.
Ctrl-Ins	Edit|**Copy**	Bemásolja a szöveget a *Clipboard*-ba.
Shift-Del	Edit|**Cut**	Törli a kijelölt szövegblokkot az aktív editáló ablakból és elhelyezi azt a *Clipboard*-on (ahol az kiválasztva marad).
Shift-Ins	Edit|**Paste**	Kimásolja a szöveget a *Clipboard*-ról az aktív ablakba.
Ctrl-L	Search|**Search Again**	Ismétli az utolsó Find vagy Replace parancsot.
F2	File|**Save**	Elmenti a file-t az aktív ablakból.
F3	File|**Open**	File nyitása betöltésre.

Az ablak (window) kezelésére vonatkozó hot key-k

Billentyűk	Menüpont	Funkció
Alt-#		A # sorszámú ablakot aktivizálja. (# = 1 .. 9)
ALT-0	Window|**List**	A nyitott ablakok listáját jeleníti meg.
ALT-F3	Window|**Close**	Lezárja az aktív ablakot.
ALT-F5	Window|**User Screen**	Megjeleníti a felhasználói képernyőt.
Shift-F6	Window|**Previous**	Visszafelé mozog az aktív ablakokon.
F5	Window|**Zoom**	Felnagyítja/kicsinyíti az aktív ablakot.
F6	Window|**Next**	Előrefelé mozog az aktív ablakokon.
Ctrl-F5	Window|**Size/Move**	Változtatja az ablak méretét és pozícióját.

Közvetlen Help-re vonatkozó hot key-k

Billentyűk	Menüpont	Funkció
F1	Help|Contents	Megnyitja a szövegkörnyezettől függő help-ablakot.
F1 F1	Help|Help on Help	A help-rendszerre vonatkozó help-et jeleníti meg.
Shift-F1	Help|Index	Megjeleníti a help-indexet.
ALT-F1	Help|Previous Topic	Megjeleníti az előző tárgyra vonatkozó help-et.
Ctrl-F1	Help|Topic Search	Adott beépített nyelvi elemre vonatkozó help (csak aktív editáló ablak esetén él; a kurzornál lévő szóra vonatkozik).

A hibakeresésre és futtatásra vonatkozó hot key-k

Billentyűk	Menüpont	Funkció
ALT-F9	Compile|Compile	Az Editor aktív ablakában lévő file fordítása.
Ctrl-F2	Run|Program Reset	A futó programot alapállapotba álállítja.
Ctrl-F4	Debug|Evaluate/Modify	Kiszámít egy kifejezést.
Ctrl-F7	Debug|Add Watch	Újabb kifejezést ad a Watch ablakhoz.
Ctrl-F8	Debug|Toggle Breapoint	Beállítja vagy törli a feltételes megállási pontokat.
Ctrl-F9	Run|Run	Futtatja a programot.
F4	Run|Go to Cursor	Futtatja a programot a kurzor által kijelölt helyig.
F7	Run|Trace Into	Az eljárást is hibakeresés üzemmódban hajtja végre.
F8	Run|Step Over	Az eljárást egy lépésben hajtja végre.
F9	Compile|Make	Újra felépíti a programot (Fordítás, szerkesztés).

2. TURBO PASCAL 6.0 INTEGRÁLT FEJLESZTŐI KÖRNYEZET (IDE)

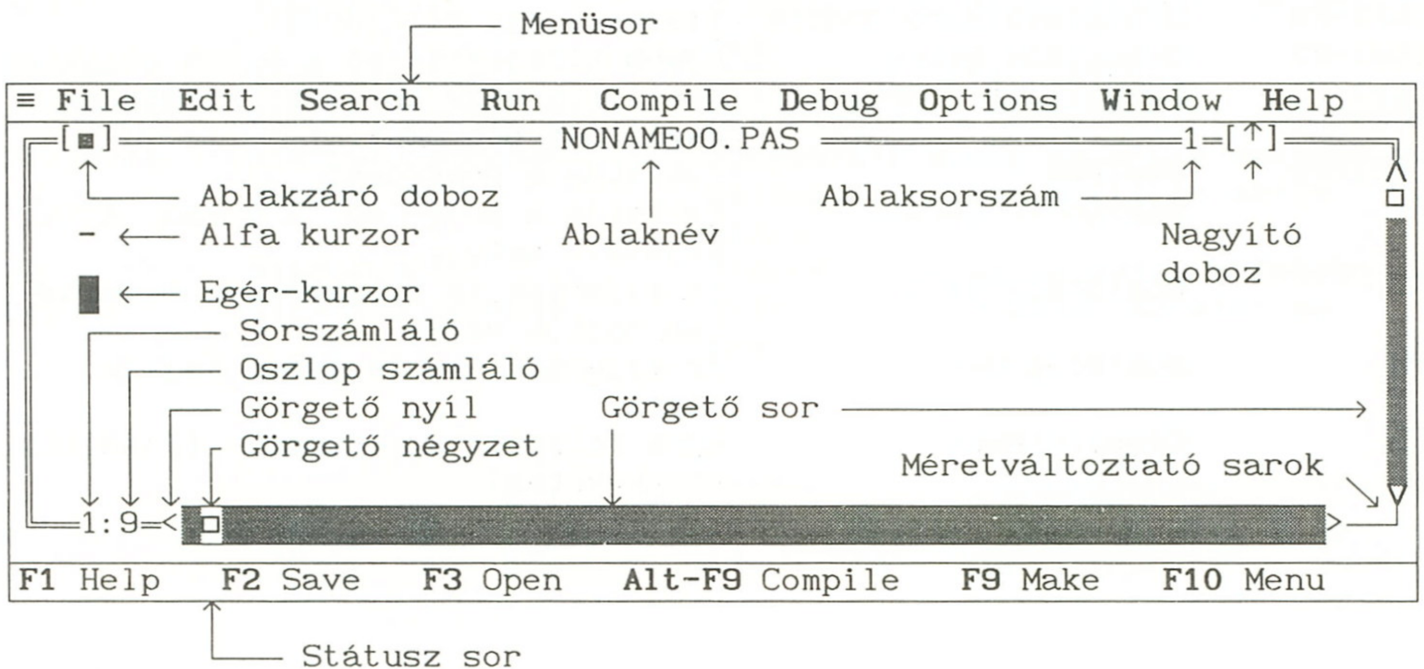
2.1. Az IDE kezelésének általános szabályai és a menürendszer bemutatása

Nagyon könnyen és hatékonyan lehet programot írni, javítani, fordítani, szerkeszteni és hibát keresni Turbo Pascal-ban. Ezt még tovább fokozza a Turbo Pascal 6.0 integrált fejlesztői rendszere (**IDE**), amely ablakozós (*window*) technikával működik. A Turbo Pascal igen hatékony Pascal fordítóprogram, amelyet hamar meg lehet tanulni és könnyen lehet használni ebben a fejlesztői környezetben.

Az Turbo Pascal **IDE** támogatja

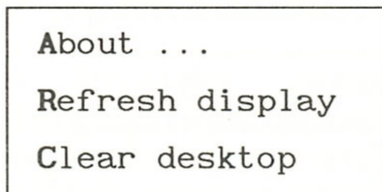
- a többszörös, mozgatható ablakkezelést,
- az egér használatát,
- a file-ok többszörös editálását 1 Mbyte méretig,
- a dialógus dobozokat,
- kivágó és másoló parancsokat (a másolást megengedve a *Help* ablakból az *Edit* ablakba),
- a kereső és újrahelyező lehetőségeket,
- kiiratási lehetőségeket,
- az editor macro nyelvét.

Indítsuk el a Turbo Pascal-t a
turbo
paranccsal. Ekkor a program az alábbi menüvel jelentkezik



2.1.1. Az ún. (*System*) menü: ≡

A főmenü első menütétele három általános, a teljes rendszerre vonatkozó almenüt tartalmaz:



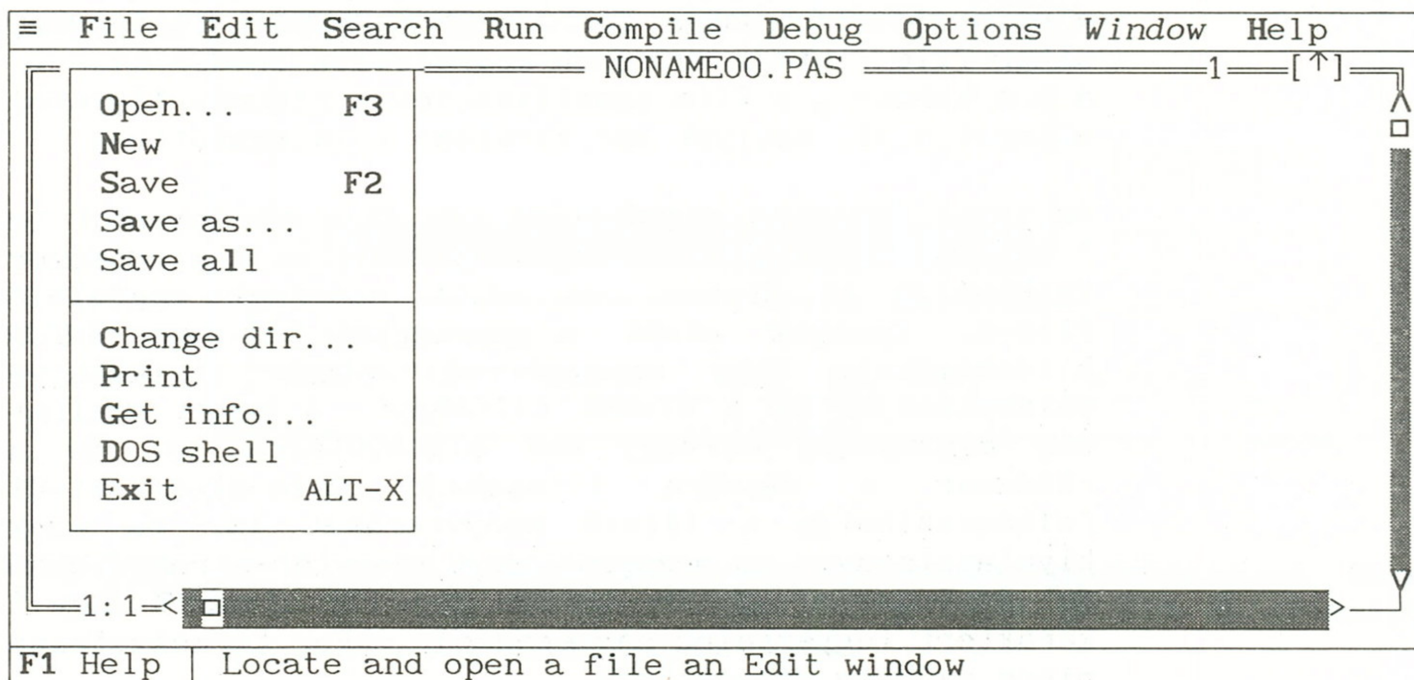
About dialógus dobozban közli a Turbo Pascal verziószámát. Az *ESC* vagy a *SPACE* vagy egér **OK** klikkentés (vagy *ENTER* leütése) lezárja a dialógus dobozt.

Refresh Display újragenerálja az **IDE** képernyőjét.

Clear Desktop lezárja az összes ablakot és a törli a dialógus dobozok (*history*) listáit.

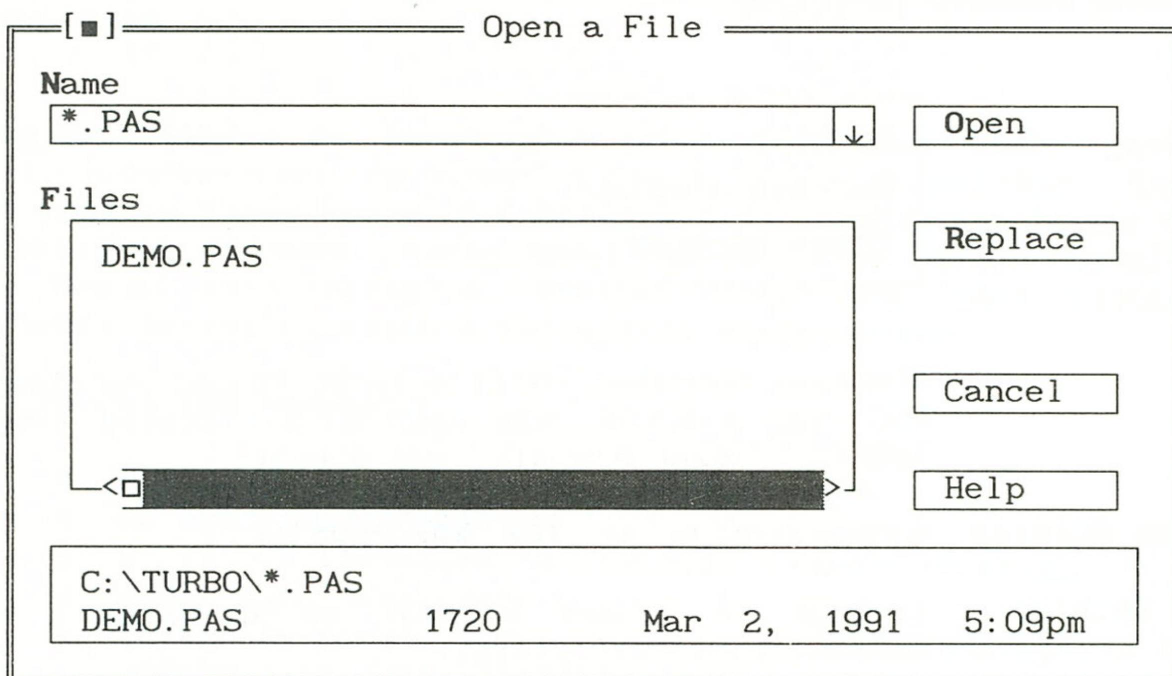
2.1.2. A *File* menü

ALT-F a **File** menübe lép, ahol az Edit ablakban megnyithatunk és létrehozhatunk program file-okat, elmenthetjük a változtatásokat, végrehajthatunk más file funkciókat, kiléphetünk DOS-ba (*shell*), vagy kiléphetünk az **IDE**-ből (*quit*).



Open

File|Open parancs (*F3*) segítségével nyithatjuk meg a file kiválasztó dialógus dobozt, amelyen keresztül a kiválasztott file az *EDIT* ablakba kerül.



A dialógus doboz tartalmaz egy ún. input (beolvasó) dobozt, file-ok listáját, *Open*, *Replace*, *Cancel* és *Help* funkciójú vezérlő gombokat, valamint egy információs mezőt, amely a kiválasztott file adatait jeleníti meg.

■ Ha teljes file nevet adunk meg és a *Open* funckiót választjuk, akkor a file egy új Edit ablakba töltődik be. A *Replace* választása esetén az Edit ablak tartalma a kiválasztott file tartalmára cserélődik ki.

■ Ha a file név joker (wildcard) karaktert tartalmaz, akkor file lista a szűrőnek megfelelően íródik ki.

■ A a kurzor ↓ a file specifikációt a history listából választja ki, amelyet már korábban kiválasztottunk.

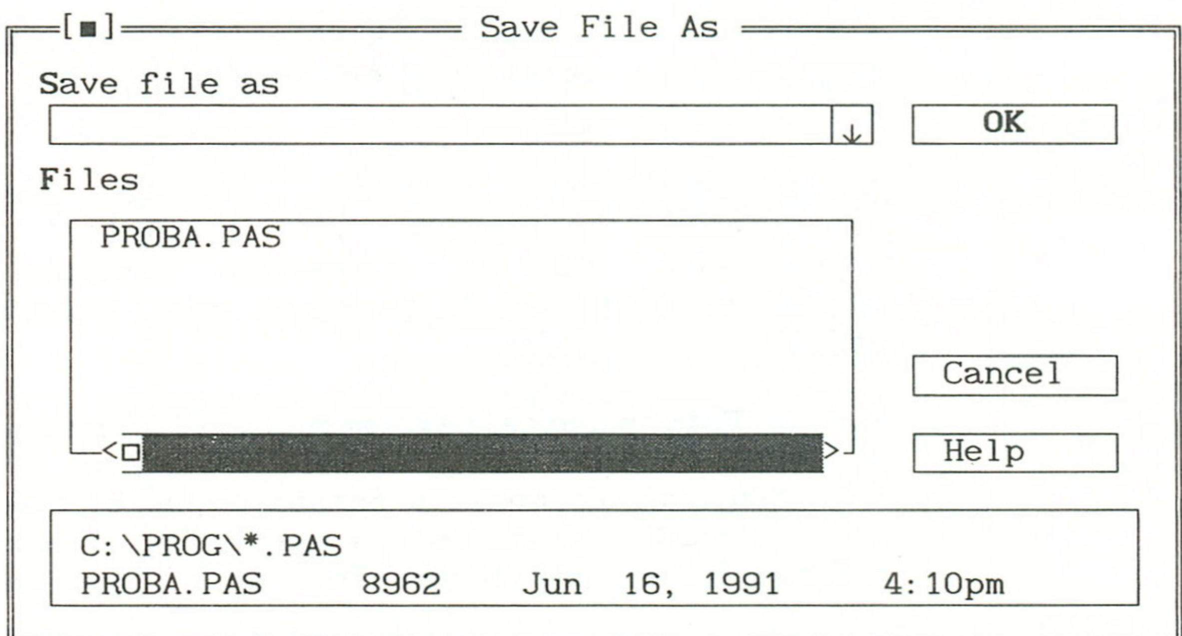
Az input dobozban megadhatunk egy file név maszkot (a * és ? joker karakterek segítségével), majd a rendszer felkínálja az összes, az adott maszknak megfelelő file-t, amelyek közül a kurzorral és az *ENTER* billentyűvel, vagy az egérrel kétszer klikkentve választhatjuk ki a kívánt állományt. A maszk utalhat más meghajtóra és/vagy más alkönyvtárra is, de a rendszer a maszkra illeszkedő file-okon kívül felhasználhatja a létező könyvtárakat is, és azok kiválasztásával is mozoghatunk a könyvtár-struktúrában fel és le egyaránt. Ha a maszk nem tartalmaz ? vagy * karaktert (egyértelmű névmegadás), akkor természetesen nincs szükség választásra.

Ha az input dobozban megnyomjuk a lefelé nyíl billentyűt, akkor egy görgethető listában jelennek meg azok a file-nevek, amelyeket már előzőleg kiválasztottunk és ezekből újraválaszthatunk az egérrel kétszer klikkentve ill. a kurzorral rápozícionálva majd az *ENTER* billentyűt leütve.
A *TAB* billyentyűvel lépkedhetünk a dobozok és az utasítások között előre és a *SHIFT TAB*-bal visszafelé.

New a File|**N**ew parancs egy új ablakot nyit meg **NONAMExx.PAS** névvel (az xx 00 és 90 közötti szám, egyenlőre még névtelen forrásállomány sorszáma). A NONAME file-ok ideiglenes EDIT file-ként használhatók. Bármelyik mentést választva az **IDE** automatikusan megkérdezi, hogy a NONAME file-t milyen néven mentse el.

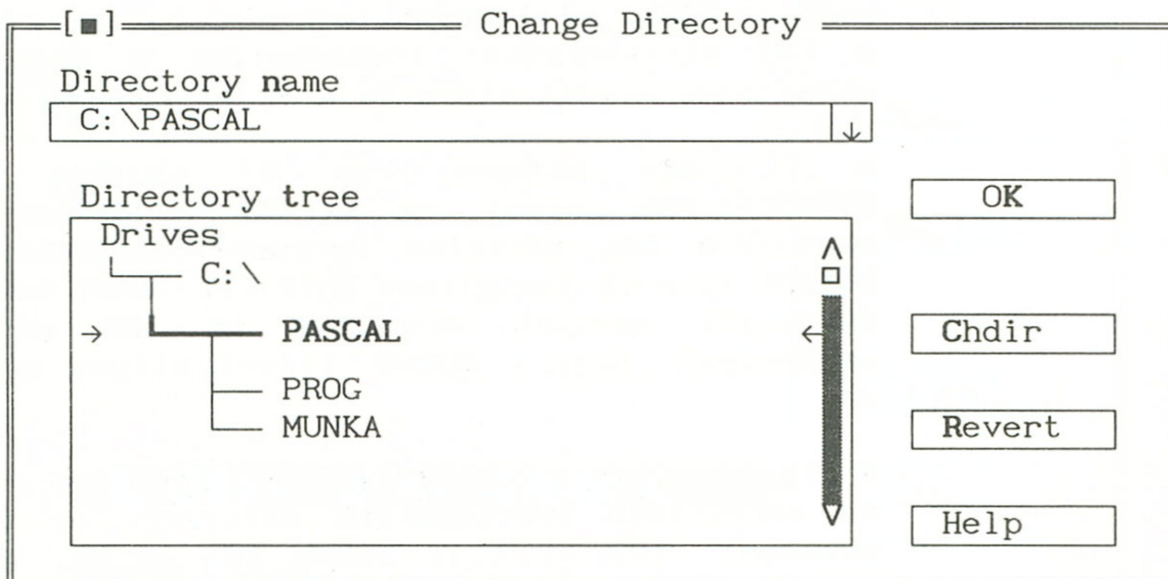
Save a File|**S**ave parancs (*F2*) segítségével diszkre írhatjuk az aktuálisan szerkesztett állományt az aktív EDIT ablakból. (Ha az EDIT ablak nem aktív, akkor ez a mentési funkció elmarad.)

Save As a File|S**a**ve As parancs lehetővé teszi, hogy a file-t az aktív ablakból más néven, más alkönyvtárba vagy más lemezegységre kiírjuk *ENTER*-rel, az **OK** választásával, vagy az egér klikkentésével.



Save All a File|Save A**l**l parancs abban különbözik a **Save** parancstól, hogy az összes módosított file-t elmenti, nemcsak az ablakban lévőt.

Change Dir a **File|Change** Dir paranccsal lemezegységet és alkönyvtárat válthatunk és a választott lesz az aktuális.



Print a **File|Print** parancs az aktív ablak tartalmát sornyomtatóra írja ki. Ha az ablak nem aktív, a menüpont aktiválása hatástalan. A kiválasztott szöveg sornyomtatóra való kiiratása a *CTRL-K P* editor parancs hatására is megtörténik.

Get Info a **File|Get** Info parancs kiválasztásával megtudhatjuk, hogy az **IDE** a számítógép erőforrásaiból mit és mennyit vesz igénybe.

DOS Shell a **File|Dos** Shell parancs lehetővé teszi, hogy ideiglenesen elhagyjuk az integrált fejlesztői környezetet, és az operációs rendszer parancsértelmező felületének (command shell) adjunk parancsokat.

Exit a **File|Exit** parancs segítségével véglegesen elhagyjuk a Turbo Pascal fejlesztői rendszert. Az *ALT-X* megnyomásával ugyanezt a hatást érjük el. Ha bármilyen változtatást tettünk és nem mentettük el azokat, akkor a Turbo Pascal a kilépés előtt erre figyelmeztet.

2.1.3. Az *Edit* (szerkesztés) menü

Az *ALT-E* az **E**dit menüre vált. Itt történik a szöveg szerkesztése, kivágása és másolása. A menü legtöbb parancsának kiadása előtt a szöveget ki kell jelölni. A kiválasztott szöveg magasabb fényerővel világít. A kijelölés történhet klaviatúráról vagy egérrel.

Klaviatúráról

- *SHIFT* billentyű mellett bármelyik nyíl (föl, le, balra, jobbra)
- a *CTRL-K B* a kijelölendő szövege eleje, *CTRL-K K* a kijelölendő szöveg vége,
- egy szó kijelölése *CTRL-K T*
- sor kijelölése *CTRL-K L*

Egérrel

- egyszeri klikkentés kijelöli a szöveg elejét, a gombot nyomva tartva az egér mozgatásával kijelöljük a további részeket, a gomb elengedésével kijelöljük a végét,
- egy sor kijelölése az egér kétszeri klikkentésével
- kettőt klikkentve az egérrel és a gombot nyomva tartva több sort is kijelölhetünk az egér mozgatásával,
- a kijelölt szöveget módosíthatjuk a *SHIFT* billentyű és az egér klikkentésével.

Ha a szöveget már kijelöltük, akkor használhatjuk az Edit menü parancsait és a *Clipboard* is használhatóvá válik.

Restore Line — az **E**dit|**R**estore Line parancs segítségével az utoljára módosított vagy törölt sor tartalma visszállítható.

Cut — az **E**dit|**C**ut paranccsal (*SHIFT-Del*) az aktív editáló ablakból a kiválasztott szövegblokk a szövegből a *Clipboard* végéhez másolódik, innen a *Paste* kiválasztásával teszőleges helyre többször is beültethetjük.

Copy — az **E**dit|**C**opy parancs (*CTRL-Ins*) az aktív editáló ablak kijelölt szövegblokkját érintetlenül hagyja és elhelyezi azt a *Clipboard*-ban. A *Paste* kiválasztásával a szöveg tetszőleges helyre beültethető. A Help ablakból is másolhatunk: a *SHIFT* és a nyilak használatával a klaviatúráról, vagy egérrel klikkentéssel.

Paste — az **E**dit|**P**aste paranccsal (*CTRL-Ins*) a *Clipboard* kijelölt blokkját az aktív editáló ablak alfanumerikus kurzorjához másolja.

Copy Example — az **E**dit|Copy **E**xample parancs egy Pascal nyelvi elem helpjének blokként kijelölt mintaprogramját helyezi el a *Clipboard*-on.

Show Clipboard az Edit|Show Clipboard parancs megnyitja a *Clipboard* ablakot, amely folytatólagos file-ként tárolja már a kivágott szövegrészeket és az egyik ablakból a másik ablakba másolt szövegrészeket. Az aktuálisan kiválasztott szöveg magasabb fényerővel világít. Így az eddigi blokkmásolataink "történetét" nézhetjük meg.

Clear az Edit|Clear parancs (*CTRL-Del*) törli a kiválasztott szövegblokkot és nem helyezi el a *Clipoard*-ba. Ezzel az paranccsal a *Clipboard*-ban is törölhetünk.

2.1.4. A *Search* (keresés) menü

A **S**earch menü (*ALT-S*) lehetővé teszi szöveg, eljárás deklaráció, vagy hibahelyek keresését a file-ban.

Find a Search|**F**ind parancs (*ALT-S F*) megjeleníti a Find dialógus dobozt, amely segítségével megadhatjuk a keresett szöveget és opciókat állíthatunk be a szöveg keresésére.

```
[■]═══════════════ Find ═══════════════
 Text to Find  [                              ] [↓]

 Options                        Direction
 [ ] Case sensitive             (•) Forward
 [ ] Whole words only           ( ) Backward
 [ ] Regular expression

 Scope                          Origin
 (•) Global                     ( ) From cursor
 ( ) Selected text              (•) Entire scope

            [ OK ]      [ Cancel ]   [ Help ]
```

Az *Option* részben beállíthatjuk a *Case sensitive* megjelölésével, hogy a Turbo Pascal különbséget tegyen a kisbetű és a nagybetű között. A *Whole words only* dobozban jelezhetjük, hogy csak egy teljes szót keresünk, melynek mindkét oldalán írásjel, vagy üres hely (space) van.
A *Regular Expression* doboz megjelölésével lehetőség

van szövegben a ^, $, ., *, +, [,] karakterek keresésére.
A *Direction* dobozban megválaszthatjuk a keresés irányát: előre (*Forward*) vagy hátra (*Backward*).

```
Direction
(•) Forward
( ) Backward
```

A *Scope* beállításával kérhetjük a keresést a teljes file-ra (*Global*), vagy csak a kijelölt szövegre (*Selected text*).

```
Scope
(•) Global
( ) Selected text
```

Az *Origin* segítségével megjelölhetjük, hogy a keresés hol kezdődik. A kurzor aktuális helyzetétől (*From Cursor*) kezdődhet a keresés. Az *Entire Scope* választása esetén a keresés az egész szövegre vonatkozik, a *Direction* dobozban kiválasztott irány szerint.

```
Origin
( ) From Cursor
(•) Entire Scope
```

Replace — a **Search|Replace** parancs (*ALT-S R*) megjeleníti dialógus dobozokat, egyikben meg kell adni a keresendő szöveget (*Text to find*), a másikban pedig amire cserélni kell (*New text*). (*CTRL-Q A*) rövid paranccsal is végrehajthatjuk.

Hasonlóan a Find dialógus dobozhoz a cserékhez is lehet opciókat megadni. A *Prompt on replace* bekapcsolása esetén minden cserénél rákérdez a művelet végrehajtására.

Miután megadtuk a szöveget keresésre a ill. a cserére, az **OK**, vagy **Change all** kiválasztására megindul a keresés, vagy **Cancel** esetén a parancs törlődik.

Search Again — a **Search|Search** Again parancs (*CTRL-L*) ismétli az utolsó Find/Replace parancsot. Minden változatlan marad, amit utoljára beállítottunk a Find vagy Replace parancs számára.

```
[■]═══════════════ Replace ═══════════════
   Text to Find  [                              ] [↓]

       New text  [                              ] [↓]

 Options                        Direction
 [ ]  Case sensitive            (•)  Forward
 [ ]  Whole words only          ( )  Backward
 [ ]  Regular expression
 [x]  Prompt on replace

 Scope
 (•)  Global                    ( )  From cursor
 ( )  Selected text             (•)  Entire scope

   OK      Change all     Cancel      Help
```

Go to Line Number a Search|Goto Line Number parancs a megadott sorszámú utasításra ugrik.

```
[■]══════════ Go to Line Number ══════════
 Enter new line number [ 1                  ] [↓]

    OK            Cancel            Help
```

Find Procedure a Search|Find Procedure parancs megkeresi a dialógus dobozban megadott eljárást vagy függvényt.

```
[■]══════════ Find Procedure ══════════
 Procedure name
 [                                        ] [↓]

    OK            Cancel            Help
```

Find Error a Search|Find Error parancs megkeresi (*ALT-F8*) a futtatási idő alatt előforduló hibákat.

2.1.5. A *Run* (futtatás) menü

A *Run* menü (*ALT-R*) parancsai futtatják a programot és vezérlik a hibakeresést.

Run — a **Run|Run** parancs (*CTRL-F9*) futtatja az aktuális programot, a programnak paramétert a **Run|Parameters** paranccsal adhatunk. Ha az a utolsó fordítás óta módosítás történt a programon, akkor automatikusan megtörténik a fordítás és a szerkesztés is.

A *CTRL-Break* hatására a Turbo Pascal befejezi a program futtatását, ha nem találja a program következő forrás sorát, akkor a következő *CTRL-Break* már véglegesen leállítja a programot és a vezérlést visszakapja az IDE.

Program Reset — a **Run|Program** Reset parancs (*CTRL-F2*) hatására a nyomkövetés befejeződik, a program által lefoglalt memória felszabadul, a megnyitott file-okat lezárja.

Go to Cursor — a **Run|Goto** Cursor (*F4*) segítségével nyomkövetési állapotba kapcsolunk - ha nem voltunk még ott -, és egyidejűleg töréspontot helyezünk el a kurzor által mutatott utasításra, majd futtatjuk a programunkat.

Trace Into — a **Run|Trace** Into parancs *(F7)* utasításonként hajtja végre a programot, eljáráshíváshoz érve az eljárás minden utasítását is lépésenként elvégzi.

Step Over — a **Run|Step** Over parancs (*F8*) a függvényhívásokat egy egységként kezelve hajtja végre.

Parameters — a **Run|Parameters** parancshoz tartozó dialógus dobozban a futtatandó program parancs-sor paramétereit állíthatjuk be (mintha a programot DOS-ból hívnánk meg; egyedül az input/output állományok átirányítását nem alkalmazhatjuk).

```
[■]═══════════ Program Parameters ═══════════
  Parameter   [                              ] [↓]

              [  OK  ]   [ Cancel ]   [ Help ]
```

2.1.6. A *Compile* (fordítás) menü

Ez a menüpont (*ALT-C*) tartalmazza a fordítással és szerkesztéssel kapcsolatos parancsokat.

Compile — a Compile|Compile parancs (*ALT-F9*) segítségével kérhetjük az aktuális forrás file fordítását linkelés nélkül.

Make — a Compile|Make parancs (*F9*) aktiválja a beépített project vezérlőt, amely elkészíti a *.EXE* file-t. Ha a Primary File nevét megadtuk, akkor file fordítása megkezdődik, vagy a file-nak az aktív Edit ablakban kell lennie a fordításhoz. A Turbo Pascal ellenőrzi az összes file-t, amely a fordításhoz még szükséges. A fordítást az egyes file-ok keletkezési ideje szerint végzi el a Turbo Pascal.

Build — a Compile|Build parancs újrafordítja a file-okat, nem veszi figyelembe a keletkezés dátumát.

Destination — a Compile|Destination parancs lehetőséget ad arra, hogy a végrehajtható program (.EXE file) diszken keletkezzék, vagy a memóriában (elveszik a fordítás eredménye, ha kilépünk a Turbo Pascal-ból).

Primary File — parancs segítségével adjuk meg, hogy melyik .PAS file kerüljön fordításra a *Make* vagy *Build* parancs hatására. Akkor szükséges megadni, ha programunk több unit-ból és include file-ból áll. Ilyenkor akármelyiket javítjuk éppen, kiadhatjuk a fordítási parancsot.

2.1.7. A *Debug* menü

A Debug menü (*ALT-D*) vezérli az integrált hibakereső rendszer összes lehetőségét.

Evaluate/Modify — a Debug|Evaluate/Modify parancs (*CTRL-F4*) hatására a megjelenő ablak *Expression* ablakába beírhatjuk a vizsgálni kívánt változót és kifejezést, a *Result* ablakban jelenik meg az eredmény és a *New value* ablakban írhatjuk be az új értéket, amely felülírja az aktuálisat.

```
[■]═══════════ Evaluate and Modify ═══════════
 Expression
 [                                    ] [↓]  [Evaluate]
 Result
 [                                    ]      [ Modify ]
 New value
 [                                    ] [↓]  [ Cancel ]

                                             [  Help  ]
```

Watches a Debug|Watches parancs nyitja meg a parancsok almenüjét, amely kezeli a figyelni kívánt pontokat. Ezen almenü segítségével kezelhetjük a nyomkövetés során folyamatosan kijelzendő változókat.

```
Add watch ...          Ctrl-F7
Delete watch
Edit watch ...
Remove all watches
```

Add Watch a Add Watch parancs (*CTRL-F7*) lehetőséget ad újabb változók és kifejezések felvételére a megfigyelési ablakba. A watch listára felvett változók aktuális értékét folyamatosan kijelzi a nyomkövetési rendszer. A kiválasztás ugyanúgy történik, mind az Evaluate parancsnál (begépeléssel vagy kurzorral). Az új elem az eddigiek alá kerül. Ha az ablak aljára érünk, az újabb kifejezések bevitelekor mindig a felső elem kilép az ablakból. A ablakból kilépett elemek az ablak visszalapozásával vagy az ablak megnagyobbításával újra elérhetővé válnak. Ha a Watch ablak aktív, akkor újabb kifejezést adhatunk az ablakhoz az *Ins* billentyű leütésével.

Delete Watch kiválasztásával a Watch ablakból törölhetjük az aktuális kifejezéseket, de törölhetjük a *Del* vagy a *CTRL-Y* billentyűk leütésével is.

Edit Watch parancs lehetővé teszi a Watch ablakban az aktuális kifejezést módosítani. A módosítás az *Enter* billentyűvel érvényesíthető.

Remove All Watches minden elemet töröl a Watch ablakból.

Toggle Breakpoint **D**ebug|**T**oggle Breakpoint parancs (*CTRL-F8*) beállít vagy töröl töréspontot abban a sorban, ahol a kurzor áll.

Breakpoints a **D**ebug|**B**reakpoint egy dialógus dobozt nyit meg a töréspontok ellenőrzésére.

```
[■]══════════ Breakpoints ══════════
  Breakpoint list    Line # Condition              Pass

  [ OK ] [ Edit ] [ Delete ] [ View ] [ Clear all ] [ Help ]
```

A dialógus doboz mutatja az összes töréspontot a sorszámukkal, feltételükkel. A töréspont csak a hibakeresés alatt létezik, ha a program végrehajtása a töréspontot eléri, a program mindig leáll, lehetőséget adva a programozónak a beavatkozásra.

Edit új töréspont hozzáadása
Delete töréspont törlése
View töréspont aktivizálása

Az **E**dit választás hatására az alábbi dialógus doboz jelenik meg:

```
[■]══════════ Edit Breakpoint ══════════
  Condition
  [                              ] [↓]  [ Modify ]
  Pass count
  [                              ]      [  New   ]
  File name
  [                              ]      [ Cancel ]
  Line number
  [                              ]      [  Help  ]
```

A *Pass Count* mellett megadhatjuk, hogy hányszor fusson le a program, mielőtt megáll a törésponton.
A töréspont csak az akkor törlődik, ha

- a **B**reakpoints dialógus dobozban töröljük a töréspontot,
- töröljük a programnak azt a sorát, amely töréspontot tartalmazta,
- **T**oggle Breakpoint segítségével töröljük a töréspontot.

2.1.8. Az *Option* (rendszerjellemzők beállítása) menü

Kiterjedt lehetőségekkel rendelkező menüpont (*ALT-O*).

Compiler az Option|Compiler parancs dialógus dobozában olyan opciókat állíthatunk be, amelyek szerepet játszanak a program fordításánál.

```
[■]══════════ Compiler Options ══════════
Code generation
 [ ] Force far calls        [x] Word align data
 [ ] Overlays allowed       [ ] 286 instructions

Runtime errors              Syntax options
 [ ] Range checking          [x] Strict var-strings
 [x] Stack checking          [ ] Complete boolean eval
 [x] I/O checking            [ ] Extended syntax

Numeric processing          Debugging
 [ ] 8087/80287              [x] Debug information
 [x] Emulation               [x] Local symbols

Conditional defines
 [                                              ]

              [ OK ]      [ Cancel ]    [ Help ]
```

A dialógus doboz parancsai:

Code Generation dobozban az alábbiakat írhatjuk elő a fordító program számára

- *Force far calls* lehetővé teszi az összes eljárásnak és függvénynek a *far* hívási modell használatát. Alapértelmezés a hívási modell *near*. (Ez azonos a **$F** fordítási direktívával.)

- *Overlays allowed* bekapcsolva olyan kód készül, amely lehetővé teszi az overlay technika alkalmazását. (Ez azonos a **$O** fordítási direktívával.)

- *Word align data* bekapcsolva utasítja a Turbo Pascal-t, hogy minden változó gépi szó határra legyen igazítva. Kikapcsolt állapotban byte-ra állítva nem történik ilyen kiigazítás, (Ez azonos a **$A** fordítási direktívával.)

- *286 instructions* bekapcsolva a Turbo Pascal 80286 utasítás kódot generál, azonban a futásnál a program nem ellenőrzi, hogy az aktuális CPU 80286-os processzor-e. (Ez azonos a **$G** fordítási direktívával.)

Run time Errors dobozban kiválaszthatók, hogy milyen futási hibáknál kapjunk hibajelzést.

- *Range Checking* bekapcsolt állapotában ellenőrzi, hogy a különböző változók (pl. tömb indexek) nem lépik-e túl a deklarációjuk által számukra megengedett értékhatárokat.
(Ez azonos a **$R** fordítási direktívával.)

- *Stack Checking* bekapcsolt állapotban minden alprogram hívása előtt ellenőrzi a program, hogy van-e elegendő hely a stack-ben a lokális változók számára.
(Ez azonos a **$S** fordítási direktívával.)

- *I/O Checking* bekapcsolt állapotban olyan program készül, amely futás közben ellenőrzi, hogy minden I/O művelet sikeresen zajlott-e le.
(Ez azonos a **$I** fordítási direktívával.)

Syntax Options dobozban a szintaxis ellenőrzésre vonatkozó opciókat állíthatjuk.

- *Strict var-strings* beállítva jelzés történik akkor, ha egy alprogram paraméterlistáján **var**-ként szereplő sztring változó típusa nem azonos a hívó paraméter típusával.
(Ez azonos a **$V** fordítási direktívával.)

- *Complete boolean evaluation* bekapcsolva azt jelenti, hogy minden kifejezés ki lesz értékelve. Kikapcsolt állapotban a logikai kifejezéssel olyan gyorsan végez, ahogy csak lehet. Például, ha egy **AND** kifejezés egyik oldalán hamis értéket talált, akkor a másik oldalt már ki sem értékeli.
(Ez azonos a **$B** fordítási direktívával.)

- *Extended Syntax* bekapcsolva a Turbo Pascal szintaxisa megengedi a felhasználónak, hogy felhasználó által készített függvényt hívjon, mint utasítást.
(Ez azonos a **$X** fordítási direktívával.)

Numeric Processing lehetővé teszi a lebegőpontos számok kezelésére vontkozó opció választását.

```
Numeric Processing
 [ ]  8087/80287
 [x]  Emulation
```

- *8087/80287* bekapcsolása közvetlen 8087 vagy 80287 kódot generál.
(Ez azonos a **$N** fordítási direktívával.)

- *Emulation* opciót választva a Turbo Pascal érzékeli a 80x87 társ-processzor jelenlétét. Ha nincs társ-processzor, akkor emulálja annak működését.
(Ez azonos a **$E** fordítási direktívával.)

Debugging dobozban lehet bekapcsolni vagy kikapcsolni a hibakeresési lehetőséget és a lokális változók generálását.

- *Debug information* bekapcsolva lehetővé teszi a felhasználó számára a hibakeresést. Mivel a hibakeresés megnöveli a program méretét és futási idejét, ezért a hiba megtalálása után a programot érdemes debug nélkül fordítani.
(Ez azonos a **$D** fordítási direktívával.)

- *Local symbols* bekapcsol állapotában debug - információ keletkezik a lokális változókra vonatkozóan.
(Ez azonos a **$L** fordítási direktívával.)

Conditional Defines dialógus dobozban adhatjuk meg a feltételes fordítási direktívákat, melyeket ; (pontosvesszővel) kell elválasztani, pl.:

TestCode;DebugCode

Memory Sizes opció segítségével konfigurálható az alapértelmezés szerinti memória a program számára.
(Ez azonos a **$M** fordítási direktívával.)

- *Stack Size* a stack méretét állítja be, melyet byte-okban kell megadni. Az alapértelmezés szerint a stack mérete: 16384 byte, a maximum 65520 byte lehet.

- *Low Heap Limit* segítségével a minimális *heap* méretét állíthatjuk be byte-ban. Az alapértelmezés szerint a *heap* mérete 0 byte.

- *High Heap Limit* a *heap* maximális méretének beállítását végzi. Az alapértelmezés szerint 655360 byte, amely azt jelenti, hogy a *heap*-hez egy teljes 640 Kbyte-os memóriaszegmens lesz hozzárendelve.

Linker parancs dialógus dobozában a program szerkesztésével kapcsolatos információkat állíthatjuk be.

```
[■]═══════════════ Linker ═══════════════════
 Map file                      Link buffer
 ┌──────────────────┐          ┌──────────────┐
 │ (•) Off          │          │ (•) Memory   │
 │ ( ) Segments     │          │ ( ) Disk     │
 │ ( ) Public       │          └──────────────┘
 │ ( ) Detailed     │
 └──────────────────┘

 [   OK   ]        [ Cancel ]        [  Help  ]
```

Mape file dobozban kiválaszhatjuk, hogy milyen típusú map (térkép) file keletkezzen a szerkesztés folyamán. Alapértelmezés szerint nem készül map file.
Ha kikapcsoljuk az *Off* gombját, akkor a map file abban a alkönyvtárban keletkezik, amelyet az *Option*I *Directories* dialógus dobozában adtunk meg.

A *Segments*, *Publics* és a *Detailed* megegyezik a /**GS**, /**GP** és /**GD** parancs-sor opcióval.

Link Buffer kijelölhető memóriára vagy diszkre. Memória esetén a fordítás gyorsabban megy végbe, de nagy program esetén *Out of memory* hibát kaphatunk. Ilyenkor át kell kapcsolni a linkelési buffert diszkre, a memória felszabadul, a fordítás lassabban megy végbe (ez ekvivalens a /**L** parancs-sor opcióval).

Debugger parancs nyitja meg azt a dialógus dobozt, amelyben különböző beállításokat végezhetünk az integrált hibakereséssel kapcsolatban.

```
[■]═══════════════ Debugger ═════════════════
 Debugging                     Display swapping
 ┌────────────────────┐        ┌──────────────────┐
 │ [x] Integrated     │        │ ( ) None         │
 │ [ ] Standalone     │        │ (•) Smart        │
 └────────────────────┘        │ ( ) Always       │
                               └──────────────────┘

 [   OK   ]        [ Cancel ]        [  Help  ]
```

Debugging dobozban kétféle választási lehetőség van:

- *Integrated* (ez az alapértelmezés) választása esetén a program hibakeresése az integrált hibakereső környezetben vagy a Turbo Debugger-rel lehetséges.
- *Standalone* bekapcsolásakor a program hibakeresése csak Turbo Debugger-rel történhet.

Display Swapping dobozban bekapcsolható az opció arra vonatkozólag, hogy az integrált hibakereső rendszer hogyan kezelje a megjelenítő ablakot a futtás alatt.

A hibakeresés két monitorral is működhet. Ekkor a **/d** parancs-sor opcióval kell inditani a Turbo Pascal-t. Ilyen esetben a program outputja megjelenik az egyik monitoron, a másikat pedig a Turbo Pascal IDE használja.

- *None* beállítás esetén a Turbo Pascal nem frissíti a képernyőt, ezt akkor használjuk, ha bizonyosak vagyunk abban, hogy a képernyőn nem lesz eredmény.
- *Smart* (alapértelmezés), a debugger figyeli a program működését, ha az ír a képernyőre, akkor vált a IDE képernyőről a felhasználó által használt képernyőre, majd visszavált.
- *Always* beállítása esetén a debugger minden alkalommal felfrissíti a képernyőt, amikor egy utasítást végrehajt. Ezt a beállítást kell választani, amikor a program felülírja az IDE képernyőjét.

Directories parancs dialógus dobozában adjuk meg azt az információt a Turbo Pascal-nak, hogy hol találja a forrás file-(oka)t, hová történjen a linkelés és hol keletkezzenek az output file-ok.

```
[■]══════════════ Directories ══════════════
EXE & TPU directory      [                    ]
Include directories      [                    ]
Unit directories         [                    ]
Object directories       [                    ]

                         [ OK ] [ Cancel ] [ Help ]
```

Az útvonal megadásának szabályai:

- Többszörös directory útvonalának neveit pontosvesszővel kell elválasztani, a ; előtt és után megengedett az üres hely, de nem kötelező, a meghajtót is meg kell adni, ha más mint az aktuális (Az így megadott hozzáférési út 127 karakter hosszú lehet az üres helyeket is beszámítva).

- Ha az *EXE & TPU directory* doboz tartalma üres, akkor ott keletkeznek az *EXE* és *TPU* kiterjesztésű file-ok, ahol a forrás file-ok találhatók. A Map file-ok is itt keletkeznek, ha az *Option|Linker* -nél nem *Off* a beállítás. Egyébként a beállított directory-ban keletkeznek a file-ok.

- *Include directories* dialógus dobozában kell megadni a standard include file-ok directory-ját (**$I** filenév fordító direktíva).

- *Unit directories* dialógus doboz specifikálja a Turbo Pascal unit file-jainak directory-ját. Több directory megadása esetén a terminátor a ; .
  Például
  \TURBO\UNITS;\TURBO\BGI

- *Object directories* dialógus dobozban kell megadni az object file-ok (assembly nyelvű rutinok) directory-ját (**$L** filenév)

Environment parancs segítségével állítjuk be a környezetre vonatkozó opciókat *(Preferences, Editor, Mouse, Startup* és *Color*-ra vonatkozólag).

Preferences dobozban az alábbiakat lehet beállítani:

```
[■]══════════════ Preferences ══════════════
 Screen sizes                   Source tracking
 (•) 25 lines                   (•) New window
 ( ) 43/50 lines                ( ) Current window

 Auto save                      Desktop file
 [x] Editor files               ( ) None
 [ ] Environment                ( ) Current directory
 [ ] Desktop                    (•) Config file directory

   [ OK ]          [ Cancel ]          [ Help ]
```

- *Screen sizes* dobozban állítható be, hogy az IDE képernyő 25 vagy 43/50 sort használjon a megjelenítésre, ez azonban függ a PC video adapterétől. A 25 sor beállítás az alapértelmezés, a Turbo Pascal 25 sort és 80 oszlopot használ. Ezt a beállítást használhatja a monokróm monitor képernyő és a CGA képernyő. A 43/50 sor beállítás csak EGA vagy VGA képernyő esetén lehetséges. Az IDE 43 sort és 80 oszlopot használ EGA, 50 sort és 80 oszlopot használ VGA esetén.

- *Source Tracking* dobozban, ha a *New window* opciót kapcsoljuk be, akkor lépkedve a forrás, vagy helyi hiba pozíciókon, az IDE új ablakot nyit, ha az előforduló file még nem volt betöltve.
  A *Current window* választása esetén az IDE felcse-léli az Edit window aktuális tartalmát az új file tartalmával, ahelyett, hogy új Edit ablakot nyitna.

- Ha az *Editor File* jelölve van az *Auto Save* opcióban, és ha a file az utolsó elmentés óta módosult, akkor a Turbo Pascal automatikusan elmenti a forrás file-t az Edit ablakból.

  Az *Environment* opció bekapcsolása esetén az összes beállítás automatikusan felülírja a TURBO.TP konfigurációs file tartalmát, amikor kilépünk a Turbo Pascal-ból.

  Desktop bekapcsolása esetén a desktop a TURBO.DSK file-ban örződik meg, ha kilépünk a Turbo Pascal-ból.

- *Desktop Files*: A desktop file megőrzését és újratöltését kérhetjük a saját (*Current Directory*) vagy a *Configuration file directory*-ból. Amikor elindítjuk a TURBO.EXE programot, az először a saját directory-ban keresi a TURBO.TP és TURBO.DSK file-okat, ha nem találja, akkor abban a directory-ban keresi, ahonnan a TURBO.EXE file-t behívtuk.

Editor options dobozban beállíthatjuk azokat az opciókat, hogy a Turbo Pascal hogyan kezelje a szöveget az Editor ablakban.

```
[■]═══════════ Editor options ═══════════
  Editor options
  [x] Create backup files          OK
  [x] Insert mode
  [x] Autoindent mode
  [ ] Use tab characters           Cancel
  [x] Optimal fill
  [x] Backspace unindents          Help
  [x] Cursor through tabs

 Tab size   8
```

- *Create Backup Files* (ez az alapértelmezés), a Turbo Pascal az Edit ablakban lévő forrás file-ról automatikusan egy .BAK kiterjesztésű file-t hoz létre, amikor a **File|Save** parancsot választottuk ki.

- *Insert mode* kikapcsolt állapotban megszünteti a beszúró üzemmódot, az Edit ablakban történő javítás felülírást jelent. Bekapcsolt állapotban, vagy az *INS* billentyű megnyomására beszúró üzemód működik, amely a javításnál a karaktereket jobbra tolja el.

- *Autoindent mode* bekapcsolása esetén *ENTER* billentyű megnyomása után a kurzor a következő sorban az előző sor első nem üreshely karaktere alá tabulál. Így könnyebben lehet a programot olvasható formában írni.

- *Use tab characters* be van kapcsolva, akkor a Turbo Pascal igazi tabulátort (ASCII 9) tesz a programba, ha tabulátort nyomtunk meg. Ha ez az opció nincs beállítva, akkor *Tab Size* darab üres hely (space) kerül a programba tabulátor helyett.

- *Optimal fill* bekapcsolása esetén a sor elejét optimálisan tölti ki.
- *Backspace unindents* (ez az alapértelmezés) bekapcsolva és a kurzor az üres sorban az előző sor nem üres karaktere alatt áll, a *Backspace* billentyű hatására az előző sor tabulált elejére kerül.
- *Cursor through tabs* bekapcsolva a kurzor a nyilakkal mozgatva a tabulátor közepére ugrik, egyébként átugorja az oszlopokat.
- *Tab Size* adja meg azt, hogy mennyi karakter legyen a tabulátor mérete. Legkisebb érték 2, legnagyobb a 16, az alapértelmezés 8. Célszerű a 8-as érték meghagyása, mert más szövegszerkesztőknél ill. más operációs rendszereknél is ez a szokásos.

Mouse

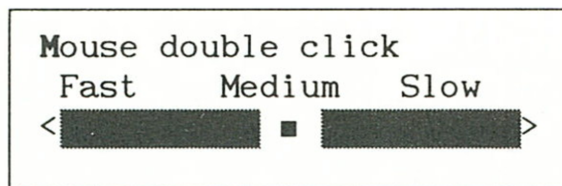
Ha a **M**ouse lehetőséget választjuk az **E**nvironment menüből, akkor a Mouse dialógus doboz tartalmazza az egérre vonatkozó összes beállítási lehetőségeket.

Ha a Right Mouse Button-t bekapcsoltuk be, akkor az egér jobboldali gombja hatásos (a **R**everse mouse esetén a baloldali).

```
Right Mouse Button
 ( )  Nothing
 (•)  Topic search
 ( )  Go to cursor
 ( )  Breakpoint
 ( )  Evaluate
 ( )  Add watch
```

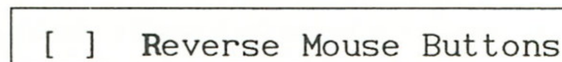
Beállítható, hogy mi történjen, ha megnyomjuk a jobb-oldali gombot az egéren:

Topic Search	ugyanaz, mint a **H**elp|**T**opic Search
Go to Cursor	ugyanaz, mint a **R**un|**G**o to Cursor
Breakpoint	ugyanaz, mint a **D**ebug|**T**oggle Break point
Evaluate	ugyanaz, mint a **D**ebug|**E**valuate
Add Watch	ugyanaz, mint a **D**ebug|**W**atches|**A**dd Watch

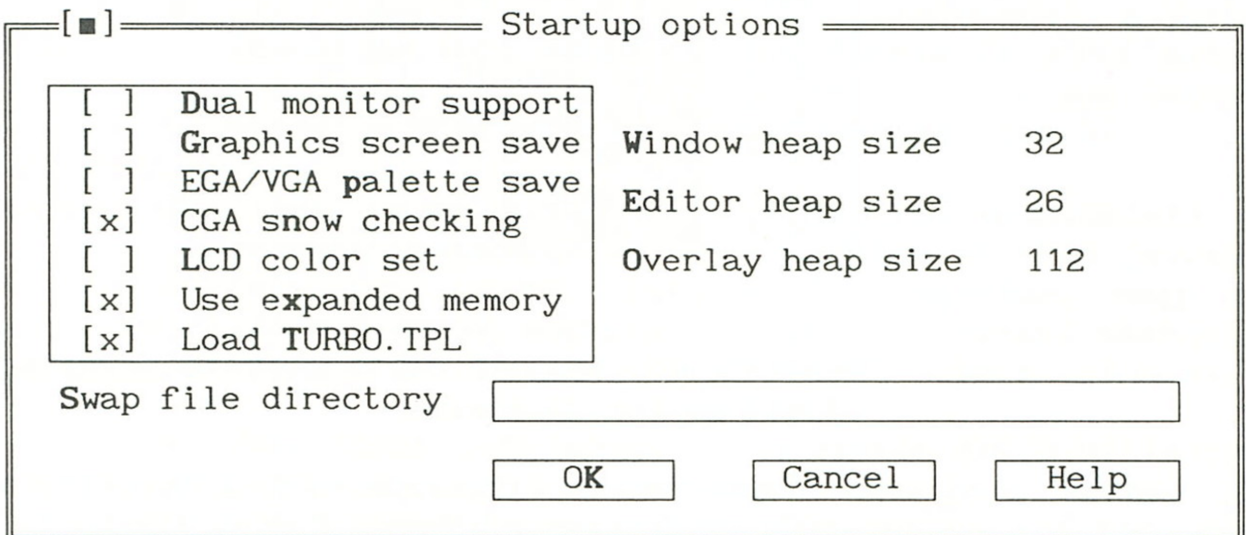


A Mouse double click lehetőségnél a két klikkentés közötti időt állíthatjuk be a dupla klikkentéshez a nyilak segítségével.
Ha a görgető négyzet a Fast (gyors)-hoz van közel a Turbo Pascal-nak rövid idő szükséges, hogy felismerje a dupla klikkentést.
Ha a Slow (lassú)-hoz van közel, a Turbo Pascal még akkor is felismeri a dupla klikkentést, ha hosszú ideig várunk is a két klikkentés között.

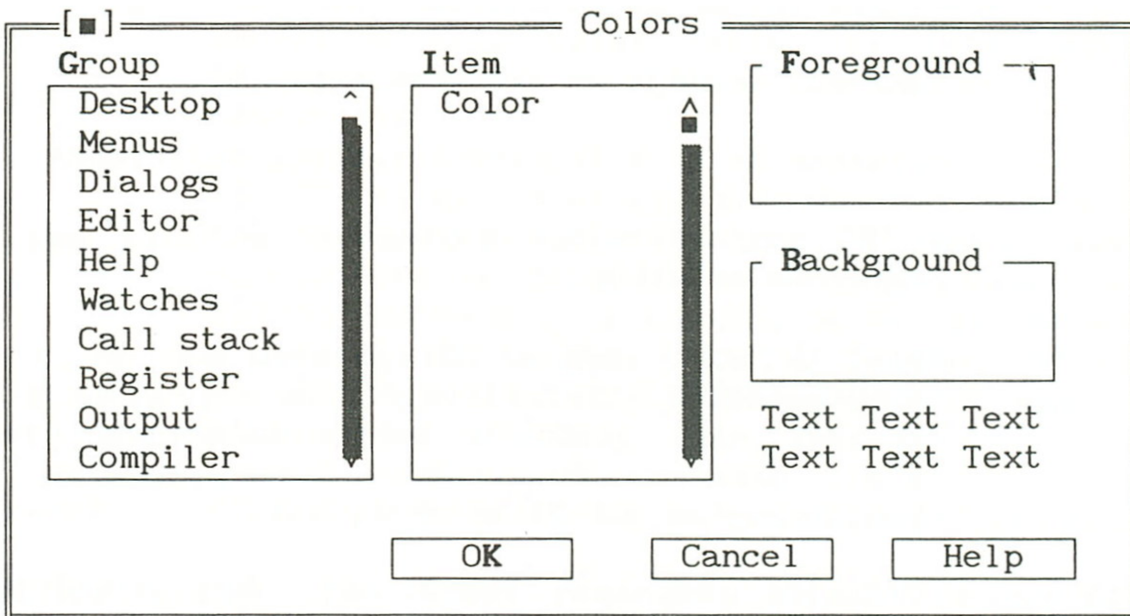
Ha a Reverse Mouse Buttons-t megjelöljük, akkor az egér aktív gombja a jobboldali lesz a baloldali helyett.



Startup dobozban állíthatjuk be az integrált környezetre vonatkozó kívánalmakat.



Colors Az Item alatt a színes vagy fekete-fehér üzemmód, valamint az előtér és a háttér színe is állítható.



Save Options — az Option|Save Options parancs hatására az összes beállítás a TURBO.TP file-ba, a history lista, a desktop állapota és a kijelölt töréspontok a TURBO.DSK file-ba kerülnek.

Retrieve Options — Ha a Desktop file (.DSK) akár a saját directory-ban van, akár Config File directory-ban, a TURBO.DSK betöltődik.

2.1.9. Window menü

A Window menü tartalmazza az ablakokra vonatkozó kezelési parancsokat.

Size/Move — *(CTRL-F5)* változtatja az aktív ablak méretét és pozícióját.

Zoom — (*F5*) az aktív ablak méretét maximalizálja.

Tile — megjeleníti az összes nyitott ablakot, hasonló méretben úgy, hogy egyik a másikat ne fedje el.

Cascade — Egymásba skatulyázva megjelenik az összes nyitott ablak. Az aktív ablak teljesen látható, a többinek csak sorszáma és a fejléce látszik.

Next — (*F6*) az ablak-listán a következő ablakot nyitja meg.

Previous (*Shift-F6*) az előző ablakot teszi aktívvá.

Close (*ALT-F3*) lezárja az aktív ablakot.

Watch ablakban beírt kifejezés értékének változását látjuk.

Register a CPU regisztereinek tartalmát jeleníti meg az ún. Regiszter ablakban.

Output ablak jeleníti meg a DOS parancs sor és a program textüzemmódban kiírt szövegét. Ha a program által írt szöveget vagy grafikát teljes képernyőn szeretnénk látni, akkor a **Window|User** Screen parancs helyett nyomjuk meg az *ALT-F5* billentyűket.

Call Stack (*CTRL-F3*) ablakot nyit meg, amely mutatja az eljáráshívások sorozatát. A Stack-be való belépéskor megjelenik a hívott eljárás neve és az átadott paramétereinek értéke.

User Screen (*ALT-F5*) választásával a program eredményét teljes képernyőn láthatjuk. Bármely billentyű leütésével vagy az egér klikkentésével visszatérhetünk a Turbo Pascal integrált környezetébe.

List (*ALT-0*) megadja az összes aktuális ablak listáját, amelyek közül kurzorral és *ENTER*-rel vagy az egér kétszeres klikkentésével választhatunk.

2.1.10. Help menü

A **Help** menü közvetlen segítséget nyújt a felhasználónak, gyakorlatilag ez a help az összes lehetséges információt tartalmazza az IDE-ről és a Turbo Pascal-ról.

A Help ablakot az alábbiak egyikével lehet megnyitni:

- Bármikor az *F1* billentyű megnyomásával.
- Ha az Edit ablak aktív, a kurzor alatt lévő szóról kapunk Pascal nyelvi információt a (*CTRL-F1*) megnyomásával.
- Egér klikkentésével kaphatunk segítséget (státusz sor, dialógus doboz).

A Help ablak lezárása történhet az *ESC* billentyű megnyomásával, vagy a doboz lezáró ikonján való egér klikkentésével vagy a **Window|Close** kiválasztásával.

Contents kiválasztása megnyitja a Help ablakot. A Help ablakban a fő tartalomjegyzék jelenik meg, ahonnan a help rendszer bármely más részéhez eljuthatunk. A help-ről

is lehet help-et kérni az *F1* billentyű lenyomásával, amikor a Help ablak aktív. El lehet érni ezt a képernyő ablakot a státusz sorban az egér klikkentésével is.

Index — (*SHIFT-F1*) egy dialógus dobozban a Help kulcsszavak teljes listáját jeleníti meg. A lista lapozható. A kulcsszóból a kezdőbetű leütésére azzal a betűvel kezdődő sorra áll, a további betűk leütésével azonnal a keresett kulcsszóra talál. A kulcsszó kiválasztását az *ENTER* billentyű leütése vagy az egér kétszeri klikkentése jelenti.

Topic Search — (*CTRL-F1*) jeleníti meg a Pascal nyelvi helpet.

Previous Topic — (*ALT-F1*) megnyitja a Help ablakot és megjeleníti az utoljára kért helpet. A Turbo Pascal 20 help képernyőt tárol visszamenőleg.

Help on Help — hatására megjelenik egy szöveges képernyő, amely elmagyarázza, hogy hogyan kell használni a Turbo Pascal help rendszerét.

3. OBJEKTUM-ORIENTÁLT PROGRAMOZÁS

3.1. Az objektum-orientált programozás alapfogalmai

Az objektum-orientált programozás (object-oriented programming = OOP) a természetes gondolkodást, cselekvést közelítő programozási mód, amely a programozási nyelvek tervezésének természetes fejlődése következtében alakult ki a korábbi változatokhoz képest. Az így létrejött nyelv sokkal struktúráltabb, modulárisabb és absztraktabb, mint egy hagyományos nyelv.
Az OOP nyelvet három fontos dolog jellemez. Ezek a következők:

- Az *egységbezárás* (*encapsulation*) azt jelenti, hogy az adatstruktúrákat és az adott struktúrájú adatokat kezelő függvényeket (metódusokat) kombináljuk; azokat egy egységként kezeljük, és elzárjuk őket a külvilág elől. Az így kapott egységeket *objektumoknak* nevezzük.

- Az *öröklés* (*inheritance*) azt takarja, hogy a meglévő objektumokból levezetett újabb objektumok öröklik a definiálásukhoz használt alap objektumok egyes adatstruktúráit és függvényeit; ugyanakkor újabb tulajdonságokat is definiálhatnak, vagy régieket újraértelmezhetnek.

- A *többrétűség* (*polymorphism*) alatt azt értjük, hogy egy adott tevékenység (metódus) azonosítója közös lehet egy adott objektum-hierarchián belül, ugyanakkor a hierarchia minden egyes objektumában a tevékenységeket végrehajtó metódus implementációja az adott objektumra nézve specifikus lehet (lásd a virtuális metódusokat).

Ezek a tulajdonságok együtt azt eredményezik, hogy programkódjaik sokkal struktúráltabbá, könnyebben bővíthetővé, könnyebben karbantarthatóvá válnak, mintha hagyományos, nem OOP-technikával írnánk őket.

3.2. Objektumok

A Turbo Pascal 6.0 egyik fontos tulajdonsága, hogy lehetőségünk van az adataink és az őket manipuláló programkód összeforrasztására, egy egységbe zárására, egy objektumba való foglalására. (Ez az ún. *encapsulation*.)

Az objektum-orientált programozás jelentősen eltér a korábbi struktúrált programozástól. A konvencionális programozási nyelvekkel szemben a objektum-orientált nyelveknél az objektum és az ennek hatására működő eljárások és függvények megbonthatatlan egységet képeznek. Az objektumoknak belső adatmezőik vannak. Ha az objektummal egy kört akarunk leírni, akkor például az adatmezői a következők lehetnek: a kör középpontja, a sugara és a kör színe. A kör nevű objektum nagyítására szolgálhat egy zoom metódus, amely a sugár

adatmezejét a kívánt értékre módosítja. Ezek szerint az objektumok belső állapotát, azaz adatmezőiket különböző metódusok módosíthatják.

Az objektum örököli az ős objektum a tulajdonságait (adatmezőit) és viselkedési módjait (metódusait).

3.2.1. Öröklés

Mi jut eszünkbe az öröklés szóról? Talán az, hogy a gyermek milyen tulajdonságokat örökölt szüleitől, vagy gondolhatunk egy családfára. Ugyanilyen családfa építhető fel objektumokkal is. A Turbo Pascal 6.0 verziójában is deklarálhatók alosztályok (leszármazott típusok), ezek öröklik az osztályok (őstípusok) adatmezőit és metódusait, ami lehetővé teszi egy hierarchikus struktúra létrehozását.

Ez az öröklés ugrásszerű fejlődést jelent az objektum-orientált Pascal-ban a hagyományos Pascal-hoz képest.

Nézzünk meg erre egy egyszerű példát először a rekord típussal, majd az objektumok felhasználásával.

Turbo Pascal-ban az objektum nagyon hasonló a rekordhoz, amely logikailag összetartozó adatokat fog össze. Ha grafikus programot készítünk, szükségünk lehet arra, hogy kezeljük a grafikus képernyőn egy pont helyének *X* és *Y* koordinátáit. Hozzunk ezért létre egy *Location* (hely) névvel egy rekord típust:

```
Location = record
  X, Y : integer;
end;
```

Ezt a *Location* típust kétféleképpen lehet használni. Először, amikor szükség van arra, hogy a két koordinátát külön kezeljük, akkor a rekord *X* és *Y* mezejét külön használjuk fel. Másodszor, ha a két koordinátával együtt akarunk dolgozni, - ami egy helyre utal a képernyőn - ez maga a *Location* típusú rekord.

Tegyük fel, hogy szükségünk van a *X*, *Y* koordinátában lévő pont tulajdonságára is, például nyilván akarjuk tartani, hogy látható vagy nem látható-e a pont. Ezt megoldhatjuk a *Visible* (láthatóság) logikai változó bevezetésével. Így deklaráljuk a *Point* nevű típust:

```
Point = record
  X, Y : integer;
  Visible : boolean;
end;
```

De lehetünk egy kicsit leleményesebbek és tovább módosíthatjuk a típusdeklarációt úgy, hogy felhasználjuk az előző *Location* típust, vagyis a *Point* rekordba bevezetünk az *X* és *Y* koordináták helyett egy

Position (helyzet) mezőt, amely *Location* típusú:

```
Point = record
  Position : Location;
  Visible : boolean;
end;
```

Miben különbözik a *Point* típus a *Location* típustól? Abban, hogy a *Point* egy hely (*Location*), amely látható vagy nem látható.

A képernyőn a pont helyzete a *Point* definíciójában explicit módon létezik. Az objektum-orientált programozás felismeri ezt a speciális kapcsolatot. Minden pont megadásához szükséges egy hely. A *Point* típus a *Location* típusnak a leszármazottja. A *Point* típus mindent örököl a *Location* típustól, de a *Point* típus tovább bővíthető bármilyen típusú változókkal. Azt az eljárást, amely által az egyik típus örökli a másik típus tulajdonságait öröklésnek nevezzük. Az öröklő a leszármazott típus (*descendant type*). Az őstípus (*ancestor type*) az, amelytől a leszármazott típus örököl. Az előbbi példa nem öröklés, hiszen ezeknek típusdeklarációk egymásbaskatulyázott használata, ugyanis maga a rekord típus nem tud örökölni. A Turbo Pascal 6.0 azonban kiterjeszti a Pascal nyelvet, úgy hogy támogatja az öröklést. Ez a kiterjesztés egy új adatstruktúrában nyilvánul meg, amely hasonló a rekordhoz, de sokkal hatásosabb. Az **object** kulcsszóval definiáljuk ezt az új adattípust.

Az előbbi példát átírva két objektum típust kell definiálni:

```
type
  Location = object            { nem örököl semmit, tehát őstípus }
    X, Y : integer;
  end;

  Point = object(Location)     { itt örököl }
    Visible : boolean;         { itt kapja az új tulajdonságot }
  end;
```

Itt a *Location* az őstípus, a *Point* pedig a leszármazott típus. Később meg fogjuk látni, hogy ez az eljárás korlátlanul folytatható. Definiálhatjuk a *Point* leszármazott típusát, majd annak a *Point* leszármazott típus leszármazottját. A program tervezésének legnagyobb részét az objektum-orientált alkalmazásban az objektum hierarchiának a felépítése teszi ki, ami tulajdonképpen az objektumok családfájának megtervezését jelenti.

Az összes típus végül is a *Location* típus leszármazott típusa, de a *Point* típus a *Location* típusnak egy közvetlen leszármazottja (*immediate descendants*). Fordítva, a *Location* típus a *Point* típusnak közvetlen őstípusa (*immediate ancestor*). Egy objektum típusnak (mint egy DOS alkönyvtárnak) bármennyi közvetlen leszármazottja lehet, de csak egy közvetlen őstípusa.

Az objektumok szerkezete hasonlít a rekordokhoz, a különbséget későbbi példák illusztrálják.

Például *Point* típus nem tartalmazza a *Location* típus *X* és *Y* mezőit explicit módon, azonban az öröklés következtében a *Point* típusnak mégis vannak közvetlenül *X* és *Y* mezői. Így ugyanúgy beszélhetünk a *Point* típus *X* mezőjéről, mint a *Location* típus *X* mezőjéről.

Az objektum típus egy változója (példánya, *instance*) éppen úgy deklarálható, mint a Pascal hagyományos típusainak egy bármilyen változója. Lehet akár egy statikus változó, vagy akár egy mutató típussal megjelölt hely a heap-ben:

```
type
  PointPtr = ^Point;

var
  StatPoint : Point;      { Statikus változó, már lehet használni }
  DynaPoint : PointPtr;   { A new utasítással kell lefoglalni
                            a helyet a heap-ben }
```

Pontosan úgy férhetünk hozzá az objektum adatmezőihez, mint a rekord mezőihez: a **with** utasítással vagy a . (pont) alkalmazásával.

Például:

```
MyPoint.Visible := false;

with MyPoint do begin
  X := 9;
  Y := 71;
end;
```

De egyet meg kell jegyeznünk: az örökölt mezőkhöz ugyanúgy férhetünk hozzá, mint azokhoz a mezőkhöz, amit az objektum típusnak a deklarációjában adtunk. Például az *X* és *Y* nem része a *Point* típus deklarációjának, mert ezeket örökölte a *Location* típustól, de úgyanúgy hivatkozhatunk rájuk, mintha a *Point* típusban deklaráltuk volna:

```
MyPoint.X := 8;
```

Az objektum adatmezőihez közvetlenül is hozzáférhetünk, azonban kerüljük ezt az eljárást, helyette az adatok elérésére az objektum metódusát kell használni.

Végül hasonlítsuk össze a *Point* típus rekordos és objektumos deklarációja esetén az *X* mezőhöz való hozzáférésnek a módját.
Legyen

```
var
    P : Point;
```

akkor a rekordos példánál a P-nek az *X* mezőjét

```
P.Position.X
```

utasítással érjük el (skatulyázás), objektum-orientált megközelítésben

egyszerűen

```
P.X
```

mivel az objektum már örököl.

3.3. Metódusok

A metódus olyan eljárás vagy függvény, amelyet az objektumon belül kell deklarálni és szorosan kapcsolódik az objektumhoz.

A metódus az objektum-orientált programozás egyik legfontosabb eleme. Tulajdonképpen arról van szó, hogy az adatmezőkhöz hasonlóan (ami miatt az object hasonlít a record-hoz) függvény vagy eljárás típusú mezőket is deklarálhatunk egy objektumtípus definiálásakor. Az ilyen jellegű objektummezőket összefoglalóan metódusnak nevezzük az objektum-orientált Pascal-ban. (A C++ a metódusokat egyszerűen csak függvénymezőknek - *member function* -nevezi.) A metódus az adatstruktúra inicializálásával kezdődik. Először nézzük a példát rekord definícióval:

```
Location = record
  X, Y : integer;
end;
```

A programozók többsége a **with** utasítással adna kezdőértéket az *X* és *Y* mezőknek:

```
var  MyLocation : Location;

with MyLocation do begin
  X := 3;
  Y := 45;
end;
```

A fenti programrészlet jól működik, de szorosan kötődik a *MyLocation* rekordhoz. Ha több *Location* típusú rekordnak akarunk kezdőértéket adni, akkor minden esetben a **with** utasítást kellene alkalmazni. Következő lépésként írjunk egy olyan eljárást, amely paraméterként adja meg egy *Location* típusú rekordnak a kezdőértékét szintén a **with** utasítással:

```
procedure Init(var Target : Location;
                   NewX, NewY : integer);
  begin
    with Target do begin
      X := NewX;
      Y := NewY;
    end;
  end;
```

Ez az eljárás már jobban megfelel a célnak, de nem ez volt az objektum-orientált programozás egy korai ösztönzője. Ezért szeretnénk egy olyan *Init* eljárást készíteni, amely csak a *Location* típusnak felel meg. Ennek érdekében kapcsoljuk össze a rekord típust és az inicializáló eljárást, hogy így megkapjuk az előbbi tervünknek megfelelő megoldást.

A megoldást metódusnak hívják. A metódus lehet egy eljárás vagy egy függvény, amely teljesen összeforr az adott adatmezőivel, mert egy láthatatlan **with** utasítás fogja őket közre (lásd az egységbezárást). Így közvetlenül hozzáférhetünk az objektum mezőihez. Az objektum definíciójának tartalmaznia kell a metódus fejrészét. A metódus teljes definícióját az objektum nevével együtt kell megadni. Az objektum típus és metódus adja azt az új struktúrát, amit objektumnak nevezünk:

```
type
  Location = object
    X, Y : integer;
    procedure Init(NewX, NewY : integer);
  end;

procedure Location.Init(NewX, NewY : integer);
begin
  X := NewX; { az X mező a Location objektumé }
  Y := NewY; { az Y mező a Location objektumé }
end;
```

Most már inicializálhatjuk a *Location* objektum egy példányát. Úgy kell hívni a metódust, mintha egy rekord mezője volna:

```
var
  MyLocation : Location;
begin
  MyLocation.Init(3, 45);
```

Az objektum-orientált programozás azon tulajdonságát, amikor egy objektumtípus deklarálásakor szorosan egymáshoz rendelt adatmezőket és metódusokat definiálunk, egységbezárásnak (*encapsulation*) nevezzük. E fogalom jól kifejezi azt, hogy az adatmezőket igyekszünk a külvilág elől elrejteni, azokat csak szorosan hozzájuk rendelt, a velük egységbe zárt metódusok érhetik el.

3.3.1. Program és az adatok

A programozónak arra kell gondolnia a program tervezése közben, hogy a utasítások és az adatok összefonódnak. Ez az objektum-orientált programozás egyik legfontosabb alapelve. Nincs értelmük az utasításoknak adatok nélkül, és fordítva. Az adatok irányítják az utasításokat, és az utasítások megváltoztatják az adatok formáját és értékét.

Ha ugyanis teljesen elválasztjuk az adatokat az utasításoktól, akkor felléphet az a veszély, hogy egy rossz adattal hívunk meg egy jó eljárást, vagy egy jó adattal egy rossz eljárást. A kettő összeillesztése a programozó dolga, és a Pascal szigorú struktúrája csak abban ad segítséget, hogy mi nem jó a programban. A Pascal nem közöl semmit arról, hogy valami összeillik-e vagy sem.

Az adatok és az utasítások együttes deklarációja nem más, mint objektumtípus definiálása. Ha az objektum egy mezőjének az értékére vagyunk kiváncsiak, hívjunk meg egy metódust, amely az objektumhoz tartozik és visszadja a kívánt mező értékét. Ha viszont egy mezőnek akarunk értéket adni, akkor hívjuk meg azt a metódust, amely átadja az új értéket a mezőnek.

A struktúrált és az objektum-orientált programozásban vannak bizonyos szabályok, melyekre rá vagyunk kényszerítve és vannak bizonyos koncepciók, melyek betartása hasznos. Ilyen koncepció pl. az, hogy az objektumok adatmezőit közvetlenül nem kezeljük. A Turbo Pascal 6.0 azonban megengedi, hogy közvetlenül - a metódusok használata nélkül - használjuk az objektum mezőit, ezt lehetőleg kerüljük el. A Turbo Pascal 5.5-höz képest a 6.0-ás változat az objektum deklarációban a **private** új kulcsszóval támogatja az igazi egységbeágyazást. Ez a kulcsszó semmi mást nem tesz, mint szabályozza az adatmezőkhöz való hozzáférést. Egy objektum private adatmezőit csak az adott objektumtípus metódusai érhetik el, a külvilágból azok teljesen láthatatlanok.

3.3.2. A private kulcsszó

Lehet olyan körülmény, amikor azt akarjuk, hogy az objektum egy része ne legyen hozzáférhető a külvilágból. Ezt a Turbo Pascal 6.0-ban könnyen megtehetjük, mert azokat a mezőket, amelyeket el szeretnénk rejteni a **private** kulcsszót követően deklaráljuk.

A **private** mezők és metódusok csak abban a unit-ban érhetők el, ahol az objektumtípus deklarálva van. Az előző példában, ha a *Point* típusnak lennének **private** adatmezői, ill. metódusai, akkor azokhoz csak a *Point* típus deklarációját is tartalmazó *Points* unit-ban férhetnénk hozzá.

A **private** mezőket és metódusokat a normál mezők és metódusok után kell deklarálni. A deklaráció a következő lehet:

```
type
     NewObject = object(ancestor)
          fields1;  (* ezek a közös mezők *)
          methods1; (* ezek a közös metódusok *)
     private
          fields2;  (* ezek a privát mezők *)
          methods2; (* ezek a privát metódusok *)
     end;
```

3.3.3. Metódusok definíciója

Az objektum metódusainak a definíciója emlékeztet a Turbo Pascal unit-jaira. Az objektum deklarációján belül helyezkedik el a metódus fejrésze:

```
type
  Location = object
    X, Y : integer;
    procedure Init(InitX, InitY : integer);
    function GetX : integer;
    function GetY : integer;
  end;
```

Az összes adatmezőt még az első metódus előtt kell deklarálni. Mint a unit interface részén az eljárás és a függvény deklarációkat, ugyanúgy kell a metódusokat deklarálni. Ez a deklaráció csak azt mutatja meg, hogy a metódusokat hogy kell aktivizálni.

A metódusok törzsrészét az objektum definícióján kívül kell megadni, különálló eljárás vagy függvény deklarációban. Az objektum definícióján kívül elhelyezkedő metódus nevét az objektum neve előzi meg, a kettőt egy pont választja el.
Tekintsük a *Location* objektumtípus metódusainak megvalósítását.

```
{ objektumtípus deklarációja }

    type
      Location = object
        X, Y : integer;
        procedure Init(InitX, InitY : integer);
        function GetX : integer;
        function GetY : integer;
      end;

{ Hely inicializálása: }

    procedure Location.Init(InitX, InitY : integer);
      begin
        X := InitX;
        Y := InitY;
      end;

{ X koordináta lekérdezése:}

    function Location.GetX : integer;
     begin
        GetX := X;
      end;
```

```
{ Y koordináta lekérdezése:}

     function Location.GetY : integer;
       begin
         GetY := Y;
       end;
```

A metódus definíciója követi a fejrészt. A *Location.GetX* metódus definíciója mellett teljesen legálisan definiálhatunk egy - a *Location* azonosítót nem tartalmazó - *GetX* függvényt. Azonban ez a *GetX* függvény nincs kapcsolatban a *Location* típusú objektummal és talán össze is téveszthetjük ezt a két függvényt a programírás során, ezért az azonos neveket kerüljük el.

3.3.4. A metódus hatásköre és a Self paraméter

Láthatjuk, hogy az előző metódusokban sehol sem található explicit módon **with** *objektum*-példány **do** ... utasítás. Az objektum adatmezői az objektum metódusai számára mégis szabadon hozzáférhetők, pedig az adatmezők és az utasítások nem egy helyen vannak. Ennek ellenére a metódus és az objektum adatmezői ugyanazt a munkaterületet használják. Ezért tartalmazhatja a GetY := Y utasítást a *Location* típus egyik metódusa, és azért nem tartalmaz kiegészítést az *Y* mező, mert ahhoz az objektumhoz tartozik, amely ezt a metódust tartalmazza. Amikor egy objektum hív egy metódust, természetesen egy láthatatlan **with** *objektum* **do** *metódus* utasítás működik, amely az objektum adatmezőit a metódus hatáskörébe rendeli.

Ez a közvetlen **with** utasítás mindig pontosan végrehajt egy láthatalan paraméterátadást, amikor meghívjuk a metódust. Ennek a paraméternek a neve *Self*, ez valójában egy 32 bites mutató, amely arra az objektumra mutat, amely a metódushívást végzi. A *Location* típus *GetY* metódusa a *Self* paraméterrel az alábbiaknak felelne meg:

```
     function Location.GetY(var Self : Location) : integer;
       begin
         GetY := Self.Y;
       end;
```

Ez a példa szintaktikailag nem teljesen korrekt, de hozzásegít ahhoz, hogy megérthessük azokat a speciális kapcsolatokat, amelyek az objektum adatmezői és metódusai között vannak. Nem fontos teljesen tisztában lenni a *Self* fogalmával. A Turbo Pascal által generált kód gyakorlatilag automatikusan kezeli az összes esetet.

Explicit módon ugyan lehet használni a *Self* paramétert, de lehetőleg kerüljük az alkalmazását. A *Self* egy aktuális értéket tartalmazó automatikusan deklarált azonosító. Ha például az azonosítók összeférhetetlenek valamilyen oknál fogva egy metóduson belül, a *Self* azonosítót felhasználhatjuk arra, hogy hozzáférjünk bármelyik adatmezőhöz, amely az objektumhoz tartozik.

A következő példában a *MouseStat* rekord az egér (mouse) tulajdonságait tartalmazza. Ezek az egér

- működési állapota (*Active*),
- helyzete (*X, Y*),
- gombjainak állapota (*LButton, Rbutton*),
- az egérkurzor láthatósága (*Visible*):

A fenti adatokat leíró rekord típust szeretnénk a *Location* objektumtípussal együtt használni. Az adatmezőkhöz való hozzáférést a *Location* típus *GotoMouse* metódusának *Self* paraméterével tesszük egyértelművé:

```
type
  MouseStat = record
    Active          : boolean;
    X, Y            : integer;
    LButton, RButton : boolean;
    Visible         : boolean;
  end;

  Location = object
    X, Y: integer;
    procedure GotoMouse(MousePos : MouseStat);
  end;

procedure Location.GoToMouse(MousePos : MouseStat);
  begin
    Hide;
    with MousePos do begin          ]
      Self.X := X;                  |  X := MousePos.X;
      Self.Y := Y;                  |  Y := MousePos.Y;
    end;                            ]
    Show;
  end;
```

A *Location.GotoMouse*, a *Location* objektum metódusa, melynek feladata, hogy az egér által érzékelt, megváltozott képernyő pozícióit átvegye és beírja a *Location* objektum helykoordinátáiba. Ebben az eljárásban az *X* és *Y* helykoordináták hovatartozása kissé ellentmondásos.

A példában kétfajta módszert is bemutattunk:

- **with** ... **do** utasítás alkalmazásával olvastuk le az egér (*MousePos*) helykoordinátáit és a *Self* paraméter segítségével pedig a *Location* objektum helykoordinátáit,

- egyszerűbbnek tűnik a programrészlet, ha a **with** utasítást elhagyjuk, így elkerüljük a *Self* paraméter használatát.

3.4. Egy objektum adatmezői és a metódusok formális paraméterei

Mivel a metódusok és az adatmezők közös munkaterületet használnak, a metódusok formális paramétereinek nevei nem lehetnek azonosak az objektumok adatmezőinek neveivel. Ezt a szabályt nemcsak az objektum-orientált programozásban kell betartani, hanem a Pascal programozási nyelv szintaktikájához is hozzátartozik. Nincs megengedve, hogy egy eljárás formális paraméterének neve és egy változójának a neve megegyezzen. Például a *Param* eljárásban a egyik bemenő paraméter az *ErrorCode* egész típusú változó és az eljárás törzse szintén tartalmazza azonos néven a változót:

```
procedure Param(Data : MyDataRec,
                L, ErrorCode : integer);
var
  A, B      : char;
  ErrorCode : integer; { Ez a deklaráció okozza a hibát }

begin
  .
```

Az eljárás lokális változói és formális paraméterei megosztják a munkaterületet, így nem lehetnek azonosak. Erre a hibára a következő hibaüzenetet kapjuk: "*Error 4: Duplicate identifier*". Ugyanezt kapjuk, ha a metódus formális paramétereinek neve megegyezik a saját objektuma bármelyik adatmezőjével.

A körülmények mások, amióta a Turbo Pascal-ban az új elv szerint az eljárás fejlécei belül vannak adatstruktúrában, azonban a Pascal hatáskörökre vonatkozó elvei nem változtak meg.

3.5. Objektumok unit-ban

Unit-okban is definiálhatunk objektumokat. Az *interface* részben kell az objektum típusát deklarálni, és az *implementation* rész pedig a metódusok törzsét tartalmazza.

A unit-oknak lehet saját objektumtípus definíciójuk az *implementation* részben, de az ilyen típusok ugyanazzal a korlátozással rendelkeznek, mint az itt elhelyezkedő többi típus. Ha egy unit *interface* részében egy objektum van definiálva, akkor őstípusa lehet az *implementation* részben definiált objektumnak. Abban az esetben, ha a B unit használja az A unitot, a B unit-ban definiálhatunk az A unit bármelyik objektumából leszármazott típust.

Az előbbiekben leírtak alapján az objektum típusok és metódusok a

következő módon definiálhatók egy unit-ban:

```
unit Points;
interface

uses Graph;

type
  Location = object;
    X, Y : integer;
    procedure Init(InitX, InitY : integer);
    function GetX : integer;
    function GetY : integer;
  end;

  Point = object(Location)
    Visible : boolean;
    procedure Init(InitX, InitY : integer);
    procedure Show;
    procedure Hide;
    procedure IsVisible : boolean;
    procedure MoveTo(NewX, NewY : integer);
  end;

implementation
{---------------------------------------------------------------}
{ Location objektum metódusainak törzse:                        }
{---------------------------------------------------------------}
procedure Location.Init(InitX, InitY : integer);
  begin
    X := InitX;
    Y := InitY;
  end;

function Location.GetX : integer;
  begin
    GetX := X;
 end;

function Location.GetY : integer;
  begin
    GetY := Y;
  end;

{---------------------------------------------------------------}
{ Point objektum metódusainak törzse:                           }
{---------------------------------------------------------------}
procedure Point.Init(InitX, InitY : integer);
  begin
    Location.Init(InitX, InitY);
    Visible := false;
  end;
```

```
procedure Point.Show;
  begin
    Visible := true;
    putpixel(X, Y, getcolor);
  end;

procedure Point.Hide;
  begin
    Visible := false;
    putpixel(X, Y, getbkcolor);
  end;

function Point.IsVisible : boolean;
  begin
    IsVisible := Visible;
  end;

procedure Point.MoveTo(NewX, NewY : integer);
  begin
    Hide;
    X := NewX;
    Y := NewY;
    Show;
  end;
end.
```

Ha ennek a *Points* unit-nak az objektum típusait és metódusait használni szeretnénk, a programunkban a **uses** kulcsszó után írjuk a unit nevét, és deklaráljuk például a *Point* objektumnak egy példányát:

```
program MakePoints;
uses Graph, Points;
var
  APoint : Point;
```

Ha az *APoint* által használt grafikus képernyőn lévő pont tulajdonságait szeretnénk megváltoztatni, használjuk az *APoint* metódusait:

```
APoint.Init(122, 53);        { a pont X és Y koordinátája :
                               122 és 53 }

Apoint.Show;                 { a pontot megjelenítjük }
APoint.MoveTo(157, 23)       { a pontot a 157, 23 pontba
                               mozgatjuk }
APoint.Hide;                 { a pontot eltüntetjük }
```

Az objektumnál, amely nagyon hasonlít a rekordra, szintén használható a **with** utasítás. Ebben az esetben a metódus neve elé nem

kell kitenni az őstípus objektum-példány nevét:

```
with APoint do begin
  Init(122, 53);          { a pont X és Y koordinátája :
                            122 és 53 }
  Show;                   { a pontot megjelenítjük }
  MoveTo(157, 23)         { a pontot a 157, 23 pontba
                            mozgatjuk }
  Hide;                   { a pontot eltűntetjük }
end;
```

Ugyanúgy, mint a rekordok, az objektumok lehetnek eljárások paraméterei és tárolhatjuk példányaikat a heap-ben is.

3.6. Egységbezárás (*encapsulation*)

Egységbezárás *(encapsulation)* nevezzük az utasítások és az adatok összefonódását egy objektumban. Mindig gondoskodnunk kell elég metódusról, hogy az objektumot felhasználó programozó ne használhassa közvetlenül az objektum mezőit. A Turbo Pascal 6.0 megengedi, hogy a **private** direktíván keresztül szabályozzuk az egységbezárás "mélységét". A *Location* típusdeklaráció példa nem tartalmazott **private** specifikációt a mezők és metódusok számára, ezért így nem vagyunk korlátozva az adatok elérésében.

A *Location* és a *Point* típusok úgy vannak definiálva, hogy teljesen szükségtelen a belső adatmezőikhez közvetlenül hozzáférni. Ezért célszerűbb, ha a külső mezőhozzáférést a **private** direktívával meg is tiltjuk:

```
type
  Location = object;
    procedure Init(InitX, InitY : integer);
    function GetX : integer;
    function GetY : integer;
    private
      X, Y : integer;
  end;

  Point = object(Location)
    procedure Init(InitX, InitY : integer);
    procedure Show;
    procedure Hide;
    procedure IsVisible : boolean;
    procedure MoveTo(NewX, NewY : integer);
    private
      Visible: boolean;
  end;
```

A fenti deklaráció azt eredményezi, hogy az adatmezőket avatatlan módon nem lehet manipulálni, hiszen csak a **private** kulcsszó hatására a *Location* ill. *Point* típusok a külvilágtól (külső unit-októl) elzárt, teljesen saját mezői lettek, csak a megfelelő objektumtípusok publikus, mindenki által elérhető metódusain keresztül manipulálhatók.

A fenti programrészlet csak három adatmezőt tartalmaz: *X*, *Y*, *Visible*. A *MoveTo* metódus új értékeket ad az *X* és *Y* mezőknek, míg a *GetX* és *GetY* metódus az *X* és *Y* mezők értékét adja vissza. Ezekben a metódusokban nincs szükség az *X* és *Y* változók közvetlen elérésére. A *Show* és a *Hide* metódusok pl. *TRUE*, illetve *FALSE* értéket adnak a *Visible* mezőnek, az *IsVisible* metódus pedig a *Visible* mező aktuális értékét szolgáltatja.

Vegyük a *Point* típus egy példányát és nevezzük *APoint*-nak. Így az *APoint* adatmezőit a metódusok segítségével a következő módon használhatjuk:

```
with APoint do begin
  Init(0, 0);              { A 0, 0 koordinátát veszi fel }
  Show;                    { A pont látható lesz }
end;
```

Megjegyezzük, hogy az objektumok mezőihez csak az objektumok metódusaival férjünk hozzá. Ahol csak lehet használjuk a **private** kulcsszót a megfelelő adatmezőre, illetve metódusok elrejtésére.

3.7. Többrétűség (*polymorphism*)

A többrétűség (vagy sokalakúság, sokoldalúság) az objektum-orientált Pascal-ban azt jelenti, hogy egy adott őstípusból származtatott további típusok természetesen öröklik az őstípus minden mezőjét, így a metódusokat is. De a fejlődés során a tulajdonságok egyre módosulnak, azaz például egy öröklött metódus névleg ugyan nem változott egy leszármazottban, de esetleg már egy kicsit (vagy éppen nagyon) másképpen viselkedik.

Amikor egy leszármazott típust definiáltunk, az örökli az őstípus metódusait, ezek azonban átalakíthatók kívánság szerint. Ha át akarunk alakítani egy örökölt metódust, egyszerűen definiálni kell ugyanazzal a névvel, de különböző törzzsel és - ha szükséges - különböző paraméterekkel, az újabb típusnak megfelelő változókat.

Egy egyszerű példa mind a két folyamatot bemutatja. Definiáljunk egy *Point* típusból leszármazott típust, amely egy kört rajzol egy pont helyett a képernyőre:

```
type
  Circle = object(Point)
    Radius : integer;
    procedure Init(InitX, InitY : integer;
```

```
                     InitRadius : integer);
    procedure Show;
    procedure Hide;
    procedure Expand(ExpandBy : integer);
    procedure MoveTo(NewX, NewY : integer);
    procedure Contract(ContractBy : integer);
  end;

procedure Circle.Init(InitX, Init
                        InitRadius : integer);
  begin
    Point.Init(InitX, InitY);
    Radius := InitRadius;
  end;

procedure Circle.Show;
  begin
    Visible := true;
    graph.circle(X, Y, Radius);
  end;

procedure Circle.Hide;
  var
    TempColor : word;
  begin
    TempColor := graph.getcolor;
    graph.setcolor(getbkcolor);
    Visible := false;
    graph.circle(X, Y, Radius);
    graph.setcolor(TempColor);
  end;

procedure Circle.Expand(ExpandBy : integer);
  begin
    Hide;
    Radius := Radius + ExpandBy;
    if Radius < 0 then Radius := 0;
    Show;
  end;

procedure Circle.Contract(ContractBy : integer);
  begin
    Expand(-ContractBy);
  end;

procedure Circle.MoveTo(NewX, NewY : integer);
  begin
    Hide;
    X := NewX;
    Y := NewY;
    Show;
  end;
```

Egy kör felfogható mint egy kövér pont: ugyanazokkal a változókkal rendelkezik mint egy pont (*X*, *Y* koordináták, *Visible* láthatóság), de még van sugara is. Úgy tűnik, hogy a *Circle* objektumtípusnak csak egy mezeje van, a *Radius*, de ne feledkezzünk meg azokról a mezőkről, amelyeket a *Point* típustól örökölt. A *Circle* objektumnak szintén vannak tehát *X*, *Y*, és *Visible* mezői, mégha nem is látjuk ezeket a *Circle* definíciójában.

Mivel a *Circle* definiált egy új mezőt, a *Radius*-t, egy új *Init* metódust kell késziteni, amely beállítja a *Radius* értékét és a többi örökölt mezőt. Készítsünk inkább metódust, mint közvetlenül adjunk értéket az örökölt *X*, *Y* mezőknek és a *Visible* mezőnek. Használjuk a *Point.Init* metódust erre a célra (ezt mutatja a *Circle.Init* első sora). Az örökölt metódust a következőképpen kell hívni : *Őstípus.Metódus*, ahol az *Őstípus* az ős objektum típus azonosítója, a *Metódus* pedig ennek a típusnak a metódus azonosítója.

A *Circle.Init* metódus a kör adatainak ad kezdőértéket felhasználva a *Point.Init* metódust is. A *Point.Init* metódus egy rejtett inicializálást hajt végre. A metódus felülírásánál gondoskodni kell az őstípustól örökölt objektumtípus metódusának hívásáról. Természetesen, ha bármilyen változtatás történik az őstípus metódusában, az kihatással lesz a leszármazottakra.

Miután meghívta a *Point.Init* metódust, a *Circle.Init* metódus végrehajtja a saját inicializálását, ami most csak abból áll, hogy a *Radius* átveszi az *InitRadius* értékét.

Ahelyett, hogy a kört pontonként rajzolnánk meg, használjuk a Graph unit Circle eljárását. Ezért van szükség a *Circle* objektum új metódusaira: a *Show* és *Hide* metódusokra, amelyek felülírják a *Point* metódusait.

Viszont meg kell oldanunk a névazonosságból adódó problémát. Az objektumunknak és a kört rajzoló eljárásnak is ugyanaz a neve. A Graph.Circle utasítás a Turbo Pascal számára teljesen egyértelműen azt jelenti, hogy azt a Circle eljárást akarjuk használni, amely a GRAPH.TPU saját eljárása és nem a *Circle* objektumot.

Bár a metódusokat át lehet alakítani, az adatmezőket nem. Ha egyszer definiáltunk egy adatmezőt az objektum hierarchiájában, a leszármazott típusnak már nem definiálhatunk ugyanilyen nevű adatmezőt.

3.7.1. Sokalakú objektumok

Ha egy metódus nem csak egy adott objektumtípus része, hanem annak tetszőleges leszármazottja is használja, akkor honnan tudja a metódus, hogy pontosan mit is kell végrehajtania? Fordítás alatt még nem tudja (pontosan), csak az őstípust. Ezért sokalakú.

Mire jó egy sokalakú objektum? Lehetővé teszi olyan objektumok kezelését, amelyeknek típusa fordításkor még nem ismert.

Pl. grafikus toolboxot készítünk, melyben különböző alakzatok vannak és mozgatásukra egy eljárás szolgál. Hagyományos Pascalban minden alakzathoz kellene egy külön eljárás, mert még ha ki is játszanánk a szigorú Pascal típusmegkötéseket, a tevékenység maga is

annyira különböző, hogy nem tudnánk általános eljárást írni.

Egy **case** utasítással alakzatok szerint elágazhatnánk, de így is körülményes és új alakzatok mozgatására alkalmatlan lenne. A toolbox (forráskód nélkül) tehát bővíthetetlen.

A megoldás: sokalakú objektumok és ún. virtuális metódusok használata. Ha a toolbox alakzatmozgató eljárása virtuális és fel van készítve sokalakú objektumokra, akkor a felhasználó által utólag definiált leszármazott objektumokat is kezelni tudja.

3.7.2. Örökölt statikus metódusok

A *Circle* objektum mindenféle változtatás nélkül örökölte a *GetX* és a *GetY* metódusokat a *Location* típustól. A *MoveTo* metódust újradefiniáltuk a *Circle* definíciójában, de mivel a *MoveTo* nem használja a *Radius* mezőt, ezért azonosnak tűnik a *Point.MoveTo* metódussal. Azért kellett újradefinálni, mivel olyan metódusokat használ (*Show*, *Hide*), amelyeket az igényeknek megfelelően megváltoztattunk. Az összes metódus szoros kapcsolatban van a *Location* és a *Point* objektumokkal. A *Circle* objektum metódusai is statikusak.

Ha örökölt statikus metódust használunk, amely más metódusokat hív meg, akkor a hívott metódusok az őstípus metódusai, még akkor is, ha a leszármazott típusnak vannak olyan metódusai, amelyek felülírták az őstípus metódusait.

3.7.3. Virtuális metódusok

Az eddig tárgyalt metódusok statikusak. A fordító ugyanúgy fordítási időben foglal nekik helyet és oldja fel a rájuk való hivatkozásokat, mint a hagyományos statikus változóknak.

A statikus metódusok és objektumok gyakran jól használható eszközök, de néha nem eléggé hatékonyak.

Az előzőekben ismertetett probléma oka a fordítási időben való hivatkozás-feloldás. A probléma megoldása a metódushivatkozások futási időben való feloldása, vagyis dinamikussá tétele. Ehhez nyújtanak segítséget a Turbo Pascal virtuális metódusai.

A virtuális metódusok a sokalakúság igen hatékony támogatói. A sokalakúság jelentése: egy metódusnak nevet adunk és hozzáférhetővé tesszük azt az objektum hierarchiájából úgy, hogy minden objektum saját magához illeszkedő módon implementálja a metódust.

Az előzőekben leírt grafikus alakzatok jó példát szolgáltatnak erre.

A mi hierarchiánkban minden objektum egy alakzatot képvisel a képernyőn: egy pontot vagy egy kört. Ezek a képernyőn megjeleníthetők (*Show*). Később, ha további objektumokat definiálnánk (négyzet, körív, stb.), azok megjelenítésére is írhatnánk metódusokat. Az új OOP technikával erre nincs szükség. Az alakzatoknak közös viselkedése

(metódusa) az önmaguk megjelenítése (és ezt egyetlen virtuális metódus realizálja).

A különböző alakzatok a megjelenítés mikéntjében különböznek. A pont megjelenítéséhez egy pixel bekapcsolása szükséges, ami nem igényel más adatot mint a koordinátákat és esetleg a színt. A kör már egy külön rutint, adatokban pedig az előzőeken kívül a sugarat, a körív pedig még a kezdő- és végszöget és még bonyolultabb rajzoló algoritmust is igényel.

Minden alakzat megjeleníthető, de a mechanizmus mindig különböző. A "*Show*" szó tehát mindig mást jelent.

Új kulcsszó: **virtual**. Ha egy metódust az ősben virtuálisnak deklaráltunk, akkor valamennyi leszármazott azonos nevű metódusa virtuális lesz.

Minden objektum típusnak, amely virtuális metódust tartalmaz, szükséges egy konstruktor (ajánlott név: *Init*), amelyet az objektum minden példányára meg kell hívni, még mielőtt bármelyik virtuális metódusát hívnánk.

3.7.4. Korai és késői kötés

A különbség egy statikus és egy virtuális metódus hívása között ugyanaz, mint egy most meghozott és egy későbbi döntés között. Egy statikus metódus hívásakor tulajdonképpen azt mondjuk a fordítónak: "Tudod mit akarok. Hívjad.", egy virtuálisnál pedig "Még nem tudod mit akarok. Ha eljön az ideje, kérdezd meg."

Gondoljuk végig így az előző *MoveTo* problémát. A *Circle.Moveto* hívása csak egy helyre mehet: a hierarchiában felfelé a legközelebbi *MoveTo* implementációra, amit a *Point* objektumban talál meg. Amelyik leszármazott típus nem írja fölül az ős metódusát, az mind ugyanazt a metódust (implementációt) fogja használni. A döntés fordítás alatt meghozható.

Más a helyzet, amikor *MoveTo* a *Show* metódust hívja. Minden alakzatnak van a saját, speciális *Show* metódusa, és hogy melyik implementációt is hívjuk, az azon múlik, hogy melyik objektum példány (leszármazott) hívta eredetileg a *MoveTo* metódust. Ezért kell futási időre elhalasztani a döntést a *Show* hívásáról: fordítási időben még nem tudjuk, melyik objektum példány fogja hívni a *MoveTo* metódust.

Korai kötésnek nevezzük azt az eljárást, amikor egy statikus metódus hívása fordítási időben egyszer s mindenkorra rögzített. Ennek futási időben történő eldöntését nevezzük késői kötésnek.

3.7.5. Példa késői kötésre

Készítsünk unit-ot, mely geometriai alakzatokat és az azokat mozgató eljárást tárolja, majd írjunk egy főprogramot, ami ezt a unit-ot használja, kiegészíti egy új alakzattal (a unit-beliekből származtatva), és azt mozgatja.

Gondoljuk meg, mi közös és mi különböző az alakzatokban! A különbség világos, a közös:

- van egy pontjuk (*Location*), aminél fogva megragadjuk őket (*X*, *Y*),

- lehetnek láthatók és láthatatlanok.

Ezek épp a korábban definiált *Point* objektumtípus tulajdonságai, az lesz hát az alakzatok őse. Ez egy általános OOP elv: keressünk őst és ruházzuk fel mindazon tulajdonságokkal, amit később át akarunk örökíteni tőle.

Lehet, hogy ebből az őstípusból soha nem definiálunk egyetlen példányt sem, s szerepe csak annyi, hogy örökít. Az ilyen ősöket absztrakt objektumtípusnak nevezzük. (A példabeli *Point* is ilyen, bár lehetne belőle példányokat deklarálni, ha akarnánk. Pl. megjeleníthetnénk egy pontot.)

Az ilyen ős "adóállomásként" is szerepel. Ha változtatni akarunk valamit a teljes hierarchiában, akkor azt itt lehet legegyszerűbben megtenni (pl. szín hozzáadása). Természetesen lehet a változtatás olyan, hogy minden leszármazottat is módosítani kell a megfelelő virtuális eljárással.

3.8. Objektum típus-kompatibilitás

Az öröklés némiképp megváltoztatja a Turbo Pascal típus-kompatibilitás szabályait. Ráadásul még a leszármazott típus az összes őstípusától örökli a típus-kompatibilitást. A kiterjesztett típus-kompatibilitásnak három esete van:

objektum példányok között
objektum példányokra mutató pointerek között
aktuális és formális paraméterek között

Ezekben az esetekben a típus-kompatibilitás csak egy irányban öröklődik: a leszármazott típustól az őstípus felé. Csak az alábbi értékadás lehetséges:

őstípus_változó := leszármazott_típus_változó.

A leszármazott típusnak vagy pontosan annyi, vagy több adata van, mint az őstípusnak, ezért nem lehetséges a fordított értékadás, mert a leszármazott típusnak definiálatlan mezői maradnának, amely illegális lenne. Példaként nézzük meg az alábbi deklarációkat:

```
type
     LocationPtr = ^Location;   { Egy objektumtípushoz mindig     }
     PointPtr    = ^Point;      { célszerű deklarálni a megfelelő }
     CirclePtr   = ^Circle;     { mutató típust is.               }
var
     ALocation   : Location;
     APoint      : Point;
     ACircle     : Circle;
     PLocation   : LocationPtr;
     PPoint      : PointPtr;
     PCicle      : CirclePtr;
```

A fenti deklarációt figyelembevéve, az alábbi értékadások megengedettek:

```
ALocation := APoint;
APoint    := ACircle;
ALocation := ACircle;
```

A típus kompatibilitás érvényes az objektumokra mutató pointerek között, ugyanaz a szabály, mint az objektum típusok példányaira. Ha pointerek mutatnak a leszármazottakra, akkor

```
PPoint    := PCircle;
PLocation := PPoint;
PLocation := PCircle;
```

Az értékadás fordított sorrendben nem lehetséges.

3.9. Eljárás vagy metódus?

Készítsünk egy olyan eljárást, amely bármilyen típusú alakzatot mozgat a képernyőn. Hagyományos Pascal gondolkodásmóddal ezt úgy csinálnánk, hogy a mozgató eljárás **var** paraméterként kapná meg az alakzatot. Legyen ez a paraméter *Point* típusú, mert akkor aktuális paraméterként *Point* minden leszármazottja is szerepelhet, még az utólag kitaláltak is.

```
procedure DragIt(var AnyFigure : Point; DragIt : integer);
  var
    DeltaX, DeltaY : integer;
    FigureX, FigureY : integer;
```

```
begin
  AnyFigure.Show;
  FigureX := AnyFigure.GetX;
  FigureY := AnyFigure.GetY;
  while GetDelta(DeltaX, DeltaY) do begin
    FigureX := FigureX + (DeltaX * DragBy);
    FigureY := FigureY + (DeltaY * DragBy);
    AnyFigure.MoveTo(FigureX, FigureY);
  end;
end;
```

Az eljárás nem tudja, hogy milyen típust kap, csak azt, hogy az a *Point* leszármazottja és ez elég is. Minden leszármazott örökli ugyanis a *Point* metódusait, amelyeket a mozgató eljárás használ.

A *GetX*, *GetY* és a *MoveTo* metódusok statikusak, amely azt jelenti, hogy a *DragIt* eljárás ismeri a fordítási időben keletkező címüket. A *Show* metódus azonban virtuális, a *DragIt* eljárás a *Show* metódus címét az aktuális objektum példány VMT-jéből (Virtual Method Table = virtuális metódus tábla) veszi. Ha a példány *Circle,* akkor a *DragIt* a *Circle.Show* metódust hívja meg. Az eljárás jól működik és mozgatja a *Point* leszármazott típusait a képernyőn, ha a típus létezik a grafikus toolbox-ban a fordításkor. Ha viszont bármilyen alakzatot akarunk mozgatni a képernyőn, akkor írjunk olyan programot, hogy a grafikus objektumok magukat mozgassák. Vagyis eljárás helyett metódust kell írnunk. Ha azt is ki akarjuk használni, hogy a későbbiek folyamán a felhasználó felülbírálhatja a *DragIt* metódust, akkor a metódusnak virtuálisnak kell lennie.

```
Point = object(Location)
  Visible : boolean;
  constructor Init(InitX, InitY : integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  function IsVisible : boolean;
  procedure MoveTo(NextX, NextY : integer);
  procedure Drag(DragBy : integer); virtual;
end;
```

3.10. Statikus vagy virtuális metódus

Általában érdemes virtuális metódusokat használni, még akkor is, ha statikus metódusok esetében a sebesség és a memória kihasználása optimálisabb.

Például deklaráljunk egy őstípusú objektumot és belül deklaráljunk névvel ellátott metódusokat. Hogyan döntsünk, virtuális vagy statikus metódusaink legyenek-e. Van egy szabály: ha virtuális metódust készítünk, akkor később az őstípus leszármazottja felülírhatja a virtuálisnak deklarált metódust, és az hozzáférhető lesz az őstípus számára.

Pontosan ezt a szabályt alkalmaztuk a grafikus objektumoknál. Ez

esetben a *Point* őstípusú objektumnál el kellett határozni, hogy melyik metódus legyen statikus és melyik virtuális: a *Show*, *Hide* metódusok virtuálisak és a *MoveTo* metódus statikus. Ennek következtében minden alakzat saját maga jeleníti meg és tünteti el magát. A grafikus alakzatok mozgatása úgy tűnik, hogy azonos mindegyik esetben. A *Point.MoveTo* statikus metódus örökli a *Point* tulajdonságait, de a *Show* és a *Hide* metódusoknak virtuálisnak kell lennie, hogy a *Point.MoveTo* metódus a leszármazott típus *Show* és *Hide* metódusait hívhassa.

A virtuális metódusok az adatmezőiket a VMT-n keresztül érik el, míg a statikus metódusok közvetlenül hívhatók, ezért gyorsabbak a virtuális metódusok hívásánál. A sebesség és a memória méret a statikus metódusoknál kedvezőbb, de a virtuális metódusok flexibilisebbek.

3.11. Dinamikus objektumok

Ezideig minden példában az objektumok a **var** deklarációban foglalták le a helyet az adat szegmensben és a stack-ben.

```
var
  ACircle : Circle;
```

Az objektumok helyfoglalása dinamikusan is történhet a heap-ben. A Turbo Pascal 6.0 változat is rendelkezik olyan utasításokkal, amelyek az objektumok dinamikus helyfoglalását és a helyfoglalás felszabadítását könnyen és hatásosan végzik. Dinamikus objektumok létrehozása a következőképpen történik: az objektumra mutató pointert deklarálunk majd a **new** eljárással helyet foglalunk számára:

```
var
  PCircle  : ^Circle;
.
.
begin
.
.
  new(PCircle);
```

Ha dinamikus objektum virtuális metódusokat tartalmaz, akkor kötelező azt a megfelelő konstruktorral (lásd 3.11.3. fejezetet) inicializálni:

```
PCircle^.Init(500, 100, 50);
```

Ha a dinamikus objektum statikus metódusokat tartalmaz az aktiválás a követekző módon történik:

```
OldXPosition := PCircle^.GetX;
```

3.11.1. Helyfoglalás és kezdeti értékadás a new eljárással

A Turbo Pascal kiterjesztette a **new** eljárás működését az objektum heap-beli helyfoglalására és kezdeti érték adására. A **new** eljárásnak két paramétere van, az első a pointer típus neve, a második a megfelelő konstruktor a szükséges paramétereivel:

```
new(PCircle, Init(500, 100, 50));
```

A **new** eljárást hívhatjuk egy paraméterrel is, ekkor a **new** függvényként fog viselkedni bemenő paramétere az objektumtípusra mutató típus neve és visszatérési értéke az objektumra mutató pointer lesz:

```
type
    ArcPtr = ^Arc;
var
    PArc  : ArcPtr;

begin
  PArc := new(ArcPtr);
```

Megjegyezzük, hogy a **new** függvény ezen hívási módja az összes adattípusra alkalmazható, nemcsak az objektumokra:

```
type
    CharPtr = ^Char;

var
    PChar : CharPtr;

begin
  PChar := new(CharPtr);
```

A **new** függvényhívásnál szintén használhatunk második paramétert is:

```
PArc := new(ArcPtr, Init(500, 100, 50);
```

3.11.2. Dinamikus objektumok felszabadítása dispose eljárással

Hasonlóan a hagyományos Pascal dinamikus változóihoz, a dinamikus objektumú példányok helyfoglalásának a felszabadítása a heap-ből a **dispose** eljárással történik:

```
dispose(PCircle);
```

Egy objektum tartalmazhat struktúrákra vagy objektumokra mutató pointereket, melyeket különleges sorrendben kell felszabadítani vagy eltávolítani, főleg, amikor bonyolult dinamikus struktúrák egymásba vannak skatulyázva. Ilyenkor kell egy dinamikus objektumot készíteni a törlés végrehajtására egy egyszerű metódusban. Így az objektum példányok egyetlen metódushívással felszabadíthatók:

```
MyComplexObject.Done;
```

3.11.3. Konstruktor

A **constructor** egy speciális eljárás, virtuális metódusok inicializálására használjuk.

Annak az objektumtípusnak, amely virtuális metódust tartalmaz, szükséges egy konstruktor (ajánlott név: *Init*), amelyet az objektum minden példányára meg kell hívni, még mielőtt bármelyik virtuális metódusát hívnánk.

Tulajdonképpen a konstruktor feladata: megteremti a kapcsolatot a konstruktort hívó objektum példány és az objektum típus VMT között. A VMT tartalmazza az objektum típus méretét és egy-egy mutatót, amely a benne lévő virtuális metódust implementáló kódra mutat.

Fontos megjegyezni, hogy a VMT csak típusonként létezik. A példányok csak kapcsolatot teremtenek vele.

Ha elfelejtjük a konstruktort hívni, akkor rendszerlefagyás lesz a következmény, ezt elkerülhetjük, ha fordításkor bekapcsoljuk a Range check opciót {$R+}. Ha a program már jól működik, ismételjük meg a fordítást {$R-} kapcsolóval.

A grafikus alakzatok objektumaiban használjuk a **virtual** kulcsszót a metódusok deklarálásakor:

```
type
     Location = object
          X, Y : integer;
          procedure Init(InitX, InitY : integer);
          function GetX : integer;
          function GetY : integer;
     end;

     Point = object(Location)
          Visible : Boolean;
          constructor Init(InitX, InitY : integer);
          procedure Show; virtual;
          procedure Hide; virtual;
          function IsVisible : Boolean;
          procedure MoveTo(NewX, NewY : integer);
     end;
```

```
Circle = object(Point)
    Radius : integer;
    constructor Init(InitX, InitY : integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Expand(ExpandBy : integer); virtual;
    procedure Contract(ContractBy : integer); virtual:
end;
```

Látható, hogy a *Point* és *Circle* metódusoknak van *Show* és *Hide* metódusa, amelyek virtuálisak. Ha egy őstípusnak vannak virtuális metódusai, akkor a leszármazottak ezeket szintén öröklik. Statikus metódust nem lehet virtuálissal helyettesíteni.
Fontos szabály: Minden objektumtípusnak, amelyek virtuális metódusokat tartalmaznak, rendelkezniük kell konstruktorral.

- A konstruktort bármely virtuális metódus hívása előtt kell aktivizálni.

- Egy objektum minden egyedi példányát inicializálni kell megfelelő konstruktor hívással. Nem elegendő egy objektum inicializált példányát egy származtatott példánynak értékként adni. A származtatott példányoknak mindig a korrekt értékkel kell rendelkezniük, értékadó utasítással nem inicializálhatók és rendszerlefagyás következik be, ha a virtuális metódusaik hívásra kerülnek. Például:

```
var
    QCircle, RCircle: Circel;     { a Circle két példánya }
                                  { jön létre             }
begin
    QCirle.Init(600, 100, 30);    { QCirle konstruktorának }
                                  { hívása                 }
    RCirle := QCirle;             { RCirle nem kap értéket!}
```

3.11.4. Destruktor

A Turbo Pascal speciális metódusa: a **destructor**, amely a dinamikusan helyet foglaló objektum példányok törlésére és megszüntetésére használható.

A **destructor**-t a típus definícióban a többi metódussal együtt kell definiálni minden objektumban:

```
type
    Point = object(Location)
      Visible : Boolean;
      Next    : PointPtr;
      constructor Init(InitX, InitY : integer);
```

```
        destructor Done; virtual;
        procedure Show; virtual;
        procedure Hide; virtual;
        function IsVisible : Boolean;
        procedure MoveTo(NewX, NewY : Integer);
        procedure Drag(DragBy : Integer); virtual;
      end;

  var
      PPoint = ^Point;
```

Mivel a **destructor** is örökölhető, ezért a deklarációban statikus és virtuális is lehet. Ha a feladat gyakran hivatkozik különböző típusú objektumokra, amelyeket meg kell szüntetni, akkor mindenképpen a **destructor** metódust virtuálisan kell deklarálni, így esetenként a megfelelő **destructor** lesz végrehajtva.

A sokalakú, heap-ben elhelyezett objektumok felszabadítására a **dispose** eljárás nem tudja, hogy hány byte-ot kell felszabadítani. A **destructor** oldja meg ezt a problémát, mert az adatok tárolt információit a VMT-n keresztül éri el.

Késői kötésnél a memória felszabadítását a kiterjesztett **dispose** eljárással kell elvégeztetni, oly módon, hogy a **destructor** hívása paraméterként szerepeljen.

```
dispose(PPoint, Done);
```

Megjegyezzük, hogy a **destructor** metódus törzse lehet üres is:

```
destructor AnObject.Done;
  begin
  end;
```

Hiába üres a törzse, a fordító program a **destructor** azonosítót felismerve a Pascal programozó számára láthatatlan, a megfelelő memóriatörlést elvégző programkódot generál az üres törzsbe.

3.12. Belső adatszerkezetek

3.12.1. Objektumok belső adatformátuma

Az objektumok adatformátuma hasonló a rekordokéhoz. Az objektum adatmezői a deklarálás sorrendjében következnek. A leszármazott típusnak az új adatmezői az őstípustól örökölt adatmezők mögé kerülnek. Ha egy objektum virtuális metódusokat, konstruktorokat vagy destruktorokat definiál, akkor a fordító egy 16 bites területet foglal le az objektumtípusnál, melyet virtuális metódus táblának (VMT) neveznek. Itt kerül tárolásra az objektumtípus adat szegmesben lévő VMT eltolási címe. A VMT mező a normál adatok tárolása után kezdődik. Amikor egy objektumtípus virtuális metódusokat, konstruktorokat vagy

destruktorokat örököl, akkor örökli a VMT mezőt is. A VMT mezők inicializálását az objektumtípusok konstruktorai végzik.

Nézzük meg újra az ismert programrészletet adattárolás szempontjából is:

```
type
     LocationPtr = ^Location;

     Location = object
          X, Y : integer;
          procedure Init(PX, PY : integer);
          function GetX : integer;
          function GetY : integer;
     end;

     PointPtr = ^Point;

     Point = object(Location)
          Color : integer;
          constructor Init(PX, PY, PColor : integer);
          destructor Done; virtual;
          procedure Show; virtual;
          procedure Hide; virtual;
          procedure MoveTo(PX, PY : integer);
     end;

     CirclePtr = ^Circle;

     Circle = object(point)
          Radius : integer;
          constructor Init(PX, PY, PColor, PRadius : integer);
          procedure Show; virtual;
          procedure Hide; virtual;
          procedure Fill; virtual;
     end;
```

Az adattárolás az alábbiakban jön létre:

A *Location* típusú objektum egy pont helyének az *X* és *Y* helykoordinátáit tartalmazza.

A *Point* típusú objektum örökli a *Location* típus *X*, *Y* változóit, amelyeket a saját *Color* változója követ, ezután helyezkedik el a virtuális metódusok használatához szükséges VMT eltolási címe.

A *Circle* objektum *Point* típusú, ennek következtében az adatmezeje a *Point* összes adatát tartalmazza, melyet a *Radius* saját változója követ.

Location	Point	Circle
X	X	X
Y	Y	Y
	Color	Color
	VMT	VMT
		Radius

A példában megadott objektumok VMT táblainak szerkezete:

Point VMT	Circle VMT
$0008	$000A
$FFF8	$FFF6
@Point.Done	@Point.Done
@Point.Show	@Circle.Show
@Point.Hide	@Circle.Hide
@Point.MoveTo	@Point.MoveTo
	@Circle.Fill

3.12.2. Virtuális metódus tábla (VMT)

Minden objektum, amely tartalmaz, vagy örököl virtuális metódusokat, konstruktorokat vagy destruktorokat, azok címei bekerülnek a VMT táblába. A VMT táblát a fordító hozza létre automatikusan.

A VMT tábla első adata a hivatkozott objektum típusok byte-okban kifejezett méretét tartalmazza. A második adata az előző adattal egyezik, de az előjele negatív (negatív méret). Ennek az adatnak a felhasználásával történik a VMT tábla helyes felépítésének ellenőrzése. A virtuális hívások érvényességét a {$R+} fordítási direktívával is ellenőrizhetjük.

Nézzük meg, hogy a *Circle* objektum hogyan örökli a *Done* és a *MoveTo* metódusokat, ugyanakkor felülírja a *Show* és a *Hide* metódusokat.

A **sizeof** függvény visszatérési értéke megadja az objektum típus VMT mezőn tárolt méretét.

3.13. Összefoglaló az ojektum-orientált programozás alapfogalmairól

ANCESTOR TYPE *(őstípus)*

Az a típus, amelytől egy másik típus örököl.

BINDING *(összerendelés)*

A hívó és a hívott eljárás összerendelése a hívott eljárás címének átadásával.

CONSTRUCTOR

Virtuális metódusokat is tartalmazó objektum inicializálására szolgáló speciális metódus.

DESCENDANT TYPE *(leszármazott típus)*

Az a típus, amely örököl egy másik típustól.

DESTRUCTOR

A feleslegessé vált objektum heap-beli helyének felszabadítására szolgáló speciális metódus.

EARLY BINDING *(korai összerendelés)*

A hívó és a hívott eljárás összerendelése fordítási/szerkesztési időben.

INSTANCE *(példány)*

Az objektum típusnak egy változója.

LATE BINDING *(késői összerendelés)*

A hívó és a hívott eljárás összerendelése futási idő alatt.

METHOD *(metódus)*

Egy objektum típushoz tartozó függvény vagy eljárás, mely lehet statikus vagy virtuális.

OBJECT *(objektum)*

Az objektum típusnak egy változója (példánya).

OBJECT HIERARCHY *(objektum hierarchia)*

Öröklés útján egymáshoz rendelt objektum típusok rendszere (családfája).

OBJECT TYPE *(objektum típus)*

A Pascal rekord típusához hasonló speciális adatszerkezet, mely együtt tartalmazza az adatokat és az azokat kezelő metódusokat (függvényeket és eljárásokat).

STATIC METHOD *(statikus metódus)*

Olyan metódus, melynek címét fordításkor rögzítettük, tehát futás alatt nem változhat (ugyanúgy, mint a hagyományos Pascalban). Tárban és időben kedvezőbb, mint a flexibilisebb virtuális metódus.

VIRTUAL METHOD *(virtuális metódus)*

Olyan metódus, melynek címét futási idő alatt kapja meg a hívó program, így egy objektum típus különböző leszármazottai igényüknek megfelelően használhatják.

3.14. Példa az objektum-orientált programozásra

A MANGEN.PAS program Mandelbrot halmaz rajzolását valósítja meg objektumok segítségével, elindításakor egy menü jelenik meg, és a következő menüpontok közül választhatunk:

Rajzolás,
Megjelenítés és
Kilépés.

Rajzolás menüben a program megkérdezi a kép nevét (ebbe a file-ba menti el a képet a program), a kép középpontjának komplex koordinátáit (valós és képzetes részét külön), a nagyítás értékét, majd a maximális iterációszámot. Az adatok bevitele után a program elkezdi a kép generálását. Ez három szakaszban történik: először durva felbontásban, majd két egyre finomabb felbontásban. A generálás után a képet kimenti a megadott file-ba és a menü jelenik meg újra.

Megjelenítés menüben a program megkérdezi a kép file-nevét, amelyet kiterjesztés nélkül kell megadni és ha a file létezik, akkor betölti, egyébként hibát jelez.

A programból a Kilépés menüponttal lehet kilépni.

A MANGEN.PAS program listája:

```
{-------------------------------------------------------------------}
{ Mandelbrot halmaz rajzolása objektumokkal                         }
{-------------------------------------------------------------------}

program Mandelbrot_objektumokkal;

uses Graph, Crt;

type
  FileNameType = string[8];
  valos = real;

  Complex = object
    Re, Im : real;
    procedure Init(R, I : real);
    procedure Osszeadas(C : Complex);
    procedure Szorzas(C : Complex);
    function Norm : real;
    function Real : real;
    function Imag : real;
  end;
```

```
  Mandelbrot = object(Complex)
    Magnify, Size : real;
    Maxiter : word;
    procedure Init(R, I, M : real; Max : word);
    function Color(R, I : real) : word;
  end;

  MandelPic = object(Mandelbrot)
    FPic  : string[4];
    n, XCenter, YCenter : word;
    S : boolean;
    Asp : real;
    procedure Init(R, I, M : real; Max : word);
    procedure Point(XX, YY, Len : word);
    procedure Drawing;
    procedure Save(FName : FileNameType);
    procedure Load(FName : FileNameType);
  end;

var
  Mandel : MandelPic;

procedure Writexy(X, Y : word; S : string); forward;

{--------------------------------------------------------------------}
{ Complex objektum metódusai                                         }
{--------------------------------------------------------------------}

procedure Complex.Init(R, I : real);
  begin
    Re := R;
    Im := I;
  end;

procedure Complex.Osszeadas(C : Complex);
  begin
    Re := Re + C.Re;
    Im := Im + C.Im;
  end;

procedure Complex.Szorzas(C : Complex);
  var
    R : valos;
  begin
    R := Re * C.Re - Im * C.Im;
    Im := Re * C.Im + Im * C.Re;
    Re := R;
  end;
```

```
function Complex.Norm : real;
  begin
    Norm := Re * Re + Im * Im;
  end;

function Complex.Real : real;
  begin
    Real := Re;
  end;

function Complex.Imag : real;
  begin
    Imag := Im;
  end;

{------------------------------------------------------------------}
{ Mandelbrot objektum metódusai                                    }
{------------------------------------------------------------------}

procedure Mandelbrot.Init(R, I, M : real; Max : word);
  begin
    Complex.Init(R, I);
    Magnify := M;
    Maxiter := Max;
  end;

{ Egy pont színének a meghatározása : }

function Mandelbrot.Color(R, I : real) : word;
  var
    Iter : word;
    Z, C : Complex;
  begin
    Iter := 0;
    Z.Init(0, 0);
    C.Init(R, I);

    while ((Iter < Maxiter) and (Z.Norm < 4)) do
      begin
        Z.Szorzas(Z);
        Z.Osszeadas(C);
        inc(Iter);
      end;

    Color := Iter mod getmaxcolor + 1;
    if Iter = Maxiter then Color := 0;
  end;

{------------------------------------------------------------------}
{ MandelPic objektum metódusai                                     }
{------------------------------------------------------------------}
```

```
{ Grafikus inicializálás : }

procedure MandelPic.Init(R, I, M : real; Max : word);
  var
    GraphDriver, GraphMode : integer;
    XAsp, YAsp : word;

  begin
    Mandelbrot.Init(R, I, M, Max);
    clrscr;
    detectgraph(GraphDriver, GraphMode);

    if ((GraphDriver <> CGA) and (GraphDriver <> EGA) and
       (GraphDriver <> VGA))
    then begin
      writeln('Grafikus hiba!');
      halt(1);
    end;

    if (GraphDriver = CGA)
    then begin
      GraphMode := CGAC1;
      n := 100;
      FPic :=  '.CGA';
    end
    else
    if (GraphDriver = EGA)
    then begin
      n := 204;
      FPic := '.EGA';
    end
    else begin
      n := 300;
      FPic := '.VGA';
    end;

    initgraph(GraphDriver, GraphMode, '');
    getaspectratio(XAsp, YAsp);
    Asp := XAsp / YAsp;

    XCenter := (getmaxx - n) div 2;
    YCenter := (getmaxy - n) div 2;
  end;

{ A pontok kirajzolása : }

procedure MandelPic.Point(XX, YY, Len : word);
  var
    R, I : valos;
    Col  : word;
  begin
    R := XX * Size * Asp + Re;
```

```
    I := YY * Size - Im;

    Col := Color(R, I);

    if Len <> 0 then begin
      setfillstyle(solidfill, Col);
      bar(XCenter + XX, YCenter + YY,
          XCenter + XX + Len, YCenter + YY + Len);
    end
    else
      putpixel(XCenter + XX, YCenter + YY, Col);

    if S then
      if Len <> 0 then begin
        setfillstyle(solidfill, Col);
        bar(XCenter + XX, YCenter + n - YY,
            XCenter + XX + Len, YCenter + n - YY + Len);
      end
      else
        putpixel(XCenter + XX, YCenter + n - YY, Col);
  end;

{ A Mandelbrot halmazt generáló ciklusok : }

procedure MandelPic.Drawing;
  var
    X, Y, NY : word;
  begin
    NY := n; S := false;
    if Im = 0 then begin
      NY := NY div 2;
      S := true;
    end;
    Re := Re - 1 / Magnify;
    Im := Im + 1 / Magnify;
    Size := 2 / (n * Magnify);

    Y := 0;
    repeat
      X := 0;
      repeat
        Point(X, Y, 3);
        inc(X, 4);
      until X > round(n / Asp);
      inc(Y, 4);
    until Y > (NY - 1);

    Y := 0;
    repeat
      X := 0;
```

```
      repeat
        Point(X + 2, Y, 1);
        Point(X, Y + 2, 1);
        Point(X + 2, Y + 2, 1);
        inc(X, 4);
      until X > round(n / Asp);
      inc(Y, 4);
    until Y > (NY - 1);

    Y := 0;
    repeat
      X := 0;

      repeat
        Point(X + 1, Y, 0);
        Point(X, Y + 1, 0);
        Point(X + 1, Y + 1, 0);
        inc(X, 2);
      until X > round(n / Asp);
      inc(Y, 2);
    until Y > (NY - 1);
  end;

{ A kép elmentése : }

procedure MandelPic.Save(FName : FileNameType);
  var
    F : file;
    Msize : word;
    P : pointer;
  begin
    Msize := imagesize(XCenter, YCenter,
                       round(XCenter + n / Asp), YCenter + n);
    getmem(P, Msize);

    getimage(XCenter, YCenter,
             round(XCenter + n / Asp), YCenter + n, P^);

    assign(F, FName + FPic);
    rewrite(F, 1);
    if ioresult <> 0 then begin
      close(f);
      closegraph;
      Writexy(25, 10, 'Hiba a file nyitásánál!');
      exit;
    end
    else begin
      blockwrite(F, P^, Msize);
      close(F);
    end;
    freemem(P, Msize);
  end;
```

```
{ A kép behívása : }

procedure MandelPic.Load(FName : FileNameType);
  var
    F : file;
    Msize : word;
    P : pointer;

  begin
    Msize := imagesize(XCenter, YCenter,
                       round(XCenter + n / Asp), YCenter + n);
    getmem(P, Msize);


    findfirst(FName + FPic, anyfile, SR);
    if doserror <> 0 then begin
      closegraph;
      textcolor(red);
      Writexy(25, 10, 'Nincs ilyen nevű file.');
      exit;
    end

    else begin
      assign(F, FName + FPic);
      reset(F, 1);
      blockread(F, P^, Msize);
      close(F);

      putimage(XCenter, YCenter, P^, normalput);
    end;

    freemem(P, Msize);
  end;

{-----------------------------------------------------------------------}
{ Segédeljárások                                                        }
{-----------------------------------------------------------------------}

procedure Writexy(X, Y : word; S : string);
  begin
    gotoxy(X, Y);
    write(S);
  end;

procedure Header(Title : string);
  begin
    window(19, 4, 65, 6);
    textbackground(black);
    clrscr;
    window(17, 3, 63, 5);
    textbackground(red);
    textcolor(white);
```

```
    clrscr;
    Writexy(round((49-length(Title)) / 2), 2, title);
    window(1, 1, 80, 25);
    textbackground(lightblue);
  end;

{------------------------------------------------------------------}
{ Menü kirajzolása és választás függvénye                          }
{------------------------------------------------------------------}

function Menu : word;
  var
    Mn : integer;
    Ch : char;
  begin
    window(1, 1, 80, 25);
    textbackground(lightblue);
    clrscr;
    Header('M A N D E L B R O T  halmaz rajzoló program');

    textbackground(black);
    window(32, 10, 52, 18);
    clrscr;
    textbackground(cyan);
    window(30, 9, 50, 17);
    clrscr;
    textcolor(yellow); Writexy(5, 2, 'R');
      textcolor(white); write('ajzolás');
    textcolor(yellow); Writexy(5, 5, 'M');
      textcolor(white); write('egjelenítés');
    textcolor(yellow); Writexy(5, 8, 'K');
      textcolor(white); write('ilépés');
    textbackground(lightblue);
    window(1, 1, 80, 25);

    Mn := 0;
    repeat
      Ch := readkey;
      case Ch of
      'R', 'r' : Mn := 1;
      'M', 'm' : Mn := 2;
      'K', 'k' : Mn :=3;
      end;
    until (Mn = 1) or (Mn = 2) or (Mn = 3);
    Menu := Mn;
  end;

{------------------------------------------------------------------}
{ Rajzolás menüpont eljárása                                       }
{------------------------------------------------------------------}
```

```
procedure Drawingmenu;
  var
    Xm, Ym, Ma : real;
    Mx : word;
    FN : FileNameType;
  begin
    textcolor(white);
    clrscr;
    Header('R A J Z O L Á S');
    Writexy(10, 10, 'A kép neve :');
    Writexy(10, 14, 'A középpont');
    Writexy(17, 16, 'valós értéke      :');
    Writexy(17, 18, 'képzetes értéke  :');
    Writexy(10, 20, 'Nagyítás (2 / képméret) :');
    Writexy(10, 22, 'Maximális iterációszám  :');

    gotoxy(23, 10);
    FN := '';
    readln(FN);

    if length(FN) <> 0 then begin
      gotoxy(36, 16);
        readln(Xm);
      gotoxy(36, 18);
        readln(Ym);

        repeat
          gotoxy(36, 20);
          readln(Ma);
        until Ma > 0;

        repeat
          gotoxy(36, 22);
          readln(Mx);
        until Mx > 0;

        Mandel.Init(Xm, Ym, Ma, Mx);
        Mandel.Drawing;
        Mandel.Save(FN);

        sound(440);
        delay(100);
        nosound;

        closegraph;
    end;
  end;

{-------------------------------------------------------------------}
{ Megjelenítés menüpont eljárása                                    }
{-------------------------------------------------------------------}
```

```
procedure Viewmenu;
  var
    FN : FileNameType;
  begin
    textcolor(white);
    clrscr;
    Header('M E G J E L E N I T É S');
    Writexy(10, 10, 'A kép neve :');

      gotoxy(23, 10);
      FN := '';
      readln(FN);

    if length(FN) <> 0 then begin
      Mandel.Init(0, 0, 0, 0);
      Mandel.Load(Fn);
      repeat
      until readkey <> #0;
      closegraph;
    end;
  end;

{-------------------------------------------------------------------}
{ Főprogram eljárása                                                }
{-------------------------------------------------------------------}

procedure Main;
  var
    Choose : word;
  begin
    repeat
      Choose := Menu;
      case Choose of
        1 : Drawingmenu;
        2 : Viewmenu;
        3 : begin
              textbackground(BLACK);
              clrscr;
            end;
      end;
    until Choose = 3;
  end;

{-------------------------------------------------------------------}
{ Főprogram                                                         }
{-------------------------------------------------------------------}

begin

  Main;

end.
```

3.15. Objektum-orientált programozáshoz kapcsolódó rutinok

Fail eljárás

```
Fail;
```

Csak a konstruktoron belül hívható, hívásakor a konstruktor felszabadítja a dinamikus objektum helyét, amit előzőleg lefoglalt. Ez az eljárás hívható, ha a konstruktorban operációs hiba lép fel.

SizeOf függvény

```
SizeOf(x) : integer;
```

Ha a függvény paramétere egy objektum típus példánya, amelynek virtuális metódus táblája van (VMT), a függvény visszatér a VMT tábla méretével.

TypeOf függvény

```
TypeOf(x) : pointer;
```

Visszatér az objektum típusú változóhoz tartozó VMT címével.

4. A TURBO PASCAL ÉS AZ ASSEMBLY NYELV KAPCSOLATA

4.1. A beépített (inline) assembler

A Turbo Pascal 6.0 tartalmaz egy beépített assembler fordítót, amely lehetővé teszi, hogy közvetlenül a Pascal forrásprogramban 8086/8087 ill. 80286/80287 assembler utasításokat helyezzünk el. Ez a fordító magában foglalja a Turbo Assembler és a Microsoft Macro Assembler lehetőségeinek nagy részét. Mivel azonban a Turbo Pascal 6.0 mégiscsak egy Pascal fordító, a fent említett két assembler ismeretében figyelembe kell vennünk az alábbi megszorításokat.

A **DB, DW,** és **DD** (byte, szó és dupla szó definiálás) kivételével semmilyen más Turbo Assembler direktíva nem használható. Definiálhatunk különféle műveleteket assembly nyelven, azonban amire ezek vonatkoznak általában valamilyen Pascal adatszerkezet. Például az **EQU**-nak a **const, var** és **type** deklaráció, a **PROC**-nak a **procedure** és **function** ill. a **STRUC**-nak a **record** struktúra felel meg. Lényegében tehát a Turbo Pascal beépített assemblere egy olyan assembler fordító, amely a deklarációiban a Pascal szintaxist használja.

4.1.1. Az asm utasítás

A beépített assembler fordító az **asm** utasítás segítségével érhető el, melynek felépítése:

asm asm_utasítás < szeparátor asm_utasítás > **end**;

ahol az asm_utasítás egy assembler utasítás, a szeparátor lehet pontosvessző, új sor vagy Pascal megjegyzés. Nézzünk néhány példát az **asm** használatára:

```
if it_ok then
   asm
     sti
   end
 else
   asm
     cli
   end;

asm
  mov ax,bal;  xchg ax,jobb;  mov bal,ax;
end;
```

```
asm
  mov   ah,0              { billentyűzet olvasása }
  int   16h                          { BIOS hívás }
  mov   kar_kod,al         { az ASCII kód tárolása }
  mov   bill_kod,ah    { a billentyűzet kód tárolása}
end;
```

4.1.1.1. Regiszterek használata

A regiszterek használatára vonatkozó szabályok az **asm** utasításban megegyeznek az **external** (külső) assembler rutinokra vonatkozókkal. Az első és legfontosabb kikötés, hogy a BP, SP, SS és DS regiszterek tartalmát mindenképpen meg kell őrizni. A többi regiszter - AX, BX, CX, DX, SI, DI, ES és az állapotregiszter (Flags) - szabadon felhasználható. Belépve az **asm** utasításba, a BP a veremben kialakított adatstruktúrára (stack frame) mutat, az SP a verem tetejére mutat, SS tartalmazza a verem és DS az adatszegmens szegmenscímét - a többi regiszter tartalma határozatlan.

4.1.2. Az assembler utasítások felépítése

Az assembler utasítás szintaxisa:

```
[címke ":" ] <prefixum> [műveleti kód [operandus <","> operandus>]]
```

ahol a műveleti kód valamely assembler utasítást vagy direktívát jelöl.

4.1.2.1. Címkék

Az **asm** utasításban kétféle címkét használhatunk. Az egyik tulajdonképpen egy Pascal címke, amelyet a **label** deklarációval adhatunk meg abban a blokkban amely az **asm** utasítást tartalmazza. A másik címke az ún. lokális címke, amelyet a @ kezdőkarakterrel az **asm** utasításon belül definiálhatunk. A lokális címke a **label** utasítással nem deklarálható és csak az őt tartalmazó **asm** utasításban ismert ill. használható. Az alábbi programrészlet a címkék lehetséges felhasználását mutatja be:

```
label  start,stop;
 ...
begin
  asm
    start:
     ...
       jz  stop
    @1:
     ...
       loop  @1
  end;
 ...
  asm
    @1:
     ...
       jc  @2
     ...
       jmp @1
    @2:
  end;
  goto start;
  ....
stop:
end.
```

4.1.2.2. Prefixum műveleti kódok

A beépített assemblerben a következő prefixumok használhatók:

LOCK	a rendszerbusz lezárása
REP	sztring művelet ismétlése
REPE/REPZ	sztring művelet ismétlése amíg egyenlő
REPNE/REPNZ	sztring művelet ismétlése amíg nem egyenlő
SEGCS	CS (kódszegmens) átdefiniálása
SEGDS	DS (adatszegmens) átdefiniálása
SEGES	ES (extra szegmens) átdefiniálása
SEGSS	SS (stack szegmens) átdefiniálása

A prefixumok használatát az alábbi programrészlet illusztrálja:

```
asm
  rep  movsb            { CX db byte másolása DS:SI-ről ES:DI-re }
  SEGES lodsw           { szó betöltése ES:SI-ről }
  SEGCS mov ax,[bx]     { mint a MOV AX,CS:[BX] }
  SEGES
  mov WORD PTR [di],0   { mint a MOV WORD PTR ES:[DI],0 }
end;
```

4.1.2.3. Az assembler utasítások

A beépített assembler támogatja a 8086/8087 és 80286/80287 mikroprocesszorok teljes utasításkészletét. A 8087 utasítások a **{$N+}**, a 80286 utasítások a **{$G+}** és a 80287 utasítások a **{$G+,$N+}** állapotok kijelölésével használhatók. Az utasítások teljes leírása a 80x86/80x87 hivatkozási kézikönyvében található. Az alábbiakban azokkal az utasításokkal foglalkozunk részletesen, amelyek működése eltér a az assembler fordítókban használtaktól.

A beépített assembler magára vállalja a **RET** és a **JMP** utasítások hatótávolságának megállapítását.

A **RET** utasítás lefordítása során vagy közeli (near), vagy távoli (far) visszatérés gépikódja keletkezik, attól függően, hogy milyen az a programkörnyezet, amelyben elhelyezkedik.

```
procedure NearProc; near;
begin
  asm
    ret        { közeli ret - retn generálódik }
  end;
end;

procedure FarProc; far;
begin
  asm
    ret        { távoli ret - retf generálódik }
  end;
end;
```

Ha a **RETN** ill. **RETF** utasításokat használjuk, akkor természetesen nem működik ez az automatizmus.

A beépített assembler optimalizálja az ugró utasítások fordítását, a legrövidebb és legmegfelelőbb utasítások kiválasztásával. Ez az automatizmus a feltétel nélküli ugrásra (**JMP**) és az összes feltételes ugrásra (**Jcc**) vonatkozik. Ha a címke az ugrás helyétől -128 - +127 byte távolságra helyezkedik el, két byte-os rövid (short) ugrás generálódik. Amennyiben feltételes ugrásnál a címke kívül esik a fenti intervallumon, a fordító egy közeli ugrást generál a következő módon:

```
        eredeti utasítás                 generált utasítások

            jc stop                               jnc skip
                                                  jmp stop
                                         skip:
```

Mint ismeretes az eljárások, függvények hívása a **CALL**, mindig feltétel nélküli vezérlésátadást valósít meg, közeli vagy távoli ugrás végrehajtásával. A hatótávolság ebben az esetben a hívott rutin elhelyezkedésétől ill. típusától (near/far) függ.

Feltétel nélküli ugrások esetén a fentiekben ismertetett automatizmus a **NEAR PTR** ill. a **FAR PTR** operátorok megadásával kikapcsolható:

```
jmp NEAR PTR stop          { közeli ugrás generálódik }
jmp FAR PTR stop           { távoli ugrás generálódik }
```

4.1.2.4. Assembler direktívák

A Turbo Pascal assembler fordítója mindössze három assembler direktíva használatát támogatja: **DB** (define byte), **DW** (define word) és **DD** (define double word). Ezek segítségével közvetlenül a kódba (kód-szegmensbe) byte-os, szavas ill. dupla szavas adatokat helyezhetünk el.

```
asm
  db  0ffh,'A'
  db  'hello!',13,10
  db  12,"Turbo Pascal"
  dw  1991
  dw  myvar               { a myvar  Pascal változó offszet címe }
  dw  myproc              { a myproc Pascal rutin offszet címe }
  dd  myvar               { pointer a myvar  Pascal változóra }
  dd  myproc              { pointer a myproc Pascal rutinra }
end;
```

Szükséges megjegyeznünk, hagy a define direktívákkal nem lehet assembler szinten változókat definiálni, mint ahogy megszoktuk pl. a Turbo Assemblernél:

```
ByteVar    db   ?
WordVar    db   ?
           ...
           mov  al,ByteVar
           mov  bx,WordVar
```

Turbo Pascalban a lehetséges megoldás:

```
var
ByteVar : byte;
WordVar : word;
```

```
asm
   mov  al,ByteVar
   mov  bx,WordVar
end;
```

4.1.2.5. Operandusok

A beépített assemblerben az operandusok kifejezések, amelyek konstansokat, regisztereket, szimbólumokat és operátorokat tartalmaznak.

Az operandusok egyik nagy csoportját az előredefiniált jelentéssel bíró foglalt szavak alkotják:

AH	AL	AND	AX	BH	BL	BP
BX	BYTE	CH	CL	CS	CX	DH
DI	DL	DS	DWORD	DX	ES	FAR
HIGH	LOW	MOD	NEAR	NOT	OFFSET	OR
PTR	QWORD	SEG	SHL	SHR	SI	SP
SS	ST	TBYTE	TYPE	WORD	XOR	

A foglalt szavaknak elsőbbsége van a felhasználó által definiált nevekkel szemben. Ha foglalt szóval megegyező azonosítóra szeretnénk hivatkozni az **asm** utasításon belül, akkor az azonosító elé a & jelet kell helyezni. Például:

```
var
  ch : char;
    ...
asm
   mov  ch,1  {hivatkozás a CH regiszterre}
end;
```

ugyanakkor:

```
asm
   mov  &ch,1  {hivatkozás a ch változóra}
end;
```

Ajánlatos azonban az ilyen egyezőségeket kerülni.

4.1.3. Kifejezések

A beépített assembler a kifejezések kezelésére a Pascal-ban használt elveket használja, azonban sok mindenben el is tér attól. Az alábbiakban ezeket a fontos különbségeket veszük sorra.

A fordító minden kifejezésnek egy 32 bites egész értéket feleltet meg és nem támogatja a lebegőpontos és a sztring értékeket, a sztring konstansok kivételével. A kifejezések *kifejezés elemekből* és *operátorokból* állnak, és minden kifejezéshez tartozik egy *kifejezés osztály* és egy *kifejezés típus*.

A legfontosabb különbség a Pascal és az assembler kifejezések között az, hogy az assembler kifejezések fordítási időben kerülnek kiértékelésre tehát *konstans értéket* kell szolgáltatniuk. Tekintsünk néhány példát e fontos eltérés bemutatására:

```
const                      var
  X = 10;                      X,Y : integer;
  Y = 20;                        Z : integer;
var                          ...
  Z : integer;               asm
  ...                          mov   ax,X
asm                            add   ax,Y
  mov   Z,Z+Y                  mov    Z,ax
end;                         end;

                             asm
asm                            mov   ax,X
  mov   ax,X+4                 add   ax,4
end;                         end;
```

4.1.3.1. Kifejezések elemei

A kifejezések alapelemei a *konstansok (numerikus és sztring), regiszterek* és *szimbólumok.*

Numerikus konstansok:

Numerikus konstansok olyan egészek értékek melyek a -2147483648 és 4294967295 tartományon belül helyezkednek el. Az egészek alapértelmezés szerint decimálisak, de lehetőség van bináris (pl. 110011**B**), oktális (pl. 3427**O**) ill. hexadecimális (pl. 0a19ea**H** vagy **$**a19ea) számok használatára is. Felhívjuk a figyelmet arra, hogy a numerikus konstansok minden számrendszerben 0..9 közötti jeggyel kell hogy kezdődjenek.

Sztring konstansok

A sztring konstansok aposztrófok (') vagy idézőjelek (") között megadott szövegek. A **DB** direktíva használatával tetszőleges hosszú szöveg megadható. A maximálisan négy karakter hosszú sztringeknek numerikus érték feleltethető meg a karaktereket 255-ös alapú számrendszer jegyeinek tekintve. Például 'dcba' numerikus értéke 64636261H.

Regiszterek

Ha az operandus csak egy regisztert tartalmaz, úgy azt regiszter operadusnak nevezzük. Látni fogjuk, hogy bizonyos regiszterek más összefüggésben is használhatók. A felhasználható regiszterek és rövid leírásuk a következő:

16 bites általános regiszterek	AX	BX	CX	DX
az első csoport 8 bites alsó fele	AL	BL	CL	DL
az első csoport 8 bites felső fele	AH	BH	CH	DH
16 bites pointer és index regiszterek	SP	BP	SI	DI
16 bites szegmens regiszterek	CS	DS	SS	ES
8087 regiszter stack	ST			

A bázis regiszterek (BX és BP) és az index regiszterek (SI és DI) szögletes zárójelek közé zárva indexelést jelölnek. Az érvényes regiszter-kombinációk: [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI] és [BP+DI].

A szegmens regisztereket kettősponttal (:) kiegészítve a mikroprocesszor alapértelmezés szerinti szegmens használatát tudjuk átdefiniálni.

Az **ST** regiszter a 8087 lebegőpontos regiszterveremének tetejét jelöli ki. Tetszőleges regiszterre az **ST(n)** kifejezéssel hivatkozhatunk, ahol n (0..7) a stack tetejétől való távolságot jelöl.

Szibólumok

A beépített assembler lehetővé teszi majdnem minden Pascal szimbólum - címkék, konstansok, típusok, változók, eljárások és függvények - felhasználását az assembler kifejezésekben. Adott továbbá három speciális szimbólum a *@Code*, a *@Data* és a *@Result*.

A @Code és a @Data szimbólumok az aktuális kód- ill. adatszegmenst reprezentálják. Ezeket általában a **SEG** operátorral együtt használjuk:

```
asm
  mov   ax,SEG @data
  mov   ds,ax
end;
```

Ha az **asm** utasítás egy függvényen belül helyezkedik el, a @Result szimbólummal hivatkozhatunk a függvény visszatérési értékére. A @Result használatának megértéséhez nézzünk egy egyszerű függvényt kétféleképpen megvalósítva:

```
function sum(x, y : integer) : integer;
  begin
    sum := x + y;
  end;
```

```
function sum(x, y : integer) : integer;
  begin
    asm
      mov    ax,x
      add    ax,y
      mov    @result,ax
    end;
  end;
```

Az alábbiakban összefoglaljuk azokat a szimbólumokat, amelyek nem használhatók assembler kifejezésekben.

- A standard eljárások és függvények (pl. *Writeln, Chr*).
- A *Mem, MemW, MemL, Port* és *PortW* nevű speciális tömbök.
- Sztring, lebegőpontos és halmaz típusú konstansok.
- Az **Inline** direktívával létrehozott rutinok.
- Az aktuális blokkon kívül deklarált címkék.
- A *@Result* szimbólum függvényen kívül.

A következő táblázat összefoglalja a szimbólumok használatához szükséges legfontosabb tudnivalókat.

Szimbólum	Érték	Osztály	Típus
címke	a címke címe	memória	SHORT
konstans	a konstans értéke	érték	0
típus	0	memória	a típus mérete
mező	a mező offszetje	memória	a típus mérete
változó	a változó címe	memória	a típus mérete
eljárás	az eljárás címe	memória	NEAR vagy FAR
függvény	a függvény címe	memória	NEAR vagy FAR
unit	0	érték	0
@Code	a kódszegmens címe	memória	0FFF0H
@Data	az adatszegmens címe	memória	0FFF0H
@Result	az eredmény offszetje	memória	a típus mérete

A továbbiakban nézzünk néhány megjegyzést a Pascal szimbólumok használatával kapcsolatban. A lokális változók (amelyeket eljárásokban és függvényekben deklarálunk) mindig a stack-en találhatók a SS:BP-től relatíven elhelyezkedve. Az alábbi függvényben a MOV AX,CNT utasítás a MOV AX,[BP-2] hivatkozást jelenti.

```
procedure Test;
var
  cnt: integer;
...
asm
  mov   ax,cnt
end;
```

Az assembler a **var** (cím szerint átadott) paraméterket mindig 32 bites pointerként kezeli. A **var** paraméterek elérésére nézzük a következő rövid példát:

```
function Sum(var X,Y : integer) : integer;
begin
  asm
    les    bx,X    { az X változó szegmenscíme ES-be, az
                     offszetcíme BX-be töltődik }
    mov    ax,es:[bx]
    les    bx,Y
    add    ax,es:[bx]
    mov    @result,ax
  end;
end;
```

Rekordokon belüli mezők azonosítására - mint a Pascal-ban - a mezőszelektor operátor, a pont (.) szolgál.

```
type
  pont =  record
    X,Y : integer;
  end;
  tegla = record
    A,B : pont;
  end;
var
  P : pont;  T : tegla;
...
asm
  mov    ax,P.X
  mov    bx,T.A.Y
end;
```

A **record** és az **object** típusok segítségével a memória tetszőleges helyére definiálhatunk egy rekordot, vagy egy objektumot. A előző deklarációkat használva az alábbi programrészlet minden sora ugyanazt a gépi kódot generálja.

```
asm
  mov    ax,(tegla PTR es:[di]).B.X
  mov    ax,tegla(es:[di]).B.X
  mov    ax,es:tegla[di].B.X
  mov    ax,tegla[es:di].B.X
  mov    ax,es:[di].tegla.B.X
end;
```

Meg kell jegyeznünk, hogy a pont (.) operátort használhatjuk unit-on belüli hivatkozáshoz is.

4.1.3.2. Kifejezések osztályai

A beépített assembler három osztályba sorolja a kifejezéseket: *regiszterek*, *memória hivatkozások* és *közvetlen értékek*. A regiszter osztályt azok a kifejezések alkotják, amelyekben a regiszter nevek önállóan szerepelnek (pl. AX, ES,). Ha egy kifejezés memóriapozíciót definiál, úgy az a memória hivatkozás osztályba tartozik. Ilyen kifejezések a Pascal címkék, változók, típusos konstansok, eljárások és függvények. Végezetül azok a kifejezések, amelyek a fenti két osztálynak nem tagjai, alkotják a közvetlen értékek osztályát, pl. a Pascal típus nélküli konstansai és a típus azonosítói. Nézzünk egy példát az elmondottak demonstrálására:

```
const
  start = 10;
var
  cnt : integer;
...
asm
  mov    ax,start              { MOV AX,xxxx    - reg.,érték  }
  mov    bx,cnt                { MOV BX,[xxxx] - reg.,memória}
  mov    cx,[start]            { MOV CX,[xxxx] - reg.,memória}
  mov    dx,OFFSET cnt         { MOV DX,xxxx    - reg.,érték  }
end;
```

Az OFFSET operátor használatával az előző programrészlet első két sorát az alábbi alakban is felírhattuk volna:

```
asm
  mov    ax,OFFSET [start] { MOV AX,xxxx    - reg.,érték  }
  mov    bx,[OFFSET cnt]   { MOV BX,[xxxx] - reg.,memória}
end;
```

A memória hivatkozások és a közvetlen értékek osztályát a továbbiakban *relokálható kifejezések* és *abszolút kifejezések* csoportjára osztjuk. A relokálható kifejezés értéke csak a szerkesztés (link) során, míg az abszolút kifejezés értéke már a fordítás (compile) során ismert. Tipikusan a címke, a változó, az eljárás és a függvény hivatkozásokat tartalmazó kifejezések relokálhatóak, ellentétben a csak konstanst tartalmazóval, ami abszolút.

4.1.3.3. Kifejezések típusai

Minden assembler kifejezés rendelkezik egy típussal, pontosabban szólva egy mérettel, amely a definiált memória terület mérete byte-ban kifejezve. A fordító ezt a típust az operandusok típusegyezésének ellenőrzésére használja. Amennyiben az operandusok típusai nem egyezőek a "*Invalid combination of opcode and operands*" hibajelzést kapjuk. Az assembler programozás során bizonyos esetekben meg kell kerülnünk ezt az ellenőrzést. Tekintsünk egy példát, a típusok használatára:

```
var
  jel : boolean;
  cnt : word;
...
asm
  mov    al,jel                 { byte,byte - rendben}
  mov    bx,cnt                 { word,word - rendben}

  {hivatkozni szeretnénk a cnt változó alsó byte-jára:}
  mov    dl,cnt                 { byte,word - HIBA!  }

  {a helyes megoldások:}
  mov    dl,BYTE PTR cnt
  {vagy}
  mov    dl,byte(cnt)
  {vagy}
  mov    dl,cnt.byte
end;
```

Vannak esetek, pl. az egyoperandusú utasítások, amikor az assembler nem tudja a típust meghatározni (pl. INC [100H]). Ebben az esetben mindenképpen használnunk kell a típuskijelölés valamelyik formáját:

```
asm
  inc    BYTE PTR [100H]
  inc    byte([100H])
  inc    [100H].byte
end;
```

Az alábbi táblázat az assembler előredefiniált típusait és a típusokat jellemző numerikus értékeket tartalmazza.

szimbólum	típus
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFEH
FAR	0FFFFH

4.1.3.4. Kifejezésekben használható operátorok

A beépített assembler oparátorai és az operátorok precedenciája eltér a Pascal nyelvben megszokottól. A következő táblázatban röviden összefoglaljuk a lehetséges operátorokat precedenciájuk csökkenő sorrendjében. Az azonos kategóriában elhelyezkedő operátoroknak megegyezik a precedenciájuk.

Operátor	Rövid leírás
&	azonosító átdefiniálása
()	részkifejezés
[]	memória hivatkozás
.	struktúramező szelektor
HIGH	szó magas helyiértékű 8 bitje
LOW	szó alacsony helyiértékű 8 bitje
+	egyoperandusú +
-	egyoperandusú -
:	szegmens átdefiniálása
OFFSET	memóriacím offszet része
SEG	memóriacím szegmens része
TYPE	kifejezés típusa (byte méret)
PTR	típuskijelölő operátor
*	szorzás
/	egész osztás
MOD	egész osztás maradéka
SHL	logikai eltolás balra
SHR	logikai eltolás jobbra
+	bináris összeadás
-	bináris kivonás
NOT	bitenkénti negálás
AND	bitenkénti ÉS művelet
OR	bitenkénti VAGY művelet
XOR	bitenkénti kizáró VAGY művelet

4.1.4. Assembler eljárások és függvények

Láthattuk, hogy minden példánkban az **asm...end** utasítás valamely Pascal blokkban (**begin...end**) helyezkedett el. A Turbo Pascal azoban lehetővé teszi, hogy az **assembler** direktíva használatával teljes függvényeket és eljárásokat assembler nyelven írhassunk meg. Ebben az esetben az alprogramok törzse nem a **begin...end,** hanem az **asm...end** blokkban kerül definiálásra. Nézzünk erre egy rövid példát:

```
function Lmul(x, y : integer) : longint; assembler;
asm
  mov    ax,x
  imul   y
end;
```

Assembler direktíva használata esetén a Turbo Pascal a fordítás során, hatékonyabb kód generálása érdekében különbző optimalizálásokat hajt végre. A következőkben ezeket az optimalizálási lépéseket vesszük sorra, hisz ezek ismerete elengedhetetlen, működő assembler rutinok kialakításához.

■ A fordító nem generál kódot az érték szerint átadott paraméterek lokális változókba való átmásolására. Ez vonatkozik az összes sztring típusú és minden olyan érték szerint átadott paraméterre, melynek hossza nem 1, 2 vagy 4 byte. Az eljáráson ill. függvényen belül az ilyen paraméterekre úgy kell hivatkozni, mint **var** típusú paraméterre.

■ A fordító nem foglal memória helyet a függvények visszatérési értéke számára, így a @Result szimbólum használata nem megengedett. E szabály alól kivételt képeznek a sztring típusú függvények, amelyek mindig rendelkeznek a @Result pointerrel.

■ A fordító a paramétereket és lokális változókat nem tartalmazó eljárások és függvények számára nem generál területet a veremben (stack frame).

■ Az assembler eljárások és függvények esetén mindig generálódik egy belépési és egy kilépési kódrészlet az alábbiak szerint:

```
push   bp            ; van ha a locals<>0 vagy a params<>0
mov    bp,sp         ; van ha a locals<>0 vagy a params<>0
sub    sp,locals     ; van ha a locals<>0
...
; itt helyezkedik el az asm...end kódrészlet
...
mov    sp,bp         ; van ha a locals<>0
pop    bp            ; van ha a locals<>0 vagy a params<>0
ret    params        ; mindig van
```

ahol a *locals* a lokális változók tárolásához szükséges memória terület méretét tartalmazza, míg a *params* a paramétereket tartalmazó stack terület mérete.

Az assembler direktívát tartalmazó függvények a függvényértéket az alábbiak szerint kell hogy visszaadják:

- A megszámlálható típusú (pl. char, word, longint) függvények visszatérő értékét az AL (8 bites), vagy az AX (16 bites) vagy a DX:AX (32 bites) regiszterekben kell elhelyezni.

- A real típusú függvényértéket a DX:BX:AX regiszterekbe kell tölteni.

- A 8087 típusú (pl. single, double) függvények eredménye a lebegőpontos stack tetején ST(0) jelenik meg.

- Pointert a DX:AX-ben tudunk visszadani.

- Sztring típusú függvény által visszaadott karaktersorozat egy ideiglenes területen a @Result pointerrel azonosítva helyezkedik el.

Az **assembler** direktíva sok tekintetben hasonlít az **external** direktívához. Így az elmodott szabályok egy része mind a két esetben érvényesnek tekinthető.

4.2. Külső assembler kód beszerkesztése

A következő két alfejezetekben áttekintjük azokat a lehetőségeket, amelyek már a 4.0-ás verziótól kezdve használhatók Turbo Pascal és assembler rutinok összeépítésére.

Külső assembler fordítóval előállított tárgykódú (.OBJ) modul beépítését teszi lehetővé a **$L** fordító direktíva. Az assembly nyelven megírt eljárásokat és függvényeket a Pascal programban külső rutinnak (**external**) kell deklarálni. A szükséges lépések tehát a következők:

```
{$L asmmod}
...
function asmproc(ch : char) : char; external;
```

Az alábbiakban áttekintjük a megfelelő assembler rutinok kialakításához szükséges ismereteket.

■ Az assembly nyelvű forrásfile-ban az eljárásokat és a függvényeket egy olyan szegmensben kell elhelyezni, amelynek a neve **CODE, CSEG** vagy pedig xxx_**TEXT** , ahol az xxx teszőleges név lehet. Másrészt, a Pascal-ból használni kívánt assembler rutinok neveit a **PUBLIC** direktívával globálissá kell tennünk. Továbbá tisztáznunk kell a meghívás módját (near vagy far), a paraméterek számát és típusát ill. a visszatérési érték típusát.

■ Az assembly nyelvű forrásfile tartalmazhat kezdőértékkel ellátott változókat a **CONST** vagy xxx_**DATA** nevű szegmensekben, ill. kezdőérték nélküli változókat a **DATA, DSEG** vagy xxx_**BSS** nevű szegmensekben. Minden assembler változó lokális az assembler modulra nézve, tehát a Pascal programból nem érhetők el, annak ellenére hogy ezek a változók is a Pascal program adatszegmensében kerülnek tárolásra. A kezdőérték nélküli változók deklarálásához mindig a kérdőjelet (?) kell használni, mint ahogy az a példánkban is látható:

```
cnt      DW   ?
puffer   DB   128 dup(?)
```

■ A fentiektől eltérő nevű szegmensek és a **GROUP** direktíva nem kerül feldolgozásra. A szegmensek definíciójában használhatunk byte-os (**BYTE**) vagy szavas (**WORD**) kiigazítást. Tudni kell azonban, hogy összeszerkesztésnél a kódszegmensek mindig byte-, míg az adatszegmensek mindig szóhatárra igazítva kerülnek beépítésre.

■ Minden olyan Pascal függvény, eljárás és változó, amely a Pascal program azon helyéről, ahova a gépi kódú rutin beépítésre kerül elérhető, az assembler rutinból is használható. Ezen igényünket azonban közölni kell az assembler fordítóval az **EXTRN** direktívában a Pascal nevek és típusok felsorolásával.

```
var               - a változók eléréséhez szükséges assembler
  a : byte;         utasítás:
  p : pointer;
  r : real;         EXTRN A:BYTE, P:DWORD, R:FWORD
```

4.2.1. A Turbo Assembler és a Turbo Pascal

Az előző részben ismertetett tudnivalókat minden assembler fordítónál figyelembe kell venni, ha külső assembler modult kívánunk Turbo Pascal programhoz illeszteni. A Turbo Assembler azonban a hagyományos lehetőségeken kívül rendelkezik olyan eszközökkel, amelyek használatával leegyszerűsödik a programozási munka.

Létezik a **TPASCAL** modell, amely hatására a Pascal hívási konvenciók - a szegmensek neve, a PROC-ban a be- (PUSH BP / MOV BP,SP) és a kimeneti (POP BP / RET n) lépések - automatikusan generálódnak. További érdekes lehetőség, hogy a **PROC** definícióban megadhatjuk a Pascal rutin paramétersorát, ill. sztring típusú függvény esetében a **RETURNS** opcióval a visszaadott sztringet tartalmazó ideiglenes területre is hivatkozhatunk. Nézzünk egy példát az elmondottakra:

```
            .MODEL TPASCAL
            .CODE
tasmproc    PROC FAR  i:BYTE, j:BYTE RETURNS res:DWORD
            PUBLIC tasmproc
            les    DI,res        ; az ideiglenes sztring címe
            mov    AL,i          ; az első paraméter
            mov    BL,j          ; a második paraméter
              .
              .
              .
            ret
            ...
```

A megfelelő Pascal deklaráció:

```
function tasmproc(i, j : char) : string; external;
```

4.3. Gépi kód beépítése a Pascal programba (inline)

Ez a lehetőség csak néhány utasításból álló gépi kódú rutinok beépítését támogatja, kikerülve ezzel a külső assembler fordító ill. az .OBJ modul használatát. Az **Inline** kódolás nagyon nehézkes, hisz a szükséges gépi kódokat "kézzel" kell előállítanunk, nem is beszélve az esetleges változtatásoknál felmerülő nehézségekről. A Turbo Pascal-ban létezik **inline** utasítás és **inline** direktíva.

4.3.1. Az Inline utasítás

Az inline utasítás az inline szót zárójelben követő osztás (/) jelekkel tagolt elemekből épül fel. Minden elem opcionálisan tartalmaz méretkijelölést (< vagy >), ezt követően konstanst vagy változó azonosítót és végezetül szintén opcionálisan a + vagy - jeleket követő konstanst (offszet specifikátort).

Pl.: `inline(>24/<$1991/Cnt+1/adat-offszet);`

A < és a > jelekkel az aktuális adathosszt lehet megváltoztatni. A > jel hatására egy 8 bites érték 16 biten kerül tárolásra, míg a < jellel egy szavas értéknek az alsó 8 bitjét használhatjuk. Ha az utasításban globális változót használunk, akkor a változó adatszegmensbeli offszetje kerül beépítésre, ellentétben a lokális változókkal, ahol az offszet a BP-hez relatív.

Az **inline** utasítás teljes mértékben helyettesíthető a Turbo Pascal 6.0-ás beépített assemblerének használatával, az **asm...end** utasítással. Az **inline** utasítás felhasználására ill. lehetséges helyettesítésére nézzük meg a következő egyszerű eljárást. Az eljárás feladata egy Memptr címmel azonosított memória terület feltöltése Cnt darab Wdata értékkel.

```
{megoldás az inline utasítás használatával}

procedure FWinline(var Memptr; Cnt: word; Wdata: word);
begin
  inline(
    $C4/$7E/<Memptr/     { LES   DI,Memptr[BP] }
    $8B/$4E/<Cnt/        { MOV   CX,Cnt[BP]    }
    $8B/$46/<Wdata/      { MOV   AX,Wdata[BP] }
    $FC/                 { CLD                }
    $F3/$AB);            { REP   STOSW        }
end;
```

```
{megoldás az asm...end utasítás használatával}

procedure FWasm(var Memptr; Cnt: word; Wdata: word);
begin
  asm
    les   di,Memptr
    mov   cx,Cnt
    mov   ax,Wdata
    cld
    rep   stosw
  end;
end;
```

4.3.2. Az Inline direktíva

Az **inline** direktíva használatával olyan függvényeket és eljárásokat írhatunk, amelyekre való hivatkozáskor a függvény törzse beépül a hívás helyére. Ebből a makró-szerű működésből következően az inline direktíva használata a formai hasonlóság ellenére eltér az

inline utasítás használatától. A legfontosabb eltérés az, hogy az inline direktíva nem igazi függvényeket és eljárásokat definiál, hanem a stack-en keresztül paraméterezhető gépi kódú utasítások sorozatát. Ebből következik az is, hogy a paraméterekre nem lehet szimbólikusan hivatkozni, hanem mindig közvetlenül a veremről kell (POP) azokat feldolgozni. Ajánlott felhasználása az **inline** direktívának - processzor utasítások közvetlen beépítése a Pascal programba, mint ahogy ez az alábbi példákból is kitűnik:

```
procedure CLI;     {a hardver megszakítások tiltása}
      inline($FA);

procedure STI;     {a hardver megszakítások engedélyezése}
      inline($FB);

{példa a paraméterek feldolgozására }
procedure lmul(x, y : integer) : integer;
      inline(
             $5A/            { POP  AX }
             $58/            { POP  DX }
             $F7/$EA);       { IMUL DX }
             );
```

4.4. A fejezet összefoglalása egy példa bemutatásával

Mint láttuk, a Turbo Pascal 6.0-ban a lehetőségek teljes tárháza áll rendelkezésünkre ahhoz, hogy a Pascal határait átlépve minden szintű feladatot meg tudjunk oldani. Ebben a fejezetben egy egyszerű függvény assembly nyelven történő megírásával össze tudjuk hasonlítani az egyes megoldási módokat. Felmerül azonban a kérdés, hogy mikor melyik megoldást válasszuk.

Hogy tudjunk dönteni, vessük össze először a *szamlal1* és a *szamlal5* függvényeket. Azonnal szembetűnő a hasonlatosság, hisz az **asm...end** utasítás teljes mértékben képes helyettesíteni az **inline** utasítást. Így a választás természetesen a beépített assemblerre esik. Ha netalán régebbi Turbo Pascal verzió alá kell egy rutint áttenni, akkor is nagy segítséget jelent az **asm...end** utasítás, amely által generált gépikód pl. a Turbo Debugger segítségével kiolvasható.

Minden függvény a 'szoveg' nevű sztringben a 'kar' karakter előfordulásainak számát adja vissza.

```
function szamlal1(szoveg : string; kar : char) : byte;
begin
  asm
    xor cx,cx
    mov dl,cl
    lea di,szoveg        { a szöveg stack-en helyezkedik el! }
    inc di               { a di a szoveg[1]-re mutat }
    mov cl,ss:[di-1]     { a sztring hossza a szoveg[0]-ból }
    mov al,kar           { a számlálandó karakter betöltése al-be}
    cld
@tovabb:                 { a kereső ciklus }
    segss repne scasb
    jnz  @nincs
    inc  dl
@nincs:
    jcxz @kesz
    jmp  @tovabb
@kesz:
    mov @result,dl       { a visszatérési érték beállítása }
  end;
end;
```

```
function szamlal5(szoveg : string; kar : char) : byte;
begin
inline(
      $31/$C9/                      {xor    cx,cx           }
      $88/$CA/                      {mov    dl,cl           }
      $8D/$BE/szoveg/               {lea    di,szoveg[bp]   }
      $47/                          {inc    di              }
      $36/$8A/$4D/$FF/              {mov    cl,ss:[di-1]    }
      $8A/$46/<kar/                 {mov    al,kar[bp]      }
      $FC/                          {cld                    }
      $36/$F2/$AE/                  {segss repne scasb      }
      $75/$02/                      {jnz    $+2             }
      $FE/$C2/                      {inc    dl              }
      $E3/$02/                      {jcxz   $+2             }
      $EB/$F5/                      {jmp    $-11            }
      $88/$56/$FF                   {mov    [bp-1],dl       }
      );
end;
```

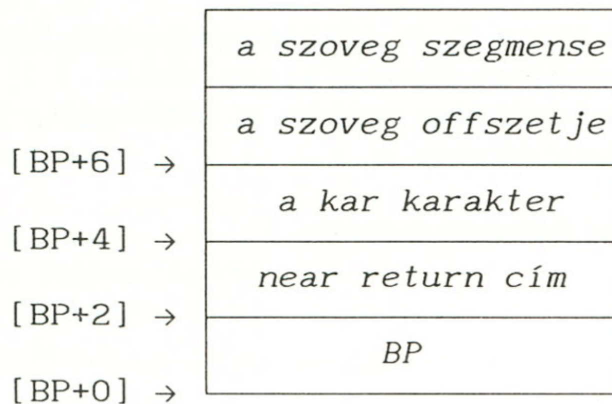
A további három függvény közös jellemzője, hogy mindegyik egy-egy assembler rutint tartalmaz. Ha összevetjük a *szamlal2* (Turbo Pascal 6.0) és a *szamlal3* (Turbo Assembler) rutinok törzsét, láthatjuk hogy csak a címkékben különböznek egymástól. Ha a Turbo Pascal nyomkövetési lehetőségeit is figyelembe vesszük, természetesen a *szamlal2* megoldást választjuk. Az azonosságok miatt itt is adódik a lehetőség, hogy Turbo Pascal 6.0-ban kifejlesztett assembler rutinokat régebbi verziók számára is felhasználhatóvá tegyünk.

```
function szamlal2(szoveg : string; kar : char) : byte; assembler;

asm
    xor cx,cx
    mov dl,cl
    les di,szoveg     {a szoveg az adatszegmensben helyezkedik el!}
    inc di            { a di a szoveg[1]-re mutat }
    mov cl,es:[di-1]  { a sztring hossza a szoveg[0]-ból }
    mov al,kar
    cld
@tovabb:              { a kereső ciklus }
    repne scasb
    jnz  @nincs
    inc  dl
@nincs:
    jcxz @kesz
    jmp  @tovabb
@kesz:
    mov al,dl         { a visszatérési érték beállítása }
end;



{$L tasmproc.obj}

function szamlal3(szoveg : string; kar : char) : byte; external;



{$L masmproc.obj}

function szamlal4(szoveg : string; kar : char) : byte; external;
```

Utolsó lépésként, ha megnézzük a *szamlal4* (MASM) függvényt, láthatjuk hogy ez lényegében megegyezik a *szamlal3* (TASM) rutinnal. Annyi az eltérés, hogy a Turbo Assembler szolgáltatásai helyett saját kezűleg kellett mindent definiálni a Turbo Pascal igényeinek megfelelően.

Ahhoz hogy az MASM rutinban is helyesen tudjunk hivatkozni a stack-en található paraméterekre, érdemes a verem felépítését felrajzolni.

A stack felépítése a *szamlal4* rutinba való belépéskor:



A *tasmproc.asm* file tartalma:

```
    .model tpascal
    .code
szamlal3 proc near szoveg:dword, kar:byte
     public szamlal3

     xor cx,cx
     mov dl,cl
     les di,szoveg
     inc di
     mov cl,es:[di-1]
     mov al,kar
     cld
tovabb:
     repne scasb
     jnz  nincs
     inc  dl
nincs:
     jcxz kesz
     jmp  tovabb
kesz:
     mov al,dl

     ret
szamlal3 endp
    end
```

Az *masmproc.asm* **file tartalma:**

```
code segment public byte        ; byte határon kezdődő kód
     assume cs:code

szamlal4 proc near
     public szamlal4

     push bp                    ; a bp mentése és beállítása a
     mov  bp,sp                 ; paraméterek eléréséhez

     xor cx,cx
     mov dl,cl
     les di,[bp+6]              ; a szoveg paraméter feldolgozása
     inc di
     mov cl,es:[di-1]
     mov al,[bp+4]              ; a kar paraméter felolvasása
     cld
tovabb:
     repne scasb
     jnz  nincs
     inc  dl
nincs:
     jcxz kesz
     jmp  tovabb
kesz:
     mov al,dl                  ; a visszatérési érték az al-be

     pop bp                     ; a bp visszaállítása
     ret 6                      ; visszatérés a 6 byte-os para-
                                ; méter terület felszabadításával
szamlal4 endp

code ends                       ; a kódszegmens lezárása
     end
```

5. TURBO VISION ALAPISMERETEK

Az objektum-orientált programozást könnyíti meg a Turbo Vision könyvtár, melynek felhasználásával készült a Turbo Pascal új integrált környezete is. A Turbo Vision tulajdonképpen egy objektum-orientált keretrendszer, amely különösen segíti az ablakozó technikával működő programokat.

A Turbo Vision objektum-orientált könyvtár az alábbi lehetőségeket tartalmazza:

- többszörös, változtatható méretű, átfedéses ablakozás,
- görgethető (pull-down) menü,
- egérkezelés,
- dialógus dobozok,
- beépített színválasztás,
- lapozható menü, input doboz, ellenőrző dobozok, állító gombok (radio buttons) kezelése,
- a billentyűleütések és az egér klikkentésének szabványos kezelése, stb.

Tudni kell azt, hogy a Turbo Vision felhasználásával készített alkalmazások kiterjedten használják az objektum-orientált technikát, az öröklődést és a sokalakúságot (lásd a virtuális metódusokat). Nagyon jellemző a pointerek és a dinamikus változók használata, majdnem mindegyik Turbo Vision objektum példány dinamikusan foglal helyet a heap-ben. Célszerű a *new* függvény kiterjesztett változatának alkalmazása, amely a konstruktort, mint paramétert használja.

A Turbo Vision eseményvezérelt keretrendszer, mellyel könnyen készíthetünk ablakozó technikát alkalmazó programokat. Ezt a keretrendszert kell felruházni a célnak megfelelően, felhasználva a Turbo Pascal objektum-orientált programozás lehetőségeit. A Turbo Vision-ban találhatók ún. alkalmazói objektumok, a *TApplication*-ok, melyekből létrehozhatunk a saját alkalmazásunknak megfelelő leszármazottakat. Példaként hozzunk létre egy leszármazottat és nevezzük célszerűen *MyApplication*-nak. Természetesen a *MyApplication*-höz hozzá kell adni még a szükséges kiegészítéseket, amelyek az alkalmazás-specifikus feladat végrehajtásához elengedhetetlenek.

A legmagasabb szinten az alkalmazói program nagyon egyszerű:

```
begin
     MyApplication.Init;       { Inicializálja az alkalmazást }
     MyApplication.Run;        { futtatja                     }
     MyApplication.Done;       { azután megszünteti           }
end;
```

A felhasználók nagy része programjaihoz könyvtárat készít, amelyben a számára szükséges eljárásokat és függvényeket tárolja. Ha módosítani kell egy programon, akkor a megfelelő eljárás forráskódjában kell a változtatásokat elvégezni, majd az új verziót újra kell könyvtározni. A Turbo Vision használatánál soha sem kell módosítani az aktuális forráskódot. Változtatni a Turbo Vision-nak a kiterjesztésével, - azaz a *TApplication* leszármazottjainak (mint amilyen pl. a *MyApplication*) újabb tulajdonságokkal való felruházásával - lehet. A *TAppliaction* keretrendszer változatlanul marad az *APP.TPU*-n belül. A módosítás úgy történik, hogy új objektum típusokat hozunk létre, és a szükséges változtatásokat úgy végezzük el, hogy új metódusokkal írjuk felül az örökölt metódusokat (lásd a sokalakúságot).

A Turbo Vision fejlesztési filozófia alapja:

- objektum-orientált technikát kell használni,

- elfogadni a teljes Turbo Vision-t.

A felhasználói felületek létrehozásával kapcsolatban nagymennyiségű, programról programra ismétlődő, szükségtelen munkát takaríthatunk meg a Turbo Vision alkalmazásával.

5.1. Turbo Vision keretrendszer elemei

Mielőtt rátérnénk a Turbo Vision keretrendszer ismertetésére, először tekintsünk bele "vajon mi van a dobozban" - melyek a Turbo Vision eszközei.

Három fontos eszköze van:

- a látvány (*view*),
- az esemény (*event*), és
- a nem látható, ún. néma objektumok (*mute objects*).

5.1.1. Látvány (*view*)

Bármely objektum jellegű programelemet látványnak nevezünk, amely látható a képernyőn. A mezők, mezők fejléce, az ablak kerete, görgető sor, menü és a dialógus dobozok megjelennek a képernyőn, láthatóak, ezeket összefoglalva csoportnak nevezzük. Ezek együttműködnek és mind a *view* (látvány) alapobjektumra épülnek.

Egy *view* általában téglalap alakú, egy karaktert, vagy egy sort tartalmaz és egy karakter magas, vagy egy karakter széles.

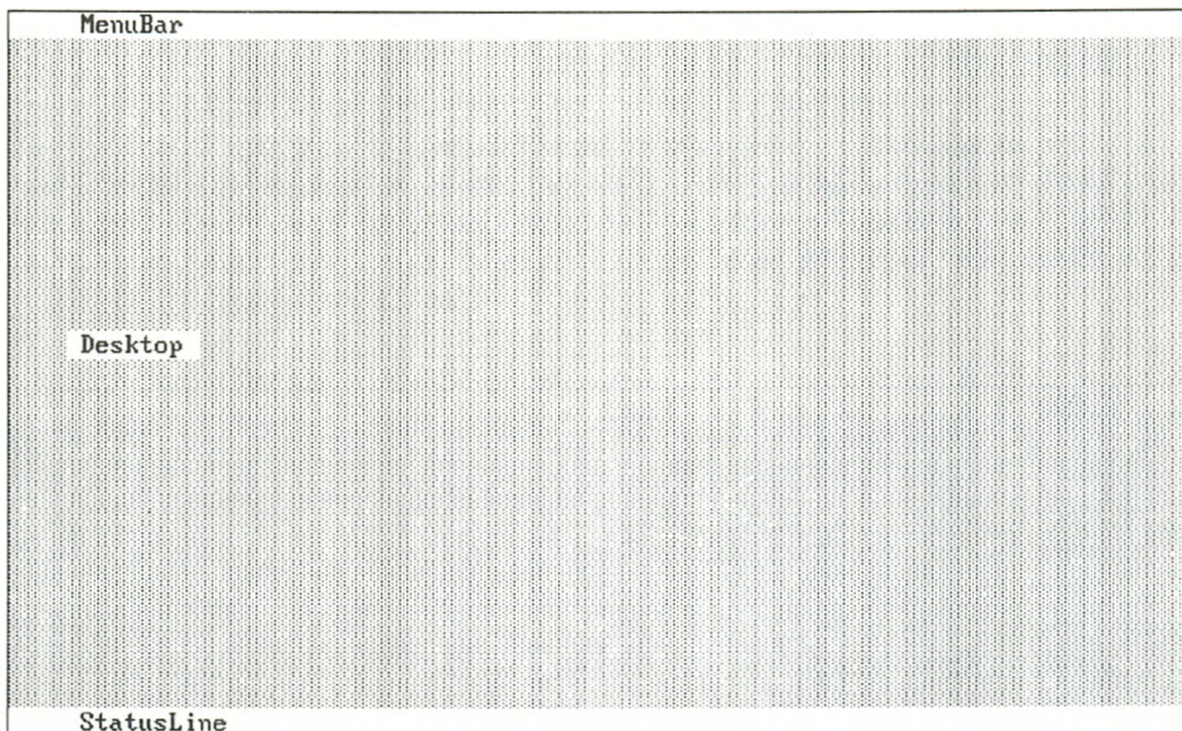
5.1.2. Esemény (*event*)

Az *event* olyan fajta esemény, amelyre az alkalmazói programnak reagálni kell. Eseményeket kelt a klaviatúra, az egér vagy a Turbo Vision más részei. Esemény tehát egy billentyű leütése, vagy az egér klikkentése, egy ablak áthelyezése, mozgatása. Az események sorban állnak a működés következtében, amelyeket az esemény végrehajtó sorrendben kezel le. A *TApplication* objektum, amelyre az alkalmazói program ráépül, tartalmazza az esemény végrehajtót.

5.1.3. Néma objektumok

A program összes egyéb objektuma néma objektum (*mute object*). Azért nevezzük "némá"-nak, mert működésük a képernyőn nem látható. A feladatuk közé tartoznak számítások végrehajtása, kommunikáció a perifériákkal, és egyéb, az alkalmazói programra nézve specifikus feladat. Ha a néma objektumnak szüksége van arra, hogy megjelenítsen valamit a képernyőn, ezt csak egy *view* objektum segítségével teheti meg. Ezt a koncepciót nagyon fontos betartani a Turbo Vision alkalmazásában.

A 5.1. ábra mutatja a közös objektumok gyűjteményét. Ezek egy Turbo Vision alkalmazás részeként jelennek meg. A *desktop* (íróasztal, azaz a háttér) is egy objektum, a *MenuBar* a képernyő tetején és a *StatusLine* a képernyő alján szintén objektumok. A szavak a *MenuBar*-ban a főmenü sora, menüket jelentenek. Ezek lapozható menük, melyek kiválaszthatók az egérrel a szóra mutatva és klikkentve, vagy megnyomva a hozzátartozó ún. a *hot key*-t. A szöveg, amely a státuszsorban jelenik meg, az alkalmazói program aktuális állapotára vonatkozó képernyő üzenet, amely megmutatja pl. a hozzáférhető *hot key-t*.



5.1. ábra
Közös objektumok gyűjteménye

Az a menüpont, amelyet a kurzor-billentyűkkel, majd az *ENTER* billentyű megnyomásával, vagy egérrel, annak baloldali gombjával klikkentve, választottunk ki, fényesebben világít. Ekkor a vezérlést a menüponthoz rendelt alkalmazói programrész kapja meg.

A Turbo Vision-nél kialakítható felhasználói felületek alapkonfigurációját részletesen ismertettük a Turbo Pascal 6.0 integrált fejlesztői környezetének leírásánál a 2. fejezetben. A következőkben azt tekintjük át, hogy mi magunk hogyan készíthetünk hasonló felhasználói felülettel rendelkező alkalmazói programokat.

5.2. Programozás Turbo Vision felhasználásával

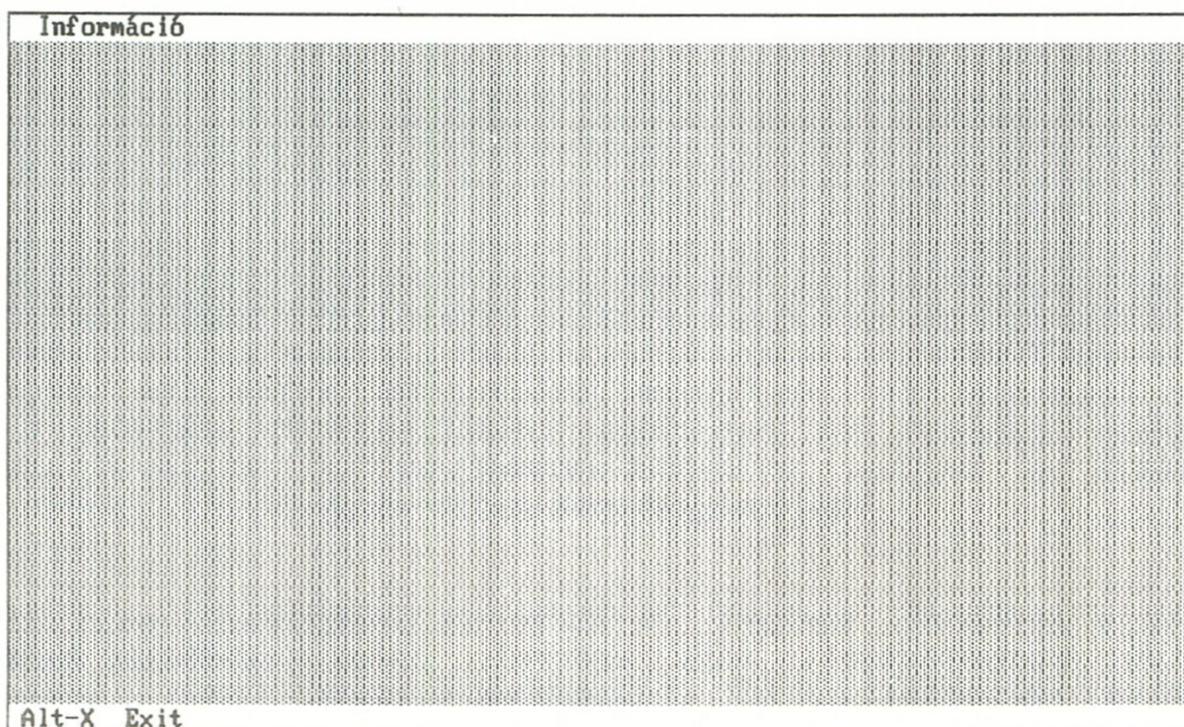
Írjuk programot, amely menüvezérelve információt ad egy Turbo Vision alkalmazás jellemzőiről.

A program főbb lépései:

- törölni kell a *desktop*-ot, féltónusú mintával.
- meg kell jeleníteni a menü sort a képernyő tetején, a státusz-sort a képernyő alján,
- kapcsolatot kell létesíteni a billentyű leütések és az egér keltette események értékelésére,
- létre kell hozni egy menü objektumot, mely kapcsolatot tart a menüpontokkal,

- dialógus dobozokat kell kapcsolni a menühöz,
- várni kell a klaviatúrán való billentyű leütésére, vagy egér műveletre.

A Tv_demo.pas program bemutatja, hogyan kell a menüt megjeleníteni a képernyőn. Az almenü csak a menüpont kiválasztásakor jelenik meg. A Tv_demo program futtatásakor a képernyő törlése után a következő *desktop* jelenik meg (5.2. ábra).



5.2 ábra
A Tv_demo.pas program menüje

Látható, hogy a képernyő tetején a menüben az "I" az Információ szóban más színnel jelenik meg, mint a "*formáció*", a képernyő alján a státusz sorban *ALT-X* Exit jelenti a kilépés módját.

Itt érdemes rámutatnunk arra a két programozási szabályra, amely bármely felhasználói környezetre érvényes:

- Ne legyen a felhasználó tanácstalan, hogy vajon mi a következő lépés a program működésében.

- Mindig meg kell adni a lehetőséget a felhasználónak, hogy előre és vissza is tudjon lépni.

A felhasználónak két lehetősége van a program futtatásakor, vagy kiválasztja az I*nformáció* menüpontot, vagy *ALT-X* leütésével kilép a programból.

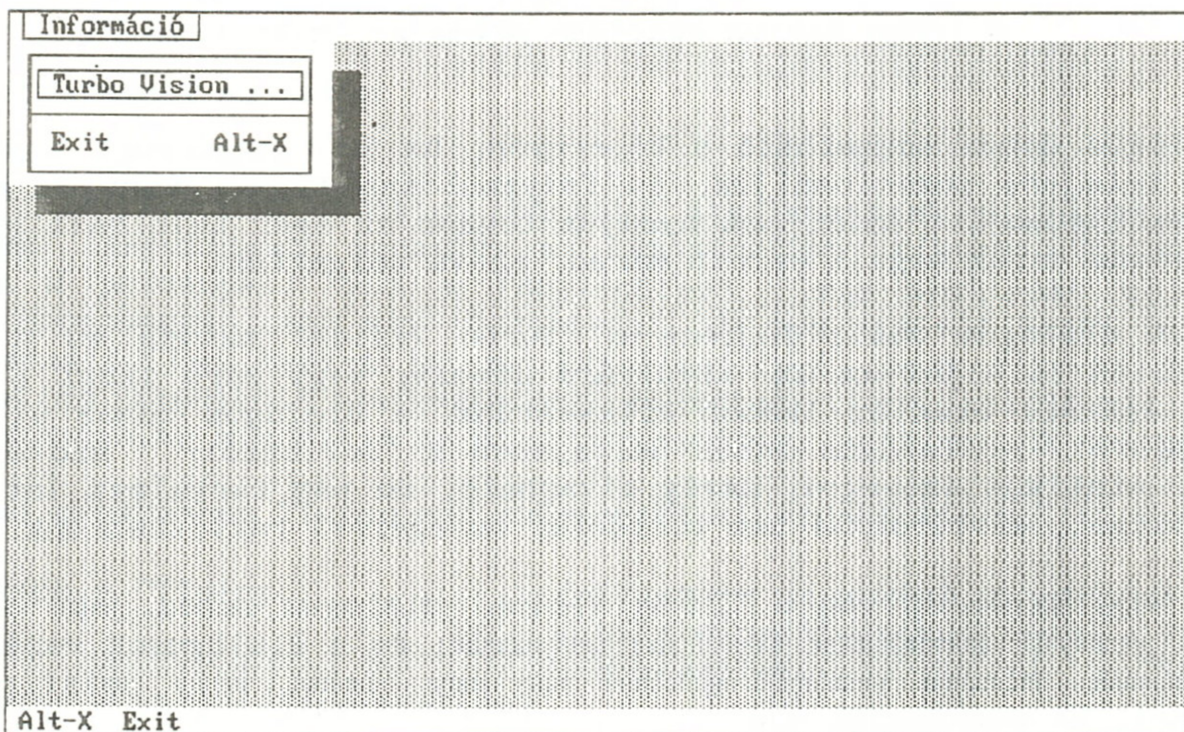
Az *Információ* menüpont kiválasztása háromféle módon történhet:

- Az egérkurzort az *Információ* menüpontra mozgatjuk és klikkentünk az egér baloldali gombjával,

- *F10* funkció-billentyűvel kiválasztjuk a felső menüsort, ahol a *Információ* menüpont nagyobb fényerővel fog világítani. Az *ENTER* billentyű leütésével kiválasztjuk az *Információ* almenüt.

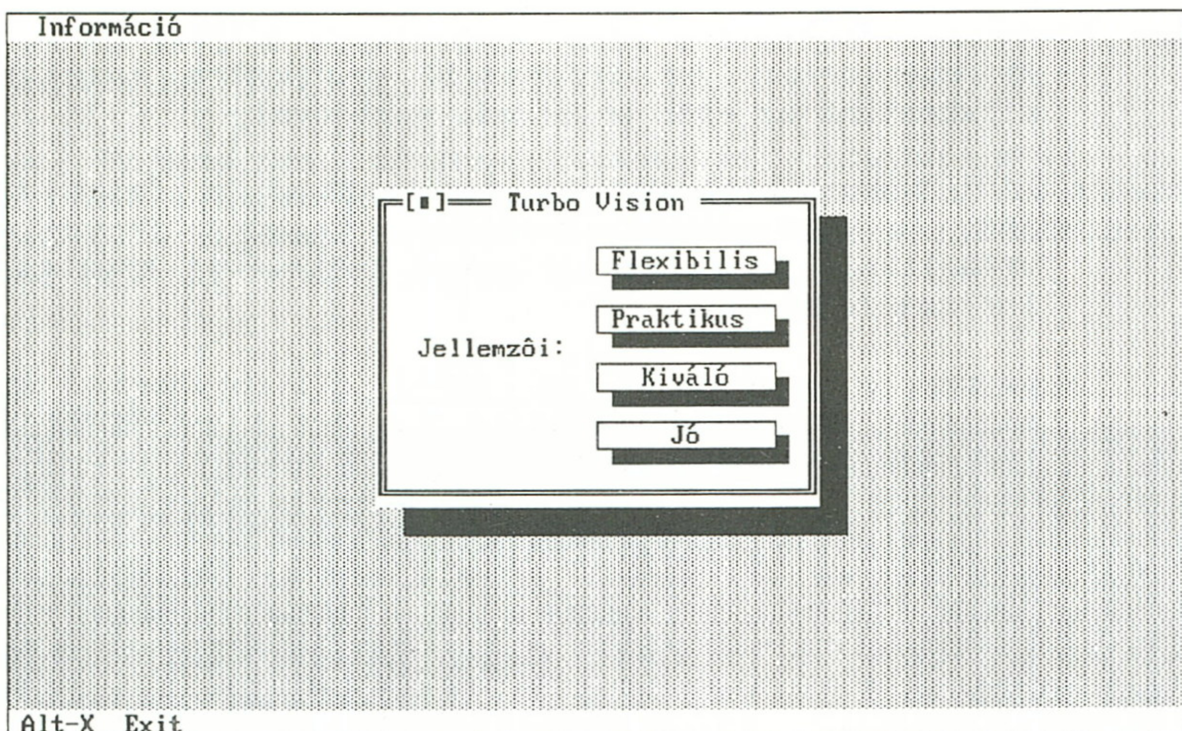
- egyszerre megnyomjuk az *ALT-I* billentyűket, ugyanis a magasabb fényerővel világító I a *hot key*-t mutatja meg az *Információ* menüpontban.

Mindhárom esetben egy almenü jelenik meg az *Információ* menüpont alatt, teljesen hasonlóan működik a Turbo Pascal integrált környezetéhez, hiszen az is Turbo Vision felhasználásával készült. A megjelent menüt a 5.3. ábra mutatja. Összesen két opció található benne. A menüt egy vonal választja két részre.



5.3. ábra
Az Információ menüpont kiválasztása

Ha a Turbo Vision menüpont került kiválasztásra, akkor egy dialógus doboz jelenik meg a képernyő közepén (lásd a 5.4 ábrát).



5.4. ábra
A Turbo Vision dialógus doboz megjelenítése

A dialógus doboz mozgatható a képernyőn, ha az egérkurzort a dialógus doboz tetejére pozícionáljuk, és folyamatosan nyomva tartjuk az egér baloldali gombját. Amint nem nyomjuk tovább az egér baloldali gombját, a dialógus doboz megáll és ott marad, ahová mozgattuk.

A dialógus doboz fejléce a "Turbo Vision" szöveg és a szöveg baloldalán találjuk az ún. ablakzáró dobozt, vagy záró ikont ([■]). Ha az egérrel ezen az ikonon klikkentünk, a dialógus doboz eltűnik a képernyőről. A dialógus doboz belsejében a "Jellemzői: " szöveg példa az ún. statikus szövegre, amely olvasható, de nem tartalmaz interaktív lehetőséget, mivel semmi sem történik, ha egérrel ráklikkentünk.

Négy téglalap van a "Turbo Vision" dialógus doboz jobb oldalán. Ezeket vezérlő gomboknak (button) nevezzük; és a dialógusok vezérlésére szolgálnak. Minden vezérlő gombon van egy címke, mely jelzi, mi fog történni, ha megnyomjuk őket.

A vezérlő gomb megnyomása az egérrel való klikkentéssel, vagy a tabulátorral (*TAB*) kiválasztva az *ENTER* billentyű leütésével történik. Figyeljük meg, mi történik akkor, ha az egér kurzorral ráállunk az egyik vezérlő gombra és megnyomjuk az egér baloldali gombját. A vezérlő gomb téglalap alakú teste egy pozícióval jobbra megnövekszik és így az árnyéka eltűnik: azt az illúziót kelti, mintha megnyomtuk volna a vezérlő gombot. Amikor elengedjük az egér baloldali gombját, a kiválasztott akció megtörténik.

Megjegyezzük, hogy az aktuálisan használható ("élő") vezérlő gomb más színnel jelenik meg, mint a többi. A gomb színe jelzi, hogy az "élő" vezérlő gomb éppen az alapértelmezés szerinti gomb-e a dialógus dobozban (ami azt jelenti, hogy az *ENTER* megnyomásakor a hozzárendelt funkció fog aktiválódni). Az alapértelemzés szerinti vezérlést a dialógus dobozban a *TAB* billentyű megnyomásával lehet változtatni. A színváltozás jelzi, hogy hová pozícionáltuk a *TAB*-bal. Ciklusban végig járhatunk a vezérlő gombokon, és az fog aktiválódni, amelynél az *ENTER* billentyűt megnyomjuk.

Ha kilépünk a dialógus dobozból, az üres *desktop* jelenik meg a főmenüvel. Lehetőségünk van újra belépni a dialógus dobozba, vagy kilépni a programból az *ALT-X* megnyomásával. Mindig úgy kell a programot megszerkeszteni, hogy a felhasználó könnyen ki tudjon lépni a programból.

Természtesen csak akkor tudjuk jól alkalmazni a Turbo Vision könyvtárat, ha először megértettük az objektumok definícióit, csak azután foglalkozzunk a metódusok implementációival. Nagyon fontos megérteni, hogy egy objektum mit tartalmaz, és milyen a kapcsolata a többi rendszerbeli objektummal.

5.2.1. Alkalmazói objektum

A *TApplication* objektum bármely alkalmazásnak a legfontosabb objektuma. Aktuálisan sohasem a *TApplication* típusú objektumnak képezzük a példányát, mert a *TApplication* - egy absztrakt objektum típus - éppenséggel csak egy "csontváz", de nem egy élő test, így nem csinál semmit. Úgy használjuk az *TApplication*-t, hogy létrehozzuk az *TApplication* típusból származtatott objektum típus példányait. Így tulajdonképpen az működik a saját programunkban, amit magunk írtunk.

A Tv_demo programban TTvApp egy származtatott objektumtípus:

```
PTvApp = ^TTvApp;              { Célszerű az objektumtípusra }
                               { mutató pointertípust is     }
                               { deklarálni.                 }
TTvApp = object(TApplication)
     procedure InformationBox;
     procedure HandleEvent(var Event: TEvent); virtual;
     procedure InitMenuBar; virtual;
     procedure InitStatusLine; virtual;
end;
```

Érdemes minden objektumtípusra pointert definiálni. A TTvApp 4 metódust tartalmaz. A belőle leszármaztatott objektumtípus mindent, ami nincs

újradefiniálva benne, változatlanul örököl az őstől.

- Fontos, hogy definiálva legyen egy *HandleEvent* metódus. A *HandleEvent* metódus a *TApplication* metódusa, amely azokkal az általános eseményekkel foglalkozik, melyek bármely alkalmazói programban előfordulhatnak. Természetesen a felhasználónak a saját alkalmazásában előforduló eseményeket kell specifikálni, hogy azokat is kezelni lehessen.

- Az *InitMenuBar* a menüpont mögötti menüt kezeli az alkalmazásban. A *TApplication* szintén tartalmaz menükezelőt, ennek ellenére a felhasználónak definiálni kell egy metódust, amely elkészíti a menüket.

- Az *InitStatusLine* metódus feladata, hogy a képernyő alján lévő státuszsorban a megfelelő szöveget kiírja. Ez a szöveg tipikusan az alkalmazói program aktuális állapotára vonatkozó információ, amely pl. a hozzáférhető *hot-key* lehetőséget mutatja meg.

- A *InformationBox* metódus jeleníti meg a dialógus dobozt az *Információ* menü Turbo Vision almenüjének kiválasztása hatására. A menüpont kiválasztásakor a *HandleEvent* metódus aktivizálja az *InformationBox*-ot. A főmenü almenüpontjaihoz érdemes önálló metódusokat fejleszteni.

A *TTvApp* metódusai elvégzik mindazt, amit a főprogram objektumainak nyújtani kell: működtetni az alkalmazói keretet, kezelni az eseményeket, a részesemények válaszaként aktiválni a megfelelő metódusokat. Ezeket a metódusokat kell hozzáadnunk a *TApplication* objektumhoz, hogy létrehozhassuk a *TApplication* objektum származtatott objektumait.

5.2.2. A dialógus doboz objektum

A Tv_demo program egyik legjelentősebb objektuma a dialógus objektum. Mivel a dialógus objektumnak semmi speciálisat sem kell végrehajtania, így a program a *TDialog* objektumtípus egy példányát használja fel. A *TDialog* típus nem tartalmaz interaktív elemeket, de ad egy "sokat tudó" keretet, amellyel bármilyen mezővel, vagy vezérléssel interaktivitást biztosíthatunk a felhasználónak.

A *TTvApp.InformationBox* a *TDialog*-ra épül. Ennek segítségével jeleníti meg a 4 felcímkézett vezérlőgombot, amelyek *view* típusúak, mivel megjelennek a képernyőn. Ez egy tipikus felhasználása a dialógus doboznak, de gyakran vezérlés céljára is alkalmazzák. Természetesen a dialógus dobozok megszólalása is esemény, így működtetésükhöz szükséges az ún. esemény vezérlő, amely szintén megtalálható a *TDialog*-ban.

5.2.3. A Tv_demo főprogram

A legmagasabb szinten, minden Turbo Vision felhasználásával készült főprogram meglehetősen hasonlít a Tv_demo főprogramjához:

```
var
    TvInf: TTvApp;
begin
    TvInf.Init;
    TvInf.Run;
    TvInf.Done;
end;
```

5.2.3.1. Az Init metódus

A három utasítás közül az első a *TvInf.Init,* egy igen fontos konstruktor hívást jelent. Mivel az összes objektum virtuális metódusokat tartalmaz, létre kell őket hozni, mielőtt az objektum más metódusait aktiválnánk. Az összes Turbo Vision konstruktort *Init*-nek nevezzük. Ez egy konvenció, amelyet mi is követtünk a mintaprogramban.

A *TvInf.Init* létrehozza a főprogram objektumait. Törli a képernyőt, kezdeti értéket ad bizonyos fontos változóknak, megjeleníti a *desktop*-ot a menüsort és a státuszsort. Hívja a többi egyéb objektum a konstruktorait, melyek közvetlenül nem látszanak.

Ha a Tv_demo programot nyomkövetéssel futtatjuk, és az *F8* funkció billentyűvel lépkedünk, és a *TvInf.Init* utasításon állunk, *ALT-F5*-tel képernyőt váltva látjuk, hogy üres a képernyő. Túl lépve az utasításon az *F8*-cal, újra *ALT-F5*-tel képernyőt váltunk, látjuk, hogy a *desktop*, a menüsor és a státuszsor megjelent és használható a főprogramban. A főprogram objektumát a konstruktora meglehetősen gyorsan felépítette.

5.2.3.2. A *Run* metódus

Meglehetősen misztikus a Turbo Vision alkalmazásban a főprogram *Run* metódusa, hiszen a *TTvApp* definícióban nem találunk *Run* metódust. Valóban nincs is, a *Run* metódus öröklődik a *TTvApp* szülőjétől, a *TApplication* őstípus objektumtól.

A *Run* a program futtatásából valószínüleg a legtöbb időt veszi igénybe. Képzeljünk el egy **repeat ... until** ciklust, amely pszeudó

utasításokkal leírva az alábbi lehet:

```
repeat
     kap egy eseményt;
     kezel egy eseményt;
until kilép;
```

A Turbo Vision alkalmazás egy ciklus utasításhoz hasonlít: Kap egy valamilyen eseményt (ami alapján eldől, hogy mit kell tenni), kiszolgálja az eseményt, és ezt ismétli mindaddig, amíg eseményként kilépő (*Quit*) parancsot nem kap (amely a ciklust leállítja).

5.3.3.3. A *Done* metódus

A *Done* metódus valóban nagyon egyszerű. Felszabadítja a felhasznált objektumokat. Ha volt valami speciális helyfoglalás, akkor a felhasználónak kell a *Done* metódussal felszabadítani azt a területet a heap-ből, amit az *Init* konstruktor lefoglalt, és csak aután kell hívni a *TApplication.Done* metódust, amely a standard elemeket kezeli. Ha felülírjuk a *TApplication.Init* metódust, akkor valószínüleg felül kell írni a *TApplication.Done* metódust is.

Ebben a fejezetben éppen csak ízelítőt adtunk, hogyan is lehet használni a Turbo Vision-t. Láthattuk az eseményvezérelt keret interaktív objektumait. Talán kezdjük érezni, hogy milyen lehetőséget ad számunkra a Turbo Vision alkalmazása. Javasoljuk az olvasónak, hogy próbáljon a *Tv_demo* programon változtatni, majd próbálja a saját programváltozatát futtatni.

A következő fejezetek megmutatják a Turbo Vision programok felépítésének lépéseit; hogy hogyan alakul a váz felhasználói programmá.

A *Tv_demo* program listája:

```
{***********************************************}
{                                               }
{   Turbo Pascal 6.0                            }
{   Demo program Turbo Vision használatára      }
{                                               }
{***********************************************}

program tv_demo;

uses Objects, Drivers, Views, Menus, Dialogs, App;
```

```
const
  GreetThemCmd = 100;

type
  PTvApp = ^TTvApp;                  { A származtatott objektumra      }
                                     { mutató pointer.                 }
  TTvApp = object(TApplication)      { TTvApp objektum származtatása.  }
    procedure InformationBox;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitMenuBar; virtual;
    procedure InitStatusLine; virtual;
  end;

procedure TTvApp.InformationBox;
var
  R: TRect;
  D: PDialog;
  C: Word;
begin
  { Dialógus doboz létrehozása. }
  R.Assign(25, 5, 55, 16);
  D := New(PDialog, Init(R, 'Turbo Vision'));

  { Vezérlések létrehozása és betétele a dialógus dobozba }
  R.Assign(3, 5, 15, 6);
  D^.Insert(New(PStaticText, Init(R, 'Jellemzői:')));

  R.Assign(14, 2, 28, 4);
  D^.Insert(New(PButton, Init(R, 'Flexibilis', cmCancel, bfNormal)));

  R.Assign(14, 4, 28, 6);
  D^.Insert(New(PButton, Init(R, 'Praktikus', cmCancel, bfNormal)));

  R.Assign(14, 6, 28, 8);
  D^.Insert(New(PButton, Init(R, 'Kiváló', cmCancel, bfNormal)));

  R.Assign(14, 8, 28, 10);
  D^.Insert(New(PButton, Init(R, 'Jó', cmCancel, bfNormal)));

  { Megjeleníti a módbeállító dialógus dobozt. }
  C := DeskTop^.ExecView(D);
end;

procedure TTvApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);        { Esemény kezelése. }
  if Event.What = evCommand then
  begin
    case Event.Command of
      GreetThemCmd: InformationBox;
    else
      Exit;
```

```
    end;
    ClearEvent(Event);
  end;
end;

procedure TTvApp.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;        { A menüsor inicializálása. }
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~I~nformáció', hcNoContext, NewMenu(
      NewItem('~T~urbo Vision ...','', 0, GreetThemCmd, hcNoContext,
      NewLine(
      NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
      nil)))), nil))));
end;

procedure TTvApp.InitStatusLine;
var
  R: TRect;
begin
  GetExtent(R);                 { A státuszsor inicializálása. }
  R.A.Y := R.B.Y-1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('', kbF10, cmMenu,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit, nil)), nil)));
end;

var
  TvInf: TTvApp;

begin
  TvInf.Init;               { Inicializálás }
  TvInf.Run;                { Futtatás       }
  TvInf.Done;               { Memória felszabadítás }
end.
```

6. TURBO VISION ALKALMAZÓI PROGRAMOK

Ebben a fejezetben részletesen foglalkozunk a Turbo Vision alkalmazásával. Segítségünkre szolgálnak a TVGUID01 - TVGUID22 Pascal demonstrációs programok, amelyeket abból a célból fejlesztettek ki, hogy a magasszintű programozás a felhasználó számára fokról fokra érthetőbbé váljék. Az első, szinte egy egyszerű keretprogram, majd a további programok lépésről lépésre kisebb részlettel bővülnek. Futtassuk egymásután a programváltozatokat, így jobban megértjük a programok működési menetét.

6.1. Menütervezés

Kezdjük el az ismerkedést a TVGUID01 programmal. Az alkalmazói objektum az *App* unit-ban található, ez az ún. *TApplication*. Hozzuk létre a leszármazottját, melyet nevezzünk el *TMyApp*-nek. Majd ennek metódusait fogjuk a későbbiekben felülírni.

A TVGUID01 program fő része:

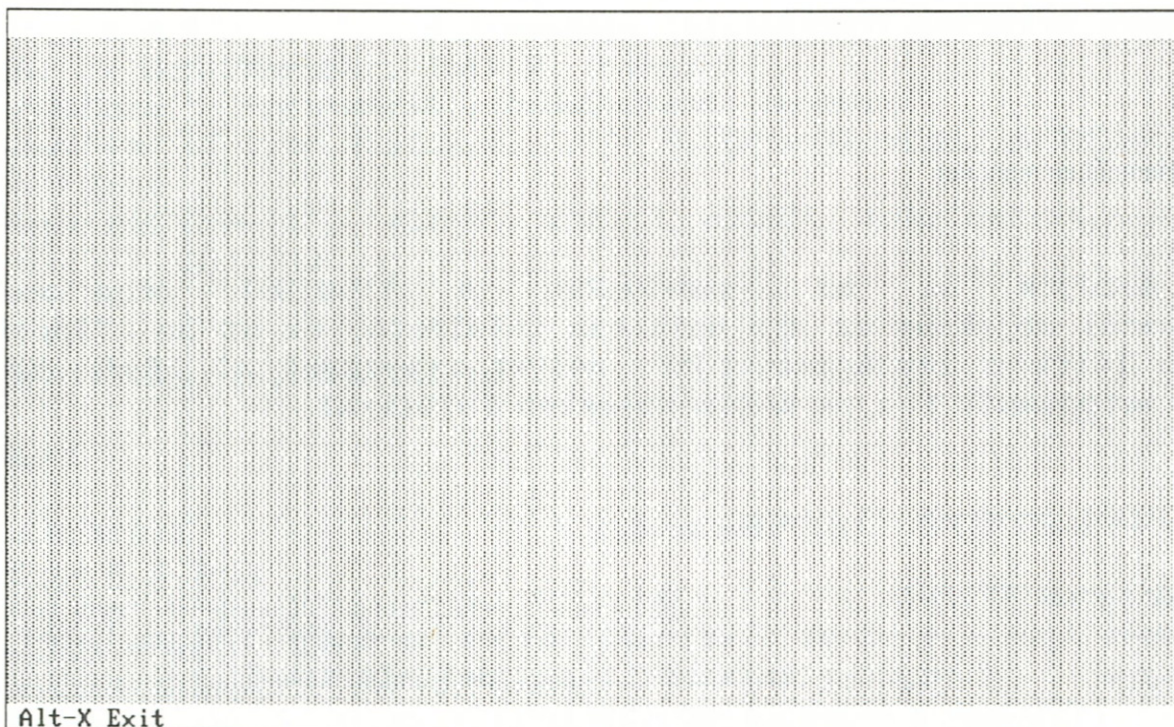
```
program TFirst;
uses App;    { ez az alkalmazói objektum a APP.TPU unit-ból }
type      { saját alkalmazói típus definiálása }
     TMyApp = object(TApplication)
     end;
var
     MyApp : TMyApp; { leszármazott deklarálása }
begin
     MyApp.Init;      {  inicializálás }
     MyApp.Run;       {  interaktív kapcsolat a felhasználóval }
     MyApp.Done;      {  törlések, memória felszabadítás }
end.
```

Ebben a feladatban a *TMyApp* objektumot még nem egészítettük ki újabb funkciókkal. Tulajdonképpen soha nem kell új objektumot létrehoznunk a mezőikkel és a metódusaikkal, hanem egyszerűen deklarálnunk kell egy *MyApp* változót, mint a *TApplication* leszármazott példányát. Abból a célból hoztuk létre a *TMyApp* típust, hogy a programot flexibilisen fejleszthessük tovább és a feladatoknak megfelelően újabb és újabb funkciókkal (metódusokkal) bővíthessük.

A program összesen három utasításból áll, a kezdeti értékadást végzi el a *MyApp.Init*, a felhasználóval való interaktív kapcsolatot a *MyApp.Run* intézi, majd a program befejezése előtt a képernyőtörlést, memória felszabadítást a *MyApp.Done* hajtja végre.

A TVGUID01 program tehát üres menüsort jelenít meg, a státuszsorban pedig közli, hogy az *ALT-X* hatására megszakítja a program futását. Ez a program többet nem tud, bővíteni kell ahhoz, hogy más funkciókat is végre tudjon hajtani.

A *TApplication* objektum alapértelmezés szerint az 6.1. ábrán látható képernyőt jeleníti meg.



6.1. ábra
Az alapértelmezés szerinti *TApplication* képernyője

Bővítsük tovább a TVGUID01 programot, hozzunk létre menüpontokat, bővítsük a státuszsorát is. A TVGUID01 program *desktop*-ját, az üres menüsort és a státuszsort a *TApplication* objektum *InitDeskTop, InitMenuBar* és az *InitStatusLine* metódusai hozták létre. Ezt a három metódust a *TApplication.Init* aktiválta. Ezeket nem kell közvetlenül hívnunk, helyette írjunk egy saját *Init* metódust, amely első utasításként meghívja a *TAppliaction.Init* metódust.

```
procedure TMyApp.Init;
begin
    TApplication.Init;   { először az ős metódusát kell hívni }
    ...          { itt kell inicializálni a saját alkalmazást }
    ...
end;
```

Természetesen, ha a program használja a menüsort, a státuszsort és a billenyűzetet, akkor a **uses** utasítássorban még meg kell adni az *Object*, a *Menus* és a *Drivers* unit-okat.

Ha a program semmilyen speciális inicializálást nem hajt végre, akkor egyszerűen az örökölt *Init* metódust használja. Mivel az *Init* *InitDeskTop*, *InitMenuBar* és az *InitStatusLine* metódusok virtuálisak, amikor hívjuk az örökölt *Init*-et, az hívja a megfelelő *InitStatusLine* és *InitMenuBar* metódusokat. Ezek bemutatására szolgál a TVGUID02.PAS program. Az *InitDeskTop*, *InitMenuBar* és az *InitStatusLine* kezdeti értéket ad a *DeskTop, MenuBar* és a *StatusLine* globális változóknak.

Mivel a *desktop* nagyon fontos objektum, ezért ügyeljünk a következőkre: sohasem szabad felülírni az örökölt inicializáló metódust, az *TApplication.InitDeskTop*-ot. Legyen a *MyApp*-nak egy saját *desktop*-ja. A *MyApp* saját példányai jelennek meg a menüponton való klikkentés hatására, az új *view*-nak csatlakoznia kell a *DeskTop*-hoz. A *desktop* saját maga tudja, hogy hogyan kell kezelni a *view*-kat.

A TVGUID02 programban bővítsük a státuszsort! Ennek érdekében módosítanunk kell a *TApplication.InitStatusLine* metódusát. A *view* típusú *TStatusLine* aktiválja a *StatusLine*-t a *hot key*-k definiálása és megjelenítése céljából. A *StatusLine* a képernyő alján a baloldalon jelenik meg, ez nem azt jelenti, hogy ide mást nem írhatunk ki. A *StatusLine* feladata még a parancsok aktiválása, amelyeket a *hot key* megnyomásával, vagy az egérrel klikkentve választhatunk ki.

A TVGUID02.PAS programban a *TApplication.InitStatusLine* metódust felülírjuk a program státuszsorának bővítése érdekében.
Az új metódus a következő:

```
procedure TMyApp.InitStatusLine;
var  R: TRect; { R a státussor határait tartalmazza }
begin
    GetExtent(R);  { Az R maximális méretűre való beállítása }
    R.A.Y := R.B.Y -1;

        { A PStatusLine által mutatott címen egy új objektum-
          példányt hoztunk létre a megfelelő konstruktor metódus
          hívásával. }
    StatusLine := New(PStatusLine, Init(R,
     NewStatusDef(0, $FFFF,
     NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
     NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,
     nil)),
    nil)
   ));
end;
```

A *StatusLine* az alkalmazói program státuszsorára mutató pointert tartalmazza. Az *TProgram.InitStatusLine* ad kezdőértéket a *StatusLine* változónak, a *TProgram.Init* hívja az eljárást. Ha a kezdőérték **nil**, akkor az alkalmazói programnak nincs státuszsora.
Ne felejtsük el, hogy a *TMyApp* deklarációját ki kell egészítenünk:

```
procedure InitStatusLine; virtual;
```

utasítással.

Az inicializálás egymásba ágyazott szabványos Turbo Vision függvények, a *NewStatusDef, NewStatusKey* és a *NewStatusBar* hívásából áll.

A *NewStatusDef* függvény deklarációja:

```
function NewStatusDef( Amin, Amax: Word; Items : PStatusItem;
                       ANext: PStatusDef): PStatusDef;
```

A *NewStatusDef* függvény memóriahelyet foglal egy új *TStatusDef* rekordnak és a memóriahelyre mutató pointert adja vissza. A rekordot az adott paraméterek inicializálják. A statátuszsort a *NewStatusDef* hívással és a beleskatulyázott *NewStatusKey* hívásával állítjuk elő.

A *NewStatusKey* függvény deklarációja:

```
function NewStatusKey(AText: String; AKeyCode : Word; ACommand: Word;
                      ANext: PStatusItem): PStatusItem;
```

A *NewStatusKey* függvény memóriahelyet foglal egy új *TStatusItem* rekordnak és a memóriahelyre mutató pointert adja vissza. A rekordot az adott paraméterek inicializálják (a *NewStr*-t használjuk helyet foglalni a *AText* számára.) Ha az *AText* üres (**nil**), akkor a státuszsor rejtett, egyébként adott billentyűleütéshez *(AKeyCode)* parancsot rendelünk (*ACommand*).

A TVGUID02 program a státuszsor help szövegének tárolására 0-tól $FFFF-ig definiál egy határt, amelyen keresztül létesül a kapcsolat a Turbo Vision *cmQuit* parancs és az *ALT-X hot key*, valamint a *cmClose* szabványos parancs és az *ALT-F3* billentyű kombináció lenyomása között. A sztringnek két tilde jel (~) közé zárt része nagyobb fényerővel világít. A parancsot aktiváljuk, ha a képernyőn megjelenített sztringen belül bárhol klikkentünk az egér bal oldali gombjával.

Definiáljunk egy új parancsot egy új ablak nyitására

```
const
    cmdNewWin = 199;
```

A **New** nevű parancsot a menüben az *F4* billentyűvel fogjuk aktiválni. Ehhez a státuszsort az alábbiak szerint kell módosítanunk:

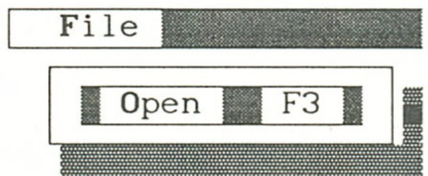
```
StatusLine := New(PStatusLine, Init(R,
  NewStatusDef(0,$FFFF,
     NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
     NewStatusKey('~F4~ New', kbF4, cmNewWin,
     NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose);
     nil))),
  nil)
));
```

Ha a példaprogramból megértettük a státuszsor inicializálását, akkor könnyen tervezhetünk hasonlót.

A Turbo Vision menüsor változója a *MenuBar*, amely a *NewMenu*, *NewSubMenu*, *NewItem* és a *NewLine* Turbo Vision függvények egymásbaskatulyázott hívásával kap értéket.

Ha a menü inicializálása elkészült, akkor a menüpont (még help nélkül ugyan) kezelni tudja a felhasználó inputját.

Inicializáljunk egy egyszerű főmenüt, amely egyetlen almenüt tartalmaz, a *hot key* legyen az *F3*.



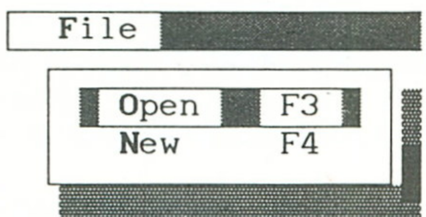
A programrészlet az alábbi:

```
const
     cmFileOpen = 200;  { új parancs definiálása }

procedure TMyApp.InitMenuBar;
var  R: TRect;
begin
  GetExtent(R);         { érzékeli az alkalmazói területet }
  R.B.Y := R.A.Y + 1; { beállítja az első sort }
  { létrehoz egy menüt a menüpontjaival }
  MenuBar := New(PmenuBar, Init(R, NewMenu(
     NewSubMenu('~F~ile', hcNoContext, NewMenu(
       NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
       nil)),         { nincs több menüpont }
     nil)             { nincs több almenü   }
   )));               { menü vége           }
end;
```

Az egyetlen *File* menünek egyetlen menüPontja van, **O***pen* . A *File*-ban az **F**, az **O***pen*-ben az **O** a közvetlen kiválasztásra szolgáló billentyű az *ALT* billentyűvel együtt. Az **O***pen* menüponot még az *F3* billentyűvel is ki lehet választani.

Mindegyik menüponthoz help-et is fűzhetünk, minden help-hez egy szám tartozik, mellyel nagyon könnyű biztosítani a hozzátartozó szövegösszefüggést. A *view*-nak alapértelmezés szerint a *hcNoContext* egy speciális help-et biztosít, amely nem változtatható. A menü inicializálásánál jelenik meg a help szövegét azonosító szám, mivel ezt az objektumnak a hierarchikus struktúrája kezeli, később az azonosítás már bonyolultabb lenne. Ha saját help-et akarunk készíteni, akkor a megfelelő értéket helyettesítsük *hcNoContext* helyén az *Init* metódusban.

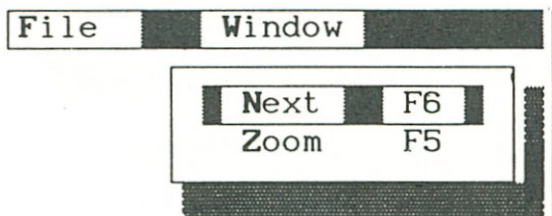
Bővítsük tovább a *File* menüt, legyen még egy menüpontja a **N***ew*, amely az *F4* billentyű hatására is működik.



A programrészlet az alábbi:

```
MenuBar := New(PmenuBar, Init(R, NewMenu(
   NewSubMenu('~F~ile', hcNoContext, NewMenu(
     NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
     NewItem('~N~ew',  'F4', kbF4, cmNewWin, hcNoContext,
     nil))),
   nil)
 )));
```

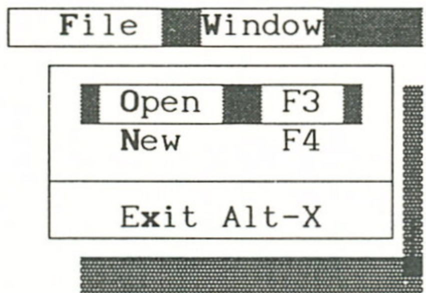
Most bővítsük a főmenüt a *Window* menüponttal. A *Window* almenü pedig két pontot tartalmazzon.



A szükséges programrészlet az alábbi:

```
MenuBar := New(PmenuBar, Init(R, NewMenu(
   NewSubMenu('~F~ile', hcNoContext, NewMenu(
     NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
     NewItem('~N~ew',  'F4', kbF4, cmNewWin, hcNoContext,
     nil))), { főmenüpont lezárása }
   NewSubMenu('~W~indow', hcNoContext, NewMenu(
     NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
     NewItem('~Z~oom',  'F5', kbF5, cmZoom, hcNoContext,
     nil)),
     nil))) { menük zárása }
 )));
```

A *File* menü pontjait bővítsük olymódon, hogy az előzőektől egy vonallal válasszunk el egy újabb menüpont csoportot:



A TVGUID03.PAS programból részlet:

```
MenuBar := New(PmenuBar, Init(R, NewMenu(
   NewSubMenu('~F~ile', hcNoContext, NewMenu(
     NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
     NewItem('~N~ew',  'F4', kbF4, cmNewWin, hcNoContext,
     NewLine(
     NewItem('~E~xit',  'Alt-X', kbAltX, cmQuit, hcNoContext,
     nil))))), { főmenüpont lezárása }
   NewSubMenu('~W~indow', hcNoContext, NewMenu(
     NewItem('~N~ext', 'F3', kbF6, cmNext, hcNoContext,
     NewItem('~Z~oom',  'F4', kbF5, cmZoom, hcNoContext,
     nil))),
     nil)) { menük zárása }
 )));
```

A TVGUID03.PAS program státuszsorában az *F10* billentyű a *cmMenu* parancshoz van rendelve. A *cmMenu* a Turbo Vision szokványos parancsa, amely arra szolgál, hogyha nincs a számítógéphez egér csatlakoztatva, akkor a főmenüsort az *F10* billentyűvel lehessen kiválasztani. Ilyenkor az első menü lesz aktív, a további menük már a kurzor billentyűkkel választhatók ki.

6.1.1. Ablak (*window*) nyitása

A Turbo Vision alkalmazásokban a *window* egy olyan objektum, amelynek az a tulajdonsága, hogy válaszol a felhasználó által kezdeményezett inputra anélkül, hogy programutasításokat írtunk volna. A Turbo Vision *window* objektuma kezelni tudja egy ablak nyitását, úrjaméretezését, mozgatását és zárását. A Turbo Vision *window* tartalmazza és kezeli a többi objektumot is, pl. azokat az objektumokat, amelyek megjelennek a képernyőn, de más funkciót töltenek be. A *window* kezeli a *view*-kat is, amelyek különbözőképpen jelenhetnek meg a képernyőn.

A közvetkező lépésként megismerkedünk azzal a módszerrel, hogy a Turbo Vision adta vázat hogyan kell bővíteni, ha egy ablakba szeretnénk írni. A TVGUID04.PAS program bemutatja, hogyan kell egy *window* típust létrehozni és a *desktop*-hoz illeszteni. Ne felejtsük el a *TMyApp* típust a deklarációból. A programban tehát definiáltunk egy új típust, a *TDemoWindow*-t anélkül, hogy bármilyen mezőt, vagy metódust adtunk volna az őstípusához.

A programrészlet a TVGUID04 programból:

```
uses Views;
const
     WindCount: Integer = 0;
               { kezdőértéket kap a window számláló }
type
     PDemoWindow =^TDemoWindow;
     TDemoWindow = object(TWindow) { definiál egy új ablaktípust,
                                     ami még semmi újdonságot nem
                                     tartalmaz az TWindow-hoz
                                     képest  }
     end;

procedure TMyApp.NewWindow;
var
     Window: PDemoWindow;      { egy ablak példányra mutató pointer}
     R: TRect;
begin
     Inc(WinCount);
     R.Assign(0, 0, 26,7);    { kezdő méret és pozíció beállítása }
     R.Move(Random(53), Random(16));       { véletlenszerűen mozog }
     Window := New(PDemoWindow, Init(R, 'Demo Window', WinCount));
            { Létrehozzuk a TDemoWindow egy példányát a heap-ben }
     DeskTop^.Insert(Window);             { leteszi a desktop-ra }
end;


procedure TMyApp.HandleEvent(var Event: TEvent);
begin
     TApplication.HandleEvent(Event);
     if Event.What = evCommand then
```

```
    begin
      case Event.Command of          { válasz az újabb parancsra }
            cmNewWin: NewWindow;
             { művelet definiálása a cmNewWin parancs számára }
      else
            Exit;
      end;
       ClearEvent(Event);
                             { törli az eseményt a végrehajtás után }
    end;
end;
```

Ezt az ablakot úgy tudjuk használni a programban, hogy a *cmNewWin* parancsot vagy egy menü opcióval, vagy a státussor *hot key*-ével kötjük össze, ahogy korábban tettük. Amikor a *cmNewWin* parancsot kap, a Turbo Vision a parancsot elküldi a *TMyApp.HandleEvent*-nek, amely erre a *TMyApp.NewWindow* meghívásával válaszol.

6.1.2. Ablak (*window*) inicializálása

Egy Turbo Vision ablak kezdőértékét három paramétere határozza meg:

- a mérete és pozíciója a képernyőn,
- a fejléce és,
- a sorszáma (*window* sorszám).

Az első paraméter, amely meghatározza az ablak méretét és pozícióját, a *TRect,* a Turbo Vision téglalap objektuma. A *TRect* nagyon egyszerű objektum, amelynek a saját *Assign* metódusa ad értéket. A méret és a pozíció megadása a téglalap bal felső és a jobb alsó sarokpontja szerint történik. Többféle módon adhatunk értéket vagy változtathatjuk a *TRect* objektum értékét.

A TVGUID04 programban az *R* a *desktop* origójában jelenik meg, azután véletlenszerű távolsággal mozog a *desktop*-on. Természetesen, normál esetben nincs szükségünk véletlenszerű mozgatásra, de a feladatunk az volt, hogy több ablakot nyissunk meg, lehetőleg nem ugyanazon a helyen.

A második inicializáló paraméter egy sztring, amely az ablak tetején, mintegy fejléc jelenik meg.

Az utolsó paramétert a *window* objektum *Number* nevű mezője tartalmazza. Ez a szám 0 - 9 között változik, az ablakkereten is megjelenik, így a számozott ablakot közvetlenül is kiválaszthatjuk az *ALT-1* ... *ALT-9* billentyűkombináció megnyomásával.

Ha nem adunk az ablaknak sorszámot, akkor erről gondoskodik a Turbo Vision a *wnNoNumber* konstans segítségével.

Az *Insert* metódus használatával egy ablak automatikusan megjelenik a *desktop*-on, ilyenkor az *Insert* metódus egy megjelenítésre vonatkozó vezérlést küld egy másik *view* metódusnak.
Amikor végrehajtjuk a

```
Desktop^.Insert(Window);
```

utasítást, akkor egy *window* jelenik meg a *desktop*-on. Hasonlóan a *desktop*-hoz, bármilyen számú megjelenítést helyezhetünk el egy csoport (*group*) objektumba. A csoport, amely a megjelenítést tartalmazza, az ún. tulajdonos (*owner) view,* és a *view,* amelyet a csoportba beirattuk az ún. *subview.* A *subview* maga is lehet csoport és lehetnek saját *subview*-i. Például ha egy *window*-t inzertálunk a *desktop*-ra, akkor a *window* is egy *subview,* amely lehet egy keret, görgetési (scroll) vezérlő vagy más *subview.*

Klikkentve a záró ikonon, a *window* ugyanazt az *cmClose* parancsot generálja, amelyet a státussor menüpontja az *ALT-F3*-mal összekötve ad ki. Az ablak nyitása az *F4* vagy a **File|Open** menüpont kiválasztásával történik, ez a *cmClose* parancsot automatikusan hatástalanítja.

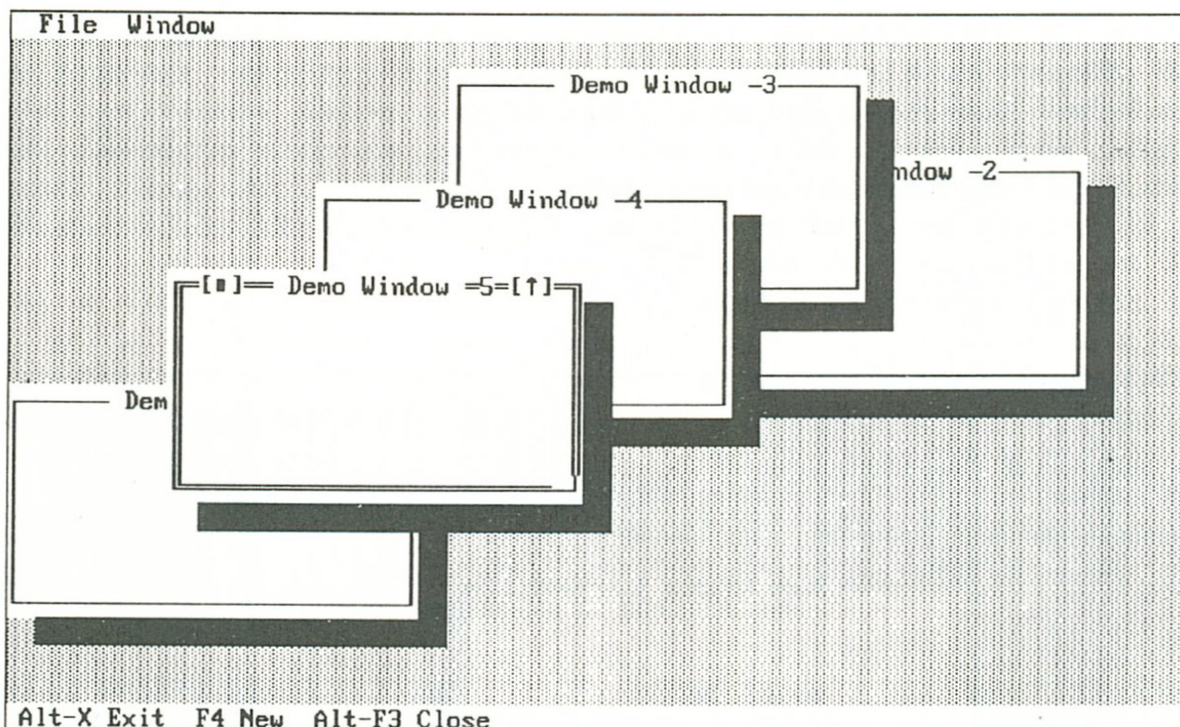
Az ablak lezárására nem kell utasítást írnunk. Amikor klikkentünk a záró ikonon, a Turbo Vision ezt is megteszi helyettünk. Alapértelemzés szerint a *window* úgy válaszol a *cmClose* parancsnak, hogy meghívja a *Done* destruktort:

```
Dispose(MyWindow, Done);
```

Egy adott *view* összes *subview*-jának *Done* metódusa hívásra kerül, ráadásul, ha egy *window* konstruktoraival memóriahelyet foglaltunk, akkor szükséges, hogy azt a megfelelő *Done* metódus hívásával felszabadítsuk.

Futtassuk le a TVGUID04 programot, amellyel több ablakot nyithatunk a *desktop*-on. Az egér klikkentésével az új ablakok sorozatát hozhatjuk létre. Ha használjuk a *cmNewWin*-t, akkor az ablakok a saját számukkal jelennek meg. Ezeknek az ablakoknak megváltoztathatjuk a méretét, kiválaszthatjuk őket és ide-oda mozgathatjuk őket a képernyőn.

A *TWindow* egy csoport, amely kiindulásként tartalmaz egy *view*-t, a *TFrame*-t. Ha az egérrel klikkentünk a keret ikonon, akkor mozgathatjuk, újraméretezhetjük és lezárhatjuk az ablakot. A keret megjeleníti a fejlécet, amelyen keresztül történik az ablak inicializálása, és megrajzolja az ablak hátterét, éppenúgy ahogy a *TBackGround* a *desktop*-ot. Minden úgy történik, ahogy a 6.2. ábrán is láthatjuk, és nem kellett egyéb utasítást írnunk.



6.2. ábra
Több ablak nyitása

A következő lépésként megimerkedünk azzal a módszerrel, hogyan kell az ablakba valamit beleírni. A *TWindow* egy Turbo Vision csoport, egy *TGroup* objektum, nincs képernyő reprezentációja, mivel csak egy keret. Valamit újra hozzá kell adnunk, hogy bővítsük a lehetőségeket. Olyan *view*-t kell terveznünk, amely megjelenik az ablakban a képernyőn. Ezt a *view*-t *interior*-nek (belvilágnak) nevezzük.

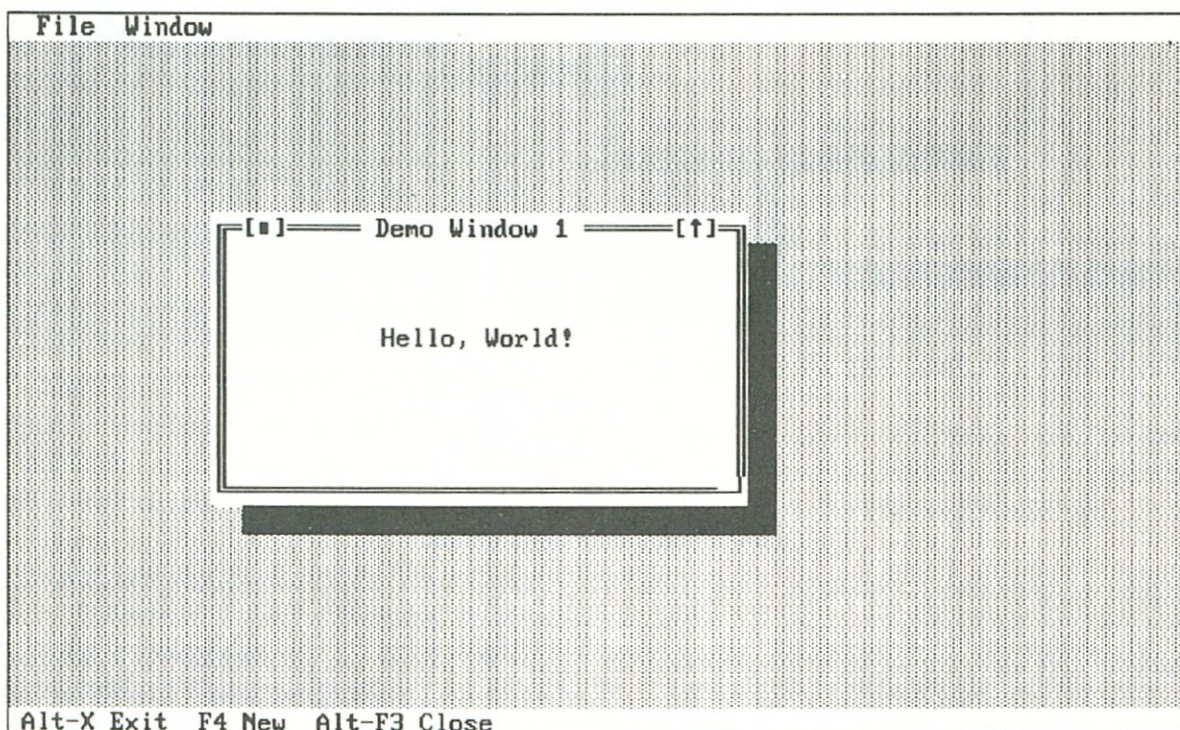
Ez az első *interior* teljesen kitölti az ablakot, későbbiekben látni fogjuk, hogy lesznek olyanok is, amelyek csökkentik a méretüket és helyet csinálnak más *view*-k számára. Az ablak tudja sokszorozni az *interior*-öket és bármely más hasznos *view*-kat, például input sorokat, cimkéket, vezérlő gombokat vagy opcióbeállító dobozokat. Tapasztalni fogjuk, hogy milyen könnyen készíthetünk az ablak keretében görgető (*scroll*) menü megjelenítőket.

Egy csoporton belül lekicsinyítve, egymás mellé rendezve, vagy átlapolva is megjeleníthetjük a *subview*-kat. A *Tile* a *TDeskTop* metódusa. Ez a kijelölt *subview*-kat úgy alakítja át, hogy mindegyik látható legyen a képernyőn. Ezt a metódust egyedül csak a *desktop*-nál lehet alkalmazni.

Az *interior* a *TView*-nak egy egyszerű leszármazottját hozza létre. Bármely *TView*-nak van kerete, amely ugyanúgy működik, mint a hagyományos ablak kerete. A keret nem klikkenthető, a megjelenítés belül történik.

Ha a *TView interior* teljesen kitölti az ablakot, semmi sem történik akkor, ha a ablak kerete lefedi az *interior* keretét. Ha az *interior* keret kisebb az ablaknál, akkor ez a belső keret fog látszani. Hogy hogyan kell sokszorozni a belső keretet az ablakon belül kerettel ábrázolva, azt egy későbbi példán fogjuk látni.

A következő program bemutatja, hogyan írjuk ki a "Hello World!" szöveget az ablakba (6.3. ábra).



6.3. ábra
Ablakba történő kiírás

Az összes Turbo Vision *view* képes "rajzolni". Ezt a "rajzolást" a *Draw* metódus hajtja végre. Ha egy új képernyő reprezentációt akarunk tervezni, akkor létre kell hoznunk egy *view* leszármazottat, felül kell írnunk a *Draw* metódust, hogy megtanítsuk az új objektumot arra, hogyan kell önmagát a képernyőn megjeleníteni. A programban definiálnunk kell a *TView* leszármazottját, a *TInterior*-t és írni kell egy új *Draw* metódust.

Megjegyezzük, hogy az új *Interrior.Draw* hívja a *TView* ősének a *Draw* metódusát, amely ez esetben éppen törli a megjelenített téglalap területét. Általában, amikor írunk valamit az ablak belső részébe, nem kell meghívni a leszármazott *Draw* metódust, mivel hívása vibrálást okoz a képernyőn a belső ablak egynél többször való rajzolása miatt. Ezt tapasztaljuk a TVGUID05.PAS programban, ha a *TView.Draw* metódus aktiválásánál megszüntetjük a kommentet. Mozgassuk és méretezzük újra az ablakot. Teljesen érthetővé válik, hogy miért kell a *view* teljes tartománya miatt a felelősséget a lefedés szempontjából a *view*-ra hárítani.

A Turbo Vision hívja a megjelenítés *Draw* metódusát mindig, amikor nyitjuk, zárjuk, mozgatjuk vagy újraméretezzük az ablakot. Ha fel kell frissíteni a képet, akkor a *Draw* helyett a *DrawView* metódust hívjuk. A *DrawView* metódus újrarajzolja a képet, ha szükséges. Meg kell jegyeznünk, hogy közvetlenül ne hívjuk a *Draw* metódust, hanem helyette a *DrawView*-t kell aktiválni.

Ha a Turbo Vision-t használjuk, akkor nem használhatjuk a Turbo Pascal Write eljárásait, mivel nem ismerjük a aktuális megjelenítés helyi koordinátáit, nem tudjuk kezelni a keret határánál a vágást, a színhasználat is probléma.

A Turbo Vision *WriteStr* eljárása nemcsak hogy tud írni a helyi koordinátákkal, hanem figyelembe veszi a kereten lévő vágást és használni tudja a színpalettát. A *WriteStr*-hez hasonlóan működik a *WriteChar* eljárás is, definíciója a következő:

```
WriteChar(X, Y, Ch, Color, Count);
```

A *WriteChar* az ablakon belül az x és y koordinátahelyre, a *Ch* karakterből *Count* számút, a színpaletta *Color* sorszámú színével ír. Minden egyes *Write* típusú metódust a *view Draw* metódusán belül kell hívni, mert csak itt van mód arra, hogy bármit is írjunk a Turbo Vision-on belül.

6.1.3. Egyszerű megjelenítő

Bővítsük tovább az ablak lehetőségeit, például adjunk hozzá egy olyan metódust, amely text file-*ból* olvas és megjeleníti a beolvasott szöveget az *interior*-ben.

A TVGUID06.PAS programban a belső ablak kinagyításakor ún. "szemét" karakterek jelennek meg. (Ez azonban így természetes, majd később meglátjuk, miért.)

A programrészlet az alábbi:

```
const
     MaxLine = 100; { a sorok száma }
var
     LineCount : Integer;
     Lines:      array[0 .. Maxline-1] of PString;
PInterior = ^TInterior;
TInterior = object(TView);
     constructor Init(var Bounds: TRect);
     procedure Draw; virtual;

end;
```

```
procedure TInterior.Draw;            { ez nem tökéletes metódus! }
var
     Y: Integer;
begin
     for Y:=0 to Size.Y-1 do              { egyszerű sorszámláló }
     begin
          WriteStr(0, Y, Lines[Y]^,$01); { minden sort kiir }
     end;
end;

                                  { text file beolvasása }
procedure ReadFile;
var
     F: Text;
     S: String;
begin
     LineCount :=0;
     Assign(F, FileToRead);
     Reset(F);
     while  not Eof(F) and (LineCount < MaxLines) do
     begin
          ReadLn(F, S);
          Lines[LineCount]:=NewStr[S];
          Inc(LineCount);
     end;
     Close(F);
end;
  { a végrehajtás után felszbadul a Lines tömb }
  { által lefoglalt memória terület            }

procedure DoneFile;
var
     I: Integer;
begin
     for I:=0 to LineCount-1 do
     if Lines[I] <> nil then DisposeStr(Lines[I]);
end;
```

A program *ReadFile* eljárása olvassa be a text file-t és tölti a *Lines* tömbbe, a *DoneFile* a végrehajtás után felszabadítja a *Lines* tömb által lefoglalt helyet. A *ReadFile* eljárásban a *PString* pointer globális. A *NewStr* függvény hívásakor a Turbo Vision a sztringet a *heap*-ben tárolja és visszaadja a sztringre mutató pointer értékét. Itt nem szabad használni a *Dispose* eljárást, hanem helyette a *DisposeStr* eljárást kell aktiválni a sztring helyének felszabadítására.

6.1.4. Pufferelt rajzolás

Az előző program futtatásakor a képernyőn különféle "szemét" karaktereket is láttunk ott, ahol a sornak üresnek kellett volna lennie. Ez annak a következménye, hogy a *Draw* metódus hívása előtt nem foglaltuk le a teljes területet, amely a biztonságos megjelenítéshez kellett volna.

A *Lines* sztring tömb nem megfelelő forma a megjelenítésre. A szöveg tipikusan változó hosszúságú sztringekből áll, sok közülük üres (zérus hosszúságú), a *Draw* metódusnak a teljes területet le kell fedni, így a szöveg sorait a megjelenítés szélességében ki kell terjeszteni. Ennek megvalósítására módosítani kell *Draw* metódust, amely minden képernyőre való írás előtt minden sort betölt egy pufferba. A *TDrawBuffer* szintén globális típus:

```
TDrawBuffer = array[0..MaxViewWidth-1] of Word;
```

A *TDrawBuffer* tartalmazza a változó tulajdonságokat és a karakterek byte-jait.

A TVGUID07 programban az új *TInterior.Draw* a következő:

```
procedure TInterior.Draw;
var
     Color : Byte;
     Y:      Integer;
     B:      TDrawBuffer;
begin
     Color: GetColor(1);
     for Y:=0 to Size.Y-1 do
     begin
          MoveChar(B,' ',Color,Size.X);
          { a sort space-val tölti fel }
          if (Y< LineCount) and (Lines[Y] <> nil) then
               MoveStr(B,Copy(Lines(Y)^,1,Size.X,Color);
                    { bemásolja a szöveget }
               Writeln(0, Y, Size.X, 1, B);
                    { kiírja a sort }
     end
end;
```

A *Draw* először a *MoveChar* eljárást hívja, amely a megjelenítés szélességével, *Size.X* darab megfelelő színű üres hellyel tölti fel a *TDrawBuffer*-t. Majd a *MoveStr* bemásolja a szöveget a pufferba, azután *WriteLine* a teljes puffer tartalmát jeleníti meg.

A Turbo Vision négy globális eljárást tartalmaz. Ezek szöveget másolnak a *TDrawBuffer*-be: a *MoveStr*, (amint láttuk a mintaprogramban)

a *MoveChar*, *MoveCstr* és a *MoveBuff*, amelyek rendre karaktereket, vezérlő sztringeket (sztring ismétlő -tilde- karakterekkel menűk és státusz pontok számára) és más puffereket mozgatnak egyenként a pufferbe.

A Turbo Vision kétfajta eljárást kínál, amely a puffer tartalmát írja a képernyőre. Az egyik a

```
WriteLine(X, Y, W, H, Buf);
```

amint a TVGUID07 programban láthattuk.

Az *TInterior.Draw* metódusban a *WriteLine* a *TDrawBuffer* tartalmát egy sorba írja, ha a negyedik paraméter, a *H* (magasság) nagyobb, mint 1, a *WriteLine* ismétli a puffer tartalmát a következő sorokba. Így, ha a puffer tartalma " Hello, World!", akkor a *WriteLine(*0, 0, 13, 4, Buf*)*, hatására

```
Hello World!
Hello World!
Hello World!
Hello World!
```

jelenik meg a képernyőn.

A másik eljárás a *WriteBuf(H, Y, H, Buf)* a képernyő téglalap alakú tartományába ír. *W* és a *H* a puffer szélessége és magassága. Ha a puffer a "ABCDEFGHIJKLMNOP" szöveget tartalmazza, akkor a

```
WriteBuf(0, 0, 4, 4, Buf);
```

utasítása következő képet eredményezi:

```
ABCD
EFGH
IJKL
MNOP
```

Másképpen működnek a puffer nélküli eljárások, *WriteStr* és a *WriteChar*.

Az *TInterior.Draw* úgy rajzol, hogy a file-ból éppen megfelelő részt tölti be a kívánt keretbe, máskülönben sok időt töltene el a file egy adott részének írásával, figyelembe véve a *TInterior* határainak vágását.

Ha a megjelenítés sok időt vesz igénybe, akkor aktiválnunk kell a *GetClipRect* eljárást. A *GetClipRect* azzal a téglalappal tér vissza, amely ki van jelölve, így rajzolhatunk oda. Például, ha van egy összetett dialógus dobozunk, amely vezérlő számot is tartalmaz és mozgatjuk a képernyőn, megnézhetjük, hogy mi van mögötte, hívjuk *GetClipRect* eljárást mielőtt rajzolnánk, hogy megőrizzük a dialógus dobozt abból a célból, hogy ne kelljen újrarajzolni.

6.1.5. Kép görgetés előre és hátra

A file megjelenítő a file-ból csak azt a néhány sort mutatta meg, amely belefért az ablakba. Hogy az információ görgethető is legyen, természetesen a *TDemoWindow* objektumot bővítenünk kell a *MakeInterior* metódussal, amely szétválasztja az ablaknyitás és működtetés funkciókat.

Progamrészlet a TVGUID08 programból:

```
type
     PInterior = ^TInterior;
     TInterior = object(TScroller);
          constructor Init(var Bounds: TRect; AHScrollBar,
                     AVScrollBar:PScrollBar);
          procedure Draw: virtual;
     end;

     PDemoWindow = ^TDemoWindow;

     TDemoWindow = object(TWindow)
          constructor Init(Bounds: TRect; WinTitle: String;
                         WindowNo: Word);
          procedure MakeInterior(Bounds: TRect);
     end;

     constructor TInterior.Init(var Bound: TRect;
                    AHScrollBar, AVScrollBar: PScrollBar);

     begin
          TScroller.Init(Bounds, AHScrollBar, AVScrollBar);
          GrowMode := gfGrowHix + gfGrowHiY;
          SetLimit(128, LineCount);
          { vizszintes, függőleges határ }
     end;

     procedure TInterior.Draw;
     var
          Color : Byte;
          Y, I  : Integer;
          B     : TDrawBuffer;
     begin
          Color :=GetColor($01);
            { használja a normál text mód színét }
          for Y:=0 to Size.Y-1 do  { sorok számolása }
          begin
               MoveChar(B,' ',Color, Size.X);
                    { tölti a puffert space-val }
               I:=Delta.Y + Y;
               if  (I<LineCount) and (Lines[I] <> nil) then
                  MoveStr(B, Copy(Lines[I]^,Delta.X +1,
```

```
                                    Size.X), Color);
                        WriteLn(0 , Y, Size.X 1, B);
                end;
            end;

            procedure TDemoWindow.MakeInterior(Bounds: TRect);
            var
                HScrollBar, VScrollBar: PScrollBar;
                Interior: PInterior;
                R: TRect;
            begin
                VScrollBar := StandardScrollBar(sbVertical);
                HScrollBar := StandardScrollBar(sbHorizontal);
                Interior := New(Pinterior, Init(Bounds, HScrollBar,
                                    VScrollBar));
                Insert(Interior);
            end;

            contructor TDemoWindow.Init(Bounds: TRect;
                                        WinTitle: String;
                                        WindowNo: Integer);
            var
                S: string[3];
            begin
                Str(WindowNo,S);
                TWindow.Init(Bounds, WinTitle + ' ' + S,
                                wnNoNumber);
                GetExtent(Bounds);
                Bounds.Grow(-1,-1);
                MakeInterior(Bounds);
            end;
```

A vizszintes és a függőleges görgető sorok (*scroll bar*) inicilizálás után bekerülnek a csoport objektumba, azután átkerülnek a *TScroller* kezdeti értéket adó részébe.

A görgető (*scroller*) *view* típusú, amely arra készült, hogy egy nagyobb, virtuális *view* egy részét jelenítse meg. A görgető sor és a hozzátartozó görgető nyíl vagy négyzet közösen hozzák létre a görgethető képet. Mindezt a *Draw* metódus hozza létre a görgetővel. A görgető négyzet pedig automatikusan vezérli a görgető vezérlőnek a *Delta.X* és a *Delta.Y* értékeit. Természetesen felül kell írni a *TScroller.Draw* metódust, hogy a megfelelő görgetést elvégezze. A görgető négyzet változása maga után vonja a *Delta* értékének változását, a *Draw* metódus fog aktiválódni, ha a *Delta* változott.

6.1.6. Többszörös *view* az ablakban

Nézzük meg, hogyan kell megírni a programot ahhoz, hogy a szöveg file két görgető ablakban jelenjen meg; az egérrel vagy a *TAB* billentyűvel automatikusan kiválasztható legyen az ablak két belső része közül az egyik. Mindegyik *view* egymástól függetlenül görgethető legyen, mindegyiknek legyen saját kurzor pozíciója.

Egy kicsivel kell csak bővítenünk az előbbi *MakeInterior* metódust, hogy tudja, az ablak melyik oldalát választották ki, a *TDemoWindow* metódust két *MakeInterior* hívással kell kiegészíteni

A programrészlet a TVGUID09 programból:

```
procedure TDEmoWindow.MakeInterior(Bounds: TRect;
                                   Left  : Boolean);
var
     Interior : PInterior;
     R: TRect;
begin
     Interior:= New(PInterior, Init(Bounds,
                StandardScrollBar(sbHorizontal),
                StandardScrollBar(sbVertical)));
     if Left then Interior^.GrowMode := gfGrowHiY
             else Interior^.GrowMode := gfGrowHiX +
                                        gfGrowHiY;
             Insert(Interior);
end;

constructor TDemoWindow.Init(Bounds: TRect; WinTitle:String;
                             WindowNo: Word);
var
     S: string[3];
     R: TRect;
begin

     Str(WindowNo, S);
     TWindow.Init(Bounds, WinTitle+' '+S,wnNoNumber);
     GetExtent(Bounds);

     R.Assign(Bounds.A.X, Bounds.A.Y, Bounds.B.X div 2 + 1;
             Bounds.B.Y);

     MakeInterior(R, True);

     R.Assign(Bounds.B.X div 2, Bounds.A.Y, Bounds.B.X,
             Bounds.B.Y);

     MakeInterior(R, False);
end;
```

A *MakeInterior* nemcsak felépítésében, hanem lényegében is változott, mivel nem két statikus görgető négyzettel történik a lapozás, hanem az *Init*-ben paraméterként hívja a *StandardScroller*-t. Példaként futtassuk le a TVGUID09.PAS programot. Szabályozhatjuk, mennyi legyen az ablak még megengedett legkisebb mérete. Ezt a *TWindow* metódus *SizeLimits* felülírásával állíthatjuk be.

A programrészlet TVGUID10 programból az alábbi:

```
procedure TDemoWindow.SizeLimits(var Min, Max: TPoint);
var R: TRect;
begin
     TWindow.SizeLimits(Min, Max);
     GetExtent(R);
     Min.X := R.B.X div 2;
end;
```

Megjegyezzük, hogy nem kell hívni a *SizeLimits* metódust. Ha a felülírás megtörtént, a hívás majd a megfelelő időben meg fog történni. Ugyanezt tettük a *Draw* metódussal, hogy megmondtuk neki, hogyan jelenítse meg sajátmagát, de nem mondtuk meg, hogy mikor. A Turbo Vision tudja, hogy mikor kell hívni a *Draw* metódust. Tehát elég beállítani a limitet és a *view* tudja, hogy a megfelelő időben ellenőrizze azt.

6.2. Dialógus doboz

A dialógus doboz a *window* speciális fajtája. Tulajdonképpen a *TDialog* a *TWindow* leszármazottja, éppenúgy kezelhetjük, mint egy másik ablakot, azonban van néhány dolog, ami különböző.

A dialógus doboz megjelenítéséhez a TVGUID11.PAS programot új menüponttal kell kiegészíteni, amely generálja a megnyitó parancsot a dialógus doboz számára, és a működéshez szükséges egy új metódus, amely a menüponthoz tartozó feladatot látja el. A *HandleEvent* metódusát egy utasítással kell bővíteni, amely a parancsot a művelettel köti össze.

A programrészlet az alábbi:

```
          const
               cmNewDialog = 200;

     procedure TMyApp.InitMenuBar;
     var
          R: TRect;

     begin
          GetExtent(R);

          R.B.Y := R.A.Y + 1;

       MenuBar := New(PMenuBar, Init(R, NewMenu(
          NewSubMenu('~F~ile', hcNoContext, NewMenu(
            NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
            NewItem('~N~ew',  'F4', kbF4, cmNewWin, hcNoContext,
            NewLine(
            NewItem('~E~xit',  'Alt-X', kbAltX, cmQuit, hcNoContext,
            nil))))), { főmenüpont lezárása }

          NewSubMenu('~W~indow', hcNoContext, NewMenu(
            NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
            NewItem('~Z~oom',  'F5', kbF5, cmZoom, hcNoContext,
            NewItem('~D~ialog', 'F2', kbF2, cmNewDialog, hcNoContext,
            nil)))),
          nil))  { menük zárása }
        )));
          end;

          procedure TMyApp.NewDialog;
          var
               Dialog: Pdialog;
               R     : Rect;
          begin
               R.Assign(0,0,40,13);
               R.Move(Random(39),Random(10);
               Dialog:=New(PDialog, Init(R, 'Demo Dialog'));
```

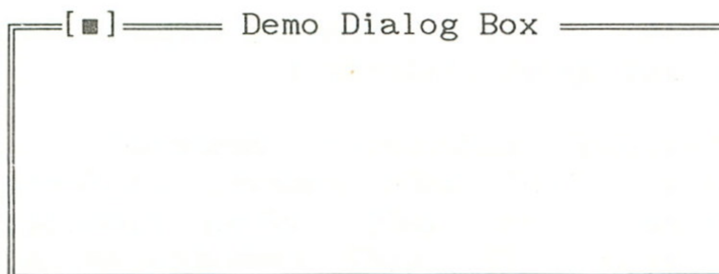
```
        Desktop^.Insert(Dialog);
    end;

    procedure TMyApp.HandleEvent(var Event: TEvent);
    begin
        TApplication.HandleEvent(Event);
        if Event.What = evCommand then
        begin
            case Event.Command of
              cmNewWin    : NewWindow;
              cmNewDialog: NewDialog;
            else
              Exit;
            end;
        ClearEvent(Event);
        end;
    end;
```

A dialógus doboz és a korábbi ablakok közötti eltérések az alábbiak:

- A dialógus doboz színe alapértelmezésben szürke (a kék helyett),
- A dialógus doboz alakja nem változtatható és nem nagyítható,
- A dialógus doboznak nincs ablaksorszáma.

A dialógus dobozt lezárhatjuk a záró ikonon klikkentve, vagy *ALT-F3* illetve az *ESC* billentyű megnyomásával. Az alapértelmezés szerint az *ESC* billentyű törli a dialógus dobozt.

A következő ábra egy egyszerű dialógus dobozt mutat be.



A dialógus dobozok gyakran opció állításra szolgálnak, amely azt jelenti, hogy egy művelet módot definiálnak.

6.2.1. Opció beállító dialógus doboz

A opció beállító dialógus doboz létrehozása nagyon egyszerűen történik, a dialógus doboz objektumot az *insert* helyett a

DeskTop^.ExecView függvény hívásával tesszük a *desktop*-ra:

A TVGUID12.PAS programból a megfelelő részlet:

```
procedure TMyApp.NewDialog;
var
     Dialog: PDialog;
     R: TRect;
     Control: Word;
begin
     R.Assign(0, 0, 40, 13);
     R.Move(Random(39), Random(10));
     Dialog:=New(PDialog, Init(R,'Demo Dialog'));
     Control:= DeskTop^.ExecView(Dialog);
end;
```

A *TDialog* már tudja, hogy kell válaszolni az *ESC* billentyűre (végrehajt egy *cmCancel* parancsot) és az *ENTER* billentyűre (a dialógus doboz alapértelmezés szerinti *TButton*-ja lesz kezelve). A dialógus doboz mindig a *cmdCancel* parancs hatására záródik le.

Az *ExecView* hívás hatására a dialógus doboz bekerül az objektum csoportba és megjelenik az opció beállító dialógus doboz. A dialógus doboz zárása vagy törlése után a *ExecView* kiveszi a csoportból a dialógus dobozt és befejezi a működését. Az *ExecView* függvény által visszaadott értéket tárolhatjuk a *Control* változóban. Ennek az értéknek a felhasználását lásd a TVGUID16.PAS programban.

6.2.2. Vezérlés fogadása

Dialógus doboz készítéséhez hozzátartozik a vezérlés megszervezése. A dialógus dobozban különféle elemek vannak, amelyek információk fogadására szolgálnak, azonban fontos dolog gondoskodni a vezérlésről.

Általában, amikor a dialógus dobozban való vezérlést tervezzük, el kell választanunk a látható képet az adat kezeléstől. Ez azt jelenti, hogy könnyebben tudunk teljes dialógus dobozt tervezni anélkül, hogy programot írtunk volna, aminek éppen a feldata, hogy felépítse és használja a dialógus doboz nyújtotta adatokat.

A legegyszerűbb vezérlő objektum a *TButton.* A státuszsor pontjaihoz nagyon hasonlóan működik: színes tartomány, rajta szöveg és ha klikkentünk rajta, akkor Turbo Vision parancs generálódik. A vezérlő gomb mögött árnyék van, így ha klikkentünk rajta a gomb háromdimenziós hatású, a gomb úgy néz ki mintha megnyomtuk volna.

A legtöbb dialógus doboznak egy,vagy két vezérlő gombja van. A legáltalánosabban használt gombok: az "**OK**" (jelentése: rendben, lezárja

a dialógus dobozt és a beállítást elfogadja) és a "**Cancel**" (jelentése: törlés, lezárja a dialógus dobozt a beállításokat nem veszi figyelembe). A *Cancel* gomb gyakran ugyanazt *cmCancel* parancsot generálja, mint a záró ikon.

A *Dialogs* unit 5 fajta általános dialógus parancsot definiál, amelyek a *TButton*-hoz köthetők:

cmOk, cmCancel, cmYes, cnNo és a *cmDefault*.

Az első négy parancs le is zárja a dialógus dobozt azáltal, hogy a *TDialog* meghívja a saját *EndModal* metódusát, amely újratölti a módbeállító státust az előző módbeállító *view*-val.

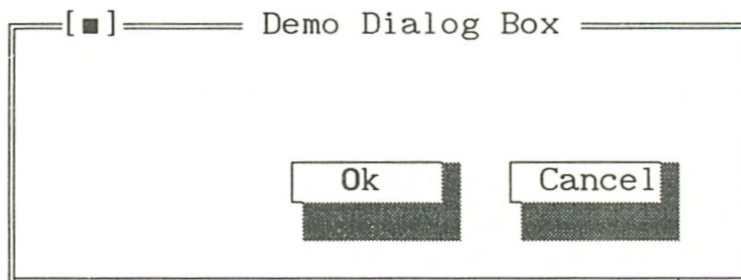
A TVGUID13.PAS programból az alábbi programrészlet a vezérlő

gombok használatát és a parancsok generálását mutatja be:

```
procedure TMyApp.NewDialog;
var
     Dialog: PDialog;
     R: Rect;
     Control: Word;
begin
     R.Assign(20, 6, 60, 19);
     Dialog :=New(PDialog, Init(R, 'Demo Dialog'));
     with Dialog^ do
     begin
          R.Assign(15, 10, 25, 12);
          Insert(New(PButton, Init(R, '~O~K', cmOk, bfDefault)));
          R.Assign(28, 10, 38, 12);
          Insert(New(PButton, Init(R, 'Cancel', cmCancel,
          bfNormal)));
     end;
Control:= DeskTop^.ExecView(Dialog);
end;
```

Az *Init* konstruktornak egy gomb működtetéséhez 4 paramétert kell létrehoznia:

1. A tartomány, amit a gomb lefed,
2. A szöveg, amely a gombon megjelenik,
3. A parancs, amely a gombhoz kapcsolódik,
4. Flag, amely a gomb típusát jelzi (normál vagy az alapértelmezés).



A "*Cancel*" szövegben a "C" nem világít nagyobb fényerővel, mivel a dialógus doboz törlésére van már *hot key*, az *ESC* billentyű. Igy a "C" felhasználható más rövid parancs kiválasztására.

Amikor létrehozunk egy vezérlő gombot, akkor egy flag-et (jelzést) rendelünk hozzá. Ennek értéke vagy *bfNormal* vagy *bfDefault* lehet. A legtöbb vezérlő gombnál a *bfNormal*-t használjuk. A *bfDefault* jelzés a default vezérlő gombot jelenti. Ez akkor lesz "megnyomva", ha az *ENTER* billentyűt nyomjuk meg. A Turbo Vision nem ellenőrzi, hogy mennyi default gombot használtunk. Természetesen ha többször használtuk fel a *bfDefault* jelzést, akkor az vezérlés eredménye megjósolhatatlan.

Gyakran az "OK" gombnál használunk *bfDefault* jelzést, mivel az információ beállítása után adott *ENTER* lezárja a dialógus dobozt és elfogadja a változtatásokat.

Ha a dialógus doboz megjelenik, akkor a vezérlések egyike magasabb fényerővel világít, aktív. Ez akkor fontos, ha a kiválasztás a klaviatúráról történik. Ha például egy gomb van kijelölve, akkor ezt a gombot "megnyomhatjuk", ha leütjük a *SPACE* billentyűt. Karaktereket is csak akkor gépelhetünk be egy input sorba, ha az input sor van kijelölve a dialógus dobozban. A *TAB* billentyűvel térhetünk át a dialógus doboz egy másik logikailag külön elhelyezett információs részére, végül a vezérlő gombokra. Körkörösen többször végigmehetünk az információs blokkokon. Ezt a mozgatást a dialógus doboz *SelectNext* metódus hívásával érhetjük el. Előre a *SelectNext(FALSE)*, hátra a *SelectNext(TRUE)* utasítással mozoghatunk a körkörös listán.

A Dialógus dobozban kétfajta objektumot használhatunk egyes opció flag-ek módosítására. Ezek az objektumok a következők:

- opció beállító doboz (*check box*),
- állító gomb (*radio button*).

Az opció beállító dobozok és az állító gombok funkciója majdnem azonos, kivéve, hogy a beállító dobozok közül egyszerre több dobozban is "bekapcsolhatjuk" az egyes opciókat, míg az állító gombok közül csak egy lehetséges választható ki. Ezen okból a jelölés kétfajta, azonban a viselkedésük hasonló; mindkettő a Turbo Vision *TCluster* objektumából származik.

Módosítani kell a *TMyApp.NewDialog* metódust a dialógus doboz információi miatt.

```
[ ] Angol
[ ] Német
[ ] Francia
```

A programrészlet az alábbi:

```
var
      B: PView;
R.Assign(3, 3, 18, 6);
B:= New(PCheckBoxes, Init(R,
      NewSItem('~A~ngol',
      NewSItem('~N~émet',
      NewSItem('~F~rancia',
      nil)))
   ));
Insert(B);
```

A kezdeti értékadás nagyon egyszerűen történik. Kijelöljük a téglalap területét, amely körülfogja az információt, azután létrehozzuk a sztringre mutató pointernek egy lácolt listáját, amelyet **nil**-el terminálunk. Az opció beállító dobozban lévő menük száma maximálisan 16 lehet.

Most bővítsük a dialógus dobozt állító gombokkal, legyen három választási lehetőség:

```
(•) Kezdő
( ) Haladó
( ) Perfekt
```

A programrészlet az alábbi:

```
R.Assign(, , , );
B:=New(PRadioButtons, Init(R,
     NewSItem('~K~ezdő',
     NewSItem('~H~aladó',
     NewSItem('~P~erfekt',
     nil)))
));
Insert(B);
```

Egy csopotból csak egy állító gomb választható ki. Alapértelmezés szerint a csoportból állító gombok közül az első van kiválasztva. A állító gombok száma szinte nincs korlátozva, mivel csoportonként 65536 lehetséges.

Opció módosítási csoportok címkézése

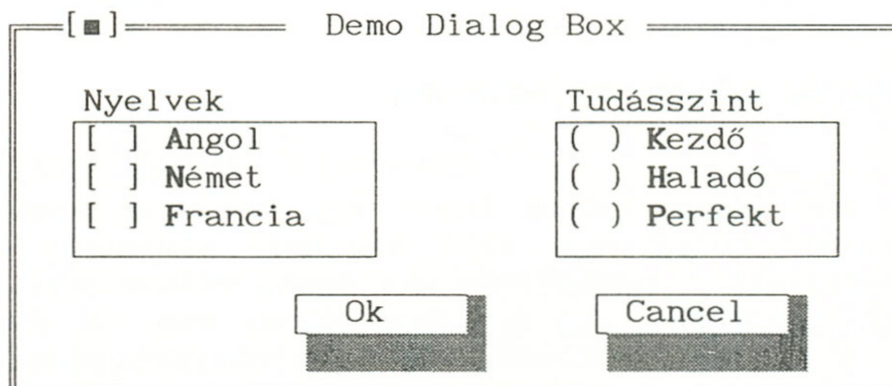
A opció módosítások csoportjaihoz cimkét is lehet rendelni, ezt a *TLabel* látja el, amely nemcsak hogy megjeleníti a szöveget, hanem összeköti egy másik *view*-hoz. A cimkén lehet klikkenteni.

Az opció beállító doboz cimkéjének megadására vonatkozó utasítás a következő:

```
R.Assign(2, 2, 10, 3);
Insert(New(PLabel, Init(R, 'Nyelvek', B)));
```

Hasonlóan kell megadni az állító gombokra vonatkozó cimkét:

```
R.Assign(21, 2, 33, 3);
Insert(New(Plabel, Init(R, 'Tudásszint', B)));
```



6.2.3. Input sor objektum

A dialógus dobozban lehetőség van sztring beolvasására, az ún. input sorban, a *TInputLine* objektum felhasználásával.

Bővítsük tovább a programot:

```
R.Assign(3, 8, 37, 9);
B:= New(PInputLine, Init(R, 128));
Insert(B);
```

```
R.Assign(2, 7, 24, 8);
Insert(New(PLabel, Init(R, 'Hol tanulta a nyelvet', B)));
```

Az input sor felépítése a következő: a téglalaphoz rendeljük az input sor hosszát a képernyőn belül. Az editálandó sztring hosszát nem kell definiálni, a sztring hossza túlnyúlhat a jelzett téglalapon, mivel a *TInputLine* objektum tudja görgetni a sztringet. Az input sor objektum kezeli a billentyűleütéseket, az editáló parancsokat és az egér klikkentését.

Az input sor szintén cimkézhető, így egyértelmű lesz a használata.

```
[■]═══════ Demo Dialog Box ════════════

 Nyelvek                Tudásszint
 [ ] Angol              ( ) Kezdő
 [ ] Német              ( ) Haladó
 [ ] Francia            ( ) Perfekt

 Hol tanulta a nyelvet
 [                                   ]

           [   Ok   ]     [ Cancel ]
```

Kezdeti értékadás és adatok változtatása

Ezek után már létre tudunk hozni egy teljesen komplex dialógus dobozt. Két alapvető dolog van, amit meg kell oldanunk: beállítani a vezérlés kezdőértékeit, amikor a dialógus doboz nyitva van, és visszaolvasni az értékeket, ha a doboz zárva van. A dialógus doboz törlése esetén, a dialógus dobozban történt változtatásokat figyelmen kívül kell hagyni.

A *SetData* és a *GetData* metódus szolgál arra, hogy adatokat másoljon a képernyőre és vissza. Minden *view*-nak van egy *SetData*, és egy *DataSize* metódusa. A *DataSise* metódus megadja a *view* adat területének a méretét és egy pointert, amely a következő *view* elejére mutat. Ha az alsó szintű *view* rossz méretű, akkor az összes utána következő rossz adattal fog kommunikálni. Minden *view*-nál, ha új *view*-t adat mezőkkel hozunk létre, ne felejtsük el felülírni a *DataSize* értékét, *SetData* és a *GetData* hívásokat, azért hogy azok a saját adataikat kezeljék.

Az adatokat beállíthatjuk a dialógus doboz adatrekordjában globális típus deklarációból:

```
DialogData = record
     CheckBoxData: Word;
     RadioButtonData: Word;
     InputLineData: string(128);
end;
```

A rekordban lévő adatokat a program elején, a *MyApp.Init*-ben érdemes beállítani, amikor belépünk a dialógus dobozba. Az adatokat visszaolvasni akkor érdemes, amikor a dialógus doboz sikeresen lezárult.

A dialógus doboz adatai:

Vezérlés	Adat típus
opció beállító doboz	-
cimke	-
vezérlő gomb	Word
cimke	-
input sor	sztrig[128]
cimke	-
Gomb	-
Gomb	-

Az erre vonatkozó részlet az alábbi:

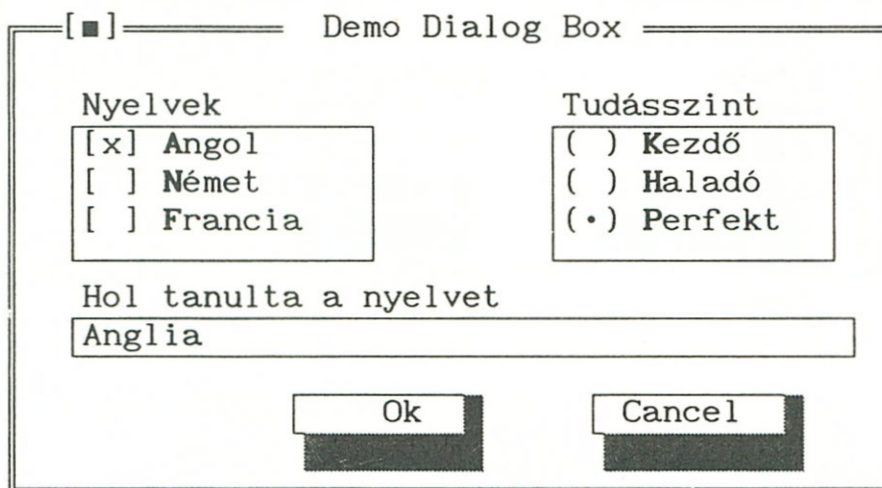
```
var
     DemoDialogData : DialogData;

     Dialog^.SetData(DemoDialogData);
     Control := DeskTop^.ExecView(Dialog);
     if Control <> cmCancel then
                         Dialog^.GetData(DemoDialogData)
     with  DemoDalogData do
          begin
               CheckboxData := 1;
               RadioButtonData := 2;
               InputLineData := 'Anglia';
          end;
```

A cimkék, az opció beállító doboz és a állító gombok egyetlen betű leütésére is működnek. Ez azt jelenti, hogy kiemelten, vastagon íródott, mint például az **P** betű a **P**erfekt állító gombon. Legjobb úgy tervezni, hogy a betű lehetőleg a kezdőbetű legyen, ha ez nem sikerül, akkor azt érdemes átfogalmazni más találó szövegre.

Ha lokalizálni akarjuk az dialógus dobozban a betűleütésre történő vezérlést, akkor a betűt két ~ (hullám) között kell megadni:

```
R.Assign(3, 3, 18, 6);
B := New(PCheckBoxes, Init(R,
     NewSItem('~A~ngol',
     NewSItem('~N~émet',
     NewSItem('~F~rancia',
     nil)))
));
B^.Option :=B^.Option and not ofPostProcess;
Insert(B);
```

A beállított értékek megjelennek a dialógus dobozban:



Ebben az esetben a *A*, *N* és *F* lesz a *hot key*, amely akkor működik, ha egérrel klikkentünk vagy a *TAB* billentyűvel a "Nyelvek" cimkére állunk, ezután az *ALT-A*, *ALT*-N és az *ALT*-F úgy működik, mint ahogy az előzőekben leírtuk.

Meg kell jegyeznünk, hogy a cimkét előre nem jelöljük ki, mivel a cimkének az ún. *ofPostProcess* bitje be van állítva a *hot key* működtetésére. A dialógus dobozokat gondosan kell megtervezni, hogy ne legyen konfliktus a *hot key*-k között.

A *Dispose*(*D*, *Done*) meghívásával a dialógus doboz eltünik a *desktop*-ról. Normál esetben nem kell hívni a *Done*-t, mivel a hívása a dialógus doboz lezárásával automatikusan megtörténik.

6.2.4. Statikus szöveg kiírása

A *TStaticText* is egy *view*, amely egyszerűen megjelenít egy sztringet a *view* téglalapján belül. A szöveg középre kerül, ha a sztring *CTRL-C*-vel kezdődik és befejeződik a *CTRL-M* után. A szöveg előre nem jelölhető ki, és természetesen az objektum az karakter adatrekordból nem vesz adatot.

A *TListViewer* egyszeres vagy többszörös oszloplistát jelenít meg, amelyből választhatunk menüpontot. A *ListViewer* két scroll görgető négyzet, amely is kapcsolatot tud teremteni.

A *GetText* metódus hívja be a lista tagjait a *Draw* metódus számára. A *TListViewer* leszármazottjának gondoskodnia kell, hogy a *GetText* az aktuális adatot tudja betölteni.

6.2.5. Lista doboz

A *TListBox* a *TListViewer* leszármazottjaként működik. Tartalmazza a *TCollection* sztringre mutató pointert. A *TListBox* csak egy görgető négyzetet támogat. A lista dobozra példa a file kiválasztása a Turbo Pascal Integrált környezetében. A lista doboz adatainak leolvasása és beállítása igen egyszerű a *TListBoxRec* rekord adatainak használatával, amely a megjelenített sztrigre mutató pointert tartalmazza, és jelzi, hogy melyik sor lett kiválasztva.

6.2.6. History

A *THistory* megőrzi azt az objektumot, amely együttműködik az input sorral és kapcsolatban van a lista dobozzal. A kiválasztott sor megörződik, így az ismételt gépelést megtakaríthatjuk. *THistory* objektumokat használ a Turbo Pascal Integrált környezete például a **File|Open** dobozban és a **Search|Find** dialógus dobozban.

6.2.7. Standard dialógus dobozok

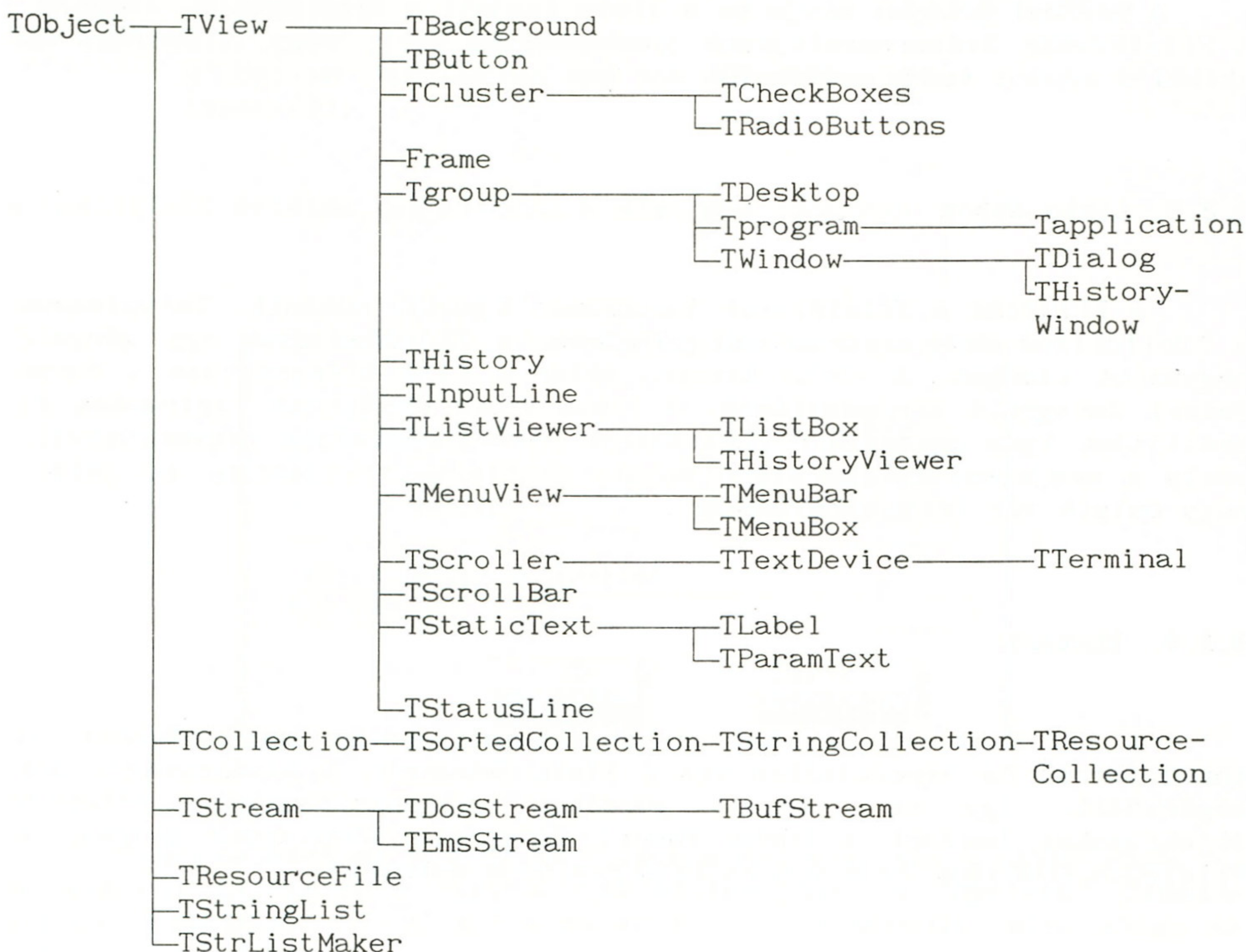
Az *StdDlg* unit tartalmaz egy beépített dialógust, a *TFileDialog*-t. Ezt a dialógust használjuk az integrált környezetben, amikor egy file-t nyitunk meg. A *TFileDialog* további objektumokat használ, *StdDlg* unit-ban, amelyek nagyon hasznosak:

```
TFileInputLine =  object(TInputLine);
TFileCollection = object(TSortedCollection);
TSortedListBox =  object(TListBox);
TFileList = object(TSortedListBox);
TFileInfoPane = object(TView);
```

6.3. Objektum hierarchia

Az előbbi fejezetekben már áttekintést kaptunk a Turbo Vision filozófiájáról, lehetőségeiről és terminológiájáról. Ebben a fejezetben megismerkedünk a Turbo Vision objektumok hierarchiájával, az objektumok öröklési kapcsolataival.

A objektumok teljes fastruktúráját a 6.5. ábra mutatja.



6.5. Objektumok hierarchiája

Példaként nézzük meg, hogy a *TDialog* honnan származik:

TWiew → *TGroup* → *TWindow* → *TDialog*

belátható, hogy a *TWiew*-ból ered a *TGroup*, a *TWindow* a *TGroup* leszármazottja és a *TDialog* a *TWindow*-ból származik. Az elemek vonallal való összekötése megkönnyíti a 6.5. ábra tanulmányozását. Minden új, származtatott objektum típus örökli a szülő tulajdonságait. Ezen

tulajdonságokat akkor kell részletesen tanulmányoznunk, amikor újabb mezőkkel és tulajdonságokkal akarjuk bővíteni az objektumot, vagy ha felülírjuk az öröklötteket.

6.3.1. Objektumok típusai

Különféle objektum típusokat tartalmaz a Turbo Vision. Funckiójuk szerint három csoportba oszthatjuk:

- primitiv objektumok,
- látvány (view) és a
- néma (mute) objektumok.

Ezek mindegyikében különböző típusú objektumok találhatók. Vannak olyan objektum - típusok, melyek azonnal felhasználhatók, mint például a *TWindow*, és vannak ún. absztrakt objektumok, amelyek közvetlen felhasználása nem sok értelmes eredménnyel jár. Az absztrakt objektumok olyan "csontvázak" a Turbo Vision-ban, amelyek jól alkalmazható keretül szolgálnak a felhasználó számára közvetlenül is hasznosítható osztályok definiálására.

Absztrakt objektumok

Sokfajta ún. absztrakt objektum típus létezik, amelyből több speciális és közvetlenül hasznos objektum származtatható. Az absztrakt típusok részben fogalmi okból, de jórészt abból a praktikus célból léteznek, hogy csökkentsék a programírási nehézségeket.

Tekintsük például a *TRadioButton* és *TCheckBox* típusokat. Ezek minden nehézség nélkül közvetlenül származtathatók lennének a *TView* objektumból. Közös tulajdonságuk: mind a kettő hasonló válasszal jelzi a vezérlések elemeit. Az állító gombok (radio button) és az opció beállító dobozok (check box) között sok a hasonlóság belül, mikor csak egy doboz jelölhető meg, habár van néhány technikai különbség is. Célszerűen tehát a *TRadioButton* és a *TCheckBox* a *TCluster* absztrakt típus leszármazottja, amely rendelkezik néhány speciális metódussal, amely támogatja az opció beállító/opcióválasztó funciókat. A példaként kiválasztott két típus a *TCluster* absztrakt objektum tulajdonságát oly módon egészíti ki, hogy az opció beállító típusokra vagy állító gombokra jellemző működést kapjunk.

Az absztrakt típusok sohasem származtathatók hasznosan. Legyen a *MyCluster* a *TCluster* példánya, ebben az esetben például nem lenne hasznos a *Draw* metódus, mivel a *TView.Draw*-t örökli felülírás nélkül, így a *MyCluster.Draw* egyszerűen egy üres téglalapot jelenít meg default színnel. Ha egy elképzelt vezérlési csoportot akarunk tervezni, amely

különböző tulajdonságú beállító gombokból és opció beállító dobozokból áll, akkor ennek megvalósítására a *TCluster*-ből származtathatjuk a *TMyCluster*-t, vagy származtathatjuk könnyebben a *TRadioButtons* vagy *TCheckBoxes* objektumokból. Minden esetben bővítjük a mezőket újabb mezőkkel, és a metódusokat újabb metódusokkal vagy felülírjuk a metódusokat a lehetőleg a legminimálisabb erőfeszítéssel.

Absztrakt metódusok

Hogy egy objektum típusból hasznos példányokat tudunk-e létrehozni, az teljesen a körülményektől függ. A legtöbb Turbo Vision standard típusnak vannak absztrakt metódusai, amelyeket a származtatott típusokban definiálni kell. A standard típusoknak lehetnek pseudo-absztrakt metódusai, amelyek tartalmaznak ugyan valami minimális, alapértelmezés szerinti műveletet, ami lehet, hogy megfelel a saját célnak, de ha nem, akkor származtatott típusként kell a megfelelő műveleteket megvalósítani.

Ha végigmegyünk a Turbo Vision objektum hierarchiáján, egyre inkább azt találjuk, hogy a sztandard típusok inkább valamire specializáltak, mint absztraktak. A neveik szinte elárulják a mezőikbe és metódusaikba foglalt tevékenységüket. A legtöbb alkalmazás esetén lesznek olyan alap osztályok, amelyekből egy sztandard felhasználói felületet (interface) készíthetünk, tehát csak ami egy *desktop*-ból, memükből, státuszsorból, dialógus dobozokból és a opciót beállító dobozokból stb. áll.

6.3.2. Objektum - példányok létrehozása, származtatott típusok

Bármilyen objektum típus esetén két alapvető művelet végezhető: létrehozható a típus példánya, vagy származtatható az utód objektum típus. Későbbiek folyamán új objektumot is létrehozhatunk, és ugyanezeket a műveleteket is elvégezhetjük rajta.

Példány

Egy objektum példányát létrehozhatjuk változó deklarálásával:

```
MyScrollBAr : TScrollBar;
```

A *MyScrollBar*-nak a *TScrollbar.Init* ad kezdőértéket. A *TScrollBar* a *TView* leszármazottja, a *TScrollBar.Init* hívja a *TView.Init*-et, amely a *TView*-tól örökölt mezőknek ad kezdőértéket. Hasonlóan a *TView.Init* a

TObject leszármazottja, így hívja a *TOBject* konstruktorát, amely gondoskodik a memória helyfoglalásáról. A *TObject* viszont már szülő, a hívási hierarchia nála véget ér.

A *MyScrollBar* objektumnak van alapértelmezés szerinti mezeje, amelyet nekünk valószínüleg változtatni kell. Ahhoz, hogy használni tudjuk a *MyScrollBar* -t, ismernünk kell, hogy mit csinálnak a metódusai pl. a *HandleEvent* és a *Draw*. Ha a számunkra szükséges funkció nincs benne definiálva, akkor egy új típust kell származtatnunk.

Származtatás

Könnyen tudunk egy létező objektumból egy új objektumot származtatni:

```
PNewScrollBar = ^TNewScrollBar;
TNewScrollBar = object(TScrollBar);
                end;
```

Mielőtt deklarálunk bármilyen *TNewScrollBar* objektumot, definiálnunk kell új metódusokat, vagy felül kell írni néhány *TScrollBar* metódust, valószínüleg új mezőkkel kell kiegészítenünk, máskülönben nincs meg az alapja, hogy új görgető sor (scroll bar) objektumot hozzunk létre.

6.3.3. A Turbo Vision metódusai

Turbo Vision négyfajta metódust tartalmaz:

Absztrakt metódusok

Az absztrakt metódus üres, nincs törzse, egyszerűen csak kivédi az illegális hívásokat. Az absztrakt metódusokat definiálni kell egy származtatott típusban, mielőtt használhatnánk. Az absztrakt metódusok mindig virtuálisak, mint pl. a *TStream.Read.*

Pszeudo absztrakt metódusok

Az alap objektum típusban egy pszeudó - absztrakt metódus valamilyen minimális tevékenységet tartalmaz. Majdnem mindig felül kell írnunk a származtatott típusban, hogy hasznosan tudjuk felhasználni, bár az öröklődési láncban egy pszeudo - absztrakt metódus mindig

végrehajt valamilyen default tevékenységet. Ilyen pl. a *TSortedCollection.Compare* metódus.

Virtuális metódusok

A virtuális metódusok a prototípus deklarációjukban a **virtual** direktívát használják. A virtuális metódusok újradefiniálhatók, azonban az új metódusoknak is virtuálisaknak kell lenniök és pontosan illeszkedniök kell az eredeti metódus fejlécéhez.

Ezeket a metódusokat nem szükséges felülírni, de előbb vagy utóbb úgyis felülírjuk őket. Ennek egy példája a *TView.DataSize.*

Statikus metódusok

A statikus metódusokat nem írhatjuk felül. A leszármazott típust definiálhatjuk ugyanazzal a névvel, de esetleg más visszatérési típussal vagy paraméterlistával, ha szükséges, de a statikus metódusok nem működnek sokalakúan. Ez akkor kritikus, amikor egy dinamikus objektumnak aktíváljuk a metódusait.

6.3.4. Turbo Vision mezői

Példaként nézzük meg a Turbo Vision három jelentős objetumának, a *TView*-nak, *TGroup*-nak és a *TWindow*-nak a mezőit. Így egyszerűen érthetővé válik az öröklődés, egyre jobban érthetővé válik az ojektumok funkcióinak kiterjedése a hierchiában előre haladva.

Tanulmányozzuk a 6.1. táblázatot! Ekkor a következőket tapasztaljuk:

A *TGroup* minden mezőt örököl a *Tview*-tól, ugyanakkor olyan mezőket ad hozzá a struktúrához, amelyek a csoport működéséhez nélkülözhetetlenek. Ilyen pl. a csoportban az aktuális és az utolsó *view*-ra mutató pointer. A *TWindow* mindent örököl *TGroup*-tól, és tartalmazza még olyan mezőket, amelyek szükségesek az ablak - műveletekhez. Ilyen pl. az ablak fejléce, az ablak sorszáma. Ezekután a *TWindow*-ról is beláthatjuk, hogy egy csoportot alkot és *view* típusú.

A *View* mezőinek öröklődése 6.1. táblázat

TView mezők	TGroup mezők	TWindow mezők
Owner	Owner	Owner
Next	Next	Next
Origin	Origin	Origin
Size	Size	Size
Cursor	Cursor	Cursor
GrowMode	GrowMode	GrowMode
DragMode	DragMode	DragMode
HelpCtx	HelpCtx	HelpCtx
State	State	State
Options	Options	Options
EventMask	EventMask	EventMask
	Buffer	Buffer
	Phsase	Phsase
	Current	Current
	Last	Last
		Flags
		Title
		Number
		ZoomRect
		Palette
		Frame

6.3.5. Primitiv objektum típusok

A Turbo Vision három egyszerű objektum típust tartalmaz, amelyeket elsődlegesen más objektumok használnak, vagy több összetett objektum hierarchiájának alapját képezik. Az összes látható objektum használja a *TPoint* és a *TRect* objektumokkal, a *TObject* pedig az hierarchia alapját képezi.

Ezek az objektum típusok közvetlenül nem jeleníthetők meg. A *TPoint* egyszerűen egy képernyő - pozíciót leíró objektum (x, y koordináták). A *TRect* úgy tünik, mintha egy *view* objektum lenne, de csak a téglalap bal felső és jobb alsó koordinátapontját tartalmazza és kapcsolatban van más, nem megjelenítést szolgáló utility metódusokkal.

TPoint

Ez az objektum egy pontot reprezentál, X és Y a mezői, definiálják a képernyő (X,Y) koordinátájú pontját. A (0,0) pont a képernyő bal felső sarka. A *TPoint*-nak nincs metódusa, más típusokban vannak olyan metódusok, amelyek elvégzik a konverziót a globális, a teljes képernyőre vonatkozó és a lokális, egy adott *view*-ra vonatkozó

koordináták között.

TRect

Ez az objektum egy téglalapot reprezentál. A *TPoint* objektum *A* és *B* mezői definiálják a téglalap bal-felső és jobb-alsó sarkát. A *TRect* objektum metódusai:

Assign, Copy, Move, Grow, Intersect, Union, Containsa, Equal, Empty.

A *TRect* típusú objektumok nem látható objektumok, nem tudják magukat megjeleníteni. Azonban minden *View* objektum téglalap alakú, az *Init* konstruktorai a *TRect* típusú a *Bounds* paraméterből veszik annak a tartománynak a határait, amelyet le kell fedniök.

TObject

A *TObject* mező nélküli absztrakt alaptípus. Az összes Turbo Vision objektum őse, kivéve a *TPoint* és a *TRect* objektumokat. A *TObject*-ben három metódust van: *Init, Free* és a *Done.*

Az *Init* az összes Turbo Vision konstruktorok alapját képezi a memória helyfoglalás tekintetében. A *Free* felszabadítja az *Init* által lefoglalt memóriát. A *Done* egy absztrakt destruktor, amelyet a leszármazottaknak felül kell írniuk. Bármely objektumot, amelyet a Turbo Vision stream-jeivel együtt akarunk használni, végülis a *TObject*-ből kell származtatnunk.

A *TObject* objektum származékai két csoportba sorolhatók: *view* és nem *view* típusúakra. A *view* típusúak a *TView*-ból származnak, meg tudják jeleníteni önmagukat és kezelik a hozzájuk irányított eseményeket. A nem *view* típusú objektumok viszont a stream-ek kezelésére szolgáltatnak közhasznú rutinokat és másfajta objektumok gyűjteményét, beleértve a *view*-kat is, amelyek viszont közvetlenül nem láthatók.

6.3.6. *View* (látvány)

A *TObject* objektum látható leszármazottait nevezzük *view*-nak és a *TView*-ból származnak, amely a *TObject* közvetlen leszármazottja. Meg kell különböztetnünk azt, hogy valami "látható" vagy "megjeleníthető", mivelhogy lehet olyan eset, mikor a *view* teljesen vagy részlegesen takart más *view*-k miatt.

A *view* egy olyan objektum, amely megjeleníthető a képernyő téglalap alakú tartományán. Csak *TView* leszármazottja lehet. A *TView* sajátmaga egy absztrakt objektum, amely a képernyőn egy üres téglalap alakú tartományt képvisel és egy minimális virtuális *Draw* metódust tartalmaz.

A Turbo Vision felhasználásával készült programoknál egyre több *TView* leszármazottat fogunk felhasználni, mivel a *TView* működőképessége a teljes Turbo Visiont betöltik, így nagyon fontos a velük kapcsolatos lehetőségek megértése.

6.3.6.1. Csoportok

Minden, ami látható a Turbo Vision alkalmazásban az a *TView*-ból származik. Közülük néhány objektum azon okból jelentős, hogy több más objektum számára lehetővé teszik az együttműködést.

Absztrakt csoport

A *TGroup* által dinamikusan kezelhetjük a láncolt listákat, interaktív almegjelenítéseket a megjelölt *view*-n keresztül, amelyet a csoport *Owner*-nek (tulajdonosának) neveznek. Minden *view*-nak van *PView* típusú *Owner* mezeje, amely mutat saját *TGroup* objektumra, ha az **nil**, akkor a *view*-nak nincs sajátja. A *Next* mező mutatja a kapcsolatot a következő *view*-ra a láncban. A program futtatása közben a lánc tartalma változik, új csoportok jönnek létre, megjelenítések csatlakoznak a listához, ill. törlődnek a listából.

Desktop

A *TDesktop* normál háttér *view*, gyakran körbe van véve menüvel és státuszsorral. A *TApplication* lesz annak a csoportnak a gazdája, amely tartalmazza a *TDesktop, TmenuBar* és a *TStatusLine* objektumokat. Más objektumok is találhatók a *desktop*-on, pl. dialógus dobozok stb. Egy alkalmazói programban a legtöbb munka a *desktop* keretein belül történik.

Program

A *TProgram* a virtuális metódusok gyűjteményét tartalmazza a saját leszármazottja, a *TApplication* számára.

Alkalmazói program

A *TApplication* olyan objektum. amely sablonként használható Turbo Vision alkalmazói programok számára. A *TApplication* a *TGroup* leszármazottja (a *TProgram*-on keresztül); rendelkezik saját *TMenuBar, TDesktop* és *TStatusLine* *subview*-kal. A *TApplication* metódusai hozzák létre és helyezik el ezt a három *subview*-t. A *TApplication* legfontosabb metódusa a *TApplication.Run*, amely végrehajtja a programot.

Window

A *TWindow* objektumok a help-pel a népszerű, keretes téglalap alakú látványok, melyeket mozgathatunk, újra méretezhetünk és eltüntethetünk a *TView*-ból származó metódusok segítségével. A *TWindow* objektum saját metódusaival nagyítható és lezárható. Kezeli a *TAB* és a *SHIFT TAB* billentyű metódusokat, amellyel az előző és a következő *subview* választható ki az ablakból. A sorszámozott ablakokat az *ALT-n hot key* segítségével is kiválaszthatjuk.

Dialógus doboz

A *TDialog* a *TWindow* leszármazottja, amelyet a felhasználó különböző interaktív tevékenységeinek kezelése érdekében hozzuk létre. A dialógus dobozok tartalmaznak vezérlő gombokat és opció beállító dobozokat. Az *ExecView* metódus megőrzi az előzőleg kiválasztott opciókat, elhelyezi a *TDialog* objektumot a csoportba és elkészíti az opció beállító dialógus dobozt. Ezután már a dialógus doboz kezeli a felhasználó által generált eseményeket, ilyen pl. egérrel való klikkentés vagy egy gomb leütése. Az *ESC* billentyű leütése megegyezés szerint *exit* (*cmCancel* = kilépés). Az *ENTER* leütése a default válasz kiválasztását jelenti (*cmDefault*).

6.3.6.2. Terminal view-ek

Azokat a *view*-kat nevezzük terminál *view*-nak, amelyek nem tartoznak a csoporthoz, azaz nem lehetnek *view*-k tulajdonosai. Tulajdonképpen a *view* láncok végét jelzik.

Keretek

A *TFrame* a *TWindow* objektumok szegéllyel és a mozgatásra szolgáló valamint ablak lezáró ikonnal ellátott kereteit szolgáltatja.

Vezérlő gombok

A *TButton* objektum egy cimkézett doboz, amely "megnyomáskor" egy különleges parancs eseményt generál. Dialógus dobozban gyakran az "*OK*" illetve a "*CANCEL*" választására használjuk fel. A gombot "megnyomhatjuk" a rajta való klikkentéssel, vagy a kiemelt betűnek megfelelő billentyű leütésével. Az *ENTER* billentyű leütése az alapértelmezés választást hagyja jóvá.

Cluster

A *TCluster* absztrakt típus, opció beállító dobozok és állító gombok megvalósítására használjuk. A csoportba (*cluster*) gyűjtött vezérléseket azonosíthatjuk a *TLabel* objektummal, rendelhetünk hozzá mezőket, a *Value* mezővel adhatunk értéket, a *Sel* mező segítségével a csoportból vezérlést is választhatunk. A kurzor vagy egérklikkentés jelzi a csoportnak a vezérlést. Az állító gombok speciális csoportok, amelyből csak egy választható ki. Minden következő kiválasztása az előző kiválasztását megszünteti. Az opció beállító dobozok szintén csoportok, amelyek közül bármennyit kiválaszthatunk.

Menü

A *TMenuView* és leszármazottai, a *TMenuBar* és a *TmenuBox* alap objetumok, amelyek a pull-down menüket és almenüket tetszőlegesen egymásbaskatulyázva hozzák létre. A menükiválasztás számára meg kell adni a szöveget (magasabb fényerővel a rövid kiválasztás betűjét) a kiválasztáskor végrehajtott paranccsal együtt. A *HandleEvent*

metódusok pedig gondoskodnak az egér és/vagy a klaviatúráról (beleértve a rövid kiválasztást és a *hot key*-t) való menü kiválasztásról.

History

Az absztrakt típusú *THistory* generál egy listát a kiválasztott elemekből az ún. pick list mechanizmust. Két mezeje van: *Link* és a *HistoryID*, minden *THistory* objektumhoz hozzárendeli a *TINputLine* és az input sorban az előző belépők listájának az ID-jét. A *THistory* együttműködik a *THistoryWindow* és a *THistoryViewer* objektumokkal.

Input sor

Az *TInputLine* egy speciális *view*. Ez nem más. mint egy sztring - inputra szolgáló soreditor. Kezeli a klaviatúrát, a kurzor mozgatást (beleértve a *Home* és az *End* végrehajtást). Használható az editorban szokásos törlés és a beszúrás, kiválasztható a beszúrásos és a felülírásos mód.

Lista megjelenítők

A *TListViewer* objektum típus olyan absztrakt típus, amelyből különböző fajta lista megjelenítők származnak. Ilyen például a *TListBox*, amelyben gyakran a file nevek, mint sztring listák jelennek meg. A lista megjelenhet egy vagy több oszlopban függőlegesen működő görgető üzemmóddal. A lista elemeit kiválaszhatjuk egérrel klikkentve, vagy billentyűvel. A kiválasztott listaelem magasabb fényerővel világít. A *TListBox* mezőjében lévő *List* pointer a *TCollection* objektumra mutat, így a kiválasztott listaelem azonosítható.

Görgethető objektumok

A *TScroller* objektum egy görgethető *view*, amely más nagyobb ún. háttér *view* kapuját kezeli. Görgetve jelenik a válaszban a klaviatúra input vagy a *TScrollBar* objektumhoz rendelt akciók. A görgető objektumnak két mezeje van, a *HScroller* és a *VScroller*, amelyek azonosítják a hozzájuk tartozó vezérlő vizszintes és függőleges scroll téglalapot. A *TScroller* objektum *Delta* mezeje határozza az X és Y görgető egységek mértékét.

Szöveg

A *TTextDevice* görgethető TTY típusú szöveg megjelenítő ill. *device driver*. Eltekintve a *TScroller* örökölt mezőitől és metódusaitól, a *TTextDevice* virtuális metódusokat definiál sztringeket eszközről való olvasására ill. eszközre való írására. A *TTextDevice* a *TScroller* konstruktorát és desktruktorát használja.

Statikus szöveg

A *StaticText* objektumok egyszerű objektumok, amelyek a *Text* mezőben tárolt sztringet jelenítenek meg. Az eseményt nem veszik figyelembe. A *Label* típus tulajdonságot rendel a szöveget tartó *view*-hoz

Státusz sor

A *TStatusLine* objektum alkalmas a képernyő alsó sorában lévő különböző státuszok és egyéb help információk megjelenítésére. A státuszsor egy karakter magas, hossza maximálisan a képernyő széléig terjedhet. A státuszsor menüpontjai kiválaszthatók egér klikkentésével, vagy *hot key* megnyomásával. A legtöbb alkalmazói objektum használja a *TMenuBar*, a *TDeskTop* és a *TStatusLine* objektumokat.

Az *Items* a *TStatusItem* rekordok aktuális láncolt listájára mutat rá. Ezek a rekordok a megjelenítendő szöveget, a *hot key* táblát és a hozzájuk rendelt *Command* változót tárolják. A *Defs* pointer a *PStatusDef* rekordok láncolt listájára mutat. A *PStatusDef* rekordok határozzák meg a help szövegeket, így egy rövid magyarázó szövegeket is megjeleníthetünk. *TStatusLine* típusú objektumok a *TApplication.InitStatusLine* felhasználásával inicializálásban hozhatók létre.

6.3.7. A Turbo Vision nem látható elemei

A Turbo Vision nem látható objektumai a *TObject*-ből származtathatók.

Stream-ek

A stream egy általánosított objektum az input/output kezeléséhez. A *TStream* absztrakt objektumból származtathatók azok a többrétű objektumok, melyek metódusai alkalmasak a tárolóeszközök kezelésére.

A *TStream.Status* mezője a hozzáférési módot (only read, only write, read/write) jelzi, az *ErrorInfo* mező pedig az I/ hibákat. Hét virtuális metódust használhatunk: *Flush, GetPos, GetSize, Read, Seek, Truncate* és a *Write*. A speciális stream típusok használatánál ezeket általában felül kell írni.

A TStream-hez tartozó további metódusok: *CopyFrom, Error, ReadStr, Reset* és a *WriteStr*.

A különböző objektum típusokat regisztrálni kell, mielőtt felhasználásra kerülnének a stream-en. A Turbo Vision a szabványos objektumokat előre regisztrálja.

A DOS stream

A *TDosStream* egy speciáis utódja a *TStream*-nek, amely a DOS puffer nélküli adatávitelét valósítja meg. A *Handle* mezőn keresztül kapcsolódik a DOS file kezelőjéhez. Az *Init* konstruktor egy adott file-névvel és hozzáférési móddal létrehozhat egy DOS stream-et. A *TDosStream* átdefiniálja a *TStream* mindegyik absztrakt metódusát, kivéve a *Flush*-t, amely csak pufferelt stream-eknél használható.

A pufferelt stream-ek

A *TBufStream* a *TDosStream* pufferes megvalósítása. A *Buffer* és a *BufSize* mezők tartalmazzák a puffer helyét és méretét. A *BufPtr* és a *BufEnd* mezők jelölik a pufferben az aktuális pozíciót és a puffer végét. Az absztrakt *TStream.Flush* metódus feladata a puffer ürítése. A stream lezárása előtt a pufferben maradt adatok menthetők el vele.

Az EMS stream

Az EMS memória felhasználását teszi lehetővé a stream számára. Az új mezők az EMS azonosító, a lapok száma, a stream mérete, az aktuális pozíció a stream-en.

Erőforrások

Az erőforrás file a stream-ek egy speciális fajtája, ahol sztringekkel indexelt objektumokat tárolhatunk. Az erőforrás file-ban

lévő elemeket a *Get(Key)* metódussal érhetjük el, ahol paraméterként a sztring indexet kell megadni. A *Put* egy adott kulcshoz tárol egy objektumot, a *Flush* az összes változást kiírja, a *Delete* törli az adott kulcshoz tartozó objektumot, a *Count* visszaadja a stream-en lévő elemek számát, a *KeyAt* egy adott pozícióhoz tartozó sztring kulccsal tér vissza.

Kollekciók

A *TCollection* tetszőleges típusú objektumokból álló készlet feldolgozását teszi lehetővé. Az alapvető műveleteken (beszúrás, törlés stb.) kivül, több iterációs elven működő metódust is tartalmaz. Például a *ForEach* metódus segítségével feldolgozhatjuk a kollekció teljes tartalmát.

Rendezett kollekciók

A *TSortedCollection* kulcs szerint rendezett kollekció feldolgozását teszi lehetővé. A rendezettség kialakításához a virtuális, absztrakt *Compare* metódus ad segítséget. Az *Insert* metódussal egy új elemet illeszthetünk be a kollekcióba úgy, hogy a rendezettségi állapot továbbra is megmarad. Az elemek gyors megtalálását a bináris keresési elven működő *Search* metódus teszi lehetővé.

Sztring kollekciók

A *TStringCollection* a *TSortedCollection*-nek egy egyszerű változata, amely lehetővé teszi a Turbo Pascal sztringek rendezett kezelését. Az alfabetikus rendezettség biztosításához a *Compare* metódus átdefiniálható. A *PutItem* és a *GetItem* a sztringek írását illetve olvasását, a *FreeItem* a sztring törlését teszi lehetővé.

Erőforrás kollekciók

A *TResourceCollection* az index szerint rendezett erőforrás file kezelését biztosítja. Az erőforrás feldolgozásához a *TStringCollection* metódusainak *(FreeItem, GetItem, KeyOf, PutItem)* átdefiniált változataival végezhetjük.

Sztring lista

A *TStringList* kezeli a sztring erőforrások egy speciális fajtáját, amelyben a *Get* metódussal a sztringeket numerikus indexük alapján érhetjük el. A *Count* mező tartalmazza az objektumban lévő sztringek számát. A *TStringList* a nemzetközi és soknyelvű szövegek alkalmazását könnyíti meg. A sztring lista a stream-ről a *Load* konstruktorral olvasható. A sztring lista létrehozását és bővítését a *TStrListMaker* metódusaival tehetjük meg.

7. A VIEW OBJEKTUMOK

Az előző fejezetekben láthattuk, hogyan néz ki a Turbo Vision kívülről. Most nézzük meg részletesebben mi is van a szinfalak mögött. A Turbo Vision alapvető eleme a látvány (*view*). A *view* egy Turbo Pascal objektum, amely a képernyőn egy téglalap alakú területet jelöl ki. Például a képernyő tetején megjelenő menüsor (menubar) is egy ilyen *view*. A menük is *view*-k, akárcsak a státuszsor (*status line*), a vezérlő gombok (button), a görgető sor, a dialógus dobozok (*dialog box*) de még egy egyszerű szövegsor is az.

Összefoglalva, minden ami a Turbo Vision programban megjelenít valamit a képernyőn, *view*-nak kell lennie. Például, ha egy menürendszert szeretnénk létrehozni, egyszerűen közölnünk kell a Turbo Vision-nel, hogy milyen menüpontokat tartalmazó menüről van szó, és a többi feladatot elvégzi a Turbo Vision. A programunk is tulajdonképpen egy *view*, amely más *view*-kat (ún. *subview*) vezérel, a felhasználóval való kapcsolat kialakítása céljából.

Mint ahogy a következőkben látni fogjuk, a *view* lehet egyszerű - mint pl. az ablak (*window*) -, de lehet csoportos is (*group*), amely egymással kapcsolatban álló *view*-kat tartalmaz.

7.1. Az egyszerű *view* objektumok

A Turbo Vision által kezelt képernyőelemek (*view*-k) szintén a *TObject* őstípusból származnak. A *TObject* tulajdonképpen minden objektum őse, de valójában a Turbo Vision maga a *TView* objektummal kezdődik.

Maga a *TView* üres téglalapként jelenik meg a képernyőn. Az egyszerű megjelenés ellenére a *TView* magában foglalja a Turbo Vision összes olyan metódusát és adatmezejét, amelyek szükségesek az alapvető képernyőműveletek elvégzéséhez.

Két dolog van, amit minden *TView*-ból származtatott objektumnak tudnia kell:

- Bármikor képes legyen megjeleníteni önmagát a képernyőn. A *TView* tartalmaz egy *Draw* nevű virtuális metódust, amelyet minden leszármazottnak is tartalmaznia kell. Ennek feladata a megjelenítés elvégzése. Ez nagyon fontos metódus, hisz a *view*-k takarhatják, átfedhetik egymást, és ha az egyik *view* kikerül a takart helyzetéből, annak feltétlenül meg kell jelennie a képernyőn.

- Alkalmasnak kell lennie minden, objektumhoz kapcsolódó esemény (*event*) lekezelésére. A Turbo Vision programok ún. esemény-vezérelt programok. Ez azt jelenti, hogy Turbo Vision összegyűjti az input adatokat a felhasználótól, majd azokat szétosztja az alkalmazói program objektumai között. Az események kezelésével a következő fejezetben foglalkozunk.

7.1.1. A *view* elhelyezése a képernyőn

Mielőtt egy *view* működésének részletes tárgyalásához hozzákezdenénk, nézzük meg, hogy mitől tud a *view* megjelenni a képernyőn.

Egy *view* helye a képernyőn két pont megadásával definiálható - ezek a bal felső sarok (az ún. origó) és a jobb alsó sarok. Mind a két sarkot egy-egy *TPoint* típusú mező tárolja. Az *Origin* mező a *view* kiindulási pontját (origó) adja meg, míg a *Size* mező a jobb alsó sarkot reprezentálja. Meg kell jegyeznünk, hogy az *Origin* pont a tulajdonos *view* koordináta rendszerében jelöli ki az új *view* origóját. Például, ha egy ablakot nyitunk a *desktop*-on, az *Origin* az ablak kiindulási pontjának a *desktop* origójához képesti relatív helyzetét tárolja. A *Size* mező, ellentétben az *Origin* mezővel, a *view* saját koordináta rendszerében jelöli ki a jobb alsó sarok helyét.

A *TPoint* típus mindössze két mezőt tartalmaz - *X* és *Y* - a koordináták tárolására, és nincsenek metódusai.

A Turbo Vison-ban ritkán használjuk közvetlenül a *TPoint* típust. Mivel minden *view* tartalmazza a *Origin* és a *Size* adatokat, ezeket általában együtt kezeljük egy *TRect* objektumban összefogva. A *TRect* típusnak szintén két adatmezeje van, az *A* és a *B*, amelyek *TPoint* típusúak.

Mind a *TRect*, mind a *TView* objektumok hasznos metódusokat tartalmaznak a *view* méretének beállítására. Például, ha szeretnénk egy *InsideView* típusú *view*-t maximális méretekkel a *ThisWindow* ablakba belehelyezni, az következő lépésekkel tehetjük meg ezt:

```
procedure ThisWindow.MakeInside
var
   R      : TRect;
   Inside : PInsideView;
begin
   GetExtent(R);    { a ThisWindow méretének tárolása az R-ben }
   R.Grow(-1,-1);   { a téglalap összezsugorítása minden irány-
                                                  ban 1 egységgel }
   Inside:=New(PInsideView,Init(R));
                    { a belső view létrehozása }
   Insert(Inside); { az új view beillesztése az ablakba }
end;
```

A Turbo Vision koordinátákat feldolgozó metódusainak használatakor figyelembe kell venni, hogy a koordináta rendszer valamelyest eltér a megszokottól. Az eltérés abban áll, hogy a Turbo Vision-ban nem a karakter pozíciók adják a koordinátákat, hanem egy képzeletbeli rács vonalai, amelyek a karakterek között futnak.

Például, ha *R* egy *TRect* objectum, az *R.Assign(0,0,0,0)* hívás egy 0 méretű téglalapot, vagyis egy pontot definiál. A legkisebb téglalap, amely alkalmas egy karakter tárolására a *R.Assign(0,0,1,1)* hívással állítható elő. A 7.1. ábrán láthatjuk az *R.Assign(2,2,5,4)* által létrehozott téglalapot:

```
  0 1 2 3 4 5 6 7 8
0 +-+-+-+-+-+-+-+-+
  | | | | | | | | |
1 +-+-+-+-+-+-+-+-+
  | | | | | | | | |
2 +-+-+-+-+-+-+-+-+
  | | |X|X|X| | | |
3 +-+-+-+-+-+-+-+-+
  | | |X|X|X| | | |
4 +-+-+-+-+-+-+-+-+
  | | | | | | | | |
5 +-+-+-+-+-+-+-+-+
```

7.1. ábra
Turbo Vision koordináta-rendszer

Bár ez a koordináta-rendszer nem hagyományos, használatát mégis indokolja, hogy a különböző műveletek, mint pl. a téglalapok méretének kiszámítása, a szomszédos téglalapok koordinátáinak meghatározása és még több más művelet is sokkal könnyebben elvégezhető.

7.1.2. A *view* megjelenítése a képernyőn

A *view* megjelenését a *Draw* metódus határozza meg. Közelebbről, minden új *view* típusnak rendelkeznie kell saját *Draw* metódussal, amely feladata általában a többi *view*-tól különböző *view* megjelenítése.

Az alábbi szabályokat a képernyőn megjelenő minden *view* használatakor figyelembe kell vennünk:

- A *view* felelős a hozzátartozó teljes képernyőterületért.

- A *view* bármikor tudja megjeleníteni önmagát.

Nézzük meg részletesebben mit is jelentenek a fenti sajátosságok.

Nyomós oka van annak, hogy a *view* felelős a saját területéért. Mint ahogy láttuk, a *view* a képernyőn egy téglalap alakú területet definiál. Ha azonban a *view*-hoz tartozó terület csak egy részét használjuk, a másik rész definiálatlan lesz. A TVGUID06.PAS program bemutatja, hogy milyen zürzavar keletkezhet a képernyőn, ha a fenti szabályok valamelyikét figyelmen kívül hagyjuk.

7.1.3. A *view* viselkedése

A *view* viselkedését csaknem teljes egészében meghatározza a *HandleEvent* metódus. A *HandleEvent* metódus egy *TEvent* típusú rekordot kap paraméterként, amelyben tárolt eseményeket a következő két lehetőség valamelyikével kell feldolgoznunk.

Vagy az eseményt lekezeljük - vagyis az eseményre reagálva végrehajtunk bizonyos műveleteket - és jelezzük, hogy az esemény lekezelése megtörtént, vagy az esemény lekezelését továbbítjuk más *view*-nak.

Például, ha egy ablak fogad egy eseményt, amely a *cmNewWin* parancsot tartalmazza, a várt viselkedés, hogy új ablak nyílik. Nézzünk egy tipikus megoldást az elmondottak illusztrálására:

```
procedure TMyApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);    { az esemény átadása }
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmNewWin:NewWindow;      { saját események feldolgozása }
    else
      Exit;
    end;
    ClearEvent(Event);        { visszajelzés a feldolgozásról }
  end;
end;
```

7.2. Összetett *view* objektumok

A *TView* legfontosabb leszármaztatott objektuma a *TGroup*. A *TGroup* objektumra és leszármazottaira mint *csoport*ra fogunk hívatkozni. Azokat a *view*-kat, amelyek nem a *Tgroup*-ból származnak *végső* (*terminal*) *view*-nak fogjuk nevezni.

7.2.1. Csoportok és *subview*-k

A *subview* olyan *view*, amely valamely más *view*-nak (tulajdonos) alá van rendelve. Ez azt jelenti, hogy bizonyos *view* (csoport) által definiált képernyőterület egyes részeit más-más *view* (*subview*) kezeli.

Kiváló példa a csoportra *(group)* a *TApplication*. A *TApplication* olyan *view*, amely gyakorlatilag a teljes képernyőt vezérli. Emellett a *TApplication* három *subview*-val (menüsor, *desktop*, státuszsor) rendelkező csoport. A 5.1. ábrán láthatjuk, hogy ez a három *subview* a képernyő mely régióit kezeli.

7.2.2. Felvétel a csoportba

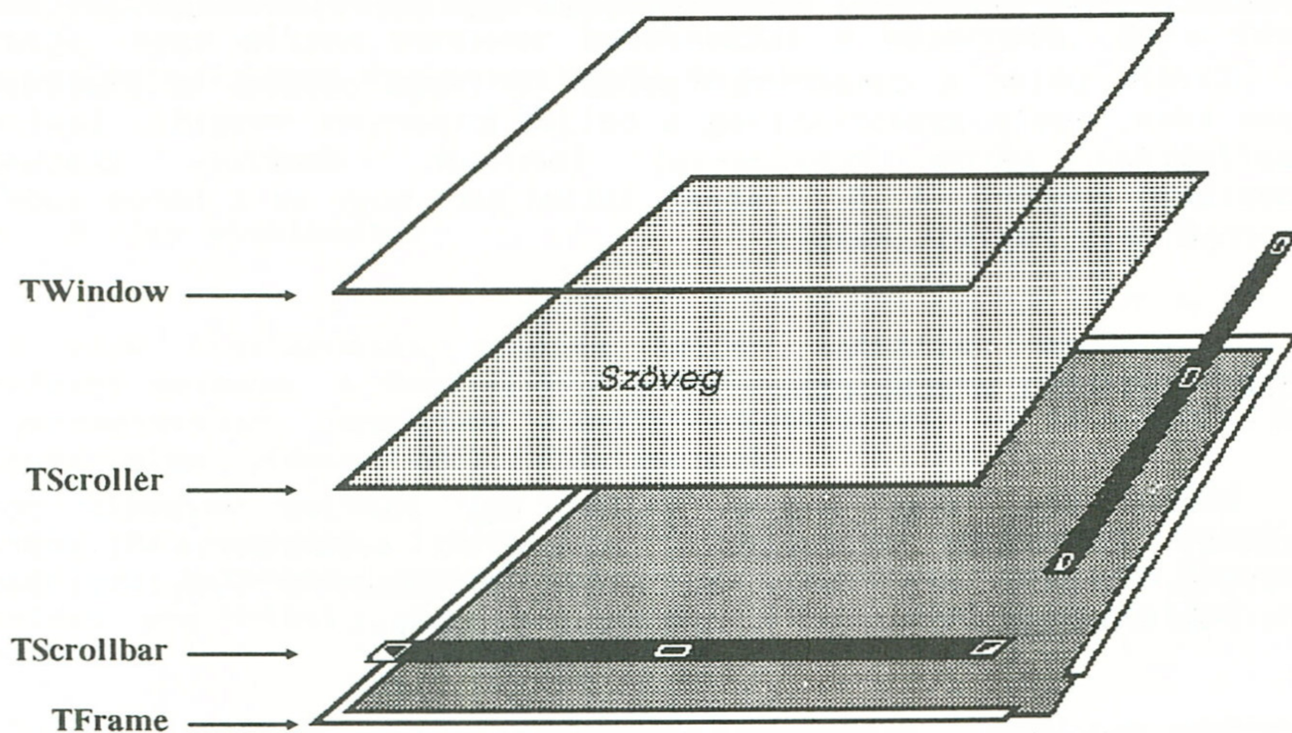
Nézzük meg, hogy hogyan válhat egy *subview* valamely csoport tagjává. Az ehhez szükséges műveletet *beszúrásnak* (*insertion*) nevezzük. Nézzük meg, hogy az előző példában a *TApplication.Init* konstruktor hogyan végzi el a szükséges lépéseket.

```
InitDeskTop;          { a desktop inicializálása    }
InitStatusLine;       { a státuszsor inicializálása }
InitMenuBar;          { a menüsor inicializálása    }
                      { ha sikerült az inicializálás, az objektumok
                        beépülnek az felhasználói programba }

if DeskTop <> nil then Insert(DeskTop);
if StatusLine <> nil then Insert(StatusLine);
if MenuBar <> nil then Insert(MenuBar);
```

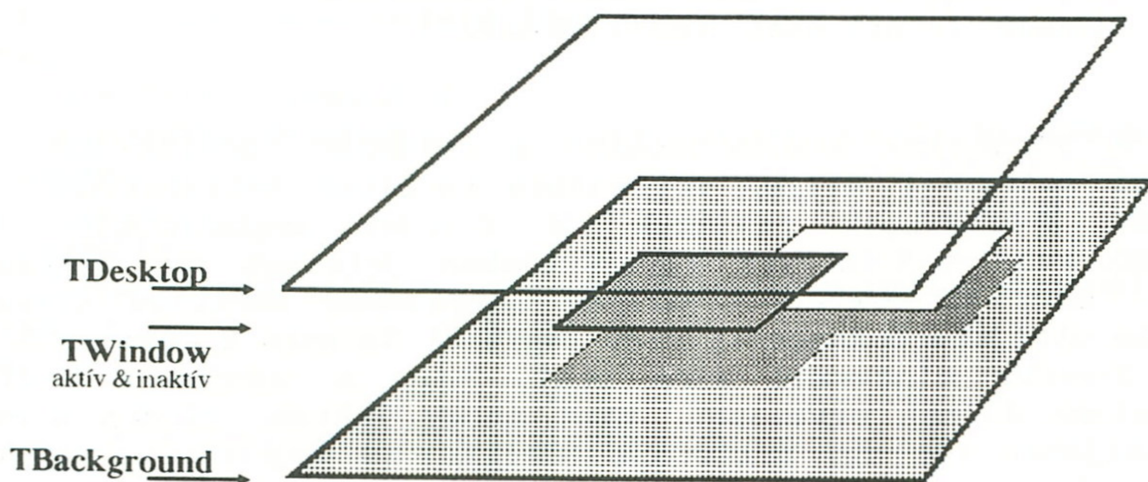
A csoportok kialakításakor a rendszer tárolja azt, hogy a különböző *subview*-k milyen sorrendben kerültek beillesztésre. Ezt a sorrendet *Z-order*-nek nevezzük. A *Z-order* meghatározza, hogy a különböző *subview*-k milyen egymásutánban jelennek meg a képernyőn, illetve, hogy az események milyen sorrendben kerülnek átadásra. A csoportba utoljára bekapcsolt *view* legfelül látható a képernyőn.

A *Z-order* elnevezés arra utal, hogy a *subview*-k 3-dimenziós kapcsolatban állnak egymással. Mint ahogy láttuk, minden *view*-nak a saját síkjában van egy poziciója (*Origin*) és egy mérete (*Size*) van. Mivel azonban a *view*-k és *subview*-k átfedésbe is kerülhetnek, a Turbo Vision-nak tudnia kell, hogy melyik *view* van elől és melyik hátul. Ezért szükség volt a háromdimenzió, azaz a Z-tengely menti "koordináták" felhasználására.

Minden csoport felbontható és ábrázolható egy ún. szendvics szerkezetben. Példaképpen tekintsük a szövegmegjelenítő ablak (*window*) felépítését 3 dimenzióban.



7.2. ábra
A szövegmegjelenítő ablak oldalnézetben



7.3. ábra
A *desktop* oldalnézetben

A *TWindow* ablak, mint egy üveglap fedi le a *view*-k csoportját. Mint látható a 7.2. ábrán, a szöveg nem az ablakon, hanem a görgető mezőn jelenik meg.

A 7.3. ábrán egy nagyobb léptékű ábrázolásban láthatjuk magának a *desktop*-nak a 3-dimenziós szerkezetét. Érdekes megjegyezni, hogy az ablakok, mint kisebb szendvicsek helyezkednek el a nagyobb szendvicsben.

7.2.3. A csoportok megjelenítése a képernyőn

A csoportok nem teljesítik azt a feltételt, hogy a *view*-nak tudnia kell megjeleníteni önmagát a képernyőn. A *TGroup* a megjenítéshez a *subview*-kat aktivizálja, hogy ahol azok rajzolják ki saját magukat.

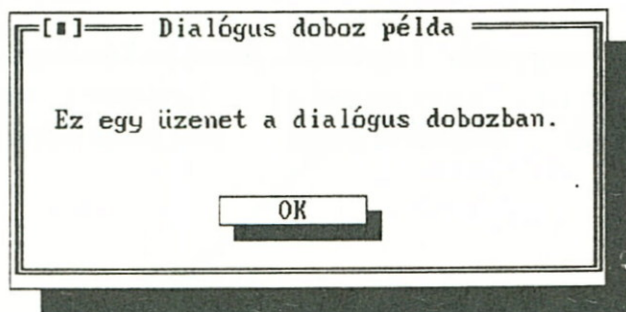
A *subview*-k *Draw* metódusai a *Z-order* által definiált sorrendben kerülnek meghívásra. Ez azt jelenti, hogy elsőként az először beillesztett *subview* jelenik meg. Ilyen módon a képernyőn legfelülre a legutoljára beillesztett *subview* kerül.

Mindig figyelembe kell vennünk, hogy egy csoporthoz tartozó *subview*-k úgy működjenek együtt, hogy teljesen lefedjék a csoport által vezérelt képernyőterületet. Példaként tekintsük a dialógus dobozt, ami szintén egy csoport. A dialógus dobozban taláható *subview*-k - a keret (*frame*), a belvilág (*interior*), vezérlők (*check box, radio button* és a *push button*) és a statikus szöveg - be kell hogy töltsék a dialógus doboz teljes területét.

Amikor valamely csoporthoz tartozó *subview* megjeleníti önmagát, a rajz automatikusan lehatárolódik (levágódik) a csoport határainál. Így fontos, hogy amikor egy *view*-t inicializálunk és beillesztünk egy csoportba, a *view* legalább részletében a csoport határain belül helyezkedjék el. Ellenkező esetben a *subview* sohasem kerül ki a képernyőre.

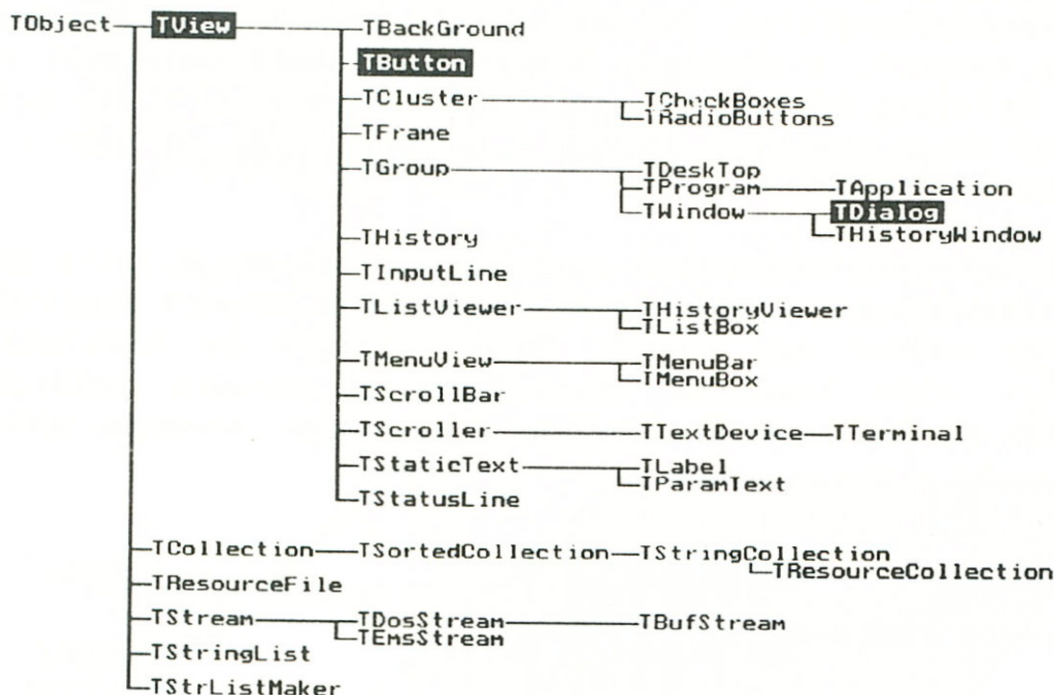
7.2.4. Kapcsolatok a *view*-k között

A *view*-k két kölünböző módon lehetnek kapcsolatban egymással: egyrészt tagjai a Turbo Vision objektum hierarchiájának, ill. másrészt tagjai az ún. *view fának*. Példaként tekintsük a 7.4. ábrán látható dialógus dobozt, amely rendelkezik kerettel, egysoros szöveggel és egy vezérlő gombbal. A Turbo Vision terminológiát használva mondhatjuk, hogy a *TDialog view* tulajdonosa a *TFrame*, a *TStaticText* és a *TButton* *subview*-knak.



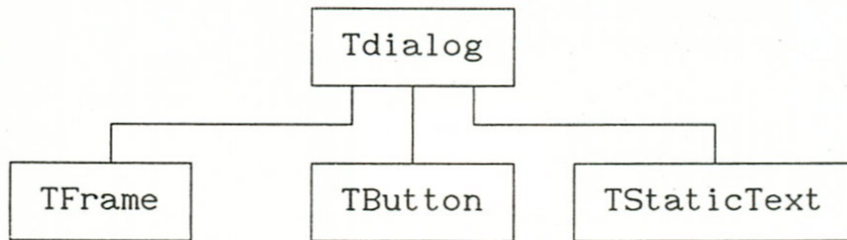
7.4. ábra
Egyszerű dialógus doboz

Az első mód, amely szerint a *view*-k szülő-gyerek kapcsolatban állnak, az objektumok hierarchiája. A 7.5. ábrán látható, hogy a *TButton* a *TView* leszármazottja. A *TDialog* szintén a *TView*-tól származik, így a *TButton*-nal sok mindenben megegyezik.



7.5. ábra
Turbo Vision objektum-hierarchia

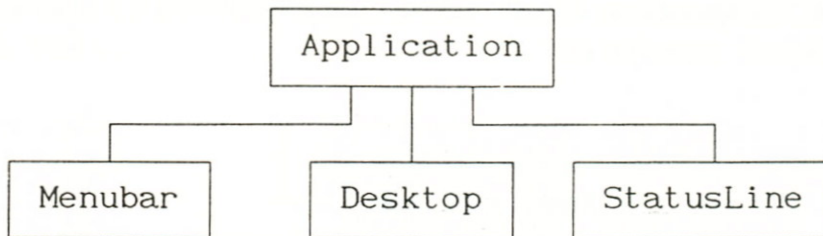
A kapcsolatok másik módja szerint a *view*-k a *view* fa szerinti viszonyban állnak. A fenti példához tartozó *view* fa az alábbi:



Látható, hogy a *TDialog* tulajdonosa a *TButton view*-nak, annak ellenére, hogy az objektumok hierarchiájában a *TDialog* nem a *TButton*-tól származik. Itt a rokonság nem a hierarchikus objektum típusok között áll fenn, hanem az egyes objektum példányok között ill. a tulajdonos és a *subview*-k között.

7.2.5. A *subview*-k és a *view* fák

Mint ahogy korábban már említettük, a *TApplication view* három *subview*-t tartalmaz és kezel. Az ezek közötti kapcsolatokat a Turbo Vision alap *view* fája mutatja:

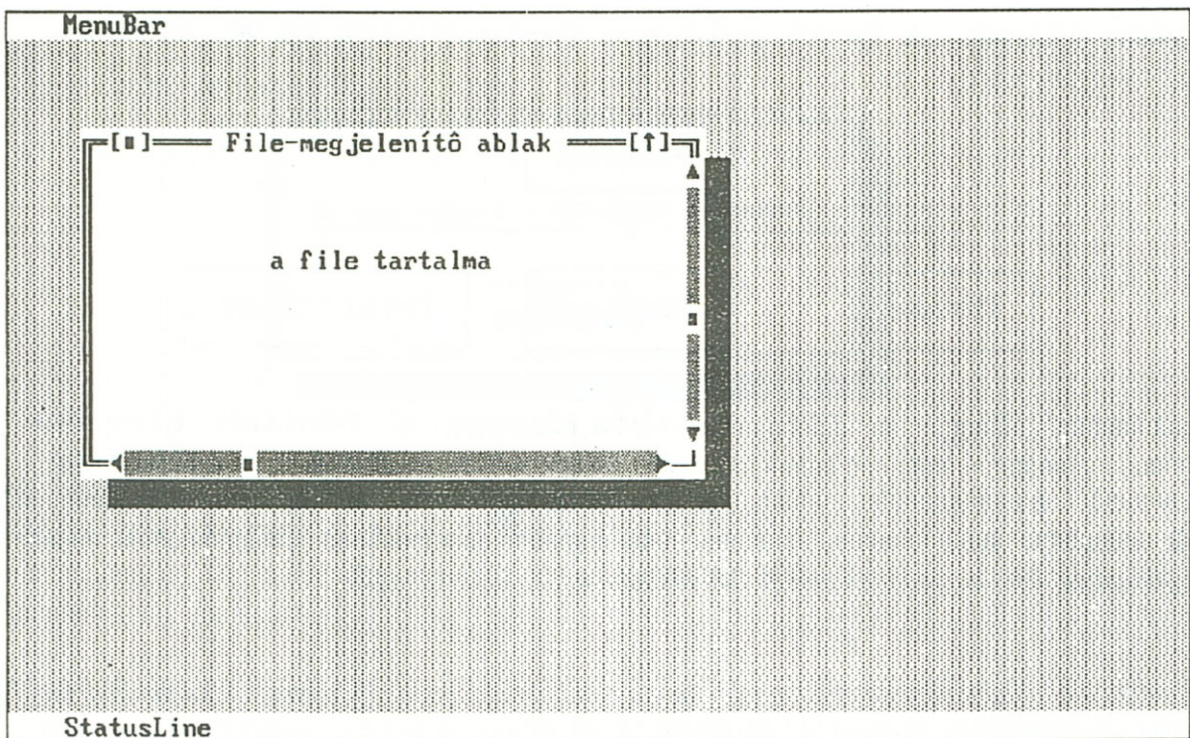


Egy tipikus Turbo Vision alkalmazói programban a felhasználó vagy az egeret, vagy billentyűzetet használva több *view*-t hozhat létre. Ezek a *view*-k megjelennek a képernyőn és a *view* fán új ágakat alakítanak ki.

Nagyon fontos megértenünk ezeket a kapcsolatokat a tulajdonos *view*-k és a *subview*-k között. Látnunk kell azt is, hogy mind a *view* megjelenése, mind a viselkedése nagyban függ attól, hogy ki a tulajdonosa.

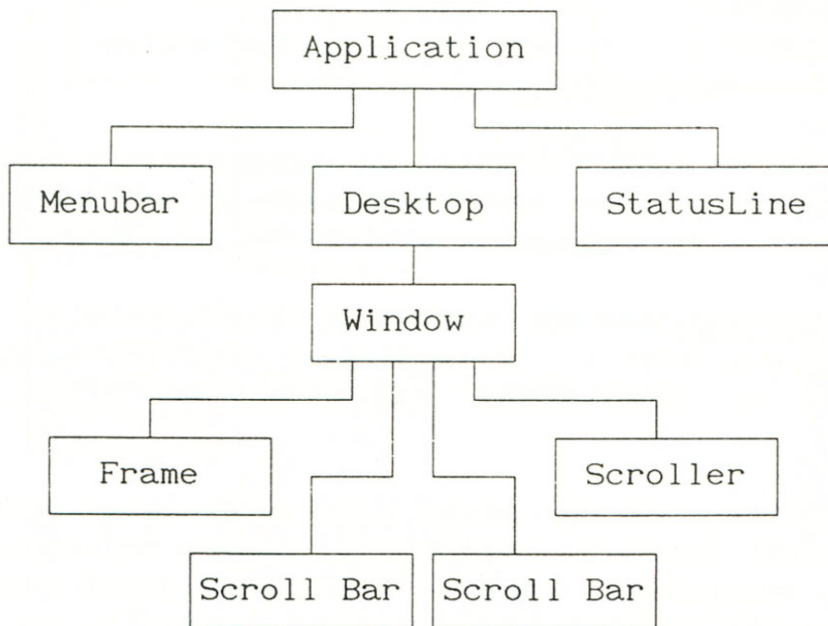
Gondolatban futtassunk egy olyan alkalmazói programot, amely file-megjelenítő ablakokat használ. Tegyük fel, hogy a felhasználó a menüből kiválasztja azt a menüpontot, amely a file-megjelenítő ablakot hívja. A file-megjelenítő ablak szintén *view*, amit a Turbo Vision létrehoz és a *desktop*-hoz kapcsol.

Az ablakot tartalmazó képernyőtartalmat a 7.6. ábrán láthatjuk.



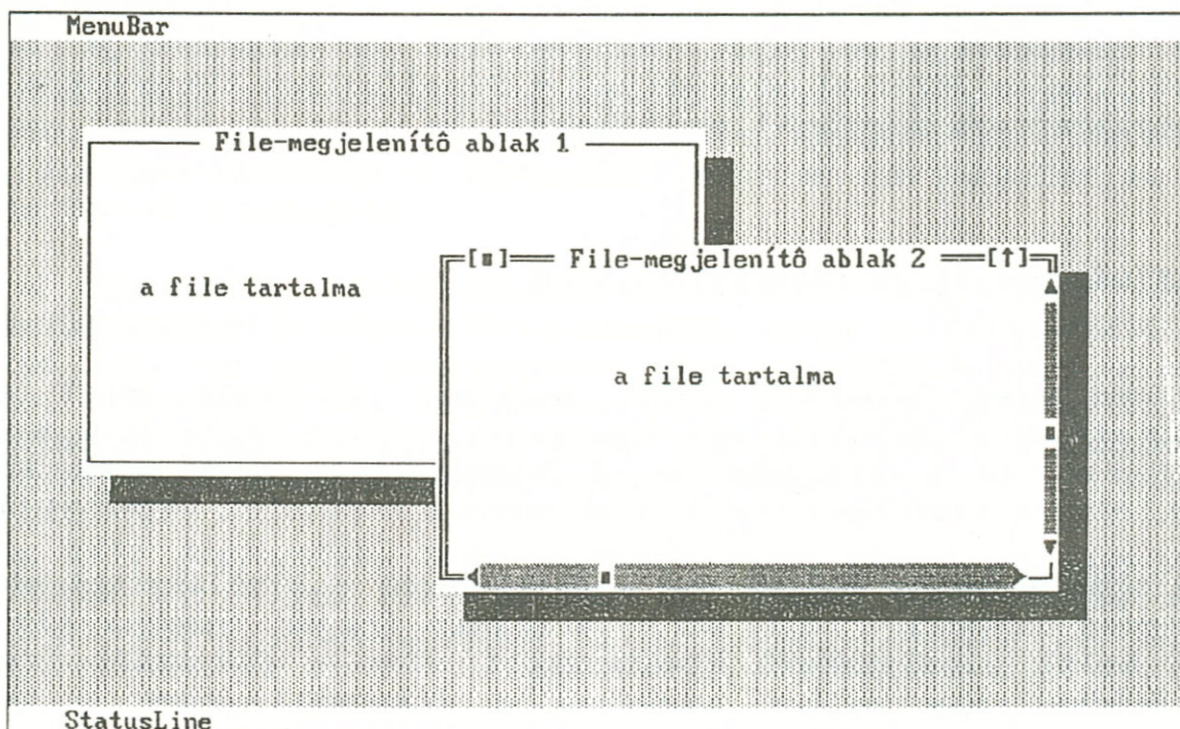
7.6. ábra
A file-megjelenítő ablak a a *desktop*-on

Az ablak hozzákapcsolásával az alap *view* fa kiterebélyesedik és létrejön az következő szerkezet:



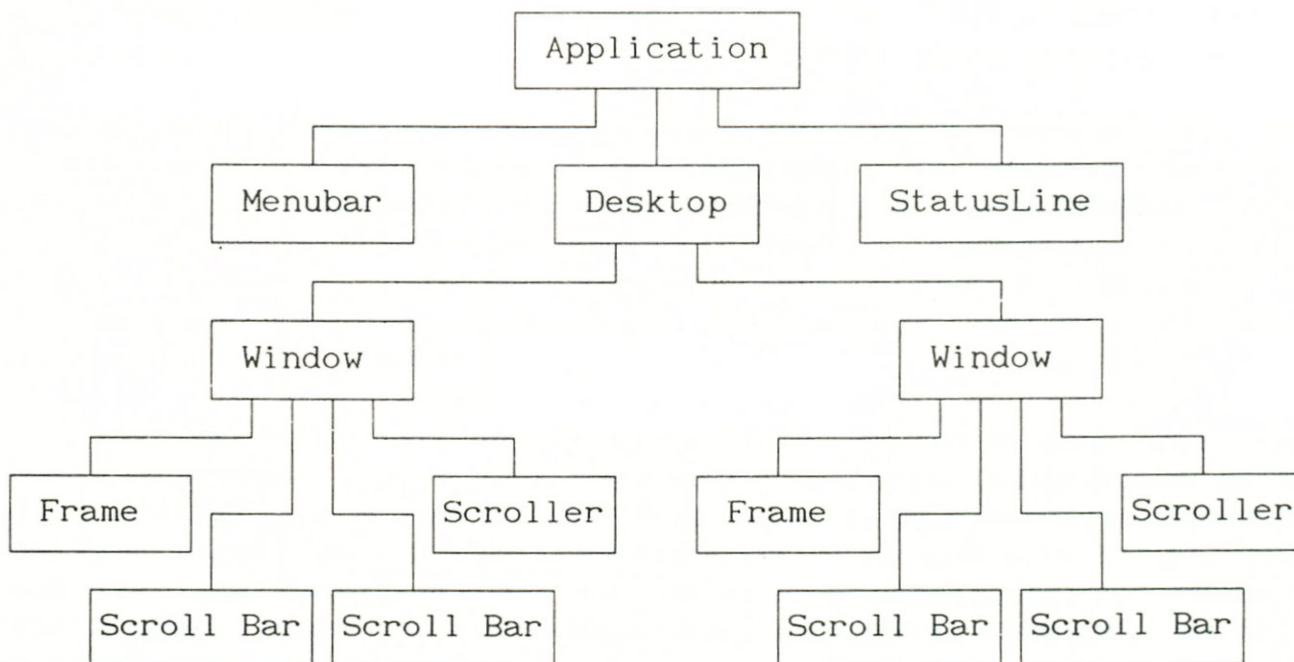
Ha ezek után a felhasználó újra kiválasztja az előbbi menüpontot, a Turbo Vision egy másik ablakot is létrehoz és szintén beilleszti a

desktop-ba. A keletkező kép a 7.7. ábrán látható.



7.7. ábra
A menüpont újra történő kiválasztása utáni képernyő tartalom

A *view* fa struktúrája az újabb ablak létrehozásával még bonyolultabbá vált:



A következő fejezetben látni fogjuk, hogy a program vezérlése a *view* fa alapján felülről lefelé megy végbe.

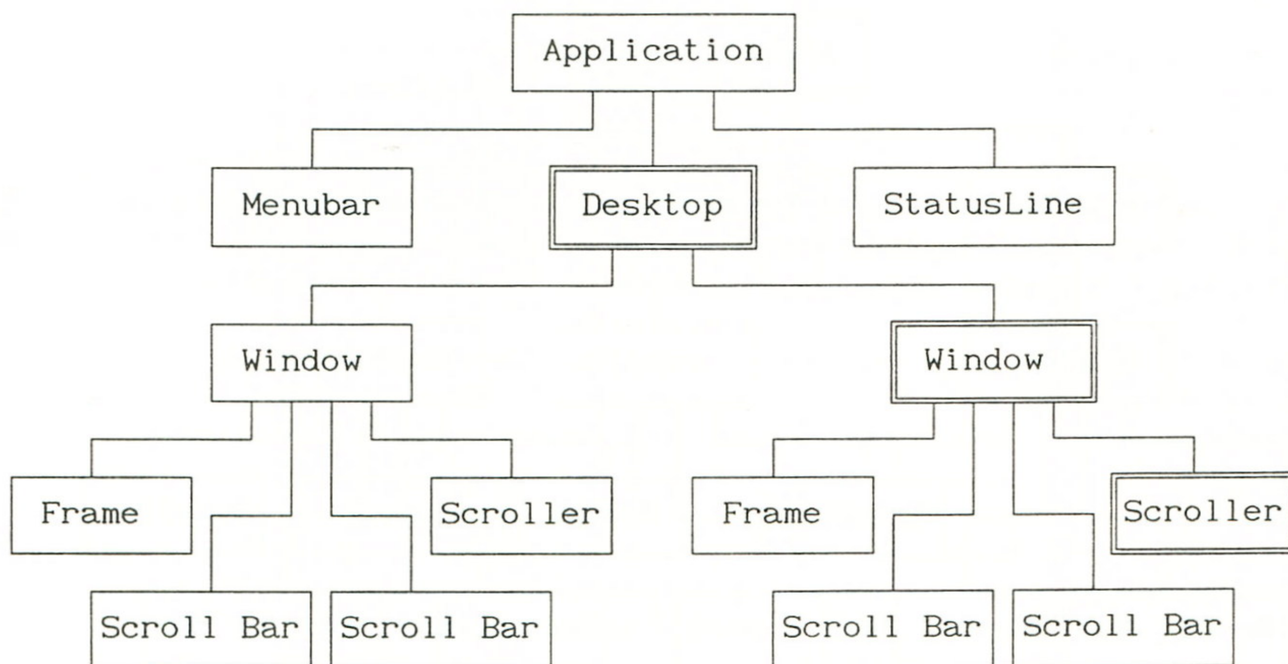
Ha a felhasználó a második ablak záró ikonjára klikkent az egérrel, a második ablak bezáródik. A Turbo Vision rendszer leveszi az ablak *view*-t a *view* fáról és kikapcsolja azt. A kikapcsoláskor az ablak először kikapcsolja a saját *subview*-ikat és kikapcsolja önmagát. Végezetül, ha az *ALT-X*-et megnyomjuk, a *TApplication* kikapcsolja mind a három hozzátartozó *subview*-t majd önmagát is.

7.2.6. Kiválasztott és fókuszált *view*-k

Minden *view* csoporton belül pontosan egy *view* lehet az ún. *kiválasztott view*. Például, egy alkalmazás magában foglalja a menüsort, a *desktop*-ot és a státuszsort. A kiválasztott *view* ezek közül a *desktop*, hisz a szükséges teendőinket ezen a területen végezhetjük el.

Ha több ablak is nyitva van a *desktop*-on, a kiválasztott ablak az, amely éppen működik - ezt *aktív* ablaknak fogjuk nevezni. Az aktív ablakon belül elhelyezkedő kiválasztott *view* a *fókuszált* (*focused*) *view*. Például egy editor *view*-ban a fókuszált *view* a belvilág (*interior*), ami a szöveget tartalmazza, egy dialógus dobozban a fókuszált *view* például a kivilágított vezérlő gomb.

Tekintsük a 7.7. ábrának megfelelő *view* fát. Az új típusú *view*-kat kettős vonallal kiemeltük.



Látható, hogy a *desktop*-on belül a második ablak a kiválasztott *view*. Ezen az ablakon belül a görgethető belső rész a kiválasztott *view*, de mivel ez a *view* olyan *view*, amely a lánc végén található, ez

lesz a fókuszált *view*. Mindig tudnunk kell, hogy melyik *view* a fókuszált *view*, hisz az a *view* fogad el csak információt a billentyűzetről.

A fókuszált *view* mindig valamilyen kiemelt módon jelenik meg a képernyőn. Például ablakok esetén a kettős vonalból álló keret jelzi ezt. A dialógus dobozban a fókuszált vezérlés (a vezérlés is *view*!) szinében tér el a többitől.

A főkuszálás kétféleképpen valósítható meg. Vagy megadható alapértelmezésként a *view* létrehozásakor, vagy a felhasználó a *TAB* billentyű ill. az egér segítségével maga állíthatja be. Amikor egy csoportot létrehozunk, a tulajdonos *view* a *Select* metódus hívásával definiálhatja, hogy a *subview*-k közül melyik legyen a fókuszált. Ily módon létrejön az *alapértelmezés szerinti fókusz*.

Meg kell jegyeznünk, hogy léteznek olyan *view*-k is, amelyek nem választhatók ki. Ilyen *view* például a *desktop* háttere, az ablakok kerete. A *view*-k létrehozásakor a kiválaszthatóságot is meg kell jelölni.

Ha futtatunk egy alkalmazást, és nyomon követjük a kiválasztott *subview*-kat, a lépéssorozat végén a fókuszált *view*-hoz jutunk. A *view*-knak ezt a láncolatát, a *TApplication* objektumtól a fókuszált *view*-ig *fókusz lánc*nak hívjuk. Ezt a láncot használja a Turbo Vision rendszer a fókuszált események irányított lekezeléséhez.

7.2.7. Modal *view*-k

Egy programnak lehet többféle *működési mód*ja (*üzemmód*ja), amelyeket általában különböző vezérlő funkciókkal és vezérlő területekkel választunk el egymástól. Példaként tekintsük a Turbo Pascal integrált fejlesztői környezetét, amelynek van editor, nyomkövető, fordító és futtató üzemmódja. Attól függően, hogy melyik működési módot aktivizáltuk, ugyanazon billentyűk lenyomásával más-más effektusokat tudunk előidézni.

A Turbo Vision-ban egy *view* szintén meghatározhat egy működési módot - az ilyen *view*-t *modal view*-nak nevezzük. Klasszikus példája a *modal view*-nak a dialógus doboz. Általában ha egy dialógus doboz aktív, semmilyen külső funkció sem aktivizálható, így gyakorlatilag a dialógus doboz a lezárásáig átveszi a program vezérlését. A *modal view* meghatározza, hogy milyen viselkedés érvényesül önmagán belül - az eseményeket a *modal view* ill. a hozzá tartozó *subview*-k kezelik. A *view* fának azon részei, ami nem a *modal view*, vagy nem a *modal view* a tulajdonosa, nem aktivizálhatók.

A státuszsor kivétel a fenti szabály alól. Ez azt jelenti, hogy nekünk lehetnek aktív státuszsor elemeink akkor is, ha pl. egy dialógus

dobozban vagyunk, amelynek nincs saját státuszsora. Azonban, a státuszsor által generált események és parancsok csak akkor kerülnek feldolgozásra, ha a *modal view*-n belül generálódtak.

Ha egy Turbo Vision alkalmazás fut, mindig létezik legalább egy *modal view*. Ez nem más, mint maga a program, a *TApplication* objektum a *view* fa legtetején.

7.2.8. Az alapértelmezés szerinti viselkedés módosítása

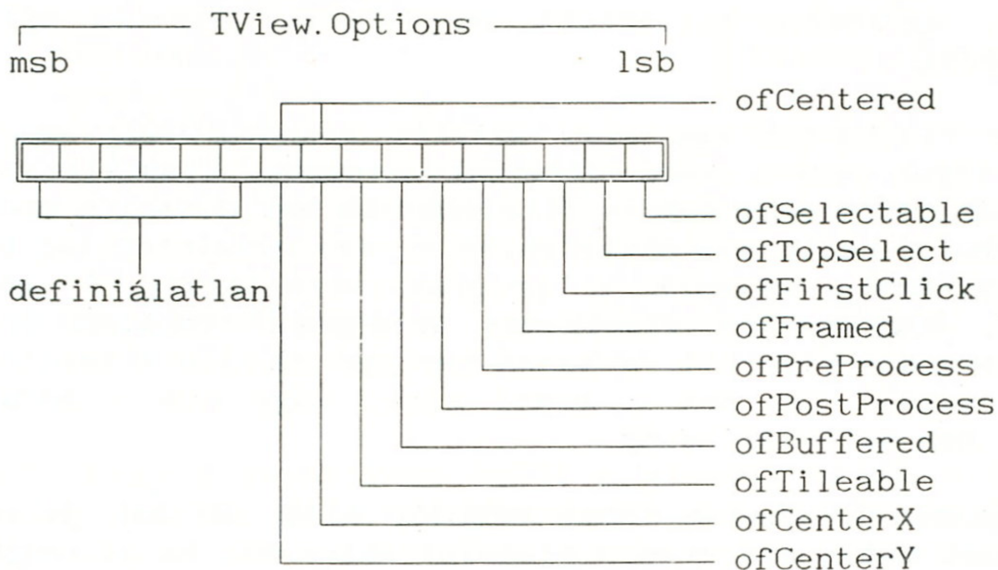
Ez idáig a szabványos *view*-kat, alapértelmezés szerinti viselkedésükkel használtuk. Ebben a részben megnézzük, hogyan lehet megváltoztatni a *view*-k viselkedését a felhasználás igényének megfelelően.

Minden Turbo Vision *view*-hoz tartozik négy bitmező, amelyek segítségével a viselkedés megváltoztatható. Hármat ezek közül (*Options*, *GrawMode*, *Dragmode*) ebben a fejezetben ismertetünk, míg a negyedik (*EventMask*) a következő fejezetben kerül tárgyalásra.

Létezik még egy ún. *állapotszó* (*State word*) is, amely a *view* aktuális állapotáról tartalmaz információkat. Ellentétben az előzőekkel az állapotszó szó csak olvasható, ill. értéke csak a *SetState* metódus hívásával állítható be.

7.2.8.1. Az Options mező

Az *Options* mező minden *view*-ban megtalálható. A *TView* különböző leszármazottjai alapértelmezésben egészen eltérő opció (*Options*) beállításokkal rendelkeznek.



ofSelectable

Ha az *ofSelectable* be van állítva, a felhasználó ki tudja választani a *view*-t az egér segítségével. Ha a *view* egy csoportban van, akkor szintén kiválasztható vagy egérrel vagy a *TAB* billentyűvel. Ha valamilyen információs *view*-t jelenítünk meg a képernyőn, általában nem szerencsés ha az kiválasztható. Így például a statikus szöveg, az ablakok kerete nem választható ki.

ofTopSelect

Ha ugyanazon csoportban található *subview*-k közül egyet kiválasztunk, az a többi *subview* fölött jelenik meg. Ezt az opciót speciálisan az ablakok számára tervezték, más *view*-k esetén használata nem ajánlott.

ofFirstClick

Az egér klikkentés - amivel egy *view*-t kiválasztunk - átadódik a *view*-nak. Ha a gombot nyomtuk meg, szeretnénk hogy a kiválasztott művelet azonnal végrehajtódjon az egér gombjának egyszeri lenyomására (ofFirstClick=1).

ofFramed

Ha az *ofFramed* értéke 1, a *view*-nak lesz egy látható kerete. Ez például nagyon hasznos, ha egy ablakot több részre szeretnénk felosztani.

ofPreProcess

Ha az *ofPreProcess* értéke 1, lehetővé válik, hogy a fókuszált események feldolgozása már azelőtt végbemenjen, mielőtt a fókuszált *view* érzékelné azokat.

ofPostProcess

Ha *ofPostProcess* be van állítva, engedélyezi a *view* számára a fókuszált események lekezelését, miután a fókuszált *view* érzékelte azokat, feltéve, ha a *view* nem törli az eseményt.

ofBuffered

Ha az *ofButtered* bit be van kapcsolva, a csoportok megjelenítése a képernyőn felgyorsul. Ha egy csoporthoz megérkezik egy kérelem, hogy jelenítse meg önmagát a képernyőn, akkor, ha ez a bit 1 és elég memória

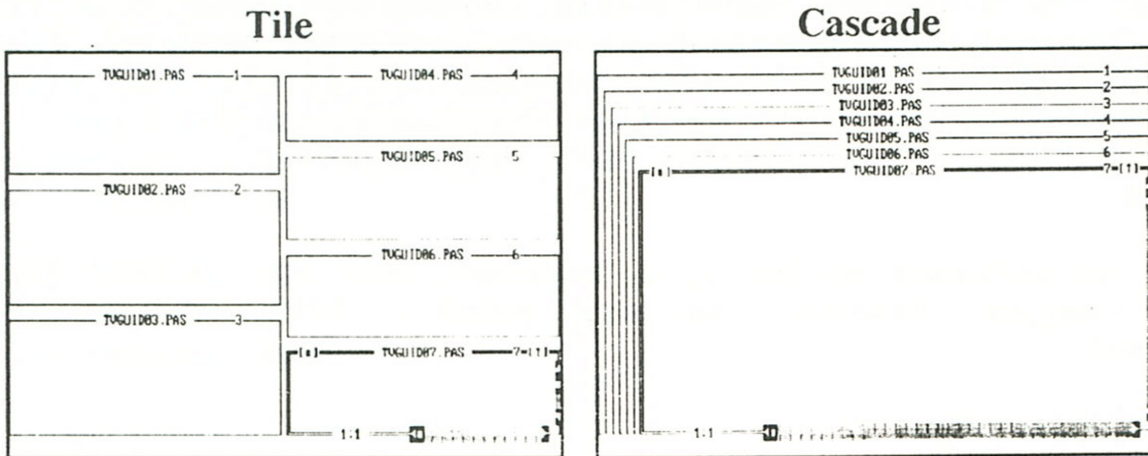
áll a program rendelkezésére, a csoport képe bekerül egy ideiglenes pufferbe is. Ha következő alkalommal érkezik egy ilyen kérelem, akkor egyszerűen a puffer tartalma bemásolódik a képernyőre, ahelyett, hogy a csoport minden tagja külön-külön megjelenítené önmagát.

Ha a *New* vagy a *Getmem* hívásoknál nem elég a memória, a Turbo Vision memória kezelője elkezdi felszabadítani a csoport puffereket, egészen addig, amíg a foglalási kérelem teljesíthetővé nem válik.

Ha egy csoportnak van ilyen gyorsító puffere (*cache buffer*), akkor a *Lock* ill. az *Unlock* metódusok hívásával szabályozhatjuk a puffer tartalmának képernyőre való kiírását.

ofTileable

A *desktop* lehet lefedett (*tile*) vagy átlapolt (*cascade*), amint azt a 7.8. ábrán is láthatjuk. Ha nem akarjuk, hogy az ablakok lefedett módon jelenjenek meg a *desktop*-on, akkor egyszerűen állítsuk 0-ba ezt a bitet.



7.8. ábra
Az ablakok megjelenítési módjai

A *TApplication.HandleEvent* rutinban kétféle megjelenítési lehetőség egyszerűen kiválasztható:

```
cmTile:
      begin
          DeskTop^.GetExtent(R);
          Deshtop^.Tile(R);
      end;
```

```
cmCascade:
        begin
            DeskTop^.GetExtent(R);
            Deshtop^.Cascade(R);
        end;
```

Ha a átlapoláshoz túl sok *view*-t akarunk felhasználni, akkor a *desktop* semmit sem csinál.

ofCenterX

Amikor egy *view*-t beillesztünk valamely csoportba, kérhetjük, hogy x-irányban középre legyen igazítva.

ofCenterY

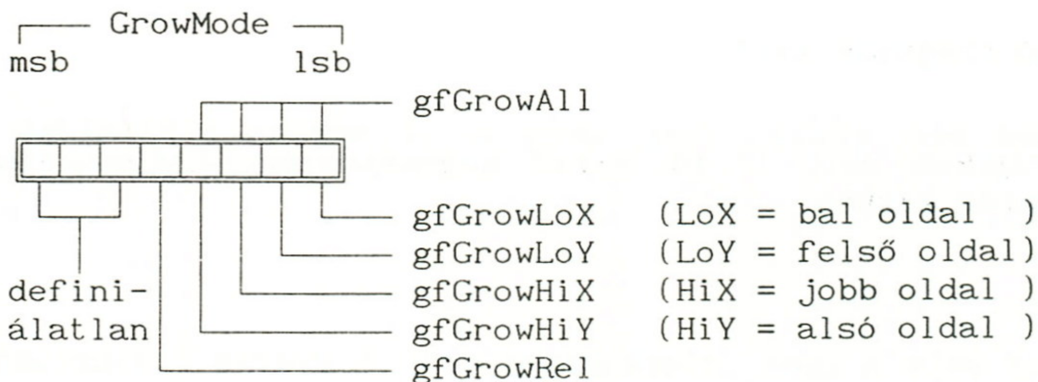
Amikor egy *view*-t beillesztünk valamely csoportba, kérhetjük az y-irányban történő a középre igazítását. Ennek használata abban az esetben fontos, ha olyan ablakokat szeretnénk készíteni, amelyek mind 25 soros, mind 43 soros szöveges módban helyesen működnek.

ofCentered

Mindkét irányban kérjük a *view* középre igazítását, amikor beillesztjük azt egy csoportba.

7.2.8.2. A GrowMode mező

A *view*-k *GrowMode* mezeje meghatározza, hogy a *view* hogyan változik meg, amikor a tulajdonos csoport mérete megváltozik (7.9. ábra).



7.9. ábra
A *GrowMode* mező egyes bitjeinek jelentése

gfGrowLoX

Ha a *gfGrowLoX* értéke 1, a *view* bal oldala konstans távolságra fog elhelyezkedni a tulajdonos csoport bal oldalától.

gfGrowLoY

Ha a *gfGrowLoY* értéke 1, a *view* teteje konstans távolságra fog elhelyezkedni a tulajdonos csoport tetejétől.

gfGrowHiX

Ha a *gfGrowHiX* értéke 1, a *view* jobb oldala konstans távolságra fog elhelyezkedni a tulajdonos csoport jobb oldalától.

gfGrowHiY

Ha a *gfGrowHiY* értéke 1, a *view* alja konstans távolságra fog elhelyezkedni a tulajdonos csoport aljától.

gfGrowAll

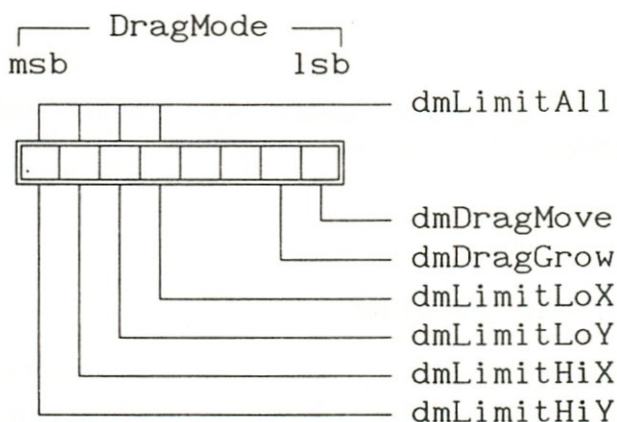
Ha a *gfGrowAll* értéke 1, a *view* mindig megtartja eredeti méretét és követi a tulajdonos jobb alsó sarkának a mozgását.

gfGrowRel

Ha a *gfGrowRel* értéke 1, a *view* mérete relativ a tulajdonos méretéhez képest. Ezt általában a *desktop*-on található *TWindow* (vagy belőle származtatott) objektumok esetén kell használnunk. A *gfGrowRel* használatával, megfelelő méretű ablakokat generál a rendszer mind 25, mind 43/50 soros képernyő esetén is.

7.2.8.3. A *DragMode* mező

A *DragMode* mező (7.10. ábra) meghatározza, hogyan viselkedjen a *view* mozgatás közben.



7.10. ábra
A *DragMode* mező egyes bitjeinek jelentése

dmDragMove

Ha a *dmDragMove* értéke 1, az ablak keretének felső oldalán klikkentve, az ablak mozgatható lesz.

dmDragGrow

Amikor a *dmDragGrow* értéke 1, a *view* képes növekedni.

dmLimitLoX

Ha a *dmLimitLoX* értéke 1, a *view* bal oldala nem kerülhet a tulajdonos *view*-n kivűlre.

dmLimitLoY

Ha a *dmLimitLoY* értéke 1, nem megengedett, hogy a *view* teteje a tulajdonos *view*-n kivűlre essék.

dmLimitHiX

Ha a *dmLimitHiX* értéke 1, a *view* jobb oldala nem kerülhet a tulajdonos *view*-n kivűlre.

dmLimitHiY

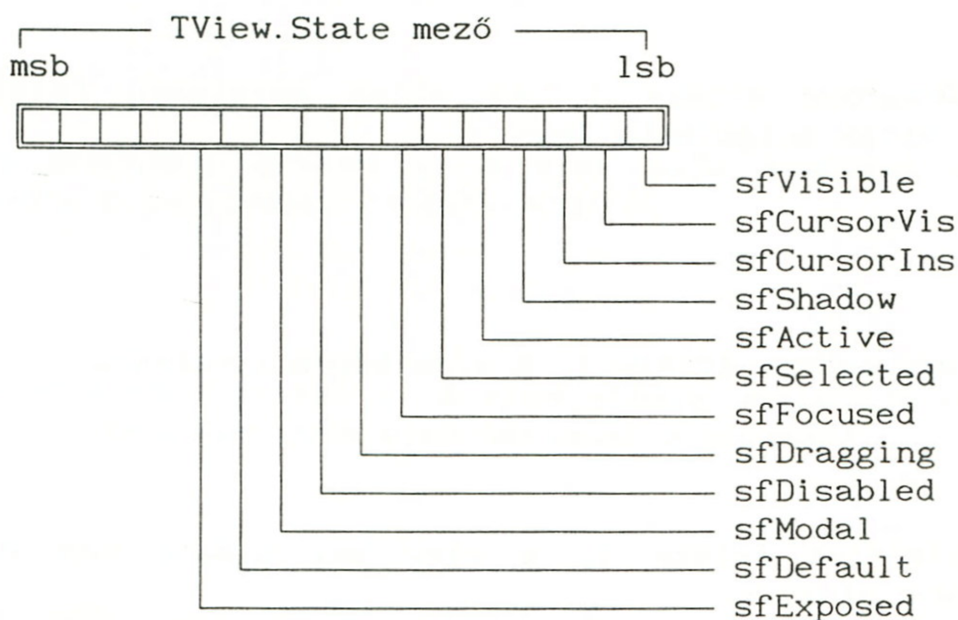
Ha a *dmLimitHiY* értéke 1, nem megengedett, hogy a *view* alsó sora a tulajdonos *view*-n kivűlre essék.

dmLimitAll

Ha a *dmLimitAll* értéke 1, a *view* semelyik része sem mehet a tulajdonos *view* területén kívülre.

7.2.8.4. A State mező és a SetState metódus

Minden *view* rendelkezik még egy bittérkép mezővel is, aminek a neve *State*. Ez a mező sok fontos információt jegyez fel a *view* állapotáról. Először nézzük az egyes bitek jelentését (7.11. ábra), majd pedig tekintsünk át néhány hasznos tudnivalót a felhasználásukról.



7.11. ábra
A *State* mező egyes bitjeinek jelentése

sfVisible

Az *sfVisible* értéke 1, ha a *view* látható a csoporton belül. Alapértelmezés szerint a *view*-k láthatóak. Értékét a *TView.Show* és a *TView.Hide* metódusok állítják. Az *sfVisible* láthatóság nem ad információt arra vonatkozólag, hogy a *view* látható-e a képernyőn, hisz ha a csoport maga nem jelenik meg, akkor egyik hozzátartozó *subview* sem jelenik meg. A képernyőn való láthatóság eldöntéséhez ajánlott a *sfExposed* bitet tesztelni, vagy a *TView.Exposed* függvényt hívni.

sfCursorVis

Az *sfCursorVis* értéke 1, ha a *view* kurzora látható. Az alapértelmezés 0. Értékét a *TView.Showcursor* és a *TView.Hidecursor* metódusok állítják.

sfCursorIns

Az *sfCursorIns* értéke 1, ha a *view* kurzora egy kitöltött téglalap, - 0, ha a kurzor aláhúzásjel. Az alapértelmezés 0. Értékét a *TView.BlockCursor* és a *TView.NormalCursor* metódusok állítják.

sfShadow

Az *sfShadow* értéke 1, ha a *view*-nak van árnyéka, különben 0.

sfActive

Az *sfActive* értéke 1, ha a *view* egy aktív ablak, vagy *subview* egy aktív ablakon belül van.

sfSelected

Az *sfSelected* értéke 1, ha a *view* az aktuálisan kiválasztott *subview* a tulajdonos csoporton belül. Minden *TGroup* objektum rendelkezik egy *Current* adatmezővel, amely az aktuálisan kiválasztott *subview*-ra mutat, ill. értéke **nil** - ha nincs kiválasztott *subview*. Egy *TGroup* objektumon belül csak egy *subview* lehet aktuálisan kiválasztva.

sfFocused

Az *sfFocused* értéke 1, ha a *view* fókuszált. Egy *view* akkor fókuszált, ha ki van választva, és minden tulajdonosa is ki van választva. Tehát az utolsó *view*-ja ennek a fókuszált láncnak a fókuszált *view*, amely végső célja minden fókuszált eseménynek.

sfDragging

Az *sfDragging* értéke 1, ha a *view* mérete változik.

sfDisabled

Az *sfDisabled* értéke 1, ha a *view* le van tiltva, vagyis semmilyen hozzáirányított eseményt sem fogad. Ha 0, akkor a *view* nincs letíltva, tehát eseményeket fogadhat.

sfModal

Az *sfModal* egyes értékkel jelzi, hogy *modal view*-t használunk. Mindig egy *modal view* létezik egy Turbo Vision alkalmazói program futtatása során. Ez általában a *TApplication* vagy a *TDialog* objektum.

sfExposed

Az *sfExposed* értéke 1, ha a *view* tulajdonosa közvetlenül vagy közvetve az *Application* objektum és emiatt valószínű, hogy az objektum látható a képernyőn. A *Tview.Exposed* metódus használja ezt a jelzőbitet.

A Turbo Vision állapotjelzőit a *SetState* metódus állítja be. Minden esetben amikor egy *view* a fókuszba kerül, vagy éppen kikerül onnan, vagy ha kiválasztjuk, a Turbo Vision meghívja ezt a metódust. Így az állapotszó működése eltér az előzőekben ismertetett más bittérkép adatmezők működésétől. Azokat inicializáláskor beállítjuk és többet nem változtatjuk. Például, ha egy ablak mérete megváltoztatható, akkor a program végéig meg is marad ez a tulajdonsága.

Egy *view* állapota - mialatt az a képernyőn jelen van - többször is megváltozhat. Ebből következően a Turbo Vision a *SetState* rutinban egy olyan speciális mechanizmust használ, amely az állapotbitek állítása mellett képes reagálni az állapotban végbemenő változásokra.

A *SetState* metódus paramétere egy állapot (*AState*) és egy jelző (*Enable*), amely jelzi, hogy az állapot beállítódik vagy törlődik. Ha az *Enable* értéke *TRUE* az *AState*-ban kijelölt bitek beállítódnak az állapotszóban, ellenkező esetben törlődnek.

A *view* általában az eredmény állapottól függően valamilyen működéssel reagál a *SetState* hívására. Például egy vezérlő gomb az állapotszó függvényében a színét világos kékre (ciánra) változtatja, ha a fókuszált lesz.

Nézzünk egy jellemző megoldást a *SetState* metódus átdefiniálására valamely *TView* leszármazott objektum esetében:

```
procedure Tbuttom.SetState(AState:Word; Enable:Boolean);
begin
      { az állapotbitek állítása/törlése }
   TView.SetState(AState,Enable);

      { a kirajzolás szükségességének eldöntése }
   if AState and (sfSelected + sfActive) <> 0 then DrawView;

      { a fókusz beállítása vagy törlése }
   if AState and sfFocused <> 0 then MakeDefault(Enable);
end;
```

Egy másik példában ismerős megoldással találkozunk. A fejlesztői környezet editor *view*-jának a *SetState* metódusa képes arra, hogy az összes szövegszerkesztő parancsot letiltsa, vagy engedélyezze attól függően, hogy az editor meg van-e nyitva vagy nincs.

```
procedure TEditor.SetState(AState: Word; Enable: Boolean);
const
    EditorCommands=[cmSearch,cmReplace,cmSearchAgain,cmGotoLine,
                    cmFindProc,cmFindError,cmSave,cmSaveAs];
begin
    TView.SetState(AState,Enable);
    if AState and sfActive <> 0 then
       if Enable then EnableCommands(EditorCommands)
       else DisableCommands(EditorCommands);
end;
```

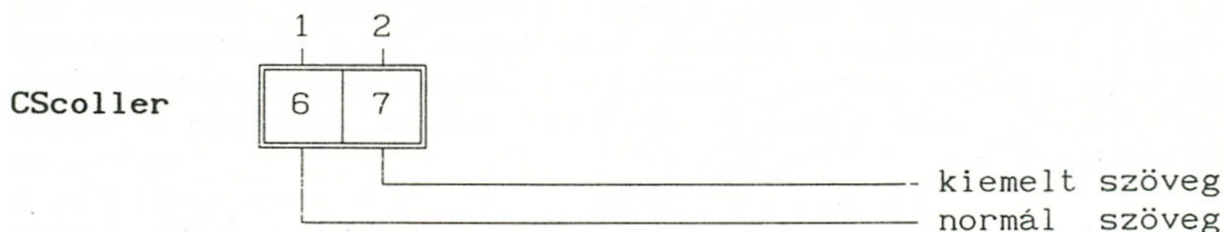
7.2.9. A *view* szinének beállítása

A Turbo Vision színpaletták használatával lehetővé teszi, hogy a *view*-k szineit meg lehessen változtatni.

A paletta egy sztringben van tárolva, ami tulajdonképpen egy változtatható hosszúságú tömböt jelent. Példaként tekintsük a *TScoller* objektum alapértelmezés szerinti palettáját:

```
CScoller= #6#7;
```

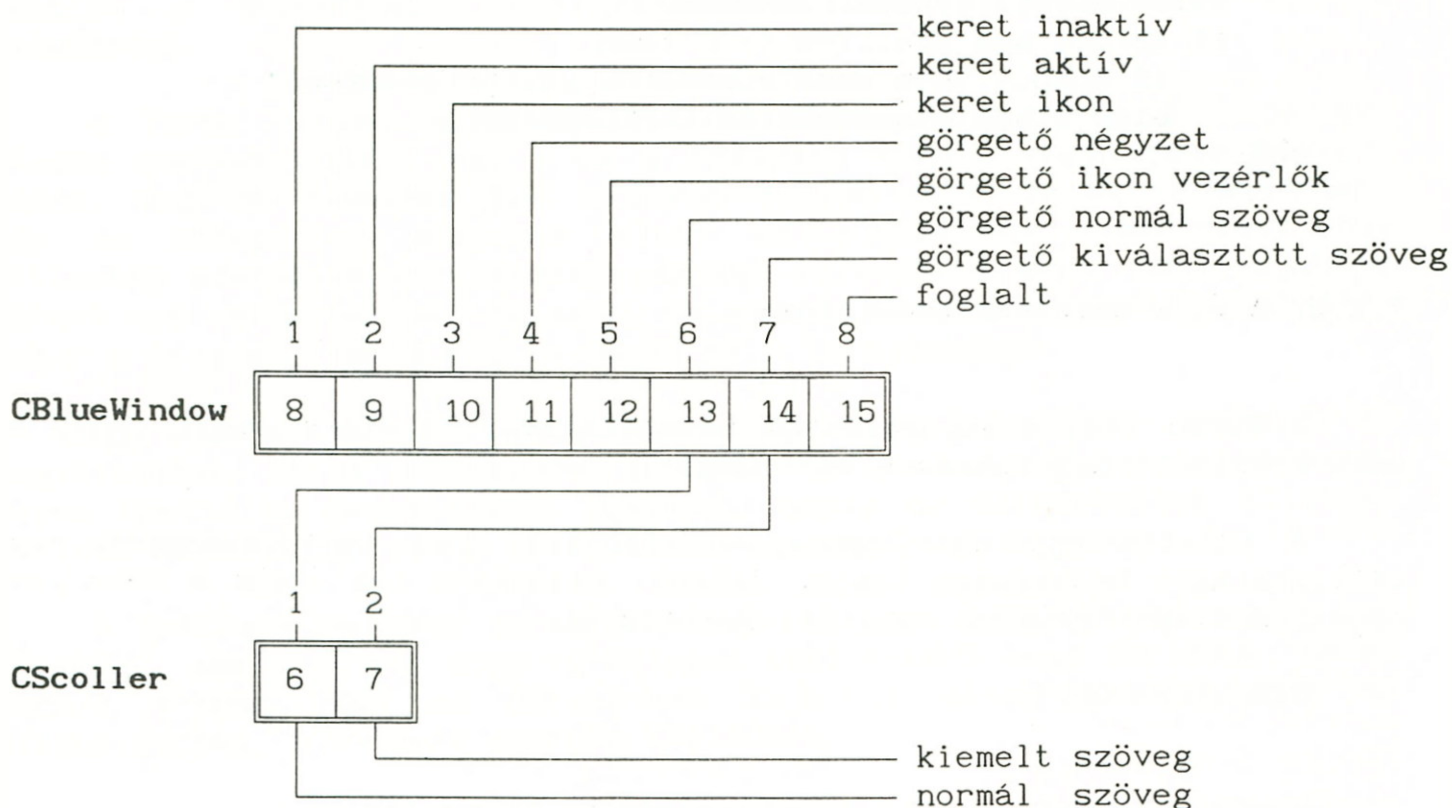
A palletta definíció a következőképpen ábrázolható:



Ez azt jelenti, hogy kétféle szöveget tud a szöveggörgető megjeleníteni. Az alapértelmezés szerinti színeket a paletta elemek határozzák meg. A normál szöveg kiírásához a *Draw* metódusnak el kell végeznie a *GetColor(1)* hívást, hogy megkapja az 1. palettaelem által definiált színt, ill. a *GetColor(2)* hívást a kiemelt szöveg színének ismeretéhez.

Az alapértelmezés szerinti színek úgy vannak összeválogatva, hogy tetszőleges képernyőtartalom esetén az egyes *view*-k jól elkülönülnek egymástól. Ezek felhasználásához semmilyen további ismeret nem szükséges. Ha azonban szeretnénk saját színeket, saját palettákat definiálni, elengedhetetlen a további fogalmak tisztázása.

A palettaelemek nem színkódokat tartalmaznak, hanem indexeket a tulajdonos *view* palettájához. Ha a szöveggörgető (*scroller*) egy ablakra illeszkedik, akkor a *TScoller* és a *TWindow* objektumok palettáinak viszonyát a következő ábra szemlélteti.



A *TWindow* palettájának 6. eleme a 13, ami szintén egy index az ablak tulajdonosának, a *desktop*-nak a palettájához, amely ugyancsak egy indexet tartalmaz az alkalmazói program palettájához. A *desktop*-nak egy **nil** palettája van, azaz a *desktop* szintjén a szinek nem változtathatók meg. Ez a **nil** paletta úgy működik, hogy minden palettaelem a saját sorszámát tartalmazza.

Az alkalmazói program rendelkezik palettával, a legnagyobbal, hisz minden Turbo Vision objektum számára tartalmaz elemeket. Ennek a palettának a 13. eleme a $1E. Az alkalmazói program a lánc végén található, mivel nincs tulajdonosa. Így a $1E már nem index, hanem szövegattribútum - 1 a háttér színe és $E (14) az előtér színe.

Láthattuk, hogy elég nehézkes az indexláncolaton végighaladva eljutni egy színkódig. Ezért a fenti feladat elvégzésére a *TView* tartalmaz egy *GetColor* függvényt, ami a indexláncon végighalad egészen

az alkalmazói program szintjéig, és a kívánt attribútum byte-tal tér vissza. A *Getcolor* függvény a lánc minden szintjén meghívja az adott objektumhoz tartozó *GetPalette* metódust, amely által visszaadott paletta infomációit felhasználva lép tovább a láncon.

Miután megismertük a paletták láncolatának működését, most már foglalkozhatunk az alpértelmezés szerinti színek átdefiniálásával.

Kézenfekvő megoldásnak tűnik a színek cseréjére a paletta lecserélése. Ebben az esetben nem szabad megfeledkezni arról, hogy a *subview*-k szintjén nem színek, hanem indexek vannak a palettában. Az indexeket természetesen fel lehet cserélni, de ezzel valószínűleg a képernyőn a csoport bizonyos elemei azonos színűvé válnak.

A *view* palettájának lecserélésére a *GetPalette* metódus átdefiniálása kínál megfelelő megoldást. Szeretnénk létrehozni egy új görgető objektum típust, amely a normál szövegszín helyett az ablak keretszínét használja kirajzoláskor. A szükséges deklarációs és implementációs lépéseket az alábbiakban láthatjuk.

```
type
    TMyScroller = object(TScroller)
       function GetPalette: PPalette; virtual;
    end;


function TMyScroller.GetPalette: PPalette;
  const
    CMyScroller = #1#7;
    PMyScroller: string[Length(CMyScroller)] = CMyScroller;
  begin
   GetPalette := @PMyScroller;
  end;
```

További lehetőségként nézzük meg, hogyan lehet új színekkel bővíteni a *TWindow* objektum típust. Tegyük fel, hogy a görgető objektumban három elemet tartalmazó palettát szeretnénk használni. Ehhez a *TWindow* objektumból származtatnunk kell egy új *TMyWindow* objektumot, ill. a *TMyScroll.Getpalette* metódust is módosítanunk kell. A megoldás főbb lépései:

```
type
    TMyWindow = object(TWindow)
       function GetPalette: PPalette; virtual;
    end;
```

```
function TMyWindow.GetPalette: PPalette;
  const
    CMyWindow = CBlueWindow + #84;
    P: string[Length(CMyWindow)] = CMyWindow;
  begin
   GetPalette := @P;
  end;
```

Ezek után a *TMyWindow* a paletta kilencedik pozícióján tartalmazza az új palettaelemet, amire a görgetőnek van szüksége. Egyetlen lépés maradt hátra, a görgető *Getpalette* metódusának módosítása:

```
function TMyScroller.GetPalette: PPalette;
  const
    CMyScroller = #6#7#9;
    PMyScroller: string[Length(CMyScroller)] = CMyScroller;
  begin
   GetPalette := @PMyScroller;
  end;
```

8. ESEMÉNY-VEZÉRELT PROGRAMOZÁS

A Turbo Vision alapvető célja az, hogy keretrendszert biztosítson felhasználói programok írásához. A programozónak csak arra kell összpontosítania, hogy hogyan töltse fel tartalommal ezt a vázat. A Turbo Vision két fő eszköze, az ablaktechnika támogatása és az események kezelése. Az előző fejezetben a *view*-k használatával ismerkedtünk meg. Ebben a fejezetben pedig azzal foglalkozunk, hogy hogyan lehet a programunkat az események köré csoportosítva felépíteni.

Először is ismerkedjünk meg az esemény-vezéreltség fogalmával. Egy hagyományos Pascal programban általában egy ciklust írunk arra a célra, hogy a billentyűzetről, az egérről vagy bármely más input eszközről adatokat fogadjunk, majd a beolvasott adatok függvényében elágaztassuk a programunkat.
Tekintsünk meg egy ilyen feladatot ellátó programrészletet:

```
repeat
      B := readkey;
      case B of
            'i': Invertalas;
            'e': Editalas;
            'g': Grafika;
            'q': Quit := true;
      end;
until Quit;
```

Egy esemény-vezérelt program szerkezete sem különbözik lényegesen az fenti példától. Hisz nehéz elképzelni olyan interaktív programot, ami ne ezt a megoldást használná. Azonban a programozó szemszögéből nézve alapvető eltéréseket fogunk tapasztalni.

Turbo Vision alkalmazásban nem kell többé a felhasználó inputját olvasnunk, ezt elvégzi helyettünk a keret-rendszer. A Turbo Vision összerakja a bejövő adatokat egy Pascal rekordba. Ezt *esemény*nek nevezzük. A Turbo Vision ezt az esemény rekordot továbbítja a programban a megfelelő *view*-hoz. Ez azt jelenti, hogy a programunknak csak azt kell tudnia, hogy hogyan bánjon a lényeges input adatokkal, és nem kell a bejövő adatok közül keresgélni a megfelelőt. Tegyük fel, hogy a felhasználó az egérrel rámutat egy inaktív ablakra. A Turbo Vision beolvassa az egérműveletet, melynek felhasználásával felépíti az esemény rekordot, majd ezt az esemény rekordot továbbítja az inaktív ablakhoz.

A *view*-k többsége képes a felhasználói inputot önállóan lekezelni. Az ablak tudja, hogyan kell megnyílni, lezáródni, elmozdulni és aktivizálódni stb.. A menü tudja, hogyan kell megnyílni, kommunikálni a felhasználóval és lezáródni. A vezérlő gomb tudja, hogyan kell

aktivizálódni, kommunikálni a többi gombbal és színét megváltoztatni. Egy inaktív ablak aktívvá tudja tenni önmagát mindenféle programozói beavatkozás nélkül.

Felvetődik a kérdés, miben áll ezek után a programozó feladata ?

Definiálni fogunk új *view*-kat, új műveletekkel, amelyeknek ismerni kell bizonyos általunk definiált eseményeket. Meg fogjuk tanítani az általunk létrehozott *view*-kat a szabványos parancsok feldolgozására vagy esetleg saját parancsok ("üzenetek") generálására további *view*-k számára. Ez a mechanizmus a Turbo Vision-ban készen a rendelkezésünkre áll.

8.1. Az események természete

Az eseményeket úgy a legjobb elképzelni, mint kis információ csomagokat, amelyek által leírt diszkrét esetekre az alkalmazói programnak reagálni kell. Minden billentyűlenyomás, minden egérművelet és a program bizonyos részei által generált állapotok különálló eseményeket képeznek. Az eseményeket nem lehet kisebb egységekre bontani, így ha a felhasználó begépel egy szót, az nem egy eseményt, hanem egyedi események (billentyűlenyomások) sorozatát jelenti.

Azt gondolhatnánk, hogy a Turbo Vision objektum-orientált világában az események szintén objektumként jelentkeznek. Ez azonban nem így van! Az események önmagukban semmilyen műveletet sem végeznek, csak információt továbbítanak az objektumainknak, ezért rekord struktúrában tárolódnak.

Minden esemény magját egy *What* nevű, *word* típusú mező képezi. A *What* mező numerikus értéke azonosítja a bekövetkezett eseményt, és az esemény rekord további részei jellemző információkat tárolnak erről az eseményről. Például a billenyűlenyomás esemény esetén a lenyomott billentyű scan kódja, egér események esetén az egér pozíciója és az egér gombok állapota tárolódik.

8.1.1. Az események fajtái

Nézzük meg kicsit közelebbről az *Event.What* lehetséges értékeit. A Turbo Vision-ban az események négy alapvető osztályba sorolhatók: egér (*mouse*) események, billentyűzet (*keyboard*) események, üzenet (*message*) esemény és üres (*nothing*) esemény. Az események minden fajtájához találunk egy előredefiniált konstanst - maszkot, amely segítségével gyorsan meg lehet állapitani az esemény típusát. Például, egér esetén

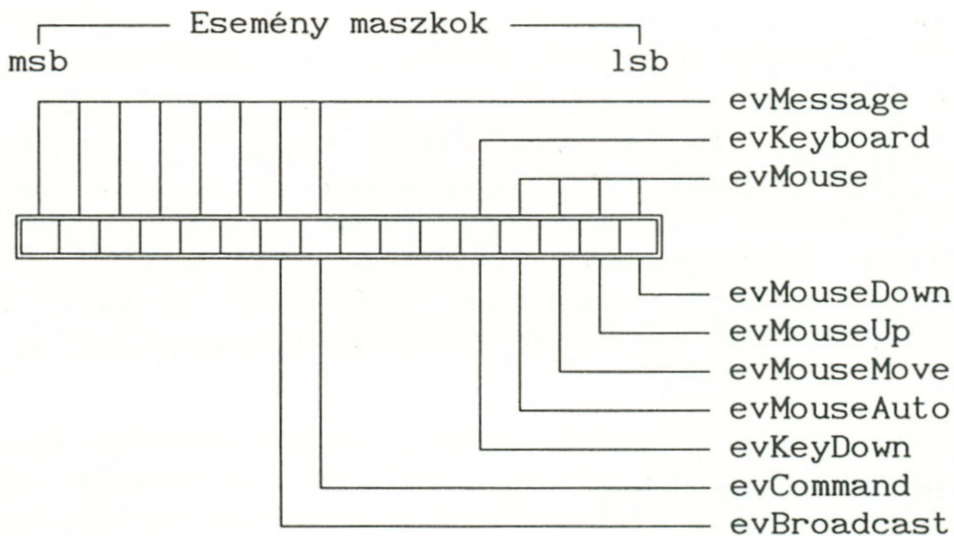
nem szükséges az összes lehetséges okot tesztelni,

```
if Event.What and (evMouseDown or evMouseUp or
                   evMouseMove or evMouseAuto) <> 0 then ...
```

hamen elegendő az *evMouse* maszkot használni

```
if Event.What and evMouse <> 0 then ...
```

A felhasználható maszkok: *evNothing* (üres esemény), *evMouse* (egér esemény), *evKeyboard* (billentyűzet esemény) és *evMessage* (üzenetek). Az események maszkolásához használható konstansok által azonosított bitek az alábbi ábrán láthatóak.



Egér események

Alaphelyzetben négy fajtáját különböztetjük meg az egér által generált eseményeknek:

- az egér gombjának lenyomása, vagy felengedése,
- az egér pozíciójának megváltozása,
- automatikus egér esemény.

Ha lenyomjuk az egér gombját, akkor egy *evMouseDown*, ha felengedjük a lenyomott gombot, egy *evMouseUp* esemény keletkezik. Az egér mozgatása az *evMouseMove* eseményt hozza magával. Ha folyamatosan lenyomva tartjuk az egér gombját az *evMouseAuto* esemény fog periódikusan generálódni, lehetővé téve a programnak bizonyos ismétlődő műveletek (pl. görgetés) végrehajtását. Minden esetben az esemény rekord tartalmazza azt az egér pozíciót ahol az esemény generálódott.

Billentyűzet események

A billentyűzet által létrehozott események sokkal egyszerűbbek. Amikor lenyomunk egy billentyűt a Turbo Vision az *evKeyDown* eseményt generálja, amely feljegyzi, hogy melyik billentyű volt lenyomva.

Üzenet események

Az üzenet eseményeket három csoportba sorolhatjuk - parancsok, adás és felhasználói üzenetek. Az egyes csoport sajátosságait a későbbiekben tárgyaljuk részletesen. A *What* mezőnek *evCommand* bitje jelzi a parancsokat, az *evBroadcast* az adásokat és felhasználó által definiálható konstansok a felhasználói üzeneteket.

Üres események

Az üres eseményt a rendszer arra használja, hogy egy esemény megszűnését jelezze. Ha a *What* mező csak a *evNothing* értéket tartalmazza, akkor tudnunk kell, hogy nincs az esemény rekordban semmi használható információ. Amikor egy Turbo Vision objektum befejezi valamely esemény feldolgozását, akkor mindig meghívja a *ClearEvent* metódust, amely a *What* mezőt az *evNothing* értékre állítja. Az objektumoknak egyszerűen figyelmen kívül kell hagyni ezt az esemény.

8.1.2. Események és parancsok

A legtöbb esemény feldolgozása valamilyen parancs létrehozásával ér véget. Nézzünk erre egy példát. Ha az egérrel a státuszsor egy elemére klikkentünk, akkor keletkezik egy egér esemény. Ezt az eseményt megkapja a státuszsor objektum és válaszol rá egy parancs esemény generálásával. Ha az egérrel az *ALT-X Exit* elemet választottuk ki, a generált parancs a *cmQuit*, amit a Turbo Vision program úgy értelmez, hogy mindent le kell zárni és be kell felyezni a működést.

8.2. Az események irányítása

A Turbo Vision *view*-k az "Akkor beszélj csak, ha megszólítottak" elven működnek. Ez azt jelenti hogy a *view* nem figyeli folyamatosan az inputot, hanem passzívan várakozik az esemény-kezelőre, aki, miután az esemény bekövetkezett, megmondja, hogy milyen eseményre kell a *view*-nak válaszolnia.

Az események helyes kezeléséhez nem elég ismernünk, hogy hogyan kell válaszolni egy bizonyos eseményre, hanem meg kell értenünk azt is, hogy hogyan jut el az esemény a kívánt *view*-hoz. Az események megfelelő helyre való beérkezésének kulcsa, az események helyes irányítása.

8.2.1. Honnan érkeznek az események ?

A Turbo Vision alkalmazások fő működési ciklusa a *TApplication* objektum *Run* metódusa, ami a *TGroup.Execute* rutint hívja. A *TGroup.Execute* eljárás tulajdonképpen egy ciklus, melynek felépítése az alábbi programrészlettel érzékeltethető.

```
var E: Event;
E.What := evNothing;          { jelzi, hogy nem érkezett esemény }
repeat
  if E.What <> evNothing then EventError(E);
  GetEvent(E);                { az esemény rekord összeállítása  }
  HandleEvent(E);             { az esemény irányítása a megfelelő
                                helyre                           }
until EndState <> Continue;
                              { ismétlés, amíg nem kell kilépni  }
```

A *GetEvent* metódus megnézi, hogy történt-e valami, aminek kapcsán eseményt kell generálni. Ha igen, a *GetEvent* felépíti a megfelelő rekordot. A következő lépésben a *HandleEvent* feladata, hogy az eseményt a megfelelő *view*-hoz irányítsa. Ha az eseményt nem dolgozták fel (nem törölték), az visszaérkezik a fenti ciklushoz, és meghívódik az *EventError* metódus. Ez *EventError* rutin semmit nem csinál, csupán jelzi, hogy egy *elhagyott* eseményről van szó.

8.2.2. Hová irányítódnak az események ?

Minden esemény irányítása az aktuális *modal view*-val kezdődik, ami az esetek többségében maga az alkalmazói objektum. Ha egy *modal* dialógus dobozban vagyunk, akkor természetesen ez az aktuális *modal view*. Mindkét esetben a *modal view* indítja el az esemény lekezelését. Az, hogy innen az esemény merre megy tovább, az függ az esemény természetétől.

Az események, fajtájuktól függően, három különböző módon juthatnak el a megfelelő *view*-hoz. Ezek a következőek: helyzeti (*positional*), fókuszált (*focused*) és szétszórt (*broadcast*).

Helyzeti események

A helyzeti eseményeket gyakorlatilag minden esetben az egér generálja (*evMouse*). Mint láttuk, első lépésben a *modal view* kapja meg az eseményt, amely azután - a *Z-order* alapján - a saját *subview*-jai között keresi azt, amely területén belül generálódott az esemény. Ha talált ilyen *subview*-t, akkor a *modal view* átadja neki az eseményt. Mivel azonban a *view*-k fedésben is lehetnek, több *view* is tartalmazhatja a keresett pozíciót. A *Z-order* szerint haladva, garantálható, hogy az adott pontban mindig a legfelül elhelyekedő *view* fogadja az eseményt.

A folyamat mindaddig folytatódik, amíg az objektum nem talál más olyan *view*-t, amelynek az eseményt továbbadhatná. Ennek két oka is lehet. Vagy a talált *view* egy végső *view* (nincsenek benne *subview*-k), vagy egyszerűen az adott pozíciót semelyik *subview* sem tartalmazza. Ezen a ponton biztosan elérkeztünk ahhoz az objektumhoz, amelyhez a helyzeti esemény tartozik és elkezdődik az esemény feldolgozása.

Fókuszált események

A fókuszált események általában billentyűlenyomások (*evKeyDown*) vagy parancsok (*evCommand*). Eltérően az előző csoporttól, ezek az események a fókusz láncon lefelé haladva adódnak át. Nézzük meg lépésről lépésre ezt a folyamatot.

Tehát először az aktuális *modal view* kapja meg az eseményt, amely azonnal továbbítja a hozzá tartozó kiválasztott *subview*-hoz. Ha ez a *subview* szintén rendelkezik kiválasztott *subview*-val, akkor az esemény ennek adódik tovább. A folyamat mindaddig folytatódik, amíg el nem jutunk egy végső *view*-hoz, a fókuszált *view*-hoz. Ez a fókuszált *view* fogadja és feldolgozza az eseményt.

Ha a fókuszált *view* nem tud mit kezdeni az eseménnyel, az elindul vissza a fókusz láncon, egészen a lánc tetején található tulajdonosig. Ha a visszaérkezett eseményt a *modal view* sem tudja lekezelni, meghívja az *EventError* metódust, és egy *elhagyott esemény* keletkezik.

A fókuszált események elvét érdemes billentyűzet eseménnyel illusztrálni. Vegyük például a Turbo Pascal fejlesztői környezetét, ahol egyidejűleg több file is lehet nyitva editor ablakokban a *desktop*-on. Ha lenyomunk egy billentyűt, tudjuk, hogy melyik file-ba fog a karakter bekerülni. Most kövessük nyomon, hogy ez a számunkra egyszerű műveletet, milyen lépésekkel végzi el a Turbo Vision.

Egy billentyű lenyomása az *evKeyDown* eseményt kelti életre, amely az aktuális *modal view*-hoz, a *TApplication* objektumhoz kerül. A *TApplication* tovább küldi az eseményt a kiválasztott *view*-hoz, a *desktop*-hoz, ami továbbítja azt az aktív ablakhoz. Az editor ablak az egyetlen lehetséges, alapértelmezés szerint kiválasztott *view*-nak, a

belvilágnak (*interior*) adja át az eseményt. A belső részhez nem tartoznak további *subview*-k, tehát ez fogja azt feldolgozni.

Szórt események

Szórt eseményeket vagy az adás események (*evBroadcast*) vagy a felhasználó által definiált üzenetek generálhatnak.

A szórt események nem irányítottak, hanem alapértelmezés szerint az aktuális *modal view*-hoz tartozó minden *subview*-hoz eljutnak.

Az aktuális *modal view* megkapja az eseményt, majd átadja azt a *subview*-knak *Z-order* sorrendben. Ha valamelyik *subview* egy csoport, akkor ez szintén továbbítja az esemény a csoport tagjaihoz, és szintén a *Z-order* alapján. A folyamat minaddig folytatódik, amíg minden *view*-hoz közvetlenül vagy közvetve el nem jut az esemény.

A szórt eseményeket leggyakrabban *view*-k közötti kommunikációra használjuk.

Felhasználó által definiált események

A Turbo Vision alkalmazásának bizonyos szintjén, szükségünk lehet arra, hogy saját magunk definiáljunk új eseményeket. Ezt az igényünket támogatja az, hogy az esemény rekord *What* mezőjében vannak szabad bitek (10.-15.). Ezekre a bitekre definiált események alapértelmezés szerint, mint szórt események kerülnek továbbításra.

A Turbo Vision tartalmaz egy olyam mechanizmust, amely lehetővé teszi, hogy a felhasználói események mint helyzeti vagy akár mint fókuszált események adódjanak át. A rendszer definiálja a *Positional* és *Focused* maszkokat, annak megfelelően, hogy az *What* mezőben jelölt események közül melyek irányítódnak mint helyzeti, és melyek mint fókuszált események. Alaphelyzetben a *Positional* az *evMouse*-t bitjeit, a *Focused* az *evKeyboard* bitjeit tartalmazza. Ilyen módon tetszőleges esemény átadási mechanizmusa az alapértelmezéstől eltérő módon is működtethető, a maszkok megfelelő bitjeinek átállításával.

8.2.3. Események tiltása

Minden *view* objektum rendelkezik egy *EventMask* nevű bittérkép mezővel, amely meghatározza, hogy mely eseményeket fogja a *view* lekezelni. A bitek *EventMask*-ban megfelenek a *TEvent.What* mezőben található biteknek. Ha egy eseményfajtának megfelelő bit 1, akkor a *view* fogadja és feldolgozza az adott típusú eseményt - ellenkező esetben figyelmen kívül hagyja azt.

8.2.4. A fázis

Bizonyos esetekben szükségünk lehet arra, hogy fókuszált eseményeket nem fókuszált *view*-k kezeljenek le. Például, ha egy görgethető szövegmegjelenítő ablakban vagyunk, nem használhatjuk a billentyűzetet a szöveg lapozására, hisz ha a szövegablak egy fókuszált *view*, a billentyűzet események őhozzá érkeznek be és nem a görgető négyzetekhez, amelyek képesek görgetni a szöveget.

A Turbo Vision biztosít számunkra egy olyan mechanizmust is, amellyel a fenti probléma megoldható, vagyis nem fókuszált *view*-k képesek fókuszált eseményeket fogadni és feldolgozni. Jóllehet a "Fókuszált események" részben ismertetett irányítási eljárás korrekt, létezik két kivétel a szigorú értelemben vett fókusz-lánc irányítás alól.

Abban az esetben, ha a *modal view*-hoz egy fókuszált esemény érkezik a továbbítása három fázisban zajlik le:

- Az esemény először eljut (a *Z-order* alapján) minden olyan *subview*-hoz, amelynél az *ofPreProcess* mezőjének *Options* jelzőbitje egyes értékkel rendelkezik.

- Ha a fenti eljárás során az esemény nem törlődött, továbbítódik a fókuszált *view*-hoz.

- Ha az esemény még mindig nincs törölve, akkor továbbítódik (ismét a *Z-order* alapján) minden olyan *subview*-hoz, amelynek a *ofPostProcess Options* jelzőbitje egy értékű.

Folytatva az előbbi példánkat, azt szeretnénk, hogy a görgető ikon (*scrool bar*) fogadja a billentyűlenyomásokat. Ehhez csak annyi szükséges, hogy a görgető ikont az *ofPreProcess* jelzőbit egyes értékével inicializáljuk.

A Phase adatmező

Minden csoport rendelkezik egy *Phase* nevű adatmezővel, amely a következő három érték valamelyikét tartalmazza: *phFocused*, *phPreProcess* és *phPostProcess*. A *Tgroup.Phase* segítségével egyszerűen megvizsgálható, hogy a csoporton belül a fókuszált események feldolgozása a fókuszált irányítás előtt, alatt, vagy után zajlik le.

Végezetül nézzünk egy egyszerű példát a fázisok adta lehetőségek felhasználására. Tekintsünk egy olyan dialógus dobozt, amely tartalmaz egy input sort és egy vezérlő gombot "**A**ll right" címkével ellátva. Az **A** az ún. *shortcut* (közvetlen) billentyű a gomb aktivizálására. Általában

nincs szükség arra, hogy a fázisokkal külön foglalkozzunk. A legtöbb vezérlő alaphelyzetben az *ofPostProcess* bit egyes értékével inicializálódik. Ez lehetővé teszi azt, hogy az **A** betű lenyomásakor a fókusz az "All right" gombra kerüljön.

Ha az input sor van a fókuszban, a billentyűnyomásokat az input sor kezeli le. Így ha lenyomjuk az **A** betűt, nem a vezérlő gomb aktivizálódik, hanem mindig megjelenik az **A** betű az input sorban. Az első gondolatunk a probléma megoldására, hogy az *ofPreProcess* opciót kellene használni az inicializáláskor. Ha ezt tennénk, akkor a gombot ugyan mindig elérnénk, csak éppen A betűt nem tudnánk beírni az input sorba. Mi hát a megoldás ?

A megoldás nagyon egyszerű: Használjunk a különböző fázisokban különböző közvetlen billentyűket. A PreProcess fázisban használjuk az **ALT+A**, a PostProcess fázisban pedig az **A** billentyűlenyomást. Ez a magyarázata annak, hogy dialógus dobozban miért tudunk mindig ALT+betű közvetlen billentyűket használni.

Ily módon mind a *ofPreProcess* és mind a *ofPostProcess* biteket be kell állítanunk. Nézzünk egy olyan programrészletet, ami a *HandleEvent* metódus részeként használható, és mint ahogy láthatjuk, a vezérlő gomb csak bizonyos billentyűlenyomásokat ellenőriz, annak függvényében, hogy a fókuszált vezérlés megkapta-e már az eseményt, vagy sem. A példában jól elkülönül az egyes fázisokhoz tartozó feltételek felépítése.

```
evKeyDown:                         { ez egy case utasítás része   }
  begin
    C := HotKey(Title^);
    if (Event.KeyCode=GetAltCode(C))     { preprocess fázisban  }
                    or
       (Owner^.Phase=phPostProcess) and (C<>#0) and
       (Upcase(Event.CharCode) = C)      { postprocess fázisban }
                    or
       (State and sfFocused <> 0)   and
       (Event.CharCode = ' ') { a fókuszban                     }
    then
       begin
         PressButton;
         ClearEvent(Event);
       end;
  end;
```

8.3. A parancsok

A legtöbb helyzeti és fókuszált esemény feldolgozása során a kezelő objektum valamilyen parancsot hoz létre. Vagyis egy objektum gyakran válaszol pl. az egérrel való klikkentésre vagy billentyű-lenyomásra parancs esemény generálásával.

Például, ha a státuszsor valamely elemét kiválasztjuk az egérrel, generálunk egy helyzeti eseményt. Az alkalmazás meghatározza, hogy az esemény a státuszsor területén belül generálódott, így átadja az esemény kezelését annak.

A státuszsor megállapitja, hogy melyik státusz vezérlőelem területére esik az esemény és beolvassa az adott elem állapotrekordját. Az elem általában parancsokkal vezérelhető, ezért a státuszsor a *What* mezőt az *evCommand* értékre állítja és a *Command* mezőn beállítja a szükséges parancsot az elem számára. Ezek után törli az *evMouse* területet, így a *GetEvent* metódus az éppen generált parancs eseményt fogja érzékelni, mint következő eseményt.

8.3.1. Parancsok definiálása

A Turbo Vision-ban számos előredefinált parancs található, de adott a lehetőség saját parancsok előállítására is. Ha új *view*-t származtatunk, szükséges, hogy egy parancsot is definiáljunk, amivel az új *view*-t előhívhatjuk. A parancs neve tetszőleges lehet, ajánlott azonban a Turbo Vision konvencióit követni, azaz minden parancs a "cm" betűkkel kezdődik. Másrészt, mint a példából is látható a parancs mint konstans szerepel a Pascal-ban.

```
const
  cmValamiparancs = 100;
```

A rendszer lefoglalja 0-99 és 256-999 tartományokon definiált parancsokat. Az programunk használhat saját parancsokat a 100-255 és 1000-65535 közötti tartományban. A tartományok megosztásának, az oka az, hogy a parancsok közül csak a 0-255 értékkel rendelkezőket lehet letiltani. Ilyen módon a Turbo Vision rendelkezik letiltható és le nem tiltható parancsokkal egyaránt.

8.3.2. Parancsok kötése

Ha létrehozunk egy menü elemet vagy egy státuszsor elemet, akkor ezzel egy parancsot is hozzájuk kötünk. Amikor aztán a felhasználó kiválasztja a kívánt elemet, generálódik egy esemény rekord, amelynek a *What* mezője az *evCommand* értéket tartalmazza, a *Command* mezője pedig a hozzákötött parancsot. A parancs lehet akár Turbo Vision szabványos parancs, vagy akár egy általunk definált parancs is. Amikor egy parancsot a menü vagy a státuszsor valamely eleméhez kötünk, egyidejűleg köthetjük azt egy *hot key*-hez is, ami által lehetővé válik a parancs billentyűlenyomással való kiváltása.

Fontos megjegyeznünk, hogy egy parancs definiálása önmagában nem írja le azt a műveletsort, amit a programunknak végre kell hajtania, amikor a parancs megjelenik az esemény rekordban. Mindig nekünk kell közölni az objektumokkal, hogy hogyan kell válaszolni az adott parancsra.

8.3.3. Parancsok engedélyezése és tiltása

Vannak esetek, amikor szükséges, hogy bizonyos parancsok adott ideig, a felhasználó által ne legyenek elérhetőek. Például, ha nincs nyitott ablak a *desktop*-on, nincs értelme a *cmClose* (ablak lezáró) parancs kiadásának. A Turbo Vison lehetőséget biztosít parancsok halmazának tiltására ill. engedélyezésére. Erre a célra tartalmaz egy globális típust *TCommandSet*, amely tulajdonképpen egy számhalmaz 0..255 közötti elemekkel. Az alábbi példa bemutatja, hogyan kell letiltani néhány ablakkezelő parancsot.

```
var
   WindowCommands: TCommandSet;
begin
   WindowCommands := [cmNext, cmPrev, cmZoom, cmResize, cmClose];
   DisableCommands(WindowCommands); { a fenti 5 parancs tiltása,}

   { bizonyos műveletek }

   EnableCommands(WindowCommands);  { majd engedélyezése         }
end;
```

8.4. Események kezelése

Ha létrehoztunk egy parancsot és létesítettünk néhány vezérlést ami ezt generálja - pl. egy menü elem vagy dialógus doboz gombja -, akkor meg kell tanítanunk a *view*-kat, hogy hogyan válaszoljanak a parancs megjelenésére.

Minden *view* örökli a *HandleEvent* metódust, amely alapértelmezésben tudja, hogyan kell válaszolni a legtöbb felhasználói inputra. Ha azonban a programunkban a *view*-t valamilyen speciális tevékenység elvégzésére szeretnénk használni, szükséges a *HandleEvent* metódus átdefiniálása. Az új *HandleEvent* metódust két dologra mindig meg kell tanítanunk: hogyan válaszoljon az általunk definiált új parancsokra, és hogyan válaszoljon az egér és a billentyűzet eseményekre.

A *view*-k *HandleEvent* metódusa meghatározza a *view* viselkedését. Ha két *view*-nak megegyezik a *HandleEvent* metódusa, akkor ezek teljesen azonos módon fognak reagálni a különböző eseményekre. Amikor származtatunk egy új *view*-t, általában azt szeretnék, hogy a származtatott *view* többé-kevésbé az előd *view*-hoz hasonlóan viselkedjen, és csak néhány kisebb módosítást hajtunk végre rajta. Messze legegyszerűbb megoldás ennek programozására, ha a származtatott objektum *HandleEvent* metódusából meghívjuk az előd objektum *HandleEvent* metódusát.

Az elmondottak alapján, egy utód *HandleEvent* metódusa valahogy így épülhet fel:

```
procedure NewUtod.HandleEvent (var Event: TEvent)
begin
    { kód az előd objektum viselkedésének módosítására,
      vagy kiküszöbölésére                                }

    Elod.HandleEvent(Event);           { az előd hívása  }

    { kód további funkciók végrehajtásához                }
end;
```

Más szavakkal, ha azt akarjuk, hogy egy új objektum bizonyos eseményeket az elődjétől eltérően kezeljen, akkor ezeket az eseményeket az előd *HandleEvent* metódusának hívását megelőzően "csapdába kell ejtenünk". Ha azonban azt szeretnénk, hogy az új objektum úgy viselkedjen mint az elődje, csak további funkciókkal kibővítve, akkor a szükséges kódot az előd *HandleEvent* metódusának hívása után kell elhelyeznünk.

8.5. Az esemény rekord

Eddig a pontig elméleti oldalról tárgyaltuk az esemény-vezérelt programozással összefüggő ismereteket. Az következő részekben a szükséges gyakorlati megoldásokkal fogunk megismerkedni.

Az első és legfontosabb kérdés, hogyan is épül fel az a sokat emlegetett esemény rekord. A DRIVERS unit tartalmaz egy változó hosszúságú rekord típust, *TEvent*-et, amely alapvető szerepet játszik a Turbo Vision eseménykezelő stratégiájában.

```
TEvent = record
   What: Word;
   case Word of

     evNothing: ();

     evMouse:   (
       Buttons: Byte;
       Double: Boolean;
       Where: TPoint
                 );

     evKeyDown: (
       case Integer of
         0: (KeyCode: Word);
         1: (CharCode: Char; ScanCode: Byte)
                 );
     evMessage: (
       Command: Word;
       case Word of
         0: (InfoPtr: Pointer);
         1: (InfoLong: Longint);
         2: (InfoWord: Word);
         3: (InfoInt: Integer);
         4: (InfoByte: Byte);
         5: (InfoChar: Char)
                 );
 end;
```

A rekordban jól elkülönülnek (üres sorokkal vannak tagolva) az egyes eseményfajtákhoz tartozó definíciók.

A *TEvent* egy változó hosszúságú rekord. Így például ha a *TEvent.What* a *evMouseDown* értéket veszi fel, a rekord további része a

```
Buttons : Byte;
Double  : Boolean;
Where   : TPoint
```

mezőket fogja tartalmazni.

Érdemes egy pillantást vetnünk a rekord utolsó mezőjére. Itt az üzenet fajtájú események esetén, tárolódik a parancs ill. a parancshoz kapcsolódóan valamilyen *Pointer*, *Longint*, *Word*, *Integer*, *Byte* vagy *Char* típusú adatmező. A *view*-k képesek önmaguk is eseményeket generálni és elküldeni azt más *view*-hoz, felhasználva ehhez a rekord *InfoPtr* nevű mezejét.

8.5.1. Események törlése

Amikor egy *view HandleEvent* metódusa befejezte az esemény feldolgozását, meghívja a *ClearEvent* metódust. A *ClearEvent* az *Event.What* mezőnek a *evNothing* értéket adja és az *Event.InfoPtr* pointer értékét a *@Self*-re állítja, jelölve ezzel azt, hogy az esemény lekezelése megtörtént. Így, ha ezek után az esemény egy újabb objektumhoz kerül, az egyszerűen figyelmen kívül fogja hagyni azt.

A *ClearEvent* metódus segíti a *view*-k egymás közötti kommunikációját is. Nem szabad arról elfelejtkeznünk, hogy az események feldolgozása a Turbo Vision rendszer számára addig nem ér véget, amíg meg nem hívjuk a *ClearEvent* metódust.

8.5.2. Elhagyott események

Normális esetben a felhasználói programunkban minden eseményt lekezelnek a *view*-k. Ha egy *view* sem hajlandó az eseménnyel foglalkozni, a *modal view* meghívja az *EventError* metódust. Az *EventError* meghívja a tulajdonos *EventError* metódusát, és így tovább egészen a *TApplication.EventError* rutinig.

A *TApplication.EventError* metódus alapértelmezés szerint semmit sem csinál. Érdekes lehetőség, ha az *EventError* metódust átdefiniáljuk, a program fejlesztése során valami visszajelzéssel a felhasználó felé, ki lehet szűrni a fel nem használt eseményeket.

8.6. Az események mechanizmusának módosítása

Az aktuális *modal view* működésének magját az alábbi kis programrészlet szemlélteti.

```
var
  E: TEvent;
begin
  E.What := evNothing;
  repeat
    if E.What<> evNothing then EventError(E);
    GetEvent(E);
    HandleEvent(E);
  until EndState <> Continue;
end;
```

8.6.1. Események centralizált összegyűjtése

Az esemény-vezérelt programozás egyik legnagyobb előnye, hogy az általunk írt programrésznek nem kell tudnia, hogy honnan érkeznek az események. Például az ablak objektumnak csak azt kell tudnia, hogy ha egy esemény megérkezik hozzá a *cmClose* paranccsal, akkor be kell záródnia. Számára közömbös, hogy az eseményt egy klikkentés okozta az ablakzáró dobozon menüből való kiválasztás során, vagy egy *hot key* esetleg egy másik objektum generálta. Egyetlen, amit tudnia kell, hogy kapott egy eseményt, amit fel kell dolgoznia, és amennyiben képes rá, meg is teszi ezt.

A kulcs ezekhez a "*black box*" (fekete doboz) eseményekhez, az alkalmazói objektum *GetEvent* metódusa. A *GetEvent* a programunk egyetlen olyan része, amely kapcsolatot tart fent az események forrásával. A programunkban az objektumok egyszerűen meghívják ezt a metódust, amely szolgáltatja az egér, a billentyűzet vagy más objektumok által generált eseményeket.

Ha újfajta eseményeket kell előállítanunk (pl. karakterek olvasása a soros portról), a saját felhasználói objektumunkban egyszerűen átdefiniáljuk a *TApplication.GetEvent* metódust. A *TApplication* objektum a *GetEvent* metódusát a *TProgram* objektumtól örökli.

```
procedure TProgram.GetEvent(var Event: TEvent);
var
  R: TRect;

function ContainsMouse(P: PView): Boolean; far;
begin
  ContainsMouse := (P^.State and sfVisible <> 0) and
    P^.MouseInView(Event.Where);
end;

begin                              { függőben levő esemény }
  if Pending.What <> evNothing then
```

```
      begin
        Event := Pending;
        Pending.What := evNothing;
      end else
      begin
        GetMouseEvent(Event);                        { egér esemény }
        if Event.What = evNothing then
        begin
          GetKeyEvent(Event);               { billentyűzet esemény }
          if Event.What = evNothing then Idle;
        end;
      end;

      if StatusLine <> nil then
        if (Event.What and evKeyDown <> 0) or
          (Event.What and evMouseDown <> 0) and
          (FirstThat(@ContainsMouse) = PView(StatusLine)) then
          StatusLine^.HandleEvent(Event);
    end;
```

Látható, hogy a *GetEvent* végigpásztázza a lehetséges eseményforrásokat, majd ha nincs sehol sem kész esemény meghívja az *Idle* (üres) rutint. Az új eseményforrás beillesztése elvégezhető akár az *Idle* metódus, vagy akár magának a *GetEvent* metódus átdefiniálásával. A megoldáshoz érdemes egy rutinhívást, pl. a *GetComEvent(Event)*-et használni, amely képes kell legyen a karakter beolvasására a soros portról ill. az esemény rekord felépítésére.

8.6.2. A GetEvent metódus átdefiniálása

Az aktuális *modal view GetEvent* metódusa meghívja a tulajdonos *GetEvent* metódusát, és így tovább egészen a *view* fa tetején található *TApplication.GetEvent* rutinig, amely felfogja az éppen bekövetkező eseményt.

Mivel az események érzékelésével csak a *TApplication.GetEvent* metódus foglalkozik, a saját almazásunkhoz elegendő ezt átdefiniálni. Például billentyűzet makrók feldolgozásához figyelnünk kell a *GetEvent* által visszaadott eseményeket, ezek közül el kell kapnunk a szükségeseket és szét kell bontanunk a makrót. Az alkalmazás többi része csak azt fogja érzékelni, hogy egy eseménysorozat érkezett a felhasználótól.

A szükséges lépéseket az alábbi eljárásdefiníció tartalmazza.

```
procedure TMyApp.GetEvent(var Event: TEvent);
begin
  TApplication.GetEvent(Event);
  { itt következhet a szükséges lépéssorozat }
end;
```

8.6.3. Az üresjárati (*Idle*) idő felhasználása

Egy másik előnye a *TApplication.GetEvent* központi szerepének az, hogy ha nincs készen használható esemény , meghívja a *TApplication.Idle* metódust. A *TApplication.Idle* tulajdonképpen egy üres (*dummy*) metódus, amely átdefiniálásával az aktuális *view*-val párhuzamosan (konkurensen) valamilyen más tevékenységet is végre tudunk hajtani.

Nézzünk erre a megoldásra is egy példát. Szeretnénk definiálni egy *view*-t *THeapView* néven, melynek az *Update* metódusának meghívásakor megjelenik a képernyőn az éppen szabad heap memória mérete. Ha átdefiniáljuk a *TApplication.Idle* metódust az alábbival, a felhasználó folyamatosan nyomon követheti a képernyőn a heap terület méretének változásait, bárhol is vagyunk a programban.

```
procedure TMyApp.Idle;
begin
  HeapViewer.Update;
end;
```

8.7. *View*-k közötti kommunikáció

Turbo Vision program objektumokba van összeolvasztva és mi is objektumok belsejében írjuk a programkódot. Tegyük fel, hogy a programon belül egy objektum valamely más objektummal szeretne információt cserélni. Hagyományos programozási nyelvekben ez egyszerűen az adatok másolását jelenti egyik adatszerkezetből egy másikba. Az objektum-orientált nyelven ez már nem ilyen egyszerű, mivel az objektumok nem biztos hogy tudják, hogy hol helyezkednek el a többi objektumok.

A *view*-k közötti kommunikáció nem olyan egyszerű, mint adatok küldése a Pascal program ekvivalens részei között. Nem beszélve arról, hogy a hagyományos Pascal alkalmazás két része, semmilyen körülmények között sem képes funkcionálisan két Turbo Vision *view*-t megvalósítani.

Ha szükségünk van a *view*-k közötti kommunikáció működtetésére, az első kérdésünk az lesz, hogy helyesen osztottuk-e meg a feladatot (*task*-ot) két *view* között. Ez az osztottság lehet a gyenge programtervezés eredménye, és talán jobban járunk, ha a két *view*-t valahogy egybeépítjük, vagy bizonyos részeiket áthelyezzük egymásba.

8.7.1. Közvetítők használata

Ha a programtervezés csakugyan kifogástalan, és szükséges, hogy a *view*-k minden más *view*-val kommunikáljanak, a lehetséges helyes út, ha létrehozunk egy közvetítő *view*-t.

Például tegyük fel, hogy van egy táblázatkezelő és egy szövegszerkesztő objektumunk, és a táblázatkezelőből bizonyos adatokat szeretnénk áttenni a szövegszerkesztőbe ill. vissza. Turbo Vision alkalmazói programban ez közvetlenül megvalósítható a *view*-to-*view* (*view*-tól *view*-nak) kommunikációval. Ha azonban szeretnénk egy harmadik objektumot - pl. egy adatbázis kezelőt - is bekapcsolni az adatcserébe, meg kell dupláznunk az első két objektum között létesített kommunikációt mind a három objektum között.

Ekkor már sokkal jobb az a megoldás, ha létesítünk egy közvetítő *view*-t, ebben az esetben pl. egy ún. "üzenőtáblát" (*clipboard*-ot), mint ahogy azt a Turbo Pascal fejlesztői környezete is teszi. Ezek után egy objektumnak csak azt kell tudnia, hogy hogyan kell valamit bemásolni a *clipboard*-ba, ill. hogyan kell onnan valamit kivenni. Nem beszélve arról, hogy ezek után semmilyen gondot sem jelent újabb objektum bekapcsolása a kommunikációba.

8.7.2. *View*-k közötti üzenetek

Ha a helyzetünket gondosan elemeztük, és biztosak vagyunk a programtervezés kifogástalan voltában, és nincs szükségünk közvetítő *view*-k használatára, akkor alkalmazhatjuk a két *view* közötti kommunikációt.

Mielőtt az egyik *view* kommunikálna a másikkal, szükséges megkeresni a másik *view*-t és ellenőrizni, hogy az létezik-e egyáltalán.

Nézzünk egy "élő" példát a kommunikációra. Az *Stddlg* unit tartalmaz egy dialógus dobozt *TFileDialog* (ez az a *view*, amely akkor jelenik meg az integrált fejlesztői környezetben, ha új file-t kívánunk betölteni). A *TFileDialog* tartalmaz egy *TFileList* objektumot, amely a lemez tartalomjegyzékét jeleníti meg, és egy *TFileInputLine* objektumot, amely az aktuálisan kiválasztott file nevét tartalmazza. Mindig, amikor

a felhasználó a *FileList*-ben egy újab file-t kiválaszt, közölni kell a *FileInputLine* objektummal az új file nevét.

Ebben az esetben a *FileList* biztos lehet abban, hogy a *FileInputLine* létezik, mivel mind a ketten egyanazon objektumon (*FileDialog*) belül kerültek inicializálásra. Hogyan közli a *FileList* a *FileInputLine*-nak a kiválasztott file nevét ?

A *FileList* létrehoz egy üzenetet és elküldi azt. Tekintsük a *TFileList.FocusItem* metódust, amely elküldi az üzenetet és a *TFileInputLine.HandleEvent* metódust, amely fogadja az üzenetet.

```
procedure TFileList.FocusItem(Item: Integer);
begin
  TSortedListBox.FocusItem(Item);  {az öröklött metódus hívása}
  { az üzenet elküldése}
  Message(Owner, evBroadcast, cmFileFocused, List^.At(Item));
end;

procedure TFileInputLine.HandleEvent(var Event: TEvent);
var
  Dir: DirStr;
  Name: NameStr;
  Ext: ExtStr;
begin
  TInputLine.HandleEvent(Event);
  if (Event.What = evBroadcast) and
     (Event.Command = cmFileFocused) and
     (State and sfSelected = 0) then
  begin                  { az üzenet fogadása, és feldolgozása }
     if PSearchRec(Event.InfoPtr)^.Attr and Directory <> 0 then
        Data^ := PSearchRec(Event.InfoPtr)^.Name + '\'+
          PFileDialog(Owner)^.WildCard
     else Data^ := PSearchRec(Event.InfoPtr)^.Name;
     DrawView;
  end;
end;
```

A *Message* függvény generál egy üzenet eseményt és visszaad egy pointert, amely arra az objektumra mutat, amelyik lekezelte az eseményt (ha van ilyen). Látható, hogy a TFileList.Focusitem a Turbo Pascal kibővített szintaxisát használja (**$X+**), mivel a *Message* függvényt eljárásként hívja meg.

8.7.3. Ki kezeli le a szórt (*broadcast*) eseményeket?

Tegyük fel, hogy a programban valamilyen műveletsor elvégzése előtt szeretnénk megállapitani, hogy van-e már ablak nyitva a *desktop*-on. Honnan lehet ezt megtudni? A válasz az, hogy használjunk szórt eseményeket. Küldjünk ki egy szórt eseményt, amelyre az ablak tudja a választ. Ez az esemény eljut minden *view*-hoz, és ha van olyan *view*, ami az eseményt lekezeli, megkapjuk annak az azonosító pointerét.

Lássunk egy konkrét példát. A Turbo Pascal fejlesztői környezetében, ha a felhasználó egy Watch ablak megnyitását kéri, a nyitást végző programrészlet először ellenőrzi, hogy van-e már nyitva Watch abblak. Ha még nincs, akkor elvégzi a megnyitást, de ha már van, akkor azt az előtérbe hozza. Nézzük sorjában a szükséges lépéseket.

Az adás (*broadcast*) üzenet kiküldése:

```
AreYouThere := Message (Desktop, evBroadcast, cmFindWindow, nil);
```

A Watch ablak *HandleEvent* metódusa a *cmFindWindow* parancsra az esemény törlésével válaszol:

```
case Event.Command of
  ...
  cmFindWindow: ClearEvent(Event);
  ...
end;
```

A *ClearEvent* metódus a *What* mező *evNothing*-val való feltöltése mellett, az *InfoPtr* mezőt beállítja a *@Self* értékre. A *Message* függvény abban az esetben, ha az eseményt lekezelték, az *InfoPtr* mező tartalmával tér vissza, tehát visszadja annak az objektumnak a címét, amely az eseményt feldolgozta. Nézzük tovább a példánkhoz kapcsolódó utasításokat.

```
if AreYouThere = nil
 then
   CreateWatchWindow       { ha nincs Watch ablak - létrehozza,}
else
   AreYouThere^.Select;    { ha van, úgy az előtérbe hozza azt.}
```

Amennyiben csak egy objektum van, ami tudja, hogy hogyan kell a *cmFindWindow* adásra válaszolni, addig teljesen biztos, hogy mindig csak egy Watch ablakot fog használni a rendszer a *desktop* előterében.

8.7.4. A *HandleEvent* metódus meghívása

Adott a lehetőség, hogy egy esemény létrehozása vagy módosítása után közvetlenül meghívjuk a *HandleEvent* metódust, az alább ismertetett módokon.

- Egy *view*-ból közvetlenül meghívhatjuk az egyenrangú *subview* *HandleEvent* metódusát. (Két *subview* akkor egyenrangú (*peer*), ha azonos a tulajdonosuk.) Az esemény így nem terjed át más *view*-khoz, hanem egyenesen a másik *HandleEvent* metódus dolgozza fel, majd a vezérlés visszakerül a hívás helyére.

- Egy *view*-ból meghívhatjuk a tulajdonos *HandleEvent* metódusát. Ebben az esetben az esemény a *view* láncon lefelé haladva fog terjedni, és az esemény lekezelése után a vezérlés visszakerül a hívó *view*-hoz. (Figyelni kell a rekurziós hívások elkerülésére.)

- Adott a lehetőség hogy egy más *view* láncon található *view* *HadnleEvent* metódusát hívjuk meg. Az esemény azon a *view* láncon lefelé haladva fog terjedni. Az esemény feldolgozása után a vezérlés visszakerül a hívás helyére.

8.7.5. Szövegösszefüggéstől függő HELP (*context sensitive*)

A Turbo Vision beépített eszközökkel rendelkezik, ahhoz, hogy a felhasználói programokban szövegösszefüggéstől függő (*context-sensitive*) Help-et használjunk. Lehetőség van arra, hogy egy *view*-hoz hozzárendeljünk egy help környezeti indexet. A Turbo Vision biztosítja azt, hogy bármikor, amikor a *view* fókuszálttá válik, ez a help környezeti index lesz a alkalmazás aktuális help-jének az indexe.

Globális *context-sensitive* help szintén kialakítható, egy *HelpView* objektum létrehozásával. Amikor a *HelpView* aktivizálódik (általában az F1 billentyű lenyomására), akkor meg kell tudnia a tulajdonostól az aktuális help környezeti indexet a *HelpView.GetHelpCtx* metódus hívásával. Ezek után a *HelpView* meg tudja jeleníteni az aktuális help szöveget.

Valószínűleg a szövegösszefüggéstől függő help az, amit legutoljára építünk be a programunkba. A Turbo Vision objektumok a *hcNoContext* help környezeti index-szel inicializálódnak, ami azt jelenti, hogy az objektumhoz nem tartozik help környezeti index. Ha mégis szükségünk van a help használatára a *SetHelpCtx* rutinnal a *view*-k *HelpCtx* adatmezejét megfelelően feltölthetjük.

9. TURBO VISION NEM LÁTHATÓ ELEMEI

9.1. Stream-ek

Az objektum orientált programozási technika és a Turbo Vision biztosítja a kód és az adat egybezárását, az objektumok struktúrái közötti kapcsolatok kialakítását.

Lehetőségünk van-e az objektumok háttértárolóra történő elmentésére?

Az objektumban lévő adatokat külön tudjuk választani és kimenthetjük egy file-ba, de jobb lenne, ha az egész objektumot egy egységként tudnánk kezelni. Ezt biztosíthatja számunkra a stream.

A Turbo Vision stream az objektumok egy kollekciója, ami lehet egy DOS file, az EMS memória, soros port, vagy valamilyen más eszköz.

Egy Pascal programozó tudja, hogy mielőtt file-műveletet hajtana végre, közölnie kell a fordítóval, hogy milyen típusú adatokat kell írnia vagy olvasnia a file-ból. A file-hoz tartozik egy típus, amit már a fordítási időben meg kell határozni.

A *BlockRead*, *BlockWrite* lehetőséget ad típussal nem rendelkező állományok kezelésére, de a programozónak különös figyelmet kell fordítania a típus hiányából adódó problémák lekezelésére.

A Turbo Pascal nem engedi meg objektum típussal rendelkező file létrehozását, mert az objektumok tartalmazhatnak virtuális metódusokat, amelyek címe a futási időben kerül meghatározásra. A virtuális metódus táblában (VMT) lévő információk elmentése értelmetlen.

A Turbo Vision stream lehetőséget ad ezen nehézségek leküzdésében és még egyéb előnyöket is biztosít.

9.1.1. A stream sokrétűsége

A Turbo Vision stream által kezelhetünk típussal rendelkező és nem rendelkező file-okat egyaránt. A típusellenőrzés még mindig megvan, de amit tárolni szeretnénk egy adatot, annak a típusát nem kell ismernünk a fordítási időben. Ennek az az oka, hogy a stream-ek képesek a *TObject*-ből leszármaztatott objektumokat kezelni.

A stream-ek számára definiálni kell, melyik objektumot akarjuk feldolgozni és az ismerni fogja a kapcsolatot a virtuális metódus táblával. Igy az objektumok elmenthetők és visszaállíthatók lesznek.

Ugyanaz a stream hogyan tud különböző típusú objektumokat írni és olvasni, hiszen ezeket a fordítási időben még nem határoztuk meg? Ez nagy mértékben különbözik a hagyományos Pascal file I/O-jától.

A választ egy szóba sűríthetjük. Ez a *regisztrálás*. Minden Turbo Vision objektum típushoz, és a hierarchiából származó újabb típushoz hozzárendelődik egy regisztrációs szám. Ez a szám elsőként kerül felírásra a stream-en elhelyezett objektum adatai között. Amikor elkezdődik egy objektum visszaállítása a Turbo Vision elsőként ezt a regisztrációs számot olvassa ki. Igy ismertté válnak számára az adatok kiolvasásához szükséges információk.

9.1.2. A stream megnyitása

Hogy valamilyen műveletet végezhessünk egy stream-en, először inicializálnunk kell azt. Az *Init* konstruktor szintakszisa különböző lesz, attól függően, hogy milyen típusú stream kerül megnyitásra.

A DOS stream megnyitásához meg kell adnunk egy DOS filenevet és hozzáférési módot (csak írható, csak olvasható, írható-olvasható). A következő példában egy DOS stream puffert inicializálunk egy objektum betöltéséhez:

```
var
  SaveFile: TBufstream;
begin
{ file-név, stream elérési mód (R/W), puffer méret }
  SaveFile.Init('PELDA.DSK', stOPen, 1024);
  ...
```

A *TStream* típus egy absztrakt stream, amely nem használható fel csak a belőle származtatott típusok.

```
TDosStream - lemez I/O
TBufStream - pufferelt lemez I/O
             (hasznos, ha sok kis méretű egységet
              akarunk a lemezre írni vagy olvasni)
TEmsStream - egy stream, amely az objektumokat az
             EMS által kezelt memóriában helyezi el
             (hasznos az erőforrás file gyors fel-
             dolgozásánál)
```

9.1.3. Irás és olvasás a stream-en

A *TStream* három alapvető metódust használ:

Get, Put, Error.

A *Get* és a *Put* hasonlít a file-kezelésben eddig használt *Read* és *Write* eljárásokhoz. Az *Error* eljárás pedig akkor kerül meghívásra, ha hiba jelentkezik a stream kezelése közben.

9.1.3.1. A Put eljárás

Általános szintakszisa:

```
SomeStream.Put(PSomeObject);
```

ahol a *SomeStream* a *TStream*-ből származtatott, megnyitott stream. A *PSomeObject* a *TObject*-ből származó regisztrált objektum.

A stream meg tudja állapitani a *PSomeObject*-ről (feltéve, hogy ez a típus regisztrálva van), hogy mi az azonosító száma és mennyi adat következik az azonosító után a file-ban.

Egy csoport elmentése esetén a csoporthoz tartozó *view*-k mindegyike automatikusan tárolásra kerül *desktop*-ként.

A program egy állapota is elmenthető, így a későbbi visszaállítás során az elmentéskor fennállt feltételek között folytatódhat a program futása.

9.1.3.2. A *Get* eljárás

A *Get* eljárás segítségével egyszerűen visszaállíthatunk egy objektumot a stream-ről. Meghívása:

```
PSomeObject:=SomeStream.Get;
```

ahol a *SomeStream* ismét egy inicializált Turbo Vision stream és a *PSomeObject* egy pointer egy Turbo Vision objektumhoz. A *Get* eljárás által visszaadott pointer típusa a stream adott pozicióéjában lévő objektum által meghatározott; nem a *PSomeObject* típusa a döntő.

9.1.3.3. Hibakezelés

Az *Error* eljárás segítségével megállapítható, hogy milyen hiba következett be a stream kezelése során. A *TStream.Error* két értéket ad vissza a *Status* és az *ErrorInfo* mezőkön keresztül. Ha a futási időben fellépő hibákat dialógus doboz segítségével akarjuk kezelni, felül kell írnunk az *Error* eljárást.

9.1.4. A stream lezárása

Ha befejeztük a stream használatát meg kell hívnunk a *Done* metódust, ami tartalmaz egy *Close* eljáráshívást. Ez ugyanúgy tehetjük meg, mint bármely más Turbo Vision objektumnál:

```
Dispose(SomeStream, Done);
```

9.1.5. Az objektumok és a stream-ek

Minden szabványos Turbo Vision objektum kezelhető a stream-en és minden Turbo Vision stream kezelhet szabványos objektumokat. Egy szabványos típusból származtatott új objektum stream-en történő kezelése előkészítést igényel.

9.1.5.1. Betöltés és tárolás

Az objektumok írása és olvasása a stream-en két metódus meghívásával történik: *Load* és *Store*. Minden egyes objektumnak rendelkeznie kell ezekkel a metódusokkal, de ezek közvetlenül nem kerülnek meghívásra. (Öket a *Get* és *Put* eljárás hívja meg.) Ez a programozáskor egyszerűsödik, mert a legtöbb mechanizmus egy ős objektumból származik.

Például egy új *view* származtatása a *TWindow*-ból az alábbi típusdefiníciót igényli:

```
type
  TM = object(TWindow)
    Painted: Boolean;
    constructor Load(var S: TStream);
    procedure Draw;
    procedure Store(var S: TStream);
  end;
```

Ennél a definíciónál egy **Boolean** típusú mezőt csatoltunk a *TWindow* típushoz.

Az objektum kiírásánál, illetve betöltésénél kezelnünk kell egy szabványos *TWindow* objektumot és még egy byte-ot, ami a **Boolean** mezőnek felel meg. A *Load* és *Store* metódusok a következőképpen néznek ki:

```
constructor TM.Load(var S: TStream);
begin
  TWindow.Load(S);
  S.Read(Painted, SizeOf(Boolean));
end;

procedure TM.Store(var S: TStream);
begin
  TWindow.Store(S);
  S.Write(Painted, SizeOf(Boolean));
end;
```

Az adatok visszatöltését a kiírás sorrendjében kell elvégezni. A beolvasott adatok mennyiségének pedig egyeznie kell a kiírt adatmennyiséggel. Fontos, hogy a programozó betartsa ezeket a szabályokat, mert a fordítás során nem kap hibajelzést.

Egy objektum módosítása során valószínűleg felül kell írnia a *Load* és a *Strore* metódusokat is.

9.1.5.2. A regisztrálás

A Turbo Vision minden objektumhoz definiálja a stream-ek regisztrációs rekordját. Minden egyes Turbo Vision unit tartalmaz egy RegisterXXXX eljárást, ami automatikusan regisztrálja a unit-ban lévő objektumokat. (A regisztrálást a *TApplication.RegisterType* eljáráshívás váltja ki.)

Egy új objektum definiálása során meg kell változtatnunk a *Store* és a *Load* metódusokat, és regisztrálnunk kell egy objektumot. A regisztrálás egy két lépésből álló folyamat. Definiálni kell a stream regisztrációs rekordját és át kell adni azt a globális *RegisterType* eljárásnak.

A stream regisztrációs rekord egy *TStreamRec* típusú Pascal rekord:

```
PStreamRec = ^TStreamRec;
TStreamRec = record
               ObjType: Word;
               VmtLink: Word;
               Load   : Pointer;
               Store  : Pointer;
               Next   : Word;
             end;
```

A konvenció szerint a Turbo Vision regisztrációs rekordja a T betű helyett R betűvel kezdődik. Például a *TDeskTop* helyett *RDeskTop*-ot használnak.

9.1.5.3. Az objektum azonosítása

A rekord mezői közül legfontosabb az *ObjType* típusú mező. Minden új típus definiálása során szükséges, hogy egy egyedi azonosító számot rendeljünk az objektumhoz. A Turbo Vision által a szabványos objektumokhoz már foglaltak 0-tól 99-ig a számok. A felhasználó saját regisztrációs számának 100 és 65535 közé kell esnie.

9.1.5.4. Az automatikus mezők

A *VmtLink* mező összekapcsolja az objektumot és a virtuális metódus táblát. Ez könnyen megtehető az objektum offszet címének a lekérdezésével.

```
RSomeObject.VmtLink := Ofs(TypeOf(TSomeObject)^);
```

A *Load* és *Store* mezők tartalmazzák az új objektum *Load* és *Store* metódusainak címét:

```
RSomeObject.Load := @TSomeObject.Load;
RSomeObject.Store := @TSomeObject.Store;
```

Az utolsó mező belső használatra van fenntartva a stream regisztrációs rekordjainak láncolásához.

A regisztrációs rekord feltöltése után meg kell hívni a *RegisterType* eljárást. Az előző példánál maradva ez a *TM* objektum esetében így néz ki:

```
const
  RM: TStreamRec = (
    ObjType: 100;
    VmtLink: Ofs(TypeOf(TM)^);
    Load   : @TM.Load;
    Store  : @TM.Store;
  );

  RegisterType(RM);
```

A fenti lépések elvégzése után az objektum elmenthető illetve visszaállíthatóvá válik a stream-ről.

9.1.6. A stream működési mechanizmusa

9.1.6.1. A *Put* eljárás

Első lépésként a regisztrációs rekordok listáján a stream rendszer megkeresi az elmentendő objektumhoz tartozó bejegyzést. Ha megtalálja a megfelelőt, kiírja a regisztrációs számot a stream által meghatározott helyre. Ezután meghívásra kerül a *Store* metódus, amely befejezi az objektum elmentését. A *Store* metódus ehhez felhasználja a stream *Write* eljárását, amelyik ténylegesen végzi az adatok kiírását.

Az objektumnak semmit nem kell tudnia a stream-ről - ami lehet egy lemezes file, vagy az EMS memória vagy más rendezett stream - a felmerülő problémák a stream által lekezelésre kerülnek.

9.1.6.2. Az olvasási folyamat

Az objektum olvasása a stream-ről a *Get* metódussal történik. Elsőként az azonosító szám kerül visszaállításra és a stream megvizsgálja, hogy a regisztrációs listán szerepel-e ez a típus. Ha megtalálta, a regisztrációs rekord felhasználásával és a *Load* metódus meghívásával visszatölti az objektumot. A stream rendszer vigyáz arra, hogy az objektum *Load* metódusa, - amelyik a *Read* metódust alkalmazza - megfelelő mennyiségű byte-ot olvasson vissza.

Ha a visszaolvasott azonosító értéke 0, a stream *NIL* pointert ad tovább. A nulla foglalt regisztrációs szám és nem használható egy objektum regisztrációs számaként.

9.1.7. A kollekciók és a stream

A kollekciók című fejezet tárgyalja, hogy a kollekció tartalmazhat különböző, de összefüggő objektumokat. A stream-ek sokrétűségi tulajdonságát felhasználva egy teljes kollekció tárolható a lemezen és máskor, egy másik program által ez visszaállítható.

Egy példán keresztül nézzük meg, hogy mit kell tudni a kollekciók tárolásáról.

Első lépésként hozzunk létre egy alap objektumot, ami grafikus objektumok őse lesz. Definiálni kell a *Store* metódusokat a *TGraphCircle* és a *TGraphRec* leszármaztatott objektumokhoz.

A *TGraphCircle* feladata egy kör rajzolása, a *TGraphRec* pedig téglalapot tud rajzolni. A feladatok elvégzéséhez bevezetett új mezők

az elsőnél a kör sugara (*Radius*), a második objektumnál pedig a szélesség (*Width*) és a magasság (*Height*). Az ős objektum tartalmaz egy *X* és *Y* mezőt, ami a leszármaztatott objektum egy jellegzetes pontja lehet.

A következő lépés a regisztrációs rekord felépítése minden objektumhoz, ami tárolásra kerül. Ezután regisztráltathatjuk az objektumainkat.

A továbbiakban úgy járunk el, mint a hagyományos file-kezelés esetén: stream változó deklarálása, a stream megnyitása, a kollekció kiírása (ez egy utasítással megtehető), a stream lezárása.

9.1.7.1. Az objektumok definíciói:

```
{ grafikus alapobjektum definiálása }
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
    X, Y: Integer;
    constructor Init;
    procedure Draw; virtual;
    procedure Store(var S: TStream); virtual;
  end;

{ pont származtatása az alapobjektumból }
  PGraphPoint = ^TGraphPoint;
  TGraphPoint = object(TGraphObject)
    procedure Draw; virtual;
  end;

{ grafikus kör származtatása az alapobjektumból }
  PGraphCircle = ^TGraphCircle;
  TGraphCircle = object(TGraphObject)
    Radius: Integer;
    constructor Init;
    procedure Draw; virtual;
    procedure Store(var S: TStream); virtual;
  end;

{ téglalap származtatása az alapobjektumból }
  PGraphRect = ^TGraphRect;
  TGraphRect = object(TGraphObject)
    Width, Height: Integer;
    constructor Init;
    procedure Draw; virtual;
    procedure Store(var S: TStream); virtual;
  end;
```

Az alábbiakban a *Store* metódusokat mutatjuk be. Ezek mindegyike tartalmazza az örökölt *Store* metódus meghívását, ami elvégzi az örökölt adatok kiírását. Ezután meghívásra kerül a stream-ek *Write* metódusa az objektum nem örökölt adatainak elmentéséhez. A *TGraphObject* nem hívja meg a *TObject.Store*-t, mert a *TObject* nem tartalmaz adatmezőket.

```
{ az ős, illetve a leszármaztatott objektumok Store virtuális }
{ metódusa }
  procedure TGraphObject.Store(var S: TStream);
  begin
    S.Write(X, SizeOf(X));
    S.Write(Y, SizeOf(Y));
  end;

  procedure TGraphCircle.Store(var S: TStream);
  begin
    TGraphObject.Store(S);
    S.Write(Radius, SizeOf(Radius));
  end;

  procedure TGraphRect.Store(var S: TStream);
  begin
    TGraphObject.Store(S);
    S.Write(Width, SizeOf(Width));
    S.Write(Height, SizeOf(Height));
  end;
```

A *TStream.Write* metódus bináris formátumot ír ki. Az első paramétere egy változó lehet, de ennek mérete nem ismert a *TStream* számára. A második paraméterrel megadhatjuk ezt az információt. Célszerű ehhez a szabványos *SizeOf* függvényt használni. Igy abban az esetben, ha az egész értékeket tartalmazó koordinátarendszerünket valósra cseréljük nem kell felülvizsgálnunk a *Store* metódusokat.

9.1.7.2. A regisztrációs rekordok

A regisztrációs rekord definiálása során meg kell adnunk egy egyedi azonosító számot. A Turbo Vision a 0-tól 99-ig terjedő értékeket lefoglalja a szabványos objektumok számára.

```
{ az előző részben definiált grafikus objektumok regisztrálásá-
  hoz szükséges rekordok felépítése típusos konstansokkal.     }
  const
    RGraphPoint: TStreamRec = (
      ObjType: 150;
      VmtLink: Ofs(TypeOf(TGraphPoint)^);
      Load: nil;
```

```
      Store: @TGraphPoint.Store);

    RGraphCircle: TStreamRec = (
      ObjType: 151;
      VmtLink: Ofs(TypeOf(TGraphCircle)^);
      Load: nil;
      Store: @TGraphCircle.Store);
    RGraphRect: TStreamRec = (
      ObjType: 152;
      VmtLink: Ofs(TypeOf(TGraphRect)^);
      Load: nil;
      Store: @TGraphRect.Store);
```

A regisztrációs rekordok *Load* pointere **nil** értéket tartalmaz, mert ez a példa csak az adatok tárolásával foglalkozik.

9.1.7.3. A regisztrálás

Fontos, hogy ezeket a rekordokat az első stream I/O művelet előtt a *RegisterType* eljárás által feljegyeztessük a stream rendszer számára. Célszerű ezt a program elején megtenni, vagy az alkalmazás *Init* metódusában.

```
procedure StreamRegistration;
begin
  RegisterType(RCollection);
  RegisterType(RGraphPoint);
  RegisterType(RGraphCircle);
  RegisterType(RGraphRect);
end;
```

A programozó felelőséggel tartozik azért, hogy minden objektum, amit a stream-en el akarunk helyezni regisztrálva legyen.

A normális file-kezeléshez tartozó műveletek maradtak hátra: a stream megnyitása, az adatok kiírása, a stream lezárása. A kollekció minden egyes elemének kiírásához nem kell használnunk a *ForEach* iterátort, mert a *Put* a teljes kollekciót elmenti.

```
var
  GraphicsList: PCollection;
  GraphicsStream: TBufStream;
begin
  StreamRegistration;
{ A grafikai objektumot tartalmazó Stream megnyitása. }
  ...
  GraphicsStream.Init('GRAPHICS.STM', stCreate, 1024);
{ kollekció kiírása }
  GraphicsStream.Put(GraphicsList);
```

```
{ A Steram lezárása. }
  GraphicsStream.Done;
  ...
end.
```

Igy létrehozott állomány minden szükséges információt tartalmazza a kollekció visszatöltéséhez. Amikor a stream-et megnyitjuk és betöltjük a kollekciót, rejtve marad minden kapcsolat a felhasználó elől, ami az objektumok és a virtuális metódus táblák között fennáll. Ugyanezt a technikát használja az Turbo Pascal integrált fejlesztői környezete (IDE) is, amikor létrehoz egy *desktop* file-t.

9.1.8. Hivatkozás subview-ra

Sokszor kellemes lokális változókban tárolni egy csoport *subview*-inak pointereit. Például a dialógus doboz gyakran tartalmazza a vezérlő objektumok pointereit könnyen megjegyezhető nevű mezőkben. Amikor egy *view* beszúrásra kerül a *view* fába, a tulajdonos két pointert birtokol. Az egyiket egy mezőben a másikat a *subview* listában. Ha nem vesszük figyelembe ezeket, az objektum stream-ről való visszaállítása során dupla hívatkozással állunk majd szembe.

A megoldás a *GetSubViewPtr* és a *PutSubViewPtr* eljárások alkalmazása. A *PutSubViewPtr* meghívása során tárolásra kerül a *subview* megfelelő pozíciója attól függően, hogy a csoporthoz tartozó *subview* listában hol helyezkedett el.

Hasonlóképp járunk el a stream-ről való visszaállítás során. A *Load* metódusnak tartalmaznia kell a *GetSubViewPtr* meghívását.

Az alábbi egyszerű példa ezt mutatja be:

```
type
  TButtonWindow = object(TWindow)
    Button: PButton;
    constructor Load(var S: TStream);
    procedure Store(var S: TSteam);
  end;


{ Metódus az objektum felolvasására a stream-ből. }
  constructur Load(var S: TStream);
  begin
    TWindow.Load(S);
    GetSubViewPtr(S, Button);
  end;
```

```
{ Metódus az objektum kitöltésére a stream-re. }
  procedure Store(var S:TStream);
  begin
    TWindow.Store(S);
    PutSubViewPtr(S, Button);
  end;
```

Az ablak elmentése után tárolásra kerül a *Button* mező hivatkozása is. A *Button* objektum, mint a Window *subview*-ja kerül kiírásra.

9.1.9. Hivatkozás egyenrangú *view*-kra

Hasonló eset áll elő, amikor egy *view* tartalmaz egy olyan mezőt, amely vele egyenrangú *view*-ra mutat. Az ugyanahhoz a csoporthoz tartozó *view*-kat egyenrangúaknak nevezzük. Egy kiváló példa erre a görgető. Ugyanaz a window tartalmaz egy görgető ikont és két görgető sort. A görgetőnek két mezője mutat a görgető sorokra. Ilyen esetekben a *TView PutPeerViewPtr* és a *GetPeerViewPtr* metódusaival érhetjük el, hogy a *view*-k a megfelelő pozícióba kerüljenek az objektumhoz tartozó *subview* listán.

9.1.10. A desktop tárolása és betöltése

Ha az elmentésre kerülő objektum a *desktop*, akkor minden a *desktop* környezetéhez tartozó *view* is elmentésre kerül.

Ehhez az szükséges, hogy minden lehetséges *view* tartalmazza a megfelelő *Store* és *Load* metódusokat és mindegyik regisztrálva legyen.

A következő programrészlet a *desktop* visszaállításában segíthet:

```
procedure TMyApp.RestoreDeskTop;
var
  SaveFile: TBufStream;
  Temp: PDeskTop;
begin

{ A létező stream megnyitása. }

  SaveFile.Init('T.DSK', stOpen, 1024);

{ a desktop felolvasása egy segédterületre. }

  Temp := PDeskTop(SaveFile.Get);
```

```
    { a stream lezárása }
      SaveFile.Done;
      if Temp <> nil then

      begin
        Dispose(DeskTop, Done); { a régi desktop lezárása }
        DeskTop := Temp;
        Append(DeskTop);       { az új desktop beépítése az
                                 programba }
        DeskTop^.DrawView;     { az új desktop megjelenítése }
      end;
      { stream hibakezelés }
      if SaveFile.Status <> 0 then ErrorReadingFile;

    end;
```

A *TStream CopyFrom(S, Count)* metódusa *Count* darab byte-ot másol át az S stream-ről. Használatára általában akkor kerül sor, ha a teljes stream tartalmát egy másik stream-en akarjuk elhelyezni. Például a lemezen lévő stream-et a gyorsabb elérés miatt áttesszük az EMS memóriába.

9.1.11. Véletlen hozzáférés a stream-en

Mindeddig úgy foglalkoztunk a stream-mel, mint egy szekvenciális elérésű eszközzel, de a Turbo Vision lehetőséget ad véletlen elérésű eszközként történő alkalmazásokra is. Analóg módon a hagyományos file-kezeléssel itt is megtalálhatók a *Seek, FilePos, FileSize* és a *Truncate* eljárások megfelelői.

A *Seek* feladata az aktuális stream-pointer mozgatása egy meghatározott pozícióra. A pozíciót byte-okban kell megadni a stream elejéhez viszonyítva.

A *GetPos* függvény fordítottja a *Seek* eljárásnak. Egy *Longint* értéket ad vissza, amely a stream-pointer aktuális pozícióját tartalmazza.

A *GetSize* függvény a stream méretét adja vissza byte-okban.

A *Truncate* eljárás az aktuális pozíciótól levágja a stream hátralévő részét és az aktuális pozíció lesz a stream vége.

Ezen lehetőségek kihasználása az erőforrás file-ok kezelésénél is jelentkezik.

9.2. Erőforrások

Az erőforrás (*resource*) file Turbo Vision objektumokat tartalmaz, amelyekre a későbbi alkalmazások során név szerint hivatkozhatunk. Ezeket az objektumokat nem szükséges a saját alkalmazói programunk által felépíteni, ezeket egy különálló program segítségével is létrehozhatjuk és az erőforrás file-ba menthetjük. A mechanizmusa egyszerű: az erőforrás file véletlen elérésű állományként kezelhető, ahol az objektumokat nevekkel azonosíthatjuk.

Az erőforrások használata során lehetőség nyílik az alkalmazói program működésének megváltoztatására anélkül, hogy program kódját módosítanánk. Például: a dialógus doboz szövege, menünevek, színek stb.

Az objektumok inicializálása gyakran összetett feladat, ami számításokat és egyéb műveleteket is tartalmazhat. Erőforrás file használata esetén ezek a problémák egyszerűsödnek. Az erőforrás file alkalmazása lehetővé teszi az alkalmazói program ország-specifikus karbantartását.

Lehetővé teszi azt, hogy az alkalmazói program különböző képességekkel rendelkezzen. Például két különböző menüt illeszthetünk hozzá, amelynél az egyik biztosítja a programban lévő összes lehetőség felhasználását, míg a másik korlátozza ezeket.

Az erőforrások működésének mélyebb megértéséhez a stream-ek és a kollekciók (*collections*) ismerete szükséges. A *TResourceFile* tartalmaz rendezett sztringeket és stream-et. A sztringek az egyes objektumokhoz tartozó kulcsok. A *TResourceFile* tartalmaz egy *Init* metódust a stream-ek feldolgozásához és egy *Get* metódust, amely egy kulcs segítségével visszaad egy objektumot.

Az erőforrás file-ok létrehozása négy lépésből áll: a stream megnyitása, az erőforrás inicializálása, egy vagy több objektum tárolása a stream-en, a file zárása. A következő példa egy egyszerű erőforrás file létrehozását mutatja be, ahol a file-ban egy státuszsort helyezünk el. A státuszsor kulcsa: 'stsor1'.

```
uses Drivers, Objects, Views, Menus;

type
  PHaltStream=^THaltStream;
  THaltStream=object(TBufStream)
                   procedure Error(Code, Info:Integer); virtual;
              end;

var
  MyRez:TResourceFile;
  MyStrm:PHaltStream;

{ Jelen példában a státuszsorban lesz az erőforrás }
```

```
procedure THaltStream.Error(Code, Info:Integer);
begin
  Writeln('Stream error:',Code,' (',Info,')');
  Halt(1);
end;

procedure CreateStatusLine;
var  StatusLine:PStatusLine;
     R:TRect;
begin
  {  a státuszsor felépítése }
  R.Assign(0,23,80,24);
  StatusLine:=New(PStatusLine,
                Init(R,
                     NewStatusDef(0,$ffff,
                     NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
                     NewStatusKey('~F3~ Open', kbF3, cmNO,
                     NewStatusKey('~F5~ Zoom', kbF5, cmYES,
                     NewStatusKey('~Alt-F3~ Close', kbAltF3, cmOK,
                     nil)))),
                    nil)
                  )
                );
   MyRez.Put(StatusLine,'stsor1'); { A státuszsor kiirása a stream-re. }
   Dispose(StatusLine,Done);        { A státuszsor megszüntetése.        }
end;

begin
  { Új stream megnyitása. }
  MyStrm:=New(PHaltStream, Init('my.rez', stCreate, 1024));
  MyRez.init(MyStrm);  { A stream inicialiáláza. }
  RegisterMenus;       { Az alapobjektumok regisztrálása a stream
                         műveletekhez. }
  CreateStatusLine;    { A státuszsor létrehozása. }
  MyRez.Done;          { A stream lezárása.        }
end.
```

Az erőforrás file-ból való visszatöltéshez a *TResourceFile*-hoz tartozó *Get* metódust kell használnunk. Ez a kulcs alapján egy általános *PObject* pointerrel tér vissza, ha sikeres volt az objektum kiolvasása. Abban az esetben, ha a kulcshoz nem érvényes, *NIL* pointert kapunk vissza.

```
uses Objects, Drivers, Views, Menus, App, Dialogs, Dos;
var
  MyRez:TResourceFile;{ TMyApp alkalmazói objektum csak egy statuszsort
tartalmaz. }
type
  PMyApp=^TMyApp;
  TMyApp = object(TApplication)
    constructor Init;
```

```
    procedure InitStatusLine; virtual;
  end;
      { A TMyApp konstruktor végzi el a státuszsor inicializálását. }
constructor TMyApp.Init;
const MyRezFileName:PathStr='my.rez';
var S:PStream;
    FileName:PathStr;
    Event:TEvent;
begin
  RegisterMenus; {Az alapobjektumok regisztrálása a stream műveletekre}
  { A stream megnyitása. }
  MyRez.Init(New(PBufStream, Init(MyRezFileName, stOpen, 1024)));
  { Stream hiba esetén kilépés a programból. }
  if  MyRez.Stream^.Status<>0 then Halt(1);
  { Ez a hívás fogja meghívni az InitStatusLine metódust. }
  TApplication.Init;
end;

procedure TMyApp.InitStatusLine;
begin
  { Státuszline feltöltése a stream-ről. }
  StatusLine:=PStatusLine(MyRez.Get('stsor1'));
end;

var
  MyApp: PMyApp;

begin      { A szokásos felhasználói programfelépités. }
  New(MyApp,Init);
  MyApp^.Run;
  MyApp^.Done;
end.
```

Egy objektum többször is visszaolvasható az erőforrás file-ból. Például gyakori a dialógus dobozok alkalmazása során az ismételt visszatöltés. Lassú I/O egység esetén gondoljunk az EMS memória használatára.

Lehetőség van sztring-listák kezelésére is a standard erőforrás mechanizmus mellett. Ilyenkor egész értékekkel történik az egyes sztringek azonosítása. Például a hibaüzenetek esetében a különböző országok szerint különböző erőforrás file-okban helyezhetjük el a számmal azonositandó üzeneteket.

9.3. Kollekciók

A Pascal programozók hagyományosan a legtöbb időt a programban lévő adatstruktúrák karbantartásával töltik. Például, ha az adat tárolása egy tömbben történik, akkor létre kell hozni ezt a tömböt, fel

kell tölteni adatokkal, fel kell dolgozni ezeket, esetleg ki kell menteni valamilyen I/O eszközre. Később, amikor a programnak egy új elemtípusra lenne szüksége mindent elölről kell kezdeni.

Jó lenne, ha az elemtípusok kiterjeszthetők lennének az eredeti programkód megváltoztatása nélkül. Ennek a megvalósítására törekszik a Turbo Vision *TCollection* típusa, amely az objektum és nem objektum típusú elemek tárolására és a tárolással kapcsolatos feladatok megoldására szolgál.

9.3.1. A kollekció (*collection*) objektum

A kollekció típusú objektumoknak két jellemzőjük van a hagyományos Pascal tömbbel szemben:

- a dinamikus méret,
- sokrétűség.

A Turbo Pascal tömböknél már a fordítási időben ismertnek kell lenni a tömb méretének. Sok esetben ez előre pontosan meghatározható és ennek megfelelő program készíthető. A tömb méretének megváltoztatása azonban a program módosítását és újrafordítását teszi szükségessé. A kollekciók használatánál, bár szükséges egy inicializáló méret, futási időben dinamikusan növelhető a bennük tárolt adatok száma. Igy sokkal rugalmasabb alkalmazás készíthető.

A Turbo Pascal tömböknek egy másik jellemzője, hogy az elemeknek azonos típussal kell rendelkezniük és ezt a típust már a fordítás során meg kell határozni. A kollekció objektum típussal nem rendelkező pointerek segítségével feloldja ezt a korlátot. Hatékonysága abban jelentkezik, hogy különböző típusú és méretű objektumokat (és nem objektum típusú adatokat) tartalmazhat. Hasonlóan a stream-ekhez nem kell tudnia semmit az általa kezelt objektumokról.

9.3.2. A típusellenőrzés és a kollekciók

A Pascal programozási nyelv szigorú típusellenőrzést ír elő, de ez a kollekciók alkalmazásával megkerülhető. Ez azt jelenti, hogy a kollekcióban tárolásra került PH1 objektumot PH2-ként is kiolvashatjuk anélkül, hogy figyelmeztetést kapnánk. Ezért szükséges, hogy a programozó kellő körültekintéssel kezelje a típusokat mielőtt tárolásra vagy visszaállításra kerülnének egy kollekcióból.

9.3.3. A nem objektum típusú elemek

A kollekcióban lehetőség van nem objektum típusok tárolására is, azonban néhány *TCollection*-höz tartozó metódus csak a *TObject*-ből származtatott adatot tud feldolgozni. Ez azt jelenti, hogy tárolható egy *PString* a kollekcióban, de ha ezt el akarjuk menteni egy stream-en, felül kell írnunk a kollekcióhoz tartozó *GetItem* és *PutItem* metódusokat. Ez a probléma a kollekcióhoz tartozó tárterület felszabadításakor is jelentkezhet. Ilyenkor a *FreeItem* eljárást kell újra definiálnunk.

9.3.4. Kollekció létrehozása

Egy kollekció létrehozása épp olyan egyszerű feladat, mint egy adattípus definiálása. Tegyük fel, hogy ügyfelek számlaszámát, nevét, telefonszámát kell tárolnunk illetve visszakeresnünk. Első lépésként az ügyfél *(TClient)* objektumot kell definiálnunk.

```
type
  PClient = ^TClient;
  TClient = object(TObject)
    Account, Name, Phone: PString;
    constructor Init(NewAccount, NewName, NewPhone: String);
    destructor Done; virtual;
  end;
```

A következő lépésben egy *Init* és egy *Done* metódust kell létrehoznunk, hogy memóriterületet tudjunk lefoglalni és felszabadítani. Az objektum mezői *PString* típusúak, így a dinamikus sztring-kezeléshez használjuk a *NewStr* és a *DisposeStr* függvényeket.

```
constructor TClient.Init(NewAccount, NewName, NewPhone: String);
begin
  Account := NewStr(NewAccount);
  Name := NewStr(NewName);
  Phone := NewStr(NewPhone);
end;

destructor TClient.Done;
begin
  DisposeStr(Account);
  DisposeStr(Name);
  DisposeStr(Phone);
end;
```

A *TClient.Done* automatikusan meghívásra kerül az egyes ügyfelekhez a kollekció felszabadítása során.

A következő programrészlet bemutatja az ügyfél-kollekció létrehozását:

```
var
  ClientList: PCollection;

begin
  ClientList := New(PCollection, Init(10, 5));

  { A kollekció felépítése }
  with ClientList^ do
  begin
    Insert(New(PClient, Init('91-100', 'Kiss, András',
          '(361) 111-2222')));
    Insert(New(PClient, Init('90-167', 'Tóth, Katalin',
          '(361) 155-1212')));
    Insert(New(PClient, Init('90-177', 'Szemes, Júlia',
          '(361) 187-4321')));
    Insert(New(PClient, Init('90-160', 'Szabó, Zoltán',
          '(361) 139-8913')));
  end;
```

Az első utasítással egy új kollekciót hoztunk létre az ügyfél-adatok számára, amelynél az inicializáló méret 50 ügyfél. Ha a feltöltés során több mint 50 ügyfél kerülne a listára, úgy a kollekció által lefoglalt tárterület megnő. A növelés mértéke 10 ügyfél. A következő utasítások egy-egy ügyfelet helyeznek el a kollekcióban.

9.3.5. Iterációs metódusok (iterátorok)

Gyakran felmerülő probléma a kollekcióban lévő összes elem megjelenítése a képernyőn vagy számítások végzése minden elem felhasználásával. Máskor a kollekció egy tagjára vagyunk kíváncsiak, amely eleget tesz valamilyen feltételnek.

Ezen feladatok ellátására iterációs metódusokat használhatunk: *ForEach, FirstThat, LastThat.* Ezek az iterátorok paraméterként kapják a kollekció feldolgozását meghatározó eljárást vagy a keresési feltételt tartalmazó függvényt.

A *ForEach* metódus paramétere egy olyan eljárás, amely paraméterként a kollekció egy elemét tudja átvenni. A *ForEach* metódus fogja a paraméterként kapott eljárást meghívni és az pedig a pointerével azonosított kollekció-elemet tudja feldolgozni. A *PrintAll* eljárás mutat példát a *ForEach* használatára.

```
procedure TClient.Print;
begin
  Writeln('  ',
    Account^, '':10-Length(Account^),
    Name^, '':20-Length(Name^),
    Phone^, '':16-Length(Phone^));
end;

procedure PrintAll(C: PCollection);

procedure CallPrint(P : PClient); far;
begin
  P^.Print;                    { A nyomtató metódus hívása }
end;

begin { Print }
  Writeln;
  Writeln;
  Writeln('Client list:');
  C^.ForEach(@CallPrint);      { Minden ügyfél kinyomtatása }
end;
```

A *PrintClient* a *PrintAll* belső eljárása, amely megjeleníti a kollekció egy elemét.

Az iterátor metódus által meghívott eljárásnak a következő feltételeknek kell eleget tenni:

- lokálisnak kell lenni-e arra a blokkra nézve, amelyben a meghívása történik,
- nem lehet egy objektum metódusa,
- távoli (**far**) típusú eljárásként kell deklárni. Ezt megtehetjük a **far** direktívával vagy a **$F+** fordítási kapcsolóval.
- az eljárás pointer segítségével tudja átvenni a kollekció egy tagját.

Gyakran előfordul, hogy a kollekció egy tagját kell kikeresnünk, amely eleget tesz valamilyen feltételnek. Ezt a szándékunkat a *FirstThat* vagy a *LastThat* iterátor alkalmazásával valósíthatjuk meg a keresési iránytól függően. A *FirstThat* illetve a *LastThat* egy pointerrel tér vissza, amely a kollekció egy olyan tagjára mutat, ami eleget tesz a keresési feltételnek. Ha nincs ilyen elem, a pointer értéke *NIL*.

A következő példában az ügyfél listában való keresést mutatjuk be. A keresési feltétel a telefonszám egy részlete (pl. a körzetszám).

```
procedure SearchPhone(C: PCollection; PhoneToFind: String);

function PhoneMatch(Client: PClient): Boolean; far;
begin
  PhoneMatch := Pos(PhoneToFind, Client^.Phone^) <> 0;
end;

var
  FoundClient: PClient;

begin { SearchPhone }
  Writeln;
  FoundClient := C^.FirstThat(@PhoneMatch);
  if FoundClient = nil then
    Writeln('No client met the search requirement')
  else
  begin
    Writeln('Found client:');
    FoundClient^.Print;
  end;
end;
```

A példában található egy *PhoneMatch* nevű, távoli *(far)* hívást igénylő *Boolean* típusú függvény. *TRUE* érték visszaadása esetén megtaláltuk az ügyfelet. Ha a lista nem tartalmazott olyan tagot, amely eleget tett volna a keresési kritériumnak a *FirstThat* egy *NIL* pointert adott volna vissza.

Emlékeztető: A *ForEach* a felhasználó által definiált *eljárást* hív, míg a *FirstThat* és a *LastThat* a felhasználó által definiált *Boolean függvényt* hív. Mindegyik esetben a kollekció egy tagját az alprogram részére a pointerével adjuk át. Az alprogramoknak *far* típusúaknak kell lenniük.

9.3.6. Rendezett kollekciók

Néha szükséges, hogy az adataink rendezettek legyenek. A Turbo Vision lehetőséget ad erre egy speciális típusú kollekció - a *TSortedCollection* - felhasználásával.

A *TSortedCollection* a *TCollection* egy leszármazottja, amely automatikusan rendezi és ellenőrzi a tárolásra kerülő objektumokat. Nem engedi meg az azonos tagok tárolását a kollekcióban.

A *TSortedCollection* egy absztrakt típus. A felhasználása során először el kell dönteni a feldolgozandó adatok típusát, majd definiálni kell két metódust. Az egyik metódus a rendezési kulcs előállítását, a másik a kollekció tagjainak az összehasonlítását végzi a kulcsok alapján. Az ügyfél példánál maradva az új ügyfél beszúrását *(Insert)* a kollekcióba illetve az ügyfél törlését *(Delete)* már végrehajthatjuk, mert ezeket a metódusokat a *TClientCollection* örökli a *TCollection*-től.

Amennyiben eldöntöttük, hogy melyik mező alapján legyen rendezett a kollekció, át kell definiálnunk a *KeyOf* és a *Compare* metódusokat.

```
  PClientCollection = ^TClientCollection;
  TClientCollection = object(TSortedCollection)
    function KeyOf(Item: Pointer): Pointer; virtual;
    function Compare(Key1, Key2: Pointer): Integer; virtual;
  end;


{ TClientCollection }
function TClientCollection.KeyOf(Item: Pointer): Pointer;
begin
  KeyOf := PClient(Item)^.Account;
end;

function TClientCollection.Compare(Key1, Key2: Pointer): Integer;
begin
  if PString(Key1)^ = PString(Key2)^ then
    Compare := 0
  else if PString(Key1)^ < PString(Key2)^ then
    Compare := -1
  else
    Compare := 1;
end;
```

A *KeyOf* adja vissza a rendezéshez szükséges kulcsot, ebben az esetben az ügyfél nevét. A *Compare* végzi az elemek összehasonlítását a kulcsok alapján. Ha a két kulcs megegyezik nullát, ha Key1 nagyobb, mint Key2 1, különben -1 a visszaadott érték. A fenti példa névsor szerinti rendezettséget tudja biztosítani. Mivel a *KeyOf* típussal nem rendelkező pointert ad át a *Compare* függvénynek, szükséges, hogy az átadott pointeren egy típuskonverziót hajtsunk végre.

9.3.7. Sztring-kollekció

A programokban szükség lehet rendezett sztringekre. Ezt a célt szolgálja *TStringCollection* típus. A *TStringCollection*-ban lévő elemek nem objektumok, hanem pointerek, amelyek Turbo Pascal sztringekre mutatnak. Az azonos sztringek tárolása kiküszöbölhető, ha a sztring-kollekció utódja a *TSortedCollection* típusnak.

A sztring-kollekció létrehozásához deklarálni kell egy pointert, amely a sztring-kollekcióra mutat. Helyet kell foglalnunk a memóriában a kollekció részére egy inicializáló méret megadásával.

```
var
  WordList: PCollection;
  WordRead: String;
  ...
begin
  WordList := New(PStringCollection, Init(10, 5));
  ...
```

Ha 10-nél több sztring kerülne a kollekcióba, a helyfoglalás inkrementálása 5 újabb sztring elhelyezését teszi lehetővé. A következő példában egy text file-ból olvasunk fel sztringeket és helyezzük el azokat a kollekcióban.

```
uses Objects, Memory;

procedure Print(C: PCollection);

procedure PrintWord(P : PString); far;
begin
  Writeln(P^);
end;

begin
  Writeln;Writeln;
  C^.ForEach(@PrintWord);
end;

procedure Abort(Msg: String);
begin
  Writeln;Writeln(Msg);
  Writeln('Program aborting');
  Halt(1);
end;

function GetWord(var F : Text) : String;
var
  S : String;C : Char;
begin
  S := '';C := #0;
  while not Eof(F) and not (UpCase(C) in ['A'..'Z']) do
    Read(F, C);
  if Eof(F) and (UpCase(C) in ['A'..'Z']) then S := C
  else
    while (UpCase(C) in ['A'..'Z']) and not Eof(F) do
    begin
      S := S + C;
      Read(F, C);
    end;
  GetWord := S;
end;
```

```
var
  WordList: PCollection;
  WordFile: Text;
  WordFileName: string[80];
  WordRead: String;

begin
  WordList := New(PStringCollection, Init(10, 5));
  if LowMemory then Abort('Out of memory');

  { Open file of words }
  if ParamCount = 1 then WordFileName := ParamStr(1) else Halt(1);
  Assign(WordFile, WordFileName);
  {$I-}
  Reset(WordFile);
  {$I+}
  if IOResult <> 0 then
    Abort('Cannot find file "' + WordFileName + '"');

  repeat
    WordRead := GetWord(WordFile);
    if WordRead <> '' then
      WordList^.Insert(NewStr(WordRead));
    if LowMemory then Abort('Out of memory');
  until WordRead = '';
  Close(WordFile);

  Print(WordList);

  Dispose(WordList, Done);
end.
```

Megjegyzés: A *NewStr* függvény készít egy másolatot a *WordRead* változóban lévő sztringről és a címét elhelyezi a kollekcióban. A teljes kollekció felszabadítása a *Dispose* eljárással történik, megadva *Done* metódust. A *ForEach* metódus segítségével itt is áttekinthetjük a teljes kollekciót. A *Print* eljáráson belül több, a kollekciót feldolgozó eljárás is elhelyezhető (adatok megjelenítése, módosítása, stb.), és mindegyiknek *far* típusúnak kell lenni.

Rendezett sztring-kollekciónál használható egy *Search* (keresési) metódus, ami visszaadja a kollekció tagjának az indexét. Ha a készlet nem rendezett, használható a *FirstThat* vagy a *LastThat* metódus definiálva hozzá a megfelelő keresési kritériumot tartalmazó *Boolean* függvényt.

9.3.8. A kollekció sokrétűsége

Láthattuk, hogy a kollekcióban az adatok tárolása dinamikusan történik, de az eddigiekben a kollekció tagjai azonos típusúak voltak. Az egyik példában ügyfeleket, a másikban sztringeket tároltunk. Lehetőségünk van különböző típusú objektumok kollekcióba történő elhelyezésére is. Természetesen ezek mindegyikének közös őse a *TObject*. Az alábbi példában 3 különböző grafikus objektumot teszünk egy kollekcióba, majd a *ForEach* iterátor felhasználásával megjelenítjük ezeket a képernyőn. A példa használja a Graph.TPU unit-ot és a megfelelő BGI drivert. A Graph unit a fordítás során, a BGI állománynak a futás során elérhetőnek kell lenni. (Lásd az Options|Directories|Unit menüt és az *Initgraph* használatát.)

Az első lépés az absztrakt ős definiálása:

```
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
    X,Y: Integer;
    constructor Init;
    procedure Draw; virtual;
  end;
```

A példából látható, hogy a grafikus objektumot az *Init* metódus inicalizálizálja és a *Draw* metódus jeleníti meg.

```
  PGraphPoint = ^TGraphPoint;
  TGraphPoint = object(TGraphObject)
    procedure Draw; virtual;
  end;


  PGraphCircle = ^TGraphCircle;
  TGraphCircle = object(TGraphObject)
    Radius: Integer;
    constructor Init;
    procedure Draw; virtual;
  end;

  PGraphRect = ^TGraphRect;
  TGraphRect = object(TGraphObject)
    Width, Height: Integer;
    constructor Init;
    procedure Draw; virtual;
  end;
```

A kollekció létrehozása:

```
procedure MakeCollection(var List: PCollection);
var
  I: Integer;
  P: PGraphObject;
begin
  List := New(PCollection, Init(10, 5));
  for I := 1 to 20 do
  begin
    case I mod 3 of
      0: P := New(PGraphPoint, Init);
      1: P := New(PGraphCircle, Init);
      2: P := New(PGraphRect, Init);
    end;
    List^.Insert(P);
  end;
end;
```

Az iterátor metódus felhasználásával a kollekcióban lévő objektumok megjelenítése:

```
procedure DrawAll(C: PCollection);

procedure CallDraw(P : PGraphObject); far;
begin
  P^.Draw;
end;

begin { DrawAll }
C^.ForEach(@CallDraw);
end;

var GraphicsList: PCollection;
begin
  ...
DrawAll(GraphicsList);
  ...
end.
```

9.3.9. Kapcsolat a memóriakezelővel

A kollekcióban tárolt elemek maximális számát a *MaxCollectionSize* konstans határozza meg. Ennek alapértéke: 16380. Egy elem mérete 4 byte, mert az objektum pointere kerül tárolásra.

Az új elemnek a kollekcióhoz csatolása során, ha nem áll elegendő memória a rendelkezésre a *TCollection.Error* meghívásra kerül. Ez a metódus lekezeli a futási időben előforduló memóriával kapcsolatos hibákat. A *TCollection.Error* felülírható, vagyis saját hibakezelést építhetünk be a programba.

10. TURBO VISION KIEGÉSZÍTÉSEK

Az objektum-orientált és az eseményvezérelt programozás még egészen új fogalmak a tapasztalt Pascal programozók körében is. Ez a fejezet néhány tanácsot tartalmaz a Turbo Vision eredményesebb használatához.

10.1. Megbízható programok írása

A hibakezelés egy interaktív programnál sokkal bonyolultabb, mint a csak parancssorral rendelkező alkalmazások esetében. Ki kell küszöbölni egy figyelmetlenül lenyomott gomb hatását, nem szabad megengedni a hamis információk feldolgozását, stb. A nem interaktív programoknál hiba esetén egy üzenetet kapunk a képernyőn és befejeződik a program futása.

A Turbo Vision elősegíti a biztonságos programok írását. Támogat egy olyan programozási stílust, amely egyszerűbbé teszi a hibák kezelését, főleg a ravasz és nehezen megfogható memória túlcsordulási hibákat. A programozási mód kialakításához meg kell ismernünk az elemi művelet fogalmát.

Az elemi művelet már további műveletekre nem bontható és végrehajtása vagy teljesen sikeres, vagy teljesen sikertelen. Egy elemi művelet végrehajtása főleg akkor hasznos, amikor memória kérelemmel lépünk fel. Például egy dialógus doboz létrehozásánál memóriát kell allokálnunk a dialógus dobozhoz és a dialógus vezérléséhez. Ilyenkor szükséges, hogy teszteljük a memória foglalás sikerességét és el kell döntenünk, hogy a tárkérelmek folytathatóak-e, vagy sem. Ideális lenne, ha először minden tárkérelmet végrehajthatnánk és csak utána ellenőriznénk ezek sikerességét.

10.1.1. Biztonsági terület

A Turbo Vision lefoglal egy 4K méretű memóriaterületet a heap tetején, amit biztonsági területnek hívunk. Ha a tárkérelmünk során a lefoglalt terület belenyúlik a biztonsági területbe a *LowMemory* függvény *TRUE* értéket ad vissza. Ez azt jelzi, hogy a további kérelmek már nem teljesíthetők biztonságosan. A biztonsági területnek nagyobbnak kell lennie, mint a legnagyobb elemi memóriafoglalásnak. Más szóval: akkora terület szükséges, hogy két memória allokáció között ellenőrizhető legyen a LowMemory állapota. Legtöbb esetben 4K elegendő erre.

Hagyományos programozási stílusnál egy dialógus doboz létrehozása valahogy így néz ki:

```
        OK := True;
        R.Assign(20,3,60,10);     { A dialógus doboz mérete.}
        D := New(Dialog, Init(R, 'My dialog'));  {A dialógus doboz
                                                   létrehozása. }
        if D <> nil then
       { A dialógus doboz elemeinek beépítése lépésenkénti }
       { ellenőrzéssel.                                    }
        begin
          with D^ do
          begin
            R.Assign(2,2,32,3);
            Control := New(PStaticText,          { statikus szöveg }
                           Init(R, 'Do you really wish to do this?'));
            if Control <> nil then Insert(Control)
            else OK := False;
            R.Assign(5,5,14,7);                  { vezérlő gomb    }
            Control := New(PButton, Init(R, '~Y~es', cmYes));
            if Control <> nil then Insert(Control)
            else OK := False;
            R.Assign(16,6,25,7);                 { vezérlő gomb    }
            Control := New(PButton, Init(R, '~N~o', cmNo));
            if Control <> nil then Insert(Control)
            else OK := False;
            R.Assign(27,5,36,7);                 { vezérlő gomb    }
            Control := New(PButton, Init(R, '~C~ancel', cmCancel));
            if Control <> nil then Insert(Control)
            else OK := False;
          end;
          if not OK then Dispose(D, Done);
        end;
```

Az allokációk sikerességéről az OK változó tájékoztat. Hiba esetén az egész dialógus dobozt fel kell szabadítanunk. Ennek a problémának a lekezelése kellemesebbé tehető a következő formában:

```
        R.Assign(20,3,60,10);
        D := New(Dialog, Init(R, 'My dialog'));
        with D^ do
        { a fenti dialógus doboz felépítése a lépésenkénti }
        { ellenőrzések nélkül.                             }
        begin
          R.Assign(2,2,32,3);
          Insert(New(PStaticText,
                 Init(R, 'Do you really wish to do this?')));
          R.Assign(5,5,14,7);
          Insert(New(PButton, Init(R, '~Y~es', cmYes)));
          R.Assign(16,6,25,7);
          Insert(New(PButton, Init(R, '~N~o', cmNo)));
          R.Assign(27,5,36,7);
```

```
    Insert(New(PButton, Init(R, '~C~ancel', cmCancel)));
  end;
  if LowMemory then
  { a LowMemory függvény segítségével ellenőrízhető, hogy }
  { van-e még elegendő memória.                           }
  begin
    Dispose(D, Done);
    OutOfMemory;
    DoIt := False;
  end
  else
        { a dialógus doboz megjelenítése }
    DoIt := DeskTop^.ExecView(D)=cmYes;
```

Mivel a biztonsági terület mérete elég nagy ahhoz, hogy a dialógus doboz létrehozása során felmerülő allokációs problémák tesztelhetők vele, csak a *LowMemory* állapotát kell megvizsgálni. Ha ennek értéke *FALSE*, a dialógus doboz használható, különben fel kell szabadítanunk. Mivel elég gyakran kellene erre a függvényre figyelnünk a *TApplication* tartalmazhat egy *ValidView* metódus hívást, amivel elvégezhetjük a szükséges ellenőrzéseket. A *ValidView* felhasználásával az előző példa feltételes utasítása a következőképp egyszerűsödik:

```
  ...
  DoIt := (ValidView(D) <> nil) and
    (DeskTop^ExecView(D) = cmYes);
```

A *ValidView* egy pointert ad vissza. Hiba esetén értéke **nil**. A *LowMemory True* értékénél a *ValidView* felszabadítja a *view*-t és meghívja az *OutOfMemory* eljárást.

10.1.2. Nem memóriakezelésből származó hibák

Természetesen nem mindegyik hiba függ össze a memóriakezeléssel. Például adatokat akarunk beolvasni a lemezről és a file név nem érvényes vagy nincs ilyen file. Az ilyen típusú hibákról tájékoztatni kell a felhasználót. A nem memória kezelésből származó hibák figyelését a *Valid* metódusok alkalmazásával tehetjük meg. Szerencsére az ilyen hibák felismerésére *ValidView*-n keresztül is meghívásra kerülhet a *Valid* függvény.

A *TView.Valid True* értékkel tér vissza alapértelmezésben.

A *TGroup Valid* csak akkor tér vissza *TRUE*-val, ha az általa kezelt mindegyik *view* érvényes.

A nem memória hibák teszteléséhez a *Valid* metódust felül kell írnunk:

```
function TMyView.Valid(Command: Word): Boolean;
begin
  Valid := True;
  if Command = cmValid then
  begin
    if ErrorEncountered then
    begin
      ReportError;
      Valid := False;
    end;
  end;
end;
```

Egy *view* bevezetésekor a *Valid* metódust meg kell hívnunk a *cmValid* paraméterrel a nem memória hibák felderítéséhez. Ezt megteszi a *ValidView* automatikusan: *ValidView(X)* meghívja az *X.Valid(cmValid)*-ot.

10.2. A Turbo Vision alkalmazói programok nyomkövetése

Ha megpróbálunk lépésenként végrehajtani egy programot, nem tapasztalunk teljes sikert, mert a Turbo Vision programok eseményvezéreltek. A megoldás a Turbo Vision alkalmazások nyomkövetéséhez a töréspontok elhelyezése a programban.

A nyomkövetés alkalmával felmerülő probléma, hogy a program bizonyos részei a lépésenkénti végrehajtás során nem figyelhetők meg. Például egy menüopció kiválasztása, amelynek feladata egy ablak megjelenítése.

A legjobb megközelítése az ilyen helyzeteknek a töréspontok elhelyezése a *HandleEvent* metódusokban. A *breakpoint*-ot beállítva a *HandleEvent* metódus elején, elkezdődhet a program lépésenkénti végrehajtása, megvizsgálhatóvá válik az esemény-rekord és megbizonyosodhatunk a várt eseményről.

Előfordulhat, hogy egy másik objektum hamarabb elkezdi az esemény feldolgozását és az általa törlődik, mielőtt az a megfelelő helyre kerülne. Ezt előidézheti:

- a dupla parancs-deklaráció,

- egy elmaradt *cmJump* törlése.

Gyakran előforduló és a nyomkövetést megnehezítő hiba egy pointer több alkalommal történő felszabadítása.

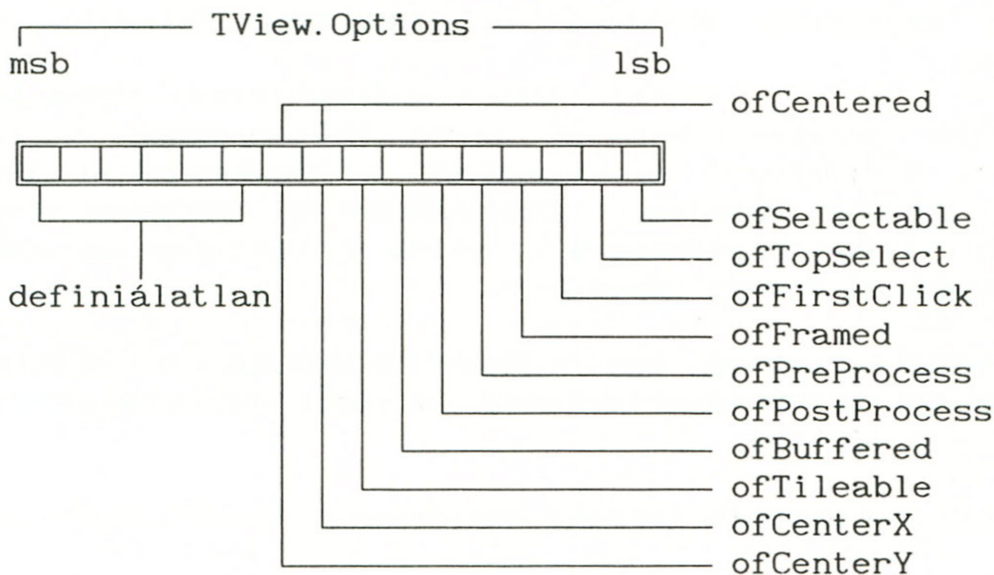
A példát nem ajánljuk lefuttatni, csak szemléltető jellegű:

```
...
var
  Bruce, Pizza: PGroup;
  R: TRect;
begin
  R.Assign(5,5,10,10);
  Pizza := New(PGroup, Init(R));
  R.Assign(10,10,20,20);
  Bruce := New(PGroup, Init(R));
  Bruce^.Insert(Pizza);
  Dispose(Bruce, Done);
  Dispose(Pizza, Done);
end;
...
```

A *view*-t felszabadítja a tulajdonosa, majd közvetlenül is megpróbálja felszabadítani.

10.3. Bittérképek felhasználása

A Turbo Vision *view*-k több esetben is használnak bittérképszerűen mezőket. A *byte* vagy a *word* típusú mező egyes bitjeit jelzőként (flag) alkalmazza a Turbo Vision vagy a programozó. Például minden *view*-nak van egy word típusú bittérképszerűen használt *Options* nevű mezője. A mező minden egyes bitje különböző jelentéssel bír, ezt általában egy egyszerű ábrával szemléltetjük.



10.1. ábra
View bittérképe

A 10.1. ábrán *msb*-vel (*most significant bit*) jelölik a legmagasabb helyiértékű bitet és *lsb*-vel (*least significant bit*) a legalacsonyabb helyiértékűt. Például a 4. bitet *ofFramed*-nek nevezik és értékétől függően a *view* kerete látható (1) vagy nem látható (0).

A *flag*-ek kezelését a maszkok használata egyszerűbbé teszi. Például a Turbo Vision-ben különböző események teszteléséhez előre definiált maszkok vannak . Az *evMouse* maszk felhasználásával vizsgálhatjuk, hogy valamilyen esemény bekövetkezését az egér okozta-e.

10.3.1. A bitműveletek öszefoglalása:

Bit beállítása az M változóban:

M := M **or** flag;

Bit törlése:

M := M **and** not flag;

Bit tesztelése:

if M **and** flag = flag **then** ...

Flag tesztelése egy maszk által:

if flag **and** maszk <> 0 **then** ...

10.4. Turbo Vision alkalmazói programok *overlay* szervezése

Minden Turbo Vision unit *overlay*-be tehető, kivéve a *Drivers* unit-ot, amelyik az interrupt-kezelést és más alacsony szintű interface-t tartalmaz.

A Turbo Vision alkalmazások *overlay* szervezésénél figyelembe kell venni, hogy mely objektumok állnak egymással kapcsolatban, mert ezen objektumok kódjainak egyidejűleg kell elférnie az *overlay* területen. Minthogy a Turbo Pascal *overlay* kezelője az egész unit-ot kicseréli, az egymással kapcsolatban nem álló objektumokat ne tegyük ugyanabba az *overlay* unit-ba. A következő példában található egy dialógus doboz, ami tartalmaz néhány vezérlést is. A dialógus a *TDialog*-ból származik és a vezérlések a *TListViewer* és a *TInputLine* objektumokból. Célszerű behelyezni mindhárom objektumot ugyanabba a unit-ba, mert ezek egymással kapcsolatban állnak a dialógus doboz működése során. Viszont más objektumot nem célszerű betenni ebbe unit-ba, mert a szükséges *overlay* terület mérete megnő és a lemezműveleteknek sem válik előnyére.

Egy Turbo Vision alkalmazásban az *App, Memory, Menus, Objects* és *Views* unit-ok 50 Kbyte kóddal majdnem mindig részei az aktuális műveletnek. Ezenkívül még a programozó által létrehozott leszármaztatott objektumok, ablakok és dialógusok is megjelennek. Ez azt okozza, hogy az *overlay* terület minimális mérete kb. 64 Kbyte.

Kísérletezéssel meghatározható az *overlay* terület ideális mérete. Általánosságban az EMS megléte nyújt jobb lehetőségeket az *overlay* kezelésében. Gyorsabbá teszi a kód cseréjét, és lehetőséget ad az *overlay* terület 25-35%-os csökkentésére is.

A következő programrészlet egy tipikus ovarlay-szevezést mutat be a Turbo Vision alkalmazásokhoz:

```
    program MyProg;

    {$F+,O+,S-}
    {$M 8192,65536,655360}

    uses Overlay, Drivers, Memory, Objects, Views, Menus, Dialogs,
      HistList, StdDlg, App;
{ A szabványos Turbo Vision unit-ok overlay-ba helyzésének előírása }
    {$O App      }
    {$O Dialogs  }
    {$O HistList }
    {$O Memory   }
    {$O Menus    }
    {$O Objects  }
    {$O StdDlg   }
    {$O Views    }

    const
      ExeFileName = 'MYPROG.EXE';  { Az overlay-t tartalmazó file }
      OvrBufDisk  = 96 * 1024;     { puffer méretek }
      OvrBufEMS   = 72 * 1024;

    type
      TMyApp = object(TApplication)  { Az alkalmazói objektum }
        constructor Init;
        destructor Done; virtual;
        ...
      end;

    procedure InitOverlays;      { Az overlay rendszer inicializálása }
    var
      FileName: string[79];
    begin
      FileName := ParamStr(0);
      if FileName = '' then FileName := ExeFileName;
      OvrInit(FileName);
      if OvrResult <> 0 then
      begin
        PrintStr('Fatal error: Cannot open overlay file.');
```

```
      Halt(1);
    end;
    OvrInitEMS;  { Ha lehetséges az overlay puffer az EMS-be kerül}
    if OvrResult = 0 then OvrSetBuf(OvrBufEMS) else
    begin
      OvrSetBuf(OvrBufDisk);  { ha nem az lenne }
      OvrSetRetry(OvrBufDisk div 2);
    end;
  end;
 { Az alkalmazói objektum metódusa. }
  constructor TMyApp.Init;
  begin
    InitOverlays;
    TApplication.Init;
    .
    .
  end;

  destructor TMyApp.Done;
  begin
    .
    .
    TApplication.Done;
  end;

  var
    MyApp: TMyApp;

  begin            { A tipikus Turbo Vision főprogram. }
    MyApp.Init;
    MyApp.Run;
    MyApp.Done;
  end.
```

Az *overlay* kezelőt inicializálni kell, mielőtt meghívásra kerül a *TApplication.Init*.

Az *OvrSetRetry*-t nem hívjuk meg, ha installálva van az EMS, mert teljesítmény növelő hatása csak a lemezes file-ok esetén van. Az *overlay* állomány hozzácsatolható az EXE file-hoz. Ezt a DOS COPY parancsával könnyen elvégezhetjük:

```
REN MYPROG.EXE TEMP.EXE
COPY /B TEMP.EXE+MYPROG.OVR MYPROG.EXE
```

Ne feledkezzünk meg a **{$F+, $O+}** kapcsolók elhelyezéséről az *overlay* unit-ok elején!

F1. Függelék Turbo Vision unit-ok rövid referenciái

Ez a fejezet egy rövid áttekintést ad a Turbo Vision egyes moduljainak a felépítéséről.

A Turbo Vision kilenc unit-ot tartalmaz:

Unit	Tartalom
App	Objektum-definíciók esemény-vezérelt alkalmazói programok készítéséhez
Dialogs	Eszközök a dialógus-dobozok használatához
Drivers	Egér-kezelés, billentyűzet-kezelés, ...
Histlist	Az input-lista tárolása
Memory	Memóriakezelő rendszer
Menus	Objektumok a menükhöz, illetve a státuszsor használatához
Objects	Alapvető definíciók a néma objektumokhoz
Textview	Szövegek megjelenítése
Views	Alapvető objektumok az ablakkezeléshez (*view*-k, keretek, ablakok, képernyőgörgetés)

A Turbo Vision globális elemei A-tól C-ig:

Elem	Típus	Unit
Abstract	eljárás	Objects
Application	változó	App
AppPalette	változó	App
apXXXX	konstans	App
AssignDevice	eljárás	TextView
bfXXXX	konstans	Dialogs
ButtonCount	változó	Drivers
CheckSnow	változó	Drivers
ClearHistory	eljárás	HistList
ClearScreen	eljárás	Drivers
cmXXXX	konstans	Dialogs
cmXXXX	konstans	Drivers
cmXXXX	konstans	Views
coXXXX	konstans	Objects
CStrLen	függvény	Drivers
CtrlBreakHit	változó	Drivers
CtrlToArrow	függvény	Drivers
CursorLines	változó	Drivers

A Turbo Vision globális elemei D-től L-ig:

Elem	Típus	Unit
DeskTop	változó	App
DisposeMenu	eljárás	Menus
DisposeStr	eljárás	Objects
dmXXXX	konstans	Views
DoneEvents	eljárás	Drivers
DoneHistory	eljárás	HistList
DoneMemory	eljárás	Memory
DoneSysError	eljárás	Drivers
DoneVideo	eljárás	Drivers
DoubleDelay	változó	Drivers
EmsCurHandle	változó	Objects
EmsCurPage	változó	Objects
evXXXX	konstans	Drivers
FNameStr	típus	Objects
FocusedEvents	változó	Views
FormatStr	eljárás	Drivers
FreeBufMem	eljárás	Memory
GetAltChar	függvény	Drivers
GetAltCode	függvény	Drivers
GetBufMem	eljárás	Memory
GetKeyEvent	eljárás	Drivers
GetMouseEvent	eljárás	Drivers
gfXXXX	konstans	Views
hcXXXX	konstans	Views
HideMouse	eljárás	Drivers
HiResScreen	változó	Drivers
HistoryAdd	eljárás	HistList
HistoryBlock	változó	HistList
HistoryCount	függvény	HistList
HistorySize	változó	HistList
HistoryStr	függvény	HistList
HistoryUsed	változó	HistList
InitEvents	eljárás	Drivers
InitHistory	eljárás	HistList
InitMemory	eljárás	Memory
InitSysError	eljárás	Drivers
InitVideo	eljárás	Drivers
kbXXXX	konstans	Drivers
LongDiv	függvény	Objects
LongMul	függvény	Objects
LongRec	típus	Objects
LowMemory	függvény	Memory
LowMemSize	változó	Memory

A Turbo Vision globális elemei M-től sf-ig:

Elem	Típus	Unit
MaxBufMem	változó	Memory
MaxCollectionSize	változó	Objects
MaxViewWidth	constant	Views
mbXXXX	konstans	Drivers
MemAlloc	függvény	Memory
MemAllocSeg	függvény	Memory
MenuBar	változó	App
Message	függvény	Views
MinWinSize	változó	Views
MouseButtons	változó	Drivers
MouseEvents	változó	Drivers
MouseIntFlag	változó	Drivers
MouseWhere	változó	Drivers
MoveBuf	eljárás	Objects
MoveChar	eljárás	Objects
MoveCStr	eljárás	Objects
MoveStr	eljárás	Objects
NewItem	függvény	Menus
NewLine	függvény	Menus
NewMenu	függvény	Menus
NewSItem	függvény	Dialogs
NewStatusDef	függvény	Menus
NewStatusKey	függvény	Menus
NewStr	függvény	Objects
NewSubMenu	függvény	Menus
ofXXXX	konstans	Views
PChar	típus	Objects
PositionalEvents	változó	Views
PrintStr	eljárás	Drivers
PString	típus	Objects
PtrRec	típus	Objects
RegisterDialogs	eljárás	Dialogs
RegisterType	eljárás	Objects
RepeatDelay	változó	Drivers
SaveCtrlBreak	változó	Drivers
sbXXXX	konstans	Views
ScreenBuffer	változó	Drivers
ScreenHeight	változó	Drivers
ScreenMode	változó	Drivers
ScreenWidth	változó	Drivers
SelectMode	típus	Views
SetVideoMode	eljárás	Drivers
sfXXXX	konstans	Views

A Turbo Vision globális elemei Sh-tól W-ig:

Elem	Típus	Unit
ShadowAttr	változó	Views
ShadowSize	változó	Views
ShowMarkers	változó	Drivers
ShowMouse	eljárás	Drivers
smXXXX	konstans	Drivers
SpecialChars	változó	Views
stXXXX	konstans	Objects
StartupMode	változó	Drivers
StatusLine	változó	App
StreamError	változó	Objects
SysColorAttr	változó	Drivers
SysErrActive	változó	Drivers
SysErrorFunc	változó	Drivers
SysMonoAttr	változó	Drivers
SystemError	függvény	Drivers
TByteArray	típus	Objects
TCommandSet	típus	Views
TDrawBuffer	típus	Views
TEvent	típus	Drivers
TItemList	típus	Objects
TMenu	típus	Menus
TMenuItem	típus	Menus
TMenuStr	típus	Menus
TPalette	típus	Views
TScrollChars	típus	Views
TSItem	típus	Dialogs
TStatusDef	típus	Menus
TStatusItem	típus	Menus
TStreamRec	típus	Objects
TStrIndex	típus	Objects
TStrIndexRec	típus	Objects
TSysErrorFunc	típus	Drivers
TTerminalBuffer	típus	TextView
TTitleStr	típus	Views
TVideoBuf	típus	Views
TWordArray	típus	Objects
wfXXXX	konstans	Views
wnNoNumber	konstans	Views
WordRec	típus	Objects
wpXXXX	konstans	Views

Az OBJECTS unit tartalmazza a Turbo Vision alapvető objektumainak definícióit, a *TObject*-et (amely Turbo Vision hierarchia alapobjektuma), a stream-eket, kollekciókat, erőforrásokat (amelyek Turbo Vision nem látható elemei).

TÍPUSOK

Típus átalakító rekordok

Típus	Felhasználás
FNameStr	Sztring egy DOS filenév tárolásához
LongRec	Egy longint alacsonyabb és magasabb helyiértékű szavának kiolvasása
PChar	Pointer egy karakterhez
PString	Pointer egy sztring-hez
PtrRec	Pointer konvertálása szegmens és offszet részre
TByteArray	Byte típusú elemeket tartalmazó tömbtípus
TWordArray	Word típusú elemeket tartalmazó tömbtípus
WordRec	Egy word alacsonyabb és magasabb helyiértékű byte-jának kiolvasása

Objektumtípusok

Típus	Felhasználás
TBufStream	Pufferelt stream-kezelés
TCollection	Absztrakt típus elemkészlet kialakításához
TDosStream	DOS stream-kezelés
TEmsStream	EMS stream-kezelés
TItemList	*TCollection* objektum által használt pointertömb
TObject	Absztrakt alaptípus, az összes TV objektum őse
TPoint	Pontot reprezentál
TRect	Téglalapot reprezentál
TResourceCollection	Erőforrás-készlet
TResourceFile	Erőforrás-file
TStream	Alapvető objektum a TV stream-kezeléséhez
TStreamRec	Rekordtípus az objektumok regisztrálásához
TStrIndex	*TStrIndexRec* típusú elemeket tartalmazó tömb
TStrIndexRec	A *TStringList*, *TStrListMaker* belső használatára
TStringCollection	Típus az ASCII sztringek rendezett kezeléséhez
TStringList	Sztringek tárolása a stream-en
TStrListMaker	Sztring lista létrehozása

KONSTANSOK

Stream megnyitási módok (jelölése: stXXXX)

Mód	Érték	Jelentés
stCreate	$3C00	Új file létrehozása
stOpenRead	$3D00	File megnyitása olvasásra
stOpenWrite	$3D01	File megnyitása írásra
stOpen	$3D02	File megnyitása írásra-olvasásra

Stream hibakódok (jelölése: stXXXX)

Hibakód	Érték	Jelentés
stOk	0	Nincs hiba
stError	-1	Megnyitási hiba
stInitError	-2	A stream nem inicializálható
stReadError	-3	Olvasási hiba
stWriteError	-4	Írási hiba
stGetError	-5	Nem regisztrált objektumtípus olvasása
stPutError	-6	Nem regisztrált objektumtípus írása

Az elemkészlet maximális mérete

Hibakód	Érték	Jelentés
MaxCollectionSize	16380	A *TCollection* maximális mérete

Elemkészlet hibakódok (jelölése: coXXXX)

Hibakód	Érték	Jelentés
coIndexError	-1	Index a határon kívül
coOverflow	-2	Túlcsordulás

VÁLTOZÓK

Változó	Típus	Kezdőérték	Jelentés
EmsCurHandle	Word	$FFFF	Aktuális EMS leíró
EmsCurPage	Word	$FFFF	Aktuális EMS lap

ELJÁRÁSOK ÉS FÜGGVÉNYEK

Eljárás	Művelet
Abstract	Alapmetódus, amely használat esetén átdefiniálandó
DisposeStr	A *NewStr* függvénnyel létrehozott sztring felszabadítása
RegisterType	Objektumok regisztrálása

Függvény	Művelet
LongDiv	Longint típusú érték osztása integer típusúval
LongMul	Integer típusú értékek szorzása
NewStr	Helyfoglalás sztringnek

A VIEWS UNIT

A VIEWS unit tartalmazza a *view*-k alapvető elemeit. (A Turbo Vision látható része.) Absztrakt típusokat (*TView*, *TGroup*) és több összetett csoportot (például: ablakkeretek, görgető sorok). A látható elemek egy másik része a DIALOGS és a TEXTVIEW unit-okban találhatók.

TÍPUSOK

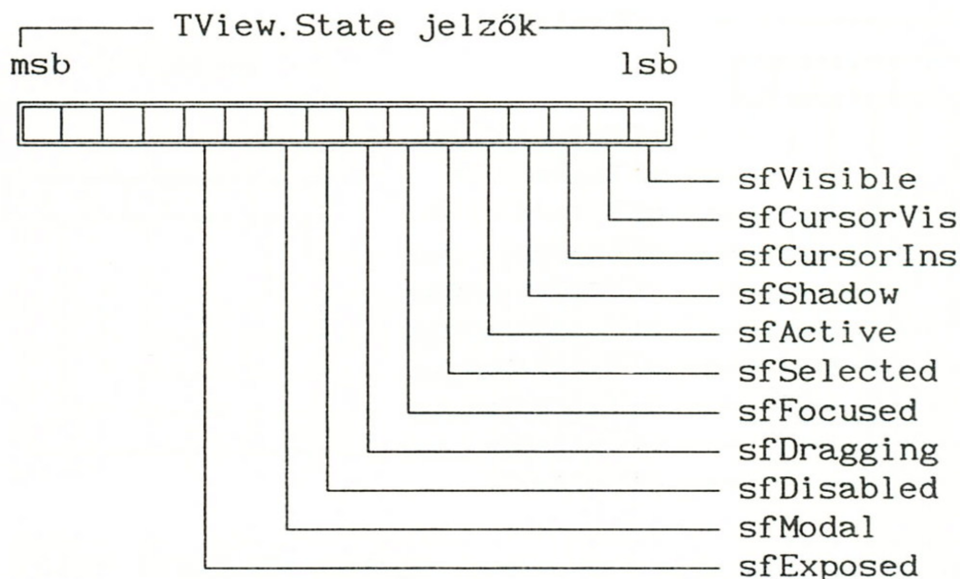
Objektum típus	Felhasználás
TFrame	Keret objektum az ablakok alkalmazásához
TGroup	Absztrakt objektum az összetett *view*-khoz
TListViewer	Alaptípus a listázó doboz számára
TScrollBar	A görgető sor objektum definíciója
TScroller	Alapobjektum a szöveges ablakok görgetéséhez (scroll)
TView	Absztrakt objektum; a látható objektumok őse
TWindow	Alapobjektum az ablakokhoz

Egyéb típusok	Felhasználás
TCommandSet	A parancsok engedélyezése vagy letiltása
TDrawBuffer	Puffer a Draw metódushoz
TPalette	Paletta típus
TScrollChars	A *TScrollBar* által használt karakterek
TTitleStr	A *TFrame* által használt cím sztring
TVideoBuf	A képernyő-kezelő által használt puffer

A státusz jelző konstansok (jelölése: sfXXXX)

TView.State mezőt sohase változtassuk direkt módon, hanem a *TView.SetState* metódussal.

Konstans	Érték	Jelentés
sfVisible	$0001	A *view* látható
sfCursorVis	$0002	A kurzor látható
sfCursorIns	$0004	A kurzor a beszúrás üzemmódot jelzi
sfShadow	$0008	A *view*-nak árnyéka van
sfActive	$0010	A *view* az aktív ablak
sfSelected	$0020	A *view* a tulajdonosa az aktiv *view*-nak
sfFocused	$0040	A *view* fókuszált *view*
sfDragging	$0080	A *view* mérete megváltozhat
sfDisabled	$0100	A *view* le van tiltva
sfModal	$0200	A *view* egy *modal view*
sfExposed	$0800	A *view* tulajdonosa az *Application* objektum



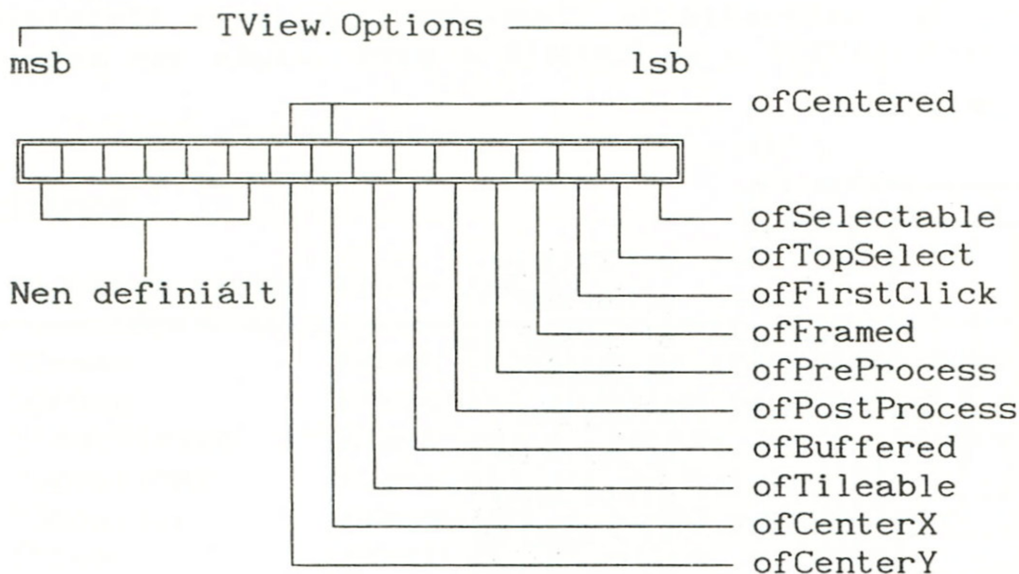
Speciális Views unit konstansok

Konstans	Érték	Jelentés
hcNoContext	0	Jelzi hogy nem tartozik help a *view*-hoz
hcDragging	1	A *Help context* amikor a *view* mozog
MaxViewWidth	132	*View*-k maximális mérete karakterben kifejezve
wnNoNumber	0	*TWindow* konstans nem számozott ablak jelölésére

Az ofXXXX konstansok

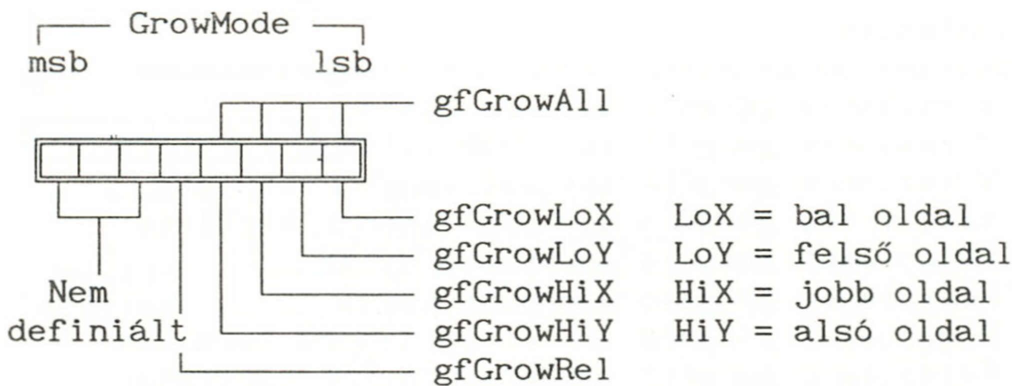
Ezeknek a konstansoknak a felhasználásával állíthatjuk be az *Options* mező bitjeit.

Konstans	Érték	Jelentés
ofSelectable	$0001	A *view* kiválasztható
ofTopSelect	$0002	A kiválasztott *view* legfelülre kerül
ofFirstClick	$0004	Az egér klikentés átadódik a *view*-nak
ofFramed	$0008	A *view*-nak látható a kerete
ofPreProcess	$0010	A fókuszált esemény feldolgozása mielőtt a fókuszált *view* érzékelné
ofPostProcess	$0020	A fókuszált esemény feldolgozása a fókuszált *view*-ban
ofBuffered	$0040	Gyorsító puffer
ofTileable	$0080	A *view* lefedhető
ofCenterX	$0100	A *view* vízszintes irányú középre igazítása
ofCenterY	$0200	A *view* függőleges irányú középre igazítása
ofCentered	$0300	A *view* mindkét irányú középre igazítása



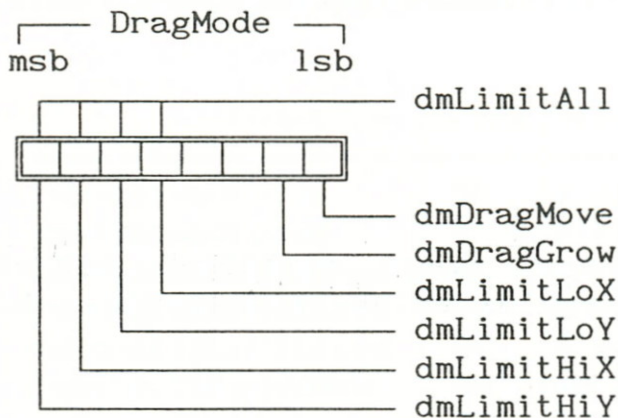
A view-k elhelyezkedésének meghatározása (jelölése: gfXXXX)

Konstans	Érték	Jelentés
gfGrowLoX	$01	Illesztés a tulajdonos bal oldalához
gfGrowLoY	$02	Illesztés a tulajdonos felső oldalához
gfGrowHiX	$04	Illesztés a tulajdonos jobb oldalához
gfGrowHiY	$08	Illesztés a tulajdonos alsó oldalához
gfGrowAll	$0F	Illesztés a tulajdonos jobb-alsó sarkához
gfGrowRel	$10	A *view* mérete relatív a tulajdonoshoz képest



A view viselkedése mozgatás közben (jelölése: dmXXXX)

Konstans	Érték	Jelentés
dmDragMove	$01	A *view* mozgatható
dmDragGrow	$02	A *view* mérete megváltoztatható
dmLimitLoX	$10	Határátlépés ellenőrzése (bal oldal)
dmLimitLoY	$20	Határátlépés ellenőrzése (felső oldal)
dmLimitHiX	$40	Határátlépés ellenőrzése (jobb oldal)
dmLimitHiY	$80	Határátlépés ellenőrzése (alsó oldal)
dmLimitAll	$F0	Területkorlátozás a tulajdonos *view*-ra



A *TView DragMode* mezője tartalmazza a *dmLimitXX* megfelelő kombinációit. Alapértelmezésként a *TView.Init* a *dmLimitLoY*-t állítja be.

Az sbXXXX konstansok

Az sbXXXX konstansok definiálják azokat a különböző területeket, ahol az egérrel választani tudunk.

Konstans	Érték	Jelentés
sbLeftArrow	0	Vízszintes görgető sor bal nyila
sbRightArrow	1	Vízszintes görgető sor jobb nyila
sbPageLeft	2	Vízszintes görgető sor bal lapozó területe
sbPageRight	3	Vízszintes görgető sor jobb lapozó területe
sbUpArrow	4	Függőleges görgető sor felső nyila
sbDownArrow	5	Függőleges görgető sor alsó nyila
sbPageUp	6	Függőleges görgető sor felső lapozó területe
sbPageDown	7	Függőleges görgető sor alsó lapozó területe
sbIndicator	8	A görgető sor pozíciójelölője

A *TScrollBar.ScrollStep* tartalmazza a aktuális 0..8 értékek valamelyikét.

Szabványos görgető sor konstansok

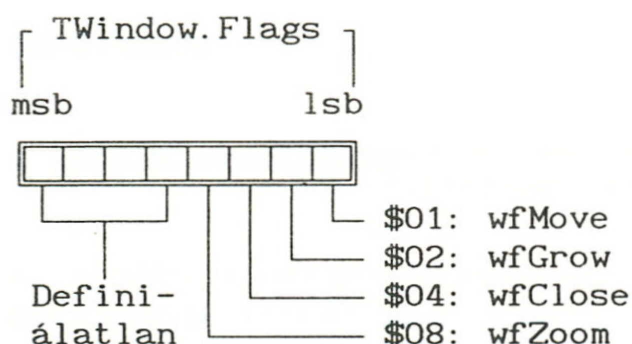
Ezeket a konstansokat a *TWindow.StandardScrollBar* tartalmazhatja.

Konstans	Érték	Jelentés
sbHorizontal	$0000	A görgető sor vízszintes
sbVertical	$0001	A görgető sor függőleges
sbHandleKeyboard	$0002	A görgető sor billentyűzetről vezérelhető

A wfXXXX konstansok

Ezek a konstansok a *TWindow* objektumok *Flags* mezőjének bitjeit reprezentálják. Ha az adott bit értéke egy az ablak rendelkezik a bit által jelölt attribútummal.

Konstans	Érték	Jelentés
wfMove	$01	Az ablak mozgatható a felső kereténél fogva
wfGrow	$02	Az ablakkeret rendelkezik újraméretező sarokkal (┘)
wfClose	$04	Az ablakkereten van ablakzáró doboz
wfZoom	$08	Az ablakkereten van nagyító doboz



A wpXXXX konstansok

Konstansok az ablakok szabványos színpalettájának kiválasztásához.

Konstans	Érték	Jelentés
wpBlueWindow	0	Kék alapon sárga szöveg
wpCyanWindow	1	Cián alapon kék szöveg
wpGrayWindow	2	Szürke alapon fekete szöveg

Alapértelmezés: *wpBlueWindow* paletta a *TWindow* objektumok számára
wpGrayWindow paletta a *TDialog* objektumok számára

TView szabványos parancsok

Konstans	Érték	Jelentés
cmReceivedFocus	50	A *view* rendelkezik fókusszal
cmReleasedFocus	51	A *view*-nak volt fókusza
cmCommandSetChanged	52	A parancskészlet megváltozásának jelzése
cmScrollBarChanged	53	A görgető sor értéke megváltozott
cmScrollBarClicked	54	A görgető sort egérrel kiválasztották
cmSelectWindowNum	55	Az ablak kiválasztható az *Alt+n* bill.
cmRecordHistory	56	A *history* lista tartalmazza az inputsor tartalmát is.

VÁLTOZÓK

Változó	Típus	Kezdő érték	Jelentés
MinWinSize	TPoint	(X: 16; Y: 6)	A minimális ablakméret
ShadowSize	TPoint	(X: 2; Y: 1)	Az ablak árnyékának a mérete
ShadowAttr	Byte	$08	Az ablak attribútuma

FÜGGVÉNY

Függvény	Művelet
Message	*View*-k közötti, felhasználó által definiált üzenetek továbbítása

A DIALOGS UNIT

A DIALOGS unit tartalmazza a dialógus dobozok konstruálásánál leggyakrabban használt elemek többségét.

Ezek az elemek (amelyek tulajdonképpen speciális ablakok) különböző vezérléseket valósítanak meg.

TÍPUSOK

Objektum típus	Felhasználása
TButton	Gomblenyomással történő parancsgenerálás
TCheckBoxes	on/off állású kapcsolók csoportja
TCluster	Absztrakt típus az opcióbeállító dobozokhoz (*check box*) és az állító gombokhoz (*radio button*)
TDialog	Speciális ablak a dialógus dobozokhoz
THistory	Az inputsor előzőleg beolvasott elemeinek listája
THistoryViewer	A history lista rendszer által használt
THistoryWindow	A history lista megjelenítőjét tartalmazza
TInputLine	Szöveges input editor
TLabel	Csoport vagy inputsor címke
TListBox	Lapozható lista, felhasználói kiválasztáshoz
TParamText	Formátumozott statikus szöveg
TRadioButtons	Gombok csoportja, amely gombok közül csak egyet lehet egyidejűleg kiválasztani
TStaticText	Egyszerű szöveg a dialógus dobozban

Más típusok	Felhasználása
TSItem	Sztring elemek a csoportok által használt láncolt listákban

KONSTANSOK

A bfXXXX konstansok

Az alábbi konstansok kombinációja a *TButton.Init* metódusnak adódik át, az újonnan létrehozott gombok stílusának meghatározására.

Konstans	Érték	Jelentés
bfNormal	$00	A gomb egy szokásos, de nem alapértelmezett gomb
bfDefault	$01	A gomb egy alapértelmezett gomb
bfLeftJust	$02	A gomb felirata balra kerül kiigazításra

ELJÁRÁSOK ÉS FÜGGVÉNYEK

Függvény	Művelet
NewSItem	Új sztringelem létrehozása a lista doboz számára

Eljárás	Művelet
RegisterDialogs	A DIALOGS unit minden objektumának regisztrálása stream használathoz

AZ APP UNIT

A forrásnyelven megtalálható APP unit a felhasználói keretrendszer alapelemeit tartalmazza. Négy nagyon fontos objektumtípus került definiálásra az APP unitban, köztük a *TProgram* és a *TApplication*, amelyek a Turbo Vision programok alapját képezik, ill. a *TDesktop*, amely biztosítja az ablaktechnika felhasználását.

TÍPUSOK

Objektumok	Felhasználása
TApplication	Alkalmazási objektum, amely rendelkezik esemény-, képernyő-, hiba- és memóriakezelővel
TBackground	A *desktop* színes háttere
TDeskTop	Objektumok csoportja, amely az ablakokat és a dialógus dobozokat tartalmazza
TProgram	Absztrakt alkalmazási objektum

VÁLTOZÓK

Változó	Típus	Kezdeti érték	Mutató az ...
Application	PProgram	nil	aktuális alkalmazáshoz
DeskTop	PDeskTop	nil	aktuális *desktop* objektumhoz
StatusLine	PStatusLine	nil	aktuális státuszsorhoz
MenuBar	PMenuView	nil	aktuális menüsorhoz.

A MENUS UNIT

A MENUS unit a tartalmazza a Turbo Vision menü-rendszerének működtetéséhez szükséges objektumokat és alprogramokat.

TÍPUSOK

Objektumok	Felhasználás
TMenuBar	Vízszintes sáv, amely a menüfejléceket tartalmazza
TMenuBox	Pull-down vagy pop-up menüdobozok
TMenuView	Absztrak objektum a menüsávokhoz és a menüdobozokhoz
TStatusLine	A felhasználói képernyő alján megjelenő üzenetsor, amely a *TStatusDef* rekordok listáját tartalmazza

Más típusok	Felhasználás
TMenu	*TMenuItem* rekordok láncolt listája
TMenuItem	Rekord, amely egy menühöz tartozó címkeszöveget, *hot key*-t, parancsot és *help context*-et kapcsolja össze
TMenuStr	Sztring típus a menücímkékhez
TStatusDef	Rekord, amely hozzákapcsolja a státuszsor elemeinek listájához a *help context*-ek valamely tartományát
TStatusItem	Rekord, amely a státuszsorban összekapcsolja a címkeszöveget, a *hot key*-t és a parancsot.

FÜGGVÉNYEK ÉS ELJÁRÁSOK

Függvény	Művelet	Típus
NewMenu	Menü létrehozása a heap-en	TMenu
NewItem	Új menüelem létrehozása	TMenuItem
NewLine	Új sor létrehozása a menüdobozban	TMenuItem
NewSubMenu	Menüsor vagy menüdoboz létrehozása	TMenuItem
NewStatusDef	Definiál egy *help context* tartományt és egy mutatót a státuszelemek listájához	TStatusLine
NewStatusKey	Definiál egy státuszsor elemet és hozzákapcsolja azt egy parancshoz, vagy opcionálisan egy *hot key*-hez	TStatusLine

Eljárások	Művelet	Típus
DisposeMenu	Felszabadítja a menü által lefoglalt heap területet	TMenu

A DRIVERS UNIT

A DRIVERS unit tartalmazza a Turbo Vision működéséhez szükséges specializált eszköz-vezérlőket. Ezen vezérlők feladatai a következők: egér- és billentyűzetkezelés, képernyőkezelés, a rendszer hibák kezelése és az események menedzselése az esemény-vezérelt programozáshoz.

TÍPUSOK

Típus	Felhasználás
TEvent	Eseményrekord típus
TSysErrorFunc	A rendszer hibakezelő függvényének típusa

KONSTANSOK

Az evXXXX esemény konstansok és maszkok

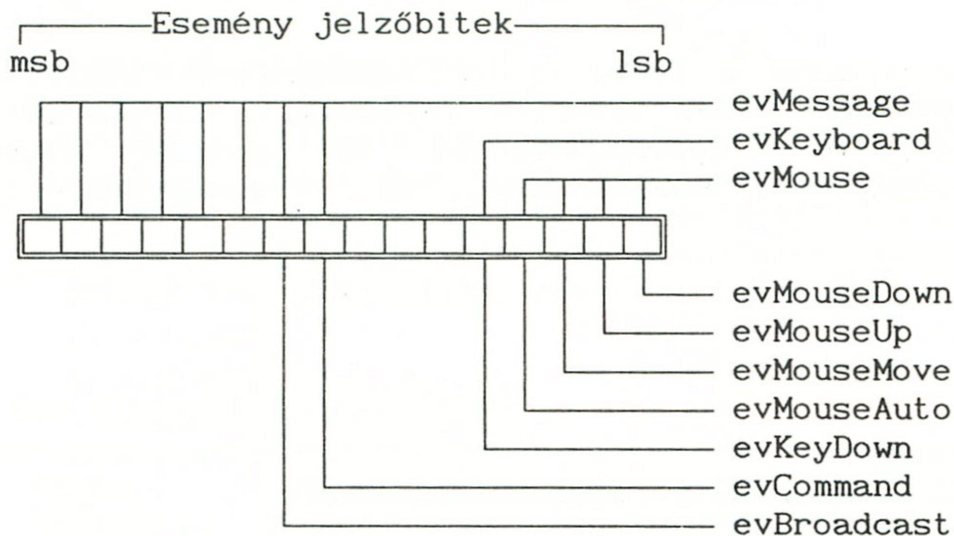
Ezek az értékek a bekövetkezett esemény típusát közlik a Turbo Vision eseménykezelőjével. Az *evXXXX* konstansok több helyen használhatók:
- Az esemény rekord *What* mezőjében
- Egy *view* objektum *EventMask* mezőjében
- A *PositionalEvents* és a *FocusedEvents* változókban

Esemény kódok:

Konstans	Érték	Jélentés
evMouseDown	$0001	Egér gomb lenyomva
evMouseUp	$0002	Egér gomb felengedve
evMouseMove	$0004	Az egér elmozdult
evMouseAuto	$0008	Az egér esemény automatikus ismétlése (mindaddig, amíg az egér gombja lenyomott állapotban van)
evKeyDown	$0010	Az esemény - billentyűlenyomás
evCommand	$0100	Az esemény - parancs
evBroadcast	$0200	Az esemény - adás

Esemény maszkok:

Kosntans	Érték	Jelentés
evNothing	$0000	Törölt esemény
evKeyboard	$0010	Billentyűzet által generált esemény
evMouse	$000F	Egér által generált esemény
evMessage	$FF00	Üzenet vagy parancs esemény



Az mbXXXX konstansok

Ezeket a konstansokat, mint maszkokat, használhatjuk az *evMouse* eseményrekord *TEvent.Buttons* mezőjének feldolgozásához.

Konstans	Érték	Jelentés
mbLeftButton	$01	A bal egérgomb volt lenyomva
mbRightButton	$02	A jobb egérgomb volt lenyomva

Billentyűzet állapot és shift maszkok (kbXXXX)

Konstans	Érték	Jelentés
kbRightShift	$0001	A jobb *SHIFT* van lenyomva
kbLeftShift	$0002	A bal *SHIFT* van lenyomva
kbCtrlShift	$0004	A *CTRL* és *SHIFT* vannak lenyomva
kbAltShift	$0008	Az *ALT* és *SHIFT* vannak lenyomva
kbScrollState	$0010	A *SCROLL LOCK* bekapcsolt állapotban van
kbNumState	$0020	A *NUM LOCK* bekapcsolt állapotban van
kbCapsState	$0040	A *CAPS LOCK* bekapcsolt állapotban van
kbInsState	$0080	Az *Insert* mód be van kapcsolva

ALT+betű billentyűkódok

Konstans	Érték	Konstans	Érték
kbAltA	$1E00	kbAltN	$3100
kbAltB	$3000	kbAltO	$1800
kbAltC	$2E00	kbAltP	$1900
kbAltD	$2000	kbAltQ	$1000
kbAltE	$1200	kbAltR	$1300
kbAltF	$2100	kbAltS	$1F00
kbAltG	$2200	kbAltT	$1400
kbAltH	$2300	kbAltU	$1600
kbAltI	$1700	kbAltV	$2F00
kbAltJ	$2400	kbAltW	$1100
kbAltK	$2500	kbAltX	$2D00
kbAltL	$2600	kbAltY	$1500
kbAltM	$3200	kbAltZ	$2C00

ALT+szám billentyűkódok

Konstans	Érték	Konstans	Érték
kbAlt1	$7800	kbAlt6	$7D00
kbAlt2	$7900	kbAlt7	$7E00
kbAlt3	$7A00	kbAlt8	$7F00
kbAlt4	$7B00	kbAlt9	$8000
kbAlt5	$7C00	kbAlt0	$8100

Speciális billentyűkódok

Konstans	Érték	Konstans	Érték
kbAltEqual	$8300	kbEnd	$4F00
kbAltMinus	$8200	kbEnter	$1C0D
kbAltSpace	$0200	kbEsc	$011B
kbBack	$0E08	kbGrayMinus	$4A2D
kbCtrlBack	$0E7F	kbHome	$4700
kbCtrlDel	$0600	kbIns	$5200
kbCtrlEnd	$7500	kbLeft	$4B00
kbCtrlEnter	$1C0A	kbNoKey	$0000
kbCtrlHome	$7700	kbPgDn	$5100
kbCtrlIns	$0400	kbPgUp	$4900
kbCtrlLeft	$7300	kbGrayPlus	$4E2B
kbCtrlPgDn	$7600	kbRight	$4D00
kbCtrlPgUp	$8400	kbShiftDel	$0700
kbCtrlPrtSc	$7200	kbShiftIns	$0500
kbCtrlRight	$7400	kbShiftTab	$0F00
kbDel	$5300	kbTab	$0F09
kbDown	$5000	kbUp	$4800

A funkció billentyűk kódjai

Konstans	Érték	Konstans	Érték
kbF1	$3B00	kbF6	$4000
kbF2	$3C00	kbF7	$4100
kbF3	$3D00	kbF8	$4200
kbF4	$3E00	kbF9	$4300
kbF5	$3F00	kbF10	$4400

SHIFT+funkció billentyűk kódjai

Konstans	Érték	Konstans	Érték
kbShiftF1	$5400	kbShiftF6	$5900
kbShiftF2	$5500	kbShiftF7	$5A00
kbShiftF3	$5600	kbShiftF8	$5B00
kbShiftF4	$5700	kbShiftF9	$5C00
kbShiftF5	$5800	kbShiftF10	$5D00

CTRL+funkció billentyűk kódjai

Konstans	Érték	Konstans	Érték
kbCtrlF1	$5E00	kbCtrlF6	$6300
kbCtrlF2	$5F00	kbCtrlF7	$6400
kbCtrlF3	$6000	kbCtrlF8	$6500
kbCtrlF4	$6100	kbCtrlF9	$6600
kbCtrlF5	$6200	kbCtrlF10	$6700

ALT+funkció billentyűk kódjai

Konstans	Érték	Konstans	Érték
kbAltF1	$6800	kbAltF6	$6D00
kbAltF2	$6900	kbAltF7	$6E00
kbAltF3	$6A00	kbAltF8	$6F00
kbAltF4	$6B00	kbAltF9	$7000
kbAltF5	$6C00	kbAltF10	$7100

Az smXXXX képernyő módok

Ezeket a konstansokat a *SetVideoMode* eljárás hívásakor használjuk a videomód értékének a *ScreenMode* változóba történő betöltéséhez.

Konstans	Érték	Jelentés
smBW80	$0002	Fekte-fehér képernyőmód
smCO80	$0003	Színes képernyőmód
smMono	$0007	Monokróm képernyőmód
smFont8x8	$0100	43- vagy 50-soros mód (EGA/VGA)

Szabványos parancskódok

Konstans	Érték	Jelentés
cmValid	0	Új *view* érvényességének ellenőrzése
cmQuit	1	Kilépés az alkalmazói programból
cmError	2	Semelyik objektum által sem kezelt parancs
cmMenu	3	A fókusz mozgatása a menüsorban
cmClose	4	Az aktuális ablak bezárása
cmZoom	5	Ablak nagyítása/kicsinyítése (*zoom*)
cmResize	6	Ablak újraméretezése
cmNext	7	Az utolsó ablak előtérbe helyezése
cmPrev	8	Az első ablak összes többi alá helyezése

TDialog szabványos parancsok

Konstans	Érték	Jelentés
cmOK	10	Az *OK* gomb volt kiválasztva
cmCancel	11	Kilépés a dialógus dobozból vagy a *CANCEL* gombbal, vagy a doboz lezáró ikonnal, vagy az *ESC* billentyűvel.
cmYes	12	A *YES* gomb volt lenyomva
cmNo	13	A *NO* gomb volt lenyomva
cmDefault	14	Az alapértelemzés szerinti gomb, vagy az *ENTER* billentyű volt lenyomva

VÁLTOZÓK

Kezdőértékkel rendelkező változók

Változó	Típus	Kezdeti Érték	Jelentés
ButtonCount	Byte	0	Az egér gombjainak száma
MouseEvents	Boolean	False	Jelzi, hogy van-e egér vagy nincs
DoubleDelay	Word	8	A dupla klikkentés közötti max. idő
RepeatDelay	Word	8	Várakozás az automatikus egér ismétlés megkezdése előtt

Kezdőértékkel nem rendelkező változók

Változó	Típus	Jelentés
MouseIntFlag	Byte	Csak belső használatra!
MouseButtons	Byte	Melyik egérgomb volt lenyomva
MouseWhere	TPoint	Az egérkurzor poziciója
StartupMode	Word	A képernyőmód a program indításkor
ScreenMode	Word	Aktuális képernyőmód
ScreenWidth	Byte	A képernyő szélessége (oszlopok száma)
ScreenHeight	Byte	A képernyő magassága (a sorok száma)
CheckSnow	Boolean	"Havazás" ellenőrzés szükséges-e (CGA)
HiResScreen	Boolean	A képernyő sorok száma lehet 43 vagy 50 (EGA/VGA)
ScreenBuffer	Pointer	Mutató a képernyőpufferre
CursorLines	Word	Kurzortípus beállításhoz a kezdeti és az utolsó *scan* sor beállítása

A rendszerhibák kezelőjének változói

Változó	Típus	Kezdeti Érték	Jelentés
SysErrorFunc	TSysErrorFunc	SystemError	Ezt a függvényt rendszerhiba esetén hívja meg a hibakezelő
SysColorAttr	Word	$4E4F	Hibaüzenetek attribútuma színes képernyőn
SysMonoAttr	Word	$7070	Hibaüzenetek attribútuma monokróm képernyőn
CtrlBreakHit	Boolean	False	*CTRL-BREAK* lenyomásának jelzése
SaveCtrlBreak	Boolean	False	A *CTRL-BREAK* ellenőrzés állapota a program indításkor

FÜGGVÉNYEK ÉS ELJÁRÁSOK

Eseménykezelő eljárások

Eljárás	Művelet
InitEvents	Az eseménykezelő inicializálása
DoneEvents	Az eseménykezelő lezárása
ShowMouse	Az egérkurzor megjelenítese
HideMouse	Az egérkurzor kikapcsolása
GetMouseEvent	Eseményrekord felépítése egérműveletből
GetKeyEvent	Eseményrekord felépítése billentyűzet inputból

Képernyőkezelő eljárások

Eljárás	Művelet
InitVideo	A képernyőkezelő inicializálása
DoneVideo	A képernyőkezelő lezárása
SetVideoMode	Képernyőmód kiválasztása
ClearScreen	Képernyőtörlés bármely videomódban

Puffermozgató eljárások

Eljárás	Művelet
MoveBuf	Puffer átmásolása egy másik pufferbe
MoveChar	Puffer feltöltése karakterekkel
MoveCStr	Vezérlő sztring bemásolása pufferbe
MoveStr	Sztring másolása pufferbe

Sztring formattáló eljárás

Eljárás	Művelet
FormatStr	Sztring formattálása

Rendszerhibákat kezelő eljárások

Eljárás	Művelet
InitSysError	A rendszerhibák kezelőjének inicializálása
DoneSysError	A rendszerhibák kezelőjének lezárása

Alapértelmezés szerinti rendszerhibakezelő függvény

Függvény	Művelet
SystemError	Hibajelzés a képernyő alsó sorában és válasz-várás az Abort/Retry kérdésre

Billentyűzetet kezelő függvények

Függvény	Művelet
GetAltChar	Billentyű karakterkód lekérdezése
GetAltCode	Billentyű scan-kód lekérdezése

Sztringhossz függvény

Függvény	Művelet
CStrLen	A vezérlő sztringek hosszát adja vissza hullámjel (~) nélkül

Alapinicializálás

Eljárás	Művelet
InitDrivers	A *Drivers* unit inicializálása

A TEXTVIEW unit több specializált *view*-t tartalmaz görgethető ablakban történő szövegmegjelenítéshez.

TÍPUSOK

Objektum típus	Felhasználás
TTerminal	TTY-szerű szöveggörgető egység
TTextDevice	Absztrakt szöveges egység objektum

Más típus	Felhasználás
TTerminalBuffer	Kör-körös szövegpuffer a *TTerminal* objektumhoz

ELJÁRÁS

Eljárás	Művelet
AssignDevice	Szöveges file egység megfeleltetése az input-nak és/vagy az output-nak

A MEMORY UNIT

A MEMORY unit a Turbo Vision memóriakezelő rutinjait tartalmazza, amelyek támogatják a biztonságos (safe) programozást.

VÁLTOZÓ

Változó	Type	Kezdeti Érték	Jelentés
LowMemSize	Word	4096 div 16	A biztonsági terület mérete

FÜGGVÉNYEK ÉS ELJÁRÁSOK

Eljárás	Művelet
InitMemory	A memóriakezelő inicializálása
DoneMemory	A memóriakezelő lezárása
GetBufMem	*Cache* puffer foglalás egy csoport számára
FreeBufMem	Csoport *cache* pufferének felszabadítása
SetMemTop	Alkalmazás memóriahatárának beállítása

Függvény	Művelet
LowMemory	Jelzi,ha a biztonsági terület mérete lecsökken
MemAlloc	Memóriafoglalás a biztonsági terület figyelembevételével

A HISTLIST UNIT

A HISTLIST unit a *history* lista megvalósításához szükséges változókat, függvényeket és eljárásokat tartalmazza.

VÁLTOZÓK

Változó	Típus	Kezdeti Érték	Jelentés
HistoryBlock	Pointer	nil	Memóriapuffer a *history* lista elemeinek tárolására
HistorySize	Word	1024	A *history* blokk mérete
HistoryzUsed	Word	0	Offszet a *history* blokkon belül, ami jelzi a használt blokk méretét

FÜGGVÉNYEK ÉS ELJÁRÁSOK

Eljárás	Művelet
HistoryAdd	Sztring hozzákapcsolása a *history* listához
ClearHistory	Az összes *history* lista törlése
InitHistory	A *history* lista kezelőjének inicializálása
DoneHistory	A *history* lista kezelőjének lezárása

Függvény	Művelet
HistoryCount	A *history* listában tárolt sztringek számát adja vissza
HistoryStr	Index alapján visszaad egy sztringet a *history* listából

F2. Függelék Turbo Pascal 6.0 fordítási direktívák

A Turbo Pascal fordítási lehetőségeinek egy része fordítási direktívák felhasználásával vezérelhetők. A fordítási direktíva, mint speciális megjegyzés jelenik meg a Pascal programban, és minden olyan helyre elhelyezhető, ahol utasítás állhat. A fordítási 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 tartományellenőrzés 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 parancsor fordító (TPC.EXE) számára */direktíva* alakban vagy a *TPC.CFG* konfigurációs file-ban.

F2.1 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

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 8088 µP, 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

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

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 hiba forrás file-beli helyének meghatározásához szükséges adatokat.

$E - 80x87 emuláció

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

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

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 mikroproceszorok ismernek. Ezekkel az utasításokkal általában gyorsabb kód állítható elő. A felhasznált új 80286 utasítások: ENTER, LEAVE, PUSH érték, kibővített IMUL, és a kibővített SHL / SHR.

$I - Az input/output műveletek ellenőrzése

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

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 mértét. 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

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 {$N+} állapotban minden valós típusú számításhoz a 8087 aritmetikai társprocesszor számára generál kódokat a fordító. Ha nincs a gépünkben 80x87 processzor akkor emuláltatnuk kell annak működését a {$E+} kapcsoló megadásával.

$O Overlay kódgenerálás

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

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

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

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

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).

F2.2 Kapcsoló direktívák és a Turbo Pascal IDE megfelelőik

direktíva	megfelelője a *Compiler Options* dialógus dobozban	
{$A+}	Code Generation	[X] Word Align Data
{$A-}	Code Generation	[ ] Word Align Data
{$B+}	Syntax	[X] Complete Boolean Evaluation
{$B-}	Syntax	[ ] Complete Boolean Evaluation
{$D+}	Debugging	[X] Debug Information
{$D-}	Debugging	[ ] Debug Information
{$E+}	Numeric Processing	[X] Emulation
{$E-}	Numeric Processing	[ ] Emulation
{$F+}	Code Generation	[X] Force FAR Calls
{$F-}	Code Generation	[ ] Force FAR Calls
{$G+}	Code Generation	[X] 286 Instructions
{$G-}	Code Generation	[ ] 286 Instructions
{$I+}	Run-Time Errors	[X] I/O Checking
{$I-}	Run-Time Errors	[ ] I/O Checking
{$L+}	Debugging	[X] Local Symbols
{$L-}	Debugging	[ ] Local Symbols
{$N+}	Numeric Processing	[X] 8087/80287
{$N-}	Numeric Processing	[ ] 8087/80287
{$O+}	Code Generation	[X] Overlays Allowed
{$O-}	Code Generation	[ ] Overlays Allowed
{$R+}	Run-Time Errors	[X] Range Checking
{$R-}	Run-Time Errors	[ ] Range Checking
{$S+}	Run-Time Errors	[X] Stack Checking
{$S-}	Run-Time Errors	[ ] Stack Checking
{$V+}	Syntax	[X] Strict Var-Strings
{$V-}	Syntax	[ ] Strict Var-Strings
{$X+}	Syntax	[X] Extended Syntax
{$X-}	Syntax	[ ] Extended Syntax

F2.3 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

Tipus : lokális
Menüből : *Include Directories*

A fordító a megadott forrásállomány tartalmát beépíti a direktíva helyére. Az alapértelmezés szerinti kiterjesztés .PAS. Az állományok maximális 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

Tipus : lokális
Menüből : *Object Directories*

A direktíva utasítja a fordítót, hogy az adott nevű tárgykódú file-t összeszerkessze az éppen fordított programmal vagy unit-tal. Ezt a direktívát assembler 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

Alapértelmezés: {$M 16384,0,655360}
Típus : globális
Menüből : *Memory Sizes*

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

Típus : lokális
Menüből : -

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} direktívákat közvetlenül a program **uses** utasítása után kell elhelyeznünk. A direktíva nincs hatással unit-ok fordítására.

F2.3 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:

{**$IF**xxx} .I.. {**$ENDIF**}
vagy
{**$IF**xxx} .I.. {**$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 fordításai szerkezetekre:

```
{Ha a Debug szimbólum definiált, kiírja 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}
```

F2.3.1 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 érvényes megkötések érvényesek.

A Turbo Pascal az alábbi szabványos feltételes szimbólumokat definiálja:

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 társprocesszor

F2.3.3 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.

```
pl.:{ 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.

F3. Függelék A 8086/80286 és a 8087/80287 processzorok utasításkészlete

F3.1 8086 és 80286 valós (real) mód

Jelölések:

src	- forrás operandus
src8	- 8 bites forrás operandus
src16	- 16 bites forrás operandus
dest	- cél operandus
reg/mem	- regiszter vagy memória operandus
flag	- állapotregiszter bitje
port8	- 8 bites periféria portcím
reg16	- 16 bites regiszter
mem16	- 16 bites memória hivatkozás
/286/	- csak 80286 proceszorral, {$G+} kapcsolóval használható

Aritmetikai és logikai utasítások

```
ADD  dest,src            dest ← (src + dest)
ADC  dest,src            dest ← (src + dest + CF)
INC  dest                dest ← (dest + 1)

SUB  dest,src            dest ← (dest - src)
SBB  dest,src            dest ← (dest - src - CF)
DEC  dest                dest ← (dest - 1)
CMP  dest,src            dest ↔ src összehasonlítás kivonással
NEG  dest                dest ← (0 - dest)  (2-es komplemens)


MUL  src                 az AL(AX) előjel nélküli szorzása (reg/mem)-vel
                         AX ← (AL * src8)
                         DX:AX ← (AX * src16)
IMUL src                 az  AL(AX) előjeles szorzása (reg/mem)-vel
                         AX ← (AL * src8)
                         DX:AX ← (AX * src16)
IMUL reg16,r/m,immed     reg16 ← (r/m8 * immed)                   \286\
                         reg16 ← (r/m16 * immed)
DIV  src                 előjel nélküli osztás (reg/mem)-vel
                         AL ← (AX / src8) , AH ← (AX MOD src8)
                         AX ← (DX:AX / src16) , DX ← (DX:AX MOD src16)
IDIV src                 az AX(DX:AX) előjeles osztása (reg/mem)-vel
                         AL ← (AX / src8) , AH ← (AX MOD src8)
                         AX ← (DX:AX / src16) , DX ← (DX:AX MOD src16)
```

```
AAA                  AL ← AL ASCII kiigazítása összeadás után
AAS                  AL ← AL ASCII kiigazítása kivonás után
AAM                  AH ← AL/10 , AL ← AL  mod 10
                     ASCII kiigazítása szorzás után
AAD                  AL ← AH * 10 + AL , AH ←0
                     ASCII kiigazítása osztás előtt
DAA                  AL ← AL BCD kiigazítása összeadás után
DAS                  AL ← AL BCD kiigazítása kivonás után


CBW                  AX ← AL előjel kiterjesztéssel
CWD                  DX:AX ← AX előjel kiterjesztéssel


AND  dest,src        dest ← (dest & src)  logikai és
TEST dest,src        flag-ek ← (dest & src eredménye elapján)
OR   dest,src        dest ← (dest | src)  logikai vagy
XOR  dest,src        dest ← (dest ^ src)  logikai kizáró vagy
NOT  dest            dest ← dest 1-es komplemense
```

Adatmozgató utasítások

```
MOV  dest,src        dest ← src
XCHG dest,src        dest csere src

IN  Ax, port8(DX)    byte: AL ← [port]
                     word: AL ← [port] ,  AH ← [port+1]
OUT  port8(DX),Ax    byte: [port] ← AL
                     word: [port] ← AL   [port+1] ← AH

XLAT                 AL ← DS:[BX+AL]
LEA  reg16,addr      reg16 ← (addr effektív címe)
LDS  reg16,mem       reg16 ← [mem16] ,  DS ← [mem16+2]
LES  reg16,mem       reg16 ← [mem16] ,  ES ← [mem16+2]

LAHF                 AH ← flag-reg alsó byte (SZAPC)
SAHF                 flag-reg alsó byte ← AH
PUSH src             SP=SP-2 , SS:[SP] ← src
PUSH érték           SP=SP-2 , SS:[SP] ← 16 bites érték           \286\
PUSHA                SP=SP+10H , PUSH AX,BX,CX,DX,SI,DI,BP,SP     \286\
PUSHF                SP=SP-2 , SS:[SP] ← flag-ek
POP  dest            dest ← SS:[SP] , SP=SP+2
POPA                 POP SP,BP,DI,SI,DX,CX,BX,AX  , SP=SP+10H     \286\
POPF                 flag-reg ← SS:[SP] , SP=SP+2
```

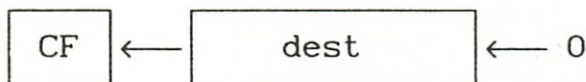
Vezérlésátadó utasítások

```
CALL     címke          eljárás hívás
                        Ha FAR CALL (inter-szegmens)
                           PUSH CS
                           CS ← cél_seg
                        PUSH IP
                        IP ← cél_offszet
RET      opcionális érték
                        POP IP
                        Ha FAR RETURN (inter-szegmens)
                           POP CS
                        SP ← SP + opcionális érték
JMP      címke          feltétel nélküli ugrás
                            short: IP ← (IP+(előjeles cél eltolás))
                             near: IP ← (IP+(16 bites cél eltolás))
                         indirekt: IP ← (regiszter vagy memória)
                              far: SS ← cél_seg , IP ← cél_offset

JCXZ     short_címke    ugrás ha CX=0

LOOP     short_címke    CX ← (CX-1) ugrás ha CX<>0
LOOPE    short_címke    CX ← (CX-1) ugrás ha CX<>0 és ZF=1
LOOPZ
LOOPNE   short_címke    CX ← (CX-1) ugrás ha CX<>0 és ZF=0
LOOPNZ

Jfelt.   short_címke    ugrás ha a feltétel teljesül
                        IP ← (IP+(8 bites eltolás ))

JE/JZ    short_címke    ugrás ha  =  (ZF=1)
JNE/JNZ  short_címke    ugrás ha  <> (ZF=0)
JA/JNBE  short_címke    ugrás ha  >  (CF=0 és ZF=0 előjel nélkül)
JAE/JNB  short_címke    ugrás ha  ≥  (CF=0 előjel nélkül)
JB/JNAE  short_címke    ugrás ha  <  (CF=1 előjel nélkül)
JBE/JNA  short_címke    ugrás ha  ≤  (CF=1 vagy ZF=1 előjel nélkül)
JG/JNLE  short_címke    ugrás ha  >  (SF=OF és ZF=0 előjeles)
JGE/JNL  short_címke    ugrás ha  ≥  (SF=OF előjeles)
JL/JNGE  short_címke    ugrás ha  <  (SF<>OF előjeles)
JLE/JNG  short_címke    ugrás ha  ≤  (SF<>OF és ZF=1 előjeles)

JC       short_címke    ugrás ha  (CF=1)
JNC      short_címke    ugrás ha  (CF=0)
JO       short_címke    ugrás ha  túlcsordult (OF=1)
JNO      short_címke    ugrás ha  nincs túlcsordulás (OF=0)
JP/JPE   short_címke    ugrás ha  paritás páros  (PF=1)
JNP/JPO  short_címke    ugrás ha  paritás páratlan (PF=0)
JS       short_címke    ugrás ha  negatív (SF=1)
JNS      short_címke    ugrás ha  nem negatív (SF=0)
```

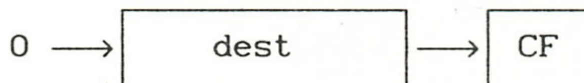
BOUND	reg16, mem	ha (reg16 < DS:[mem]) vagy (reg16 > DS:[mem+2]) , akkor INT 5 \286\
ENTER	méret, szint	paraméter átadáshoz veremterület kijelölése \286\
LEAVE		az ENTER által lefoglalt veremterület felszabadítása \286\
INT szám		szoftveres megszakítás hívása PUSHF, IF← 0, TF← 0, PUSH CS , PUSH IP , IP ← 0000:[szám * 4]; CS ← 0000:[(szám * 4) + 2]
INTO		ha OF=1, akkor INT 4
INT		INT 3
IRET		visszatérés megszakításból POP IP, POP CS, POPF

Bitforgató és biteltoló utasítások

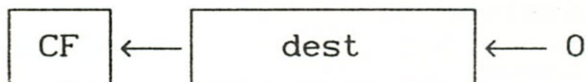
SHL dest,1/CL Logikai eltolás balra



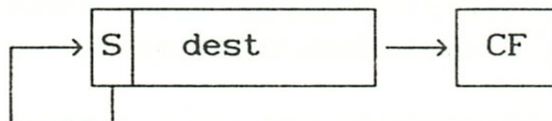
SHR dest,1/CL Logikai eltolás jobbra



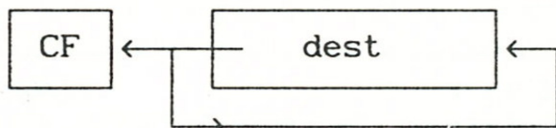
SAL dest,1/CL Aritmetikai eltolás balra



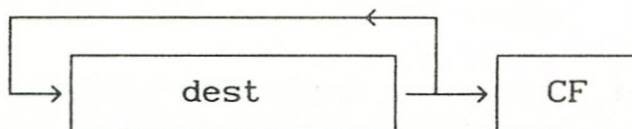
SAR dest,1/CL Aritmetika eltolás jobbra



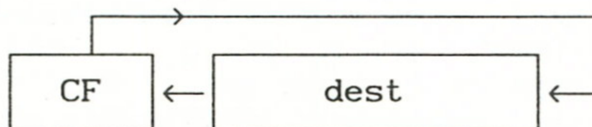
ROL dest,1/CL Forgatás balra



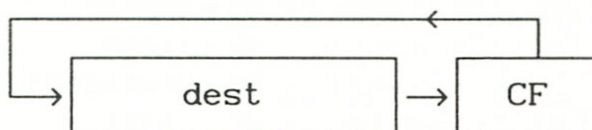
ROR dest,1/CL   Forgatás jobbra



RCL dest,1/CL   Forgatás balra CF-fel



RCR dest,1/CL   Forgatás jobbra CF-fel



SHL/SHR/SAL/SAR dest,db   \286\
ROL/ROR/RCL/RCR dest,db   \286\
\286\-nál megadható a lépések száma: db=0 - 31

Sztringkezelő utasítások

Jelölések: n = 1 (byte) vagy 2 (szó)
± = + (DF=0) vagy - (DF=1)

REP/REPE/REPZ   (prefixum)
ameddig CX <> 0   (MOVS,LODS,STOS)
string utasítás
CX ← CX-1

---

ameddig CX <> 0   (CMPS,SCAS)
string utasítás
CX ← CX-1
ha ZF = 0 az ismétlés befejezése

REPNE/REPNZ   (prefixum)
ameddig CX <> 0   (MOVS,LODS,STOS)
string utasítás
CX ← CX-1

---

ameddig CX <> 0   (CMPS;SCAS)
string utasítás
CX ← CX-1
ha ZF = 1 az ismétlés befejezése

```
MOVSB          ES:[DI] ← DS:[SI]  (byte vagy szó)
MOVSW          SI ← SI ± n
               SI ← SI ± n

LODSB          AL ← DS:[SI]
               SI ← SI ± 1
LODSW          AX ← DS:[SI]
               SI ← SI ± 2


STOSB          ES:[DI] ← AL
               DI ← DI ± 1
STOSW          ES:[DI] ← AX
               DI ← DI ± 2

CMPSB          flag-ek ← (eredménye a CMP DS:[SI],ES:[DI])
CMPSW          SI ← SI ± n
               DI ← DI ± n

SCASB          flag-ek ← (eredménye a CMP ES:[DI],AL)
               DI ← DI ± 1
SCASW          flag-ek ← (eredménye a CMP ES:[DI],AX)
               DI ← DI ± 2

INSB           ES:[DI] ← ( byte vagy szó a DX portról)            \286\
INSW           DI ← DI ± n
OUTSB          (DX portra) ← DS:[SI] (byte vagy szó)              \286\
OUTSW          SI ← SI ± n
```

80x86 vezérlő utasítások

```
CLC            CF ← 0
CMC            CF ← CF komplemense
STC            CF ← 1
CLD            DF ← 0 (string utasítások automatikus növelés)
STD            DF ← 1 (string utasítások automatikus csökkentés)
CLI            IF ← 0 (hardware megszakítások tiltása)
STI            IF ← 1 (hardware megszakítások engedélyezése)

HLT            a processzor leállítása egy hardver IT-ig
WAIT           várakozás a TEST (BUSY) vonalak aktív állapotára
ESC opkód,src
                utasítás átadása valamely társprocesszornak
LOCK           (prefixum) a buszok lezárása az utasítás végrehajtásának
               idejére
segreg:        (prefixum) az alapértelmezett szegmenskijelölés átállí-
               tása
```

F3.2 80286 védett (protected) mód

Csak a {G+} fordítási kapcsoló megadásával használhatók.

Jelölések:

src	- forrás operandus
dest	- cél operandus
reg16	- 16 bites regiszter
mem16	- 16 bites memória hivatkozás
seg	- szegmensregiszter
GDT	- Globális leírótábla (Global Descriptor Table)
LDT	- Lokális leírótábla (Local Descriptor Table)
IDT	- Megszakítás leírótábla (Interrupt Descriptor Table)
MSW	- Gépi állapotszó (Machine Status Word)

LGDT	src	GDT regiszter betöltése 6 byte-os memóriaterületről
SGDT	dest	GDT regiszter kimentése 6 byte-os memóriaterületre
LIDT	src	IDT regiszter betöltése 6 byte-os memóriaterületről
SIDT	dest	IDT regiszter kimentése 6 byte-os memóriaterületre
LLDT	src	LDT regiszter (GDT szelektor) betöltése (reg16/mem16)
SLDT	dest	LDT regiszter tárolása (reg16/mem16)
LMSW	src	MSW betöltése (reg16/mem16)
SMSW	dest	MSW tárolása (reg16/mem16)
LTR	src	Taszkregiszter betöltése (reg16/mem16)
STR	dest	Taszkregiszter tárolása (reg16/mem16)
LAR	dest,src	A hozzáférési jogokat tartalmazó byte betöltése
LSL	dest,src	A szegmenshatár betöltése
ARPL	r/m16,r16	A szelektor RPL mezőjének beállítás
VERR	seg	Szegmens ellenőrzése íráshoz
VERW	seg	Szegmens ellenőrzése olvasáshoz

F3.2 8087/80287 aritmetikai processzor

Csak az {N+} fordítási kapcsoló megadásával használhatók.

Jelölések:

ST(i)	a lebegőpontos stack i. eleme; (ST(0)=ST, ST(1),..)
mem10r	memóriában tárolt 10 byte-os lebegőpontos szám
mem8r	memóriában tárolt 8 byte-os lebegőpontos szám
mem4r	memóriában tárolt 4 byte-os lebegőpontos szám
mem10d	memóriában tárolt 10 byte-os BCD egész szám
mem4i	memóriában tárolt 4 byte-os hosszú egész szám
mem2i	memóriában tárolt 2 byte-os rövid egész szám
mem14 mem94	14 és 94 byte-os pufferek a 8087 állapotának tárolása

Adatmozgató utasítások

FLD ST(i)	push, ST(0) ← előző ST(i)
FLD mem10r	push, ST(0) ← mem10r
FLD mem4r	push, ST(0) ← mem4r
FLD mem8r	push, ST(0) ← mem8r
FILD mem2i	push, ST(0) ← mem2i
FILD mem4i	push, ST(0) ← mem4i
FILD mem8i	push, ST(0) ← mem8i
FBLD mem10d	push, ST(0) ← mem10d
FST ST(i)	ST(i) ← ST(0)
FST mem4r	mem4r ← ST(0)
FST mem8r	mem8r ← ST(0)
FIST mem2i	mem2i ← ST(0)
FIST mem4i	mem4i ← ST(0)
FSTP ST(i)	ST(i) ← ST(0), pop
FSTP mem10r	mem10r← ST(0), pop
FSTP mem4r	mem4r ← ST(0), pop
FSTP mem8r	mem8r ← ST(0), pop
FISTP mem2i	mem2i ← ST(0), pop
FISTP mem4i	mem4i ← ST(0), pop
FISTP mem8i	mem8i ← ST(0), pop
FBSTP mem10d	mem10d← ST(0), pop
FXCH ST(i)	ST(0) ↔ ST(i) (felcserélése)
FXCH	ST(0) ↔ ST(1) (felcserélése)
FLDL2E	push, ST(0) ← $\log_2$ e
FLDL2T	push, ST(0) ← $\log_2$ 10

FLDLG2	push, ST(0) ← $\log_2 2$ (lg 2)
FLDLN2	push, ST(0) ← $\log_n 2$ (ln 2)
FLDPI	push, ST(0) ← Π
FLD1	push, ST(0) ← +1.0
FLDZ	push, ST(0) ← +0.0

A négy alapművelet

FADD	ST(1) ← ST(1) + ST(0), pop
FADD ST(i)	ST(0) ← ST(0) + ST(i)
FADD ST(0),ST(i)	ST(0) ← ST(0) + ST(i)
FADD ST(i),ST(0)	ST(i) ← ST(i) + ST(0)
FADDP ST(i),ST(0)	ST(i) ← ST(i) + ST(0), pop
FADD mem4r	ST(0) ← ST(0) + mem4r
FADD mem8r	ST(0) ← ST(0) + mem8r
FIADD mem2i	ST(0) ← ST(0) + mem4i
FIADD mem4i	ST(0) ← ST(0) + mem2i
FSUB	ST(1) ← ST(1) - ST(0), pop
FSUB ST(i)	ST(0) ← ST(0) - ST(i)
FSUB ST(0),ST(i)	ST(0) ← ST(0) - ST(i)
FSUB ST(i),ST(0)	ST(i) ← ST(i) - ST(0)
FSUBP ST(i),ST(0)	ST(i) ← ST(i) - ST(0), pop
FSUB mem4r	ST(0) ← ST(0) - mem4r
FSUB mem8r	ST(0) ← ST(0) - mem8r
FISUB mem2i	ST(0) ← ST(0) - mem2i
FISUB mem4i	ST(0) ← ST(0) - mem4i
FSUBR	ST(1) ← ST(0) - ST(1), pop
FSUBR ST(i)	ST(0) ← ST(i) - ST(0)
FSUBR ST(0),ST(i)	ST(0) ← ST(i) - ST(0)
FSUBR ST(i),ST(0)	ST(i) ← ST(0) - ST(i)
FSUBRP ST(i),ST(0)	ST(i) ← ST(0) - ST(i), pop
FSUBR mem4r	ST(0) ← mem4r - ST(0)
FSUBR mem8r	ST(0) ← mem8r - ST(0)
FISUBR mem2i	ST(0) ← mem2i - ST(0)
FISUBR mem4i	ST(0) ← mem4i - ST(0)
FMUL	ST(1) ← ST(1) * ST(0), pop
FMUL ST(i)	ST(0) ← ST(0) * ST(i)
FMUL ST(0),ST(i)	ST(0) ← ST(0) * ST(i)
FMUL ST(i),ST(0)	ST(i) ← ST(i) * ST(0)
FMULP ST(i),ST(0)	ST(i) ← ST(i) * ST(0), pop
FMUL mem4r	ST(0) ← ST(0) * mem4r
FMUL mem8r	ST(0) ← ST(0) * mem8r
FIMUL mem2i	ST(0) ← ST(0) * mem2i
FIMUL mem4i	ST(0) ← ST(0) * mem4i

```
FDIV                  ST(1) ← ST(1) / ST(0), pop
FDIV   ST(i)          ST(0) ← ST(0) / ST(i)
FDIV   ST(0),ST(i)    ST(0) ← ST(0) / ST(i)
FDIV   ST(i),ST(0)    ST(i) ← ST(i) / ST(0)
FDIVP  ST(i),ST(0)    ST(i) ← ST(i) / ST(0), pop
FDIV   mem4r          ST(0) ← ST(0) / mem4r
FDIV   mem8r          ST(0) ← ST(0) / mem8r
FIDIV  mem2i          ST(0) ← ST(0) / mem2i
FIDIV  mem4i          ST(0) ← ST(0) / mem4i

FDIVR                 ST(1) ← ST(0) / ST(1), pop
FDIVR   ST(i)         ST(0) ← ST(i) / ST(0)
FDIVR   ST(0),ST(i)   ST(0) ← ST(i) / ST(0)
FDIVR   ST(i),ST(0)   ST(i) ← ST(0) / ST(i)
FDIVRP  ST(i),ST(0)   ST(i) ← ST(0) / ST(i), pop
FDIVR   mem4r         ST(0) ← mem4r / ST(0)
FDIVR   mem8r         ST(0) ← mem8r / ST(0)
FIDIVR  mem2i         ST(0) ← mem2i / ST(0)
FIDIVR  mem4i         ST(0) ← mem4i / ST(0)
```

Vizsgálatok

```
FCOM                  CMP ST(0) , ST(1)
FCOM   ST(i)          CMP ST(0) , ST(i)
FCOM   mem4r          CMP ST(0) , mem4r
FCOM   mem8r          CMP ST(0) , mem8r

FCOMP                 CMP ST(0) , ST(1), pop
FCOMP  ST(i)          CMP ST(0) , ST(i), pop
FCOMP  mem4r          CMP ST(0) , mem4r, pop
FCOMP  mem8r          CMP ST(0) , mem8r, pop

FCOMPP                CMP ST(0) , ST(1), pop, pop

FICOM   mem2i         CMP ST(0) , mem2i
FICOM   mem4i         CMP ST(0) , mem4i
FICOMP  mem2i         CMP ST(0) , mem2i, pop
FICOMP  mem4i         CMP ST(0) , mem4i, pop
```

C3	C2	C1	C0	
0	*0*	*?*	*0*	*ST > op*
0	*0*	*?*	*1*	*ST < op*
1	*0*	*?*	*0*	*ST = op*
1	*1*	*?*	*1*	*ST nem hasonlítható össze*

FTST	CMP ST(0) , 0.0

C3	C2	C1	C0	
0	*0*	*?*	*0*	*ST > 0.0*
0	*0*	*?*	*1*	*ST < 0.0*
1	*0*	*?*	*0*	*ST = 0.0*
1	*1*	*?*	*1*	*ST nem hasonlítható össze*

FXAM	C3 -- C0 ← ST(0) típusa

Függvények

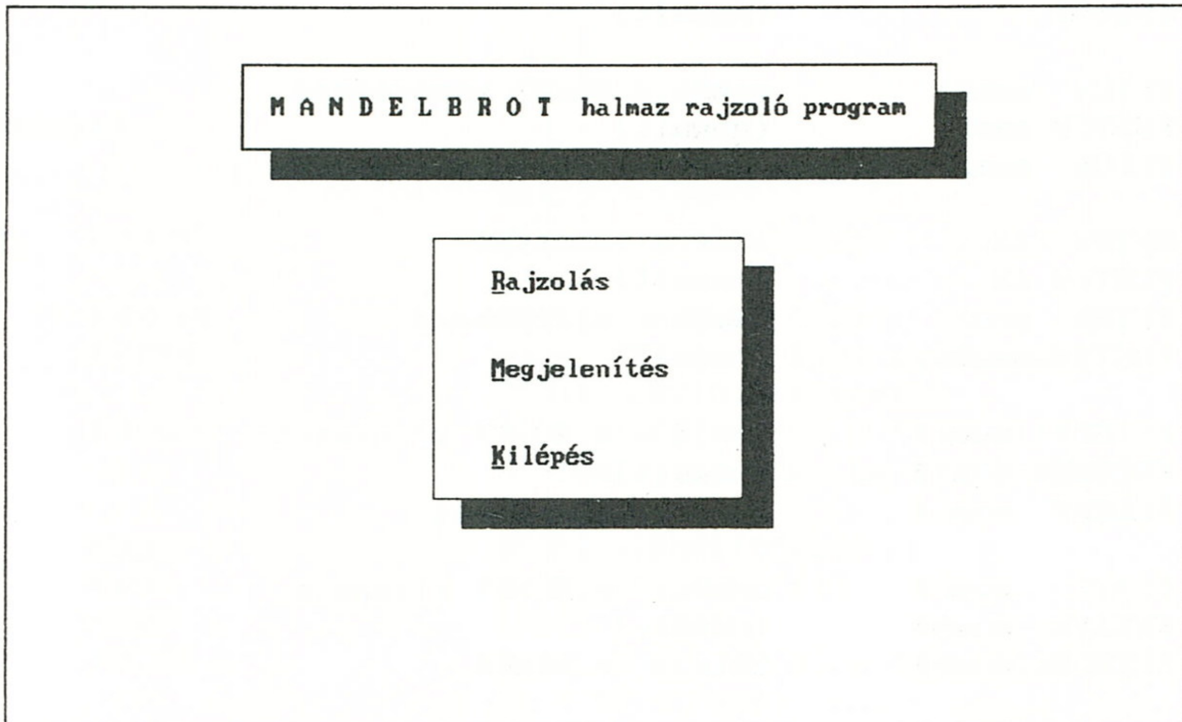
FPREM	ST(0) ← REPEAT(ST(0) - ST(1)) (maradék)
FRNDINT	ST(0) ← round(ST(0))
FXTRACT	push, ST(1) ← ST(0) exponense, ST(0) ← ST(0) törtrésze
FSCALE	ST(0) ← ST(0) * 2.0 ** ST(1) (FXTRACT inverze, ha ST(1)=egész)
FSQRT	ST(0) ← sqrt(ST(0))
FABS	ST(0) ← abs(ST(0))
FCHS	ST(0) ← -ST(0)
FPTAN	push, ST(1)/ST(0) ← TAN(előző ST(0)) (0 < ST < PI/4)
FPATAN	ST(0) ← ARCTAN(ST(1)/ST(0)), pop (az FPTAN inverze)
F2XM1	ST(0) ← (2.0 ** ST(0)) - 1.0 (0 < ST < 0.5)
FYL2X	ST(0) ← ST(1) * $\log_2$ ST(0), pop
FYL2XP1	ST(0) ← ST(1) * $\log_2$ (ST(0)+1.0), pop (0 < ST < 1-SQRT(2)/2)

A 80x87 vezérlése

Utasítás	Leírás	
FINIT	a 80x87 inicializálása	
FNINIT	(nowait)	
FENI	megszakítások engedélyezése	/8087/
FNENI	(nowait)	
FDISI	megszakítások tiltása	/8087/
FNDISI	(nowait)	
FSTCW mem2i	mem2i ← 80x87 vezérlőszó	
FNSTCW mem2i	(nowait)	
FLDCW mem2i	80x87 vezérlőszó ← mem2i	
FSTSW AX	AX ← állapotszó	/287/
FNSTSW AX	(nowait)	
FSTSW mem2i	mem2i ← állapotszó	
FNSTSW mem2i	(nowait)	
FSTENV mem14	mem14 ← a 80x87 környezete	
FNSTENV mem14	(nowait)	
FLDENV mem14	környezet ← mem14	
FSAVE mem94	mem94 ← a 80x87 állapota	
FNSAVE mem94	(nowait)	
FRSTOR mem94	állapot ← mem94	
FCLEX	a kivételes állapotjelzők törlése	
FCNLEX	(nowait)	
FINCSTP	a stack pointer növelése	
FDECSTP	a stack pointer csökkentése	
FFREE ST(i)	az ST(i) ürítése	
FNOP	üres művelet	
FWAIT	= WAIT 80x86 utasítás	
FSETPM	a protected mód beállítása	/287/

1. Program : MANGEN.PAS

A Mandelbrot halmaz rajzoló program a MANGEN.PAS futtatásával hajtható végre. A program az alábbi menüvel indul:



A program CGA, EGA és VGA típusú képernyőn működik. A aktuális típust a program automatikusan érzékeli és a típust adja a kép file kiterjesztésének (pl. VGA-nál a kiterjesztés: .VGA). A lemezen mellékeltünk két képet (mind a három típusban futtatva).

MANDEL Futtatási adatai:
A középpont
valós része : 0
képzetes része : 0

Képméret : 0.5
Maximális iterációszám : 32

MANZOOM Futtatási adatai:
A középpont
valós része : -0.1
képzetes része : 0.96

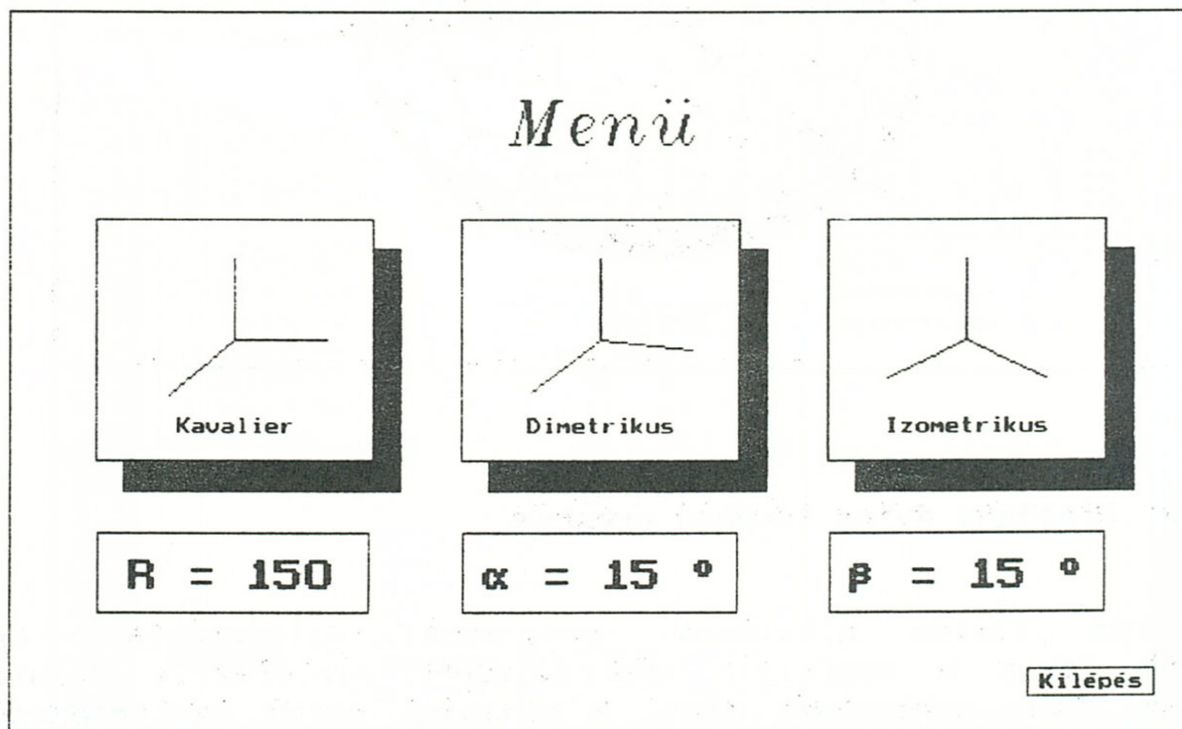
Képméret : 5
Maximális iterációszám : 128

2. Program : GOMB.PAS

A gömbrajzoló program futtatásához EGA vagy VGA típusú képernyő és egér szükséges, ezért a program indítása előtt az egeret inicializálni kell.

A kezdő képernyőn lévő gömb animáció képei a FILMKESZ.PAS programmal készült. Az animáció után egy idő múlva vagy egér klikkentésére vagy egy billentyű megnyomására megjelenik a menü.
A program animációval indul:

A program menüje:



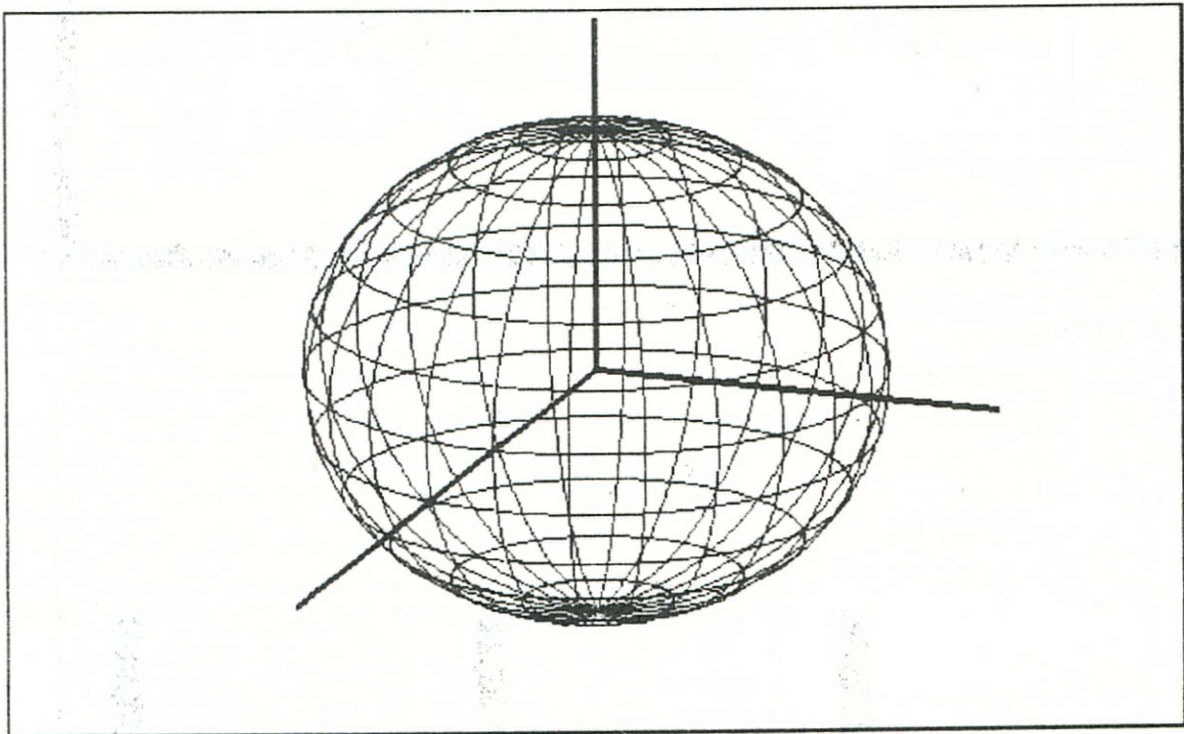
A program három fajta axonometriában :

- Kavalier,
- Dimetrikus,
- Izometrikus

rajzolja meg a gömböt.

A gömb paraméterei a sugár (R), a hosszúsági kör osztásszöge (α), és a szélességi kör osztásszöge (β) a következőképpen állítható: a nyíllal a paraméter keretébe állunk, majd klikkentünk. Erre jelenik meg a szöveges kurzor, amellyel törölhetjük a régi értéket és új értéket írhatunk be. A bevitel után nyomjuk meg az *ENTER*-t. A paraméterek a következő értékek között állíthatók (mindegyik egész szám) : R = 20 - 180, $\alpha = 1 - 90^\circ$, $\beta = 1 - 90^\circ$.

A rajzolás elindítása a kiválasztott axonometria rajzára való állás utáni klikkentéssel érhető el. A rajzolás végét hang jelzi. Klikkentésre vagy egy billentyű lenyomására újra a menübe kerülünk. A programból a Kilépés mezőre való állás és klikkentéssel lehet kilépni.

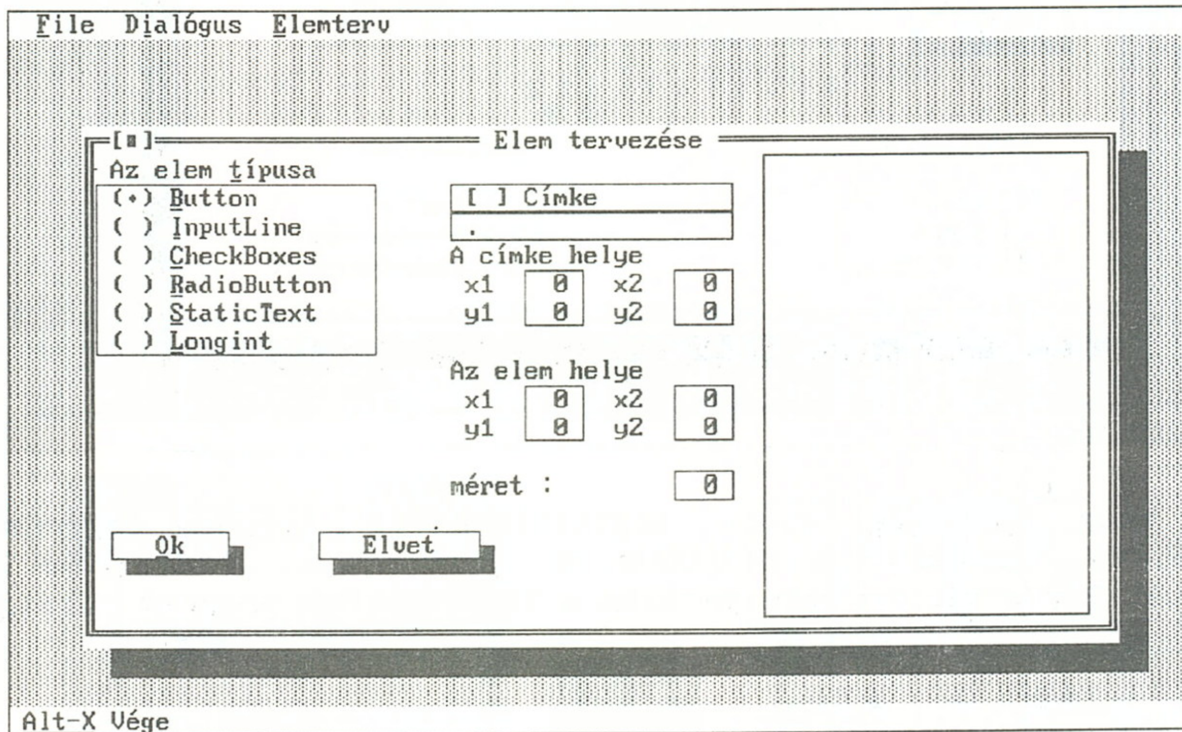


3. Program: Dialógus doboz tervező program

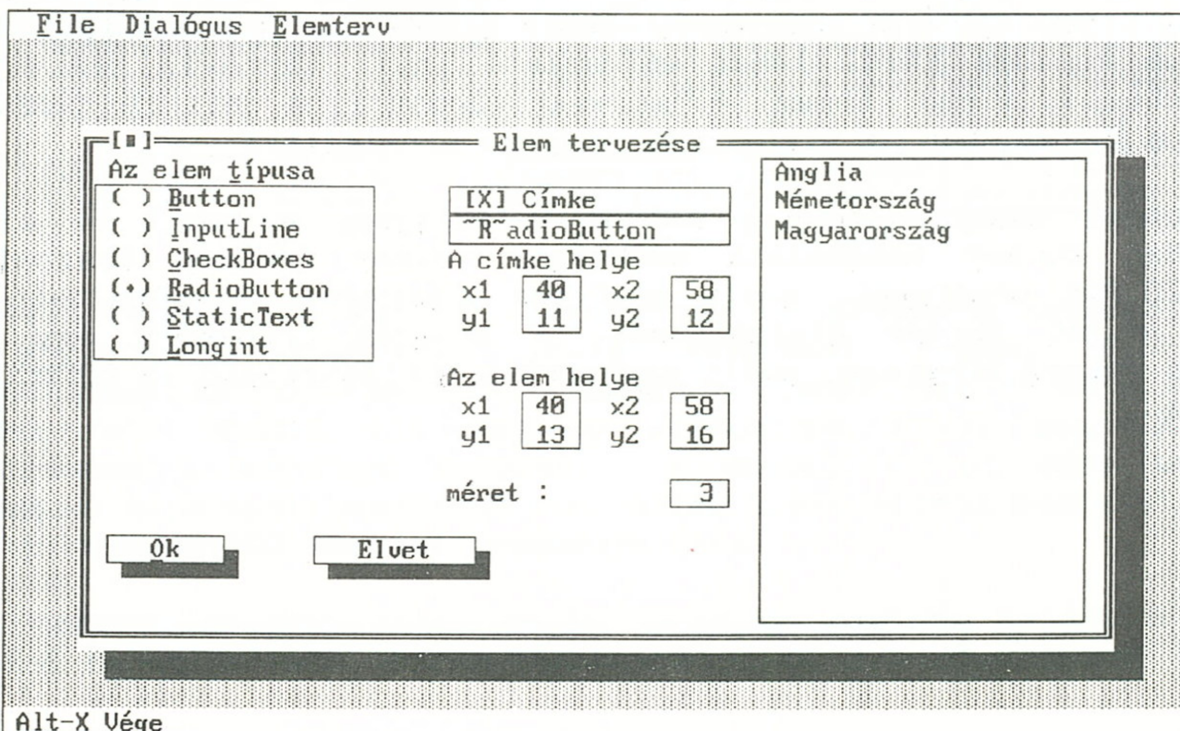
A Turbo Vision alkalmazói programok fejlesztésénél egyik legnehezebb fázis a megfelelő képernyőképek, interaktív felületek kialakítása. Ezen nehézségek közül a dialógus doboz megtervezésében segít a TVDDT.PAS program. A program lehetővé teszi, hogy egy üres dialógus dobozból kiindulva egyenként helyezzük fel a szükséges

elemeket. A tervezés végeztével a kész dialógus doboz .DBX file-ba menthető későbbi módosítások elvégzéséhez. Más programban történő felhasználáshoz egy Pascal forrásállomány .INC kiterjesztéssel is generálódik, ahol a dialógus dobozt felépítő eljárás található.

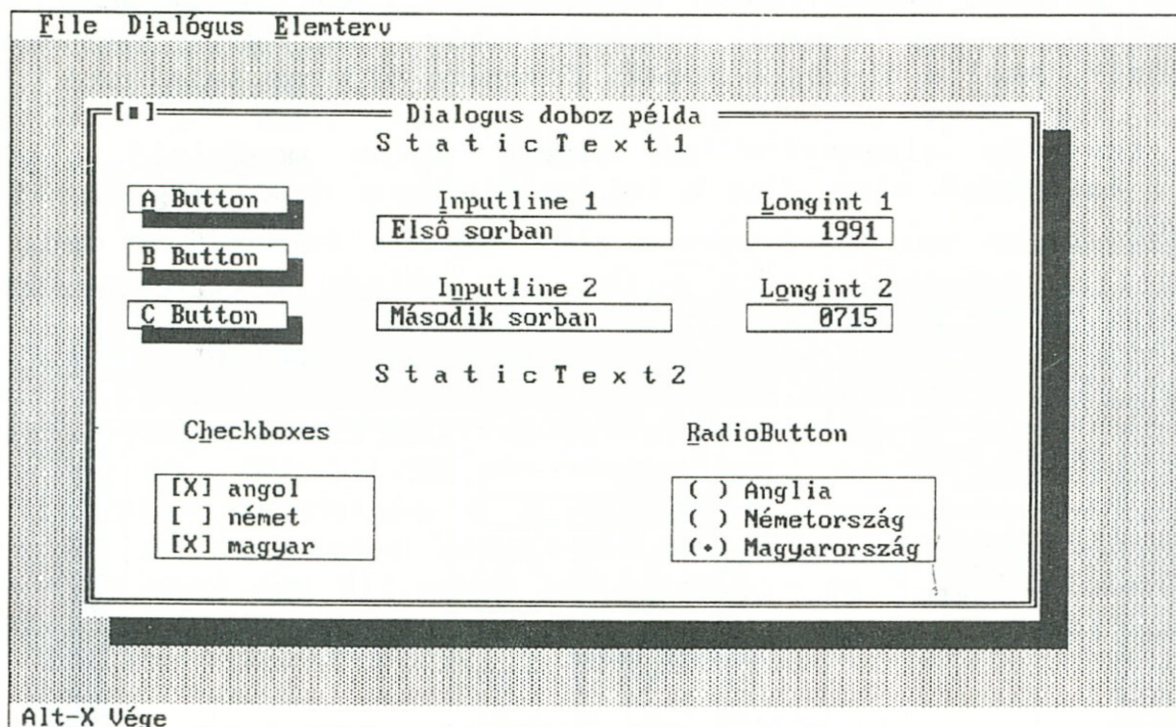
A tervezés elemenként az alábbi ablak megfelelő kitöltését jelenti, és minden elem után a teljes dialógus doboz megjeleníthető a képernyőn.



Vegyünk egy példát, amelyen a fenti input dobozt feltöltöttük egy három elemű állító gombnak megfelelően:



Végezetül nézzük meg a tervezés eredményét, ahol a jobb alsó sarokban jelent meg az előzőkben definiált állító gomb:



A kész dialógus doboz megtalálható a lemezen egyrészt a DIALOGUS.DBX, másrészt a DIALOGUS.INF file-okban. A dialógus doboz beépítését Turbo Vision alkalmazásba a TVDDTDEM.PAS program mutatja be.

4. Program: Egérkurzor tervező program

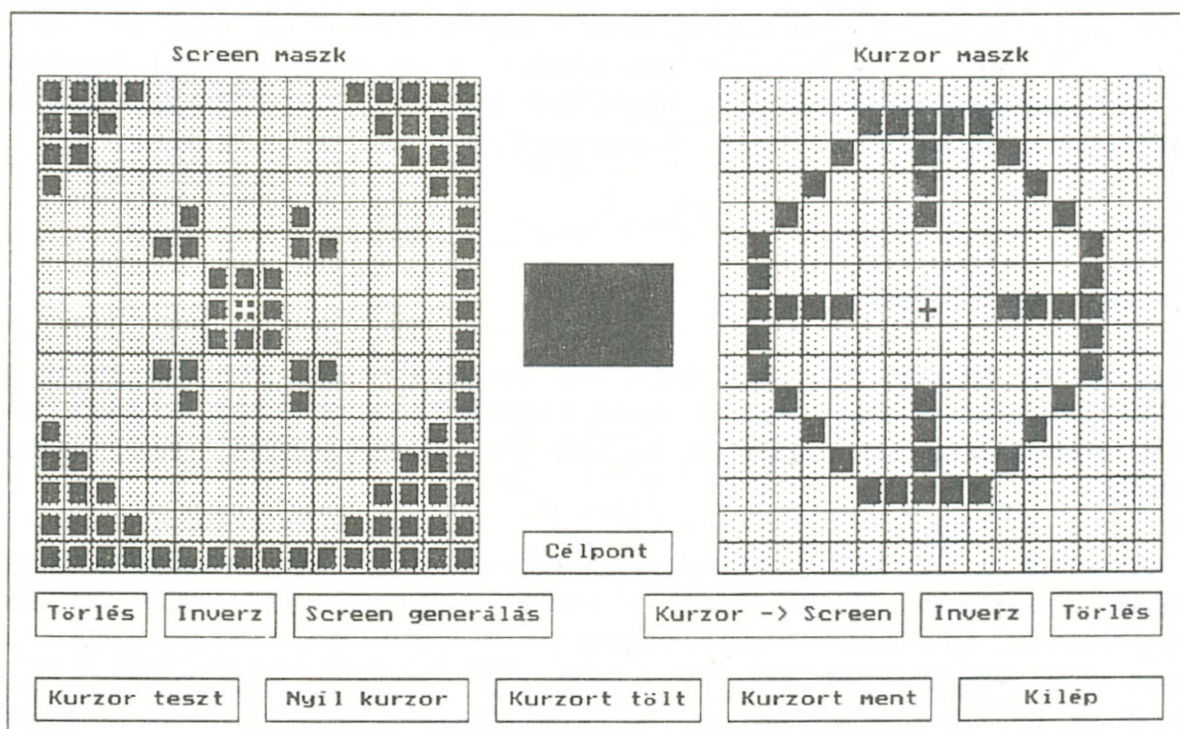
A tervező program MTERVEZO.PAS által használt OOPMOUSE.PAS unit érdekessége, hogy az egérkezelő rutinok mint egy egér objektum metódusaiként kerültek megírásra. Ebben a unit-ban jelen állapotában öt grafikus kurzort közül lehet választani (nyíl (arrow), kéz (glove), pipa (check), I-rúd (ibeam) és kereszt (cross)). A unit felhasználását grafikus üzemmódban az MTESZT.PAS program szemlélteti.

Vannak esetek azonban, amikor valamilyen, a fenti öt típustól különböző kurzor használata szükséges. Ekkor érdemes használni az MTERVEZO.PAS programot, amely grafikus képernyőn, kipróbálható módon segíti saját kurzor kialakítását. A program eredménye egy Pascal típusos rekord konstans, amely egyszerűen beilleszthető az OOPMOUSE.PAS unit-ba.

----CROSS.CUR----

```
const
   CROSS : GCursor =
   (  ScreenMask : ( $F01F, $E00F, $C007, $8003,
                     $0441, $0C61, $0381, $0381,
                     $0381, $0C61, $0441, $8003,
                     $C007, $E00F, $F01F, $FFFF );
      CursorMask : ( $0000, $07C0, $0920, $1110,
                     $2108, $4004, $4004, $783C,
                     $4004, $4004, $2108, $1110,
                     $0920, $07C0, $0000, $0000 );
      hotX : $0007; hotY : $0007 );
```

Tekintsük a tervező program képernyőjét, amint éppen a kereszt alakú kurzor rajzolása folyik.



Mielőtt munkához látnánk, nézzük meg mi okozza a nehézséget. Az egér kurzora a képernyőn 16x16 képpontot foglal el. Ennek megfeleltetünk két bit-mátrixot, amelynek elemei csak 0 és 1 értéket vehetnek fel. Minden bit egy-egy képpontnak felel meg. Az első bit-mátrixban a képernyő maszkot, a másodikban az egér maszkját tároljuk. A képernyőn megjelenő pont színe a két mátrix megfelelő eleme között elvégzett XOR művelet eredménye lesz.

A tervező program abban segít, hogy ne papíron keljen a kurzor maszkokat rajzolgatnunk, hanem képernyőn.

A program rövid ismertetése:

Kurzor maszk műveletek:
- a maszk törlése
- a maszk invertálása
- a kurzor maszk átmásolása a képernyő maszkba

Képernyő maszk műveletek:
- a maszk törlése
- a maszk invertálása
- a képernyő maszk automatikus generálása a kurzorból
- az egérkurzor tesztelése
- az alap nyíl alakú kurzor visszaállítása
- kurzordefiníció lemezre mentése
- kurzordefiníció felolvasása file-ból
- az egér középpontjának megadása

A tervezés első lépése a kuzor maszk feltöltése a kivánt ábrával. Ha kész vagyunk, a *Screen generálás* menüpontot kiválasztva, generáltatunk egy képernyő maszkot. Majd a *Kuzror teszt* menü választása után megjelenik az új egérkurzor, amit mozgassunk mind a fekete, mind a fehér területeken, hogy ellenőrizzük, megfelelő-e az alakja. Az esetek azon részében, amikor a kurzor nem tömör, általában kézzel korrigálnunk kell az automatikusan generált képernyő kurzort. Ha elégedettek vagyunk az eredménnyel mentsük file-ba a kurzort.

5. Program : HAJIT.PAS

A program demonstrálja a unit-ok használatát, bemutatja a grafika és a zene programozását, amely együttműködik a program algoritmusával, valamint bemutatja a .BGI file és a fontkészlet programba való beépítését.

A program a ferdehajítás tulajdonságainak tanulmányozását mutatja be játékos formában. A játékban a dobás szögének és kezdősebességének változtatásával egy adott távolságra lévő célt kell eltalálni. A dobás kezdőpontjába egy tankot, célpontjába egy házikót rajzolunk. A házikót kell eltalálni a paraméterek állításával. A programot igyekeztünk grafikával és hanghatásokkal, sőt zenével is gazdagitani.

A program részeinek unit-ba szervezése volt a másik célünk. A HAJIT.PAS főprogram viszonylag rövid, néhány számításon kívül csak eljáráshívásokat tartalmaz. A KOZOS unit mindazokat a deklarációkat tartalmazza, melyeket több másik unit használ. A TUZELES unit a lövedék pályáját számítja ki, és jeleníti meg, valamint változó magasságú hanggal kíséri a *loves* eljárás, a találatot képpel, hanggal és felirattal jelzi a *robban* eljárás. GRAFIKA unit a grafikus képeket állítja elő, a ZENE unit a N.J. Rubenking Pianoman nevű programjának felhasználásával készült, annak belső adatformátumát használja illetve dolgozza fel.

A program még arra is demostráció, hogy hogyan kell a .BGI file-t a programba beszerkeszteni. Ennek a módja a következő:

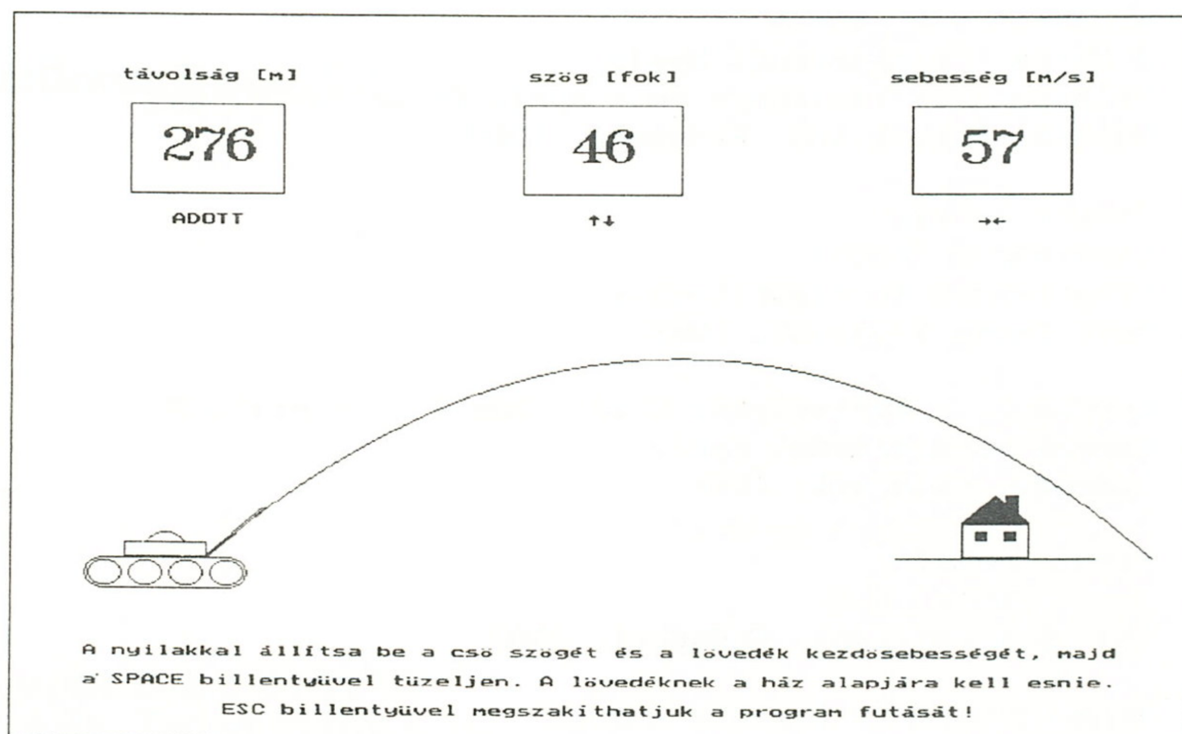
1. A BINOBJ utility segítségével object formátumra alakítjuk a BGI file-t, pl. EGA vagy VGA grafikát használunk, akkor az EGAVGA.BGI file-t. Ez a következő paranccsal történik:

   ```
   binobj egavga.bgi egavga.obj egavga
   ```

   Hatására az EGAVGA.OBJ nevű file-ban előáll az object formátumú grafikus meghajtó, melyet programunkban EGAVGA eljárásnéven deklaráltunk.

2. Programunk elején $L direktívával beszerkesztjük az EGAVGA.OBJ nevű file-t, majd EXTERNAL kulcsszóval külső eljárásnak deklaráljuk EGAVGA néven. Az erre vonatkozó egyéb vizsgálatokat az *elokészítés* eljárás tartalmazza.

A játékmező képe:



Egyéb programok

A könyv egyes fejezeteinek mélyebb megértését segítő rövid példaprogramok.

A lemezen a **Help.bat** file tartalmazza a programok tartalomjegyzékét. Minden program forrásnyelven került tárolásra, amely azt jelenti, hogy fordítani és szerkeszteni kell. A programokban található megjegyzések segítenek a működésük megértéséhez. Ajánlatos a programokról listát készíteni és úgy tanulmányozni.

1. Turbo Pascal Version 6.0
User's Guide
Borland International Inc., 1990.

2. Turbo Pascal Version 6.0
Programmer's Guide
Borland International Inc., 1990.

3. Turbo Pascal Version 6.0
Turbo Vision Guide
Borland International Inc., 1990.

4. Turbo Assembler Version 2.0
User's Guide
Borland International Inc., 1990.

5. K.Jensen - N. Wirth
A Pascal Programozási Nyelv
Felhasználói kézikönyv és a nyelv formális leírása
Műszaki Könyvkiadó, Budapest, 1988.

6. Pongor György
Szabványos Pascal
Programozás és algoritmusok
Novotrade, Budapest, 1988.

7. Gordon E. - Körtvélyesi G.né - Sós I. - Székely Z.
Pascal programozási nyelv
Számalk, Budapest, 1982.

8. Pirkó József
Turbo Pascal 5.5
LSI Oktatóközpont, Budapest, 1990.

9. Benkő Tiborné - Hegedüs András
IBM PC programozása Turbo Pascal nyelven
BME Mérnöktovábbképző Intézet, Budapest, 1989.

10. Ben Ezzell
Object-Oriented Programming in Turbo Pascal 5.5
Addison-Wesley Publishing Company, Inc. 1989.

A könyv készítése során a Szerzők és a Kiadó a legnagyobb gondossággal járt le. Ennek ellenére hibák előfordulása nem kizárható. Az ismeretanyag felhasználásának következményeiért sem a Kiadó sem a Szerző felelősséget nem vállal.

Minden jog fenntartva. Jelen könyvet vagy annak részleteit a Kiadó engedélye nélkül bármilyen formátumban vagy eszközzel reprodukálni, tárolni és közölni tilos.

© szerzők,
© ComputerBooks Kiadó, 1995

© Kiadó: ComputerBooks Kiadói Kft
1126 Bp., Tartsay Vilmos u. 12.
Tel.: 175-15-64; tel./fax: 175-35-91
Felelős kiadó: ComputerBooks Kft ügyvezetője
ISBN: 963 618 079 2

Borítóterv:Székely Edith

ÁFÉSZ Nyomda Vác

TUNGSRAM

MAX floppy disk

MINŐSÉG A PROFIKNAK: 21 ORSZÁGBAN A VILÁGON

A TUNGSRAM-MAX mágneslemez japán, amerikai alapanyagokból, amerikai technológiával, high-tech berendezéseken készül. Minden egyes mágneslemez hibamentességét a teljes felület számítógépes mérőrendszerrel történő tesztelése garantálja.

TUNGSRAM-MAX mágneslemez				
		Színes, műanyag dobozos		Mágneslemezek
Standard	Formattált	Standard	Formattált	No name, bulk csomagolásban
5.25 TM 2S2D	5.25 TMF 2S2D	5.25 TMP 2S2D	5.25 TMPF 2S2D	Tárolódobozok
5.25 TM 2SHD	5.25 TMF 2SHD	5.25 TMP 2SHD	5.25 TMPF 2SHD	Tisztítókészletek
3.5 TM 2S2D	3.5 TMF 2S2D	3.5 TMP 2S2D	3.5 TMPF 2S2D	3.5" és 5.25" méret
3.5 TM 2SHD	3.5 TMF 2SHD	3.5 TMP 2SHD	3.5 TMPF 2SHD	Gyűjtődobozok

SZOLGÁLTATÁSAINK:

— *LEMEZEK FORMÁLÁSA*
— ***PROFESSZIONÁLIS SZOFTVER MÁSOLÁS***
— *IGÉNY SZERINTI CSOMAGOLÁS*
— ***FOLYAMATOS RAKTÁRI KISZOLGÁLÁS***
— *IGÉNY SZERINTI SZÁLLÍTÁS A VEVŐHÖZ*
— ***VISZONTELADÓKNAK ÁRKEDVEZMÉNY***
— *IRÁSVÉDŐ KIVÁGÁS NÉLKÜLI LEMEZEK*

Input: MAX

Output: maximum

TUNGSRAM MAGNETIC MEDIA RT
H-1340 Budapest, IV. Váci u. 77.
Tel.: 160-2233 Fax: 160-0925

Megrendelőlap

Megrendelem az alábbi kiadványokat postai utánvéttel. Tudomásul veszem, hogy a postaköltség felszámításra kerül, és a szállítási idő 2–3 hét.

Utánnyomásoknál árváltozás lehetséges.

... pl *Füzi János:* **3 dimenziós grafika és animáció IBM PC–n** – lemezmelléklettel 1.283.–

... pl *Nagy G.:* **Kézikönyv az adattömörítéshez – ARJ. PKZIP & Co.** – lemezmelléklettel 1.298.–

... pl *Pintér M.:* **AutoCAD parancsok és változók** – Release 13 – angol & magyar – DOS & WINDOWS & UNIX 1.176.–

... pl *Pintér M.:* **AutoCAD R13 szerkesztési újdonságok** 599.–

... pl *Pintér M.:* **AutoCAD tankönyv** – DOS & WINDOWS; AutoCAD LT; AutoCAD R12 angol & magyar 899.–

... pl *Pintér M.:* **Rajzkészítés AutoCAD Release 12–vel** 590.–

... pl *Benkő–Poppe–Benkő*: **Bevezetés a BORLAND C++ programozásába** 945.–

... pl *Benkő L.–Benkő T.né–Tóth*: **Programozzunk C nyelven kezdőknek * középhaladóknak** – lemezmelléklettel 1.199.–

... pl *Dr.Dedinszky F.:* **CA–VISUAL OBJECTS** 1.559.–

... pl *Dr.Dedinszky F.:* **CLIPPER 5 – 5.0, 5.01 és segédprogramjai** 899.–

... pl *Nagy Z.–Spányik B.–Weisz T.:* **CorelDRAW! 5** 795.–

... pl *Tamás–Kiss–Tóth:* **MS–DOS 6 – 6.2; 6.22 kiegészítéssel** 985.–

... pl *Dr.Janurik T.*: **MS–DOS hibaüzenetek a 3–, 4–, 5–, 6. verziókhoz** 199.–

... pl *Kóczy A. J.*: **MS–DOS 5.0, 6 kis☹kos** 295.–

... pl *Kovalcsik G.:* **EXCEL for Windows 5.0 kezdőknek * haladóknak** – magyar és angol változathoz 1.147.–

... pl *Dr. Kovácsné C. J. – Ozsváth M.:* **EXCEL 5 függvényei** – magyar változathoz 990.–

... pl *Krizsák L.:* **EXCEL 5 for Windows Kis☹kos** – magyar és angol változathoz 398.–

... pl *Balogh J.–Dr.Dedinszky F.*: **FoxPRO 2.0** 895.–

... pl *Gazsó Z.:* **FoxPRO 2.5, 2.6** – Windows/DOS – lemezmelléklettel 1.475.–

... pl *Abonyi Zs.*: **PC hardver kézikönyv** (bővített, átdolgozás kiadás) 875.–

... pl *László J.:* **Hangkártya programozása Pascal és Assembly nyelven** – lemezmelléklettel 1.568.–

... pl *Stolnicki Gyula:* **Hálózatokról kezdő felhasználóknak** 1.369.–

A 175–35–91 telefonszámon könyvszolgálatunk tájékoztatja Önt a lakó– vagy munkahelyéhez legközelebb eső szaküzletről, ahol kiadványainkat megvásárolhatja.

Ha mégis a postai utat választja, kérjük levélcímre visszaküldeni
ComputerBooks Kft – 1253 Bp., Pf.: 71 .

... pl *Lengyel Veronika:* **Az INTERNET világa** 1.456.–

... pl *Székely V.:* **Képkorrekció, hanganalízis, térszámítás PC–n** – lemezmelléklettel 1.258.–

... pl *Fehérvári A:* **LOTUS for WINDOWS és a Freelance Graphics** 447.–

... pl *Benkő–L. –Benkő T.né:* **MS WORKS 3.0 a mindennapi életben** – magyar verzióhoz 793.–

... pl *Rudnai P.né:* **Novell NetWare 3.11, 3.12 felhasználóknak és rendszergazdáknak** 945.–

... pl *Tóth Dezső:* **OS/2 Warp felhasználói ismeretek** 1.680.–

... pl *Benkő–Tóth–Varga*: **Programozzunk TURBO PASCAL nyelven** – lemezmelléklettel (javított, átdolgozott kiadás) 796.–

... pl *Benkő T.né–Kiss Z. –Tóth B:.* **Objektum–orientált programozás Turbo Pascal 6.0–ban és a Turbo VISION** – lemezmelléklettel 979.–

... pl *Benkő T.né–Kiss Z. –Tamás P. –Tóth B.*: **Programozás Borland Pascal 7.0 rendszerben /DPMI, WINDOWS** – példaprogramok lemezmellékleten 1.586.–

... pl *Dr. Rubicsek Gy.:* **PC 1x1** (amit a számítógépről és eszközeiről tudni illik) 298.–

... pl *Kovácsné C. J. –Pergelné B. I. –Benkő L.*: **Mindenkinek! a PC–ről** 699.–

....pl *Gerő Judit:* **PowerPoint 4** 1.426.–

... pl *dr. Kovácsné C. J. –Ozsváth M.:* **QuarkXPress for Windows** 979.–

... pl *Pergel J.né:* **QuattroPRO 5** 770.–

... pl *Borgulya I.*: **Szakértői rendszerek, technikák és alkalmazások** 1.375.–

... pl *Stolnicki Gy.:* **SQL Kézikönyv** – bővített, átdolgozott kiadás – lemezmelléklettel 1.499.–

... pl *László József:* **VGA kártya programozása** Pascal és Assembly nyelven – lemezmelléklettel 1.375.–

... pl *Nagy Gábor:* **Vírusvédelem a PC–n** – lemezmelléklettel 1.157.–

... pl *dr.Tamás–Horváth–Kiss–Tóth*: **WINDOWS 3.1 felhasználóknak** 698.–

... pl *dr.Kovácsné Cohner Judit:* **Magyar WINDOWS 3.1** 990.–

... pl *Benkő T.né–Kuzmina J. –Kiss Z. –dr.Tamás P. –Tóth B.:* **Könnyű a WINDOWS–t programozni!?** – lemezmelléklettel 1.683.–

... pl *Tóth B. –Tamás P és trsai:* **WINDOWS 95 & Microsoft Plus felhasználóknak** 1.995.–

... pl *Dr.Kovácsné C. J. –Ozsváth M.:* **Windows for Workgroups 3.11** – hálózattal vagy anélkül 1.115.–

... pl *Rudnai P.né – Rudnai T.:* **Windows for Workgroups 3.11 kis☹kos** – angol és magyar változathoz 399.–

... pl *Gerő Judit–Reich Gábor*: **WORD for WINDOWS 6.0** – magyar & angol nyelvű verzióhoz 980.–

... pl *Gerő Judit–Krizsák László:* **WORD for WINDOWS 6.0 kis☹kos** 498.–

Megrendelő neve: ______________________

szállítási cím: ______________________

város: ______________________

utca: ______________________

irányítószám: ______________________

Ára: 979.– Ft ÁFA-val, lemezmelléklettel

Keresse könyveinket!

COMPUTERBOOKS

1126 BUDAPEST, TARTSAY V. u. 12.

Tel.: 17-51-564, 17-53-591