Bakos Tamás

Az új PASCAL generáció

BAKOS TAMÁS

AZ ÚJ PASCAL GENERÁCIÓ

Eljárások, modulok és objektumok

Számítástechnika-alkalmazási Vállalat
Budapest, 1991

Lektorálta
Esztergár Zsolt

© Bakos Tamás, 1991

ISBN 963 553 280 6

Számalk Könyvkiadó
A kiadásért felel a Számítástechnika-alkalmazási Vállalat vezérigazgatója
Felelős kiadó: Kis Ádám
Felelős szerkesztő: Gyömöre Mihályné
Műszaki szerkesztő: Lajtai Gábor
Fedélterv: Hörömpő Gabriella
Készült 8 (A/5) ív terjedelemben
Számalk–Kelenföld Kft. Nyomda 91/
Felelős vezető: Nagy Sándor
Megjelent az Állami Biztosító támogatásával

Tartalomjegyzék

BEVEZETÉS 7

1. A PASCAL RENDSZEREK FEJLŐDÉSE 9

1.1. Az eredeti nyelv 9
1.2. Út a standard nyelvtől a Turbo Pascalig 10
1.3. Turbo Pascal 14

2. ELJÁRÁSOK 17

2.1. Az eljárás mint programozási eszköz 19
2.2. Az eljárástípus 23
2.3. Standard eljárások 29
2.3.1. Bevitel és kivitel (B/K) 30
2.3.2. Programok vezérlése 34
2.3.3. A dinamikus memória kezelése 34
2.3.4. Típuskonverzió 35
2.3.5. Aritmetika 36
2.3.6. A megszámlálható típusok kezelése 37
2.3.7. Karakterláncok kezelése 37
2.3.8. Mutató- és címkezelés 38
2.3.9. Vegyes célú eljárások és függvények 39

3. MODULOK 41

3.1. Programok és modulok 41
3.2. Standard modulok 46
3.2.1. A SYSTEM modul 46
3.2.2. A PRINTER modul 46
3.2.3. A DOS modul 47
3.2.4. A CRT modul 53
3.2.5. A GRAPH modul 57
3.2.6. A TURBO3 modul 81
3.2.7. A GRAPH3 modul 82

4. OBJEKTUMOK 85
4.1. Egyesítés 85
4.2. Öröklődés 90
4.3. Polimorfizmus 97
4.4. Objektumok a dinamikus tárban 102
4.5. Egerek és objektumok – példaprogram 109

IRODALMI AJÁNLÁSOK 122

Bevezetés

A lassan már több mint két évtizedes múltja alapján egyértelműen megállapítható: a Pascal sikeres és népszerű nyelv. Használhatóságának egy újabb bizonyítéka – a nyelv elméleti és gyakorlati szempontból egyaránt jelentős bővítése – szolgáltatta az ürügyet e könyv kiadásához.

Az említett bővítés a Pascalt **objektum orientált programozásra** (a közös angol–magyar rövidítés: OOP) teszi alkalmassá. Jelentőségét és céljait tekintve a strukturált programozáshoz hasonlítható objektum orientált programozás önmagában is megkülönböztetett figyelmet érdemel, az a tény pedig, hogy ezúttal a korszerű programozás jóformán teljes fegyvertárát felvonultató Pascal köntösében jelenik meg, további előnyöket ígér.

A könyv természetesen nem szorítkozhat csupán a Pascal objektum orientált kiterjesztésére, hanem bemutatja azt a fejlődést, amely a nyelv születése után nem sokkal megindult, majd az utóbbi öt évben felgyorsulva, a programozás elméletének és gyakorlatának élvonalában tartotta azt.

Ugyanakkor nem volt szándékunkban Pascal referenciakönyvet készíteni sem, hiszen ezek a gépi megvalósításokat fejlesztő és forgalmazó cégek kiadásában (és némi késéssel magyarul is) folyamatosan megjelennek.

Ennek megfelelően feltételezzük az Olvasóról, hogy a Pascal alapjait már ismeri, vagy valamelyik ilyen célú kiadványból el tudja sajátítani, s ezt a könyvet a nyelv eddigi karrierje, valamint jelenlegi állapota megismerésének szándékával veszi kézbe.

E célt tükrözi a könyv szerkezete is:

Az **első fejezet** a Pascal rendszerek fejlődését tekinti át a Wirth-féle eredeti nyelvtől a Turbo Pascalig, illetve annak az OOP kiterjesztést tartalmazó 5.5-ös változatáig, ami a nyelv egy új generációjának tekinthető, és a többi három fejezet témáját adja.

A **második fejezet** a többi típussal lényegében azonos rangra emelt **eljárás** (procedure) típust tárgyalja. Az eljárás típusú változók és a rajtuk értelmezett műveletek bevezetése fontos lépés volt a Pascal fejlődésében, s ez szinte egyenesen a moduláris programszerkesztés újszerű, hatékony formájához vezetett.

A **harmadik fejezet** a Microsoft Pascalban megjelenő, majd később a Turbo Pascal 4.0-ás változatában általánosított, **unit**-nak nevezett programmodulokkal foglalkozik. Ide nemcsak a modulok létrehozási és felhasználási lehetőségei tartoznak, hanem az ún. standard (előredefiniált) modulok ismertetése is.

A **negyedik fejezet** tárgya az objektum orientált programozásnak a Turbo Pascal 5.5-ös változatában megvalósított formája. Az objektumokat, mint a rekordból általánosítással származtatott típust tárgyaljuk, melyek két alapvető új tulajdonsággal rendelkeznek:

* Az adatszerkezethez tartozó algoritmusokat (módszereket) az objektum adatmezőivel azonos módon és helyen lehet definiálni;
* Az egyszerűbb objektumokból további mezők (adatszerkezetek vagy módszerek) hozzáadásával utódobjektumok hozhatók létre, amelyek öröklik az elődtulajdonságait.

Végül a könyvben használt terminológiáról annyit kell elmondani, hogy néhány, magyarul még nem elterjedt fogalomhoz megpróbáltunk a jelentést minél jobban kifejező magyar vagy magyarban használatos megfelelőt találni. Így lett a unit-ból **modul**, az object-ből **objektum** és a method-ból **módszer**. E kifejezéseket ideiglenes használatra ajánljuk addig, amíg az élő nyelv létre nem hozza végleges változatukat.

1. A Pascal rendszerek fejlődése

Akárcsak az élő nyelvek, a széles körben használt programozási nyelvek is folyamatosan változnak, fejlődnek. Ha a Pascal fejlődését akarjuk áttekinteni, a nyelvet **környezetével együtt** kell vizsgálni. Ebben az esetben a környezet az alkalmazott számítógép hardverjellemzőit, architektúráját, a fordítóprogramhoz tartozó járulékos szolgáltatásokat, segédprogramokat (egyszóval a Pascal rendszert) jelenti. Ha egy nyelv fejlődését „történelmi távlat"-nak tekinthető, két évtizedes időszakra nézve vizsgáljuk, a környezetből nem szabad kihagyni a fontosabb programozáselméleti irányzatokat sem.

1.1. Az eredeti nyelv

A Pascal első változata 1968–69-ben egyetemi környezetben jött létre az N. Wirth professzor vezetésével dolgozó csoport munkájaként. Elődjének az ALGOL–60, illetve ennek egy alig ismert változata, az ALGOL–W tekinthető. A Pascal az adatdefiníció terén általánosítást, az algoritmusleírás (utasítások) terén pedig egyszerűsítést jelentett.

Talán legnagyobb érdeme, hogy a programozó kezébe új adattípusok definiálására alkalmas eszközt adott. Az új típusok létrehozásához kiindulásként a nyelv csupán néhány alapvetőnek nevezhető egyszerű típust (**integer**, **char**, **Boolean**, **real**), néhány összetett típust (**array**, **set**, **record**, **file**), valamint az ún. **megszámlálható** típusok családjának fogalmát adta. Új megszámlálható típust a hozzá tartozó értékek felsorolásával vagy már létező megszámlálható típusú értékek egy intervallumának megadásával lehet létrehozni.

A dinamikus (a program futása során létrejövő és megszűnő) változók kezelését a **pointer** típus oldja meg igen általános formában.

A típusdefiníció két lényeges jellemzője az egymásra épülés és az ortogonalitás. Az összetett adatszerkezetek fokozatosan építhetők fel az egyszerűbbek felhasználásával, a már létező típusok pedig jóformán korlátozás nélkül használhatók fel új típus kialakításában. Így lehet pl. rekordokra hivatkozó mutatókból álló tömbök halmazát vagy állományát létrehozni.

Az ésszerű korlátok között szabaddá tett típusgenerálás lehetővé teszi, hogy a programozó a feladat lényegére koncentrálhasson különböző, többékevésbé erőltetett kódrendszerek kiagyalása helyett.

A típusok szabad virágzásának és a programok belső rendjének egyidejű biztosítására a nyelvhez **ún. szigorú típusellenőrzés** tartozik. Ez azt jelenti, hogy műveleteket általában csak azonos típusú értékekkel lehet végezni. Kivétel csak néhány teljesen kézenfekvő esetben (pl. valós és egész értékek összeadása) tehető.

A programok másik összetevőjének, az algoritmusoknak leírására a Pascal mindössze nyolc utasítást tartalmaz (**értékadás**, **goto**, **if**, **for**, **while**, **repeat**, **case**, **eljáráshívás**). Mivel ezt a kis csoportot Wirth alkalmassá akarta tenni az éppen akkor kialakulóban lévő korszerű elmélet, a **strukturált programozás** teljes körű támogatására, a rekurzív gondolkodás egy szép példájaként bevezette (vagy az ALGOL–60-ból átvette) az **összetett utasítás** fogalmát, melynek segítségével bármely utasítássorozatból a **begin** és **end** ún. utasítászárójelek használatával egyetlen összetett utasítás képezhető.

Ez az utasításkészlet – mint tudjuk – azóta sem szorult bővítésre vagy módosításra.

Hasonlóan időállónak bizonyult a Pascal eljárás- és függvényfogalma is. Itt ugyan elsősorban a nyelv elődjének – az ALGOL–60-nak – jól bevált elvei érvényesülnek, a Pascalban ezek letisztulnak, egyszerűsödnek és a laikus felhasználó számára is egyértelműen kettős célt szolgálnak:

* Az algoritmusok többször felhasználható, könyvtárakba szervezhető önálló egységekbe foglalása;
* A felhasználó által létrehozott fogalmak (típusok, változók, címkék, eljárások stb.) életterének, hatáskörének egyértelmű kijelölése.

Egyetlen be- és kilépési pontjukkal, valamint szabályozott paraméterkezelő módszereikkel az eljárások nemcsak sugallják, hanem jóformán előírják a strukturált programozás elveinek betartását. A kétféle (érték és cím szerinti) paraméterátadás ugyanakkor praktikus, hatékonysági célokat is szolgál: terjedelmes adatszerkezetek esetén a cím szerinti átadás megtakarítja az aktuális érték átmásolásához szükséges időt és memóriát.

A Wirth-féle Pascalt viszonylag gyorsan szabványosították, a Nemzetközi Szabványügyi Hivatal (ISO) erre vonatkozó kiadványa 1983-ban jelent meg, ezért a nyelv ezen változatára **standard** Pascal néven is szoktak hivatkozni.

A szabványosítás azonban korántsem jelentette a nyelv befagyasztását. Mint minden sokat használt eszköz, a standard Pascal is folyamatosan változott. A következőkben az elmúlt, több mint másfél évtized folyamán történt változások lényegét foglaljuk össze.

1.2. Út a standard nyelvtől a Turbo Pascalig

Amikor az egyetemi (elsősorban oktatási célú) környezetből kilépve, a Pascal az alkalmazások „mély vizébe" jutott, néhány gyenge pontja azonnal felszínre került.

Ilyen volt például a szöveges információ kezelésének nehézkessége. Egy szöveget karakterek tömbjeként kezelni azért nem célszerű, mert ezzel elrejtjük a felhasználó elől a szövegek néhány alapvető tulajdonságát. Egy igazi szöveges változó hossza csak dinamikus lehet, szövegek szétvágását, összerakását, rendezését és egyéb műveleteket pedig minél egyszerűbben megoldhatóvá kell tenni. Ezért a standard Pascal igen mostohán kezelt string típusát igen hamar felváltotta a B/K műveletekben használható, általános karakterlánc típus. A korszerű Pascal rendszerekben, így a Turbo Pascalban is a típusdefinícióban a karakterlánc maximális hosszát lehet megadni, melynek alapfeltételezés szerinti értéke 255. Külön problémát jelent a különböző méretű karakterláncok értékadásra, valamint eljárások paramétereként való felhasználásra vonatkozó kompatibilitása. A jó megoldáshoz itt az a felismerés vezetett el, hogy ezt a kérdést a felhasználó hatáskörébe kell utalni, mivel egy fejlesztés alatt álló programban hasznos lehet, ha a futási időben jelzést kapunk például arról, hogy egy karakterláncba annak maximális hosszát meghaladó értéket akarunk írni, ugyanakkor a már bejáratott programot esetleg feleslegesen terheljük soha be nem következő események figyelésével. Ennek megfelelően több Pascal rendszer – így a Turbo Pascal is – fordítóprogram-direktívák segítségével ki- és bekapcsolhatóvá teszi a karakterláncok aktuális hosszának ellenőrzését értékadás és az eljárások formális-aktuális paramétereinek megfeleltetése során. Ezzel engedtek valamit a nyelv szigorú típusegyezésre vonatkozó előírásaiból a rugalmasabb nyelvhasználat és a hatékonyabb tárgyprogram érdekében.

A típusoknál maradva több, általában nem igazán szerencsés kísérlet történt a standard Pascal statikus tömbfogalmának dinamikussá tételére. Egyes fejlesztők talán megirigyelték a Basic tömbkezelését, ahol a tömb mérete a program futása során is megadható (nem kell azt a program megírásakor rögzíteni). Így jött létre például a Microsoft Pascal **super array** típusa, melynél a tömb felső indexhatárát *-gal jelölve, később változó méretű tömböket deklarálhattunk. Ezek a megoldások – úgy tűnik – nem állták ki az idő próbáját és ma sem népszerűek. Ami az alapjában véve értelmezéses technikával feldolgozott Basic esetében jóformán magától adódik, az fordítóprogram használata esetén indokolatlanul nagy terhet ró a tárgyprogramra, illetve az annak működését támogató rendszerre. Ugyanakkor a Pascalban rendelkezésünkre áll a mutatókkal elérhető **dinamikus memória** (heap), amit egy kis fáradsággal úgy is szervezhetünk, hogy elérése a tömböknél megszokott módon történjen.

A legszembetűnőbb fejlődés talán az előre definiált (standard) eljárások és függvények körében észlelhető. Amíg a standard nyelv egy mai szemmel igen puritánnak nevezhető eljáráscsaládot ad a felhasználónak, a Turbo Pascal programozók a bőség zavarával küzdve válogathatnak az eljárások és függvények között. Közvetlenül használhatják az MS–DOS összes szolgáltatását, futási időben megkérdezhetik, hogy milyen grafikus kártya van a gépben, vagy állományokat kereshetnek a háttértárban különböző ismérvek alapján.

A **B/K eljárások** használata nagymértékben egyszerűsödött. A standard Pascal az állományok írását és olvasását mutatók segítségével valósította meg, ami nehézkes és természetellenes volt. A korszerű megoldásban az állományokat egy tetszőleges, közös alaptípushoz tartozó elemek sorozatának tekintik, melyhez egy egész értékeket felvevő implicit index (mutató) tartozik. Az implicit index kezelését egyes eljárások (pl. **read** és **write**) automatikusan elvégzik, de értéke pozicionáló eljárással (**seek**) is beállítható. A beolvasott elemek a **read** utasításban megadott változók értékeiként közvetlenül elérhetők lesznek. Külön említést érdemel a Turbo Pascal ún. típus nélküli állománya, amelynél nem kell megadni az alaptípust, az állomány elemei (rekordok) tetszőleges méretű változóba beolvashatók vagy abból kivihetők. A rekord mérete az állomány megnyitásakor (**reset** vagy **rewrite**) adható meg, alapfeltételezés szerinti értéke 128. A standard eljárások készletének ilyen mértékű bővülését csak részben magyarázza a technika (pl. a grafikus hardver) fejlődése. A Turbo Pascal esetében valószínűleg nagy szerepet játszott a felhasználó **komplex kiszolgálásának** igénye, hiszen sok olyan eljárást találunk itt, amely a húsz évvel ezelőtti technikai szinten is megvalósítható lett volna (pl. számok és karakterláncok kölcsönös konverziója, a szöveges képernyő kezelése stb.).

Amikor a Pascalt összetett feladatok megoldására, nagyobb programrendszerek kidolgozására is kezdték használni, felmerült a **moduláris** programfejlesztés igénye. Több, kevésbé sikeres megoldás mellett a Microsoft Pascalban jelent meg először az ott **unit**-nak nevezett programmodul, ami külön fordítható egység saját konstansokkal, típusokkal, változókkal, eljárásokkal és függvényekkel. A modulok két részből állnak. Az ún. **interface** részben a más modulok vagy programok által felhasználható (nyilvános) azonosítók szerepelnek, az eljárások törzs nélkül, csupán fejükkel képviselve. A modul másik, **implementation** elnevezésű részében kell definiálni a nyilvános eljárások törzsét, minden olyan azonosítót, amely a modul saját céljaira szolgál (privát), más modulok vagy programok számára nem elérhető, ugyancsak itt lehet megadni egy kezdő helyzetet beállító programtörzset, amely a modult használó program indításakor automatikusan végrehajtódik. Egy modulinterfészhez több implementáció is tartozhat (pl. assembly-ben vagy más nyelven írott variánsok). Ezt a modulfogalmat átvette és továbbfejlesztette a Turbo Pascal a 4.0-ás változattól kezdve. A fejlesztés elsősorban formai egyszerűsítést és kisebb mértékű elvi átértékelést jelentett. Ezzel a modulfogalommal részletesebben a 3. fejezetben foglalkozunk.

Az idők folyamán sokat változott a Pascal feldolgozásának technikája is. Wirth eredeti elképzelése a fordítóprogram Pascalban való megírását és egyik gépről a másikra való minél egyszerűbb átvitelét helyezte előtérbe. Ennek érdekében bevezetett egy gépektől független, közbenső nyelvet, az ún. **P-kódot**. A Pascal programot egy eredetileg Pascal nyelven írt fordítóprogram első lépésben P-kódra fordítja, és ez a tevékenység nem függ a használt géptől. A kapott P-kódú program ember számára már nem értelmes, de

géppel viszonylag egyszerűen feldolgozható akár értelmezéses, akár fordítási technikával. Ahhoz tehát, hogy a Pascal egy újabb gépen használható legyen, „csak" egy fordítóprogramot kell a P-kódról az adott gép nyelvére előállítani. A módszer előnye (a Pascal rendszer portabilitása) már régen nem súlyponti kérdés, hiszen a piac túlnyomó többségét csupán néhány processzor uralja. Ugyanakkor a kereskedelmi forgalomban lévő fordítóprogramokat erre szakosodott szoftvercégek (Microsoft, Borland stb.) állítják elő, és elsődleges szempont a felhasználók kényelmes és hatékony programozási eszközzel való ellátása. Ennek megfelelően a mai Pascal fordítóprogramok már egymenetesek és olyan tárgykódot (object forma) állítanak elő, amit a kapcsoló-szerkesztő (linkage editor) hoz végrehajtható formára.

A feldolgozási technikához tartozik a fordítóprogram vezérlése is. A hívási sorban kiadható néhány paraméterrel szemben igen praktikusnak és hatékonynak bizonyult a fordítóprogram-direktívák rendszere, amit például a Turbo Pascal alkalmaz. A több mint húsz direktíva segítségével igen simulékonnyá tehető a fordítás. Alkalmazkodni lehet speciális memóriaigényekhez, elő lehet írni a logikai kifejezések kiértékelésének módszerét, ki és be lehet kapcsolni a B/K műveletek helyességének ellenőrzését vagy az indexhatárok megsértésének figyelését, overlay programokat lehet kezelni, szövegrészeket lehet fordítás közben a programba illeszteni, feltételes fordítást lehet végezni stb. A direktívák nagymértékben hozzájárultak ahhoz, hogy a felhasználó egy fordítóprogram helyett jól használható fejlesztőrendszert kapjon.

A nyelv és környezete természetesen igen sok kitérővel, kerülővel és zsákutcával érte el a jelenlegi szintet. Ha mérföldköveket keresünk, szinte magától adódik a személyi számítógépek megjelenése. Az ezt megelőző időszakban a folyamatok lassúbbak voltak, a változások kevésbé mélyrehatónak tűntek. Ez részben a gépteremhez, munkahelyhez kötött számítógépeknek volt köszönhető. A mikroprocesszoron alapuló gépek, különösen az IBM PC elterjedése új lendületet adott a szoftver szakmának. Most került annyi gép forgalomba, hogy világméretekben is kialakulhatott a már régebben emlegetett **szoftveripar**, ami szoftvertermékek ipari mennyiségekben való fejlesztését és forgalmazását jelenti. A fejlődés során több értékes, ma is figyelemre méltó Pascal-változat jött létre, melyek viszonylag rövid pályafutás után ma már csak történeti érdekességnek számítanak. Ezek közül is érdemes egyet, a San Diegoi Egyetemen készült UCSD Pascal rendszert kiemelni. Ez a változat tulajdonképpen a mikrogépes Pascal-megvalósítások első szabványának volt tekinthető. A fejlesztő (Softech Microsystems) P-kódot használt, amit egy értelmezőprogram hajtott végre. Az UCSD Pascal nemcsak nyelvi, hanem operációs rendszer is volt. 1982–83-ban az MS–DOS még nem nyert egyértelműen csatát, ezért reális fejlesztési megoldásnak látszott a nyelv alá – természetesen szintén Pascalban írt – operációs rendszert is kidolgozni, amely nagyjából az MS–DOS szintjén állt. A rendszer gyenge pontja a P-kód értelmező program volt, ami a tárgyprogramok futtatását lassúvá és nehézkessé tette.

1.3. Turbo Pascal

Az UCSD Pascal rendszerrel párhuzamosan (talán kicsit később) indult a Borland cég **Turbo** jelzővel ellátott, nyelvi rendszereket létrehozó programja. A jelző a rendszer hatékony működésére utalt. A gépi kódban írt Turbo fordítóprogramok igen gyorsak voltak, a kapott tárgykód pedig közvetlenül végrehajtható, jól szervezett bináris program volt. A közbenső P-kódot elhagyták, a Pascal különböző gépek, illetve operációs rendszerek közötti átvitele nem a felhasználók, hanem a fejlesztő feladatává vált.

A Turbo Pascal első igazán sikeres változata 1983 táján került piacra 3.0-ás változatszámmal. Ezt a 4.0-ás, 5.0-ás, majd 1988-ban az 5.5-ös változatok követték. A sort jelenleg (1991 elején) a 6.0-ás változat zárja le. A következőkben az egyes változatok részleteinek mellőzésével összefoglaljuk a standard Pascal és az 5.5-ös változat közti különbségeket. Az eltérések egyrészt a standard Pascal egyes aktualitását vesztett vagy nem praktikus tulajdonságainak elhagyásából, másrészt a korszerű programozási igényeket elismerő bővítésekből állnak.

A standard szűkítése olyan megszorításokat tartalmaz, mint az azonosítók szignifikáns részének 63 karakterben való korlátozása, a függvények törzsében a függvényazonosítóra vonatkozó értékadási kötelezettség feloldása és az egyszerűbb B/K rendszer következtében feleslegessé vált **get** és **put** standard eljárások elhagyása. A gyakorlat szempontjából ezeknél sokkal érdekesebbek a bővítések:

- A lefoglalt szavak jegyzéke 16 új szót tartalmaz a különböző bővítésekkel kapcsolatban. Az új lefoglalt szavak a következők:

absolute	implementation	object	unit
constructor	inline	shl	uses
destructor	interface	shr	virtual
external	interrupt	string	xor

- Azonosítókban (nem első helyen) az aláhúzás (_) karakter is megengedett.
- Egész konstansok 16-os számrendszerben is megadhatók a $ prefix alkalmazásával.
- Címkeként azonosítók is használhatók.
- A karakterlánc (string) konstansok vezérlőkaraktereket és egyéb, nem nyomtatható szimbólumokat is tartalmazhatnak.
- Egy blokkban a címke, konstans, típus, változó, eljárás és függvénydeklarációk tetszőleges sorrendben, többször is előfordulhatnak.
- A típusok köre több irányban bővült. Az új típusok a következők:

  Egészek: shortint, longint, byte, word.
  Valósak: single, double, extended, comp.
  Objektum: object.

- A char és string típusok kompatibilisak.
- Változók előre rögzített memóriacímre is deklarálhatók az absolute alapjel segítségével.
- A változóra való hivatkozás tartalmazhat olyan mutató típusú függvényhívást, amelynek értéke egy dinamikus változóhoz vezet.
- A string típusú változók – egyes karaktereik elérése céljából – tömbként indexelhetők.
- Egy változó típusa a rá való hivatkozás idejére módosítható a típusváltás segítségével.
- Kezdőértékkel rendelkező, tetszőleges típusú változók is deklarálhatók (a file típus kivételével).
- Három új, logikai operátor használható: az shl, shr és xor.
- A not, and or és xor műveletek egész típusú értékekre is alkalmazhatók, bitenkénti műveletvégzést eredményezve.
- A + műveleti jel karakterláncok összekapcsolására (konkatenáció) is szolgál.
- A relációs operátorok karakterláncok összehasonlítására is használhatók.
- A @ új operátor, amely változók, eljárások és függvények címének meghatározására alkalmas.
- Egy kifejezés típusa az ún. érték-típusváltással módosítható.
- A case utasításban címkeként intervallumok is szerepelhetnek, és egy else ág is megengedett.
- Az eljárások és a függvények external, inline és interrupt direktívákkal is deklarálhatók az assembly nyelvű szubrutinok, a Pascal szövegbe írt gépi kódú blokkok és megszakításhoz kapcsolt eljárások alkalmazásának támogatására.
- Eljárásokban és függvényekben megengedett a típus nélküli, név szerinti (változó) paraméter használata. Ennek bármilyen típusú változó megfelelhet aktuális paraméterként.
- A moduláris programozás és a modulonkénti fordítás támogatására bevezették a programmodul (unit) fogalmát és kezelését.
- Az állományok kezelésére szolgáló standard eljárások és függvények csoportja a következő 16 elemmel bővült:

append	close	flush	rmdir
blockread	erase	getdir	seek
blockwrite	filepos	mkdir	seekeof
chdir	filesize	rename	seekeoln

- A read, readln, write és writeln eljárások string típusú értékek beolvasására, illetve kivitelére is alkalmasak.

◇ A standard eljárások és függvények csoportja a következő elemekkel bővült:

addr	freemem	maxavail	randomize
cseg	getmem	memavail	relase
concat	halt	move	runerror
copy	hi	ofs	seg
dseg	inc	paramcount	sizeof
dec	insert	paramstr	sptr
delete	int	pi	sseg
exit	length	pos	str
fillchar	lo	ptr	swap
frac	mark	random	upcase
		val	

◇ A standard modulokban további konstansok, típusok, változók, eljárások és függvények definíciója szerepel, melyek a modult felhasználó programokban rendelkezésre állnak (l. 3. fejezet).

2. Eljárások

Ha megpróbáljuk felkutatni az eljárások eredetét, a számítástechnika korai időszakában, jóformán a belső programozású gépek megjelenésével egyidőben megtaláljuk a **szubrutinokat**, melyek az eljárások „törzsfejlődésében" a kezdőpontnak tekinthetők. A szubrutinok szerepe a gépi kódot, illetve alacsony szintű nyelveket használó programozásban elsősorban a program szövegének rövidítése volt. Egy speciális vezérlésátadás (a CALL utasítás) segítségével a program egyes részeit akkor is végre lehetett hajtani, amikor a vezérlés éppen a program másik részén volt. Ez az utasítás ugyanis megjegyzi, hogy a program mely pontján hajtódott végre utoljára, és egy másik, a hívott szubrutin végén lévő RETURN utasítás segítségével a vezérlés visszaadható a folytatási pontra. Ilymódon elegendő volt egyszer elhelyezni a programban azokat a részeket, melyeket több helyről is végre akartunk hajtani. Az assembly típusú nyelvek esetében ez a mechanizmus ma is így működik.

A programok struktúrája azonban már a legegyszerűbb szubrutinok használatával is megváltozott. A monolitikus szerkezet fellazult, létrejött a „főprogram" (ami szubrutinokat hív) és az „alprogramok" (a hívott szubrutinok) fogalma. Mivel egy szubrutin további szubrutinokat is hívhat, a fő- és alprogram elnevezés viszonylagos, hiszen egy alprogram valamelyik általa hívott szubrutin szempontjából főprogramnak számít. Az így létrejött hierarchikus programszerkezet később a strukturált programozás egyik kiindulópontjául szolgált.

Mivel a programokhoz azonban nemcsak utasítások, hanem **adatok** is kellenek, a főprogram és az alprogramok közötti adatcsere problémája szintén a programozással egyidős. A kézenfekvő megoldás közös memóriaterületek, illetve változók alkalmazása volt. A főprogram a memória egy „megbeszélt" helyére tette a felhasználandó adatokat, az alprogram ezeket feldolgozta, majd az esetleges eredményeket hasonló módon adta vissza. Ma nem csak az assembly, hanem ennél magasabb szintűnek számító nyelvek – például a Basic – is ezt a módszert alkalmazzák. A hívó program és a szubrutin közötti adatcserét tehát lényegében a programozóra bízták, aki minden hívás előtt és után az alprogram aktuális felhasználására jellemző módon oldotta meg ezt a feladatot. Ezáltal a hívó és a hívott program között olyan, nemkívánatos

kapcsolatok is létrejöttek (pl. hogy hol kell elhelyezni egy négyzetgyökvonó szubrutin számára az adatot, és hol kapjuk az eredményt), amelyek lehetetlenné vagy nagyon nehézkessé tették a szubrutin másik programban való felhasználását, holott a fejlődés éppen a sokoldalúan használható szubrutinkönyvtárak irányába mutatott.

A megoldást a magas szintű nyelvek általános paraméterkezelő mechanizmusa hozta, amely szabványosította az adatok és eredmények cseréjét a hívó és hívott program között, és ezzel függetlenné tette azt a konkrét felhasználástól. Ez a módszer világméretekben az ALGOL–60 programozási nyelvvel együtt terjedt el, és egyben módosította a terminológiát is. Szubrutinok helyett **eljárásokról** beszélünk, melyeket külön definiálni, azaz **deklarálni** kell; a deklarációban szereplő paramétereket **formális**, az eljárás hívásában használtakat pedig **aktuális** paraméternek nevezzük. Az aktuális és formális paramétereknek számukban és típusukban illeszkedniük kell egymáshoz, de nevük (memóriabeli helyük) tetszőleges lehet, és a közöttük bonyolódó információcsere nem terheli a programozót. Az ilymódon egységesített paraméterátadás lényegében két módszert alkalmaz az aktuális és a formális paraméterek közötti adatcserére:

* Átmásolható az aktuális paraméter értéke a formális paraméterbe, ezt **érték szerinti** paraméterátadásnak nevezik.
* Átvihető az aktuális paraméter címe az eljárásba, s ebben az esetben **név szerinti** (cím vagy hivatkozás szerinti) átadásról beszélünk.

Az érték szerinti formális paraméternek az eljárás hívásakor egy (megfelelő típusú) tetszőleges kifejezés feleltethető meg aktuális paraméterként, míg a név szerinti formális paraméterek aktuális megfelelője csak egy (megfelelő típusú) változó lehet.

A paraméterkezelés egységes megoldása nemcsak bármely programban könnyen felhasználható eljárások kidolgozását tette lehetővé, hanem a további fejlődés útját is megnyitotta. Így az egyetlen értéket szolgáltató eljárásokat például bizonyos mértékig elkülönítették és használatukat a megszokott jelölésrendszerhez közelítették. Ezeket **függvénynek** nevezik, értéküket nem paraméter, hanem saját nevük hordozza és ezért nem külön hívó utasítással aktivizálhatók, hanem kifejezésekben közvetlenül felhasználhatók, s hívásukra a kifejezés kiértékelése folyamán kerül sor. A programozási fogalmak és az algoritmusok rekurzív természetének elismeréseként alakult ki az eljárások és függvények egy speciális osztálya. Ezek végrehajtásuk során önmagukat felhasználó algoritmusokat valósítanak meg, ezért kapták a **rekurzív** jelzőt. A rekurzív eljárások kezelése egy időben különleges programozási teljesítménynek számított, ezért a magas szintű nyelvek közül sem mindegyik engedi meg használatukat.

A Turbo Pascal esetében az eljárások már az 5.0-ás változatnál elérték a mai szintet, amivel a következőkben megismerkedünk.

2.1. Az eljárás mint programozási eszköz

A Pascal-eljárások egyben programblokkok is, amelyek nemcsak névvel és paraméterekkel (ezeket együttesen **eljárásfejnek** nevezik), hanem saját belső konstansokkal, típusokkal, változókkal, illetve eljárásokkal rendelkezhetnek. Az eljáráson belül (annak **törzsében**) deklarált azonosítókat a **lokális** jelzővel lehet megkülönböztetni a program egyéb helyein szereplő, ún. **globális** azonosítóktól. A lokális azonosítók csak az eljáráson belül használhatók, és az sem baj, ha egy lokális és globális azonosító megegyezik, a közös névvel ilyenkor csak a lokális azonosítóra lehet hivatkozni.

Az eljárás feje a **procedure** vagy **function** lefoglalt szóval kezdődik, melyet az eljárás neve, valamint a formális paraméterek nevének és típusának felsorolása (függvény esetén még az eredmény típusa is) követhet.

Az eljárás törzse általános esetben egy programblokk, amint ez a következő eljárásdeklarációból is kitűnik:.

```
procedure csere(var a, b: integer);
var m: integer;
begin m:=a; a:=b;
      b:=m
end;
```

Két speciális esetben az eljárástörzs nem blokk, hanem a **forward**, illetve az **external** lefoglalt szavakból áll.

Mivel egy eljárástörzsben csak már deklarált eljárásokra szabad hivatkozni, két egymást kölcsönösen hívó eljárás esetén valamelyik deklarációt késleltetni kell. Ezt oldja meg a **forward** lefoglalt szó, amely a teljes törzset helyettesíti. Később természetesen meg kell adni a forward szóval pótolt törzset, de ekkor az eljárás feje helyett csak a procedure szót és az eljárás nevét kell használni. Ezt az eljárás/függvényhívás szerkezetet **kölcsönös rekurziónak** nevezik. A következőkben egy kölcsönös rekurzió vázlatát mutatjuk be.

```
procedure proc1(i, j: integer);
csak forward;

procedure proc2(x, y: real);
begin
      .
      .
      .
     proc1(1, 2);
      .
      .
end;
```

```
    {Itt további, proc1-et hívó eljárásdeklarációk is lehetnének}
procedure proc1;                          {A tényleges deklaráció}
begin
      .
      .
      .
      proc2(1.5, 2.99);
      .
      .
end;
```

Az assembly nyelven írt, külön fordított eljárások törzsét a Pascal programban az **external** lefoglalt szóval helyettesítjük. Az eljárás feje a szokásos marad:

```
procedure bytevitel(var honnan,hova;mennyit:word);external;
```

Az eljárás assemblerrel generált tárgykódját tartalmazó állomány nevét **(obj_kod)** a Pascal programban a

{$L obj_kod}

direktívával kell a főprogramhoz kapcsolni. A Turbo Pascal természetesen a Turbo Assembler használatát feltételezi, és olyan további előírásokat tartalmaz (pl. a szegmensnevekkel kapcsolatban), melyek az assembly programozás legalább alapfokú ismeretét feltételezik.

Ha rövid gépi kódú betéteket akarunk az eljárás törzsében elhelyezni, nem érdemes külön fordított assembly programot írni, elegendő az **inline** direktíva használata. A direktívát makróként dolgozza fel a rendszer, törzséből a hívás helyén és idejében 1 vagy 2 byte-os egységekből álló kód generálódik. Például a megszakítás ki és bekapcsolására szolgáló inline eljárások a következők:

```
procedure MegszakitasKi; inline($FA);
procedure MegszakitasBe; inline($FB);
```

($FA és $FB a CLI és STI utasítások kódjai).

Az inline direktíva zárójelek között álló törzse egymástól /-lal elválasztott elemekből áll. Minden elem vagy konstans, vagy egy változó neve lehet, melyhez + vagy − előjellel növekmény is csatlakozhat.

Például: **inline**(15/$FFFF/X1+2/Y1−10);

Kifejtéskor egy elemből 1 byte generálódik, ha az elem 255-nél nem nagyobb értékű konstans. Ha az elem változónév vagy 255-nél nagyobb konstans, kifejtés után 2 byte-ot foglal el. Változó esetén ez a szegmensrelatív cím.

Minden inline elem előtt megengedett a < vagy > méretjelző használata is. A < méretjelző csak 1 (az alsó) byte generálását engedélyezi 2 byte-os elem esetén is, a > jel pedig minden esetben 2 byte generálását kényszeríti ki (esetleg üres felső byte-tal).

Ennek értelmében az

inline(<$AABB/>$FF)

direktívából 3 byte generálódik: $BB, $FF, $00.

Valamivel hosszabb, de még inline kezelést kívánó függvényt tartalmaz a következő példa. A függvény két egész számot szoroz össze, értéke **longint** típusú.

```
function hszor(a, b: integer): longint;
inline($5A/          POP AX
       $58/          POP DX
       $F7/EA);      IMUL DX;  DX:AX := a*b
```

A függvény működésének megértéséhez tudni kell, hogy a lokális változók a veremszegmensben (SS regiszter) kapnak helyet.

Az érték szerint átadott paraméterek csak annyiban különböznek az eljárás lokális változóitól, hogy a fejrészben kell őket felsorolni, és az eljáráshíváskor értéket kapnak a megfelelő formális paramétertől. Az eljárás hívásakor az értékparaméterek a lokális azonosítókkal együtt a veremben kapnak helyet. Ennek megfelelően, ha az eljárás törzsében a címképző @ operátort egy értékparaméterre alkalmazzuk, a verem azon elemére hivatkozó mutatót kapunk, amely a paraméter értékét tartalmazza.

Ha tehát **par1** értékparaméter, **m1** mutató és az eljárásban végrehajtódik az

```
m1 := @par1;
```

utasítás, **m1**^ a verem azon elemére hivatkozik, amelyik **par1** értékét tartalmazza.

A név szerint átadott (ún. változó) paramétereket az eljárás fejében a **var** lefoglalt szó különbözteti meg az értékparaméterektől. A változó paraméterek esetében az eljárás hívásakor a verembe az aktuális paraméterként adott változó címe kerül. Ha a @ operátort egy ilyen paraméterre alkalmazzuk, az aktuális paraméterre hivatkozó mutatót kapunk. Ha tehát **nev** egy változó paraméter, **m1** mutató és **Irma** az eljárás hívásában a **nev**-nek megfelelő változó, az

```
m1 := @nev;
```

utasítás **m1**-be **Irma**-ra hivatkozó mutatóértéket ír be, ezért **m1**^ az aktuális paraméter (**Irma**) értékét jelöli.

Az érték és név szerinti paraméterek használata közötti választás a programozó hatáskörébe tartozik, azzal a megszorítással, hogy az eljárásból értéket (eredményt) kihozni csak változó paraméterrel lehet, és állomány típusú (file) paraméter csak név szerint kezelhető. A fentiek alapján ezt a szabályt

azzal a hatékonysági szempontokat figyelembe vevő tanáccsal egészíthetjük ki, hogy nagyméretű paramétereket (tömbök, terjedelmes rekordok) akkor is célszerű változó paraméterként kezelni, ha nem hoznak ki értéket az eljárásból, mivel ellenkező esetben a verembe való átmásolásuk minden hívásnál számottevő hely- és időpazarlást okoz.

A Turbo Pascal egyik újítása, az ún. **típus nélküli** változó paraméter, sok esetben igen általános eljárás megírására ad lehetőséget. Ha egy változó paraméterhez nem adunk meg típust, akkor ahhoz az eljárás hívásakor tetszőleges típusú aktuális paraméter rendelhető.

Az eljárás törzsében a típus nélküli paraméter egyetlen típussal sem kompatibilis, értéket csak típusváltással (typecast) kaphat.

A következő példában egy univerzális függvényt mutatunk be, amely két, bármilyen típusú változó értékének összehasonlítását elvégzi, melyek mérete nem haladja meg a 10 000 byte-ot.

```
function azonos(var egyik, masik; hossz: word): Boolean;
type vektor = array [1..10000] of byte;
var i: integer;

begin i:=1;
      while(i<=hossz)and(vektor(egyik)[i]=vektor(masik)[i])do
            inc(i);
      azonos:=(i=hossz+1)
end;
```

Legyen például **t1** egy 2$*4-es tömb és **t2** egy 10 elemű vektor:

```
var t1: array[1..2,1..4] of integer;
    t2: array[1..10] of integer;
```

és tekintsük a következő kifejezéseket:

```
azonos(t1[1],t2[1],4*sizeof(integer));
```

t1 első sorát hasonlítja össze **t2** első 4 elemével,

```
azonos(t1[2,2],t2[6],3*sizeof(integer));
```

t1 2. sorának utolsó 3 elemét hasonlítja össze **t2** 6., 7. és 8. elemével.

A később sorra kerülő **sizeof** standard függvény valamely típushoz tartozó memóriaterület méretét adja meg byte-okban.

Az a tény, hogy a Turbo Pascal által generált kód, valamint a tárgyprogram megszakítható, a futásakor használt szubrutinok nagy része pedig többszörösen folytatható (reentrant), lehetővé teszi megszakításkezelő eljárások Pascalban való írását. Ezek feje azonban speciális paramétereket és az **interrupt** lefoglalt szót tartalmazza:

```
procedure MegszakKezeles(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,
interrupt;                                          ES,BP: word);
begin
      .
      .
end;
```

Láthatjuk, hogy a fejben a regiszterek szerepelnek paraméterként. Ha az eljárás törzsében nem akarunk rájuk hivatkozni, a paramétersorozat bal végéről indulva, kihagyás nélkül egy vagy több (akár az összes) paraméter felsorolása elhagyható. A megszakításkezelő eljárásokat nem szabad a szokásos eljáráshívással aktivizálni, a hozzájuk tartozó megszakításkód standard eljárásokkal (l. később) állítható be. Ha egy megszakításkezelő eljárás módosítja paramétereit, ennek eredménye csak a megszakítás kezelése után érvényesül. A hardver által generált megszakításkezelő eljárásokban nem szabad Pascal B/K eljárásokat és MS–DOS függvényeket hívni, mert ezek nem többszörösen folytathatók. A megszakításkezelő eljárás hívásakor a regiszterek mentése, kilépéskor pedig visszaállítása automatikusan megtörténik.

2.2. Az eljárástípus

A standard Pascal egyik jelentős bővítése a Turbo Pascal **eljárás** (**procedure** vagy **function**) típusa. Ennek segítségével az eljárások a többi azonosítóval jelölt fogalommal azonos kategóriába kerültek (például változóknak értékül adhatók, eljárás vagy függvény paramétereként szerepelhetnek stb.).

Az eljárástípus deklarációja a többi típusdefinícióhoz igazodik, maga a típus pedig olyan eljárás (vagy függvény) fej, amelyből elhagytuk az eljárás nevét:

```
type e1 = procedure;
     txt = procedure(var st: string);
     numfv = function(z: real): real;
     regrfv = function(x, y: real; fz: numfv): real;
```

Az első példa egy paraméter nélküli eljárástípust deklarál, az utolsó pedig paraméterként felhasználja az előző sorban definiált **numfv** típust. A típusdeklarációban szereplő formálisparaméter-nevek csakis a paraméterek számának és típusának leírására szolgálnak, más jelentőségük nincs. Mint látjuk, a függvénytípusoknál meg kell adni a függvény értékének típusát is, ami **nem lehet** eljárástípus.

Az eljárásváltozók deklarációja a szokásos formájú. A fenti típusokat felhasználva:

```
var stf: txt;
      f: numfv;
```

Ezzel szert tettünk két eljárásváltozóra, amelyek kézenfekvő módon eljárásértékeket vehetnek fel. Eljárásértékek vagy egy másik eljárásváltozóból, vagy eljárásazonosítóból származhatnak. Az eljárásazonosító itt a más (pl. integer) típusoknál megszokott konstansok szerepét játssza. Figyeljük meg a következő eljárást:

```
procedure fordit(var s: string);
var i, h: integer;
    m: char;
begin h:=length(s);
      for i:=1 to h div 2 do
          begin m:=s[i]; s[i]:=s[h-i+1];
                s[h-i+1]:=m
          end;
end
```

Az eljárás a paraméterként kapott karakterláncban megfordítja a karakterek sorrendjét. A **fordit** eljárás kompatibilis a **txt** típussal (mindkettőnek egyetlen, string típusú változó paramétere van), ezért elvégezhető a következő értékadás:

```
stf := fordit;
```

Ezután pedig a

```
fordit('abcdefgh') és az  stf('abcdefgh')
```

eljárásutasítások hatása azonos lesz.

Hasonló okokból helyes az f:=log2 értékadás is, ha **log2** valós paraméterű és valós értékű függvény.

Az eljárások értékadásának szabályai elsősorban az ún. **értékadás kompatibilitást** írják elő (ez más értékadásnál is követelmény), ami akkor teljesül, ha a két eljárástípushoz azonos számú paraméter tartozik, és az egymásnak pozíciójuk szerint megfelelő paraméterek azonos típusúak. Függvények esetén természetesen az eredménynek is azonos típusúnak kell lennie. A formális paraméterek elnevezése a kompatibilitás szempontjából közömbös.

Ahhoz, hogy egy eljárást (eljárás „konstanst") vagy függvényt valamely eljárásváltozó értékül kaphasson, néhány további feltételnek is teljesülnie kell:

- Az eljárást {**$F+**} **állapotban** (az ún. távoli (FAR) CALL utasítások generálásával) kell fordítani.
- Az eljárás **nem lehet standard eljárás** vagy függvény.

- Az eljárás **nem lehet skatulyázott**, azaz egy másik eljáráson belül deklarált.
- Az eljárás **nem lehet interrupt** vagy **inline** típusú.

A standard eljárásokra vonatkozó tilalom könnyen megkerülhető oly módon, hogy a standard eljárást egy saját eljárásba ágyazva használjuk. Ezt szemlélteti az alábbi példa is.

```
procedure stringir(s: string);
begin writeln(s)
end;
```

A skatulyázott eljárásokra vonatkozó értékadási tilalmat mutatjuk be a következő programban.

```
                program skatulya;
                $F+

                type proc = procedure;

                var xx: proc;

                procedure p1;

                procedure p11;
                begin
                end;

                procedure p12;
                begin
                end;

p1 törzse:    begin xx:=p11;      {Mindkét értékadás hibás, mert}
                    xx:=p12     { p11 és p12 skatulyázott eljárás}
              end;

A főprogram: begin p1
              end.
```

A **p1** eljárás törzsében az **xx proc** típusú változóra vonatkozó értékadások nem megengedettek, **p11** és **p12** deklarációja **p1** fejében van.

Az eljárás típusú változók természetesen nemcsak önmagukban, hanem más strukturált típusok részeiként is használhatók. Így például beszélhetünk eljárásokból alkotott tömbről vagy olyan rekordokról, melyekben egyes mezők eljárás típusúak (lásd az alábbi programot).

```
. program profil; {eljárásokat "tartalmazó" állomány}
. {$F+}
. type proc = procedure(par1, par2: string);
.      adat = record m1, m2: string;
.                    alg: proc
.             end;
.
. procedure sp1(s1, s2: string);
. begin writeln(s1,'-',s2)
. end;
.
. procedure sp2(s1, s2: string);
. begin writeln(s2,'-',s1)
. end;
.
. var r1: adat;
.     procfile: file of adat;
.
. begin assign(procfile,'pf.prc'); rewrite(procfile);
.       with r1 do
.            begin m1:='Bal oldali'; m2:='Jobb oldali';
.                  alg:=sp1; alg(m1,m2); write(procfile,r1);
.                  alg:=sp2; alg(m2,m1); write(procfile,r1)
.            end;
.       close(procfile)
. end.
```

Az **r1** rekord harmadik mezője (**alg**) két **string** paraméterrel rendelkező eljárás (**proc**) típusú, **procfile** pedig **adat** típusú rekordokat tartalmazó állomány. A program beállítja **m1** és **m2** tartalmát, kétféle érteket ad az **alg** mezőnek, mindkettőt mint eljárást végrehajtja, és kiviszi **procfile**-ba is. A program kifogástalanul működik, kiír két azonos szöveget a képernyőre (mivel a két eljáráshívásnál felcseréltük a paramétereket), és létrehozza a **pf.prc** nevű állományt.

Sajnos, ez az állomány egy másik programban nem használható úgy, ahogy ezt a Pascal állományoktól eddig megszoktuk. Ha megnyitjuk és beolvassuk a **pf.prc** első rekordját, az egyszerűség kedvéért újra az **r1** változóba:

```
. assign(procfile,'pf.prc'); reset(procfile);
. read(procfile,r1);
```

még minden rendben van. Az **r1.alg** eljárás hívása azonban végzetes a Pascal vagy (ha innen hívtuk) az MS–DOS számára, s gépünk csak újraindítás után lesz üzemképes. A jelenség magyarázata kézenfekvő: amikor egy eljárás típusú változó értéket kap, egy cím (vagy mutató) kerül bele, az értékül adott eljárás kezdőcíme. Ez a cím kiíródik az állományba és vissza is kerül,

amikor a másik program beolvassa. Ebben a programban azonban az **r1.alg** változóba beolvasott címen nyilván nem szerepel az **sp1** vagy **sp2** eljárás, mint a fenti, **profil** nevű program esetén.

A példából az a tanulság vonható le, hogy az eljárástípus sokban hasonlít ugyan a többi típusra, de jelentős eltérések is mutatkoznak. Ha azt akarjuk, hogy az eljárásokat a hagyományos értelemben vett adatok módjára is (és természetesen eljárásként is) kezelhessük, ennél a típusnál valamilyen értelemben erősebb adatszerkezetre van szükség. Ez a gondolatmenet tulajdonképpen a 4. fejezetben sorra kerülő **objektumokhoz** vezet.

Ezen „hiányossága" ellenére az eljárástípus sok mindenre használható. Ezek egyike az eljárás típusú paraméter. A következő program két valós érték háromféle átlagát határozza meg úgy, hogy a kívánt átlagot számító függvényt paraméterként adjuk át az eredményt kiíró eljárásnak.

```
. program procpar;
. {$F+}
. type atlag = function(a, b: real): real;
.
. function szamtani(a, b: real): real;
. begin szamtani:=(a+b)/2
. end;
.
. function mertani(a, b: real): real;
. begin mertani:=sqrt(a*b)
. end;
.
. function harmonikus(a, b: real): real;
. begin harmonikus:=2/(1/a+1/b)
. end;
.
. {$F-}
. procedure atlagol(x, y: real; melyik: atlag);
. begin write(melyik(x,y):8:2)
. end;
.
. var x, y: real;
. begin write('x= '); readln(x);
.       write('y= '); readln(y);
.       atlagol(x,y,szamtani);
.       atlagol(x,y,mertani);
.       atlagol(x,y,harmonikus);
.       writeln
. end.
```

Az eljárásváltozókat (különösen, ha függvényre hivatkoznak) kifejezésekben is használhatjuk. Ilyenkor általában a változóban „tárolt" eljárás vagy

függvény végrehajtódik. Van azonban olyan helyzet is, amikor a fenti szabály nem érvényes. Ha egy értékadó utasítás bal oldalán eljárásváltozó áll, a jobb oldali eljárás vagy függvény végrehajtása helyett, annak címe íródik be az eljárásváltozóba. Ebben az esetben a kivételes helyzet a bal oldali eljárásváltozóról egyértelműen felismerhető. A fenti program jelöléseinél maradva legyen **fv** egy **atlag** típusú, **z** pedig valós változó:

```
var fv: atlag;
    z: real;
```

Az fv:=szamtani;
és a z:=szamtani(2,6);
utasításokban a függvény paramétereinek hiánya vagy jelenléte is támpontot nyújt. Nézzük azonban a következő deklarációkat:

```
type nincspar = function: real;

function f1: real;
begin f1:=Pi
end;

var x: real;
    y: nincspar;
```

Ha végrehajtjuk az

```
x:=f1;,        illetve az
y:=f1;
```

értékadást, az első esetben **f1** végrehajtódik, és **x** értéke **Pi** lesz; a második esetben viszont **f1** nem hajtódik végre, hanem címe **y**-ba íródik. Itt a tájékozódás nehezebb, de még egyértelmű.

Az eljárásváltozók használata során azonban adódhat az eddigiek alapján el nem dönthető helyzet is. Az

```
if y = f1 then write('Mi mennyi?')
```

utasítás egyaránt értelmezhető annak vizsgálataként, hogy a **nincspar** típusú változó (**y**) értéke éppen az **f1** függvény-e, valamint úgy is (ahogyan ezt a standard Pascal és annak minden utóda előírja), hogy egy függvényazonosító megjelenése a kifejezésben a függvény hívását és értékének behelyettesítését eredményezi. Ennek megfelelően a fenti utasítás az **y**-ban tárolt értéket, illetve az **f1** függvény helyettesítési értékét hasonlítja össze. A Turbo Pascal ezen utóbbi értelmezést választotta, és a **@ címoperátort** használja eljárás típusú értékek összehasonlítására:

```
if @y = @f1 then write('y értéke f1')
```

A címoperátor egy eljárásváltozóra alkalmazva (**@y**), az eljárás hívása helyett az eljárásváltozót egy típus nélküli mutatóvá alakítja. A **@f1** kifejezés pedig – az operátor eredeti szerepe szerint – **f1** címét adja. Ezek viszont már problémamentesen összehasonlíthatók.

Ezért, ha egy eljárás típusú változó tényleges címére vagyunk kíváncsiak, a @ operátort kétszer kell alkalmazni: **@@y** a fenti változó fizikai címét adja.

Az eljárásváltozók kifejezésekben való használata során a típusváltás (typecast) teljes fegyvertára változatlan formában alkalmazható. Legyenek érvényesek a következő deklarációk:

```
type fuggveny = function(i: integer): integer;
var   f: fuggveny;
    mut: pointer;
      j: integer;
```

A típusváltás igen érdekes kombinációkra ad lehetőséget:

```
f:=fuggveny(mut);        {A mut-ban levő eljárásérték íródik f-be.}
fuggveny(mut):=f;        {Az f-ben levő eljárásérték íródik mut-ba.}
@f:=mut;                 {A mut-ban levő mutatóérték íródik f-be.}
mut:=@f;                 {Az f-ben levő mutatóérték íródik mut-ba.}
j:=f(j);                 {Az f-ben levő függvény hívása.}
j:=fuggveny(mut)(j);     {A mut-ban levő mutató által megcímzett
                          függvény hívása j paraméterrel.}
```

Az eljárástípus ilyen általános formában való alkalmazása sok esetben hasznos, azonban olyan következményekkel is jár, amelyek már túlmutatnak a standard Pascalon. Ha figyelembe vesszük, hogy ezzel a típussal a Pascal felzárkózott a korszerű programozási nyelvek élvonalához, értékelése mindenképpen pozitív.

2.3. Standard eljárások

A Turbo Pascal standard eljárásainak és függvényeinek deklarációja a minden program által automatikusan használt **System** standard modulban (l. 3. fejezet), régebbi nevén a programok futtatását támogató könyvtárban található. A standard eljárások neve a programban más célra korlátozás nélkül felhasználható (bár nem ajánlott, mert a program megértését nehezebbé teszi). A többi standard modulban további eljárásdeklarációk szerepelnek, ezekkel a 3. fejezetben foglalkozunk. A System modul eljárásainak a többi standard modultól elkülönített tárgyalását pusztán a hagyományok tisztelete diktálja; ezen eljárások jó része már a standard Pascalban, majd a Turbo Pascal standard modulokat még nem ismerő, korábbi változataiban is szerepelt. A standard eljárásokat funkciójuk szerint csoportosítjuk, az egyes csoportokon belül pedig a betűrendet követjük.

2.3.1. Bevitel és kivitel (B/K)

Ebbe a csoportba a különböző állományok kezelését végző eljárások és függvények tartoznak. A Turbo Pascal állományok három csoportba sorolhatók: **általános, szöveges és típus nélküli** állományok. Két standard **szöveges** állományváltozóhoz, az **Input**-hoz és az **Output**-hoz tartozó állomány minden program indulásakor automatikusan megnyílik. Az **Input** a billentyűzethez rendelt, csak olvasható, az **Output** pedig a képernyőhöz rendelt, csak írható állomány. Ha szöveges állományokra vonatkozó eljárásokban vagy függvényekben az állomány nevének megfelelő paramétert elhagyjuk, az eljárás értelemszerűen a standard állományok egyikével dolgozik.

A rendszer alaphelyzetében a B/K műveletek ellenőrzése automatikusan történik. Hiba előfordulásakor a program végrehajtása egy hibaüzenet kiírása után megszakad. Az automatikus ellenőrzés a {$I−} és a {$I+} fordítóprogram-direktívákkal ki és be kapcsolható. Kikapcsolt állapotban a program futása B/K hiba esetén sem szakad meg, de egy esetleges következő B/K művelet végrehajtása felfüggesztődik az **IOresult** standard függvény (l. később) hívásáig.

A továbbiakban **ftip** valamilyen file-típust jelöl.

Először az összes állományra nézve közös B/K eljárásokkal foglalkozunk:

procedure Assign(var av: ftip; dosnev: string);

Az **av** állomány változóhoz a **dosnev string** típusú kifejezés értékét rendeli külső névként. **dosnev** az MS–DOS-ban állománynévként elfogadható karakterlánc. Ha **dosnev** üres (nulla hosszúságú) karakterlánc, **av**-hoz az **Input** és az **Output** standard állomány rendelődik. Meglevő állomány megnyitásának vagy új létrehozásának első lépése az Assign hívása.

procedure ChDir(dirnev: string);

Az aktuális könyvtárat **dirnev**-re változtatja. **dirnev string** típusú kifejezés, melynek értéke az MS–DOS-ban könyvtárnévként elfogadható karakterlánc.

procedure Close(var av: ftip);

Az **av** nyitott állomány lezárása az esetleg függőben levő adatátvitel lebonyolítása után.

function Eof(var av): Boolean;

Értéke **true**, ha az aktuális állománypozíció túl van az utolsó elemen, különben **false**.

procedure Erase(var av: ftip);

Az **av**-hoz tartozó külső állomány törlése az MS–DOS könyvtárból. Az állományt törlés előtt le kell zárni (l. Close).

procedure GetDir(l: byte; var s: string);

Az **l** egész kifejezéssel meghatározott lemezegység aktuális könyvtárának nevét adja **s**-ben. **l**=0 az aktuális lemezegységet, **l**=1 az A:, **l**=2 a B: stb. egységet jelenti.

function IOResult: word;

Az utolsó B/K művelet kimenetelére jellemző egész értéket ad. Ha értéke 0, a művelet sikeres volt. Az egyes hibakódokat a Turbo Pascal Reference Guide tartalmazza.

procedure MkDir(dirnev: string);

Létrehozza a **dirnev** nevű alkönyvtárat. **dirnev string** típusú kifejezés, melynek értéke az MS–DOS-ban könyvtárnévként elfogadható karakterlánc.

procedure Rename(var av: ftip; ujnev: string);

Az **av** állomány külső nevét **ujnev**-re változtatja. **ujnev string** típusú kifejezés, melynek értéke az MS–DOS-ban állománynévként elfogadható karakterlánc. Az eljárás csak lezárt állományra használható.

procedure Reset(var av: ftip [; elemhossz: word]);

Az **av** létező állomány megnyitása. Ha **av text** típusú, csak olvasható lesz. Ha **av** típus nélküli állomány, **elemhossz** megadhatja az adatátvitelhez szükséges rekordméretet byte-okban. Ennek alapértéke 128.

procedure ReWrite(var av:ftip; [; elemhossz: word]);

Az **av** új állomány létrehozása és megnyitása. Ha **av text** típusú, csak írható lesz. Ha **av** típus nélküli állomány, **elemhossz** megadhatja az adatátvitelhez szükséges rekordméretet byte-okban. Ennek alapértéke 128.

procedure RmDir(dirnev: string);

Megszünteti a **dirnev** nevű üres alkönyvtárat. **dirnev string** típusú kifejezés, melynek értéke az MS–DOS-ban könyvtárnévként elfogadható karakterlánc.

A következő eljárások a **típus nélküli állományok kivételével** minden állományra használhatók:

procedure read(var av: text; e1 [, e2, ...en]);

A szöveges állományokra vonatkozó olvasó eljárás egy vagy több értéket olvas be **av**-ból és helyez el **e1**, **e2**, ... **en**-ben. Az állománypozíció automatikusan módosul. **e1**, **e2**, ... **en** típusa **char**, valamilyen **integer**, **real** vagy **string** lehet.

Ha **e1**, **e2**, ... **en char** típusú, az állományból **n** karakter olvasódik be. A sorok végén beolvasódik a Chr(13), az állomány végén pedig a Chr(26) (CTRL–Z) karakter is.

Ha **e1**, **e2**, ... **en** valamilyen **integer** típusú, az állományból **n** darab, előjeles egész számot alkotó karaktersorozat olvasódik be, melyek értéke

e1, **e2**, ... **en**-ben tárolódik. A karaktersorozatokat egymástól legalább egy szóköz, TAB vagy sorvége karakter választja el.
Ha **e1**, **e2**, ... **en** valamilyen **real** típusú, az egész számokhoz hasonló szabályok szerint, előjeles valós számokat alkotó karakter sorozatok olvasódnak be.
Ha **e1**, **e2**, ... **en** valamilyen **string** típusú, egy sor olvasódik be **e1**-be a sorvége karakter nélkül. n>1 esetén a többi **e üres marad**.

procedure `read(var av: ftip; e1 [, e2, ... en]);`
Tetszőleges, nem szöveg típusú elemeket tartalmazó állományokra vonatkozó olvasó eljárás; egy vagy több értéket olvas be **av**-ból és helyez el **e1**, **e2**, ... **en**-ben. Az állománypozíció automatikusan módosul. **e1**, **e2**, ... **en** típusa **av** alaptípusával azonos.

procedure `write(var av: text; e1 [, e2, ...en]);`
A szöveges állományokra vonatkozó író eljárás az **e1**, **e2**, ... **en** kifejezések értékét írja az **av** állományba. Az állománypozíció automatikusan módosul. **e1**, **e2**, ... **en** típusa **char**, valamilyen **integer**, **real**, **string** vagy **Boolean** lehet.
A kifejezések után, azoktól kettősponttal elválasztva egy, a kiírt mező minimális szélességét előíró paraméter is megadható egy egész értékű kifejezés formájában. Valós értékek írása esetén a szélesség paramétertől szintén kettősponttal elválasztva, a tizedesjegyek számát rögzítő paraméter is szerepelhet. Ezen paraméterek hiányában a kiírt mezők az értéküknek megfelelő számú pozíciót foglalnak el. A valós értékek exponenciális formában, 17 pozíción jelennek meg.

procedure `write(var av: ftip; e1 [, e2, ...en]);`
A tetszőleges, **nem szöveg** típusú állományokra vonatkozó író eljárás az **e1**, **e2**, ... **en** kifejezések értékét írja az **av** állományba. Az állománypozíció automatikusan módosul. **e1**, **e2**, ... **en** típusa megegyezik **av** alaptípusával.

A következő eljárások és függvények **szöveges állományok kivételével** minden állományra használhatók:

function `FilePos(var av)`: **longint**;
Értéke az **av** állomány aktuális pozíciója. Az állomány elején ez 0.

function `FileSize(var av)`: **longint**;
Értéke az **av** állományban levő elemek (komponensek) száma.

procedure `Seek(var av; n:` **longint**`);`
Az **av** állomány aktuális pozícióját **n**-re állítja. Az első elem pozíciója 0. A **Seek(av, Filesize(av))** hívás megengedett, az állomány pozícióját az utolsó elem utánra állítja. Ilyenkor végrehajtott írással az állomány bővíthető.

procedure Truncate(**var** av);

Az **av** állományt csonkítja oly módon, hogy méretét az aktuális pozícióhoz igazítja. Az állomány aktuális pozíciója utáni elemek törlődnek.

A szöveges (**text** típusú) állományokra vonatkozó függvények és eljárások:

procedure Append(**var** av: **text**);

Az **av** szöveges állományt úgy nyitja meg, hogy az bővítésre készen álljon. Az állomány pozícióját az utolsó elem utánra állítja. Az ott levő CTRL–Z állományvége jel törlődik. Az állomány csak írhatóvá válik.

function Eoln(**var** av: **text**): **Boolean**;

Értéke **true**, ha az aktuális állománypozícióban állományvége (CTRL–Z) vagy sorvége karakter (Chr(13)) van, különben **false**.

procedure Flush(**var** av: **text**);

Írásra nyitott, szöveges állományhoz tartozó puffer azonnali ürítését eredményezi. A puffer tartalma fizikailag átíródik a külső állományba.

procedure Readln(**var** av: **text** [; e1, e2, ... en];

A **read** (l. fent) végrehajtása után az állománypozíció automatikusan a következő sor elejére lép. Az **e1**, **e2**, ... **en** paraméterek nélkül csak az új sorra való áttérés következik be.

function SeekEof(**var** av: **text**): **Boolean**;

Értéke azonos az **Eof** függvényével, de az állományvége-helyzet vizsgálata előtt átlép minden szóköz-, TAB és sorvége (Chr(13)) karaktert.

function SeekEoln(**var** av: **text**): **Boolean**;

Értéke azonos az **Eoln** függvényével, de a sorvége-helyzet vizsgálata előtt átlép minden szóköz- és TAB karaktert.

procedure SetTextBuf(**var** av: **text**; **var** puffer [; hossz: **word**]);

A szöveges állományhoz automatikusan hozzárendelt, 128 byte hosszú **puffer** helyett másik (paraméterként adott) **puffer** kijelölése. A **hossz** paraméterrel a **puffer** hossza korlátozható, hiánya esetén ez **SizeOf(puffer)** lesz. Az eljárás csak lezárt állományokra vagy legkésőbb közvetlenül a **Reset**, **Rewrite** vagy **Append** után hívható, megelőzve minden egyéb B/K műveletet.

procedure Writeln(**var** av: **text** [; e1, e2, ... en];

A **Write** eljárás végrehajtása után egy sorvége karaktert (Chr(13)) visz ki az állományba. Az **e1**, **e2**, ... **en** paraméterek nélkül csak a sorvége karakter íródik az **av** állományba.

A típus nélküli állományok írására és olvasására külön eljárások szolgálnak:

procedure BlockRead(**var** av: **file**; **var** pu, n: **word** [;**var** r:**word**]);

Az **av** állományból **n** blokk (rekord) olvasódik be **pu** első byte-jától kezdve. A ténylegesen átvitt teljes blokkok száma az opcionális **r** változóba kerül. Ha **r** nem szerepel, és az állományból nem lehet **n** blokkot

behozni, B/K hiba keletkezik. r segítségével ez a helyzet n és r összehasonlításával a program megszakadása nélkül felderíthető. Az összes átvitt byte-ok száma n*blokkméret (ami nem haladhatja meg a 64 kbyte-ot), ahol a blokk méretét az állomány megnyitásakor lehetett megadni, illetve – ennek hiányában – 128 byte. Az eljárás az állománypozíciót megfelelően módosítja.

procedure `BlockWrite(var av: file; var pu, n: word [;var r:word]);`
A **pu** változó első byte-jától kezdve **n** blokk (rekord) íródik ki az **av** állományba. A ténylegesen átvitt teljes blokkok száma az opcionális **r** változóba kerül. Ha **r** nem szerepel, és az állományba nem lehet **n** blokkot kivinni (mert betelt a lemez), B/K hiba keletkezik. **r** segítségével ez a helyzet **n** és **r** összehasonlításával a program megszakadása nélkül megoldható. Az összes átvitt byte-ok száma **n***blokkméret (ami nem haladhatja meg a 64 kbyte-ot), ahol a blokk méretét az állomány megnyitásakor lehetett megadni, illetve – ennek hiányában – 128 byte. Az eljárás az állománypozíciót megfelelően módosítja.

2.3.2. Programok vezérlése

A ritkán használatos Goto utasítás helyett egyes esetekben igen hasznos lehet valamelyik vezérlőeljárás.

procedure `Exit;`
Azonnali kilépés az aktuális blokkból. Eljárás vagy függvény esetén a hívóprogramba, a főprogram esetén az operációs rendszerbe, illetve a programot indító környezetbe.

procedure `Halt [(kod: word)];`
A program megszakítása és kilépés az operációs rendszerbe, illetve a programot indító környezetbe. Az opcionális paraméter, melynek alapfeltételezés szerinti értéke 0, a program kilépő kódjának megadására szolgál egy **word** típusú kifejezés formájában.

procedure `RunError [(kod: word)];`
A Halt-hoz hasonlóan megszakítja a programot, és egy futási idejű hibát is generál, melynek kódja az opcionális paraméterrel adható meg. Ennek elhagyása esetén a kód 0.

2.3.3. A dinamikus memória kezelése

A program betöltése után szabadon maradt teljes memória vagy annak egy része dinamikus tárként (heap) kezelhető az alábbi eljárásokkal és függvényekkel. A dinamikus memória minimális és maximális mérete a $M fordítóprogram-direktívával szabályozható.

procedure `Dispose(var m: pointer);`

Az **m** által hivatkozott dinamikus változó megszűnik, területe felszabadul, és **m** értéke definiálatlan lesz. **m** eredeti értékét a **New** eljárással vagy megfelelő értékadással kapta.

procedure `FreeMem(var m: pointer; h: word);`

Az **m** által hivatkozott, a **h** kifejezés által megadott számú byte-ot elfoglaló dinamikus változó megszűnik, területe felszabadul, és **m** értéke definiálatlan lesz. **m** eredeti értékét a **GetMem** eljárással vagy megfelelő értékadással kapta. A **GetMem**-ben és a **FreeMem**-ben szereplő méreteknek pontosan egyezniük kell.

procedure `Mark(var m: pointer);`

A dinamikus memória szabadhely-mutatójának aktuális értéke átmásolódik **m**-be. A tárolt érték a **Release** (l. később) eljárás paramétereként használható.

function `MaxAvail: longint;`

Értéke a dinamikus memória legnagyobb lefoglalható (összefüggő és szabad) blokkjának mérete.

function `MemAvail: longint;`

Értéke a dinamikus memória szabad blokkjai hosszának összege.

procedure `New(var m: pointer);`

Létrehozza az **m**^ nevű dinamikus változót.

procedure `Release(var m: pointer);`

A dinamikus memória szabadhely-mutatóját visszaállítja az **m** értékére. **m** előzőleg **Mark** (l. ott) eljárással kapott értéket. A dinamikus memória azon része, amelyet az utolsó **Mark(m)** hívás óta használtak fel, felszabadul.

2.3.4. Típuskonverzió

Néhány gyakran használt alaptípus közötti átalakítást végeznek a következő függvények:

function `Chr(b: byte): char;`

Értéke az ASCII karakterkészlet **b**-edik sorszámú eleme (a **b** kódú karakter).

function `Ord(o: <tetszőleges, megszámlálható típus>): longint;`

Értéke **o** sorszáma a paraméter típusán belül.

function `Round(x: real): longint;`

Értéke az **x** valós kifejezésből kerekítéssel képzett egész szám.

function Trunc(x **real**): **longint**;
Értéke az x valós kifejezésből kerekítés nélkül képzett egész szám.

2.3.5. Aritmetika

A Pascal aritmetikai függvényeinek nagy része a matematikában elemi függvényeknek nevezett csoporthoz tartozik. A függvények használatához szükséges matematikai ismereteket feltételezve, csak annyit jegyzünk meg, hogy a paraméterként megadható vagy eredményül kapható szögek értéke radiánban értendő.

function Abs(x: <tetsz. egész v. valós típus>):
<a paraméter típusa>;
Értéke az x valós vagy egész kifejezés abszolút értéke.

function ArcTan(x: **real**): **real**;
Értéke az x valós kifejezés arcus tangense.

function Cos(x: **real**): **real**;
Értéke az x valós kifejezés cosinusa.

function Exp(x: **real**): **real**;
Értéke az x valós kifejezéshez tartozó e^x érték.

function Frac(x: **real**): **real**;
Értéke az x valós kifejezés törtrésze.

function Int(x: **real**): **real**;
Értéke az x valós kifejezés egész része.

function Ln(x: **real**): **real**;
Értéke az x valós kifejezés természetes logaritmusa.

function Pi: **real**;
Értéke 3.14159265358979323 85.

function Sin(x: **real**): **real**;
Értéke az x valós kifejezés sinusa.

function Sqr(x: <tetsz. egész v. valós típus>):
<a paraméter típusa>;
Értéke az x egész vagy valós kifejezés négyzete.

function Sqrt(x: **real**): **real**;
Értéke az x valós kifejezés négyzetgyöke.

2.3.6. A megszámlálható típusok kezelése

A megszámlálható (ordinal) típusok egységes kezelését valósítják meg a következő eljárások és függvények. Az első (gyakran egyetlen) paraméter tetszőleges, megszámlálható típusú kifejezés, hacsak típusát külön meg nem adjuk.

procedure `Dec(var o [; n: longint]);`

Az **o** megszámlálható típusú változó értékét csökkenti **n**-nel (ha a második paraméter szerepel) vagy 1-gyel (ha a második paraméter nem szerepel).

procedure `Inc(var o [; n: longint]);`

Az **o** megszámlálható típusú változó értékét növeli **n**-nel (ha a második paraméter szerepel) vagy 1-gyel (ha a második paraméter nem szerepel).

function `Odd(n: longint): Boolean;`

Értéke **true**, ha az **n** egész típusú kifejzés értéke páratlan, különben **false**.

function `Pred(o): <o-val azonos típus>`

Értéke az **o** megszámlálható típusú kifejezés értékét a típusdeklaráció szerint megelőző érték.

function `Succ(o): <o-val azonos típus>`

Értéke az **o** megszámlálható típusú kifejezés értékét a típusdeklaráció szerint követő érték.

2.3.7. Karakterláncok kezelése

A karakterlánc (**string**) típusok jelentőségének egyik elismerése a kezelésükkel foglalkozó eljárások és függvények viszonylag nagy száma.

function `Concat(s1 [, s2, s3, ...sn]: string): string;`

Értéke a paraméterként adott egy vagy több karakterlánc típusú kifejezés konkatenációja. Hatása a + műveletével azonos.

function `Copy(s: string; i, n: integer): string;`

Értéke az **s** karakterlánc típusú kifejezés **i**-edik karakterével kezdődő, **n** hosszúságú karakterlánc. Az eredmény hossza **n**-nél kisebb is lehet, ha **s**-ben nincs elegendő karakter.

procedure `Delete(var s: string; i, n: integer);`

Az **s** karakterlánc típusú változó értékéből törlődik az **i**-edik karakterrel kezdődő, **n** hosszúságú karakterlánc. A törölt rész hossza **n**-nél kisebb is lehet, ha **s**-ben nincs elegendő karakter.

procedure `Insert(mit: string; var s: string; i: integer);`

Az **s** karakterláncba, annak **i**-edik pozíciójától kezdve beszúródik a **mit** karakterlánc típusú kifejezés.

function `Length(s: string): integer;`

Értéke az **s** karakterlánc típusú kifejezés hossza.

function `Pos(mit, s: string): byte;`

Ha a **mit** karakterlánc típusú kifejezés előfordul az **s** karakterlánc típusú kifejezésben, a függvény értéke az első előfordulás pozíciója, különben pedig 0.

procedure `Str(x [:meret [:tizedesek]]; var s: string);`

Az **x** egész vagy valós típusú kifejezést karakterlánccá alakítva **s**-ben tárolja. **meret** és **tizedesek** opcionális egész típusú kifejezések. Ha szerepelnek, **meret** az eredmény teljes hosszát, **tizedesek** pedig (valós **x** esetén) a tizedesjegyek számát írják elő. Hiányukban egész típusú **x** esetén az eredmény hossza **x** értékétől függ, valós **x** esetén pedig 17 lesz (exponenciális formájú számábrázolás mellett).

procedure `Val(s:string;var e:<egész v. valós>;var hiba:integer);`

Az **s** karakterlánc típusú kifejezést konvertálja **e** típusával (valamilyen egész vagy valós) megegyező számmá. Az eredményt **e**-ben kapjuk. Hibátlan konverzió esetén **hiba** értéke 0, hibás karakter előfordulásakor a konverzió megszakad, **hiba** tartalmazza a hibát okozó karakter pozícióját **s**-ben, **e** értéke pedig definiálatlan.

2.3.8. Mutató- és címkezelés

Az ún. **gépközeli programozás** támogatására szolgálnak a következő függvények. Céljuk, hogy a Turbo Pascal segítségével olyan feladatokat is megoldhassunk, amelyek általában csak assemblerrel vagy más alacsony szintű programozási eszközzel kezelhetők. A függvények alkalmazásához az Intel 808x processzor legalább nagyvonalú ismeretére van szükség.

function `Addr(a): pointer;`

Értéke a tetszőleges típusú (eljárás- vagy függvényazonosító is lehet) **a** változóhoz tartozó memóriacím. Eredménye azonos a @ operátoréval.

function `CSeg: word;`

Értéke a CS regiszter aktuális tartalma.

function `DSeg: word;`

Értéke a DS regiszter aktuális tartalma.

function `Ofs(a): word;`

Értéke a tetszőleges típusú (eljárás- vagy függvényazonosító is lehet) **a** változóhoz tartozó memóriacím szegmens-relatív része.

function Ptr(s, o: **word**): **pointer**;
Értéke az **s** szegmens és **o** szegmens-relatív címre hivatkozó, típus nélküli mutató.

function Seg(a): **word**;
Értéke a tetszőleges típusú (eljárás- vagy függvényazonosító is lehet) **a** változóhoz tartozó szegmenscím.

function SPtr: **word**;
Értéke az SP regiszter aktuális tartalma.

function SSeg: **word**;
Értéke az SS regiszter aktuális tartalma.

2.3.9. Vegyes célú eljárások és függvények

Az előző nyolc csoport egyikébe sem sorolható standard eljárásokat és függvényeket gyűjtöttük itt össze.

procedure FillChar(**var** x; n: **word**; c: <**megszámlálható típus**>);
Az **x** tetszőleges típusú változóhoz tartozó memóriaterület első **n** byteját a **c** megszámlálható típusú értékkel tölti fel.

function Hi(x: <**integer** vagy **word**>): **byte**;
Értéke az **x** egész típusú kifejezés értékének magasabb helyértékű byteja (felső byte).

function Lo(x: <**integer** vagy **word**>): **byte**;
Értéke az **x** egész típusú kifejezés értékének alacsonyabb helyértékű byte-ja (alsó byte).

procedure Move(**var** innen, oda; n: **word**);
n folytonosan elhelyezkedő byte-ot visz át az **innen** tetszőleges típusú változó memóriaterületének elejéről az **oda** tetszőleges típusú változó memóriaterületének elejére.

function ParamCount: **word**;
Értéke a programot hívó parancssorban adott paraméterek száma.

function ParamStr(n: **word**): **string**;
Értéke a programot hívó parancssorban adott **n**-edik paraméter (karakterlánc formájában).

function Random [(max: **word**)]: <**real** vagy **integer**>;
Értéke a **max** paraméter hiányában egy 0-nál nem kisebb, és 1-nél kisebb valós véletlen szám. Ha **max** is szerepel a hívásban, az eredmény egy véletlen egész lesz, amely legalább 0 és legfeljebb **max–1** értékű.

procedure Randomize;

A véletlenszám-generátor inicializálása a rendszerórából vett véletlen értékkel. A kezdőérték a **RandSeed** előre deklarált, **logint** típusú változóban tárolódik. Ide azonos kezdőértéket töltve, azonos véletlenszám sorozatok generálhatók.

function SizeOf(x): **word**;

Értéke az **x** paraméter által elfoglalt memória mérete. **x** tetszőleges változó neve vagy bármilyen típus azonosítója lehet.

function Swap(x: <**integer** vagy **word**>): <**integer** vagy **word**>;

Értéke az **x** egész kifejezésből úgy származik, hogy annak alsó és felső byte-ját felcseréljük.

function UpCase(k: **char**): **char**;

Értéke **k in** [**a..z**] esetén a megfelelő nagybetű, egyébként **k**.

3. Modulok

A standard Pascal születésekor még távolról sem volt természetes, hogy egy nyelvi rendszer a moduláris programozás támogatásáról is gondoskodjon. Ennek csírái az assemblerekből talán először a C nyelvbe kerültek át, ahol már a korai fordítóprogramok a teljes forrásprogramok helyett különálló vagy csoportokba szervezett függvényeket is elfogadtak.

A Pascal terjedésének is gátjává vált a modulkezelés hiánya, ezért már 1980 körül a Microsoft Pascalban megjelenik a **module** és **unit** fogalma. Ezeket fordítási egységnek nevezték, ami a programokhoz hasonlóan a forrásszöveg olyan darabja, melyet a fordítóprogram önállóan képes feldolgozni. Míg a **module** egy törzs nélküli program, a **unit** kettéválasztja a modul ún. **interface** (kapcsolati) és **implementation** (megvalósítási) részét, és ezzel lehetővé válik még meg nem valósított modulok fordítása, illetve egy kapcsolati részhez több megvalósítás elkészítése és kipróbálása.

A Turbo Pascal átvette és továbbfejlesztette a moduláris programozás alapjául szolgáló **unit** fogalmát. E fejezet tárgyát a Turbo Pascalban a 4.0-ás változattól kezdve használható **unit**-ok képezik, melyeket magyarul **modulnak** fogunk hívni. Az elnevezés nem zavaró, mert a Turbo Pascalban (eltérően a Microsoft Pascaltól) más modulfogalom nem szerepel.

3.1. Programok és modulok

Egy moduláris felépítésű Pascal program egy főprogramból és több modulból tevődik össze. A főprogram legfontosabb feladata az egyes modulokban lévő, részfeladatokat megoldó eljárások és függvények megfelelő sorrendben való hívása. Emellett a főprogram saját típusokat, adatszerkezeteket, eljárásokat stb. is tartalmazhat. A kapcsolatot a főprogram és az általa használt modulok között a közvetlenül a programfej után álló **uses** direktíva hozza létre, amely a **uses** lefoglalt szó után egyszerűen felsorolja – egymástól vesszővel elválasztva – a főprogram tevékenysége során felhasználandó modulok nevét. Ennek megfelelően a

```
uses menu, uzenet, hiba;
```

direktíva azt fejezi ki, hogy a program a fenti három modult használja, azaz olyan típusokra, konstansokra, eljárásokra, függvényekre és változókra is hivatkozhat, amelyek deklarációja ezek valamelyikében szerepel.

Egy program moduláris felépítését lényegében három körülmény indokolja:

* A feladat mérete, amely mind a fejlesztés, mind a karbantartás során csoportmunkát igényel, a modulok pedig egyben a feladat természetes felosztását is képviselik.
* A modulok önálló fordításával és javításával sok munka takarítható meg a tesztelés folyamán, hiszen csak a módosított modulokat kell újra fordítani.
* Egyes modulok olyan általános algoritmust valósítanak meg, amelyet más feladatoknál esetleg más fejlesztők is alkalmazhatnak. Egy gazdag **modulkönyvtár** számottevő mennyiségű programozást pótolhat.

Az első körülmény független a használt programozási nyelvtől, a másik kettő súlya viszont annál nagyobb, minél több támogatást kap az alkalmazott nyelvi rendszertől.

A modulok **önálló fordítását** a Turbo Pascal úgy támogatja, hogy a fordítóprogram a forrásmodulból speciális (TPU) kiterjesztésű tárgymodult készít, ami majd az őt használó program fordításakor automatikusan beépül a tárgykódba. Ennek érdekében a programoktól való megkülönböztetésül a modulok a **unit** lefoglalt szóval kezdődnek, amit a modul neve követ. Egy modul természetesen további modulokat is használhat saját **uses** direktívájában, az önálló fordításhoz e moduloknak forrás szinten vagy TPU állomány formájában rendelkezésre kell állni. A modulok ilyen jellegű hierarchiája tetszőleges mélységű lehet, ezért jól alkalmazkodhat bármely feladat szerkezetéhez.

A fordítóprogram számára kijelölhető egy főprogram; a fordítás pedig végrehajtható olyan üzemmódban is, hogy automatikusan csak az utolsó fordítás óta **módosított** modulok fordítása **ismétlődjön** a teljes tárgyprogram összeállítása előtt (MAKE opció a Turbo Pascal COMPILE menüjében).

A **modulkönyvtár** kialakításához, modulok más programozók számára való átengedésének gyakorlati támogatásában lényeges szempont, hogy a modul a forrásszöveg átadása nélkül, TPU állomány formájában más programokban is felhasználható. Ezen túlmenően, a modulok szerkezete lehetővé teszi, hogy a szerző a modul egyes típusait, konstansait, változóit, algoritmusait vagy akár az általa használt további modulokat elrejtse a csupán használati jogot élvező nyilvánosság elől. Ennek megfelelően a Turbo Pascal modulok a modul nevét megadó fejen kívül további három részből állnak, melyek között üresek is lehetnek. A részek elnevezésénél átvették a Microsoft Pascal terminológiáját, a fogalmak azonban jórészt más tartalmat kaptak.

A modulok első része, az **interface** (kapcsolati rész) a nyilvános, azaz a modult használó, minden programban elérhető típusok, konstansok, eljárások és változók deklarációját tartalmazza. Az eljárásdeklarációkat itt nem teljes egészében kell megadni, a **forward** deklarációhoz hasonlóan az eljárás (vagy függvény) feje elegendő. A modult használó program szempontjából a kapcsolati részben szereplő fogalmak úgy tekinthetők, mint amelyeket a

programot körülvevő blokkban deklaráltak. Ebben a részben természetesen uses direktívák is szerepelhetnek a felhasznált további modulok megnevezésére.

A modulok második része, az **implementation** (megvalósítási rész) a modul „magánügyeit" tartalmazza. Itt is uses direktívák, illetve deklarációk állhatnak, az itt deklarált típusok, változók stb. azonban csak a modulon belül hozzáférhetők, a modult használó programban vagy modulokban nem. Eljárások és függvények esetén a megvalósítási részbe kerülnek egyrészt a kapcsolati részben deklarált ejárások törzsei (itt elegendő egy rövidett fej használata, amely csak a **procedure** vagy **function** szóból és az eljárás nevéből áll, a paraméterek felsorolása elhagyható), másrészt az ide tartozó, most már teljes deklarációk. A kapcsolati részben szereplő teljes eljárásfejek átvétele is megengedett, de ebben az esetben a két paraméterlistának pontosan egyeznie kell.

A modulok utolsó része a **törzs**, amely utasítások tetszőleges, **end**-del lezárt sorozata, ami a modult használó program indulásakor, annak első utasítása előtt hajtódik végre. Több modul használata esetén a törzsek végrehajtása a uses direktívában való előfordulás sorrendjében történik. A modultörzs üres is lehet, ebben az esetben az **end** lefoglalt szóból áll, amit egy pont követ.

A modulok használatát bemutató egyszerű példaként tekintsük a következő algoritmust:

1. Vegyünk egy tetszőleges egész számot, ami egy sorozat első tagja.
2. Ha a sorozat utolsó tagja páros, az új tag legyen ennek a fele, különben az új tagot úgy kapjuk, hogy a páratlan értéket annak nagyobbik felével növeljük (5 után pl. 8 következik).
3. Ismételjük az előző pontban leírtakat addig, ameddig a sorozat valamelyik tagja 1 nem lesz.

A tapasztalat azt mutatja, hogy bármely értékből indulva, előbb-utóbb eljutunk az 1-hez.

Az alábbi programban a páros és páratlan eset kezelését külön modulokban (**le**, ill. **fel**) oldjuk meg.

```
program fel_le;
uses crt, felu, leu;
var n: longint;
begin clrscr;
      write('Induló érték: '); readln(n);
      while n>1 do
            if odd(n)
            then fel(n)
            else le(n)
end.
```

```
unit felu;
interface
procedure fel(var i: longint);
implementation
procedure fel;
begin i:=i+(i+1) div 2;
      writeln(i)
end;
end.

unit leu;
interface
procedure le(var i: longint);
implementation
procedure le;
begin i:=i div 2;
      writeln(i)
end;
end.
```

A modulok törzse üres, a megvalósítási részben is csak az egyetlen nyilvános eljárás deklarációja található.

A hierarchikusan felépített moduláris rendszerekben előfordulhat, hogy a főprogram vagy egy modul **közvetve** használ egy másik modult. Ha az A modul használja B-t és B használja C-t, akkor A közvetve használja C-t. A közvetett és közvetlen használat a fordítóprogram szempontjából egyenértékű, a fenti példában A csak akkor fordítható le, ha B és C egyaránt rendelkezésre áll.

A modulok módosításakor nem mindegy, hogy a kapcsolati részen vagy a megvalósításon, illetve a törzsön változtatunk. Ha egy modul **kapcsolati** része módosul, minden ezt használó programot vagy modult **újra kell fordítani**. Ha a változás csak a megvalósítási részt, illetve a törzset érinti, a modult használó programot és modulokat **nem kell** újra fordítani. A fordítási rendszer folyamatosan számon tartja a programhoz tartozó modulok kapcsolati részének változását, és megfelelő üzemmódban használva, elvégzi az automatikus újrafordítást.

A **uses** direktíva kötetlen alkalmazása, nemcsak a közvetett, hanem az ún. **ciklikus** modulhasználatot is lehetővé teszi. A ciklikus használat legegyszerűbb esete, ha két modul **kölcsönösen** használja egymást, de létrejöhet egy közvetett használati lánc során is (pl. A használja B-t, B használja C-t, C használja A-t).

A ciklikus modulhasználat a fordítóprogramot szokatlan feladat elé állítja: két, egymást kölcsönösen (közvetlenül vagy közvetve) használó modul közül egyik fordítását sem tudja **befejezni** a másik fordítása nélkül. A

megoldást az egymásra utaló uses direktívának a modulok megvalósítási részében való elrejtése adja. Ha két modul csak megvalósítási részében hivatkozik egymásra, a fordító mindkettő kapcsolati részét elő tudja állítani, és elfogad egy nem teljesen lefordított modulra való hivatkozást a megvalósítási részben. A ciklikus modulhasználat bemutatására az előző programot valamelyest átalakítottuk, az algoritmus 2. lépésében szereplő két lehetőségnek egy-egy modult feleltetünk meg, amelyek egymásra hivatkoznak, amint ez a következő programból látható.

```
program fel_le;
uses crt,felu,leu;
begin clrscr; write('Induló érték: ');
      readln(n);
      if odd(n)
      then fel(n)
      else le(n)
end.

unit felu;
interface
var n: longint;
procedure fel(var i: longint);
implementation
uses leu;
procedure fel;
begin repeat i:=i+(i+1) div 2;
              writeln(i);
      until not odd(i);
      le(i)
end;
end.

unit leu;
interface
procedure le(var i: longint);
implementation
uses felu;
procedure le(var i: longint);
begin repeat i:=i div 2;
              writeln(i);
      until odd(i);
      if i>1
      then fel(i)
end;
end.
```

Figyeljük meg, hogy a főprogram csak a folyamat elindítására szolgál, ezután a két modul egymás között eldönti, hogy melyik eljárást kell hívni.

3.2. Standard modulok

A Turbo Pascal hatékony és kényelmes használatához nélkülözhetetlenek a rendszerhez tartozó, előre megírt és lefordított, ezért a standard eljárásokhoz hasonlóan szabadon felhasználható ún. **standard modulok.**

A standard modulok a TURBO.TPL könyvtárban találhatók, innen a tényleges igények szerint épülnek be a kapcsolatszerkesztés folyamán a felhasználó programjába, illetve moduljaiba, ezért alkalmazásuk memóriatakarékos is. Egy-egy alkalmazási rendszerbe csak a ténylegesen használt modulok szerkesztődnek be; ha egy program nem használ nyomtatót vagy grafikát, az ezt megvalósító nyelvi elemek nem növelik feleslegesen a rendszer memóriaigényét.

A standard modulok – a felhasználó saját moduljaihoz hasonlóan – kapcsolati részükben típusok, konstansok, változók, eljárások és függvények deklarációját tartalmazzák. Ezek egy részére a programozás során gyakran kell hivatkozni, vannak azonban olyan nyelvi elemek is közöttük, melyekre csak speciális alkalmazások esetén, ritkán van szükség. Ezért a standard modulok bemutatásakor elsősorban az eljárásokra és függvényekre szorítkozunk, a típusok, konstansok és változók közül nem térünk ki a nagyon különleges szerepűekre, mivel ezek használatához a Turbo Pascal eredeti kézikönyve úgysem nélkülözhető.

3.2.1. A SYSTEM modul

Ez a modul tulajdonképpen a Pascal programok futtatásához szükséges alapkönyvtár. A nyelv különböző szolgáltatásait megvalósító alsó szintű rutinokat tartalmazza, (pl. állományok kezelése, karakterlánc-műveletek, lebegőpontos aritmetika, overlay-kezelés, dinamikus memória vezérlése stb.). A SYSTEM modult **minden program és modul** automatikusan használja, ezért nem kell **uses** direktívában szerepelnie.

A modulhoz tartoznak a 2. fejezetben tárgyalt standard eljárások és függvények, valamint néhány konstans és változó, amelyek elsősorban a dinamikus memória, valamint a megszakítások kezelésével kapcsolatosak, jelentőségük még felsorolásukat sem indokolja.

3.2.2. A PRINTER modul

A sornyomtató használatát kényelmessé tevő, kis modul. Tulajdonképpen csak az **Lst** nevű **text** állomány deklarációját és az LPT1 külső egységhez való rendelését tartalmazza, valamint gondoskodik annak lezárásáról is.

A modul használatakor a **write(Lst,...);** és a **writeln(Lst,...);** utasítások a nyomtatóra írnak. Használatával megtakarítható az állomány deklarációja, hozzárendelése, megnyitása és lezárása.

3.2.3. A DOS modul

Az operációs rendszer szolgáltatásaival és az alacsony szintű állománykezeléssel kapcsolatos hasznos eljárásokat és függvényeket tartalmaz.

A modulhoz tartozó típusok a regiszterekkel, a dátummal, adott tulajdonságú állományok keresésével és az állományneveken végzett műveletekkel kapcsolatosak.

A Pascal állományoknak a DOS számára való leírásához a következő rekordtípusok használatosak:

```
type FileRec = record    {Közönséges és típus nélküli állományok}
                    Handle, Mode, RecSize: word;
                    Private: array[1..26] of byte;
                    UserData: array[1..16] of byte;
                    Name: array[0..79] of char;
               end;

     { text típusú állományok}

     TextBuf = array[0..127] of char;
     TextRec = record Handle, Mode, BufSize,
                      Private, BufPos, BufEnd: word;
                      BufPtr: ^TextBuf;
                      OpenFunc, InOutFunc, FlushFunc,
                      CloseFunc: pointer;
                      UserData: array[1..16] of byte;
                      Name: array[0..79] of char;
                      Buffer: TextBuf;
               end;
```

A regiszter típus olyan változók deklarációját teszi lehetővé, amelyek az MS-DOS hívásokhoz szükséges adatokat közvetítik:

```
type Registers = record
                      case integer of
                           0:(AX,BX,CX,DX,BP,SI,DI,DS,ES,
                             Flags: word);
                           1:(AL,AH,BL,BH,CL,CH,DL,DH:byte);
                      end;
```

Figyeljük meg, hogy a 16 bites regiszterek 8 bites részei is használhatók.

A dátum-idő típus a rendszeridőt és dátumot kezelő eljárások és függvények paramétereként használatos:

```
type DateTime = record Year, Month, Day, Hour,
                       Min, Sec: integer;
                end;
```

A kereső típus egy olyan rekord, amelyet a könyvtárban különböző tulajdonságú állományokat kereső eljárások használnak:

```
type SearchRec = record Fill: array[1..21] of byte;
                        Attr: byte;
                        Time: logint;
                        Size: logint;
                        Name: string[12];
                 end;
```

Végül az állományneveket szétvágó eljárás használja a következő **string** típusokat:

```
type  DirStr  = string[67];
      NameStr = string[8];
      ExtStr  = string[4];
```

A modulban deklarált konstansok a következők:
A **Flags** regiszter bitjeinek vizsgálatára:

```
const FCarry     = $0001;
      FParity    = $0004;
      FAuxiliary = $0010;
      FZero      = $0040;
      FSign      = $0080;
      FOverflow  = $0800;
```

Az állománymódoknak megfelelő konstansok:

```
const fmClosed  = $D7B0;
      fmInput   = $D7B1;
      fmOutput  = $D7B2;
      fmInOut   = $D7B3;
```

Az állományok jellemzőit a következő konstansok képviselik:

```
const ReadOnly  = $01;
      Hidden    = $02;
      SysFile   = $04;
      VolumeId  = $08;
      Directory = $10;
      Archive   = $20;
      AnyFile   = $3F;
```

A DOS modul eljárásai az esetleges hibákat egy integer típusú változó, a **DosError** segítségével jelzik. A 0 érték hibátlan műveletet jelez, a lehetséges hibakódok pedig a következők:

2: Az állomány nem található
3: Az állományhoz vezető út nem található
5: Az állományhoz való hozzáférés jogtalan
6: Az állomány leírása szabálytalan
8: Memóriahiány
10: Szabálytalan környezet
11: Szabálytalan forma
18: Nincs több állomány

A modul eljárásait és függvényeit tevékenységük szerint csoportosítjuk.

Megszakításkezelés

procedure `GetIntVec(MSzam: byte; var Vektor: pointer);`
Az **MSzam** index által meghatározott megszakítási vektorban tárolt címet adja **Vektor**-ban.

procedure `Intr(MSzam: byte; var Reg: Registers);`
Végrehajtja az **MSzam** indexű szofvermegszakítást. A megszakítás előtt a regisztereket **Reg**-ből feltölti, majd visszatérés előtt tartalmukat **Reg**-be visszaírja.

procedure `MsDos(var Reg: Registers);`
Hatása azonos a 21-es indexű **Intr** hívásával.

procedure `SetIntVec(Mszam: byte; Vektor: pointer);`
Adott indexű megszakításhoz adott címet (a megszakítást kezelő eljárás címét) rendel.

Dátum- és időkezelés

procedure `GetDate(var Ev, Ho, Nap, Napsorszam: word);`
A rendszerben lévő dátumot adja.
Értékhatárok:

```
Ev: 1980..2099;
Ho: 1..12;
Nap: 1..31;
Napsorszam: 0..6  (0=Vasárnap).
```

procedure `GetFTime(var a; var Ido: longint);`
Az **a** paraméter egy tetszőleges, **megnyitott** állomány azonosítója. Eredményül az állomány utolsó írásának dátumát és idejét kapjuk az **Ido**-ben, tömörített formában.

procedure GetTime(**var** Ora, Perc, Mp, Mp100: **word**);
Az aktuális rendszeridőt adja, **Ora** értéke 0..23 lehet.

procedure PackTime(**var** DI: DateTime; **var** Ido: **longint**);
A **DI DateTime** típusú rekordot 4 byte-os tömörített alakra hozza, amit a **SetFTime** eljárás (l. később) használ.

procedure SetDate(Ev, Ho, Nap: **word**);
Beállítja a rendszerdátumot. A paraméterek érvényes értékei a következők:

```
Ev:  1908..2099;
Ho:  1..12;
Nap: 1..31.
```

Érvénytelen paraméter használata esetén az eljárás hatástalan.

procedure SetFTime(**var** a; Ido: **longint**);
Az **a** paraméter egy tetszőleges, **megnyitott** állomány azonosítója. Az eljárás az **Ido** paramétert beírja az állomány utolsó módosításának dátumát és idejét őrző helyre. Az **Ido** értéke a PackTime eljárással (l. előbb) állítható elő.

procedure SetTime(Ora, Perc, Mp, Mp100: **word**);
Beállítja a rendszeridőt. A paraméterek érvényes értékei a következők:

```
Ora:  0..23;
Perc: 0..59;
Mp:   0..59;
Mp100: 0..99.
```

Érvénytelen paraméter használata esetén az eljárás hatástalan.

procedure UnpackTime(Ido: **longint**; DI: DateTime);
A 4 byte-os **Ido** paraméter értékét **DateTime** típusú rekorddá alakítja (a PackTime inverze).

Mágneslemez-vizsgálat

function DiskFree(Egyseg: **word**): **longint**;
Értéke a paraméterként adott lemezegységen lévő szabad byte-ok száma. Az egységeket a 0 (aktuális egység), 1 (A:), 2 (B:) stb. számok azonosítják. Érvénytelen egységszám esetén a függvény értéke −1.

function DiskSize(Egyseg: **word**): **longint**;
Értéke a paraméterként adott lemezegységen lévő összes byte száma. Az egységeket a 0 (aktuális egység), 1 (A:), 2 (B:) stb. számok azonosítják. Érvénytelen egységszám esetén a függvény értéke −1.

function FExpand(Ut: PathStr): PathStr;

Az **Ut** paraméter által meghatározott állomány nevét teljes, a lemezegység nevével kezdődő, nagybetűs neveket tartalmazó névvé alakítja.

procedure FindFirst(Ut: **string**; Attr: **word**; **var** K: SearchRec);

Az adott (vagy az aktuális) könyvtárban az **Attr** paraméterrel jellemzett első állományt keresi (több jellemző egyidejűleg a kódok összegével fejezhető ki). Az attribútumkódok helyett a **ReadOnly**, **Hidden** stb. konstansok is használhatók. Az **Ut** paraméter egy könyvtármaszkot (pl. 'A:*.PAS') tartalmazhat. Az eredményt a **K SearchRec** típusú (l. előbb) paraméter hordozza. A **DosError** által jelzett lehetséges hibák: 2 (A könyvtár nem található) és 18 (Nincs több állomány).

procedure FindNext(**var** K: SearchRec);

Csak egy korábbi **FindFirst** hívás után használható, ekkor az ott leírt tulajdonságú, következő állomány adatait adja **K**-ban. A **DosError** által jelzett lehetséges hiba: 18 (Nincs több állomány).

function FSearch(Ut: PathStr; DL: **string**): PathStr;

Az **Ut** paraméter által meghatározott állományt keresi a **DL** karakterlánc formájában adott könyvtárakban. Értéke valamelyik könyvtár és az állomány nevének egyesítése vagy üres karaktarlánc, ha a keresett állomány egyik megadott könyvtárban sem található. A keresés az aktuális lemezegység aktuális könyvtárában kezdődik. **DL**-ben a könyvtárneveket pontosvesszővel kell elválasztani (mint az MS–DOS PATH parancsában).

procedure FSplit(Ut: PathStr; **var** D: DirStr; **var** N: NameStr;
var K: ExtStr);

Az **Ut** paraméter által meghatározott állomány nevét három komponensre bontja:

D: Lemezegység és könyvtár;
N: Alapnév;
K: Kiterjesztés.

procedure GetFAttr(**var** a; **var** Attr: **word**);

Az **a** paraméter egy kijelölt, de meg **nem nyitott** tetszőleges állomány azonosítója. Eredményül az **Attr** paraméterben az állomány jellemzőit (ReadOnly, Hidden stb.) kapjuk egy kombinált kód formájában. A **DosError** által jelzett lehetséges hibák: 3 (Az állományhoz vezető út nem található) és 5 (Az állományhoz való hozzáférés jogtalan).

procedure SetFAttr(**var** a; **var** Attr: **word**);

Az **a** paraméter egy kijelölt, de meg **nem nyitott** tetszőleges állomány azonosítója. Az állomány megkapja az **Attr** paraméter által kijelölt jellemzőket (ReadOnly, Hidden stb.). A **DosError** által jelzett lehetséges

hibák: 3 (Az állományhoz vezető út nem található) és 5 (Az állományhoz való hozzáférés jogtalan).

Folyamatvezérlés

function `DosExitCode: word;`

Értéke egy alfolyamat kilépési kódja. Az alsó byte-ra a befejeződő folyamat által küldött érték kerül, a felső byte-on 0 van normális befejezés és 1 van CTRL–C-vel való megszakítás esetén.

procedure `Exec(Ut, Parametersor: string);`

Ha kell betölti és alfolyamatként végrehajtja az **Ut** által kijelölt programot az adott paramétersorral. A memóriakiosztás nem változik az eljárás során, ezért az **Exec** hívást tartalmazó programok fordításánál célszerű maximális méretű dinamikus memóriát lefoglalni ($M direktíva). A DosErrorban a 2, 8, 10 és 11-es hibakódok jelenhetnek meg.

procedure `Keep(Exkod: word);`

Leállítja és a memóriában tartja a programot. Az **Exkod** értéke a **Halt** standard eljárásnak adódik át. Az ilyen programok gyűjtőnevének angol rövidítése TSR (Terminate Stay Resident), kezelésük az MS–DOS idevágó tulajdonságainak részletes ismeretét igényli. A dinamikus memóriát mindenesetre célszerű minél nagyobbra választani ($M direktíva).

procedure `SwapVectors;`

Kicseréli a **System** modulban lévő, a megszakításfeldolgozó rutinok címeit tartalmazó vektorokat az aktuális vektorokkal. Tipikus alkalmazása az **Exec** eljárás (l. előbb) előtt, illetve után azt biztosítja, hogy az eredeti és az új folyamat ne ugyanazokat a megszakításkezelő eljárásokat használja.

Környezeti információk

function `EnvCount: integer;`

Értéke a DOS környezeti változóinak száma. Ezek „Változó=Érték” alakú karakterláncok (ide tartozik pl. a PATH és a PROMPT is).

function `EnvStr(Index: integer): string;`

Értéke a DOS adott indexű környezeti karakterlánca (pl. PATH=C:\; C:\DOS), amely „Változó=Érték” alakú. Ha **Index** értéke 1-nél kisebb vagy **EnvCount**-nál nagyobb, az eredmény üres karakterlánc.

function `GetEnv(KV: string): string;`

Értéke a **KV** nevű környezeti változóhoz tartozó vagy az üres karakterlánc (ha az adott nevű környezeti változó nem definiált).

Vegyes eljárások és függvények

function `DosVersion`: **word**;
Értéke a DOS változatszáma. Az alsó byte-on a pont előtti, a felső byte-on a pont utáni rész jelenik meg.

procedure `GetCBreak`(**var** `CBR`: **Boolean**);
Az MS–DOS CTRL–BREAK figyelésére vonatkozó állapotjelzőjének értékét adja **CBR**-ben. (**false** esetén a DOS csak B/K műveleteknél, **true** esetén minden rendszerhívásnál figyeli a CTRL–BREAK billentyűt).

procedure `GetVerify`(`var Ell`: **Boolean**);
Az MS–DOS ellenőrzésjelzőjének értékét adja **Ell**-ben (**false** esetén a mágneslemezes átvitelek ellenőrzés nélkül, **true** esetén ellenőrzéssel történnek).

procedure `SetCBreak`(`CBR`: **Boolean**);
CBR alapján beállítja az MS–DOS CTRL–BREAK figyelésére vonatkozó állapotjelzőjének értékét (**false** esetén a DOS csak B/K műveleteknél, **true** esetén minden rendszerhívásnál figyeli a CTRL–BREAK billentyűt.)

procedure `SetVerify`(`Ell`: **Boolean**);
Az MS–DOS ellenőrzésjelzőjének értékét állítja be **Ell** alapján. (**false** esetén a mágneslemezes átvitelek ellenőrzés nélkül, **true** esetén ellenőrzéssel történnek.)

3.2.4. A CRT modul

A modul a képernyő és billentyűzet használatát támogató eljárásokat és függvényeket tartalmaz. Idetartozik a szöveges képernyőmódok beállítása, a bővített billentyűkódok, ablakok, valamint a szövegek színének kezelése.

A modul törzsében az **Input** és **Output** standard **text** típusú állományok a Pascal CRT nevű külső egységéhez rendelődnek az eredeti standard DOS állományok helyett.

Az ablakkezelés alapelve, hogy az egyszer létrehozott ablak a teljes képernyőt helyettesíti, az ablakon kívüli részek nem érhetők el. A képernyőkoordináták az aktuális ablak bal felső sarkához viszonyítva értendők, melynek koordinátája (1, 1). A kezdő ablak a teljes képernyő.

A modulban nyilvános típusdeklaráció nem szerepel, viszont többféle konstans használatát teszi lehetővé.

A képernyő módok (video mód) konstansai:

```
const BW40= 0;          {40x25 F/F színes adapterrel}
      BW80=2;           {80x25 F/F színes adapterrel}
      Mono=7;           {80x25 F/F egyszínű adapterrel}
```

```
C040=1;          {40x25 színes}
C080=3;          {80x25 színes}
Font8x8= 256;    {EGA/VGA 43,illetve 50 sor}
C40=C040;        {Kompatibilitás a 3.0-ás változattal}
C80=C080;        {Kompatibilitás a 3.0-ás változattal}
```

A szöveg és a háttér színeinek konstansai:

```
const Black=0;           {Fekete}
      Blue=1;            {Kék}
      Green=2;           {Zöld}
      Cyan=3;            {Türkiz}
      Red=4;             {Piros}
      Magenta=5;         {Lila}
      Brown=6;           {Barna}
      LightGray=7;       {Világosszürke}
      DarkGray=8;        {Sötétszürke}
      LightBlue=9;       {Világoskék}
      LightGreen=10;     {Világoszöld}
      LightCyan=11;      {Világostürkiz}
      LightRed=12;       {Rózsaszín}
      LightMagenta=13;   {Világoslila}
      Yellow=14;         {Sárga}
      White=15;          {Fehér}
      Blink=128;         {Villogó}
```

A modul nyilvános változói általában különböző kapcsolók szerepét játsszák, amelyek közvetlenül (értékadással) vagy eljárások segítségével állíthatók:

var CheckBreak: **Boolean**;

A CTRL–BREAK billentyű figyelését kapcsolja be, illetve ki. Ha **true**, a CTRL–BREAK leütése a következő képernyőre íráskor megszakítja a programot, **false** esetén hatástalan. Kezdőértéke **true**.

var CheckEOF: **Boolean**;

Az állomány vége (CTRL–Z) karakter generálásának kapcsolója. Ha **true**, a képernyőhöz rendelt állomány olvasása megszakítható a CTRL–Z leütésével (mert egy állományvége karakter generálódik), ha **false**, a CTRL–Z hatástalan. Alapértéke **false**.

var CheckSnow: **Boolean**;

CGA képernyőadaptereknél bizonyos körülmények között a képen „hangyásodás" jelentkezhet, ami a videomemóriába való közvetlen írással kiküszöbölhető. Színes szövegmód beállításakor a változó értéke automatikusan **true** lesz. Ha a CGA adapter jó minőségű, átállításával a képernyőre írás sebessége növelhető (az átállítást a **TextMode** eljárás minden hívása után ismételni kell). Ha **DirectVideo** (l. később) értéke **false**, a **CheckSnow** hatástalan.

var DirectVideo: **Boolean**;

A **write** és **writeln** eljárásokban ki és be kapcsolja a közvetlen videomemóriába írást. **true** érték esetén a karakterek a BIOS-t megkerülve, közvetlenül a videomemóriába íródnak; ha **false**-ra állítjuk, a karaktereket a BIOS dolgozza fel, ami lényegesen lassúbb folyamat. Alapértéke **true**, ami minden **TextMode** (l. később) hívás után visszaállítódik.

var LastMode: **word**;

Itt tárolódik a **TextMode** eljárás minden hívásakor az aktuális képernyőmód.

var TextAttr: **byte**;

A szövegjellemzőket általában a **TextColor** és a **TextBackground** (l. később) eljárásokkal állítják be. A jellemzők azonban közvetlenül is állíthatók a **TextAttr**-nek való értékadással. A szöveg színkódját a 0..3, a háttér színkódját a 4..6. bitekre kell írni. (16 szöveg- és 8 háttérszín használható). A 7. biten lévő 1 a szöveg villogtatását váltja ki.
Ennek megfelelően az értékadás a következő:

TextAttr:=Betűszín+16*Háttérszín {Villogás nélkül}
TextAttr:=Betűszín+16*Háttérszín+Blink, {Villogással}

ahol Háttérszín≤7.

var WindMin, WindMax: **word**;

Az aktuális ablak bal felső és jobb alsó sarkának koordinátáit tárolják. Értéküket a Window eljárás (l. később) állítja. Az X koordináta az alsó, az Y koordináta a felső byte-on helyezkedik el.

A modulban deklarált eljárások és függvények a következők:

procedure AssignCrt(a: **text**);

Az **a** szövegállományt a képernyőhöz rendeli.

procedure ClrEol;

A kurzortól a sor végéig törli a képernyőt, a kurzor helyben marad. Az eljárás ablak-relatív.

procedure ClrScr;

A teljes képernyőt törli, a kurzor a bal felső sarokba kerül. Az eljárás ablak-relatív.

procedure Delay(ms: **word**);

A programot kb. **ms** ezredmásodpercig késlelteti.

procedure DelLine

Törli a kurzort tartalmazó sort. A törlés alatti sorok eggyel feljebb lépnek. Az eljárás ablak-relatív.

procedure GotoXY(x, y: **byte**);

A kurzort az aktuális ablak koordináta-rendszere szerinti (**x**,**y**) pontra viszi. Ha valamelyik koordináta bármely okból is érvénytelen (pl. kívül esik az aktuális ablakon), az eljárás hatástalan.

procedure HighVideo;

A szöveg színének intenzitását növeli, a **TextAttr** változóban a 0..7-es színkódot 8..15-re változtatja.

procedure InsLine;

Egy üres sort létesít a kurzor pozíciójában. A kurzor alatti sorok eggyel lejjebb lépnek. Az eljárás ablak-relatív.

function KeyPressed: **Boolean**;

Értéke **true**, ha egy billentyűt leütöttek, különben **false**. A leütött karakter(ek) a billentyűzet pufferjében marad(nak). A függvény nem érzékeli az ún. váltóbillentyűket (SHIFT, ALT stb.).

procedure LowVideo;

A szöveg színének intenzitását csökkenti, a **TextAttr** változóban a 8..15-ös színkódot 0..7-re változtatja.

procedure NormVideo;

Visszaállítja a program indulásánál érvényben levő szövegjellemzőket a **TextAttr** változóban.

procedure NoSound;

Kikapcsolja a gép hangszóróját.

function ReadKey: **char**;

Egy karaktert olvas a billentyűzetről. A karakter a képernyőn nem jelenik meg. Ha a függvény hívása előtt a **KeyPressed** értéke **false**, azaz billentyűt nem ütöttek le, a program megvárja egy billentyű leütését.

procedure Sound(HZ: **word**);

A gép hangszóróját HZ rezgésszámú hanggal indítja. A hangszórót kikapcsolni a **NoSound** (l. előbb) eljárással lehet.

procedure TextBackground(Szin: **byte**);

Beállítja a háttér színét. **Szin** értéke, amely 0-tól 7-ig terjedhet, beíródik a **TextAttr** változó 4..6. bitjeire.

procedure TextColor(Szin: **byte**);

Beállítja a szöveg színét. **Szin** értéke, amely 0-tól 15-ig terjedhet, beíródik a **TextAttr** változó 0..3. bitjeire.

procedure TextMode(Mod: **word**);

Beállítja **Mod** alapján az aktuális képernyőmódot. Ha a paraméter értéke nem egyezik egyik – a modulban deklarált konstansok közt felsorolt – értékkel sem, C80-as mód állítódik be. Az eljárás hívásakor az aktuális ablak a teljes képernyő lesz, az eredeti képernyőmód tárolódik **LastMode**-ban, visszaállnak a program indulásakor érvényben levő szövegjellemzők, **DirectVideo** és – színes mód választása esetén – **CheckSnow** értéke is true lesz.

function WhereX: byte;

Értéke az aktuális kurzorpozíció ablak-relatív X koordinátája.

function WhereY: byte;

Értéke az aktuális kurzorpozíció ablak-relatív Y koordinátája.

procedure Window(x1, y1, x2, y2: byte);

Szövegablakot definiál a képernyőn. (**x1**,**y1**) a bal felső, (**x2**,**y2**) a jobb alsó sarok koordinátái. Ha ezek bármelyike érvénytelen (pl. a képernyőn kívülre esik), az eljárás hatástalan. Az új ablak megnyítása után a kurzor annak bal felső sarkába (1,1) kerül, **WindMin**-be és **WindMax**-ba beíródnak az ablak paraméterei.

3.2.5. A GRAPH modul

A modul tulajdonképpen különféle grafikus tevékenységet végző szubrutinok gyűjteménye, amelyek eljárások és függvények formájában öltöttek testet. A grafikai szoftver alapkérdése a használt grafikus kártya (adapter). Ebben a tekintetben a Turbo Pascal igyekszik teljes választékot nyújtani a következő adapterek támogatásával:

* CGA
* MCGA
* EGA
* VGA
* Hercules
* AT&T 400 soros
* 3270 PC
* IBM–8514

A támogatás azt jelenti, hogy e grafikus adapterekhez fizikai szintű kezelőrutinok (ún. **driver**-ek) állnak rendelkezésre. A grafikus kezelőrutinok az adapter nevéből képzett, BGI kiterjesztésű állományokban helyezkednek el:

* CGA.BGI	(CGA és MCGA kezelés)
* EGAVGA.BGI	(EGA és VGA kezelés)
* HERC.BGI	(Hercules kezelés)
* ATT.BGI	(AT&T 6300 kezelés)
* PC3270.BGI	(IBM 3270 PC kezelés)
* IBM8514.BGI	(IBM–8514 kezelés)

Mint látható, néhány adapter kezelése közös állományba került. A BGI állományokra a megfelelő grafikus adaptert használó program futtatásához van szükség.

A Graph modul által megvalósított grafikus alrendszer nem karakteres (szöveges), hanem ún. **grafikus** módban működik, amelyben a képernyőt X (vízszintes) és Y (függőleges) koordinátájukkal leírható képpontokra, ún. **pixelekre** osztják. A kezdőpont (0,0) a képernyő bal felső sarka. A képpontok soronkénti és oszloponkénti száma a grafikus adaptertől függ (például egyszerű CGA esetén $0 \leq X \leq 319$ és $0 \leq Y \leq 199$). A grafikus képernyőn kurzor nincs, szerepét az aktuális képpont (AK) veszi át, ami azonban általában nem látható.

Mivel szöveges és grafikus mód egy képernyőn egyszerre nem használható, gondoskodni kellett arról, hogy grafikus módban is írhassunk szöveget a képernyőre. Ezt a megfelelő eljárások segítségével tehetjük, amelyek – kihasználva a grafikus mód lehetőségeit – több betűtípust kínálnak különböző méretekben. A betűtípusok bitképei különálló állományokban helyezkednek el, melyek neve a betűtípusra utal, kiterjesztése pedig CHR. A CHR állományokra a megfelelő betűtípust használó program futtatásához van szükség.

A szöveges módnál bevezetett ablakok szerepét it az ún. **grafikus ablakok**, (Turbo Pascal terminológia szerint: **viewport**-ok) veszik át. A kétféle ablak kezelése logikailag közel azonos.

A B/K hibák jelzéséhez hasonló módon, a grafikus műveletek során előforduló hibákat a **GraphResult** nevű függvény hívásával tehetjük kezelhetővé. A függvény értéke az utoljára végrehajtott grafikus műveletre vonatkozó kód, melynek jelentése a következő:

```
 0: Nincs hiba
-1: Nem grafikus módban vagyunk
-2: A grafikus hardver hiányzik
-3: A grafikus alaprutin hiányzik
-4: Szabálytalan grafikus alaprutin-állomány
-5: Nincs elég memória a grafikus alaprutin betöltéséhez
-6: Nincs elég memória vonalhatár szerinti kitöltésnél
-7: Nincs elég memória színhatár szerinti kitöltésnél
-8: A betűtípus bitképét tartalmazó állomány hiányzik
-9: Nincs elég memória a betűtípus betöltéséhez
-10: A választott alaprutin és a grafikus mód ellentmondóak
-11: Grafikus hiba
-12: Grafikus B/K hiba
-13: Szabálytalan betűtípus-állomány
-14: Szabálytalan betűtípuskód
```

A **GraphResult** minden hívás után visszaáll 0-ra, ezért értékét – esetleges ismételt felhasználás céljából – külön tárolni kell.

A modul kapcsolati részében többféle típust deklaráltak a grafikus eljárások és függvények által kezelt adatok számára:

A grafikus adapterek színkezelési képessége eltérő, ezért a színekre vonatkozó információt egy rekordban, az ún. **palettában** tárolják.

```
type Palettetype = record Size   : byte;
                          Colors: array[0..MaxColors]
                                  of shortint
                   end;
```

A Colors tömb elemei rendre a színkódókat tartalmazzák.
Vonalak rajzolásánál használatos az a rekord, amely a vonal stílusát, mintáját és vastagságát írja le:

```
type LineSettingsType = record LineStyle: word;
                               Pattern: word;
                               Thickness: word
                        end;
```

A szövegek jellemzőit (betűtípust, írási irányt, betűméretet, a vízszintes és függőleges igazodást) a következő rekord írja le:

```
type TextSettinsType = record Font: word;
                              Direction: word;
                              CharSize: word;
                              Horiz: word;
                              Vert: word
                       end;
```

A különböző alakzatok kitöltésére használható „előre gyártott" mintákat, illetve ezek színét is egy rekord fejezi ki:

```
type FillSettingsType = record Pattern: word;
                               Color: word;
                        end;
```

Az előre gyártott mintákon kívül a felhasználó is készíthet kitöltésre használható mintákat. Ezek egy 8x8-as bitmátrixszal írhatók le, s a mintát tulajdonképpen e bitminta ismétlődése adja:

```
type FillPatternType = array[1..8] of byte;
```

A programozó kényelmét szolgálja a képernyő pontjainak kezelését támogató rekord:

```
type PointType = record X, Y: integer
                 end;
```

A grafikus ablakok leírásához a bal felső és jobb alsó sarok koordinátáin kívül még egy logikai változó is tartozik, amely megmondja, hogy az ablakból kivezető esetleges vonalakat le kell-e szabni az ablak méretére. Ha a következő típushoz tartozó rekordban a **Clip** értéke true, az ablakban induló vonalak nem hagyhatják el annak területét:

```
type ViewportType = record x1, y1, x2, y2: integer;
                           Clip: Boolean
                    end;
```

A kör-, illetve ellipszisívek kezeléséhez középpontjuk, valamint két végpontjuk koordinátáit kell egy rekordba foglalni:

```
type ArcCoordsType = record X, Y: integer;
                            Xstart, Ystart: integer;
                            Xend, Yend: integer
                     end;
```

A modulhoz meglehetősen sok konstans tartozik, ezek használata a programozó munkáját egyszerűsíti, hiszen könnyen összekeverhető kódszámok helyett a konstans szerepére emlékeztető nevekkel dolgozhat. Itt néhány ritkán használt konstans felsorolásától eltekintünk.

A grafikus alaprutinok nevei és értékei:

```
const Detect     = 0;
      CGA        = 1;
      MCGA       = 2;
      EGA        = 3;
      EGA64      = 4;
      EGAMono    = 5;
      IBM8514    = 6;
      HercMono   = 7;
      ATT400     = 8;
      VGA        = 9;
      PC3270     = 10;
      CurrentDriver = 128;
```

A **Detect** a Pascalra bízza a grafikus adapter felderítését és a megfelelő alaprutin, illetve grafikus mód kiválasztását.

A grafikus adapterekhez nemcsak alaprutinok, hanem általában különböző grafikus (üzem) módok is tartoznak. Ezek nevei és értékei a következők:

```
const CGAC0      = 0;      {320x200, 0-ás paletta (3 színnel)}
      CGAC1      = 1;      {320x200, 1-es paletta (3 színnel)}
      CGAC2      = 2;      {320x200, 2-es paletta (3 színnel)}
      CGAC3      = 3;      {320x200, 3-as paletta (3 színnel)}
      CGAHi      = 4;      {640x200, egyszínű}
      MCGAC0     = 0;      {320x200, 0-ás paletta (3 színnel)}
      MCGAC1     = 1;      {320x200, 1-es paletta (3 színnel)}
      MCGAC2     = 2;      {320x200, 2-es paletta (3 színnel)}
      MCGAC3     = 3;      {320x200, 3-as paletta (3 színnel)}
      MCGAMed    = 4;      {640x200, egyszínű}
```

```
MCGAHi      = 5;      {640x480, egyszínű}
EGALo       = 0;      {640x200, 16 szín}
EGAHi       = 1;      {640x350, 16 szín}
EGA64Lo     = 0;      {640x200, 16 szín}
EGA64Hi     = 1;      {640x350, 4 szín}
EGAMonoHi   = 3;      {640x350, egyszínű}
HercMonoHi  = 0;      {720x348, egyszínű}
ATT400C0    = 0;      {320x200, 0-ás paletta (3 színnel)}
ATT400C1    = 1;      {320x200, 1-es paletta (3 színnel)}
ATT400C2    = 2;      {320x200, 2-es paletta (3 színnel)}
ATT400C3    = 3;      {320x200, 3-as paletta (3 színnel)}
ATT400Med   = 4;      {640x200, egyszínű}
ATT400Hi    = 5;      {640x400, egyszínű}
VGALo       = 0;      {640x200, 16 szín}
VGAMed      = 1;      {640x350, 16 szín}
VGAHi       = 2;      {640x480, 16 szín}
PC3270Hi    = 0;      {720x350, egyszínű}
IBM8514Lo   = 0;      {640x480, 256 szín}
IBM8514Hi   = 1;      {1024x768, 256 szín}
```

A Crt modulban megismert színkonstansok ebben a modulban is szerepelnek, de megismétlésük felesleges.

A különböző vonalstílusok és -vastagságok konstansai:

```
const SolidLn   = 0;      {Folytonos vonal}
      DottedLn  = 1;      {Pontozott vonal}
      CenterLn  = 2;      {Pontokkal szaggatott vonal}
      DashedLn  = 3;      {Szaggatott vonal}
      UserBitLn = 4;      {Felhasználó által definiált vonalstílus}
```

A vonalak vastagsága kétféle lehet:

```
const NormWidth  = 1;     {Szokásos vastagság}
      ThickWidth = 2;     {Vastagított vonal}
```

Az egyes betűtípusok (fontok) megadására szolgáló konstansok:

```
const DefaultFont   = 0;   {Alaptípus, 8x8-as pontmátrixban
      TriplexFont   = 1;  definiálva.}
      SmallFont     = 2;   {Rugalmas, nagyításnál is alaktartó}
      SansSerifFont = 3;   {betűtípusok. A pontmátrix mérete}
      GothicFont    = 4;   {a karakterrel együtt változik.}
```

Az írás iránya vízszintes vagy függőleges (lentről felfelé) lehet, melyhez szintén konstansok tartoznak:

```
const HorizDir = 0;       {Vízszintes}
      VertDir  = 1;       {Függőleges}
```

A szövegek vízszintes és függőleges irányban, irányonként háromféle módon illeszkedhetnek AK-hoz. Ennek kódolásához öt konstansra van szükség:
Vízszintes illesztés:

```
const LeftText   = 0;      {A szöveg bal széle van AK-nál}
      CenterText = 1;      {A szöveg közepe van AK-nál}
      RightText  = 2;      {A szöveg jobb széle van AK-nál}
```

Függőleges illesztés:

```
const BottomText = 0;      {A szöveg alja van AK-nál}
      TopText    = 2;      {A szöveg teteje van AK-nál}
```

A grafikus ablakból kilépő vonalak levágását, illetve meghagyását lehet megadni a következő logikai konstansokkal:

```
const ClipOn  = true;      {Levág}
      ClipOff = false;     {Meghagy}
```

A háromdimenziós hasábdiagramok egymásra helyezésénél a közbenső hasábtetők behúzhatók vagy elhagyhatók (ez esetben csak az oszlop tetejére kerül tető). Ezt szintén logikai konstansokkal lehet megadni:

```
const TopOn  = true;       {Rajzolja a közbenső tetőket}
      TopOff = false;      {Nem rajzolja a közbenső tetőket}
```

A különféle alakzatok kitöltésére előre gyártott vagy a felhasználó által tervezett minták használhatók. Ezeket is konstansokkal lehet megadni:

```
const EmptyFill      = 0;  {Nincs minta, háttérszín}
      SolidFill      = 1;  {Nincs minta, töltőszín}
      LineFill       = 2;  {— töltés}
      LtSlashFill    = 3;  {/// töltés}
      SlashFill      = 4;  {Vastagított /// töltés}
      BkSlashFill    = 5;  {Vastagított \\\ töltés}
      LtBkSlashFill  = 6;  {\\\ töltés}
      HatchFill      = 7;  {Vékony satírozás}
      XHatchFill     = 8;  {Kétirányú, vastag satírozás}
      InterleaveFill = 9;  {Vonalkázás}
      WideDotFill    = 10; {Ritka pontozás}
      CloseDotFill   = 11; {Sűrű pontozás}
      UserFill       = 12; {Töltés a felhasználó által tervezett
                            mintával}
```

Az egyes képpontok új tartalma – a szöveges módban megszokottól eltérően – nemcsak felülírással, hanem az eredeti és az új tartalom közötti, különböző logikai műveletek eredményeként is kialakítható. A következő konstansok egyes rajzoló eljárások paraméterei lehetnek, illetve az írási mód beállítására szolgálnak:

```
const CopyPut        = 0;   {Felülírás}
      XORPut         = 1;   {XOR}
      OrPut          = 2;   {OR}
      AndPut         = 3;   {AND}
      NotPut         = 4;   {Felülírás az új tartalom inverzével}
```

A maximális színkód – melynek értéke 15 – helyett a **MaxColor** konstans is használható.

E típusok és konstansok birtokában áttekinthetjük a grafikus alkalmazásokat támogató eljárásokat és függvényeket.

procedure `Arc(x, y: integer; a1, a2, r: word);`

Az (**x**,**y**) középpont köré **r** sugarú körívet rajzol az **a1** kezdőszögtől az **a2** szögig. A szögeket fokban, az óramutató járásával ellentétes irányban mérjük, a 0° 3 óránál van.

procedure `Bar(x1, y1, x2, y2: integer);`

Az aktuális töltőmintával és színnel kitöltött téglalapot rajzol, melynek bal felső sarka az (**x1**,**y1**), jobb alsó sarka pedig az (**x2**,**y2**) pont.

procedure `Bar3D(x1, y1, x2, y2: integer; M: word; T: Boolean);`

Az aktuális töltőmintával és színnel kitöltött háromdimenziós hasábot rajzol, melynek bal felső sarka az (**x1**,**y1**), jobb alsó sarka pedig az (**x2**,**y2**) pont. A hasáb mélységét az **M** paraméter szabályozza, **T** pedig a hasáb tetőlapjának megjelenését vezérli. **T**=**true** esetén a tetőt megrajzolja, különben elhagyja. A tetőt több hasáb egymásra helyezésekor célszerű elhagyni. Az eljárás (eltérően a **Bar**-tól) a rajzolt hasábot az aktuális tintaszínnel bekeretezi. Ezért a 0-ás mélységű hasáb keretezett téglalapként jelenik meg.

procedure `Circle(x, y: integer; r: word);`

Az (**x**,**y**) középpont körül **r** sugarú kört rajzol az aktuális tintaszínnel.

procedure `ClearDevice;`

Törli a teljes grafikus képernyőt (az aktuális háttérszínt használva), és AK-t a (0,0) pontra állítja.

procedure `ClearViewPort;`

Törli az aktuális grafikus ablakot (az aktuális háttérszínt használva), és AK-t a (0,0) pontra állítja.

procedure `CloseGraph;`

Kilépés a grafikus módból. Visszaáll az előzőleg érvényes képernyőmód, és felszabadul minden, a grafikus alrendszer által lefoglalt memória.

procedure `DetectGraph(var gd, gm: integer);`

Megvizsgálja a hardvert, és eredményül az alkalmazható grafikus alaprutin kódját (**gd**) és a neki megfelelő (egyik) grafikus mód értékét (**gm**) adja. A kapott értékek az **InitGraph** eljárásban (l. később) használhatók. Ha grafikus adaptert nem talál, **GraphResult** és **gd** értéke –2 lesz. A grafikus alaprutinok nevei és kódjai:

```
const Detect    = 0;
      CGA       = 1;
      MCGA      = 2;
      EGA       = 3;
      EGA64     = 4;
      EGAMono   = 5;
      IBM8514   = 6;
      HercMono  = 7;
      ATT400    = 8;
      VGA       = 9;
      PC3270    = 10;
      CurrentDriver = 128;
```

A grafikus módok nevei és értékei a következők:

```
const CGAC0       = 0;      {320x200, 0-ás paletta (3 színnel)}
      CGAC1       = 1;      {320x200, 1-es paletta (3 színnel)}
      CGAC2       = 2;      {320x200, 2-es paletta (3 színnel)}
      CGAC3       = 3;      {320x200, 3-as paletta (3 színnel)}
      CGAHi       = 4;      {640x200, egyszínű}
      MCGAC0      = 0;      {320x200, 0-ás paletta (3 színnel)}
      MCGAC1      = 1;      {320x200, 1-es paletta (3 színnel)}
      MCGAC2      = 2;      {320x200, 2-es paletta (3 színnel)}
      MCGAC3      = 3;      {320x200, 3-as paletta (3 színnel)}
      MCGAMed     = 4;      {640x200, egyszínű}
      MCGAHi      = 5;      {640x480, egyszínű}
      EGALo       = 0;      {640x200, 16 szín}
      EGAHi       = 1;      {640x350, 16 szín}
      EGA64Lo     = 0;      {640x200, 16 szín}
      EGA64Hi     = 1;      {640x350, 4 szín}
      EGAMonoHi   = 3;      {640x350, egyszínű}
      HercMonoHi  = 0;      {720x348, egyszínű}
      ATT400C0    = 0;      {320x200, 0-ás paletta (3 színnel)}
      ATT400C1    = 1;      {320x200, 1-es paletta (3 színnel)}
      ATT400C2    = 2;      {320x200, 2-es paletta (3 színnel)}
      ATT400C3    = 3;      {320x200, 3-as paletta (3 színnel)}
      ATT400Med   = 4;      {640x200, egyszínű}
      ATT400Hi    = 5;      {640x400, egyszínű}
      VGALo       = 0;      {640x200, 16 szín}
      VGAMed      = 1;      {640x350, 16 szín}
      VGAHi       = 2;      {640x480, 16 szín}
      PC3270Hi    = 0;      {720x350, egyszínű}
      IBM8514Lo   = 0;      {640x480, 256 szín}
      IBM8514Hi   = 1;      {1024x768, 256 szín}
```

procedure DrawPoly(n: word; var pontok);

Törtvonalat rajzol az aktuális színnel és vonalstílussal. Az első paraméter (**n**) a töréspontok számát adja meg, a típus nélküli második paraméter pedig a csúcspontok koordinátáit tartalmazza **word** típusú (**x,y**) párok formájában. Ez célszerűen egy **PointType** típusú tömb lehet. Ha zárt törtvonalat (sokszöget) akarunk rajzolni **c** csúccsal, akkor **c+1** töréspontot kell használni, melyek közül az első és az utolsó azonos.

procedure Ellipse(x, y: integer; a1, a2, xr, yr: word);

Az (**x,y**) középpont köré **xr** vízszintes és **yr** függőleges sugarú elliptikus ívet rajzol az **a1** kezdőszögtől az **a2** szögig. A szögeket fokban, az óramutató járásával ellentétes irányban mérjük, a 0° 3 óránál van.

procedure FillEllipse(x, y: integer; xr, yr: word);

Az (**x,y**) középpont köré **xr** vízszintes és **yr** függőleges sugarú, kitöltött ellipszist rajzol az aktuális tintaszínnel. Az ellipszis az érvényes töltőszínnek és mintának megfelelően töltődik ki.

procedure FillPoly(n: word; var pontok);

Egy kitöltött sokszöget rajzol. Az első paraméter (**n**) a csúcspontok számát adja meg, a típus nélküli második paraméter (**pontok**) pedig a csúcspontok koordinátáit tartalmazza **word** típusú (**x,y**) párok formájában. Ez célszerűen egy **PointType** típusú tömb lehet. A sokszög határvonala az aktuális tintaszínnek és vonalstílusnak, töltése pedig az aktuális töltőszínnek és mintának felel meg. Ha a kitöltés folyamán elfogy a memória, **GraphResult** értéke –6 lesz.

procedure FloodFill(x, y: integer; hatar: word);

A grafikus képernyő egy zárt területét tölti ki az aktuális töltőszínnel és mintával. (**x,y**) a terület egy belső pontja, **hatar** pedig a területet határoló szín kódja. Ha a kitöltés folyamán elfogy a memória, **GraphResult** értéke –7 lesz

procedure GetArcCoords(var iv: ArcCoordsType);

Az **Arc** vagy **Ellipse** eljárás (l. előbb) utolsó hívásának paramétereit reprodukálja. Akkor hasznos, ha egy ívet más vonallal akarunk folytatni.

procedure GetAspectRatio(var fx, fy: word);

A grafikus képernyő **x** és **y** irányú tényleges felbontását (a képpontok soronkénti, illetve oszloponkénti számát) adja, amelyből az **fx/fy** torzítási arány meghatározható.

function GetBkColor: word;

Értéke az aktuális háttérszín kódja.

function GetColor: word;

Értéke az aktuális tintaszín kódja.

procedure GetDefaultPalette(var p: PaletteType);

A grafikus alrendszer indításakor az alaprutin által beállított palettakiosztást (a színek számát és kódjait) adja **p**-ben.

function GetDriverName: string;

Értéke a grafikus alrendszer indításakor kiválasztott alaprutin neve.

procedure GetFillPattern(var tm: FillPatternType);

Az aktuális töltőmintát adja **tm**-ben, egy 8x8-as bitmátrix formájában. Ha még nincs beállított minta, **tm** csupa 1-t tartalmaz.

procedure GetFillSettings(var tr: FillSettingsType);

Az aktuális töltőminta kódját és színét adja **tr**-ben. Az egyes minták kódjai és konstansai:

A grafikus módok nevei és értékei a következők:

```
const EmptyFill      = 0;    {Nincs minta, háttérszín}
      SolidFill      = 1;    {Nincs minta, töltőszín}
      LineFill       = 2;    {— töltés}
      LtSlashFill    = 3;    {/// töltés}
      SlashFill      = 4;    {Vastagított /// töltés}
      BkSlashFill    = 5;    {Vastagított \\\ töltés}
      LtBkSlashFill  = 6;    {\\\ töltés}
      HatchFill      = 7;    {Vékony satírozás}
      XHatchFill     = 8;    {Kétirányú, vastag satírozás}
      InterleaveFill = 9;    {Vonalkázás;}
      WideDotFill    = 10;   {Ritka pontozás}
      CloseDotFill   = 11;   {Sűrű pontozás}
      UserFill       = 12;   {Töltés a felhasználó által
                              tervezett mintával}
```

Ha a mintakód **UserFill**, a mintát a **GetFillPattern** (l. előbb) hívásával kaphatjuk meg.

function GetGraphMode: integer;

Értéke a grafikus alrendszer indításakor beállított grafikus mód kódja, amely az alaprutinnal együtt meghatározza a grafikus környezetet.

procedure GetImage(x1, y1, x2, y2: integer; var bitkep);

A grafikus képernyő (**x1,y1**) bal felső és (**x2,y2**) jobb alsó sarokpontok által kijelölt téglalapján lévő képet átmásolja a **bitkep** pufferbe. **bitkep** első két szava a terület szélességét és magasságát tartalmazza, teljes méretét az **ImageSize** (l. később) függvény segítségével lehet meghatározni. Egy kép tárolására legfeljebb 64 kbyte memória vehető igénybe.

procedure GetLineSettings(var vi: LineSettingsType);

Az aktuális vonalstílus, vonalminta és -vastagság kódját adja **vi**-ben.

A vonalstílusok és -vastagságok konstansai:

```
const SolidLn       = 0;    {Folytonos vonal}
      DottedLn      = 1;    {Pontozott vonal}
      CenterLn      = 2;    {Pontokkal szaggatott vonal}
      DashedLn      = 3;    {Szaggatott vonal}
      UserBitLn     = 4;    {Felhasználó által definiált
                             vonalstílus}
const NormWidth     = 1;    {Szokásos vastagság}
      ThickWidth    = 2;    {Vastagított vonal}
```

function GetMaxColor: **word**;

Értéke az aktuális grafikus környezetben használható maximális színkód.

function GetMaxMode: **word**;

Értéke az aktuális grafikus alaprutinhoz tartozó maximális mód-kód.

function GetMaxX: **integer**;

Értéke a grafikus képernyőn lévő legnagyobb X koordináta, 320x200-as képernyő esetén pl. 319.

function GetMaxY: **integer**;

Értéke a grafikus képernyőn lévő legnagyobb Y koordináta, 320x200-as képernyő esetén pl. 199.

function GetModeName: **string**;

Értéke az aktuális grafikus mód neve szöveges formában (pl. 320x200 CGA P1).

procedure GetModeRange(gd: **integer**; **var** m1, m2: **integer**);

A **gd** kódú grafikus alaprutinhoz megadja a legkisebb (**m1**) és legnagyobb (**m2**) mód kódját. Ha **gd** értéke érvénytelen, **m1**=**m2**= −**1** lesz. A grafikus alaprutinok nevei és kódjai:

```
const Detect     = 0;
      CGA        = 1;
      MCGA       = 2;
      EGA        = 3;
      EGA64      = 4;
      EGAMono    = 5;
      IBM8514    = 6;
      HercMono   = 7;
      ATT400     = 8;
      VGA        = 9;
      PC3270     = 10;
      CurrentDriver = 128;
```

A grafikus módok nevei és értékei a következők:

```
const CGAC0       = 0;      {320x200, 0-ás paletta (3 színnel)}
      CGAC1       = 1;      {320x200, 1-es paletta (3 színnel)}
      CGAC2       = 2;      {320x200, 2-es paletta (3 színnel)}
      CGAC3       = 3;      {320x200, 3-as paletta (3 színnel)}
      CGAHi       = 4;      {640x200, egyszínű}
      MCGAC0      = 0;      {320x200, 0-ás paletta (3 színnel)}
      MCGAC1      = 1;      {320x200, 1-es paletta (3 színnel)}
      MCGAC2      = 2;      {320x200, 2-es paletta (3 színnel)}
      MCGAC3      = 3;      {320x200, 3-as paletta (3 színnel)}
      MCGAMed     = 4;      {640x200, egyszínű}
      MCGAHi      = 5;      {640x480, egyszínű}
      EGALo       = 0;      {640x200, 16 szín}
      EGAHi       = 1;      {640x350, 16 szín}
      EGA64Lo     = 0;      {640x200, 16 szín}
      EGA64Hi     = 1;      {640x350, 4 szín}
      EGAMonoHi   = 3;      {640x350, egyszínű}
      HercMonoHi  = 0;      {720x348, egyszínű}
      ATT400C0    = 0;      {320x200, 0-ás paletta (3 színnel)}
      ATT400C1    = 1;      {320x200, 1-es paletta (3 színnel)}
      ATT400C2    = 2;      {320x200, 2-es paletta (3 színnel)}
      ATT400C3    = 3;      {320x200, 3-as paletta (3 színnel)}
      ATT400Med   = 4;      {640x200, egyszínű}
      ATT400Hi    = 5;      {640x400, egyszínű}
      VGALo       = 0;      {640x200, 16 szín}
      VGAMed      = 1;      {640x350, 16 szín}
      VGAHi       = 2;      {640x480, 16 szín}
      PC3270Hi    = 0;      {720x350, egyszínű}
      IBM8514Lo   = 0;      {640x480, 256 szín}
      IBM8514Hi   = 1;      {1024x768, 256 szín}
```

procedure GetPalette(var p: PaletteType);

Az aktuális paletta méretét és a hozzá tartozó színkódokat adja **p**-ben.

function GetPaletteSize: word;

Értéke az aktuális paletta mérete, azaz a hozzá tartozó színek száma.

function GetPixel(x, y: integer): word;

Értéke az (**x**,**y**) koordinátájú képernyőpont színkódja.

procedure GetTextSettings(var txt: TextSettingsType);

Az aktuális betűtípust, írási irányt, betűméretet és a szöveg illesztési módját adja **txt**-ben. Az értékeket kódjuk képviseli.

Az egyes betűtípusok (fontok) megadására szolgáló konstansok:

```
const DefaultFont    = 0;    {Alaptípus, 8x8-as pontmátrixban}
      TriplexFont    = 1;    {definiálva.}
      SmallFont      = 2;    {Rugalmas, nagyításnál is alaktartó}
      SansSerifFont  = 3;    {betűtípusok. A pontmátrix mérete}
      GothicFont     = 4;    {a karakterrel együtt változik.}
```

Az írás iránya vízszintes vagy függőleges (lentről felfelé) lehet, melyhez szintén konstansok tartoznak:

```
const HorizDir = 0;        {Vízszintes}
      VertDir  = 1;        {Függőleges}
```

procedure GetViewSettings(var aa: ViewPortType);

Az aktuális grafikus ablak bal felső és jobb alsó sarkának koordinátáit, valamint az ablakból kilépő vonalak levágását vezérlő kapcsoló értékét adja **aa**-ban.

function GetX: integer;

Értéke az AK ablak-relatív X koordinátája. Az aktuális ablak bal felső sarkán értéke 0.

function GetY: integer;

Értéke az AK ablak-relatív Y koordinátája. Az aktuális ablak bal felső sarkán értéke 0.

procedure GraphDefaults;

Visszaállítja a grafikus alrendszer alaphelyzetét. Ennek eredményeként a következő paraméterek kapják vissza alapfeltételezés szerinti értéküket:

- grafikus ablak,
- paletta,
- tinta- és háttérszín,
- vonalstílus és -minta,
- töltőstílus, töltőminta és -szín,
- betűtípus,
- szövegstílus, szövegillesztés,
- a felhasználó által beállított karakterméret.

function GraphErrorMsg(hibakod: integer): string;

Értéke a paraméterként adott hibakódhoz tartozó hibaüzenet. A hibakódokat (**GraphResult** lehetséges értékeit) és jelentésüket lásd a fejezet elején. A –3-hoz tartozó üzenet pl. „Device driver not found".

function GraphResult: integer;

Értéke az utolsó grafikus művelet hibakódja. Értékét minden olyan grafikus eljárás és függvény módosíthatja, amelynek végrehajtása során hiba léphet fel. A 0 hibátlan műveletet jelez, maguk a hibakódok negatív számok, melyek értékét és jelentését a fejezet elején soroltuk fel. A függvény értéke minden hívás után törlődik, ezért ismételt felhasználásához a tárolásról a programnak kell gondoskodnia.

function ImageSize(x1, y1, x2, y2: integer): word;

Értéke az (**x1,y1**) bal felső és (**x2,y2**) jobb alsó sarkai által meghatározott, téglalap alakú grafikus képernyőterületen lévő bitkép tárolásához szükséges byte-ok száma. Ha ez az érték meghaladná a 64 kbyte-ot, a függvény értéke 0, **GraphResult** (l. előbb) értéke pedig −11 lesz.

procedure InitGraph(var gd, gm: integer; dp: string);

A grafikus alrendszer kezdeti állapotának beállítása és indítása. Az első paraméter a grafikus alaprutin kódja. Ha ennek értéke **Detect**, azaz 0, a rendszer megvizsgálja az aktuális hardvert és az ennek megfelelő alaprutint, illetve grafikus módot használja. Ez esetben a második paraméter (grafikus mód) értéke közömbös. Ha a felhasználó nem bízza a rendszerre az alaprutin-választást, megadhatja annak és a kívánt grafikus módnak a kódját. A harmadik paraméter a grafikus alaprutint tartalmazó könyvtár (és a hozzá vezető út) neve karakterlánc formájában. Ha ez a karakterlánc üres, az alaprutinnak az aktuális könyvtárban kell lennie.

A grafikus alaprutinok nevei és kódjai:

```
const Detect     = 0;
      CGA        = 1;
      MCGA       = 2;
      EGA        = 3;
      EGA64      = 4;
      EGAMono    = 5;
      IBM8514    = 6;
      HercMono   = 7;
      ATT400     = 8;
      VGA        = 9;
      PC3270     = 10;
      CurrentDriver = 128;
```

A grafikus módok nevei és értékei a következők:

```
const CGAC0      = 0;      {320x200, 0-ás paletta (3 színnel)}
      CGAC1      = 1;      {320x200, 1-es paletta (3 színnel)}
      CGAC2      = 2;      {320x200, 2-es paletta (3 színnel)}
      CGAC3      = 3;      {320x200, 3-as paletta (3 színnel)}
      CGAHi      = 4;      {640x200, egyszínű}
      MCGAC0     = 0;      {320x200, 0-ás paletta (3 színnel)}
      MCGAC1     = 1;      {320x200, 1-es paletta (3 színnel)}
      MCGAC2     = 2;      {320x200, 2-es paletta (3 színnel)}
      MCGAC3     = 3;      {320x200, 3-as paletta (3 színnel)}
      MCGAMed    = 4;      {640x200, egyszínű}
      MCGAHi     = 5;      {640x480, egyszínű}
      EGALo      = 0;      {640x200, 16 szín}
      EGAHi      = 1;      {640x350, 16 szín}
```

```
EGA64Lo     = 0;    {640x200, 16 szín}
EGA64Hi     = 1;    {640x350, 4 szín}
EGAMonoHi   = 3;    {640x350, egyszínű}
HercMonoHi  = 0;    {720x348, egyszínű}
ATT400C0    = 0;    {320x200, 0-ás paletta (3 színnel)}
ATT400C1    = 1;    {320x200, 1-es paletta (3 színnel)}
ATT400C2    = 2;    {320x200, 2-es paletta (3 színnel)}
ATT400C3    = 3;    {320x200, 3-as paletta (3 színnel)}
ATT400Med   = 4;    {640x200, egyszínű}
ATT400Hi    = 5;    {640x400, egyszínű}
VGALo       = 0;    {640x200, 16 szín}
VGAMed      = 1;    {640x350, 16 szín}
VGAHi       = 2;    {640x480, 16 szín}
PC3270Hi    = 0;    {720x350, egyszínű}
IBM8514Lo   = 0;    {640x480, 256 szín}
IBM8514Hi   = 1;    {1024x768, 256 szín}
```

Az eljárás végrehajtásakor többféle hiba fordulhat elő, ezek kódjai:

−2: Hiányzik a grafikus adapter
−3: Hiányzik a grafikus alaprutin
−4: Az alaprutint tartalmazó állomány hibás
−5: Nincs elég memória az alaprutin betöltéséhez
−10: Az alaprutin és a grafikus mód nincs összhangban

function `InstallUserDriver(nev: string; af: pointer): word;`

Értéke a nyilvántartásba vett új, felhasználói grafikus alaprutin kódja. Az első paraméter (**név**) az alaprutint tartalmazó állomány (amely a többi, *.BGI állománnyal azonos formájú) neve. A második paraméter (**af**) egy opcionális, egész értékű függvényre hivatkozó mutató. Ez a függvény az alaprutin része, s annak automatikus felismerésére képes, hogy az alaprutin az aktuális grafikus adapterrel használható-e. Ha ilyen eljárás nem tartozik az alaprutinhoz, **af** értékét **nil**-nek kell választani. A nyilvántartásba vett alaprutint az **InitGraph** eljárás (l. előbb) a standard alaprutinokhoz hasonlóan kezeli, az automatikus felismerés esetleges hiányától eltekintve. A nyilvántartásba vétel egy belső táblázatba való bejegyzést is jelent. Ha ez a táblázat betelik, a függvény értéke a −11 hibakód.

function `InstallUserFont(fn: string): integer;`

Értéke a nyilvántartásba vett új, a Pascal rendszerben nem szereplő betűtípus kódja. Az **fn** paraméter a betűtípust tartalmazó állomány (amely a többi, *.CHR állománnyal azonos formájú) neve. A sikeres nyilvántartásba vétel után a betűtípusra – akár az eredeti típusokra – kódja segítségével lehet hivatkozni. A nyilvántartásba vétel egy belső táblázatba való bejegyzést is jelent. Ha ez a táblázat betelik, a függvény értéke 0 (**DefaultFont**).

procedure `Line(x1, y1, x2, y2: integer);`

Egyenes vonalat húz az (**x1,y1**) és az (**x2,y2**) pont között. A vonal színét, stílusát, vastagságát a **SetLineStyle** és a **SetColor** eljárásokkal (l. később) szabályozhatjuk. A vonal pontjai kétféle módon kerülhetnek a képernyőre: **másolással** (ekkor a képernyő előző tartalmától függetlenül megjelenik az új tartalom) vagy **XOR művelettel** (ekkor minden képpont-információra bitenként hajtódik végre a művelet az eredeti és az új tartalom között, majd ennek eredménye jelenik meg). Az írás módját a **SetWriteMode** eljárással (l. később) lehet állítani.

procedure `LineRel(dx, dy: integer);`

Egyenes vonalat húz **AK** és a tőle (**dx,dy**) távolságban lévő pont között. **AK** új értéke a vonal utolsó pontja lesz. A vonal színét, stílusát, vastagságát a **SetLineStyle** és a **SetColor** eljárásokkal (l. később) szabályozhatjuk. A vonal pontjai kétféle módon kerülhetnek a képernyőre: **másolással** (ekkor a képernyő előző tartalmától függetlenül megjelenik az új tartalom) vagy **XOR művelettel** (ekkor minden képpont-információra bitenként hajtódik végre a művelet az eredeti és az új tartalom között, majd ennek eredménye jelenik meg). Az írás módját a **SetWriteMode** eljárással (l. később) lehet állítani.

procedure `LineTo(x, y: integer);`

Egyenes vonalat húz **AK** és az (**x,y**) pont között. **AK** új értéke (**x,y**) lesz. A vonal pontjai kétféle módon kerülhetnek a képernyőre: **másolással** (ekkor a képernyő előző tartalmától függetlenül megjelenik az új tartalom) vagy **XOR művelettel** (ekkor minden képpont-információra bitenként hajtódik végre a művelet az eredeti és az új tartalom között, majd ennek eredménye jelenik meg). Az írás módját a **SetWriteMode** eljárással (l. később) lehet állítani.

procedure `MoveRel(dx, dy: integer);`

Áthelyezi **AK**-t jelenlegi helyétől számított **dx** vízszintes és **dy** függőleges távolságra lévő képpontra.

procedure `MoveTo(x, y: integer);`

AK-t az aktuális grafikus ablak (**x,y**) képpontjára állítja. Mivel **AK** a teljes képernyőn mozoghat, előfordulhat, hogy elhagyja az aktív ablakot.

procedure `OutText(txt: string);`

A **txt** szöveget kiírja az aktuális grafikus ablakba **AK**-tól kezdve. A szöveg nem lépheti át az ablakhatárt. Ha valamelyik rugalmas betűtípus aktív, a szöveg az ablakhatárnál csonkul, az alaptípus esetén azonban a teljes kiírás elmarad, ha a szöveg nem fér el az ablakban. Az írásnál használt betűtípust, irányt és karakter-méretet a **SetTextStyle**, a szöveg **AK**-hoz való illesztését pedig a **SetTextJustify** eljárás (l. később)

szabályozza. Az írás után **AK** csak akkor áll át automatikusan a szöveg végére, ha az írás iránya vízszintes és a szöveg **AK**-hoz vízszintesen balra illesztett.

procedure `OutTextXY(x, y: integer; txt: string);`

A **txt** szöveget kiírja az aktuális grafikus ablakba, annak (**x,y**) pontjától kezdve. A szöveg nem lépheti át az ablakhatárt. Ha valamelyik rugalmas betűtípus aktív, a szöveg az ablakhatárnál csonkul, az alaptípus esetén azonban a teljes kiírás elmarad, ha a szöveg nem fér el az ablakban. Az írásnál használt betűtípust, irányt és karakterméretet a **SetTextStyle**, a szöveg (**x,y**)-hoz való illesztését pedig a **SetTextJustify** eljárás (l. később) szabályozza.

procedure `PieSlice(x, y: integer; a1, a2, r: word);`

Kitöltött „tortaszeletet" rajzol (**x,y**) középponttal, **r** sugárral az **a1** kezdő középponti szögtől az **a2** szögig. A szögeket fokban, az óramutató járásával ellentétes irányban mérjük, a 0° 3 óránál van. A tortaszelet határvonalának színe az aktuális tintaszínhez, töltésének színe és mintája pedig az aktuális töltőszínhez és mintához igazodik. Ha a kitöltés során a memória elfogyna, **GraphResult** értéke –6 lesz.

procedure `PutImage(x, y: integer; var bitkep; op: word);`

A **bitkép** paraméterrel megadott képet kiviszi az aktuális grafikus ablak azon téglalap alakú területére, amelynek bal felső sarka az (**x,y**) pont. Az **op** paraméter a képpontonként alkalmazott művelet kódja, amellyel a képernyő új tartalma létrejön. **op** lehetséges értékei:

```
const CopyPut = 0;      {Felülírás}
      XORPut  = 1;      {XOR}
      OrPut   = 2;      {OR}
      AndPut  = 3;      {AND}
      NotPut  = 4;      {Felülírás az új tartalom inverzével}
```

A **CopyPut** egyszerű, a **NotPut** az új tartalom inverzével való felülírást jelent, a többi három pedig a régi és az új tartalom között bitenként elvégzett logikai művelet eredményét viszi a képernyőre. Ha a kép nem fér el az aktuális grafikus ablakban, de elfér a teljes képernyőn, csonkulás nélkül jelenik meg. Ha a kép nem fér el a képernyőn, a kiírás el sem indul, hacsak nem a képernyő alsó határát kellene átlépni, mert ekkor megjelenik a csökkentett magasságú kép.

procedure `PutPixel(x, y: integer; szin: word);`

A képernyő (**x,y**) koordinátájú helyére a harmadik paraméter által adott színű pontot rajzol.

procedure `Rectangle(x1, y1, x2, y2: integer);`

Téglalapot rajzol (**x1,y1**) bal felső és (**x2,y2**) jobb alsó sarokkal. A vonal színét, stílusát, vastagságát a **SetLineStyle** és a **SetColor** eljárásokkal (l. később) szabályozhatjuk. A téglalap kerületi pontjai kétféle

módon kerülhetnek a képernyőre: **másolással** (ekkor a képernyő előző tartalmától függetlenül megjelenik az új tartalom) vagy **XOR művelettel** (ekkor minden képpont-információra bitenként hajtódik végre a művelet az eredeti és az új tartalom között, majd ennek eredménye jelenik meg). Az írás módját a **SetWriteMode** eljárással (l. később) lehet állítani.

function `RegisterBGIdriver(ar: pointer): integer;`

Értéke a felhasználó által betöltött vagy a tárgyprogramhoz kapcsolt eredeti Borland, ún. BGI grafikus alaprutin regisztrálásakor kapott kódszám. Ha később a grafikus alrendszer a regisztrált alaprutint használja, azt nem kell már betöltenie. A regisztrálandó alaprutint betölthetjük a dinamikus memóriába, vagy .OBJ állománnyá alakíthatjuk a BINOBJ.EXE program segítségével, és hozzákapcsolhatjuk végrehajtható programunkhoz ({$L} direktíva). Az eljárás főleg ez utóbbi esetben használatos, hiszen az alaprutin egyszerű betöltését az **InitGraph** eljárás is elvégzi. Ekkor a teendők sorrendje a következő:

- ▷ Futtassuk le a BINOBJ.EXE programot a kívánt alaprutin-állomá-
  nyokra, ezzel előállnak a megfelelő .OBJ állományok.
- ▷ A {$L} direktíva használatával kapcsoljuk az .OBJ állományokat programunkhoz.
- ▷ Az **InitGraph** hívása előtt hívjuk a regisztráló függvényt.

A következő modul három grafikus alaprutin felhasználói programhoz való kapcsolását végzi el. A CGA.BGI, EGAVGA.BGI és a HERC.BGI alaprutinokból a BINOBJ.EXE program hozta létre a CGA.OBJ, EGAVGA.OBJ és a HERC.OBJ állományokat. A három external eljárás csak nevet ad a beépített alaprutinoknak, amire a **RegisterBGIdriver** függvény hívásakor hivatkozni lehet.

```
unit Grutinok;
interface
procedure CgaRutin;
procedure EgaVgaRutin;
procedure HercRutin;

implementation
procedure CgaRutin; external;
$L CGA.OBJ

procedure EgaVgaRutin; external;
$L EGAVGA.OBJ

procedure HercRutin; external;
```

```
$L HERC.OBJ

end.
```

A programban a grafikus alrendszer aktivizálása (InitGraph) előtt a regisztrálás hibavizsgálattal egybekötve a következő lehet:

```
if RegisterBGIdriver(@CgaRutin)<0
then begin writeln(GraphErrorMsg(GraphResult));
           halt(1)
     end;

stb...
```

A regisztrálással kapcsolatos hibakódok: −4 (az alaprutint a rendszer nem ismerte fel) és −11 (aktív grafikus alrendszer mellett hívtuk a regisztráló függvényt).

function RegisterBGIfont(bt: **pointer**): **integer**;

Értéke a felhasználó által betöltött vagy a tárgyprogramhoz kapcsolt, eredeti Borland betűtípus (*.CHR állomány) regisztrálásakor kapott kódszám. Ha később a grafikus alrendszer a regisztrált betűtípust használja, azt nem kell már betöltenie. A regisztrálandó betűtípust betölthetjük a dinamikus memóriába, vagy .OBJ állománnyá alakíthatjuk a BINOBJ.EXE program segítségével, és hozzákapcsolhatjuk végrehajtható programunkhoz ({$L} direktíva). Az eljárás főleg ez utóbbi esetben használatos, hiszen a betűtípus egyszerű betöltését a **SetTextStyle** (l. később) eljárás is elvégzi. A betűtípusoknak a felhasználó programjába való beépítése mindenben megegyezik a grafikus alaprutinok beépítésével (l. az előző **registerBGIdriver** függvényt), csak a *.BGI helyett a *.CHR állományokat kell használni.
A regisztrálással kapcsolatos hibakódok: −11 (aktív grafikus alrendszer mellett hívtuk a regisztráló függvényt), −13 és −14 (a betűtípusállományt a rendszer nem ismerte fel).

procedure RestoreCrtMode;

Visszaállítja a grafikus alrendszer aktivizálása előtti képernyőmódot.

procedure Sector(x, y: **integer**; a1, a2, xr, yr: **word**);

Az (**x**,**y**) középpont köré **xr** vízszintes és **yr** függőleges sugarú, kitöltött ellipsziscikket rajzol az **a1** kezdőszögtől az **a2** szögig. A szögeket fokban, az óramutató járásával ellentétes irányban mérjük, a 0° 3 óránál van. A határvonal színe az aktuális tintaszín, a töltésnél pedig az aktuális töltőszín és minta érvényesül. Ha kitöltés közben a memória elfogy, **GraphResult** értéke −6 lesz.

procedure SetActivePage(lsz: **word**);

A grafikus adapter **lsz** sorszámú lapját aktivizálja. Ezután minden grafikus kivitel erre a lapra kerül. Csak az EGA(256K), VGA és Hercules

adapterek rendelkeznek több lappal. Ezek segítségével elérhető, hogy egy nem látszó lapon előkészítjük a képet, majd a **SetVisualPage** eljárás (l. később) hívásával igen gyorsan láthatóvá tesszük. Ez a technika elsősorban animációknál hasznos.

procedure SetAllPalette(var p);

A paletta színeit módosítja a paraméternek megfelelően. A **p** típus nélküli paraméter első byte-ja a paletta módosítandó részének hossza, a következő byte-ok pedig rendre új színkódokat, illetve −1-et tartalmaznak. A −1 hatására megmarad a paletta eredeti színe. A paletta módosulásának eredménye azonnal a képernyőre is kerül. A színkódok és a megfelelő konstansok a következők:

```
const Black        = 0;    {Fekete}
      Blue         = 1;    {Kék}
      Green        = 2;    {Zöld}
      Cyan         = 3;    {Türkiz}
      Red          = 4;    {Piros}
      Magenta      = 5;    {Lila}
      Brown        = 6;    {Barna}
      LightGray    = 7;    {Világosszürke}
      DarkGray     = 8;    {Sötétszürke}
      LightBlue    = 9;    {Világoskék}
      LightGreen   = 10;   {Világoszöld}
      LightCyan    = 11;   {Világostürkiz}
      LightRed     = 12;   {Rózsaszín}
      LightMagenta = 13;   {Világoslila}
      Yellow       = 14;   {Sárga}
      White        = 15;   {Fehér}
```

procedure SetAspectRatio(xa, ya: word);

Az aktuális grafikus módhoz tartozó, alapfeltételezés szerinti torzítási arány módosítása **xa/ya**-ra. Az x és y irányban nem azonos felbontású képernyőkön a torzítási arány alkalmazásával lehet szabályos kört rajzolni. Ha az automatikusan beállított arány (amely a **GetAspect Ratio** eljárással lekérdezhető) valamilyen okból nem megfelelő, ezzel az eljárással megváltoztatható.

procedure SetBkColor(szin: word);

Beállítja a háttérszínt az aktuális paletta és a paraméter (**szin**) alapján. Ha **szin**=0, a háttérszín a palettától függetlenül fekete lesz.

procedure SetColor(szin: word);

Beállítja a tintaszínt az aktuális paletta és a paraméter (**szin**) alapján. A grafikus alaprutintól és grafikus módtól függő színkódok 0 és **GetMaxColor** értéke közé esnek.

procedure SetFillPattern(minta: FillPatternType; szin: **word**);

Beállít egy – a felhasználó által definiált – töltőmintát és -színt. A 8x8-as bitmátrix formájában megadható minta lehetővé teszi, hogy az előre definiált és a **SetFillStyle** (l. később) eljárással aktivizálható minták körét bővítsük. Hibás paraméter esetén **GraphResult** értéke −11 lesz.

procedure SetFillStyle(minta, szin: **word**);

Beállít egy előre definiált töltőmintát és -színt.

Az egyes minták kódjai és konstansai:

```
const EmptyFill      = 0;    {Nincs minta, háttérszín}
      SolidFill      = 1;    {Nincs minta, töltőszín}
      LineFill       = 2;    {— töltés}
      LtSlashFill    = 3;    {/// töltés}
      SlashFill      = 4;    {Vastagított /// töltés}
      BkSlashFill    = 5;    {Vastagított \\\ töltés}
      LtBkSlashFill  = 6;    {\\\ töltés}
      HatchFill      = 7;    {Vékony satírozás}
      XHatchFill     = 8;    {Kétirányú, vastag satírozás}
      InterleaveFill = 9;    {Vonalkázás;}
      WideDotFill    = 10;   {Ritka pontozás}
      CloseDotFill   = 11;   {Sűrű pontozás}
      UserFill       = 12;   {Töltés a felhasználó által tervezett
                              mintával}
```

A 12-es kód (**UserFill**) a felhasználó által definiált és a **SetFillPattern** eljárással aktivizálható mintákra való áttérést jelenti. Hibás paraméter esetén **GraphResult** értéke −11 lesz.

procedure SetGraphBufSize(n: **word**);

A különböző alakzatok kitöltéséhez használt, a dinamikus memóriában elhelyezkedő pufferterület méretének beállítása **n** byte-ra. Az eljárás csak az **InitGraph** hívása előtt használható, utána hatástalan. Az **InitGraph** hívásakor az alapfeltételezés szerinti pufferméret 4 kbyte lesz, ami általában elegendő.

procedure SetGraphMode(mód: **integer**);

A grafikus módot a paraméterrel adottra változtatja, törli a képernyőt és alaphelyzetbe állítja a grafikus paramétereket (AK, paletta, szín, grafikus ablak stb.). Akkor használatos, ha az **InitGraph** által beállított grafikus módon változtatni akarunk.

A grafikus módok nevei és értékei a következők:

```
const CGAC0     = 0;    {320x200, 0-ás paletta (3 színnel)}
      CGAC1     = 1;    {320x200, 1-es paletta (3 színnel)}
      CGAC2     = 2;    {320x200, 2-es paletta (3 színnel)}
      CGAC3     = 3;    {320x200, 3-as paletta (3 színnel)}
      CGAHi     = 4;    {640x200, egyszínű}
```

```
MCGAC0      = 0;      {320x200, 0-ás paletta (3 színnel)}
MCGAC1      = 1;      {320x200, 1-es paletta (3 színnel)}
MCGAC2      = 2;      {320x200, 2-es paletta (3 színnel)}
MCGAC3      = 3;      {320x200, 3-as paletta (3 színnel)}
MCGAMed     = 4;      {640x200, egyszínű}
MCGAHi      = 5;      {640x480, egyszínű}
EGALo       = 0;      {640x200, 16 szín}
EGAHi       = 1;      {640x350, 16 szín}
EGA64Lo     = 0;      {640x200, 16 szín}
EGA64Hi     = 1;      {640x350, 4 szín}
EGAMonoHi   = 3;      {640x350, egyszínű}
HercMonoHi  = 0;      {720x348, egyszínű}
ATT400C0    = 0;      {320x200, 0-ás paletta (3 színnel)}
ATT400C1    = 1;      {320x200, 1-es paletta (3 színnel)}
ATT400C2    = 2;      {320x200, 2-es paletta (3 színnel)}
ATT400C3    = 3;      {320x200, 3-as paletta (3 színnel)}
ATT400Med   = 4;      {640x200, egyszínű}
ATT400Hi    = 5;      {640x400, egyszínű}
VGALo       = 0;      {640x200, 16 szín}
VGAMed      = 1;      {640x350, 16 szín}
VGAHi       = 2;      {640x480, 16 szín}
PC3270Hi    = 0;      {720x350, egyszínű}
IBM8514Lo   = 0;      {640x480, 256 szín}
IBM8514Hi   = 1;      {1024x768, 256 szín}
```

Hibás paraméter esetén **GraphResult** értéke −10 lesz.

```
procedure SetLineStyle(st, mt, va: word);
```

A grafikus alrendszerben rajzolt vonalak stílusát (**st**), mintáját (**mt**) és vastagságát (**va**) állítja be.

A vonalstílusok és -vastagságok konstansai:

```
const SolidLn     = 0;      {Folytonos vonal}
      DottedLn    = 1;      {Pontozott vonal}
      CenterLn    = 2;      {Pontokkal szaggatott vonal}
      DashedLn    = 3;      {Szaggatott vonal}
      UserBitLn   = 4;      {Felhasználó által definiált vonalstílus}
const NormWidth   = 1;      {Szokásos vastagság}
      ThickWidth  = 2;      {Vastagított vonal}
```

Az **mt** paraméter csak akkor érvényesül, ha a stíluskód értéke 4 (**UserBitLn**). Ilyenkor egy legfeljebb 16 bites minta adható meg, ami a vonal rajzolása során ismétlődik. Hibás paraméter esetén **GraphResult** értéke −11 lesz.

procedure SetPalette(n: **word**; szin: **shortint**);

A paletta **n**-edik színkódját a **szin** által definiált színre módosítja. Az esetleges színváltozások a képernyőn azonnal megjelennek. Hibás paraméter esetén **GraphResult** értéke −11 lesz. A színkódok és a megfelelő konstansok a következők:

```
const const Black   = 0;      {Fekete}
      Blue          = 1;      {Kék}
      Green         = 2;      {Zöld}
      Cyan          = 3;      {Türkiz}
      Red           = 4;      {Piros}
      Magenta       = 5;      {Lila}
      Brown         = 6;      {Barna}
      LightGray     = 7;      {Világosszürke}
      DarkGray      = 8;      {Sötétszürke}
      LightBlue     = 9;      {Világoskék}
      LightGreen    = 10;     {Világoszöld}
      LightCyan     = 11;     {Világostürkiz}
      LightRed      = 12;     {Rózsaszín}
      LightMagenta  = 13;     {Világoslila}
      Yellow        = 14;     {Sárga}
      White         = 15;     {Fehér}
```

procedure SetRGBPalette(n, p, z, k: **integer**);

Az ún. RGB módban is működő grafikus adapterek (IBM–8514, VGA) palettáján egy szín módosítása. Az első paraméter (**n**) a módosítandó színpaletta indexét adja meg. Ez 0..255 lehet IBM–8514 és 0..15 VGA esetén. A másik három paraméter (**p**, **z**, **k**) a komponensszíneket határozza meg. VGA adapternél e paraméterekből csak az alsó byte hat felső bitje kerül a palettába.

procedure SetTextJustify(v, f: **word**);

A grafikus szöveg AK-hoz való illesztésének módját állítja be. Az első paraméter (**v**) a vízszintes, a második paraméter (**f**) a függőleges irányú illesztést adja meg.

A vízszintes illesztés konstansai:

```
const LeftText   = 0;      {A szöveg bal széle van AK-nál}
      CenterText = 1;      {A szöveg közepe van AK-nál}
      RightText  = 2;      {A szöveg jobb széle van AK-nál}
```

A függőleges illesztés konstansai:

```
const BottomText = 0;      {A szöveg alja van AK-nál}
      TopText    = 2;      {A szöveg teteje van AK-nál}
```

Hibás paraméter esetén **GraphResult** értéke −11 lesz.

procedure SetTextStyle(bt, ir, m: **word**);

A grafikus szöveg betűtípusát (**bt**), irányát (**ir**) és karaktereinek méretét (**m**) állítja be.

Az egyes betűtípusok (fontok) megadására szolgáló konstansok:

```
const DefaultFont   = 0;    {Alaptípus, 8x8-as pontmátrixban
      TriplexFont   = 1;    definiálva.}
      SmallFont     = 2;    {Rugalmas, nagyításnál is alaktartó}
      SansSerifFont = 3;    {betűtípusok. A pontmátrix mérete}
      GothicFont    = 4;    {a karakterrel együtt változik.}
```

Az írás iránya vízszintes vagy függőleges (lentről felfelé) lehet, melyhez szintén konstansok tartoznak:

```
const HorizDir = 0;       {Vízszintes}
      VertDir  = 1;       {Függőleges}
```

m értéke 1 az alaptípusnál és 4 a rugalmas betűtípusoknál. Ez utóbbiak mérete a **SetUserCharSize** eljárással (l. később) szabályozható. Hiba esetén **GraphResult** értéke −8, −9, −11, −12, −13 és −14 lehet.

procedure `SetUserCharSize(mx, dx, my, dy: word);`

A rugalmas betűtípusok X és Y irányú méretét szabályozza. A karakterek szélessége **mx/dx**-szel, magassága pedig **my/dy**-nal szorzódik.

procedure `SetViewPort(x1, y1, x2, y2: integer; v: Boolean);`

Új grafikus ablakot aktivizál, melynek bal felső sarka az (**x1,y1**), jobb alsó sarka az (**x2,y2**) pont. A **v** paraméter **true** értéke az ablakból kilépő vonalak levágását, **false** értéke pedig meghagyását eredményezi. A grafikus ablak alaphelyzete a teljes képernyő. Hiba esetén **GraphResult** értéke −11 lesz.

procedure `SetVisualPage(n: word);`

A grafikus adapter **n**-edik lapját teszi a képernyőre. Csak több lapot kezelő adapterekkel (EGA(256K), VGA és Hercules) használható.

procedure `SetWriteMode(mod: integer);`

A különböző eljárásokkal húzott vonalak képernyőre kerülésének módját állítja be. Ha **mod**=0, a vonal átírja a képernyő előző tartalmát, ha **mod**=1, a vonal pontjai és a képernyő megfelelő pontjai között elvégzett bitenkénti XOR művelet eredménye kerül a képernyőre. A megfelelő konstansok:

```
const CopyPut = 0;        {Felülírás}
      XORPut  = 1;        {XOR}
```

function `TextHeight(s: string): word;`

Értéke az **s** szöveg magassága képpontokban. A függvény figyelembe veszi a betűtípust és a karakterek aktuális méretét.

function `TextWidth(s: string): word;`

Értéke az **s** szöveg hossza képpontokban. A függvény figyelembe veszi a betűtípust és a karakterek aktuális méretét.

3.2.6. A TURBO3 modul

A modul egyetlen célja a Turbo Pascal 3.0-ás változatával való kompatibilitás megőrzése. Ez jól látszik a modul kapcsolati részén is:

```
unit Turbo3;
interface
uses Crt;
var Kbd: text;
CBreak: Boolean absolute CheckBreak;

function IOResult: integer;
function MemAvail: integer;
function Maxavail: integer;
function LongFileSize(var f): real;
function LongFilePos(var f): real;
procedure LongSeek(var f; Pos: real);
procedure HighVideo;
procedure Norm Video;
procedure LowVideo;
```

A **Kbd** állomány a 3.0-ás változat azonos nevű standard külső egységét, a billentyűzetet szimulálja, a **CBreak** pedig az új **CheckBreak** standard változót jelöli a régi névvel.

Az eljárások és függvények részletes leírását mellőzzük, mivel a modul aktualitása egyre csökken. A 3.0-ás változat kézikönyvében megtalálhatók, itt csak igen röviden utalunk tevékenységükre.

procedure HighVideo;
: A tintaszínt sárgára (színes képernyő) vagy fehérre (fekete-fehér képernyő), a háttérszínt pedig feketére állítja.

function IOResult: **integer**;
: Értéke az utolsó B/K művelet hibátlan voltára (0), illetve hibájára utaló egész szám.

function LongFilePos(**var** f): **real**;
: Értéke az állomány aktuális pozíciója valós szám formájában.

function LongFileSize(**var** f): **real**;
: Értéke az állomány elemeinek száma (valós).

procedure LongSeek(**var** f; p: **real**);
: Az állomány mutatóját a **p**-edik elemre állítja.

procedure LowVideo;
: A tintaszínt világosszürkére, a papírszínt feketére állítja.

function MaxAvail: **intger**;
Értéke a dinamikus memóriában rendelkezésre álló legnagyobb összefüggő terület mérete (16 byte-os paragrafusokban).

function MemAvail: **integer**;
Értéke a dinamikus memóriában rendelkezésre álló 16 byte-os pragrafusok száma.

procedure NormVideo;
A tintaszínt sárgára (színes képernyő) vagy fehérre (fekete-fehér képernyő) a háttérszínt pedig feketére állítja. (Hatása azonos a **HighVideo** eljárásával.)

3.2.7. A GRAPH3 modul

Ez a modul a Turbo Pascal 3.0-ás változatban szereplő teknőcgrafikai alrendszer új, de változatlan szolgáltatásokat nyújtó megvalósítása. A modul négy konstanst és számos eljárást, illetve függvényt tartalmaz. Az eljárások és függvények részletes leírását mellőzzük, mivel a modul aktualitása egyre csökken. A 3.0-ás változat kézikönyvében megtalálhatók, itt csak igen röviden utalunk tevékenységükre.

A konstansok az égtájak nevei:

```
const North = 0;
      East  = 90;
      South = 180;
      West  = 270;
```

Az eljárások és függvények különböző grafikai alaptevékenységeket, illetve a teknőc irányítását és mozgatását végzik.

procedure Arc;
Körívet rajzol.

procedure Back;
A teknőcöt hátrafelé mozgatja.

procedure Circle;
Kört rajzol.

procedure ClearScreen;
Törli az aktív ablakot és a teknőcöt az ablak közepére állítja.

procedure ColorTable;
Egy színtranszformációs táblát definiál, ami újrarajzoláskor érvényesül.

procedure Draw;
Két pontot összekötő egyenest rajzol.

procedure `FillPattern;`
Egy téglalap alakú területet tölt ki az aktuális mintával és adott színnel.

procedure `FillsScreen;`
A teljes aktuális ablakot kitölti az adott színnel.

procedure `FillShape;`
Tetszőleges alakú területet tölt ki az adott színnel.

procedure `Forwd;`
A teknőcöt előre mozgatja.

function `GetDotColor;`
Értéke egy képpont színkódja.

procedure `GetPic;`
A képernyő adott részének tartalmát egy pufferbe másolja.

procedure `GraphBackground;`
Beállítja a háttérszínt.

procedure `GraphColorMode,`
A CGA adapter 320x200-as grafikus módját állítja be.

procedure `GraphMode;`
320x200-as fekete-fehér grafikus módot állít be.

procedure `GraphWindow;`
Grafikus ablak megnyitása.

function `Heading;`
Értéke a teknőc pillanatnyi iránya.

procedure `HideTurtle;`
A teknőcöt láthatatlanná teszi.

procedure `HiRes;`
640x200-as felbontású grafikus módot állít be.

procedure `HiResColor;`
Beállítja a nagyfelbontású grafikus módhoz tartozó színt.

procedure `Home;`
A teknőcöt az aktív ablak közepére állítja.

procedure `NoWrap;`
A teknőc csak ablakhatárig közlekedhet.

procedure `Palette;`
Az adott palettát aktivizálja.

procedure `Pattern;`
Egy 8x8-as bitmintát, a töltőmintát definiálja.

procedure `PenDown;`
A teknőc leteszi „tollát” a papírra. Ha ezután mozog, nyomot hagy.

procedure PenUp;
A teknőc felemeli „tollát". Ha ezután mozog, nem hagy nyomot.

procedure Plot;
Egy képpontot rajzol adott színnel.

procedure PutPic;
A pufferben tárolt képet (l. GetPic) a képernyőre másolja.

procedure SetHeading;
A teknőcöt egy szöggel adott irányba fordítja.

procedure SetPenColor;
Beállítja a teknőc tintájának színét.

procedure SetPosition;
A teknőcöt vonalhúzás nélkül egy adott koordinátájú pontra viszi.

procedure ShowTurtle;
A teknőcöt láthatóvá teszi.

procedure TurnLeft;
A teknőcöt adott szöggel balra fordítja.

procedure TurnRight;
A teknőcöt adott szöggel jobbra fordítja.

procedure TurtleDelay;
A teknőc lépései közé szünetet iktat.

function TurtleThere;
Értéke true, ha a teknőc az aktív ablakban van és látható.

procedure TurtleWindow;
A teknőc számára nyit ablakot a képernyőn.

procedure Wrap;
A teknőc kiléphet az aktív ablakból (ez esetben visszajön a szemközti oldalon).

function XCor;
Értéke a teknős aktuális X koordinátája.

function YCor;
Értéke a teknős aktuális Y koordinátája.

4. Objektumok

A programozás régóta közismert és legjobban talán Wirth által megfogalmazott alapelve, hogy a programok adatszerkezetekből és algoritmusokból épülnek fel. Ebben a szereposztásban az adatszerkezetek statikus, **passzív** szerepet játszanak, úgy képzelhetők el, mint címkézett (esetenként egymásba skatulyázott) fiókok, melyekbe – későbbi felhasználásra – különböző típusú értékek helyezhetők el. Az elhelyezést és a felhasználást az eredendően aktív programok végzik.

Ezzel a szemlélettel kialakult és elterjedt a programozás korszerű módszertana, amit **strukturált programozás** néven ismerünk. E módszertan továbbfejlesztése az adatszerkezetek passzivitásának felszámolása, ami az új adattípusban, az **objektumokban** ölt testet.

Az objektum fogalmának lényege az a felismerés, hogy a környező világ dolgai jobban modellezhetők számítástechnikai eszközökkel, ha jellemző adataikat és működési módszereiket nem egymástól elválasztva, hanem egységes egészként kezeljük. Ezzel elérhető, hogy – mint a rekordok esetében csak az adatmezők – az objektumok esetében bizonyos **módszerek** is tartoznak az adatszerkezethez. A Pascal nyelvben a módszereket, kézenfekvő módon, eljárások és függvények alkotják.

Kis túlzással élve azt mondhatjuk, hogy az objektumokban az adatszerkezetek életre kelnek, és mint színészek egy darabban, különböző szerepek játszására válnak alkalmassá. A szerepeket a módszerek képviselik, a rendező feladatát pedig az alkalmazási program játssza. Az ily módon „megszemélyesített" adatszerkezetek jelentősége sokkal nagyobb, mint a hagyományos, statikus szemléletnél.

A Turbo Pascal 5.5-ös változatában az objektumokat a rekordtípusok fogalmának többirányú bővítésével célszerű bevezetni. Az első lépést és egyben az objektumok fent leírt tulajdonságát **egyesítés**nek (az eredeti szóhasználat szerint: encapsulation) nevezzük.

4.1. Egyesítés

Az adatok és a hozzájuk tartozó módszerek egyesítése formailag egyszerű: a rekordtípus mintájára bevezetjük az **object** nevű új típust, amely olyan rekord, melyhez nemcsak adatmezők, hanem eljárások és függvények formájába öntött módszerek is tartozhatnak.

Tekintsük például az **Ember** nevű típust, ami – eddigi ismereteink alapján – a nevet és a személyi számot tartalmazó rekord:

```
type Ember = record Nev: string[40];
                    Szsz: string[13]
             end;
```

Ha **e1 Ember** típusú változó, az adatmezők értéket kaphatnak például az

```
e1.Nev :='Kis György';
e1.Szsz:='1-123156-0010';
```

utasításokkal.

Ezek csak egy konkrét változóra érvényesek, ha több **Ember** típusú változóval dolgozunk, érdemes lehet egy eljárást deklarálni, amely elvégzi az értékadásokat:

```
procedure Legyen(var e: Ember; N, Sz: string);
begin with e do
          begin  Nev:=N;
                 Szsz:=Sz
          end
end;
```

Az eljárás segítségével az értékadás

```
Legyen(e1, 'Kis György','1--123156--0010');
```

alakú lesz. Ez – különösen többszörös használat esetén – egyszerűbb a közvetlen értékadásoknál, de kizárólag **Ember** típusú rekordokra működik.

Ha az eljárást egyesítjük az adatleírással, eljutunk az új típushoz, az objektumhoz.

```
type Ember = object Nev: string[40];
                    Szsz: string[13];
                    procedure Legyen(n, sz: string);
             end;
```

A típusdeklarációban, mint látható, csupán az eljárás feje szerepel, a törzset a **forward**, illetve a modulok **interface** és **implementation** részében is használt, ismert technikával később, a többi eljárásdeklarációval együtt kell megadni a rekordmezőknél megszokott hivatkozási mód alkalmazásával:

```
procedure Ember.Legyen(N, Sz: string);
begin Nev:=N;
      Szsz:=Sz
end;
```

Az eltérés annyi, hogy az első paraméter, ami csak egy **Ember** típusú változó neve lehetett, ilyen minőségében megszűnt, „elő lépett", bekerült az eljárás nevébe. Ettől az eljárás törzse kicsit egyszerűbb lett. Ha most deklarálunk egy **e1** változót:

```
var e1: Ember;
```

automatikusan rendelkezésünkre áll a hozzá tartozó módszer **e1.Legyen** néven (mintha egy rekord egyik mezője lenne).

A konkrét hívás pedig például

```
e1.Legyen('Kis György','1--561231--0010');
```

alakú lehet, ami a **with** utasítással is használható a szokott módon.

Az egyesítés több előnnyel jár, amelyekre később visszatérünk. Most csak arra hívjuk fel a figyelmet, hogy a módszernek az objektum leírásába való beépítésével megtartottuk azt a lehetőséget, hogy egy másik objektumban szintén használjunk egy **Legyen** nevű módszert (ami esetleg hasonlít, de akár teljesen eltérhet az elsőtől). Erre a „hagyományos" módszer nem adott lehetőséget.

Az objektumok a Turbo Pascalban megengedik az adatmezőkre való közvetlen hivatkozást is (**e1.Nev**, illetve **e1.Szsz** formában). Az ilyen hivatkozások azonban ellenkeznek az objektumok alkalmazását előtérbe helyező, ún. **Objektum Orientált Programozás** (OOP) alapelveivel. A most itt részletesen ki nem fejtett indokok hasonlóak a strukturált programozással kapcsolatban elhangzott **goto** ellenes okfejtésekhez. Egyes objektum orientált nyelvek (mint pl. a SMALLTALK) kifejezetten tiltják az objektumok adatmezőire való közvetlen hivatkozást. A Pascal alapelveihez hűen (amelyek a **goto** használatát sem zárják ki) csak annyit mond, hogy az OOP nem szigorú szabályok gyűjteménye, hanem egyfajta programozási szemlélet, melynek elveit a programozó saját érdekében (és nem kényszer hatására) tartja szem előtt.

Az OOP elvei szerint minden objektumhoz meg kell adni azokat a módszereket, amelyek az objektum alkalmazásánál szóba jöhetnek. A módszerek deklarálásának **sorrendje közömbös**, de csak az objektum **adatmezői**nek leírása **után** kerülhetnek sorra. Ennek megfelelően a fenti **Ember** objektumot a következő módszerekkel bővíthetjük, ha arra számítunk, hogy szükségünk lesz az egyes adatmezők értékére is:

```
type Ember = object Nev: string[40];
                    Szsz: string[13];
                    procedure Legyen(n, sz: string);
                    function Neve: string;
                    function Szszama: string;
             end;
```

A megfelelő függvények tényleges deklarációja igen egyszerű:

```
function Ember.Neve: string;
begin Neve:=Nev
end;
```

illetve:

```
function Ember.Szszama: string;
begin Szszama:=Szsz
end;
```

Ezután az **e1.Neve**, illetve az **e1.Szszama** függvények értéke használható a konkrét mezőnevek helyett.

Azt a lehetséges ellenvetést, hogy az objektum tervezésekor nem lehet az összes szükséges módszert előre látni, a Turbo Pascal úgy védi ki, hogy minden szóba jöhető módszer deklarálására bátorít, de a fordítás során **nem generál kódot** az egyáltalán nem használt módszerekből. Ily módon a programot nem terhelik felesleges, soha nem hívott eljárások. Azt pedig, hogy egy objektum esetében egy módszer egyáltalán szóba jöhet-e, már a tervezés során illik eldönteni.

Az adatok és módszerek egyesítésének elvi alapját az a megfontolás adja, hogy a programozó az adatra és a módszerre mint **egyetlen egységre** gondoljon már a program tervezésekor is. Ha ezek külön egységek, fennáll a veszélye annak, hogy egy eljárást nem a hozzá illő paraméterrel hívunk vagy nem a megfelelő eljárást hívjuk. Az eljárások és az adatok jó eredményt adó kombinációinak kiválasztása a programozó feladata, s ebben a Pascal viszonylag szigorú típusellenőrzése csak annyit segít, hogy megmondja, mit **nem lehet** párosítani.

A fejlesztés, majd később a karbantartás során elkerülhetetlen módosítások kivitelezése sokkal biztonságosabb az objektumok, mint a rekordok és a külön deklarált eljárások esetén, különösen akkor, ha egy-egy eljárás nem csak egyetlen adatszerkezetet „szolgál ki".

Az objektumok alkalmazása kiválóan harmonizál a Turbo Pascal moduljainak használatával. Az objektumokat célszerű **unit**-okban deklarálni: az objektum típusát a kapcsolódási, az eljárásokat pedig a megvalósítási részben. A modul megvalósítási részében természetesen tartalmazhat saját objektumdeklarációkat is, ezek külső modulok számára hozzáférhetetlenek lesznek. Ha a fenti egyszerű példát egy modulba foglaljuk össze, a következőt kapjuk:

```
unit Emberek;

interface

type Ember = object Nev: string[40];
                    Szsz: string[13];
                    procedure Legyen(n, sz: string);
```

```
                    function Neve: string;
                    function Szszama: string;
            end;

implementation

procedure Ember.Legyen(N, Sz: string);
begin Nev:=N;
      Szsz:=Sz
end;

function Ember.Neve: string;
begin Neve:=Nev
end;

function Ember.Szszama: string;
begin Szszama:=Szsz
end;

end.
```

Figyeljük meg, hogy a módszerek törzsében nem kell explicit **with...do** utasítást használni, mert egy objektum módszereiben annak adatmezői szabadon elérhetők. Ez úgy is felfogható, hogy az objektum egy implicit hívással hívja valamelyik módszerét:

```
. with objektumnév do módszer
```

Ez valóban így történik, a módszer hívásakor egy előre deklarált, implicit, mutató típusú paraméter is átadódik, amelynek neve **Self** és magát a módszert birtokló objektumot képviseli.

A **Self** explicit használatára általában nincs szükség, de előfordulhatnak olyan névütközések, melyek feloldásában hasznos lehet. Tekintsük például a következő deklarációt:

```
type R1  = record m1: ...;
                  m2: ...;
                  X: ...;
           end;

     OB1 = object obm1: ...;
                  obm2: ...;
                  X: ...;
                  procedure obp1(rek: R1);
           end;
```

```
. procedure OB1.obp1(rek: R1);
. begin with R1 do
.             begin obm1:=m1;
.                   obm2:=m2;
.                   Self.X:=X
.             end
. end;
```

Itt a **Self** biztosítja, hogy az **OB1** objektum **X** mezőjére vonatkozzék a with utasításon belüli utolsó értékadás. A **Self** használata itt egyszerűen elkerülhető, ha lemondunk a **with** utasításról, azonban ez némely esetben kényelmetlenséget okozhat. A **Self** használatát az OOP alapelvei elkerülendőnek minősítik, a Turbo Pascal most is a programozóra bízza a döntést, elismerve ezzel, hogy egyes helyzetekben a **Self** jelentheti a kisebbik rosszat.

Annak, az egyébként igen hasznos ténynek, hogy az objektum saját hatáskörébe vonja módszereit, egyenes következménye, hogy a módszerek formális paraméterei nem lehetnek azonosak az objektum egyik adatmezőjével sem. Ez ugyanis ugyanolyan hibát eredményezne, mintha egy eljárás valamelyik formális paramétere megegyezne az egyik lokális változójával.

Az egyesítés az objektumok fontos, de nem egyetlen alaptulajdonsága. A környezet tárgyainak és jelenségeinek leírásakor a tudományok gyakran alkalmazzák a fogalmak egymásból származtatott leírását, az ún. **taxonómiát**. Ezt teszik a biológusok a fajok, az orvosok a betegségek, a nyelvészek a nyelvtanok, a matematikusok az egyes alakzatok leírásának során. Az objektumok világában a taxonómia megfelelője az **öröklődés**.

4.2. Az öröklődés

Mivel az alapvetően különböző adatszerkezetek száma nem nagy (skalárok, tömbök, halmazok, rekordok és ezek kombinációi), kézenfekvő az objektumok egymásból származtatott leírásának bevezetése.

Ez azt jelenti, hogy valamilyen egyszerű objektumból kiindulva, egymásra épülő sorozatot hozunk létre olymódon, hogy minden újabb szinten azt vizsgáljuk, az új objektum **miben egyezik meg** az előző szinten lévővel, és **milyen új tulajdonságai** lesznek. Az öröklődés (eredeti angol szóhasználat szerint inheritance) lényege, hogy ha egyszer valamilyen tulajdonságot deklaráltunk, azzal minden utód (nemcsak a közvetlen, hanem a távolabbiak is) rendelkezni fog.

Az öröklődés az OOP számára lehetővé teszi, hogy adatszerkezetek családfáit építsük fel az egyszerűbb és ezért általánosabb típusoktól a bonyolultabb és ezért specifikusabb típusok felé haladva.

Az öröklődés jelzésére az objektum deklarációjában igen egyszerű eszközök szolgálnak. Tekintsük a már ismert **Ember** nevű objektumunkat:

```
type Ember = object Nev: string[40];
                    Szsz: string[13];
                    procedure Legyen(n, sz: string);
                    function Neve: string;
                    function Szszama: string;
             end;
```

Tegyük fel, hogy valamilyen utazással kapcsolatos feldolgozáshoz a néven és személyi számon kívül még az emberek útlevélszámára is szükség van.

Az új objektumot a már létező **Ember**-ből származtatjuk a következő módon:

```
Utas = object(Ember) Ulsz: string[12];
       end;
```

A zárójelben álló objektumnév azt fejezi ki, hogy az új, **Utas** nevű objektum örökli az **Ember** minden tulajdonságát, azaz adatmezőit és módszereit. Ebben a kapcsolatban **Ember** az **Utas** közvetlen elődje és **Utas** az **Ember** közvetlen utódja. A taxonómia alaptulajdonsága, hogy bármely objektumnak akárhány utódja, de legfeljebb csak egy közvetlen elődje lehet (az **Ember**-nek egy sincs).

Az **Utas** jelenlegi formájában csupán egyetlen új adatmezővel (az **Ulsz**-el) bővítette az **Ember**-t, vizsgáljuk meg azonban, mit jelentett számára az öröklődés. Ehhez deklaráljunk egy **Utas** típusú változót:

```
var u1: Utas;
```

Az OOP terminológia szerint **u1** az **Utas** egy **példánya**, amely a következő mezőkkel és módszerekkel rendelkezik:

```
u1.Nev
u1.Szsz
u1.Utl

u1.Legyen
u1.Neve
u1.Szszama
```

Az első két adatmező és mindhárom módszer öröklött.

Gyakorlati szempontok az **Utas** objektum bővítését igénylik, hiszen jelenleg a **Legyen** eljárás csak a név és a személyi szám értékét állítja be, az útlevélszámét nem. Ezenkívül szükség lehet az útlevélszám lekérdezésére, valamint a teljes objektum képernyőn való megjelenítésére is.

Ezekkel a kiegészítésekkel az **Utas** deklarációja a következőképpen alakul:

```
type Utas=object(Ember) Ulsz: string[12];
                        procedure Legyen(n,sze,szu:string);
                        function Ulszama: string;
```

```
                        procedure Irdki;
          end;

procedure Utas.Legyen(n, sze, szu: string);
begin Ember.Legyen(n, sze);
      Ulsz:=szu
end;

function Utas.Ulszama: string;
begin Ulszama:=Ulsz
end;
procedure Utas.Irdki;
begin writeln('Név: ',nev);
      writeln('Személyi Szám: ',Szsz);
      writeln('Utlevél szám: ',Ulsz);
end;
```

Figyeljük meg, hogy a **Legyen** eljárást újradefiniáltuk az **Utas**-ban és a törzsében használjuk az eredeti módszert is. A módszerek újradefiniálása (és az elől-módszer ebben való hasznosítása) az utódokban nemcsak megengedett, hanem az OOP filozófiába jól illeszkedő gyakorlat, ami a pontos öröklődést biztosítja.

Az újradefiniálás azonban **csak módszerekre** vonatkozhat, **adatmezőkre nem**, az ősök adatmezői változatlanul öröklődnek.

Végül képezzünk újra modult a fenti objektumdeklarációkból, és készítsünk egy egyszerű programot ezek mozgatására:

```
unit Emberek;

interface
uses crt;
type ember=object Nev: string[40];
                  Szsz: string[11];          {Személyi szám}
                  procedure Legyen(n, sz: string);
                  function Neve: string;
                  function Szszama: string;
           end;

     Utas=object(ember) Ulsz: string[12];      {Útlevélszám}
                        procedure Legyen(n,sze,szu:string);
                        function Ulszama: string;
                        procedure Irdki;
          end;
```

```
implementation
procedure Ember.Legyen(n, sz: string);
begin Nev:=n;
      Szsz:=sz
end;

function Ember.Neve: string;
begin Neve:=Nev
end;

function Ember.Szszama: string;
begin Szszama:=Szsz
end;

procedure Utas.Legyen(n, sze, szu: string);
begin Ember.Legyen(n, sze);
      Ulsz:=szu
end;

function Utas.Ulszama: string;
begin Ulszama:=Ulsz
end;

procedure Utas.Irdki;
begin writeln('Név: ',Nev);
      writeln('Személyi Szám: ',Szsz);
      writeln('Utlevél szám: ',Ulsz)
end;

end.

program oop1;

uses crt, emberek;

var u1, u2: utas;

begin clrscr;
      with u1 do
           begin Legyen('Kiss Gyuri','15612310010','AB122');
                 irdki; writeln;
                 writeln(Neve,' ',Szszama,' ', Ulszama);
           end;
```

```
        u2:=u1;                          {Ilyen direkt értékadást}
        u2.Nev:='Nagy Lajos';                  {vérbeli OOP hívő}
        writeln;                  {messze elkerül, inkább az objektum-}
        u2.irdki                    {deklarációt bővíti egy módszerrel.}
    end.
```

Az öröklődéssel kapcsolatban foglalkozni kell a **típusok** kompatibilitási kérdéseivel is, mivel a Pascal eredetileg igen szigorú szabályai itt egy kicsit lazulni látszanak. Alapszabály, hogy az **utódtípus örökli az összes elődjével való kompatibilitást**. Ez a „bővített" kompatibilitás három formában jelentkezhet:

* az objektumok példányai között,
* az objektumok példányaira vonatkozó mutatók közt,
* a módszerek formális és aktuális paraméterei között.

Mindhárom formánál szem előtt kell tartani, hogy a kompatibilitás csak egyirányú, **az utód kompatibilis az ősével**, azaz az utódtípusok szabadon használhatók az őseik helyett, de fordítva ez általában nem igaz.

A fenti példánál maradva, deklaráljunk néhány példányt az **Ember** és az **Utas** objektumból:

```
type emut: ^Ember;
     umut: ^Utas;

var e1, e2: Ember;
    u1, u2: Utas;
    ep: emut;
    up: umut;

procedure p1(x: Ember);
begin ...
...
end;
```

Ekkor a következő értékadások megengedettek:

```
e1:=e2;                          {Ez öröklődés nélkül is igaz volt}
e1:=u2;                                {De u2:=e1 hibás lenne}
ep:=up                                 {De up:=ep hibás lenne}
```

És hasonlóképpen korrekt a

```
p1(u1);
```

eljáráshívás is.

Az egyirányú kompatibilitás magyarázatául elmondható, hogy értékadáskor biztosítani kell, hogy a célterület **egyetlen mezője se** maradjon definiálatlan. Ez csak akkor teljesül, ha a forrásterület a célterület utódja, mert

ekkor annál szűkebb nem lehet (általában ténylegesen bővebb). Természetesen a bővebb forrásterületről csak a célterülettel közös mezők másolódnak át.

Ezzel elmondtunk minden lényegeset az adatmezők és valamennyit a módszerek öröklődéséről. Ami azonban a módszerekkel kapcsolatban hátra van, kis túlzással az OOP „sava-borsának" is nevezhető.

Azt, hogy az adatmezőkre és a módszerekre teljesen azonos öröklési szabályok alkalmazhatók, nem várhatjuk, hiszen az egyik röghöz kötött, statikus, a másik pedig skatulyázott (esetleg rekurzív) hívási lehetőségeket tartalmazó dinamikus fogalom. Az öröklődés eddig megismert módját nyugodtan nevezhetjük statikusnak, hiszen az elől tulajdonságai módosulás nélkül kerülnek át az utódba.

A módszerek statikus örökítésével kapcsolatos problémát legegyszerűbben egy példával szemléltethetjük.

A fenti **Emberek** modult módosítsuk úgy, hogy mind az **Utas**, mind az **Ember** típust bővítjük a **Kerdez** módszerrel, amely megkérdezi és a billentyűzetről beolvassa a nevet és a személyi számot, majd a **Legyen** hívásával ezeket tárolja. Tegyük fel továbbá, hogy az **Utas** típushoz tartozó útlevélszámot nem kell külön megadni, hanem azt a név első két betűjéből és a személyi számból egyszerű konkatenációval nyerjük (az **Utas.Legyen** módszer is módosul).

Az **Emberek** új változata a következő:

```
unit Emberek;

interface
uses crt;
type ember = object Nev: string[40];
                    Szsz: string[11];          {Személyi szám}
                    procedure Legyen(n, sz: string);
                    function Neve: string;
                    function Szszama: string;
                    procedure Kerdez;
             end;

     Utas = object(ember) Ulsz: string[12];    {Útlevélszám}
                          procedure Legyen(n,sze:string);
                          function Ulszama: string;
                          procedure Irdki;
                          procedure Kerdez;
            end;

implementation
procedure Ember.Legyen(n, sz: string);
begin Nev:=n;
```

```
        Szsz:=sz
end;

function Ember.Neve: string;
begin Neve:=Nev
end;

function Ember.Szszama: string;
begin Szszama:=Szsz
end;

procedure Ember.Kerdez;
var n, s: string;
begin clrscr;
        write('Név: '); readln(n);
        write('Személyi Szám: '); readln(s);
        Legyen(n,s)
end;

procedure Utas.Legyen(n, sze: string);
begin Ember.Legyen(n, sze);
        Ulsz:=copy(n,1,2)+sze        {Az útlevélszám a név első két}
end;                                 {betűjéből és a személyi számból áll}

function Utas.Ulszama: string;
begin Ulszama:=Ulsz
end;

procedure Utas.Irdki;
begin writeln('Név: ',Nev);
        writeln('Személyi Szám: ',Szsz);
        writeln('Utlevél szám: ',Ulsz)
end;

procedure Utas.Kerdez;
var n, s: string;
begin clrscr;
        write('Név: '); readln(n);
        write('Személyi Szám: '); readln(s);
        Legyen(n,s)
end;

end.
```

Ez a modul jelen formájában kifogástalanul működik, mindössze egy „szépséghibája" van: az **Ember.Kerdez** és az **Utas.Kerdez** módszerek a nevüktől eltekintve pontosan azonosak. Jogos tehát a kérdés: miért nem bízzuk ezt is az öröklésre, mint pl. a **Neve** és **Szszama** függvények esetén?

Ha ezt megpróbáljuk, azaz elhagyjuk a modulból az **Utas.Kerdez** módszert, formailag ismét helyes programhoz jutunk, ami azonban nem azt csinálja, amit elvárnánk tőle. Ha az utas egy **u1** példányával behívnánk az eljárást (**u1.Kerdez** formában), majd kiíratnánk a kapott eredményt (**Irdki**), azt tapasztalnánk, hogy az **útlevélszám generálása elmaradt**. Ennek okát a **Legyen** módszerben kell keresni, hiszen ez helyezi el a beolvasott értékeket az adatmezőkben. Mivel a **Kerdez** deklarációja az **Ember**-hez kötődik, a benne szereplő **Legyen** hívás is tulajdonképpen az **Ember.Legyen** módszer aktivizálását jelenti, és ezt a hívást a Pascal fordítóprogram **statikusan** generálja, tehát az öröklésekor is megmarad. Az **Ember.Legyen** viszont láthatólag nem foglalkozik az útlevélszámmal. Ezért kellett a **Kerdez**-t – bár változatlan formában – újradefiniálni, mert az **Utas.Kerdez**-ben szereplő **Legyen** már nem **Ember.Legyen**, hanem **Utas.Legyen**, és ez kezeli az útlevélszámot is.

Az öröklődés tehát eredendően **statikus jelenség**, amely módszerek örökítése esetén nem képes észrevenni, hogy az öröklött módszer által hívott valamelyik módszert az utód objektumban újradefiniálták. Ez talán önmagában még nem teszi használhatatlanná az egész OOP-t, de mindenesetre olyan terhet ró a programozóra, ami mindenkit elijeszt a gyakorlati alkalmazástól. A megoldást természetesen kidolgozták és az több néven is ismert; leggyakrabban **polimorfizmusnak** nevezik, de a Pascal irodalomban használják a **virtuális** módszerek kifejezést is.

4.3. Polimorfizmus

Ha jobban átgondoljuk az objektumok kompatibilitásának szabályait, fennakadhatunk a formális és aktuális paraméterekre vonatkozó kitételen, ami kimondja, hogy a formális paraméter helyére annak bármely utódtípusához tartozó aktuális paraméter helyettesíthető. Ez más szavakkal azt jelenti, hogy fordításkor az eljárás még nem „tudhatja", milyen paraméterrel fogják majd behívni, ezért a statikus (fordítási időben megoldott) paraméterkezelés helyett a módszerek paramétereit **polimorfikus** dolgokként, dinamikusan kell kezelni.

A paraméterek polimorfikus kezelése azt jelenti, hogy az eljárás és a paraméter összekapcsolását a fordítási időből későbbre, a program **futási idejére** helyezzük át, amikor már ismert a paraméter konkrét típusa.

A módszerek polimorfikus kezelése azt jelenti, hogy az objektumok hierarchiájában közös névvel szereplő eljárások mindegyikének megadjuk a lehetőséget arra, hogy törzsét saját környezetének megfelelően értelmezze. Ez szintén azt jelenti, hogy a polimorfikusan kezelt módszerek törzsének

fordítását nem szabad a futás előtt „befagyasztani", mert az a fenti példában mutatott fogyatékossághoz vezet. Az ilyen módszerek törzse csak futási időben és konkrét környezetében kaphatja meg tényleges értelmét.

A standard Pascal számára merőben új elvek és módszerek a nyelv szókincsének bővítését is megkövetelték. A polimorfikus eljárásokat a Turbo Pascal **virtuális** módszereknek nevezi az egyéb, **statikus** módszerektől való megkülönböztetés céljából. A virtuális módszerek kezelése bonyolultabb, ezért az objektumokban deklarált módszerek közül csak azok lesznek virtuálisak, melyek deklarációjában (a törzs helyett) az új lefoglalt szó, **virtual** áll.

Ha egy módszert virtuálisnak deklaráltunk, az összes utódtípusban **minden** vele azonos nevű módszert is virtuálisnak kell nyilvánítani, és ezek fejrésze sem változhat. (A statikus módszereknél ilyen megkötések nincsenek).

A virtuális módszerek kezelése egy speciális táblázat a VMT (Virtual Method Table) segítségével történik. A VMT-t minden objektumtípushoz a fordítóprogram állítja fel, de kitöltésére (a többi között olyan mutatókkal, amelyek a virtuális módszer aktuálisan végrehajtandó tárgyprogramjára hivatkoznak) csak futási időben kerül sor. Az objektum **minden példányát** hozzá kell kapcsolni a VMT-hez, hogy az abban szereplő virtuális módszerek hívásakor a megfelelő információ az objektum rendelkezésére álljon.

Az objektumok és a VMT közötti kapcsolat felépítésére egy speciális eljárás szolgál, amit **konstruktornak** neveznek. Ennek deklarációja a **procedure** szó helyett a **constructor**, új lefoglalt szóval kezdődik.

Minden olyan objektumtípushoz, amely virtuális módszert tartalmaz, egy konstruktor deklarálása kötelező. Az OOP-gyakorlatban javasolt konvenció, hogy a konstruktor neve **Init** legyen. A konstruktor speciális szerepe mellett közönséges eljárás, amelyben általában az objektum mezőinek kezdőértékét adják meg. Ha ilyen nincs, a törzs üres is lehet.

A konstruktor használatára két alapvető szabály vonatkozik:

* A konstruktort bármely virtuális módszer aktivizálása **előtt** kell hívni.
* Az objektum **minden** példányát **külön** konstruktorhívással kell „felavatni" (nem elegendő pl. egy már felavatott objektumot értékül adni az új objektumnak).

E szabályok betartását a Pascal rendszer nem képes ellenőrizni, megsértésük általában hibaüzenet nélkül a gép teljes „lefagyását" eredményezi.

A programfejlesztés folyamán azonban a {$R+} fordítóprogram-direktíva segítségével bekapcsolhatjuk az értékek érvényességének ellenőrzését. Mivel egy felavatatlan objektum virtuális módszerének hívása egy, a programból definiálatlan helyre vezető vezérlésátadást eredményez (a ki nem töltött VMT tábla miatt), a direktíva hatására futási idejű hibajelzést kapunk. Ez az ellenőrzés természetesen növeli a program méretét és lassítja működését, ezért a végleges változatból célszerű eltávolítani, illetve a fejlesztés folyamán is csak a program kritikus részein bekapcsolni. A direktíva kikapcsolása ({$R−}) a programban bárhol végrehajtható.

A polimorfizmus egyszerű bemutatására alakítsuk át az **Emberek** modult, tegyük feleslegessé a **Kerdez** módszer duplikálását. A **Kerdez**-t azért kellett az **Ember** és az **Utas** objektumban egyaránt szerepeltetni, mert a statikus módszerként deklarált **Legyen**-t csak egyféle értelmezésben (ahogy azt az **Ember** objektum előírta) tudta használni. Ha a **Legyen**-ből virtuális módszert csinálunk, az **Ember.Kerdez**-ben automatikusan az **Ember.Legyen**, az **Utas.Kerdez**-ben az **Utas.Legyen** hajtódik végre.

```
unit Emberek;

interface
uses crt;

type ember = object Nev: string[40];
                    Szsz: string[11];        {Személyi szám}
                    Constructor Kezd;
                    procedure Legyen(n,sz:string);virtual;
                    function Neve: string;
                    function Szszama: string;
                    procedure Kerdez;
             end;

     Utas = object(ember) Ulsz: string[13];
                          procedure Legyen(n, sz: string);
                                    virtual;
                          function Ulszama: string;
                          procedure Irdki;
            end;

implementation

constructor ember.Kezd;
begin
end;

procedure Ember.Legyen(n, sz: string);
begin Nev:=n;
      Szsz:=sz
end;

function Ember.Neve: string;
begin Neve:=Nev
end;
```

```
function Ember.Szszama: string;
begin Szszama:=Szsz
end;

procedure Ember.Kerdez;
var n, s: string;
begin clrscr;
      write('Név: '); readln(n);
      write('Személyi Szám: '); readln(s);
      Legyen(n,s)
end;

procedure Utas.Legyen(n, sz: string);
begin Ember.Legyen(n, sz);
      Ulsz:=copy(n,1,2)+sz        {Az útlevélszám a név első két}
end;                              {betűjéből és a személyi számból áll.}

function Utas.Ulszama: string;
begin Ulszama:=Ulsz
end;

procedure Utas.Irdki;
begin writeln('Név: ',Nev);
      writeln('Személyi Szám: ',Szsz);
      writeln('Utlevél Szám: ',Ulsz)
end;
end.
```

Mint látható, a konstruktort **Kezd**-nek hívjuk. Ez egy üres eljárás, s csupán a VMT töltésére szolgál. Ha kezdőértéket akartunk volna adni az objektumok mezőinek vagy egyéb bevezető tevékenységet akartunk volna végrehajtani, ezt elhelyezhettük volna a **Kezd** törzsében.

Csak egy **Kezd** konstruktort deklaráltunk az **Ember**-ben, s ezt az **Utas** úgyis örökli.

A **Kerdez**-ből most már egy példány is elég, a többit a polimorfizmus elintézi (ha nem is ingyen).

A modul használatával kapcsolatban annyit kell megjegyezni, hogy ha a főprogramban deklaráljuk az **Utas** egy példányát

```
var u1: Utas;
```

formában, azt legjobb azonnal, de legkésőbb az **u1.Kerdez** hívás előtt felavatni az

```
u1.Kezd;
```

utasítással. Ennek elmaradása végzetes következményekkel járhat.

A moduláris programozással kombinált OOP lehetővé teszi, hogy a fenti **Emberek** modul lefordított (**EMBEREK.TPU**) változatának módosítása nélkül tovább bővítsük az objektumok hierarchiáját.

Tegyük fel, hogy egy alkalmazásnál az **Utas** objektum egy bővített változatára van szükség, amely tartalmazza az **Utas** szállodai szobaszámát is. E célból bevezetjük a **vendég** objektumot, amely az **Utas**-ra épül, azt egy egész típusú, a szobaszámot tartalmazó adatmezővel bővíti, valamint saját céljainak megfelelően újradefiniálja a **Legyen** és az **Irdki** eljárást. A **Legyen** az elődökre tekintettel kötelezően virtuális. A módosításokat tartalmazó program a következő:

```
program oop2;
uses crt, emberek;
type vendeg = object(utas) Szosz: integer;        {Szobaszám}
                           procedure Legyen(n,sz:string);
                                     virtual;
                           procedure Irdki
              end;

procedure vendeg.Legyen(n, sz: string);
var i, h: integer;
begin Utas.Legyen(n, sz);
      val(copy(sz,length(sz)-2,3),i,h);          {A szobaszám
      Szosz:=i                {a személyi szám utolsó három jegye.}
end;

procedure vendeg.Irdki;
begin Utas.Irdki;
      writeln('Szobaszám: ',Szosz)
end;

var v1: vendeg;

begin v1.kezd;
      v1.kerdez;
      writeln;
      v1.irdki
end.
```

Arra a kérdésre, hogy mely módszerek legyenek statikusak és melyeket célszerű virtuálissá tenni, az általános válasz az, hogy legyen **minden módszer virtuális**; s csak olyan esetekben használjunk statikus módszert, ha a futási idő és a tárfoglalás különösen kritikus. Az így nyert **módosíthatóság**

kárpótol az idő- és tárveszteségért. Ez kezdő OOP programozók számára kielégítő lehet, de a gyakorlat során előbb-utóbb felmerülnek olyan kételyek, hogy miért kell a soha újra nem deklarált módszerekhez is VMT-t kitölteni, és ezzel a programot lassítani.

Ennek figyelembevételével a fenti kérdésre a válasz már nem egyértelmű, inkább csak mérlegelési szempontokra hívja fel a figyelmet: a későbbiekben (várhatóan) újradeklarált módszerek közül azokat célszerű virtuálissá tenni, amelyeket az alapobjektum vagy annak valamelyik utódja **környezetfüggően** használ. Ha a programban egyetlen virtuális módszer is előfordul, a VMT már létrejön, a konstruktor hívására szükség van, ilyenkor már nem sokat számít, ha néhánnyal több virtuális módszert használunk, hátha jól jön még egy esetleges későbbi továbbfejlesztésnél. Előfordul, hogy egy objektum, hosszú idővel létrehozása és az őt tartalmazó modul lefordítása után, önálló életet él és felhasználói olyan esetekben alkalmazzák (a megfelelő bővítésekkel), melyekre az eredeti szerző soha nem is gondolt.

A polimorfizmustól búcsúzóul bevezetjük az **absztrakt objektum** fogalmát. Azokat az objektumokat, amelyek egy feladat során ténylegesen használt objektumok vagy objektumcsaládok **közös** tulajdonságait tartalmazzák, absztrakt objektumoknak nevezzük. Az absztrakt objektumokat azért vezetik be, hogy örökölni lehessen tőlük, ezeknek a programban csak **utódaik** vannak, **példányaik** általában nincsenek. A fenti, **oop2** programban szereplő **Utas** egy absztrakt objektum, amelyből a programban ténylegesen használt **vendég** objektumot származtattuk.

4.4. Objektumok a dinamikus tárban

Az objektumok – egyéb adatszerkezetekhez hasonlóan – a Pascal rendszer dinamikus memóriájában (heap) is tárolhatók, és mutatók segítségével kezelhetők. Az így keletkező, ún. **dinamikus objektumok** jelentősége azért nagy, mert az összetett objektumhierarchiákat méretük miatt gyakran nem is célszerű az adatszegmensben, a statikus változók helyén tartani.

Mivel az objektumokhoz (különösen, ha a dinamikus tárba kerülnek) általában egy konstruktoreljárás is tartozik, a Turbo Pascal OOP változatában kiterjesztették a **new** standard eljárást. A kiterjesztés eredményeként a **new** két paraméterrel is hívható (az eredeti használata is természetesen megmaradt), melyek közül az első a megszokott mutató, a második pedig a megfelelő konstruktor nevéből és esetleges aktuális paramétereiből áll.

Az előző példában szereplő **Utas** objektum esetében a kétféle hívás a következő lehetne:

```
var umut = ^Utas;
new(umut);                                    {Az eredeti forma}
new(umut,Kezd);          {Az új forma, nemcsak umut kap értéket,}
          {hanem végrehajtódik a hozzátartozó Kezd konstruktor is}
```

A **new** további általánosításaként, és ez nem csak objektumokkal kapcsolatban igaz, bevezették a **new** függvényként való hívását is.

Ilyenkor paraméterként nem egy mutatót, hanem **annak típusát** kell használni:

```
type uptr = ^@Utas;
cptr = ^char;

var umut: uptr;
cmut: cptr;

umut:=new(uptr);          { umut a létrehozott új objektumra mutat}
cmut:=new(cptr);          { cmut a létrehozott új karakterre mutat}
```

Objektumokkal kapcsolatban a **new** függvény kétparaméteres formája is használható:

```
umut:=new(uptr,Kezd)
```

A dinamikus memória másik, a felszabadítást végző eljárása a **dispose**. Eredeti formájában használata közismert, a fenti, **Utas** típusú objektum által elfoglalt terület felszabadítása a

```
dispose(umut);
```

hívással történhet.

Bonyolultabb objektumszerkezeteknél azonban általában nem elegendő csupán az elfoglalt memória szabaddá tétele, gyakran az objektum által használt mutatókat valamilyen kötött sorrendben szintén „rendbe kell tenni". Ezért célszerű, ha a programozó a dinamikus tárbeli objektumok megszüntetésével kapcsolatos teendőket egy módszerbe foglalja össze.

A Turbo Pascal ezt a módszert – a konstrukturok mintájára – megkülönbözteti a többitől, és a **procedure** helyett a **destructor** lefoglalt szóval való bevezetését írja elő.

A destruktor tehát olyan módszer, amely a dinamikus memória felszabadítását bármilyen egyéb, az objektum törlésekor elvégzendő tevékenységgel kapcsolhatja össze. Egy objektumtípushoz több destruktor is megadható, melyek elvileg akár statikus, akár dinamikus módszerek lehetnek. Mivel azonban általában a különböző objektumok megszüntetéséhez más és más eljárásra van szükség, a **destruktort célszerű virtuális módszerként** megadni, így a polimorfikus objektumok megszüntetésekor mindig a megfelelő destruktor fog működni (a VMT alapján). A Pascal OOP kiterjesztése a destruktor neveként a **Done** azonosítót javasolja.

A destruktorok virtuális módszerként való hatékony alkalmazásához általánosítani kellett a **dispose** eljárást is. A **new** mintájára bevezették a kétparaméteres változatot, melynek formája pl.

```
dispose(umut,Done);
```

lehet, ha feltesszük, hogy az **Utas** típusú objektumhoz tartozik egy **Done** nevű destruktor is. Ha az objektum megszüntetésekor a memória felszabadításán kívül egyéb teendő nincs, a destruktor törzse üres is lehet.

A dinamikus tárban kezelt polimorfikus objektumok gyakorlati bemutatására a már ismert **Emberek** modult használjuk. A modulon két apró, a lényeget nem érintő módosítást végzünk.

Az **Ember** objektum módszereit egy üres törzsű, **Kesz** nevű destruktorral bővítjük, valamint a modulban szereplő **Kezd** nevű konstruktor eddig üres törzsében elhelyezzük a **Kerdez** módszer hívását. A modul ezek után a következőképpen fest:

```
unit Emberek;
interface
uses crt;

type ember = object Nev: string[40];
                    Szsz: string[11];          {Személyi szám}
                    Constructor Kezd;
                    destructor Kesz;
                    procedure Legyen(n,sz:string);virtual;
                    function Neve: string;
                    function Szszama: string;
                    procedure Kerdez;
             end;

     Utas = object(ember) Ulsz: string[13];
                          procedure Legyen(n,sze:string);
                                    virtual;
                          function Ulszama: string;
                          procedure Irdki;
            end;

implementation

procedure Ember.Legyen(n, sz: string);
begin Nev:=n;
      Szsz:=sz
end;

function Ember.Neve: string;
begin Neve:=Nev
end;
```

```
function Ember.Szszama: string;
begin Szszama:=Szsz
end;

procedure Ember.Kerdez;
var n, s: string;
begin clrscr;
      write('Név: '); readln(n);
      write('Személyi Szám: '); readln(s);
      Legyen(n,s)
end;

constructor Ember.Kezd;
begin Kerdez
end;

destructor Ember.Kesz;
begin
end;

procedure Utas.Legyen(n, sze: string);
begin Ember.Legyen(n, sze);
      Ulsz:=copy(n,1,2)+sze      {Az útlevélszám a név első két}
end;                             {betűjéből és a személyi számból áll.}

function Utas.Ulszama: string;
begin Ulszama:=Ulsz
end;

procedure Utas.Irdki;
begin writeln('Név: ',Nev);
      writeln('Személyi Szám: ',Szsz);
      writeln('Utlevél szám: ',Ulsz)
end;

end.
```

A módosított modul birtokában a feladat a következő: építsünk fel a dinamikus memóriában egy láncolt listát, melynek elemei két mutatóból állnak. Az egyik egy **Utas** típusú objektumra, a másik a lista esetleges következő elemére hivatkozik. A lista kezdetére egy, a statikus tárban lévő mutató hivatkozzon, melynek kezdeti értéke (ameddig a lista üres) **nil**. A listához két egyszerű művelet, egy új elem hozzácsatolása, valamint az elemek kiírása a képernyőre, is tartozzon. A lista megszüntetésekor nemcsak magát a listaelemet, hanem az általa mutatott **Utas** típusú objektumot is fel kell számolni.

A programban a következő adatszerkezeteket használjuk:

```
. type umut  = ^Utas;    {Utas típusú objektumra hivatkozó mutató}
.      elmut = ^elem              {Lista típusú objektum egy elemére}
.      elem  = record adat: umut;                hivatkozó mutató}
.                     folyt: elmut
. end;
.
.      lista = object eleje: elmut;
.                     constructor init;
.                     destructor done; virtual;
.                     procedure folytat(p1: umut);
.                     procedure kiir;
.              end;
```

A következő lépés az objektumban szereplő módszerek kidolgozása. A kezdőhelyzetet beállító **init** dolga igen egyszerű:

```
. constructor lista.init;
. begin eleje:=nil;
. end;
```

A listát felszámoló destruktorban egyrészt fel kell szabadítani a listán „lógó" **Utas** típusú objektumok helyét, másrészt a lista elemei által elfoglalt memóriát. Az előbbi feladatra a **dispose** eljárás általánosított, kétparaméteres formáját használjuk az **Utas** által az **Ember**-től örökölt **Kesz** destruktorral a második helyen. Esetünkben a destruktor csak a jó OOP stílus kedvéért virtuális.

```
. destructor lista.done;
. var p: elmut;
. begin while eleje<>nil do
.             begin p:=eleje;
.                   dispose(p^.adat,Kesz);
.                   eleje:=p^.folyt;
.                  dispose(p)
.             end
. end;
```

A lista folytatásához egy **umut** típusú paramétert, a listához csatolandó **Utas** típusú objektum aktuális példányára vonatkozó mutatót kell megadni. Az új elemet nem a lista végéhez, hanem elejéhez csatlakoztatjuk, mivel ide könnyű eljutni az **eleje** segítségével, a végét viszont keresni kellene.

```
. procedure lista.folytat(p1: umut);
. var p: elmut;
. begin new(p);
```

```
        p^.adat:=p1;
        p^.folyt:= eleje;
        eleje:=p
end;
```

A lista aktuális tartalmának kiírása egyszerű ciklus. Magát az írást az **Utas** típusú objektumhoz tartozó **irdki** módszer végzi, amelyet kettős mutató alkalmazásával érünk el.

```
procedure lista.kiir;
var p: elmut;
begin p:=eleje; clrscr;
        while p<>nil do
              begin p^.adat^.irdki;
                    p:=p^.folyt
              end
end;
```

Végül a fentieket egy változódeklarációval és egy rövid programtörzzsel kiegészítve eljutunk a program végleges formájához:

```
program oblist;
uses crt, emberek;

type umut = ^utas;
     elmut = ^elem;
     elem = record adat: umut;
                   folyt: elmut
            end;
     lista = object eleje: elmut;
                    constructor init;
                    destructor done; virtual;
                    procedure folytat(p1: umut);
                    procedure kiir;
             end;

var l1: lista;

constructor lista.init;
begin eleje:=nil
end;
```

```
destructor lista.done;
var p: elmut;
begin while eleje<>nil do
            begin p:=eleje;
                  dispose(p^.adat,kesz);
                  eleje:=p^.folyt;
                  dispose(p)
            end

end;

procedure lista.folytat(p1: umut);
var p: elmut;
begin new(p);
      p^.adat:=p1;
      p^.folyt:= eleje;
      eleje:=p
end;

procedure lista.kiir;
var p: elmut;
begin p:=eleje; clrscr;
      while p<>nil do
            begin p^.adat^.irdki;
                  p:=p^.folyt
            end
end;

 {A program törzse:}

begin l1.init;
      l1.folytat(new(umut,kezd));
      while l1.eleje^.adat^.nev<>'*' do
      l1.folytat(new(umut,kezd));
      l1.kiir;
      l1.done
end.
```

A program a **lista** egy példányának, (**l1**), inicializálásával indul és megszüntetésével fejeződik be. Figyeljük meg a lista építésénél a **new** kétparaméteres függvényként való hívását! A tényleges adatbevitelt a módosított **Kezd** konstruktor végzi, a **new** egy **Utas** típusú objektumot hoz létre, és az erre vonatkozó mutatót saját értékeként átadja a **folytat** módszernek. A lista építése úgy fejezhető be, hogy az **Utas** neveként *-ot adunk. A program

lényegét nem érintő szépséghiba, hogy ez a csillag és a hozzá tartozó személyi szám is bekerül a listába. Ezért talán kárpótol a **while** utasításban szereplő szép, „mutatós" szerkezet. Az elkészült lista a bevitelhez képest fordított sorrendben jelenik meg a képernyőn.

4.5. Egerek és objektumok – példaprogram

A feladat legegyszerűbben interaktív adatbeviteli problémaként mutatható be. Folyóiratok olvasószolgálati rovatában gyakran közölnek egy számokból álló táblázatot (ún. információs kártyát), melyen a számok a hirdető cégek különféle árucikkeinek kódjai. A kódolást a hirdetés tartalmazza, az olvasó pedig bekarikázza a számára érdekes számokat, a kártyát beküldi a szerkesztőségbe és ingyen jut színes füzetekhez, udvarias hangú tájékoztató levelekhez, illetve egyéb információhoz.

Vajon hogyan dolgozzák fel a szerkesztőségben az ezrével érkező kártyákat? Az ilyen célú rendszerek egyik részével, az általában többszáz számot tartalmazó kártya adatainak bevitelét megoldó alrendszerrel kívánunk foglalkozni. A feladat meglehetősen speciálisnak tűnik, de ha elég általános megoldást találunk, nincs kizárva, hogy más területeken is használható módszerhez jutunk.

A korszerű oktatási formák terjedésével egyre gyakoribb a tesztrendszerű vizsgáztatás. A tesztlapok minden kérdésre több lehetséges választ tartalmaznak, melyek közül a vizsgázó megjelöli a szerinte helyes megoldást. A tesztlapok gépi kiértékelését végző rendszer is hasonló adatbeviteli igényeket támaszt, mint az előbb említett információs kártyák. Az ilyen adatok közös tulajdonsága, hogy sok, de még egyszerűen **felsorolható** értékből kell egy tetszőlegesen népes csoportot (matematikusok szerint: részhalmazt) kijelölni. Egy adategységen belül minden elemhez egy logikai érték rendelhető (kijelölték-e vagy nem).

Mivel az információs kártya az 1, 2,...n számsorozatot tartalmazza, kézenfekvő lenne egy logikai tömbre gondolni, melynek i-edik eleme **true**, ha i kiválasztott szám, különben pedig **false**.

Ez a megoldás tulajdonképpen elfogadható, de láthatóan nem tökéletes. Egyrészt elegendő lenne csupán a kiválasztott (**true** értékű) elemeket tárolni, a többiről úgyis tudjuk, hogy értékük **false**. Másrészt a Pascal minden **Boolean** értéket egy byte-on tárol, ami esetünkben pazarlás. A gondolatmenet egyenesen a **halmazokhoz** vezet. Ugyanis a

```
. type bitset = set of 0..255;
```

típusú halmaz éppen egy 256 bitből álló vektor, amelyet 32 byte-ba sűrítettek össze. Az i-edik bit értéke 1, ha i a halmazban van, különben 0. Ennél tömörebben az információs kártya tartalma már elméletileg sem tárolható.

Az interaktív adatbevitelt úgy képzeljük el, hogy az összes lehetséges érték megjelenik a képernyőn, a választás pedig **kurzor** segítségével történik. A

kurzort vagy a szokásos **vezérlőbillentyűkkel** vagy – ha elérhető – **egér** segítségével oldjuk meg. A választást célszerű **állapotváltásként** felfogni: a már kiválasztott elem újraválasztása az elemet ki nem választottá teszi.

A programban használt adatszerkezetből természetesen egy objektumot készítünk a következő mezőkkel és módszerekkel:

```
type setobject = object adatok:bitset;      {256 elemű halmaz}
                        kurrens:integer;    {az aktuális elem}
                        szin:byte;     {tintaszín+16*papírszín}
                        constructor init(szinkod: byte);
                        procedure torol;
                        procedure kiir;
                        procedure szerkeszt;
                 end;
```

A kitöltött kártyát reprezentáló objektum az **adatok** mezőn kívül a **kurrens**, a szerkesztés során az aktuális elem (amelyikre a kurzor mutat) értékét tartalmazza, a **szin** pedig a kiírásnál a papír és a tinta színét határozza meg. Az egyszerűség kedvéért a két színt „16*papírszín+tintaszín" formában kell 1 byte-ként megadni, ahogy az a **TextAttr** előre definiált változóban majd tárolódik.

Az objektum konstruktora egyszerűen törli a halmazt, **kurrens** értékét 0-ra, **szin**-ét pedig a kapott paraméter értékére állítja.

```
constructor setobject.init(szinkod: byte);

 {A halmaz kezdeti tartalmának beállítása}

begin adatok:=[ ];
      kurrens:=0;
      szin:=szinkod;
end;
```

A halmaz törlése olyan egyszerű, hogy külön említeni sem kell, jelenlegi formájában a törlés után az üres halmazt ki is írja.

A halmaz kiírása a képernyő első 16 sorában, elemenként 5 pozíció elfoglalásával megy végbe (háromjegyű számok, jobbról-balról szóközzel határolva). A számok 1-től 256-ig a **szin** mező előírása szerint jelennek meg, a halmazhoz tartozó elemeket invertáljuk. A számokat egy kétszeres ciklus generálja, az inverz írást pedig egy egyszerű segédeljárás (**inverz**) kapcsolja be és ki a tinta- és háttérszín felcserélésével **TextAttr**-ben. A színt minden kiírás elején újraállítjuk, ezért ez két kiírás között megváltoztatható. Magát az írást egy külön eljárás végzi, amely a határoló szóközökre, valamint az esetleges vezető nullákra összpontosít:

```
procedure ir(n: integer);

begin egerki; write(' ');
      if n<10
         then write('00')
         else if n<100
              then write('0');
         write(n,' ');
         egerbe
end;
```

Az eljárásban két egérkezelő hívás is szerepel. Ezek az egérkurzort kiteszik, illetve elveszik a képernyőről. Azért van rájuk szükség, mert az egérkurzor helyére a **write** eljárás nem ír. Az egérkezelő eljárásokkal később, a **szerkeszt** módszerrel kapcsolatban foglalkozunk. Maga a **kiir** módszer a következő:

```
procedure setobject.kiir;

var i, j: integer;

begin clrscr; textattr:=szin;
      for i:=0 to 15 do
          for j:=0 to 15 do
              begin gotoxy(5*j+1,i+1);
                    if 16*i+j in adatok
                       then begin inverz;
                                  ir(16*i+j);
                                  inverz
                            end
                       else ir(16*i+j)
              end
end;
```

A képernyőn megjelenő halmaz szerkesztése az objektum legbonyolultabb módszere. Több segédeljárást használ, melyek közül az egérkezeléssel foglalkozó eljáráscsokor a legfontosabb.

A MicroSoft és a hozzá hasonló egerek vezérlése az 51-es ($33) megszakítás segítségével történik. A programban szereplő egérkezelő eljárások a következők:

Az egér jelenlétének vizsgálata

Egy **Boolean** típusú függvény, amely azt használja ki, hogy egér hiányában a megfelelő megszakítási vektor első byte-jára egy IRET utasítás ($CF) kerül.

```
function vaneger: Boolean;

type bytep = ^byte;
var v: bytep;
begin getintvec(51,pointer(v));
      if (v=NIL) or (v^=$CF)
      then vaneger:=false
      else vaneger:=true
end;
```

Általános egérkezelés

Az eljárás első paramétere (m1) határozza meg a kívánt egérfunkciót. Az eljárás a processzor regisztereit és az 51-es szoftvermegszakítást használja:

```
procedure eger( var m1, m2, m3, m4: word);
var r: registers;
begin with r do
            begin ax:=m1; bx:=m2;
                  cx:=m3; dx:=m4
            end;

      intr(51,r);
      with r do
            begin m1:=ax; m2:=bx;
                  m3:=cx; m4:=dx
            end
end;
```

m1 különböző értékeinek jelentése:

0: Az egér alaphelyzetbe kerül (reset), a többi paraméter értéke közömbös.
1: Az egérkurzor megjelenik, a többi paraméter értéke közömbös.
2: Az egérkurzor eltűnik, a többi paraméter értéke közömbös.
3: Az egér bal oldali gombjának figyelése. Ha híváskor le van nyomva, **m2** értéke páratlan lesz, különben páros. Lenyomott gomb esetén **m3** az egérkurzor vízszintes (X irányú), **m4** a függőleges (Y irányú) koordinátáját tartalmazza.
8: Az egér mozgásának korlátozása. A vízszintes és függőleges irányú koordináta-határ hívás előtt **m3**-ban, ill. **m4**-ben adható meg. A 0 érték korlátozást nem jelent. Az egér-koordináták intervallumai: $0 \leq X \leq 639$ és $0 \leq Y \leq 199$.

Az egérkezelő eljárásokon alapszik néhány egyszerű és összetett egértevékenység, amelyek elsősorban csak nevet adnak az általános egérkezelő különböző hívásainak.

A szerkesztő módszer további segédeljárásaihoz tartozik a **bitváltás**, amely egy számot betesz a halmazba, ha nem volt benne, és kivesz belőle, ha benne volt; valamint a **kurzor**, amely a paraméterként adott két karaktert kiírja a képernyőre az aktuális elem két oldalára.

Maga a szerkesztő eljárás pedig a következő:

```
procedure setobject.szerkeszt;

var i: integer;
    m1, m2, m3, m4, mx, my: word;
    ki: Boolean;
    ch: char;

function holeger(mx, my: integer): integer;

 {Az egérkoordináták transzformálása halmazelemmé (0..255)}

var sx, sy: word;

begin sx:=(mx div 8); sy:=(my div 8);
      holeger:=16*sy+(sx div 5)
end;

procedure bitvaltas(n: integer);

 { n törlődik a halmazból, ha benne volt, és belekerül,
 ha nem volt benne}

begin gotoxy(5*(n mod 16)+1,(n div 16)+1);
      if n in adatok
      then begin adatok:=adatok-[n];
                 ir(n)
           end
      else begin adatok:=adatok+[n];
                 inverz;
                 ir(n);
                 inverz
           end
end;

procedure kurzor(c1, c2: char);

 {A c1, c2 karakterekből álló kurzort kiteszi
 az aktuális elem két oldalára}
```

```
begin egerki;
      gotoxy(5*(kurrens mod 16)+1,(kurrens div 16)+1);
      write(c1); gotoxy(wherex+3,wherey); write(c2);
      egerbe
end;

 {szerkeszt törzse:}

begin clrscr; kiir;
      kurzor(bal szel, jobb szel);
      if egervan
      then begin m1:=0; eger(m1, m2, m3, m4);  reset egér
                 m1:=8; m3:=0; m4:=120;
                 eger(m1, m2, m3, m4);        {Az egérmozgás}
                                                {korlátozása}
                 egerbe;                {Az egérkurzor megjelenik}
           end;
      repeat ki:=false;
             if egervan
             then if lenyomva(mx, my)
                  then begin ereszdel; i:=holeger(mx,my);
                             kurzor(' ',' ');
                             kurrens:=i;          {Az egeres}
                             bitvaltas(i);         {vezérlés}
                             kurzor(balszel, jobbszel);
                       end;
             if keypressed   {A párhuzamos billentyűs vezérlés}
             then begin ch:=readkey; i:=kurrens;
                        if ch = #0    {Ha kurzormozgató bil-}
                        then begin ch:=readkey;     {lentyű}
                                   case ord(ch) of
                             {fel}          72: if kurrens>15
                                                then i:=kurrens-16
                                                else aha;
                             {balra}        75: if kurrens>0
                                                then i:=kurrens-1
                                                else aha;
                             {jobbra}       77: if kurrens<255
                                                then i:=kurrens+1
                                                else aha;
                             {le}           80: if kurrens<240
                                                then i:=kurrens+16
                                                else aha;
                           {Home}          71: i:=0;
```

```
                    {PgUp}          73: i:=15;
                    {PgDn}          81: i:=255;
                    {End}           79: i:=240;
                                  else aha;
                                  end;
                                  kurzor(' ',' ');
                                  kurrens:=i;
                                  kurzor(balszel, jobbszel);
                            end
                        else case ord(ch) of
          {Enter}                 13: begin bitvaltas(kurrens);
                                            kurzor(balszel,
                                                   jobbszel)
                                      end;
                                      27: ki:=true;        {Esc-re}
                                  else aha                 {kilépés}
                                  end
                    end
          until ki;
          if egervan
          then egerki;
          kurzor(' ',' ');
end;
```

Az eljárás az aktuális halmaz kiírásával és az esetleges egér aktivizálásával indul. A kurzort az egérrel vagy a szokásos kurzorvezérlő billentyűkkel (a Home, PgUp, PgDn és az End is szerepet kapott) pozicionálhatjuk. Egy elem felvétele vagy törlése az egér bal gombjával vagy az ENTER leütésével történhet. A szerkesztésből az ESC billentyűvel lehet kilépni, ekkor az egér és az aktuális elemet jelző kurzor (erre a célra a #16, ill. a #17 karaktereket használtuk) eltűnik.

A teljes modul, néhány apró kiegészítéssel a következő:

```
unit setob;
interface
uses dos, crt;

type bitset = set of 0..255;
     setobject = object adatok:bitset;     {256 elemű halmaz}
                        kurrens:integer;    {az aktuális elem}
                        szin:byte;     {tintaszín+16*papírszín}
                        constructor init(szinkod: byte);
                        procedure torol;
                        procedure kiir;
                        procedure szerkeszt;
```

```
                end;

implementation

const balszel = #16;          {Az aktulis elem két szélén megjelenő
      jobbszel = #17;                                    kurzorok}

var egervan: Boolean;

procedure eger(var m1, m2, m3, m4: word);

 {Egérkezelő eljárás, amely m1 értékétől függően
 különböző funkciókat lát el}

var r: registers;
begin with r do
           begin ax:=m1; bx:=m2;
                 cx:=m3; dx:=m4
           end;
      intr(51,r);
      with r do
           begin m1:=ax; m2:=bx;
                 m3:=cx; m4:=dx
           end
end;

procedure egerbe;                    {Az egérkurzor megjelenik}
var m1, m2, m3, m4: word;
begin m1:=1;
eger(m1, m2, m3, m4)
end;

procedure egerki;                    {Az egérkurzor eltűnik}
var m1, m2, m3, m4: word;
begin m1:=2;
      eger(m1, m2, m3, m4)
end;

procedure ereszdel;

 {Ha híváskor az egér bal gombja le van nyomva,
 megvárja amíg elengedjük}

var m1, m2, m3, m4: word;
```

```
begin m1:=3;
      repeat eger(m1, m2, m3, m4)
      until not odd(m2)
end;

function vaneger: Boolean;

 {Az egér jelenlétének vizsgálata. Ha nincs egér,
 a hozzá tartozó 51-es ($33) megszakítási vektor
 egy IRET ($CF) utasításra mutat.
 A NIL-t csak a biztonság kedvéért vizsgáljuk.}

type bytep = ^byte;
var v: bytep;
begin getintvec(51,pointer(v));
      if (v=NIL) or (v^=$CF)
      then vaneger:=false
      else vaneger:= true
end;

function lenyomva(var mx, my: word): Boolean;

 {Ha az egér bal gombja le van nyomva, az értéke true
 és ekkor mx, my az egér aktuális koordinátáit tartalmazza.}

var m1, m2: word;
begin m1:=3;
      eger(m1, m2, mx, my);
      if odd(m2)
      then lenyomva:=true
      else begin lenyomva:=false;
                 mx:=0; my:=0
           end
end;

procedure aha;          {Hangjelzés, ha rossz billentyűt nyomunk le}
begin sound (50); delay(100); nosound; delay(50);
      sound (50); delay(100); nosound
end;

procedure inverz;                  {Kicseréli a tinta- és papírszínt}
begin textattr:=(textattr  div 16)+16*(textattr  mod 16)
end;
```

```
procedure ir(n: integer);

 {Egy halmazelem kiírása 3 jegyre, egy-egy szóközzel határolva}

begin egerki; write(' ');
      if n<10
         then write('00')
         else if n<100
              then write('0');
      write(n,' ');
      egerbe
end;

constructor setobject.init(szinkod: byte);

 {A halmaz kezdeti tartalmának beállítása}

begin adatok:=[ ];
      kurrens:=0;
      szin:=szinkod;
end;

procedure setobject.kiir;

 {A halmaz összes lehetséges elemének kiírása a képernyő
  első 16 sorába, soronként 16 elemmel. A ténylegesen
  a halmazhoz tartozó elemek inverz formában jelennek meg.}

var i, j: integer;

begin clrscr; textattr:=szin;
     for i:=0 to 15 do
         for j:=0 to 15 do
             begin gotoxy(5*j+1,i+1);
                   if 16*i+j in adatok
                      then begin inverz;
                                 ir(16*i+j);
                                 inverz
                           end
                      else ir(16*i+j)
             end
end;

procedure setobject.torol;                    {A halmaz törlése,}
begin adatok:=[ ]; kurrens:=0;          {az üres halmaz kiírása.}
      kiir
end;
```

```
procedure setobject.szerkeszt;

 {A halmaz szerkesztése, elemek felvétele, illetve törlése a
 halmazból. Az aktuális elemre a kurzormozgató billentyűkkel,
 illetve az egérrel lehet ráállni, a felvétel vagy törlés az
 ENTER-rel vagy az egér bal oldali gombjával történik. A halmazba
 felvett elem inverz formára íródik át. A szerkesztésből kilépni
 az Esc billenyűvel lehet.}

var i: p integer;
    m1, m2, m3, m4, mx, my: word;
    ki: Boolean;
    ch: char;

function holeger(mx, my: integer): integer;

 {Az egérkoordináták transzformálása halmazelemmé (0..255)}

var sx, sy: word;

begin sx:=(mx div 8); sy:=(my div 8);
      holeger:=16*sy+(sx div 5)
end;

procedure bitvaltas(n: integer);

 { n törlődik a halmazból, ha benne volt, és belekerül,
 ha nem volt benne}

begin gotoxy(5*(n mod 16)+1,(n div 16)+1);
      if n in adatok
      then begin adatok:=adatok-[n];
                 ir(n)
           end
      else begin adatok:=adatok+[n];
                 inverz;
                 ir(n);
                 inverz
           end
end;
```

```
procedure kurzor(c1, c2: char);

 {A c1, c2 karakterekből álló kurzort kiteszi
  az aktuális elem két oldalára}

begin egerki;
      gotoxy(5*(kurrens mod 16)+1,(kurrens div 16)+1);
      write(c1); gotoxy(wherex+3,wherey); write(c2);
      egerbe
end;

 {szerkeszt törzse:}

begin clrscr; kiir;
      kurzor(balszel, jobbszel);
      if egervan
      then begin m1:=0; eger(m1, m2, m3, m4);     {reset egér}
                 m1:=8; m3:=0; m4:=120;
                 eger(m1, m2, m3, m4);        {Az egérmozgás}
                 egerbe;                              {korlátozása}
           end;                          {Az egérkurzor megjelenik}
      repeat ki:=false;
             if egervan
             then if lenyomva(mx, my)
                  then begin ereszdel; i:=holeger(mx,my);
                             kurzor(' ',' ');
                             kurrens:=i;           {Az egeres}
                             bitvaltas(i);          {vezérlés}
                             kurzor(balszel, jobbszel);
                       end;
             if keypressed   {A párhuzamos billentyűs vezérlés}
             then begin ch:=readkey; i:=kurrens;
                        if ch = #0        {Ha kurzormozgató}
                        then begin ch:=readkey;   {billentyű}
                                   case ord(ch) of
                  {fel}              72: if kurrens>15
                                         then i:=kurrens-16
                                         else aha;
                  {balra}            75: if kurrens>0
                                         then i:=kurrens-1
                                         else aha;
                  {jobbra}           77: if kurrens<255
                                         then i:=kurrens+1
                                         else aha;
```

```
                    {le}              80: if kurrens<240
                                          then i:=kurrens+16
                                          else aha;
                    {Home}            71: i:=0;
                    {PgUp}            73: i:=15;
                    {PgDn}            81: i:=255;
                    {End}             79: i:=240;
                                    else aha;
                                    end;
                                    kurzor(' ',' ');
                                    kurrens:=i;
                                    kurzor(balszel,jobbszel);
                              end
                         else case ord(ch) of
          {Enter}                 13: begin bitvaltas(kurrens);
                                          kurzor(balszel,
                                                 jobbszel)
                                    end;
                                  27: ki:=true;     {Esc-re kilépés}
                              else aha
                              end
                 end
      until ki;
      if egervan
      then egerki;
      kurzor(' ',' ');
end;       {szerkeszt}
begin egervan:=vaneger;                {Inicializáláskor eldöntjük,}
end.                                   {van-e egerünk}
```

Végül a **setob** modult felhasználó programra adunk példát:

```
program halmaz;
uses crt, setob;
var matrix: setobject;
begin matrix.init(yellow+16*blue);
      matrix.adatok:=[0,128,129,255];
      matrix.szerkeszt;
      readln;
      matrix.kiir;
      readln;
      matrix.torol;
      readln
end.
```

Az egyszerű program létrehozza, és néhány kezdőelemmel feltölti a halmazt, majd behívja a szerkesztőt. Szerkesztés után kiírja az eredményhalmazt, majd törli az egészet.

Irodalmi ajánlások

Az OOP-vel kibővített Turbo Pascal bibliája természetesen a Borland International által kiadott kézikönyv. A **Turbo Pascal Reference Guide** az 5.0-ás változattal együtt jelent meg 1987–88-ban. Az 5.5-ös (OOP) változat kiadásakor ezt egy **Object Oriented Programming Guide** c. könyvecskével egészítették ki 1988 végén, amely a nyelvben az OOP-vel kapcsolatos módosításokat írja le.

A hazai kiadású Pascal irodalom is örvendetesen gyarapszik, részletekbe menő, OOP Pascal könyv még sajnos nem jelent meg. A magyarul ajánlható, jelenlegi legátfogóbb és legfrissebb művek: Kris Jamsa – Steven Nameroff „Turbo Pascal programozói könyvtár" (Novotrade Rt., 1989.); Angster E. – Béres E. „Turbo Pascal" (SZÁMALK, 1990.).

A legfrissebb külföldi, hozzánk valószínűleg még el nem jutott művek közül – folyóiratbeli ajánlás alapján – Tom Swan: **Mastering Turbo Pascal 5.5** (H.W. Sams & Co., 1989. ISBN 0-672-48450-1) c. művét lehet ajánlani.

A gyakorlati alkalmazók számára nagy segítséget nyújthat az objektumdefiníciók sokaságát (kb. 130 objektumot) tartalmazó **Object Professional 1.0** (OPRO), a Turbo Power Software cég által kidolgozott és forgalmazott könyvtár, amelyhez mintegy 1600 oldal dokumentáció tartozik.

A folyóiratokat tekintve, legmelegebben a **Dr. Dobb's Journal** ajánlható, amely Structured Programming rovatában 1989 közepe óta rendszeresen foglalkozik az OOP-vel és ezen belül az objektum orientált Pascallal is.

A könyv aktualitását a Turbo Pascal objektum orientált kiterjesztése adja. Ezzel a bővítéssel a közel két évtizedes múltú nyelv megerősítette pozícióját a korszerű programozási eszközök között.

A részletesebb elemzés során kiderült, hogy az objektumok bevezetése tulajdonképpen szükségszerű minden olyan nyelvben, ahol az eljárás (vagy függvény) típus általános formában használható. Az objektum fogalmának lényege az a felismerés, hogy a környező világ dolgai jobban modellezhetők számítástechnikai eszközökkel, ha jellemző adataikat és működési módszereiket nem egymástól elválasztva, hanem egységes egészként kezeljük.

Kis túlzással azt mondhatjuk, hogy az objektumokban az adatszerkezetek életre kelnek, és mint a színészek egy darabban, különböző szerepek játszására válnak alkalmassá.

A könyv a Pascal alapjait ismertnek feltételezve, az eljárásoktól indul és a moduláris programozáson keresztül vezeti el az olvasót az Objektum Orientált Programozás világába.

338,– Ft