

450546

10414134



Szerzők: 1. fejezet: Sági Gábor
2-11 fejezet: Molnár Csaba

Szerkesztő: Bártfai Barnabás

ISSN 1418-8791
ISBN 963 03 7152 9

Kiadja a BBS-E Betéti Társaság
1630 Budapest, Pf. 21.
Felelős kiadó: a BBS-E Betéti Társaság ügyvezetője

Minden jog fenntartva! A könyv vagy annak oldalainak másolása,
sokszorosítása csak a kiadó írásbeli hozzájárulásával történhet.

Nyomdai munkák: Bíró Family Kft.
Felelős vezető: Bíró Endre

Tartalomjegyzék

Előszó	5
1. A programozás elmélete.....	11
1.1. Bevezetés	11
1.2. A program készítésének menete	12
1.2.1. A program elkészítésének lépései	12
1.2.2. Az algoritmus és szerkezete.....	12
1.2.2.1. Szekvencia	13
1.2.2.2. Iteráció.....	13
1.2.2.3. Szelekció.....	14
1.2.3. Az algoritmussal szemben támasztott követelmények	14
1.2.4. Folyamatábrák.....	15
1.2.5. A strukturált problémamegoldás	16
1.2.6. A program készítésének folyamata	17
1.3. Programozási alapfogalmak.....	18
1.3.1. A programírás konvenciói	19
1.3.2. Az azonosítók.....	19
1.3.3. A konstans.....	19
1.3.4. A változók	19
1.3.5. Kifejezések.....	20
1.3.6. A műveletek végrehajtásának sorrendje (prioritása)	20
1.3.7. Boole-féle algebra	21
1.3.8. Az értékadó utasítás	22
1.3.9. Input és output	23
1.3.9.1. Az input	23
1.3.9.2. Az output	23
1.3.10. A soros programozás	23
1.3.11. A programok szerkezete	24
1.4. Alternatívák és ciklusok kezelése a programban.....	26
1.4.1. Egyágú döntések.....	26
1.4.2. Kétágú döntés.....	28
1.4.3. Több lehetséges érték közötti választás	29
1.4.4. Feltétel nélküli vezérlés átadása	30
1.4.5. A ciklikus tevékenység (iteráció) kezelése a programban.....	31
1.4.5.1. 4.5.1 Előletesztelő ciklus.....	31
1.4.5.2. Hátulatesztelő ciklus.....	33
1.4.5.3. Növekményes ciklus	35
1.5. Tömbök.....	36
1.5.1. A tömbök használata.....	36
1.5.2. Tömb elemeire való hivatkozás.....	37

1.5.3. Tömb feltöltése.....	37
1.5.4. Tömb elemeinek kiírása.....	37
1.6. Alprogramok.....	37
1.6.1. Függvények.....	38
1.6.2. Eljárás.....	38
1.6.3. A főprogram és az alprogram kapcsolata.....	39
1.6.4. Az azonosítók hatásköre.....	39
2. A programozás alapjai Turbo Pascalban.....	40
2.1. Az első program, avagy ismerkedés a nyelvvel.....	40
2.1.1. A Turbo Pascal programozási nyelvről.....	40
2.1.1.1. Program és nyelv.....	40
2.1.1.2. A Turbo Pascal programok felépítése.....	41
2.1.1.3. A nyelv elemei.....	41
2.1.1.4. Programkönyvtárak (unitok) szerepe.....	42
2.1.1.5. Azonosítók.....	43
2.1.1.6. Forrás és lefordított programok.....	43
2.1.1.7. Fordító direktívák.....	44
2.1.2. Az első program elkészítése.....	44
2.1.2.1. A Turbo Pascal indítása.....	44
2.1.2.2. A program begépelése.....	44
2.1.2.3. Megjegyzések elhelyezése a programban.....	45
2.1.2.4. A program elmentése.....	46
2.1.2.5. A program fordítása és futtatása.....	47
2.1.2.6. Kilépés a Turbo Pascaltól.....	47
2.1.2.7. Program megnyitása lemeztől.....	47
2.1.2.8. Egy kis módosítás.....	48
2.1.2.9. Mentés másként.....	48
2.1.2.10. Fordítási hibák kezelése.....	48
2.1.2.11. Új program készítése.....	48
2.2. A képernyő kezelése: kiírások.....	49
2.2.1. A képernyő felépítése.....	49
2.2.2. A CRT unit használata.....	50
2.2.3. A képernyő letörlése.....	50
2.2.4. A kurzor helyének megadása.....	50
2.2.5. Több szöveg kiírása.....	52
2.2.6. Számok kiírása, egyszerű számolások.....	54
2.2.7. Színek használata.....	56
2.2.7.1. Színes szövegek.....	56
2.2.7.2. A teljes képernyő színének megváltoztatása.....	58
2.3. Változók.....	59
2.3.1. Adatok, adattípusok.....	59
2.3.1.1. Egész típusok.....	59
2.3.1.2. Valós típusok.....	60
2.3.1.3. Szöveges típusok.....	62
2.3.1.4. Logikai típus.....	62
2.3.1.5. Sorszámozott típusok.....	62
2.3.1.6. Típusok kompatibilitása.....	63

2.3.2. Műveletek, relációk.....	63
2.3.2.1. Numerikus típusokon értelmezett műveletek.....	63
2.3.2.2. Szöveges típusokon értelmezett műveletek.....	63
2.3.2.3. Logikai műveletek.....	64
2.3.2.4. Relációk.....	64
2.3.2.5. Függvények.....	65
2.3.2.6. Kifejezések.....	65
2.3.2.7. Precedencia-szabály.....	66
2.3.3. Változók deklarálása.....	66
2.3.4. Értékadás.....	66
2.3.5. A változók egyszerűbb alkalmazásai.....	67
2.3.5.1. Értékadás és a változó értékének kiírása.....	67
2.3.5.2. A változók értékének megváltoztatása.....	68
2.3.5.3. Kifejezések kiszámítása.....	69
2.3.5.4. Feladatok változókra.....	69
2.3.6. Értékadás a program futása közben (adatbekérés).....	71
2.3.6.1. Az adatbekérés egy speciális alkalmazása: várakozás.....	75
2.3.7. Állandók (konstansok) használata.....	76
2.3.7.1. Beépített konstansok.....	78
2.4. Számlálós ciklusok.....	78
2.4.1. A ciklus.....	78
2.4.2. A számlálós ciklus.....	78
2.4.3. Egyszerű ciklusok.....	80
2.4.4. A ciklusváltozó felhasználása a ciklusmagban.....	81
2.4.5. Lépésköz kezelése.....	87
2.4.6. Egy másik ágyazott ciklusok.....	89
2.5. Elágazások.....	93
2.5.1. Elágazás kétfelé.....	94
2.5.2. Elágazás többfelé.....	97
2.5.3. Elágazás többfelé esetszétválasztással.....	98
2.6. Feltételes ugrások.....	100
2.7. Tesztelős ciklusok.....	104
2.7.1. Előtesztelős ciklusok.....	105
2.7.2. Hátteltesztelős ciklusok.....	105
2.7.3. Példa a tesztelős ciklusokra.....	105
2.7.4. Feladatok tesztelős ciklusokra.....	107
2.7.5. Végtelen ciklusok.....	112
2.8. Alprogramok.....	113
2.8.1. Eljárások.....	114
2.8.2. Paraméterátadás.....	115
2.8.3. Függvények.....	118
2.8.4. Lokális és globális változók.....	119
2.9. Összetett adattípusok.....	121
2.9.1. Vektorok.....	121
2.9.2. Tömbök.....	123
2.9.2.1. Kétdimenziós tömbök.....	123
2.9.2.2. Három- és többdimenziós tömbök.....	124
2.9.3. Saját típusok létrehozása.....	124

2.9.3.1. Típusdeklaráció.....	124
2.9.3.2. Tömbök típusmegadása.....	125
2.9.4. Rekordok.....	125
2.9.4.1. A rekordkezelés megkönnyítése: a With utasítás.....	128
2.9.5. Fájlok.....	129
2.9.6. Objektumok.....	130
2.9.7. Halmazok.....	130
2.9.8. Egyéb sorszámozott típusok.....	132
2.9.8.1. Résztartomány típus.....	132
2.9.8.2. Felsorolás típus.....	132
2.9.9. Tipizált konstansok.....	133
2.10. Összefoglalás.....	134
3. Algoritmusok készítése.....	136
3.1. A programkészítés folyamata.....	136
3.2. A program tervezése.....	137
3.3. Algoritmus-leíró eszközök.....	137
3.3.1. Folyamatábra.....	137
3.3.2. Mondatszerű leírás.....	140
3.3.3. Struktogram.....	142
4. Függvények.....	144
4.1. Matematikai függvények.....	144
4.1.1. Alapvető matematikai függvények.....	144
4.1.2. Kerekítés.....	146
4.1.3. Véletlenszám generálása.....	147
4.1.4. Trigonometriai függvények.....	149
4.1.5. Hatvány, logaritmus.....	150
4.2. Szövegek kezelése.....	151
4.2.1. A szöveg hossza.....	151
4.2.2. Szöveg kezelése karakterenként.....	152
4.2.3. Keresés a szövegben.....	154
4.2.4. A szöveg alakítása.....	154
4.3. Sorszámozott típusok függvényei és eljárásai.....	157
4.4. Típuskonverziós függvények.....	158
5. Fájlok kezelése.....	162
5.1. Deklaráció és hozzárendelés.....	163
5.2. Fájl megnyitása.....	163
5.3. Fájl bezárása.....	164
5.4. A rekordmutató pozicionálása.....	164
5.5. Adat beolvasása fájlból.....	165
5.6. Adat írása fájlba.....	165
5.7. A fájl törlése.....	166
5.8. Nem tipizált fájlok.....	166
5.9. Szövegfájlok kezelése.....	167
5.9.1. Hozzárendelés, megnyitás és bezárás.....	167
5.9.2. Beolvasás és írás.....	167
5.10. Fájl átnevezése.....	168
5.11. Könyvtárak kezelése.....	169
5.12. A fájlkezelés hibáinak elhárítása.....	169

6. Grafika a Turbo Pascal-ban.....	171
6.1. A grafikus képernyő felépítése.....	171
6.2. Inicializálás.....	172
6.3. Színek használata.....	174
6.4. A grafikus képernyő törlése.....	174
6.5. CP – a grafikus kurzor.....	175
6.6. Pontok.....	175
6.7. Vonalak.....	175
6.8. Alakzatok.....	177
6.9. Kitöltött alakzatok.....	179
6.10. Szövegek.....	181
7. Saját unitok készítése.....	184
8. Dinamikus adatszerkezetek.....	187
8.1. Dinamikus adatok deklarálása.....	187
8.2. Értékadás = memóriaterület lefoglalása.....	188
8.3. A lefoglalt terület felszabadítása.....	188
8.4. Egy egyszerű példa.....	189
9. Alkalmazások.....	189
9.1. Gyakori feladatok vektorokra.....	190
9.1.1. Összegzés.....	190
9.1.2. Keresés.....	191
9.1.3. Maximum-kiválasztás.....	193
9.1.4. Rendezés.....	193
9.1.4.1. Rendezés minimum-kiválasztással.....	193
9.1.4.2. Buborékos rendezés.....	195
9.1.4.3. Beillesztéses rendezés.....	196
9.1.5. Logaritmikus keresés.....	197
9.2. Menük készítése.....	198
9.2.1. Egyszerű menü.....	198
9.2.2. Speciális billentyűk kezelése.....	202
9.3. Rekurzio.....	205
10. Egy komolyabb program elkészítése.....	206
10.1. A feladat.....	207
10.2. A programterv.....	208
10.3. Kódolás.....	209
10.4. Hibajavítás, tesztelés.....	213
10.5. Hatékonyságvizsgálat.....	214
10.6. Dokumentáció.....	215
10.7. A teljes programlista.....	217
10.8. Feladatok.....	219
11. Mellékletek.....	219
11.1. Az ASCII kód tábla.....	220
11.2. A Turbo Pascal védett szavai.....	221
11.3. A Turbo Pascal kulcsszavai és utasításai.....	222
11.4. A Turbo Pascal nyelv eljárásai és függvényei.....	223
11.5. Fordítási hibák jegyzéke.....	228
11.6. Futási hibák jegyzéke.....	232



Előszó

A könyv két jól elkülöníthető részből áll. Az első fejezet a programozás elméletével foglalkozik, így elsősorban azoknak szól, akik még nem próbálkoztak programírással. Könyvünk második, gyakorlati részében a Turbo Pascal programozási nyelvet ismerheti meg a kedves olvasó.

Ha még nem programozott, de szeretne, és nincs lehetősége tanórán vagy tanfolyamokon elsajátítani egy programozási nyelvet, akkor ez a könyv e célra kiválóan alkalmas. Ez a kezdők könyve. A szerző igyekezett úgy felépíteni, hogy bárki, aki kedvet érez magában programok írására, e könyvecske segítségével megtanulhassa az alapokat, és talán még azon túl is eljuthasson egy kicsit. Szeretnénk, ha mindenki megszeretné a programozást, hiszen az egy nagyon jó és rendkívül kreatív játék! De ha már belekóstolt a programozásba Turbo Pascal nyelven, de elege van az olyan könyvekből, amelyek inkább egy szó-tárra hasonlítanak, vagyis ABC rendben felsorolják a létező összes utasítást és eljárást, akkor erre a könyvre van szüksége. (Bár természetesen az ilyen típusú könyvekre is szükség van!) Ha már programozott, de nagyrészt a saját erődből jöttél rá a dolgokra, ezzel a könyvvel rendszerezheti a tudását.

Ha már ismeri a Turbo Pascal nyelvet, és tudása elmélyítésére vágyik, arra, hogy új dolgokat tanulj meg a nyelvről, akkor valószínűleg ez nem az Ön által keresett könyv. Persze sohasem lehet tudni...

De ha olyan könyvet keres, amelyet oktató és diák egyaránt használna forgathat, akkor ez a könyv az.

Összefoglalva: ez a könyv a kezdetektől indulva vezet be olvasóját a programozásba, illetve a Turbo Pascal programozási nyelvbe. Minden új ismeretet példákon keresztül magyaráz, és sok feladatot tartalmaz, amelyekkel gyakorolhatók ezek az újdonságok. Egyaránt alkalmas lehet oktatói segédletnek, tankönyvnek, illetve az önálló tanulást is támogatja.

Sokszor feltették nekem azt a kérdést, hogy manapság, amikor szinte minden problémára létezik szoftver, kell-e programozási ismeret azok-

nak, akik nem profi programozók. A válaszom az, hogy nem kell azoknak, akik a számítógépet csak munkaeszköznek tekintik, de akit már egy kicsit is érdekel a masinája, annak melegen ajánlható. A számítógépes ismeretekhez, a számítógép kultúrájához mindenképpen hozzátartozik a programozás, és segít abban, hogy egy kicsit megismerhessük és megérthessük gépünk „lelkivilágát”. Azonfelül a programok írása megtanít bennünket sok olyasmire – logikus gondolkodásra, pontos tervezésre, a kis részletek el nem hanyagolására, stb. –, amit az élet egyéb területein is alkalmazhatunk. Főleg gyerekeknek nagyon hasznosak ezek a képességek; nem is kell magyarázni, hogy mekkora hasznát vehetik ezeknek az iskolában, például a matematika (vagy bármely más) órán. Jobb is, ha a figyelmüket ezzel kötjük le: ha már „számítógépezik” egész nap, mindenképpen jobb, ha programokat ír, mintha harci játékokkal játszana. A programozás van olyan hasznos és kreatív, mint bármelyik építőjáték (egy kicsit az is: építőköveink az egyes utasítások, ezekből kell valami olyat alkotni, ami „megáll a saját lábán”). És végezetül még egy érv: a legkedveltebb szoftverek, a szövegszerkesztők és táblázatkezelők világába is belopóztak mára a programozás alapfogalmai. A szövegszerkesztők körlevél szolgáltatása, a Word-ben a mezők kezelése, az Excelben néhány függvény (és persze még sokáig sorolhatnám) megtanulása sokkal egyszerűbbé válhat a programozás alapfogalmainak ismeretével. Adatbázis-kezelő programok használatánál pedig szinte nélkülözhetetlenek a programozási ismeretek (sokszor nagyon is magas szinten!). Mondhatnánk erre azt persze, hogy akkor halljuk azokat az alapfogalmakat, aztán fejezzük is be! Lehetne így is. Azonban sokkal jobban megtanulható mindez, ha a gyakorlatban is kipróbáljuk egy programozási nyelv segítségével.

A könyv használatáról annyit, hogy ha önállóan halad vele, akkor szigorúan sorban kell haladni, semmit sem kihagyva. Az egyes fejezetekben példákon keresztül igyekszünk bemutatni az új ismereteket. A példaprogramokat célszerű begépelni, és a javasolt módokon kipróbálni. Minden fejezet végén feladatok találhatók, amelyekkel gyakorolható az adott témakör. Minden feladatnak több megoldása lehetséges, bár ezek eltérhetnek egymástól hosszban, hatékonyságban, bonyolultságban ...stb.

Az első programozásméleti fejezet után megismerkedhet a programozás, és ezzel együtt a Turbo Pascal alapjaival is, vagyis bemutatjuk azokat az alapfogalmakat és „építőköveket” amelyekből egy program összerakható. Ezen ismeretek birtokában már komoly programok is

írhatók, de a programkészítés csak „öszönösen” megy. A továbblépés a strukturált programozás lehet, ennek alapelemeivel, a programkészítés lépéseivel, illetve az algoritmusok készítésével foglalkozik a következő fejezet.

A további két nagyobb témakörben az alapismereteken túlmenő, de hasznos nyelvi elemeket igyekszünk bemutatni, amellyel a tudásunk biztosabbá és gazdagabbá válhat. Ezek részben a Turbo Pascal lehetőségei, részben olyan alkalmazási példák, amelyek hasznos építőköveként beépülhetnek majd saját programjainkba is.

Az utolsó fejezetben egy összetettebb program elkészítését mutatjuk be lépésről lépésre, ezáltal is a programozási gyakorlatot szeretnénk erősíteni.

A könyv nem törekszik sem a Turbo Pascal, sem a programozás teljes ismeretkörének bemutatására. Nem szerepel tehát e lapokon a nyelv összes eljárása és függvénye, és hiába keresi a kedves olvasó benne a magas szintű programozási ismereteket. Nem része továbbá a könyvnek a Turbo Pascal integrált fejlesztő környezetének ismertetése sem.

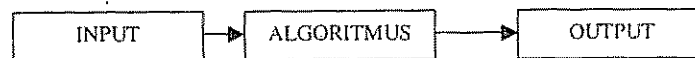
Még egy megjegyzés: a könyv használatához megkívántatik némi gépkezelői alapismeret (programok indítása, kész anyagok mentése, betöltése, ...stb.). Ez nem szerepel a könyvben, de a könyv végén található néhány olyan könyv ajánlása, amely hasznos segítséget nyújthat ezek megismeréséhez is.

1. A programozás elmélete

1.1. Bevezetés

A számítástechnikában, hasonlóan az élet más területeihez, folyamatosan valamilyen problémát kell megoldanunk. A probléma megoldása során az általunk ismerttől - megfelelően összekapcsolt - műveletekkel jutunk el az ismeretlenig. Ezt a megoldási menetet nevezzük algoritmusnak.

Az algoritmus tehát nem más, mint műveletek megfelelő módon való összekapcsolása. Az ismert adatokat INPUT-nak, az ismeretlen adatok OUTPUT-nak nevezzük.



Gyakran előfordul, hogy egy feladat első ránézésre egyszerűnek tűnik, azonban egy kis gondolkodás után rájöhethetünk, hogy a feladat nem is olyan egyszerű. Érdemes elgondolkodnunk azon, milyen bonyolult algoritmust kell végrehajtani ahhoz, hogy telefonáljunk. Első pillanatokban nem is gondolunk arra, hogy mit kell tennünk, ha foglalt a vonal vagy ha nem veszik fel a telefont stb. A mindennapok, a gyakran ismétlődő feladatmegoldás során ezek a felmerülő problémák számunkra már nem okoznak gondot, hiszen nagyon sokszor megoldottuk már, automatikusan cselekszünk ilyen helyzetekben. Másik példa, talán mindenki emlékszik arra, mikor először beült egy kocsiba vezetőként. Először komoly gondot okozhatott már a pedálok használata is, de később már a sebességváltás is könnyedén ment és mindez néhány kilométer vezetés után.

A számítástechnikában a programozó készíti el a probléma megoldásának eszközét. Ez az eszköz a program. A program egy adott feladat megoldására készített - számítógép által megértett - algoritmus alapján felépített utasítások összessége. Az adott programot fel kell készíteni arra, hogy az összes lehetőséget figyelembe vegye. Ne következzen be

az az eset – az előző példa alapján –, hogy a program ne tudjon mit csinálni akkor, ha netalántán az üzenetrögzítő kapcsol be. A programban nagyon részletesen meg kell adnunk, hogy a program mit is csináljon.

1.2. A program készítésének menete

Egy program elkészítése általában nem egy lépésben történik. Nézzük végig mi történik akkor, ha a főnök azt mondja a titkárnőjének, hogy hívja fel telefonon az egyik beosztottját. A titkárnő először megérti a problémát, majd eltervezi és végül végrehajtja a megoldási lépéssorozatot. Az algoritmus a következő fő lépésekből áll:

1. kinyitja a telefonregisztert
2. felemeli a telefonkagylót
3. tárcsáz, amíg a beosztott fel nem veszi a telefont
4. ha felvette a beosztott a telefont, bekapcsolja a főnökhöz
5. a telefonregisztert bezárja

1.2.1. A program elkészítésének lépései

Ahhoz, hogy egy programot megfelelő minőségben el tudjunk készíteni mindenképpen be kell tartani a program elkészítésének lépéseit, melyek a következők:

- elemzés
- tervezés
- végrehajtás
- ellenőrzés

1.2.2. Az algoritmus és szerkezete

A feladat megoldása során nagyon fontos a program algoritmusának elkészítése. A kérdés csak az, hogy ezt milyen részletességgel tegyük meg. Minden feladat további részfeladatokra bontható. Érdemes elgondolkodni azon, kell-e olyan problémákkal foglalkozni, mint:

Hol van a telefonregiszter?

Melyik oldalon van a keresett személy?

Elérhető-e telefonon?

1.2.2.1. Szekvencia

Nézzük meg az előző feladat 5 lépését (a főnök a beosztottal szeretne beszélni):

P1: a titkárnő kinyitja a telefonregisztert

P2: a titkárnő felemeli a telefonkagylót

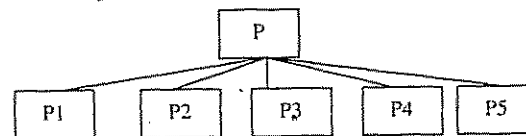
P3: tárcsáz, amíg a valaki fel nem veszi a telefont

P4: ha felvette valaki a telefont, kéri a beosztottat, és bekapcsol a főnökhöz

P5: a telefonregisztert bezárja

A probléma megoldása a részfeladatok egymás utáni megoldása.

A probléma megoldásának hierarchikus ábrája:



Ebben a feladatban az egyes részfeladatokat egymás után kell végrehajtani. A P1..P5 részprogramok szekvenciát (sorrendet) alkotnak.

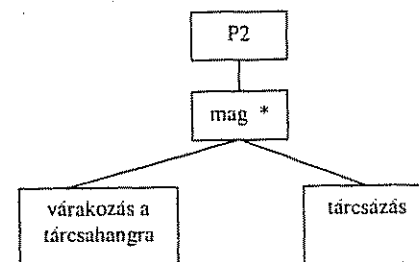
1.2.2.2. Iteráció

Gyakran előfordul, hogy bizonyos cselekménysorozatot többször kell végrehajtanunk.

Erre a példa a P3-as részfeladatot, ebben a feladatban egész addig kell tárcsáznunk, míg valaki fel nem veszi a telefont, ha senki sem veszi fel, később újra megpróbálhatjuk:

1. a kagyló felvétele után várjuk a tárcsahangot
2. ha megjött, tárcsázunk

Ezt a végrehajtási módot iterációnak nevezzük. Az iteráció magját az ismétlődő műveletek adják.



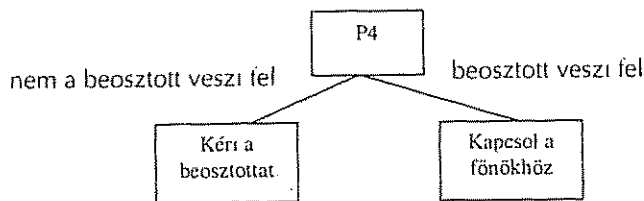
1.2.2.3. Szelekció

A mindennapi életben a döntés az egyik legfontosabb tevékenység. A döntés nem más, mint a dolgok közötti választás. A programozásban és az élet sok területén a választást szelekciónak nevezzük.

Példaként nézzük meg a P4-es részfeladatot részletesen:

1. ha valaki felvette a telefont, meg kell kérdezni ki az,
2. ha a beosztott, kapcsol a főnökhöz
3. ha nem a beosztott, akkor kérni kell a beosztottat.

Az a folyamatot, mikor kiválasztjuk, hogy a beosztott vette fel a telefont vagy sem, döntésnek nevezzük.



Az eddig bemutatott ábrákat program-szerkezeti vagy hierarchia-ábrának nevezzük, mivel a program hierarchikus szerkezetét tükrözi.

1.2.3. Az algoritmussal szemben támasztott követelmények

Teljesség: minden információt tartalmaznia kell, ami a feladat megoldásához szükség.

Egyértelműség: az algoritmus leírása legyen egyértelmű, minden lépés csak egyféleképpen legyen értelmezhető.

Befejeződés: az algoritmusnak véges számú lépésben be kell fejeződni.

Az algoritmus minden egyes lépése egy műveletet valósít meg. Az emberi olvasásra szánt algoritmusok tartalmazhatnak meglehetősen összetett lépéseket is (pl.: lemegyek telefonálni), de a számítógépes algoritmusnak mindenképpen rendkívül részletesnek kell lennie.

A programozás során alapvetően 4 féle algoritmus lépést alkalmazunk:

számítás: egyszerű numerikus, logikai vagy karakterművelet

döntés: számok, karakterek összehasonlítása, és ennek eredménye alapján egy vagy több alternatív lépés kiválasztása

input: számításra adatok bevétele

output: a kiszámított eredmények kiadása

1.2.4. Folyamatábrák

A hierarchia-ábra a probléma, illetve a megoldás szerkezetének ábrázolására alkalmas. Célszerű azonban egy másik ábrázolási módot alkalmazni, amely a folyamatot helyezi előtérbe és közérthető formájú, mégis félreérthetetlenül leírhatjuk vele az algoritmust. A végleges program-kód elkészítése előtt célszerű lehet egy korlátozott formájú magyar nyelvű leírást használni. Ezt a nyelvet pszeudonyelvnek nevezzük. A magyar nyelv sokszínűsége miatt ez a leírási mód nem biztos, hogy mindenki számára ugyanazt fogja jelenteni. Ezért az algoritmusok leírásához egy, a hétköznapi nyelvi formánál precízebb, ámde „emberi fogyasztásra alkalmas” formára is szükség van. Ezt valósíthatja meg az algoritmust kifejező, képi ábrázolás, a folyamatábra. Ez az ábrázolási módszer képi megjelenítési formája miatt áttekinthető és érthető.

Az algoritmus egy-egy lépésének leírásához a folyamatábrán egy-egy szimbólum felel meg. A lépések időben egymáshoz való viszonyát a szimbólumok síkbeli elhelyezkedésével, illetve összekapcsolásával ábrázolhatjuk.

Kezdő felhasználók számára mindenképpen célszerű a folyamatábra elkészítése és a folyamatábra alapján a program kódjának megírása, hiszen minden egyes szimbólum egy-egy utasításnak felel meg az adott programozási nyelven, így egy jól elkészített folyamatábra megadja a program kódját.

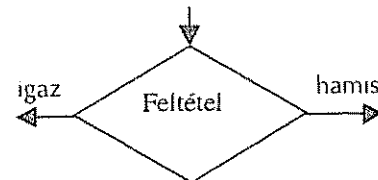
A folyamatábra szimbólumai:

Azt az algoritmus lépést, amelyet minden feltétel nélkül, egyszer hajtunk végre így jelöljük:



A téglalapba írjuk be az algoritmus lépést. A nyílak az algoritmusok időben lefolyását mutatják. Az ábrába bevezető nyíl előtt az időben korábban zajló folyamatok vannak, az ábrából kifelé mutató nyíl pedig a következő utasításokat mutatja.

A választás lehetőségét a hegyére állított rombuszsal ábrázoljuk. Ebben az esetben a választási lehetőségek száma kettő: igaz vagy hamis. Több egymás után következő rombusz esetében már bonyolultabb feltételrendszert is ki tudunk alakítani.



A rombusz belsejébe írhatjuk a választható feltételeket. Abban az esetben, ha az állítás hamis, akkor a „hamis nyíl” irányába fog az algoritmus folytatódni, ha igaz, értelemszerűen az „igaz nyíl” irányába.

1.2.5. A strukturált problémamegoldás

Egy adott feladatra a problémát megoldó algoritmust általában nem kapjuk meg készen, ezt nekünk kell elkészíteni, ki kell fejlesztenünk azokat. A probléma megoldása sikeresen elvégezhető strukturált módszerekkel.

A strukturált problémamegoldás három részből áll:

1. a probléma megfogalmazása és megértése
2. az algoritmus megtervezése
3. a program elkészítése

1. A probléma megfogalmazása és megértése: ez a rész segít megérteni az adott problémát, félreértéseket tisztázni és a nyitott kérdésekre válaszolni. Ebben a részben pontosan le kell írunk:

- a modell elemeit
- az inputok és az outputok tartalmát és formáját
- a feldolgozás fő elemeit, az adatok közötti összefüggéseket

2. Az algoritmus megtervezése: az első pontban leírtak alapján fejlesztjük ki az algoritmust, definiáljuk a modell változóit, leírjuk az adatok szerkezetét és kapcsolatát. A változók közötti kapcsolatok meghatározása alapján összeállítjuk a megoldás elemeit és azokat logikus rendbe fűzzük.

Az algoritmus elkészítése lépésenként finomítva történik: először általában valamilyen pszeudonyelven írjuk le az algoritmust, utalva az egyes algoritmus részek kapcsolatára., majd a feladatot részekre bontjuk. Ezt követi a részfeladatok pontosítása addig, míg az egyes elemek egyértelművé nem válnak, vagyis az adott programnyelven közvetlenül leírhatók nem lesznek.

3. A második lépés elkészülte után következik a program elkészítése, vagyis a kódolás. Ez nem más, mint az adott algoritmust megvalósító program elkészítése.

A strukturált problémamegoldás előnye, hogy az algoritmusok részekre bontásával bármilyen bonyolultságú probléma megoldható, ha megfelelő sorrendben hajtjuk végre a lépéseket. Előnye még, hogy az első és második lépés megfelelő minőségű elvégzése után a programozónak már csak a kódolással kell foglalkoznia és nem kell menetköz-

ben a program kódján módosítani. Ezen előnyök jelentősen csökkentik a hibalehetőségeket, leegyszerűsítik a programozást, valamint egy esetleges későbbi programmódosítás során a javítások végrehajtását.

Nézzünk egy feladatot a strukturált feladat megoldási módszerre!

Két általunk megadott numerikus érték (szám) közül azt a számot kell kinyerni a képernyőre amelyik nagyobb. Tervezzük meg azt az algoritmust amelyik bekéri tőlünk a két számot és eldönti, hogy az első szám vagy a második szám a nagyobb!

Jelöljük a két számot így: SZAM1, SZAM2. A nagyobbat tegyük a NAGYOBB változóba.

Amelyik szám nagyobb, írjuk ki a képernyőre.

Az algoritmus első közelítésben az alábbi lépésekből fog állni:

- SZAM1 és SZAM2 bekérése
- a nagyobb szám kiválasztása
- az eredmény kinyomtatása:

Az első és a harmadik lépés egyszerű, de a második már bonyolultabb, ezért ez további finomításra szorul:

ha $SZAM1 > SZAM2$, akkor $NAGYOBB = SZAM1$

ha $SZAM1 < SZAM2$, akkor $NAGYOBB = SZAM2$

Ezek után már bármilyen számunkra ismert programnyelven kódolhatjuk a feladatot.

1.2.6. A program készítésének folyamata

A probléma megfogalmazása, az algoritmus elkészítése és a kódolás után még további munkafázisok vannak:

- formális ellenőrzés
- program tesztelés
- éles használat

A formális ellenőrzés során az alkalmazott fordítóprogram segítségével az kódolás szintaktikai (nyelvhelyességi) ellenőrzése történik meg. Ha a kód szintaktikailag helyes, akkor a fordítóprogram a gép számára használható kódot készít. Abban az esetben, ha szintaktikai hibát talál a fordítóprogram, jelzi azt és a kódot nem hagyja gépi nyelvre lefordítani. A hiba javítása után újbóli ellenőrzéskor - ha minden hibát kijavítottunk - a fordítóprogram gépi nyelvre fordítja a programot.

A következő lépés a tesztelés. Ebben a fázisban olyan próbaadatokkal töltjük fel a gépet, amelyek lehetőséget biztosítanak a hibák feltárására (pl.: az előző feladat esetén a program nem fog helyesen működni, ha a két szám azonos). Itt még célszerű kevés adattal dolgozni, hiszen

az itt végzett munka jó program esetén hiábavaló. Ettől függetlenül ezt a lépést nem célszerű kihagyni, mert egy esetlegesen rosszul működő program hibáira fényt derít, mielőtt még élesben használnánk a programot. Ha ebben a fázisban hibásan működik a program a probléma megfogalmazásától újra kell kezdeni a munkát. Ez első ránézésre időrablónak tűnhet. Általában nem az egész feladatmegoldás rossz, hanem annak egy kis hányada. A hibák felderítése érdekében mindegyikre célszerű a korábban készített dokumentáció áttekintése és javítása.

Ha a teszteléssel végeztünk, jöhet az éles használat, amikor valódi adatokkal töltjük fel a programot. Ha a tesztelésünk lelkiismeretes volt és jó eredménnyel zárult, az esetek többségében az éles üzemeltetés sem fog gondot okozni. Bonyolultabb programok esetében azonban előfordulhat, hogy a tesztelés során van olyan programrész, amit nem teszteltük le és éles futtatásnál derül csak ki a hiba. Ekkor is vissza kell menni a feladat megoldásának elejéhez és ki kell javítani a programot.

A program írója, illetve tesztelője, ha mód van rá, két különböző személy legyen.

1.3. Programozási alapfogalmak

A programokat karakterenként írjuk. A programnyelvek általában rögzített karaktereket használhatnak.

Ezek a jelek:

az angol ABC betűi és az aláhúzásjel: A...Z, a...z, _

a számok: 0...9

speciális karakterek: +, -, *, /, =, <, >, (,), [,], .., ., ' , #, \$

A programok legkisebb egységei az illető nyelv szimbólumai. Ezek az elemek, melyekből egy programot fel lehet építeni:

- a programozási nyelv által előírt kulcsszavak
- a programozó által képzett nyelvi elemek (változók, azonosítók, konstansok)
- különböző műveleti és elhatároló jelek.

E szimbólumokból meghatározott szabályok szerint utasításokat lehet összerakni, amelyek valamilyen műveletet végeznek el a gépen. Az utasítások sorozata adja a megoldandó programot.

A fejezetben a példaprogramok TURBO PASCAL-ban és BASIC-ben íródtak, bemutatva ezáltal, mennyire mások és egyben mennyire ha-

sonlóak a különböző programnyelvek. A példaprogramok szemléltetik az adott probléma megoldását a különböző programnyelveken.

1.3.1. A programírás konvenciói

A nyelv szintaxisát azok a szabályok alkotják, amelyek a programozási nyelv elemeinek helyes használatát biztosítják. A megengedett nyelvi szerkezetek jelentéstartama a szemantika.

A programozási nyelvekben vannak bizonyos „kulcsszavak”, amelyeknek speciális jelentése van, ilyen szavak például az utasítások, függvények is. Ezek az úgynevezett fenntartott szavak, melyek csak igen korlátozott módon használhatók bizonyos feladatokra.

1.3.2. Az azonosítók

Programozás során gyakran szükséges olyan karaktersorozatot használni, amit a programozó definiál változónévként vagy egyéb azonosítóként. Programnyelvektől függetlenül ezek nem kezdődhetnek számjeggyel, nem tartalmazhatnak speciális karaktereket, valamint szóközt. Az azonosító maximális hossza programnyelvtől függő, de általában 8 karaktert használnak.

1.3.3. A konstans

Konstans olyan adat, amely a műveletek során nem változik meg, értéke állandó marad. A legtöbb programnyelvben többféle típusú konstans lehet:

- Egész konstans: pozitív, negatív számok, valamint a nulla: (pl.: 1, -134, 0)
- Valós konstans: minden olyan szám, amelyben tizedespont szerepel (pl.: 1,2 vagy -0,13)
- Karakter konstans: egyetlen karakter, ami lehet szám, betű, valamint speciális karakter
- Szó konstans: több karakterből álló karaktersorozat. Néhány karakter kivételével minden karakter használható.
- Boole-féle konstans: két lehetséges értéke van: igaz vagy hamis

1.3.4. A változók

A változó olyan objektum, amelynek értéke a program futása során változik. A változókra azonosítókkal hivatkozunk. A változó típusai hasonlóan programnyelvtől függően változhatnak, de megegyeznek a

konstansok típusaival. Egy adott típusú változó csak a neki megfelelő értékeket tartalmazhatja (nem lehet egy egész változóba egy szót tárolni), illetve művelet csak azonos típusú változók között lehetséges. Ha nem ilyen műveletet próbálunk végrehajtani, az típushibát fog okozni.

Minden programozási nyelvben vannak olyan függvények, amelyek a változók közötti típusváltást elvégzik, abban az esetben, ha ez lehetséges, ilyen átalakítás lehet például:

- számból karakter
- karakterből szám, ha a karakter csak számokat tartalmaz

Ezek programnyelvtől függenek.

1.3.5. kifejezések

A kifejezések konstansokból, változókból, műveleti jelekből illetve zárójelekből és függvényekből állhatnak. A zárójelek a műveletek végrehajtásának sorrendjét határozzák meg.

Példa kifejezésre:

$$(a+2) - (\cos(30^\circ) / b) * d$$

a leggyakrabban használt műveleti jelek:

- + összeadás
- kivonás
- * szorzás
- / osztás

1.3.6. A műveletek végrehajtásának sorrendje (prioritása)

A műveleteket mindig a matematika szabályai szerint kell végrehajtani.

Először mindig a zárójelben lévő műveletet kell végrehajtani.

A szorzás és az osztás magasabb prioritású művelet, mint az összeadás és a kivonás, tehát azokat előbb kell végrehajtani. A szorzás és az osztás, valamint az összeadás és a kivonás azonos prioritású. Azonos prioritású műveletek esetén a végrehajtási sorrend balról-jobbra történik. Az aritmetikai kifejezésekben a függvények egy számított értékkel térnek vissza, ezt az értéket fogja a kifejezésbe behelyettesíteni (a függvényekről később részletesebben lesz szó).

1.3.7. Boole-féle algebra

A programozás során nagyon gyakori feladat, hogy valamilyen adatok összehasonlításának eredményétől függően kell valamilyen utasítássorozatot végrehajtani. Egy összehasonlítás eredménye vagy igaz vagy hamis lehet, több választási lehetőség nincs. Sokszor akad olyan feladati is, hogy több adatot kell valamilyen logikai rendszer szerint összekapcsolni, ehhez nyújt segítséget a Boole-féle algebra ismerete.

A Boole-féle algebra három elemet tartalmazhat az összehasonlítások kombinálására:

AND (és): Akkor és csak akkor igaz, ha A és B is igaz,

OR (vagy): Akkor igaz, ha A vagy B igaz,

NOT (nem): Akkor igaz, ha A hamis,

A és B logikai értékek. Értékük vagy igaz (true) vagy hamis (false)

A logikai értékek ábrázolására igazságtáblázatot szoktak használni, amely tartalmazza a lehetséges alternatívákat.

Az AND (és) művelet igazságtáblája:

A	B	A és B
0	0	0
1	0	0
0	1	0
1	1	1

Az OR (vagy) művelet igazságtáblája:

A	B	A vagy B
0	0	0
1	0	1
0	1	1
1	1	1

A NOT (nem) művelet igazságtáblája:

A	Nem A
1	0
0	1

A műveletek prioritása:

Zárójelben lévő műveletek

NOT művelet

AND, OR művelet

Ahhoz, hogy két vagy több adatot össze tudjunk hasonlítani, meg kell határozni, milyen összehasonlító (reláció) műveletet alkalmazunk:

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

A reláció jelekkel aritmetikai kifejezéseket hasonlíthatunk össze, az összehasonlítás általános alakja:

„kifejezés1” „relációs jel” „kifejezés2”

pl.: a következő aritmetikai művelet csak akkor igaz, ha A nagyobb, mint B, ellenkező esetben hamis.

A > B

Az egyes kifejezésekben használhatók aritmetikai műveletek, ezeket célszerű zárójelbe tenni.

Pl.: (A-5) > (B*3)

Logikai műveleti jeleket csak logikai kifejezésekre lehet alkalmazni. Ha A, B és C numerikus változók, és azt akarjuk hogy a logikai művelet csak abban az esetben legyen igaz, ha A nagyobb B-nél és C-nél is azt így kell felírnunk:

(A > B) AND (A > C)

Nem így:

A > (B AND C),

mivel B és C aritmetikai értékek és nem logikai értékek.

1.3.8. Az értékadó utasítás

A számítások végrehajtásához a számítógépnek meg kell adni milyen műveletet hajtson végre, milyen adatokkal, és melyik változóba kerüljön. Az értékadás formája a következő:

Változó = kifejezés,

Pl.: A = 12 vagy
 C = 3*12*(R+39)*sin(30°)

Ahol az egyenlet bal oldalán lévő változóba kerül az egyenlet jobb oldalán lévő kifejezés értéke. Látható, hogy a kifejezésben szerepelhet

szám, függvény, változó és konstans is. Természetesen az értékadáskor a változó típusától függően megadhatunk karaktereket, szavakat, illetve logikai értékeket is.

BETU = 'P'
 SZO = 'De jó!'
 LOGIKA = true (igaz)

Az értékadás művelete programnyelvenként eltérő lehet.

1.3.9. Input és output

A fejezet elején már volt szó arról, hogy a probléma megoldása során az ismert (input) adatoktól jutunk el az ismeretlenig (output), megfelelő algoritmussal. Az input/output utasítások a környezettel való kapcsolattartást biztosítják. A kapcsolattartás billentyűzet, mágneslemez egységen, mágnesszalag egységen, illetve a képernyőn keresztül történhet.

1.3.9.1. Az input

Az input célja, hogy a program számára adatokat vigyünk be a gép memóriájába, amit a program futása során feldolgoz.

1.3.9.2. Az output

A program az input feldolgozása után valamilyen eredményre jut, ezt az eredményt nevezzük outputnak. Az output megjelenése a képernyőn vagy a nyomtatón, gyakran mágneses tárolóegységen történhet. Az output megjelenítése az esetek többségében formázottan történik, ez azt jelenti, hogy meg kell adni a megjelenített adat megjelenítési környezetét, milyen hosszú lehet a kiírandó karaktersorozat, milyen legyen a színe stb.

1.3.10. A soros programozás

Minden programnyelvben megvannak azok a szabályok, amelyek meghatározzák a program szerkezetét. Ilyen szabályok az utasítások szerkezete, megadásának sorrendje. A soros programozás nem biztosít lehetőséget döntések (szelekció), illetve ciklusok (iteráció) kezelésére.

Ettől függetlenül fontos ismernünk a soros programozás lehetőségét, hiszen az utasítások nagy része sorosan követi egymást, például egy cikluson belül is.

Az utasítók sorrendjére a programozás során nagyon oda kell figyel-nünk, ha két utasítást összecserélünk, az az egész program eredményé-re hatással lehet, nagyon valószínű, hogy a program nem azt fogja csi-nálni amit elvárunk tőle.

1.3.11. A programok szerkezete

A program írásának első lépése, hogy deklaráljuk a program számá-ra a számítógépbe bekerülő és a végrehajtás során használandó válto-zók helyét. Ez a deklaráció néhány nyelvben nem kötelező, de a pro-gramnyelvek többségében elkerülhetetlen (BASIC-ben nem szükséges). A deklarációs rész tartalmazza a konstansokat, valamint a program futási környezetének beállításait. Ezt a részt a program deklarációs ré-szének nevezzük. Ezek az utasítások a nem végrehajtandó utasítások körébe tartozik. Magasszintű programozási nyelveknél a fordítóprog-ram dönti el, hogy az adott változóhoz mekkora tárterületet foglal le, a változó típusa alapján. A tárolási helyekre az azonosítókkal hivatkoz-hatunk.

A deklarációs részt követik a végrehajtandó utasítások. Az utasításo-kon kívül itt célszerű elhelyezni a megjegyzéseket (commenteket), melyek a program működésének leírására szolgálnak.

Példák soros programozásra:

1, Egy általunk megadott szöveg kiírása a képernyő középső sorba.

A probléma megoldásának menete:

1. Változó deklarálása
2. Képernyőtörlés
3. Adat bekérése
4. Pozicionálás középső sorba
5. Az adat kiírása

A feladat megoldása BASIC nyelven:

```
CLS
INPUT „Kérem a megadott szöveget:”, szoveg$
CLS
LOCATE 12,1
PRINT szoveg$
```

A feladat megoldása PASCAL nyelven:

```
PROGRAM P1;
USES CRT;
VAR
Szoveg: string[80]
```

```
BEGIN
CLRSCR;
READ (szoveg);
CLSSCR;
WRITELN (szoveg);
END.
```

2, Másodfokú egyenlet megoldása. A program megoldása során nem foglalkozunk azzal, hogy az egyenletnek esetleg nincs megoldása.

A megoldáshoz szükséges képlet:

$$GYOK_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Megoldás:

A, B, C bekérése
GYOK1 és GYOK2 kiszámítása
GYOK1 és GYOK2 kiírása

Megoldás BASIC-ben:

```
CLS
INPUT „Kérem, adja meg az A-t:”: A
INPUT „Kérem, adja meg az B-t:”: B
INPUT „Kérem, adja meg az C-t:”: C
CLS
D = SQR ( B * B - 4*A*C)
GYOK1 = ( - B + D ) / ( 2 * A )
GYOK2 = ( - B - D ) / ( 2 * A )
LOCATE 5, 5
PRINT „ Az első gyök:”, GYOK1
LOCATE 7, 5
PRINT „A második gyök:”, GYOK2
```

Megoldás PASCAL-ban:

```
ROGRAM masodfoku;
{ A program nem foglalkozik azzal, ha az egyenletnek
nincs megoldása.
Nem megfelelő értékek esetén a program hibaüzenettel
leáll! }

USES crt;

VAR
{Változók deklarálása}
a, b, c : integer;
d, gyok1, gyok2: real;
```



```

BEGIN
  CLRSCR;
  WRITE('Kérem, adja meg az A értékét:');
  READLN(a);
  WRITE('Kérem, adja meg a B értékét:');
  READLN(b);
  WRITE('Kérem, adja meg a C értékét:');
  READLN(c);

  d := sqrt( b * b - 4 * a * c );
  gyok1 := ( - b + d ) / ( 2 * a );
  gyok2 := ( - b - d ) / ( 2 * a );

  WRITELN('Az egyenlet első gyöke: ', gyok1);
  WRITELN('Az egyenlet második gyöke: ', gyok2);
  READLN;
END.

```

1.4. Alternatívák és ciklusok kezelése a programban

A mindennapi életben az egyik leggyakoribb művelet a döntés. A cselekvések döntő többsége függ valamilyen környezeti elemtől. Pl.: Milyen az idő, mennyi ismeretünk van a döntéshez stb. Hasonló módon a programozás során is használni kell döntési mechanizmusokat. Ezek használata bizonyos probléma megoldásoknál elkerülhetetlen. A programozás során egy vagy több változó összehasonlításából jön ki egy logikai eredmény (igaz vagy hamis), a logikai eredmény alapján fut az A vagy a B algoritmus felé a program.

A logikai eredmény aszerint igaz vagy hamis, hogy a két vagy több érték közötti összehasonlítás során milyen eredményre jutunk.

Pl.:

ha $A=3$ és $B=6$, akkor

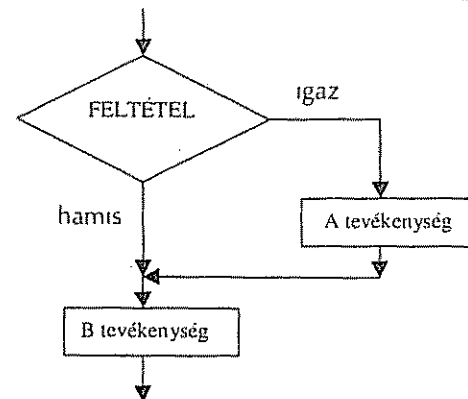
$A > B$ eredménye HAMIS

$A < B$ eredménye IGAZ

Az összehasonlítások között a Boole-féle algebra elemei használhatók.

1.4.1. Egyágú döntések

Az döntések lehetnek egyágúak abban az esetben, ha azt akarjuk, hogy az A algoritmus csak akkor hajtsdjon végre, ha a feltétel teljesül, abban az esetben, ha a feltétel nem teljesül, fusson tovább a program, de az A algoritmust ne végezze el.



Feladat: Kérjen be a program egy számot, és csak akkor írja ki az üzenetet („A szám nagyobb tíznél”), ha a szám nagyobb, mint tíz.

A program megoldása BASIC-ben: *

```

CLS
INPUT „Kérem, adjon meg egy számot:”, szam
IF szam > 10 THEN
  PRINT „A szám nagyobb tíznél”
END IF

```

A program megoldása PASCAL-ban:

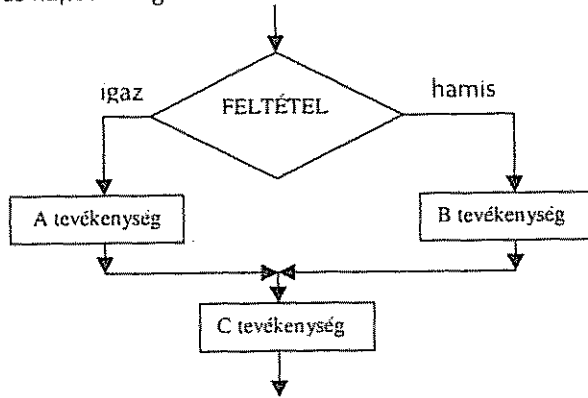
```

PROGRAM nagyobb10;
{ Ha 10-nél nagyobb számot adunk meg a program kiírja azt }
USES crt;
VAR
  szam : integer;
BEGIN
  CLRSCR;
  WRITE('Kérem, adjon meg egy 10-nél nagyobb számot:');
  READLN(szam);
  IF szam > 10 THEN
    WRITELN('A szám nagyobb, mint 10!');
  READLN;
END.

```

1.4.2. Kétágú döntés

Abban az esetben beszélünk kétágú döntésről, ha a feltétel IGAZ voltakor az A algoritmus hajtódik vége, a feltétel HAMIS voltakor a B algoritmus hajtódik végre.



Feladat: Kérjük be egy személy életkorát. Ha több mint 18 éves, írjuk ki, hogy a személy nagykorú, ha nincs még 18 éves, írjuk ki, hogy nem nagykorú.

A program megoldása BASIC-ben:

```

CLS
INPUT „Kérem adja meg az életkorát:”, kor
IF kor > 18 THEN
  PRINT „Nagykorú”
ELSE
  PRINT „Nem nagykorú”
ENDIF
  
```

A program megoldása PASCAL-ban:

```

PROGRAM nagykoru;
USES crt;

VAR
  kor : integer;

BEGIN
  CLRSCR;
  WRITE('Kérem adja meg a személy életkorát:');
  READLN(kor);
  IF kor > 18 THEN
    WRITE('A személy nagykoru.')
  
```

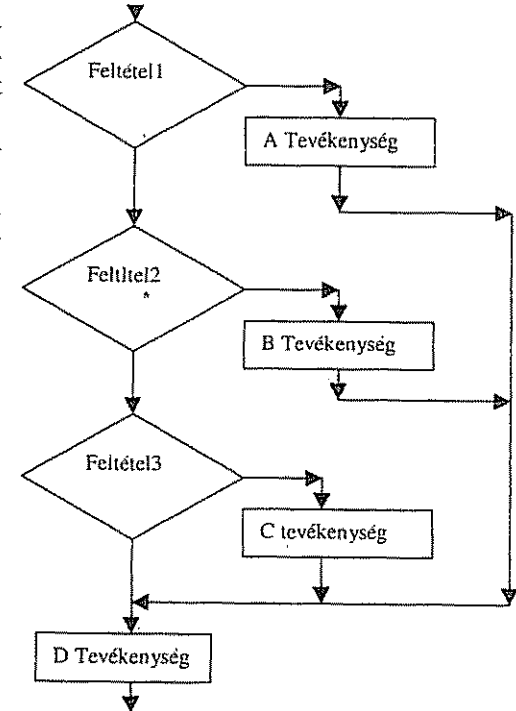
```

else
  WRITE('A személy meg nem nagykoru!');
  READLN;
END.
  
```

1.4.3. Több lehetséges érték közötti választás

A választások lehetősége sok esetben azonban nem egy vagy kettő, hanem esetenként akár több száz is lehet. Ezt többágú szelekciónak nevezzük. Erről akkor beszélünk, ha a feltétel1 teljesülése esetén az A algoritmus, feltétel2 teljesülése esetén a B algoritmus, feltétel3 teljesülése esetén a C algoritmus hajtódik végre.

A különböző programnyelvek ennek megvalósítására az úgynevezett CASE szerkezetet használják. Ez a szerkezet jelentősen meggyorsítja és egyszerűsíti a programozási munkát.



Feladat: A program kérje be tőlünk az osztályzatokat és írja ki azokat betűvel.

- 1 – elégtelen
- 2 – elégséges
- 3 – közepes
- 4 – jó
- 5 – jeles

A feladat megoldása BASIC-ben:

```

CLS
INPUT „Kérem, adja meg az osztályzatot:”, osztalyzat
CLS
SELECT CASE osztalyzat
  CASE 1
    PRINT „Elegtelen”
  CASE 2
    PRINT „Elegseseges”
  CASE 3
    PRINT „Közepes”
  CASE 4
    PRINT „Jó”
  CASE 5
    PRINT „Jeles”
END SELECT

```

A feladat megoldása PASCAL-ban:

```

PROGRAM osztalyzat;
USES crt;
VAR
  osztalyzat: integer;
BEGIN
  CLRSCR;
  WRITE('Kérem, adja meg az osztályzatot:');
  READ(osztalyzat);
  CASE osztalyzat OF
    1: WRITE('Elegtelen');
    2: WRITE('Elegseseges');
    3: WRITE('Közepes');
    4: WRITE('Jó');
    5: WRITE('Jeles');
  END;
  READLN;
END.

```

1.4.4. Feltétel nélküli vezérlés átadás

Ez a programozási megoldás nem túl szerencsés, ha lehet a használatát kerülni kell. A lényege, hogy egy ugró utasításban megadjuk honnan folytassa a program a futását. A vezérlés átadás címke használatával történik.

1.4.5. A ciklikus tevékenység (iteráció) kezelése a programban

A számítógépet eleinte adatfeldolgozásra használták, a feldolgozása nagy mennyiségű adattal, de általában azonos művelettel történik. Mindennapi példa egy gyár dolgozóinak fizetésemelésére. Ha a gyárban 1000-en dolgoznak nem szükséges a programban 1000-szer egymásután kiadni azt a parancsot, hogy a dolgozó fizetését emelje fel, hanem egyszer kell ezt megtenni, de a végrehajtást 1000-szer megismételni. Ezt nevezzük ciklusnak (iterációnak). Az hogy ezt a műveletet hányszor kell végrehajtani a dolgozók létszáma határozza meg. A ciklus egészen addig tart míg az úgynevezett ciklusváltozó el nem éri a dolgozók számát. A ciklusváltozó tehát, nem más mint egy érték, amelytől függ a ciklus végrehajtásának száma. Előkészítés során ezt a ciklusváltozót is be kell állítani.

A ciklus felépítése:

- előkészítés
- ciklusmag
- ciklusfeltétel vizsgálata

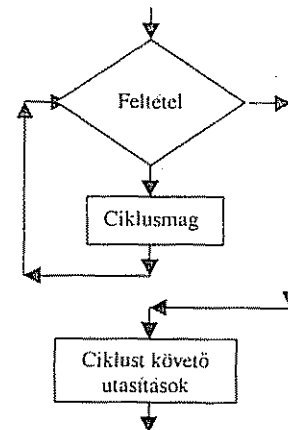
A ciklusmag magában foglalja a végrehajtandó utasításokat, valamint a ciklusváltozó értékének módosítását.

A ciklusokat működési módjuk szerint három csoportba sorolhatjuk:

- előtesztelő
- hátultesztelő
- növekményes

1.4.5.1. 4.5.1 Előtesztelő ciklus

A nevében is benne van, csak akkor fogja a ciklusmagban lévő utasításokat végrehajtani, ha a ciklusba a belépési feltétel teljesül. Ez azt jelenti, hogy a ciklus addig fog végrehajtódni, míg a belépési feltétel igaz. Abban az esetben, ha a belépési feltétel már nem igaz, a vezérlés átadódik a ciklus utáni első utasításra.



1. feladat: A program addig kérjen be számokat, míg a számok összege nem lesz nagyobb 1000-nél.

A feladat megoldás BASIC-ben:

```
CLS
szam = 0
osszeg = 0
DO WHILE osszeg < 1000
  CLS
  INPUT „Adjon meg egy számot:”, szam
  osszeg = osszeg + szam
LOOP
PRINT „A számok összege:”, osszeg
END
```

A feladat megoldása PASCAL-ban:

```
PROGRAM _1000ig;
{ A program addig kér be számokat, míg azok összege
nem lesz nagyobb 1000-nél }
USES crt;
VAR
  szam, osszeg : integer;
BEGIN
  CLRSCR;
  osszeg := 0;
  szam := 0;
  WHILE osszeg < 1000 do
    BEGIN
      CLRSCR;
      WRITE('Kérem, adjon meg egy számot:');
      READLN(szam);
      osszeg := osszeg + szam;
    END;
  WRITE('A számok összege:', osszeg);
  READLN;
END;
```

2. feladat: A program kérjen be tőlünk számokat addig, amíg a bekért szám nagyobb nem lesz 100-nál.

A feladat megoldás BASIC-ben:

```
CLS
DO WHILE szam < 100
  INPUT „Kérem adjon meg egy számot:”, szam
LOOP
```

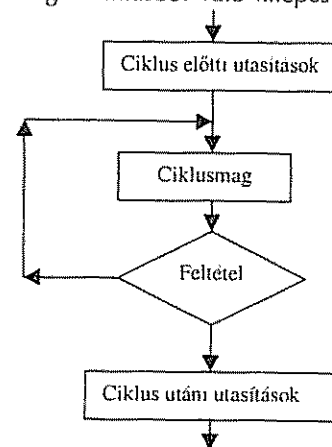
A feladat megoldása PASCAL-ban:

```
PROGRAM nagyobb100;
{ Ha 100-nál nagyobb számot adunk meg, a program befejezi a futását }
USES crt;
VAR
  szam : integer;
BEGIN
  CLRSCR;
  szam := 0;
  WHILE szam < 100 DO
    BEGIN
      CLRSCR;
      WRITELN('Ha 100-nál nagyobb számot adunk meg a programnak befejeződik a futása!');
      WRITE('Kérem, adjon meg egy számot:');
      READLN(szam);
    END;
  END.
```

1.4.5.2. Hátultesztelő ciklus

A hátultesztelő ciklus abban különbözik az előtesztelőstől, hogy a ciklus egészen addig végre fog hajtódni míg a ciklusból való kilépési feltétel nem teljesül. A ciklus során a ciklusmag egyszer legalább le fog futni, hiszen csak a ciklusmag lefutása után fog a kilépési feltétel vizsgálata megtörténni, ellentétben az előtesztelő ciklusnál, ahol a ciklus csak akkor futhat le egyszer, ha a belépési feltétel teljesül.

Feladat: Készítsünk átlagszámító programot! A program addig kérje be tőlünk a számokat, míg nullát nem adunk meg. Ha nullát adtunk meg, számolja ki és írja ki a képernyőre az átlagot.



A feladat megoldás BASIC-ben:

```
CLS
szam = 0
darabszam = 0
osszeg = 0
DO
  INPUT „Kérem, adja meg a következő számot:”, szam
  darabszam = darabszam + 1
  osszeg = osszeg + szam
LOOP UNTIL szam = 0
darabszam = darabszam - 1
atlag = osszeg / darabszam
PRINT „A beírt számok átlaga:”, atlag
```

A feladat megoldása PASCAL-ban:

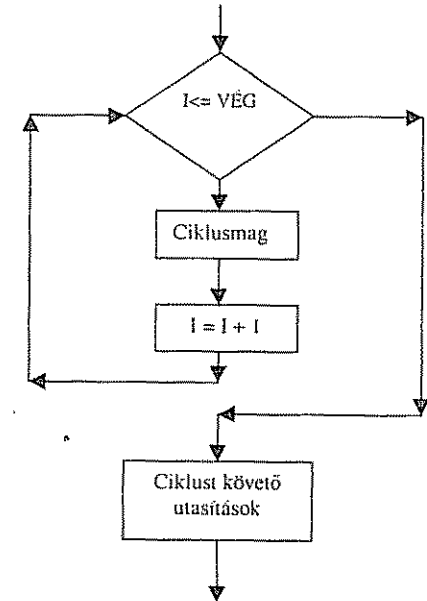
```
PROGRAM atlag;
{ A program addig kér be számokat, míg 0-t nem adunk
meg, majd a bekért
számoknak kiszámolja az átlag t }
USES crt;

VAR
  szam, osszeg, darabszam: integer;
  atlag: real;

BEGIN
  szam := 0;
  osszeg := 0;
  darabszam := 0;
  atlag := 0;
  REPEAT
    CLRSCR;
    WRITE('Kérem adja meg a következő számot:');
    READLN(szam);
    osszeg := osszeg + szam;
    darabszam := darabszam + 1;
  UNTIL szam = 0;
  darabszam := darabszam - 1;
  atlag := osszeg / darabszam;
  WRITE('A számok átlaga:', atlag);
  READLN;
END.
```

1.4.5.3. Növekményes ciklus

A ciklusmagban van egy olyan változó, amely a ciklus végrehajtása után változik (növekszik vagy csökken). Ebben a cikluskezelő formában meghatározhatjuk, hányszor fusson le a ciklusmag. A program a ciklusmagot mindaddig végrehajtja, míg a feltétel igaz, tehát a ciklusváltozó meg nem haladja a ciklus végértékét. Ezt a cikluskezelést FOR-NEXT féle ciklusnak nevezzük.



Feladat: 1 és 100 között adjuk össze a számokat és az eredményt írjuk ki a képernyőre.

A feladat megoldás BASIC-ben:

```
CLS
ciklus = 0
osszeg = 0
FOR ciklus = 1 TO 100
  osszeg = osszeg + ciklus
NEXT osszeg
PRINT „Az első 100 szám összege:”, osszeg
END
```

A feladat megoldása PASCAL-ban:

```
PROGRAM osszegzes;
{ Az első 100 szám összeget írja ki. }
USES crt;

VAR
  szam, osszeg: integer;

BEGIN
  CLRSCR;
  osszeg := 0;
  ciklus := 0;
  FOR ciklus := 1 TO 100 DO
```

```

BEGIN
    osszeg := osszeg + ciklus;
END;
WRITE('Az első 100 szám összege:', osszeg),
READLN;
END.

```

1.5. Tömbök

A számítástechnikai gyakorlatban, csakúgy mint a mindennapi életben, gyakran használunk olyan adatokat, amelyek összetartozóak. A hétköznapi életben ezek számsorok, táblázatok, névsorok alakjában jelennek meg. Ilyen lehet egy osztálynévsor, egy busz megálló stb., az ilyen táblázatokat egynemű (egy dimenziós) táblázatnak nevezzük. Gyakran találkozhatunk olyan táblázatokkal, amelyek több bemenete van. Ezeket több bemenetű (több dimenziós) táblázatnak nevezzük. Ilyen táblázat lehet az autóstérképek végen lévő táblázat, amelyből a városok távolságát kereshetjük ki egymástól. A táblázat adott sorának és oszlopának találkozásából olvashatjuk ki a két város távolságát. A példa egy kétbemenetű táblázat. Az ilyen szerkezetű adattárolás: tömböknek nevezzük.

1.5.1. A tömbök használata

A tömböt használata előtt deklarálnunk kell, a tömb típusát (szám, karakter, szöveg, logikai stb.), és a tömb nagyságát (hány sorból álljon).

A tömb elemeit indexek segítségével azonosítjuk. Az indexek 0-tól vagy 1-től kezdődnek, programnyelvtől függően. Az indexek segítségével érhetjük el a tömb adatait.

Nézzünk egy példát:

	1.	2.
Index	Név	Szakma
1.	Mekk Elek	Ezermester
2.	Vihar Viktor	Meteorológus
3.	Csőp Csaba	Vízvezeték szerelő
4.	Bú Béla	Humorista

Az első sorban az oszlopok száma van megadva.

1.5.2. Tömb elemeire való hivatkozás

A tömbök használata során elkerülhetetlen, hogy a tömb elemeivel valamilyen műveletet végezzünk, ehhez az szükséges, hogy az tömb elemeit el tudjuk érni. Ezt úgy tehetjük meg hogy a tömb neve után megadjuk az elérni kívánt elem sorszámát és oszlopszámát. A sor- és oszlopszám kereszteződésében találhatjuk meg az elemet.

Ha a tömb 3. Sorára és annak 2. Oszlopára akarunk hivatkozni („Vízvezeték szerelő”) azt a következő módon tehetjük meg:

Tömb (3, 2)

1.5.3. Tömb feltöltése

Ahhoz, hogy egy tömböt használni tudjunk, a tömböt fel kell tölteni elemekkel. Ehhez szükség van az elem tömbbeli helyére és a feltöltendő adataira.

A tömb elemeinek értékadása a következőképpen történhet:

Tömb (3, 2) = „esőjós”

1.5.4. Tömb elemeinek kiírása

A tömb értékeit kiírhatjuk, de itt is meg kell adni, melyik elemet akarjuk kiírni:

```
print (tömb (1, 2)),
```

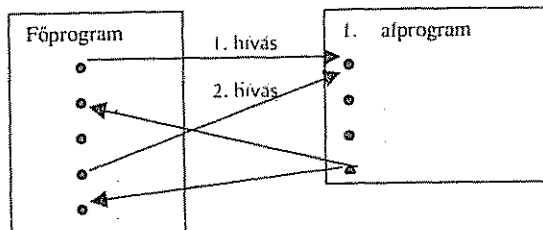
ekkor az utasítás a „Ezermester”-t fogja megjeleníteni.

1.6. Alprogramok

Az eddig bemutatottak már elviekben bármilyen bonyolultságú program végrehajtását lehetővé teszik, ez azonban gyakorlatban nem teljesen igaz. Vannak olyan problémák, melyek az eddig tanultak alapján csak nehezen és bonyolultan valósíthatók meg. Gondoljunk egy korábban már említett példára a telefonálásra. Ha egész nap telefonálunk, akkor ezt ciklikusan megtehetnénk – tegyük fel, hogy mindig ugyanazt a számot hívjuk –, de mi a teendő, ha a telefonálások közben valami mást is kell csinálni, például le kell gépelni egy levelet. Ha ezt lefordítjuk a számítógép nyelvére, arra az eredményre jutunk, hogy egy állandóan ismétlődő feladat többször is szerepelni fog a program kódjába. Ez nemcsak megnöveli a program hosszát, hanem a tesztelések során sem lesz könnyű egy esetleges hiba felderítése. E probléma meg-

oldására találták ki az alprogramokat. Az alprogramot a program során többször egymástól függetlenül végrehajtandó feladatok végrehajtására kell használni.

Ha egy alprogramot használni akarunk, akkor az alprogramot meg kell hívni.



Az alprogramoknak általában maximum 7 további szinten al-programja lehet. Ezeket az al-alprogramokat az alprogramból kell meghívni.

Azt, hogy az alprogram, milyen adatokkal, illetve változókkal dolgozik, az eljárás paramétereivel lehet megadni.

Az alprogramoknak két csoportjuk van:

- függvények
- eljárások

1.6.1. Függvények

A függvények olyan alprogramok, melyek egy vagy több megadott argumentum megadásával számolnak ki egy eredményt. A függvények használatának ez a korlátozása bizonyos feladatok alkalmazására lehetetlenné teszi. A függvény használatának van azonban egy nagy előnye, hogy a függvény beilleszthető bármilyen kifejezésbe. A függvényben zárójelben lévő értékek (x,y,z) a függvény argumentumai:

$$C = \text{függvény}(x, y, z) * 3$$

Ez adott esetben jelentős könnyítést eredményezhet.

A függvényt, mint egyfajta alprogramot használatkor meg kell hívni.

1.6.2. Eljárás

Az eljárás és a függvény közötti legnagyobb különbség az, hogy az eljárás több eredményt is visszaadhat, viszont az eljárás nem használható kifejezésekben.

1.6.3. A főprogram és az alprogram kapcsolata

A főprogram és az alprogram együtt alkotják a programot. A kettő kapcsolatát az adott programnyelv határozza meg. Főprogram nélkül nem lehet használható alprogramot készíteni.

1.6.4. Az azonosítók hatásköre

Az alprogramokban, hasonlóan a programokhoz, definiálhatunk azonosítót. Az ilyen azonosítót lokális változónak nevezzük, csak az adott alprogramban, illetve az alprogram al-alprogramjaiban használhatók. Ha egy főprogramban definiáltunk egy azonosítót, és az alprogramban is definiáltunk egyet ugyanolyan névvel, akkor az alprogramban az ott definiált változó lesz az érvényes, de mihelyt kilépünk az alprogramból, ismét a főprogram által definiált változó lesz érvényes. A főprogramban definiált azonosítókat globálisnak nevezzük, és ezeket a program futása során bármikor elérhetjük, kivéve az előbb említett esetet.

2. A programozás alapjai Turbo Pascalban

2.1. Az első program, avagy ismerkedés a nyelvvel

2.1.1. A Turbo Pascal programozási nyelvről

2.1.1.1. Program és nyelv

Legelőször is arra a kérdésre kellene választ adnunk, hogy mi a program. A program (logikus) utasítások sorozata, amelyeket a számítógép megért, és egy jól meghatározott sorrendben végre is tud hajtani.

Ahhoz, hogy a számítógép megértse az utasításainkat, egy programozási nyelvet kell igénybe vennünk. Programozási nyelvekből rendkívül sokféle van, ezért nem árt valamiféle csoportokba sorolnunk őket. A Turbo Pascal a magas szintű nyelvek közé tartozó, általános célú programozási nyelv. A „magas szintű” kifejezés annyit jelent, hogy a programozási nyelv inkább a felhasználóhoz áll közelebb, mint a géphez, vagyis az emberi gondolkodásmódot közelíti inkább. Másik jellemzője, hogy sok olyan feladat van, amit nem kell saját magunknak elvégeznünk, mert a program készítői¹ már megtették helyettünk. Például, ha egy kört akarunk rajzolni a képernyőre, akkor nem kell minden egyes kört alkotó pont helyét külön-külön kiszámolni, és a képernyőre kitenni, hanem elég csak annyit mondani: „rajzolj egy kört”. Az „általános célú programozási nyelv” pedig azt jelenti, hogy a nyelv nem valamely specializált feladat elvégzésére alkalmas², hanem általában programozásra.

¹ A Pascal programozási nyelvnek nagyon sok változata létezik. A Turbo Pascal a Borland Institute cég terméke, és talán a legkiforrottabb változata a nyelvnek.

² Néhány példa: a Clipper programozási nyelv adatbázisok mellé készíti azokat kezelő programokat. A Prolog logikai nyelv, és elsősorban relációk alapján felépített programok írhatók vele. De léteznek speciális programozási nyelvek a szimulációnak, és persze egyéb területnek is.

Bár jelen anyagban a Turbo Pascal program 7.0-ás változatával foglalkozunk majd, de a könyvből megismertek más verzióknál is jól használhatók. A leírtak tehát DOS alatti programfejlesztést tesznek lehetővé, a Windows valamely változata alá írt programok egy kicsit más alapokon épülnek fel. (A Borland cég a Borland Pascal nyelvet kínálja Windows-programok írására.)

2.1.1.2. A Turbo Pascal programok felépítése

Mint minden idegen nyelvnek, a programozási nyelveknek is nagyon szigorú szabályrendszere van. Ezek részben szintaktikai, részben szemantikai szabályok. Az előbbi a nyelv „nyelvtanát” jelenti, vagyis azt, hogy az egyes utasításokat pontosan hogyan kell leírni (kicsit olyan ez, mint az írott nyelvekben a helyesírás), az utóbbi a tartalmi szabályokat jelenti.

A Turbo Pascal-ban a programoknak nagyon kötött szerkezete van, ettől általában nem térhetünk el. Ez a szerkezet leginkább egy ételreceptre hasonlít: legelőször szerepel az étel neve, majd a hozzávalók pontos felsorolása, és a legvégén az étel elkészítésének menete. Itt is először a program nevét kell felírunk (bár ez elhagyható), majd az ún. deklarációs rész következik, amelyben fel kell sorolnunk a programunk „hozzávalóit”: típusokat, változókat, stb., és csak ez után következhet a program leírása.

```
Program   Programnév; (ez az ún. programfej)
          deklarációs rész
Begin    (ez vezeti be a program leírását)
          a program leírása
End.     (ezzel végződik a program)
```

Ez a legegyszerűbb programszerkezet, összetettebb programok esetén ez sokkal bonyolultabb lehet, de alapvetően minden program így néz ki. Az általam vastagon szedett szavak a TP saját szavai, ezeket minden programban betűről betűre így kell leírni.

2.1.1.3. A nyelv elemei

A programozási nyelv a következő elemekből épül fel:

- a nyelv által használt kulcsszavak (leggyakrabban utasítások, eljárások, függvények, blokk elejét és végét jelölő szavak, stb.)
- a programozó (azaz mi) által definiált saját azonosítók
- számok

- speciális feladattal bíró jelek (műveleti jelek, különböző zárójel-jelek, relációs jelek, illetve a = , , ' , # \$ ^ jelek)
- kommentek (megjegyzések)

Minden programsort karakterenként kell begépelni. A kis- és nagybetűk között a Turbo Pascal nem tesz különbséget. Ékezetes karakterek alapesetben nem használhatók, csak a képernyőre íratott szövegek vagy szöveges változók esetén.

A Turbo Pascal nyelv jellemzője – pl. a Basic-kel szemben –, hogy utasításkészlete rendkívül szegény, viszont gazdag tárházát kínálja a különféle alprogramoknak. Az alprogramok a programtól függetlenül megírt, és működő programocskák, amelyeket a programunkban felhasználhatunk (ilyen alprogram a program és nyelv fejezetben leírt „rajzolj egy kört!” is). Az alprogramok kétféleképpen lehetnek: eljárások és függvények. Az eljárások önállóan is szerepelhetnek egy programsorban, de a függvények csak kifejezésekben, hiszen a függvény olyan alprogram, amely valamely paraméterek alapján számolt értéket ad vissza a program számára. (Azaz egy eljárás végrehajtásakor az eljárást alkotó programot futtatjuk le, mintha lépésről lépésre beírnánk a programomba. Függvény esetén az általa szolgáltatott értékre van szükségünk.)

2.1.1.4. Programkönyvtárak (unitok) szerepe

Eljárásokból és függvényekből rengeteg áll a rendelkezésünkre. Miután a programok nagy részében a legtöbbjükre nincs is szükség, ezért ezek nem töltődnek be a memóriába a Turbo Pascal programmal együtt. Külön programkönyvtárakban – ún. unitokban – vannak elhelyezve, általában tematikusan csoportokba gyűjtve. A legfontosabb unitok a következők:

System	a legfontosabb eljárások és függvények
Crt	a karakteres képernyőt kezelő eljárások és függvények
Graph	a grafikus képernyőt kezelő eljárások és függvények
Dos	a DOS funkcióit kezelő eljárások és függvények
Printer	a nyomtatás
Overlay	a Turbo pascal átlapolásos technikáját kezelő eljárások és függvények

A System unit (a programkönyvtárak eredeti neve) eljárásai és függvényei bármikor a rendelkezésünkre állnak, de egyéb unitokban lévő eljárásokhoz csak akkor férünk hozzá, ha a program elején (a deklarációs részben) megadjuk a használni kívánt unitot a **USES** kulcsszó után.

2.1.1.5. Azonosítók

A program szinte elképzelhetetlen azonosítások nélkül. Ilyen a program vagy a unitok neve, a változók és konstansok nevei, saját eljárások és függvények nevei, címkek, stb. Minden azonosítóra a következő szabályok érvényesek:

- tetszőleges hosszúságú lehet, de csak az első 63 karaktert vesz figyelembe a program,
- tartalmazhat betűket (az angol ABC kis- és nagybetűit), számokat, és aláhúzás karaktert, de nem tartalmazhat szóközt, egyéb jeleket és ékezetes betűket,
- az azonosító első karaktere csak betű lehet,
- azonosítónak nem adhatók a Turbo Pascal ún. védett szavai, amelyek jegyzéke a **Mellékletben** megtalálható.

2.1.1.6. Forrás és lefordított programok

A Turbo Pascal nyelven elkészített program – vagyis a forrásprogram – a processzor számára érthetetlen, hiszen az csak a saját nyelvén, az úgynevezett gépi kódon ért. Ahhoz, hogy a programunk lefusson, vagyis a számítógép végrehajthassa, ahhoz először le kell fordítani a programot gépi kódra. A programozási nyelvek kétféle fordítóprogramot tartalmazhatnak: interpretert (értelmezőprogramot), illetve compilert (fordítóprogramot). Az interpreter a programot soronként fordítja és azonnal végrehajtja, míg a compiler a teljes programot lefordítja, majd egyszerre futtatja le (Ez utóbbit azonban a későbbiekben bármikor máskor fordítás nélkül is lefuttathatjuk). A Turbo Pascal nyelv – és a BASIC kivételével a magas szintű programozási nyelvek általában – compilert tartalmaz. Azaz a program megírása után először fordítanunk kell, és csak utána tudjuk futtatni azt.

A forrásnyelvi (általunk írt) program általában a PAS kiterjesztést kapja a mentés során.

A lefordított program kerülhet a memóriába, de fordíthatjuk a lemezre is. Ekkor futtatható, EXE kiterjesztésű állomány készül³, amelynek nevét DOS parancssorból kiadva a programunk a Turbo Pascal rendszer nélkül is lefut, sőt azt más gépre átmásolva is használhatjuk.

³ A TP 4.0 előtti verziók a fordítás során COM kiterjesztésű állományt készítettek. A COM állományok egyetlen memóriaszegmenst használhattak csupán, ezért méretük nem haladhatja meg a 64 Kb-ot. Az EXE állományok már fejlettebben, itt már nincs ez a megkötés.

2.1.1.7. Fordító direktívák

A programban elhelyezhetünk a fordítóprogramnak szánt utasításokat, amelyeket fordító direktíváknak nevezünk. Ezek általában a dollárjelből (\$) és egy betűből állnak, amelyeket kapcsos zárójelek között kell megadnunk. Pl. a {\$I+} forma az I direktívát bekapcsolt állapotúvá teszi.

A fordítási direktívák tárgyalására nem térünk ki ebben a könyvben, de néhány általános megjegyzést teszünk. A direktívák lehetnek globálisak és lokálisak. A globális direktívák az egész programra hatással vannak, és a deklarációs részben kell szerepeltetni őket. A lokálisak csak a programnak abban a blokkjában, ahová beírjuk.

A direktívákat három csoportba oszthatjuk: kapcsoló direktívák, amelyek lehetnek aktívak (+) vagy passzívak (-); paraméteres direktívák, amelyeknél a betűjelet a paraméter követi (pl. külön fájlba elkészített modul beszerkesztése a forráskódba); illetve feltételes direktívák, amelyek a fordítóprogram menetét adják meg (pl. lehetőségünk van géptípustól függően eltérő programrészeket fordítani).

Általában a fordítási direktívák megadásával ekvivalens menüpontokat találhatunk a Turbo Pascal integrált fejlesztői környezetében (pl. az Options menüben).

2.1.2. Az első program elkészítése

2.1.2.1. A Turbo Pascal indítása

Amennyiben a program telepítve van a merevlemezre, úgy a programot a prompt után írt „turbo” paranccsal indíthatjuk:

```
C:\> turbo
```

A program indítása után a képernyőn megjelenik az a munkaterület, ahová a programot gépelhetjük (ez többnyire kék alapú üres terület). A képernyő felső sorában menüket találunk, ezek segítségével tudunk majd menteni, futtatni, stb. A képernyő alsó sora az információs sor, amely a használható billentyűkről ad felvilágosítást.

2.1.2.2. A program begépelése

Gépelj be a következő programot, és ügyelj a pontosságra: minden karakternek szerepe van.

1. példa Írassuk ki a nevünket a képernyőre!

```
Program NevKiir;  
Begin  
Write('Zseni Alfonz');  
End.
```

A Turbo Pascal program a kis- és nagybetűket nem különbözteti meg egymástól, ezért tetszőlegesen lehet gépelni; ki-ki ahogy szereti. A könyvben általában nagy kezdőbetű után kisbetűket alkalmazunk.

A gépelés során látható, hogy bizonyos szavak más-más színnel kerülnek a képernyőre. Alapvetően a sárga színnel írunk, de a Turbo Pascal védett szavai fehérrel jelennek meg – ezeket félkövér betűkkel jelöljük a könyvben⁴.

A **Program** NevKiir; sor a programunk nevét adja meg (NevKiir). A program neve tetszőleges lehet, de nem tartalmazhat szóközt, nem tartalmazhat jeleket (kivéve az aláhúzás karaktert), nem tartalmazhat ékezetes karaktereket, és nem kezdődhet számmal. (A nagy K betűvel a szöközt igyekeztem pótolni.) Ez a sor elmaradhat, nem kötelező megadni.

A **Begin** jelzi a program kezdetét, az **End** a végét. A program végét továbbá egy pont is jelzi az **End** után.

A képernyőre írást egy eljárás, a **Write** valósítja meg. Minden eljárásnál zárójelek között kell megadni a paramétereket, vagyis jelen esetben azt, hogy mit írjon ki. Amit kiíratunk, az kétféle lehet (legalábbis kezdetben): szám, vagy szöveg. A kettőt meg kell különböztetnünk egymástól, ezért a szövegeket aposztrófok között kell szerepeltetni.

A **Write** eljárás a System unitban van, ezért ezt nem kell deklarálnunk.

Az egyes sorokat a ; (pontosvessző) zárja le. A **Begin** után nem kell kitenni, mindenhol máshol kötelező.

2.1.2.3. Megjegyzések elhelyezése a programban

Lehetőségünk van arra, hogy a programban kommenteket, azaz megjegyzéseket helyezünk el. Ezeket a fordítóprogram nem veszi figyelembe, csak saját magunknak szól. Iól alkalmazhatók egy-egy sor magyarázatára, esetleg az egyes programrészek tagolására.

A kommentek tetszőleges hosszúságúak lehetnek, és kapcsos zárójelek közé kell tenni őket. Használható még a komment elejének jelzé-

⁴ Ezek a színek a Turbo Pascal alapbeállításaira igazak, de megváltoztathatók. Ezért előfordulhat, hogy más színekkel találkozunk azon a gépen, amelyikkel dolgozunk.

sére a „(*)” (nyitó zárójel és csillag). Ilyenkor a komment végét a „(*)” (csillag és záró zárójel) jelzi.

Időnként szükség lehet arra, hogy egyes programsorokat ideiglenesen kivegyünk a programból. Ha kitöröljük, esetleg később elfelejtjük visszatenni, és az is lehet, hogy nem emlékszünk pontosan arra, hogy hogyan is néztek ki ezek a sorok. Ilyenkor célszerű a kívánt programrészt kapcsos zárójelek közé tenni, mivel az ilyen programrészeket a fordítóprogram nem veszi figyelembe, a program futása szempontjából olyan, mintha ott sem lenne. Amikor pedig újra szükségünk lesz ezekre a sorokra, elég csak a kapcsos zárójeleket kitörölni.

2.1.2.4. A program elmentése

Ha a programunkat begépeltük, szerencsés elmenteni valamilyen háttértárolóra (winchesterre vagy floppy-ra). Ehhez a File menü Save menüpontját (a továbbiakban ezt így jelöljük: File/Save) kell választani, amelyet háromféleképpen lehet megtenni:

Ha van, akkor egérrel egyet kattintunk a File menün, majd egyet a Save menüponton is.

Billentyűzettel bármely menüpont kiválasztható az ALT-kezdőbetű (a kezdőbetű általában eltérő színű) billentyűkombinációval. A lenyíló menüből a kiemelt betű lenyomásával, vagy a nyilakkal lehet egy menüpontot kiválasztani (az utóbbi esetben egy ENTER-re is szükség van).

A harmadik módszer az úgynevezett gyorsbillentyű használata. A megfelelő menüpont mellett feltüntetett billentyű vagy billentyűkombináció ez, amely alkalmazásával sokkal gyorsabban lehet elérni a kívánt célt. A képernyő alján az információs sorban is a használható gyorsbillentyűk láthatók. A mentés gyorsbillentyűje az F2.

A megjelenő ún. párbeszédpanelen először meg kell adnunk az állomány helyét a Files feliratú ablakban, ahová a TAB billentyűvel, vagy egérrel juthatunk el (pl. TP\). Majd a TAB billentyű ismételt lenyomásával visszajuthatunk a Save file as feliratú sorhoz, ahová az állomány nevét kell gépelnünk (pl. nevkiir). A Turbo Pascal programokat általában a PAS kiterjesztéssel látjuk el, aminek a megadása nem kötelező (ilyenkor a pontot sem szabad kitenni), a program megteszi helyettünk. Az ENTER lenyomásával a mentés végrehatódik.

Ha minden rendben volt, akkor a szerkesztő ablak tetején megjelenik a programunk neve.

2.1.2.5. A program fordítása és futtatása

A programot most már végre szeretnénk lefuttatni, ehhez azonban először le kell fordítanunk. A fordításhoz a Compile/Compile menüpontot kell választanunk, vagy az ALT-F9 billentyűkombinációt lenyomni. Ha a programunk hibátlan, akkor a megjelenő kis Compiling feliratú ablakban villog a „Press any key” (nyomj egy billentyűt) felirat. Ezzel a fordítás készen van.

Azt, hogy a lefordított program a memóriába kerüljön, vagy a lemezre EXE-fájl formájában, azt a Compile/Destination menüponttal állíthatjuk be. A lemezre fordítás lassabb, de kevesebb üres memória meglétét kívánja és a kész program a későbbiekben is használható.

A program futtatása a Run/Run menüpont, vagy a CTRL-F9 alkalmasításával kezdeményezhető. (Amennyiben a program még nem volt lefordítva, akkor a futtatás előtt automatikusan megtörténik a fordítás is.)

Most futtassuk le ezt a programocskát! Látszólag semmi nem történt, ezért jó lenne a szerkesztő képernyő „mögé” nézni, vagyis megtekinteni a program eredményét. Ezt a Debug/User screen menüpont, vagy az ALT-F5 billentyűvel tehetjük meg. A képernyőn megjelent a „Zseni Alfonz” felirat, vagyis a programunk rendben lefutott. Figyeljük meg, hogy a kurzor közvetlenül a kiírt szöveg után villog!

Egy tetszőleges billentyű segítségével visszatérhetünk a program szerkesztéséhez.

2.1.2.6. Kilépés a Turbo Pascalból

Ha befejeztük a munkát, akkor a File/Exit vagy ALT-X segítségével léphetünk ki. Amennyiben a programon az utolsó mentés óta módosítottunk, akkor a Turbo Pascal rákérdez, hogy akarjuk-e menteni. Erre a megfelelő válasz kezdőbetűjével (Y vagy N) lehet válaszolni.

Lépünk ki, majd indítsuk újra a programot (**turbo**)!

2.1.2.7. Program megnyitása lemezzel

Próbáljuk meg az elmentett programunkat megnyitni, azaz a lemezzel beolvasni. Ehhez a File/Open menüpont, vagy az F3-as billentyű szükséges. A megjelenő párbeszédpanel ugyanolyan, mint a mentésnél. Először válasszuk ki a megfelelő könyvtárat (a példánkban ez a TP volt), majd ugyanitt megjelennek a már itt lévő állományok, ezek közül válasszuk ki a megfelelőt (jelen esetben a **nevkiir.pas**-t). Természetesen megtehetjük, hogy a Name mezőbe beírjuk a nevet.

Most már az előző kis programunk újra szerkeszthető.

2.1.2.8. Egy kis módosítás...

2. példa Módosítsa a programot úgy, hogy a neve NevKiir2 legyen, és a Write eljárás helyett a Writeln szerepeljen! (Ekkor ugyanis a szöveg kiírása után sorváltás is történik.)

```
Program NevKiir2;
Begin
  Writeln('Zseni Alfonz');
End.
```

Futtassa le a programot (CTRL-F9), majd nézze meg az eredményét (ALT-F5)! A „Zseni Alfonz” szöveg most is a képernyőre került, de a kurzor a következő sor elején villog. A Writeln tehát abban különbözik a Write-től, hogy még egy soremelést is végrehajt.

2.1.2.9. Mentés másként

Amennyiben ezt a kis programocskát is el szeretnénk menteni, akkor két választásunk van. Vagy a már lemezen lévő programmal egyszerűen csak kicseréljük (File/Save), vagy új néven ezt is lementjük. Tegyük most ezt! Válasszuk a File/Save as... menüpontot, és a már ismert párbeszédpanelben adjuk meg újra a helyet, és az új nevet! Legyen ez most **nevkiiir2**! Ez tehát most már egy másik állományba került.

2.1.2.10. Fordítási hibák kezelése

A **Program Nevkiir2:** sor végéről töröljük le a pontosvesszőt, és próbáljuk meg lefordítani a programot (ALT-F9)! A legfelső sorban piros alapon kapjuk a hibaüzenetet: **Error 85: „;” expected**. Annyit jelent, hogy a „;” hiányzik. A kurzor ott villog, ahol a hiba van (ez jelen esetben nem egészen pontos, hiszen a következő sor elején, a **Begin** előtt villog), és semmi más dolgunk nincs, mint javítani a hibát. Pótoljuk vissza a pontosvesszőt!

A fordítási hibák a könyv végén, a **Mellékletben** található meg.

2.1.2.11. Új program készítése

Amikor egy új programot akarunk írni, a régit el kell tüntetnünk, mert már nincsen rá szükség. Ehhez a meglévő programot le kell zárni a Window/Close menüpont vagy az ALT-F3 billentyűk segítségével. A lezárásnál – ugyanúgy, mint a kilépésnél – amennyiben valami változtatás történt a programon az utolsó mentés óta, a Turbo Pascal rákérdez, hogy akarjuk-e menteni.

A lezárás után a File/New menüpont segítségével kérhetünk új, tiszta lapot, amelyre aztán dolgozhatunk.⁵

2.2. A képernyő kezelése: kiíratások

A programok a legtöbb esetben úgy működnek, hogy valamilyen információt várnak a felhasználótól (bemenő adat), és ebből aztán valamilyen kimenő adatot gyártanak, amit a felhasználóval közölni illik. Tehát a számítógép folyamatos kommunikációban áll a kezelőjével, és a kommunikáció színtere a monitor. Azaz kérdéseket és válaszokat kell a képernyőre kiírni, pontosabban kiíratni.

A számítógép-felhasználók általában igényesek. Ezért általában törekedünk arra, hogy a képernyőre írt kérdéseink és adataink esztétikusan nézzenek ki, tehát nem elég csak úgy kiíratni; tudnunk kell a képernyő egy adott pontjára írni, tudnunk kell színeket használni, stb.

2.2.1. A képernyő felépítése

Az előző fejezet példáiban már megtanultuk azt, hogy hogyan lehet a képernyőre íratni, de ahhoz, hogy ezek a kiíratások szépek legyenek, és bárhová tudjunk írni, ehhez ismernünk kell a képernyő felépítést, elsősorban a méreteit.

A képernyő alapállapotban 25 sort és 80 oszlopot tartalmaz (egy sor és egy oszlop által meghatározott téglalapba egyetlen karakter kerülhet). A sorok számozása 1-től 25-ig, az oszlopoké 1-től 80-ig terjed, a kiindulási pont a bal felső sarok.

A képernyőn mindig tudnunk kell, hogy éppen hol járunk, mindig van egy aktuális hely. Ezt jelzi számunkra a kurzor. A kurzor pozícióját meg tudjuk változtatni. Ez azért fontos, mert a képernyőre íratás mindig a kurzor helyén kezdődik. Alapállapotban a kurzor a képernyő bal felső sarkában van (1. sor, 1. oszlop).

A monitoron egy sor-oszlop koordinátákkal megadott helyen egy karakter lehet. Minden karakterhez hozzárendelhetünk színeket is, nevezetesen egy karakterszínt, ami a karakter vonalainak és pontjainak színt jelent; illetve egy háttérszínt, ami a karakter mögötti képernyőrész (az a bizonyos kis téglalap) színét jelent.

⁵ Amennyiben úgy kértünk új lapot, hogy nem zárjuk be a régit, akkor mindkettő a rendelkezésünkre áll egy-egy külön ablakban. Az egyszerre több program készítésének hasznát és a több ablak kezelését a unitokról szóló fejezetben mutatjuk be.

2.2.2. A CRT unit használata

A Turbo Pascalban a képernyőt kezelő eljárások és függvények a Crt unitban vannak. Ahhoz, hogy használni lehessen őket, először deklarálni kell a Crt unit használatát.

```
Uses Crt;
```

2.2.3. A képernyő letörlése

Az előző fejezet példáiban látható volt, hogy a szövegeink egy korábban már teleírt képernyőre kerültek. Ha azt szeretnénk, hogy a kiírásaink mindig a szép tiszta, üres képernyőre kerüljenek, akkor le kell törölnünk a képernyőt. A képernyőtörlés eljárása a ClrScr. (A ClrScr eljárás a Crt unitban van.)

3. példa Írassuk ki a nevünket a képernyőre, de előbb töröljük le azt!

```
Program NevKiir3;
Uses Crt;
Begin
  ClrScr;
  Write('Zseni Alfonz');
End.
```

*A programot lefuttatva, majd az eredményt megnézve látható, hogy most már a képernyő tiszta, a kiíratott szöveg pedig a képernyő bal felső sarkába került. Amennyiben a Crt unit használatát elfelejtjük a Uses kulcsszó segítségével deklarálni, akkor a fordítás az **Error 3: Unknow identifier** (ismeretlen azonosító) hibaüzenetet adja a ClrScr sorra, vagyis nem ismeri fel, hogy mi is ez az utasítás.*

A ClrScr eljárás tehát letörli a képernyőt, és a kurzort a bal felső sarokba teszi. Miután olyan program gyakorlatilag nem létezik, amelyikben nem írunk a képernyőre, ezért majd minden program a ClrScr eljárással kezdődik.

2.2.4. A kurzor helyének megadása

Az előbbieken láthattuk, hogy a Write és Writeln eljárások oda írják ki a megadott szöveget, ahol a kurzor van. Ha a képernyő egy tetszőleges helyére szeretnénk írni, akkor először a kurzort „oda kell tenni”. Erre való a GotoXY eljárás. Használata:

```
GotoXY(oszlop, sor)
```

Vagyis meg kell adni, hogy a képernyőn hányadik sorba és hányadik oszlopba kerüljön a kurzor. Vigyázat! Először az oszlop koordinátáját kell megadnunk (azaz először a vízszintes koordinátát, hasonlóan a

koordináta-rendszerhez). Az eljárás a ClrScr-hez hasonlóan a Crt unitban van.

4. példa Írassuk ki a nevünket a képernyő közepére!

Megoldás: A képernyő közepe nagyjából a 12. sor 40. oszlop. Ahhoz, hogy ide kerüljön a szöveg, alkalmazzuk a GotoXY eljárást. Arra is figyelünk, hogy a szöveg közepét szeretnénk a 40. oszlopba tenni, ezért egy kicsit előrébb kell elkezdeni a kiírást (kb. a 34. oszlopban).

```
Program NevKiir4;
Uses Crt;
Begin
  ClrScr;
  GotoXY(34,12);
  Write('Zseni Alfonz');
End.
```

A következő három példa három lehetséges hibára hívja fel a figyelmet.

5. példa Írassuk ki a „Próbaszöveg” feliratot az 5. sor 75. oszlopába!

```
Program Hibal;
Uses Crt;
Begin
  ClrScr;
  GotoXY(75,5);
  Write('Próbaszöveg');
End.
```

Azt tapasztaljuk, hogy a „Próbaszöveg” felirat elkezdődik ugyan az 5. sorban, de mivel nem fér ki, a következő sorban folytatódik.

6. példa Írassuk ki a „Próbaszöveg” feliratot az utolsó sorba a Writeln alkalmazásával!

```
Program Hiba2;
Uses Crt;
Begin
  ClrScr;
  GotoXY(1,25);
  Writeln('Próbaszöveg');
End.
```

Azt látjuk, hogy a felirat a 24. sorban van és a kurzor villog a 25. sor elején. A Writeln a kiíratás után sort emel, és miután már nem lenne helye, ezért a képernyőn minden feliratot egy sorral feljebb tol.

7. példa Írassuk ki a „Próbaszöveg” íelíratot a 32. sorba!

```
Program Hiba3;
Uses Crt;
Begin
  ClrScr;
  GotoXY(1,32);
  WriteLn('Próbaszöveg');
End.
```

32. sor nincsen, tehát a képernyő nem létező helyére akartuk helyezni a kurzort. Ekkor a kurzor alapállapotba kerül, vagyis a képernyő bal felső sarkába. Ugyanez lenne a helyzet akkor is, ha a kurzort nem létező oszlopba akarnánk helyezni.

Feladatok:

1. Írassa ki a 12. sor 30. oszlopába: „Hello!!!”!
2. Írassa ki a 8. sor 50. oszlopába: „Gyurika szép madárrrr!”!
3. Írassa ki az 1. sor jobb szélére: „Ez a jobb felső sarok”!
4. Írassa ki a 25 sor elejére: „Ez a bal alsó sarok”!
5. Írassa ki a képernyő jobb alsó sarkába: „Ez a jobb alsó sarok”!
6. Írassa ki a 20. sorba, a képernyő jobb szélére a nevé!
7. Írasson ki a képernyő 7. sorának nagyjából a közepére egy tetszőleges szöveget!

2.2.5. Több szöveg kiírása

Általában nem egyetlen dolgot szoktunk a képernyőre írni, hanem többet. Ezek a szövegek lehetnek egymás mellett, és egymás alatt is.

8. példa Írassuk ki a képernyőre a nevünket, majd mellé a telefonszámunkat!

Megoldás: A Write eljárással egyszerre több szöveget is kiíratunk, akkor ezeket vesszővel kell egymástól elválasztani.

```
Program NevKiir5;
Uses Crt;
Begin
  ClrScr;
  Write('Zseni Alfonz','123-45-67');
End.
```

A program eredménye, hogy a bal felső sarokban közvetlenül egymás mellett ott szerepel a két szöveg. Ha egy kis szünetet szeretnénk

köztük tartani, akkor szóközöket kell kiíratnunk (pl. a nevet megtoldhatjuk szóközökkel:

```
Write('Zseni Alfonz ','123-45-67');
```

Most már a két kiírás között háromkarakternyi hely szerepel.

9. példa Írassuk ki a nevünket és a telefonszámunkat egymás alá!

Megoldás: A WriteLn eljárást alkalmazzuk kétszer.

```
Program NevKiir6;
Uses Crt;
Begin
  ClrScr;
  WriteLn('Zseni Alfonz');
  WriteLn('123-45-67');
End.
```

A WriteLn eljárás alkalmazásával a kurzor a kiírás után a következő sor elejére ugrik, tehát a második szöveg ide fog kerülni. (Ezzel ellentétben a Write nem emel sort.) A telefonszám kiírása után már mindegy, hogy a kurzor hol lesz, ezért mindegy, hogy a Write vagy WriteLn eljárások közül melyiket alkalmazzuk.

10. példa Írassuk ki a nevünket és a telefonszámunkat egymás alá, de a képernyő közepére!

Megoldás: Most már mind a két kiírás előtt szükséges a kurzor pozícionálása a GotoXY eljárással.

```
Program NevKiir7;
Uses Crt;
Begin
  ClrScr;
  GotoXY(34,12);
  WriteLn('Zseni Alfonz');
  GotoXY(36,13);
  WriteLn('123-45-67');
End.
```

Feladatok:

8. Írassa ki a képernyő 3. sorának közepére, hogy „Ez a 3. sor közepe”, és a 20. sor jobb szélére, hogy „Ez meg a 20. sor széle”!
9. Írassa ki a képernyő öt különböző helyére öt különböző gyümölcs nevét!
10. Írassa ki a képernyő négy szélének közepére az égtájakat (a felső sor közepére azt, hogy „Észak”, a középső sor jobb szélére azt, hogy „Kelet”, stb.)!
11. Írassa ki a képernyőre a *Iános vitéz* első négy sorát, nagyjából középre!

„Tüzesen süt le a nyári nap sugára
Az ég tetejéről a juhászbojtárra.
Felesleges dolog sütnie oly nagyon
A juhásznak úgyis nagy melege nagyon.”

12. Készítsen el a képernyőre egy kis névjegyet (tartalmazza a név, foglalkozás, cím, telefon adatokat)!
13. Készítsen a képernyőre egy kis meghívót, amelyben a barátait hívja meg egy kirándulásra!

2.2.6. Számok kiírása, egyszerű számolások

11. példa Írassuk ki a képernyő közepére a 666-os számot!

Megoldás: Számokat kiírni ugyanúgy a Write és Writeln eljárásokkal lehet, csak a kiírni kívánt számot nem kell, sőt nem szabad aposztrófok közé tenni.

```
Program SzamKiir;
Uses Crt;
Begin
  ClrScr;
  GotoXY(39,12);
  Write(666);
End.
```

A számokat általában arra használjuk, hogy számoljunk is velük:

12. példa Írassuk ki a 6*8 művelet eredményét!

```
Program Szorzas;
Uses Crt;
Begin
  ClrScr;
  Writeln('6*8');
  Writeln(6*8);
End.
```

A programot lefuttatva a következőket látjuk: az első sorban a „6*8” szöveget, a második sorban a 6*8 számítás eredményét, a 48-at. Amennyiben egy karaktersorozatot aposztrófok közé teszünk, akkor azt a Turbo Pascal szövegnek veszi, akármilyen karakterekből álljon is. Ha nem tesszük idézőjelek közé, akkor a program felismeri, hogy számokról és egy műveleti jelekről van szó, és el tudja végezni a számítás, majd a kiíró eljárás hatására az eredmény fog megjelenni a monitoron.

13. példa Írassuk ki a 7/13 művelet eredményét!

Megoldás: A feladatot az előző alapján könnyen megoldhatjuk.

```
Program Osztas;
Uses Crt;
Begin
  ClrScr;
  Writeln('7/13=',7/13);
End.
```

A programot lefuttatva azt tapasztaljuk, hogy az eredmény valami nagyon furcsa érték: 5.3846153846E-01. Ez a szám normál alakját jelenti, vagyis kb. $5.38 \cdot 10^{-1}$, azaz kb. 0,538. A Turbo Pascal-ban az osztás eredménye mindig valós típusú szám (erről majd kicsit később bővebben lesz szó), és a valós típusú számokat mindig normál alakban jeleníti meg. Ha rendes tört alakban szeretnénk megjeleníteni, akkor ki kell használnunk a Write és Writeln eljárások egy további szolgáltatását.

14. példa Írassuk ki a 7/13 művelet eredményét három tizedesjegyre kerekítve!

Megoldás: A Write és Writeln eljárásokat a következőképpen is alkalmazhatjuk:

```
Write(szam :karakterhelyek száma :tizedes jegyek száma)
```

Ekkor a kiírandó szám annyi karakterhelyet foglalhat el, amennyit megadunk, és annyi tizedesre kerekítve jelenik meg, amennyit megadunk. Ha három tizedesre kerekítünk, akkor 6 karakternyi hely kell (előjel, egészrész, tizedespont, tizedes jegyek).

```
Program Osztas2;
Uses Crt;
Begin
  ClrScr;
  Writeln('7/13=',7/13:6:3);
End.
```

15. példa: A 24 és 8 számokkal végezzük el az alpműveleteket, és írassuk ki az eredményt!

Megoldás: A műveleti jelek: + (összeadás), - (kivonás), * (szorzás) és / (osztás).

```
Program Muvelet;
Uses Crt;
Begin
  ClrScr;
  Writeln('24+8=',24+8);
  Writeln('24-8=',24-8);
  Writeln('24*8=',24*8);
  Writeln('24/8=',24/8:1:0);
End.
```

Figyeljük meg a programot! Egy nagyon fontos iratlan szabályt alkalmazunk: nem írunk ki a képernyőre adatokat anélkül, hogy meg ne magyaráznánk azokat. A programokat legtöbb esetben nem a saját magunk számára írjuk, és bár mi tudjuk, hogy mi az a négy szám, ami a monitoron megjelenik, de más már nem fogja tudni. Ezért soha ne sajnáljuk az időt arra, hogy a programot ellássuk az efféle, a program működésére hatással nem lévő, de nagyon fontos kiírásokra! Ezzel a programjainkat „intelligenssé” tehetjük.

Feladatok:

14. Számítsa ki a $(6+3-5)*2$ kifejezés értékét!
15. Számítsa ki a $13+5/6$ és a $(13+5)/6$ kifejezések értékeit! Van különbség?
16. Írasson ki két számot (pl. 25 és 90) egymás mellé, majd végezze el az alapműveleteket velük, és az eredményt írja a képernyő 4-7-ik soraiba, a 20. oszlopban kezdve. Az eredményeket lássa el szöveges magyarázatokkal (pl. Összeg: 115)!

2.2.7. Színek használata

2.2.7.1. Színes szövegek

A képernyő sokkal látványosabbá és izgalmasabbá tehető, ha mondanivalónkat színek alkalmazásával közöljük.

A képernyő minden karakterhelyéhez tartozik egy karakterszín (amivel a karaktert színezzük ki) és egy háttérszín (ami a karaktert tartalmazó téglalap színe). Azt, hogy milyen színeket használunk a kiírás során, az aktuális karakterszín, illetve aktuális háttérszín adja meg. Az aktuális színek kezdetben: háttérszín: fekete, karakterszín: világoszürke. A színeket beállító eljárások:

TextColor(szín) a karakterek színét állíthatjuk vele
TextBackground(szín) a karakterek háttérszínét állíthatjuk vele

A lehetséges színeket a következő táblázat tartalmazza:

szín	angol név	kód
fekete	black	0
sötétkék	blue	1
zöld	green	2
cián	cyan	3
vörös	red	4
bíbor	magenta	5
barna	brown	6

szín	angol név	kód
világoszürke	gray	7
sötétszürke	darkgray	8
világoskék	lightblue	9
világoszöld	lightgreen	10
világoscián	lightcyan	11
világosvörös	lightred	12
világosbíbor	lightmagenta	13
sárga	yellow	14
fehér	white	15

A fenti eljárásokban a színeket akár a kódjukkal, akár az angol nevükkel⁶ megadhatjuk.

Egy **TextColor** vagy **TextBackground** eljárás használata után mindaddig a megadott színnel írunk a képernyőre, amíg csak valamelyiket meg nem változtatjuk!

A világos színek általában sötét megfelelőik intenzív változatai. Háttérszínnek csak a 0-7 kódú színek adhatók, ellenkező esetben a program a kódból 8-at kivon (így lesz pl. a sárgára definiált háttérszínből barna).

A szöveget villogtathatjuk, ha a karakterszín kódjához 128-at hozzáadunk (pl. villogó piros szöveget kapunk, ha a vörös kódjához (4) 128-at hozzáadva a **TextColor**(132) utasítást adjuk ki).

16. példa: Írassuk ki a nevünket a képernyő közepére világoszöld színnel!

Megoldás: A világoszöld szín kódja a 10, tehát ezt adjuk meg. Fontos, hogy a **TextColor** mindig előzze meg azt a kiíratást, amelyre vonatkoztatni akarjuk!

```
Program Szines;
Uses Crt;
Begin
  ClrScr;
  TextColor(10);
  GotoXY(34,12);
  Write('Zseni Alfonz');
End.
```

A **TextColor**(10) alak helyett használhattuk volna a **TextColor**(lightgreen) alakot is. A **GotoXY** és a **TextColor** eljárások egymáshoz képesti sorrendje nem befolyásolja a végeredményt, mivel egymásra nincs hatásuk; akár fordítva is szerepelhettek volna.

⁶ Ezek valójában konstansok, amelyek használatáról a későbbiekben még sok szó esik.

17. példa: Írassuk ki a nevünket és telefonszámunkat egymás alá, különböző karakter- és háttérszíneket használva!

Megoldás: Minden kiíratás előtt meg kell adnunk az új karakter- és háttérszínt.

```
Program Szines2;
Uses Crt;
Begin
  ClrScr;
  TextColor(lightgray);
  TextBackground(red);
  GotoXY(34,12);
  Write('Zseni Alfonz');
  TextColor(yellow);
  TextBackground(blue);
  GotoXY(36,13);
  Write('123-45-67');
End.
```

2.2.7.2. A teljes képernyő színének megváltoztatása

Sokszor szükségünk lehet arra is, hogy a képernyő teljes területének hátterét megváltoztassuk. Ehhez egy kicsit térjünk vissza a már megismert eljárásra, a ClrScr-re. A ClrScr feladata a képernyőtörlés. Igen ám, de hogyan? Úgy, hogy a képernyő minden egyes karakterpozícióját az aktuális háttérszínre színezi! Ez tehát alkalmas lehet arra, hogy a teljes képernyő hátterét megváltoztassuk, csak az kell hozzá, hogy a ClrScr előtt kell egy másik háttérszínt megadni a TextBackground eljárással.

18. példa: Színezzük a képernyőt zöldre, és erre fehérrel írjuk ki a nevünket!

```
Program Szines3;
Uses Crt;
Begin
  TextBackground(green);
  ClrScr;
  TextColor(white);
  GotoXY(34,12);
  Write('Zseni Alfonz');
End.
```

Feladatok:

- Írassa ki a képernyőre sárgával: „Hello!!!”
- Vörös képernyőre villogó fehér színnel írassa ki: „Ez nem olvasható!!!”

- Írjon ki a képernyő öt különböző helyére öt különböző színnel öt keresztnévet!
- A névjegyes feladatot (12.) alakítsa át színesre!
- A meghívós feladatot (13.) alakítsa át színesre!
- Készítsen egy színes plakátot a képernyőre, amely a kedvenc könyvét mutatja be!

Végül egy megjegyzés: a képernyő kezelésének még egyéb lehetőségei is vannak, amelyeket a Crt unit biztosít. Az egyes eljárások és függvények azonban a **Mellékletben** szerepelnek, és érdemes egy kicsit nézegetni, próbálgatni mindegyiket.

2.3. Változók

A programokban sok és sokféle adatot használunk, és ezeket valamilyen módon tárolni kell. A programokban az adattárolás eszközei a változók. Egy-egy változó egy-egy adatot tárolhat, és ez az adat a program futása során változhat (innen a név). Minden változó rendelkezik egy azonosítóval, vagyis egy tetszőleges névvel, amellyel hivatkozhatunk rá; illetve minden változónál meg kell adnunk, hogy milyen típusú adatot tárolunk benne, azaz a változónak típusa is van.

Ahhoz, hogy az adatainkat helyesen tudjuk kezelni, meg kell ismerkednünk az adattípusokkal.

2.3.1. Adatok, adattípusok

A adattípusok alapvetően kétfélék lehetnek: egyszerű típusok és összetett típusok (struktúrák). A továbbiakban az egyszerű típusokkal ismerkedünk meg, az összetett típusokra később kerül sor.

Az egyszerű típusok háromfélék⁷ lehetnek: numerikus típusok (azaz számok), szöveges típusok és logikai típusok. A numerikus típusoknak két további, egymástól eltérő típusa van: egészek és valósok.

A típusok ismerete azért nagyon fontos, mert ki kell tudnunk választani az adatainknak megfelelő legkedvezőbb típust, valamint műveleteket csak azonos típusokon végezhetünk el.

2.3.1.1. Egész típusok

A számítógép a számokat kettes számrendszerben tárolja. A kettes számrendszer két számjegyet ismer, a nullát és az egyest. Egyetlen kettes számrendszerbeli számjegy tárolására 1 bitet használunk. A biteket

⁷ A felsoroltaktól eltérő, egyéb típusok is vannak, de ezekkel egyelőre nem foglalkozunk.

nyolcasával csoportosítva bajtokat kapunk, és egy-egy számot valahány darab bajton tárolhatunk.

1 biten tehát összesen kétféle számot tárolhatunk: vagy 0 vagy 1 lehet ott. Két bit már négyféle különböző állapotot vehet fel: 00, 01, 10, 11. Hasonlóan három biten nyolc, négy biten tizenhat, 5 biten 32 különböző szám tárolható. Általánosan felírva: n darab biten 2^n különféle szám tárolható 0-tól (2^n-1)-ig. 1 bajton tehát, ami 8 bitből áll, 2^8 , azaz 256-féle számot ábrázolhatunk 0-tól 255-ig. 2 bajton, ami 16 bitből áll, 2^{16} , azaz 65536-féle számot ábrázolhatunk 0-tól 65535-ig. És így tovább.

A pozitív egész számokat egyszerűen kettes számrendszerbe átváltva tárolja a gépünk. A negatív számokat pedig az ún. kettes komplementum kód segítségével, amelynek a lényege az, hogy n bites negatív számok esetén azt a számot tároljuk, ami a számot 2^n -re egészíti ki (kettes komplementum). Felírni úgy lehetne, hogy a szám pozitív megfelelőjét kettes számrendszerben felírjuk, bitenként negáljuk, majd hozzáadunk 1-et. A legnagyobb helyértékű bit az ún. előjelbit, amelyen a szám előjelét tároljuk (negatív számok esetében ez 1, különben 0). Ez azt jelenti, hogy egy 1 bajton ábrázolt előjeles egész szám esetén egy bitet elhasználunk az előjel számára, tehát magára a száma csak 7 marad. Ebből az következik, hogy az előjeles számoknál az ábrázolható tartomány feleződik, pontosabban csak a legnagyobb ábrázolható szám lesz feleakkora, de ugyanekkora negatív számokat is ábrázolhatunk vele.

Ennyi bevezetés után nézzük a lényeget, vagyis a Turbo Pascal-ban használható egész típusokat:

Típus neve	Helyfoglalás	Értéktartomány
byte	1 bajt (8 bit)	$0 \Rightarrow 255$
shortint	1 bajt (8 bit) előjeles	$-128 \Rightarrow 127$
word	2 bajt (16 bit)	$0 \Rightarrow 65\,535$
integer	2 bajt (16 bit) előjeles	$-32\,768 \Rightarrow 32\,767$
longint	4 bajt (32 bit) előjeles	$-2\,147\,483\,648 \Rightarrow 2\,147\,483\,647$

2.3.1.2. Valós típusok

A törtszámok ábrázolása a fentebb ismertetett módszerekkel nem lehetséges. Hát akkor hogyan írjuk le őket? Első ötletünk az lehetne, hogy egyszerűen képzeljük oda a tizedesvesszőt valamelyik két bit közé. Például, ha 1 bajton ábrázolunk, helyezzük a tizedesvesszőt a 2. és 3. bit közé (ezzel 5 bitet hagyunk a szám egész részének, hiszen a biteket 0-tól számozzuk). A szám valódi értékét pedig úgy kapjuk meg, ha a tárolt számot (ami persze egész lehet csak, a tizedesvessző csak a

mi képzeletünkben él. a processzor nem tud róla) elosztjuk 2^3 -onnal, vagyis 8-cal. Általában, ha n db bitet használunk a törtrész tárolására, a tárolt értéket 2^n -nel kell osztani, hogy a tényleges értéket megkapjuk. Ez az ún. fixpontos ábrázolás. (Természetesen az egészeket is fixpontosan ábrázoljuk, csak ott a tizedesvessző a szám mögött van, azaz a törtrész 0 bit hosszúságú.) Az eljárás hátránya, hogy a törtrészre elkülönített bitek az ábrázolható értéktartomány rovasára mennek, és a szám pontossága sem kielégítő (ahhoz, hogy a tízes számrendszerben felírt szám n tizedes jegy pontos legyen, ahhoz legalább háromszor annyi bit kell a kettes számrendszerben).

A fenti hátrányok miatt a valós számok esetén inkább az ún. lebegőpontos ábrázolást alkalmazzák. A lebegőpontos ábrázolás lényege, hogy az N számot a következőképpen írjuk fel:

$$N = M \cdot 2^K$$

ahol az M az ún. mantissza, a K pedig a karakterisztika. Mindkét számot (tehát a mantisszát és a karakterisztikát) fixpontosan ábrázoljuk. A mantissza egy 0,5 és 1 közötti szám (a tizedesvesszőt a legnagyobb helyértékű bit után, vagyis a szám legelejére képzeljük oda). A karakterisztika előjeles egész szám⁸. Minden valós típus előjeles, a szám előjelét az előjelbit mutatja.

A valós számoknál is jellemző tulajdonság a helyfoglalás, illetve az ábrázolható értéktartomány, de nagyon fontos még a pontosság is: az, hogy hány darab tizedes jegy pontosságot várhatunk el tőlük. (A pontosság a mantissza hosszától függ, az ábrázolható értéktartomány pedig elsősorban a karakterisztika függvénye.)

A Turbo Pascal-ban használható valós típusok (minden valós típus előjeles):

Típus neve	Helyfoglalás*	Értéktartomány	Pontosság (tizedesjegy)
single	4 bajt (1+8+23=32 bit)	$\pm 1,5 \cdot 10^{-45} \Rightarrow 3,4 \cdot 10^{38}$	7-8
real	6 bajt (1+8+39=48 bit)	$\pm 2,9 \cdot 10^{-39} \Rightarrow 1,7 \cdot 10^{38}$	11-12
double	8 bajt (1+11+52=64 bit)	$\pm 5,0 \cdot 10^{-324} \Rightarrow 1,7 \cdot 10^{308}$	15-16
extended	10 bajt (1+15+64=80 bit)	$\pm 3,4 \cdot 10^{-4932} \Rightarrow 9,2 \cdot 10^{4932}$	19-20
comp	8 bajt (64 bit)	$-9,2 \cdot 10^{18} \Rightarrow 9,2 \cdot 10^{18}$	-

* A zárójelekben az előjelbit, a karakterisztika és a mantissza tárolásához szükséges bitek száma van megadva ebben a sorrendben

⁸ A karakterisztika nem kettes komplementum kóddal van ábrázolva, hanem ún. eltolt nullapontu ábrázolással. A lényege az, hogy pl. 1 bajtos karakterisztika esetén a számnál 128-cal nagyobb számot írunk le. Az értéktartomány megfelel az előjeles 1 bajtos számokénál leírtnak, azaz -128 és 127 között lehet.

A comp típus valójában az egészek közé tartozik, mind az ábrázolt értékeket, mind a fizikai tárolást figyelembe véve. Logikailag azonban, vagyis a használat módját tekintve inkább a valósok közé való.

A real típus kivételével az összes valós típus (single, double és extended), valamint a comp típus használatához koprocesszor szükséges. Ahhoz, hogy a processzor a koprocesszor számára lebegőpontos kódot generáljon, az kell, hogy a program elején, a deklarációs részben az N fordítási direktívát bekapcsoljuk. Használatának módja: { \$N+ }. Amennyiben nem rendelkezünk koprocesszorral, akkor a működését szoftveresen emulálni kell. Erre szolgál az E fordítási direktíva: { \$E+ }. Ezeknek a típusoknak a használatára azonban csak akkor van szükség, ha nagyon nagy, vagy nagyon pontos számokra van szükségünk.

2.3.1.3. Szöveges típusok

A szöveges típusok egy karaktersorozat tárolására használhatók. A szöveges típusok a következők lehetnek:

Típus	Leírás
char	1 darab karakter
string	maximum 255 karakterből álló karaktersorozat
string[n]	maximum n darab karakterből álló karaktersorozat

Minden szöveges típus memóriabeli helyfoglalása karakterenként egy byte.

2.3.1.4. Logikai típus

A logikai típus mindössze két értéket vehet fel: IGAZ vagy HAMIS (TRUE or FALSE). Bár elvileg tárolásához egy bit is elég lenne, mégis egy bájt (8 biten) ábrázoljuk. Neve: boolean.

2.3.1.5. Sorszámozott típusok

A fentebb ismertetett típusok közül az egészek (bájt, shortint, integer, word és longint), a char és a boolean típus az úgynevezett sorszámozott típusok közé tartozik. Olyan típusok ezek, amelyek véges számú elemmel rendelkeznek, és minden elemről tudható, hogy ő hányadik az adott típusban, vagyis ismerjük a sorszámt.

2.3.1.6. Típusok kompatibilitása

Azonos kifejezésekben vagy egy értékadásban szereplő adatok típusainak kompatibilisnek kell lennie egymással. Két típus (a felsoroltak közül) kompatibilis, ha mindkettő egész, vagy mindkettő valós típusú, illetve mindkettő valamilyen szöveges típusú.

2.3.2. Műveletek, relációk

Az egyes adattípusokra értelmezhetők műveletek és relációk. A műveleti jeleket operátoroknak, a műveletben résztvevő kifejezéseket operandusnak nevezzük.

2.3.2.1. Numerikus típusokon értelmezett műveletek

Művelet (operátor)	Leírás	Operandusok típusa	Eredmény típusa
+	összeadás	egész valós	egész valós
-	kivonás	egész valós	egész valós
*	szorzás	egész valós	egész valós
/	osztás	egész valós	valós valós
div	egészosztás	egész	egész
mod	maradékképzés	egész	egész

2.3.2.2. Szöveges típusokon értelmezett műveletek

Művelet (operátor)	Leírás	Operandusok típusa	Eredmény típusa
+	összefűzés (konkatenáció), azaz a szövegek egymás után írása	char vagy string	string

2.3.2.3. Logikai műveletek

Művelet (operátor)	Leírás	Operandusok típusa	Eredmény típusa
NOT	logikai tagadás bitenkénti negáció	logikai egész	logikai egész
AND	logikai ÉS bitenkénti ÉS	logikai egész	logikai egész
OR	logikai VAGY bitenkénti VAGY	logikai egész	logikai egész
XOR	logikai kizáró VAGY bitenkénti kizáró VAGY	logikai egész	logikai egész

A logikai műveletek igazságtáblázatai:

A	B	NOT A	NOT B	A AND B	A OR B	A XOR B
IGAZ	IGAZ	HAMIS	HAMIS	IGAZ	IGAZ	HAMIS
IGAZ	HAMIS	HAMIS	IGAZ	HAMIS	IGAZ	IGAZ
HAMIS	IGAZ	IGAZ	HAMIS	HAMIS	IGAZ	IGAZ
HAMIS	HAMIS	IGAZ	IGAZ	HAMIS	HAMIS	HAMIS

Ugyanezek az igazságtáblázatok nem logikai értékekkel, hanem bitekkel:

A	B	NOT A	NOT B	A AND B	A OR B	A XOR B
1	1	0	0	1	1	0
1	0	0	1	0	1	1
0	1	1	0	0	1	1
0	0	1	1	0	0	0

2.3.2.4. Relációk

A relációk bármilyen numerikus, szöveges és logikai értékeken értelmezve vannak, közös jellemzőjük, hogy az eredményük minden esetben logikai típusú (IGAZ vagy HAMIS).

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

2.3.2.5. Függvények

A függvények közös jellemzője, hogy a kifejezésekbe az általa szolgáltatott érték kerül be, ezért aztán nem is állhatnak önállóan a programban, csak egy értékadás jobb oldalán, vagy egy kiíratásban. A paraméter megadásánál figyeljünk arra, hogy csak a megadott típussal kompatibilis típus használható; illetve az a változó, amely értéként a függvény visszatérési értékét kapja, kompatibilis típusú legyen a függvény eredményével.

Néhány ismertebb matematikai függvény a Turbo Pascal-ban (az alábbi megadott függvények mindegyike a System unit része):

Függvény neve	Leírása	Paraméter típusa	Eredmény típusa
Abs (x)	abszolút érték	numerikus	numerikus*
Sqr (x)	négyzetre emelés	numerikus	numerikus*
Sqrt (x)	gyökvonás	real	real
Int (x)	egészrész	real	real
Trunc (x)	egészrész	real	longint
Round (x)	egészre kerekítés	real	longint
Frac (x)	tötrész	real	real
Pi	π konstans, értéke: 3,14159265...	-	real
Sin (x)	szinusz	real**	real
Cos (x)	koszinusz	real**	real
ArcTan (x)	arkusz tangens	real	real**
Exp (x)	exponenciális függvény, az e kitevője	real	real
Ln (x)	természetes alapú logaritmus	real	real

* A paraméterrel megegyező típusú

** Radiánban

2.3.2.6. Kifejezések

A kifejezések számokból, változókból, függvényekből és műveleti- illetve zárójelekből állnak. Például:

$(a+b)*2, r*r*pi, a*b*cos(30)/2$, de kifejezés egy önálló szám is, pl. 26.

2.3.2.7. Precedencia-szabály

A kifejezésekben a műveletek végrehajtási sorrendje a következő alapelveken alapul:

1. **Prioritás:** a magasabb rendű (nagyobb prioritású) műveleteket előbb végezzük el, mint az alacsonyabb rendű műveleteket.
2. **Balról jobbra szabály:** azonos prioritású műveletek közül először a bal oldalit végezzük el, majd jobbra haladva a többit.
3. **Zárójelek:** zárójelek segítségével felülbírálhatjuk a végrehajtási sorrendet: először a zárójelben lévő művelet végzendő el.

A Turbo Pascal-ban a precedencia szerinti végrehajtási sorrendet a következő táblázat mutatja (precedencia szerinti csökkenő sorrendben):

Precedencia	Kategória	Operátorok
1. (legmagasabb)	Zárójelek	
2.	Függvények	
3.	egy operandusú műveletek	+ - (előjel), NOT
4.	multiplikatív operátorok	*, /, div, mod, AND
5.	additív operátorok	+, -, OR, XOR
6.	relációk	=, <>, <, >, <=, >=
7 (legalacsonyabb)	Balról jobbra szabály	

2.3.3. Változók deklarálása

Egy változót a használatba vétel előtt deklarálni kell a deklarációs részben. A változókat a **Var** kulcsszó után, azonosítójuk és típusuk megadásával deklarálhatunk. Formálisan:

```
Var
    azonosító : típus;
    vagy
    azonosító1, azonosító2, ... : típus;
```

2.3.4. Értékkadás

A program során a változóink értékét változtatnunk kell. Egy változó az ún. értékkadás művelettel kaphat értéket. A műveleti jel a „:=” (legyen egyenlő). Használata:

változó:=kifejezés

Ha egy változót úgy akarunk használni, hogy még nem kapott értéket, akkor a Turbo Pascal automatikusan hozzárendel egy kezdőértéket. Ez a kezdőérték a következő lehet:

Típus	Kezdőérték
numerikus típusok	0
szöveges típusok	'' (ez az ún. üres szöveg, vagy üres string, két aposztróf egymás mellett)
logikai típus	FALSE (HAMIS)

2.3.5. A változók egyszerűbb alkalmazásai

Végre eljutottunk odáig, hogy programokban is alkalmazhatjuk a változókat.

2.3.5.1. Értékkadás és a változó értékének kiírása

19. példa Adjunk értéket néhány változónak, és írassuk ki ezeket az értékeket!

Megoldás:

```
Program Valtozo;
Uses Crt;
Var
    a,b,c : integer;
    d : real;
    e,f,g : char;
    n,i,j : string;
    k : string[4];
Begin
    a:=12; b:=(2*3)-4; c:=a+b;
    d:=33/4;
    e:='K'; f:='4'; g:='@';
    h:='Hello '; i:='Alfonz';
    j:=n+i;
    k:='ABCDEFGHIJKLMNPOQRSTUVWXYZ';
    ClrScr;
    Writeln(a,b);
    Writeln(c);
    Writeln(d);
    Writeln(e,f,g);
    Writeln(h);
    Writeln(i);
    Writeln(j);
    Writeln(k);
End.
```

A programot tanulmányozva látható, hogy szükség esetén több utasítás is írható egy sorba, ha ezeket pontosvesszővel választjuk el. Mindazonáltal nem ajánlom az alkalmazását, mert sokkal célszerűbb (könnyebben áttekinthető, és könnyebb a hibakeresés is) az

„egy utasítást egy sorba írunk” elv. A program eredménye a következő lett:

122 (az *a* és *b* változók egymás mellé írt értékei: 12 és 2)

14 (a *c* változó értéke)

8.2500000000E+00 (a *d* változó értéke, mivel valós típus, a kijelzés normálalakos (=8,25))

K4@ (az *e*, *f* és *g* változók egymás mellé írt értékei)

Hello (a *h* változó értéke)

Alfonz (az *i* változó értéke)

Hello Alfonz (a *j* változó értéke)

ABCD (a *k* változó értéke. Hiába adtunk sokkal hosszabb értéket, mivel a típus csak 4 karaktert tesz lehetővé, így a változóban a megadott érték első négy karaktere van csak)

2.3.5.2. A változók értékének megváltoztatása

20. példa Adjunk értéket egy numerikus és egy szöveges változónak, majd változtassuk meg az értéküket!

Megoldás:

```
Program Valtozo2;
Uses Crt;
Var
  a : integer;
  s : string;
Begin
  Clrscr;
  a:=25;
  s:='Hello';
  Writein(a);
  Writein(s);
  a:=(100-45)*2;
  s:='Sz'+'ia';
  Writein(a);
  Writein(s);
  a:=a+10;
  s:=s+'!';
  Writein(a);
  Writein(s);
End.
```

Látható, hogy a változó értékadásnál kaphatnak egy konkrét értéket (25 ill. „Hello”), kaphatják értékül egy kifejezés eredményét ((100-45)*2 ill. „Sz”+„ia”), valamint megváltoztathatják a saját értéküket. A kissé furcsa *a:=a+10* sor azt jelenti, hogy az *a* új értéke legyen egyenlő az *a* régi értéke+10. Ugyanígy az *s* változó értéke felveszi a régi érték és a „!” jel összefűzésének eredményét.

2.3.5.3. Kifejezések kiszámítása

21. példa Számítsuk ki a következő kifejezés értékét:

$$\frac{a+b}{b-1}$$

ha *a*=15 és *b*=9! Az eredményt a *c* változóba tegyük!

Megoldás: A fenti kifejezést a következőképpen adhatjuk meg a Turbo Pascalban: *(a+b) / (b-1)*. A zárójelek alkalmazására azért van szükség, mert az osztás prioritása nagyobb az összeadás és kivonás prioritásánál, tehát a zárójelek nélkül írt kifejezésben először az osztást számolná ki a gép. A *c* változó értékét valósra kell deklarálni, mert egy osztás eredménye mindig valós.

```
Program Kifejezes;
Uses Crt;
Var
  a,b : integer;
  c : real;
Begin
  a:=15;
  b:=9;
  c:=(a+b) / (b-1);
  ClrScr;
  Writein('A kifejezés eredménye: ',c:5:2);
End.
```

A kiírás során a *c* változó értékét formázottan írjuk ki, hiszen e nélkül a normálalakot kapnánk meg.

2.3.5.4. Feladatok változókra

22. példa: Változók használatával számítsuk ki, hogy 1200 forint hány dollárnak felel meg, és az eredményt jelenítsük meg a képernyőn! (1\$ a könyv írásakor kb. 250 Ft-ot ér.)

Megoldás: Egy-egy változót használunk a forint, illetve a dollár értékének tárolásához. Legyenek ezek **forint** és **dollar**. (Célszerű mindig ún. „beszédes” változónevet használni, ami megmondja, hogy milyen jellegű értéket tárolunk benne.)

```
Program PenzValtas;
Uses Crt;
Var
  forint : integer;
  dollar : real;
```

```

Begin
forint:=1200;
dollar:=forint/250;
ClrScr;
WriteLn(forint,'Ft ',dollar:6:2,'$-nak felel meg. ');
End.

```

Az utolsó sorban az eredmény közlésénél alkalmazkodtunk ahhoz a szabályhoz, hogy az eredményünket magyarázzuk. A WriteLn eljárásban négy értéket íratunk ki egymás mellé: először a **forint** változó értékét, majd a „Ft” szöveget, utána a **dollar** változó értékét (tulajdonképpen a számítás eredményét), végül a magyarázó szövegét: „\$-nak felel meg”.

Feladatok: (a feladatokban az összes adatot és az eredményeket is változóban tárolja!)

23. Számítsa ki, hogy 15\$ hány forintnak felel meg!
24. Számítsa ki, hogy 120MB hány kbyte-nak felel meg! Adja meg az eredményt bájt-okban is!
25. Számítsa ki, hogy hány karakter tárolható egy 1,6GB-os winchesteren!
26. Adja meg egy 182 cm magas ember magasságát mm-ben!
27. Adja meg egy 12000 m²-es földdarab területét hektárban (1, ha=1000 m²)!
28. Módosítsa a 22. példaprogramot úgy, hogy az árfolyamot is változóban tárolja!
29. Módosítsa a 23. feladatot úgy, hogy az árfolyamot is változóban tárolja!
30. Számítsa ki egy 6 cm oldalhosszúságú négyzet kerületét és területét!
31. Számítsa ki egy 34 mm sugarú kör kerületét és területét! Az eredményt cm²-ben adja meg!
32. Egy téglalap oldalai 12 és 8 cm. Adja meg a kerületét és a területét!
33. Számítsa ki, hogy egy 6 láb 4 hüvelyk magas ember hány centiméter (láb=30,48 cm; hüvelyk: 2,54 cm)!
34. Számítsa ki, hogy délután 17 óra 20 perckor hány perc telt el a napból! Adja meg másodpercben is!
35. Számítsa ki, hogy május 12-e az évnek hányadik napja (az egyszerűség kedvéért legyen minden hónap 30 napos)!
36. Számítsa ki, hogy 120 000 Ft mennyit kamatozik egy év alatt 15%-os kamatláb mellett!
37. Számítsa ki, hogy mennyibe kerül egy utazás autóval Hatvanba (60 km), ha a benzín literje 150 Ft és a kocsi fogyasztása 7l./100km.!

38. Az a változó értéke legyen 150, a b változó értéke pedig 43. Cserélje meg a két változó értékét! (Az a:=43; b:=150; megoldás nem megengedett, a program működjön bármilyen számok megadása esetén!)⁹

2.3.6. Értékadás a program futása közben (adatbekérés)

Az előző fejezet feladatai jók, de nem általánosak. Nem olyan programot írtunk, ami dollárt vált át forintra, hanem egy olyat, ami csak a 15\$-t tudja átszámolni. Az ilyen programot állandóan meg kell változtatni ahhoz, hogy más számokkal is kipróbálhassuk. Képzeljük el, ha a számológépbe állandóan bele kellene nyúlnunk egy csavarhúzóval, hogy más műveletet is elvégezzon, mint az 5+6-ot. Az ilyen program nem sokat ér.

A helyes működés az, ha a programcskánk bármilyen értékkel tud, amit a felhasználó megad neki. Ennek pedig az a legegyszerűbb módja, hogy a program kérdezzen. Jelen példánkban például megkérdezné: „hány dollárt akarsz váltani?” – és a kérdésre adott válasz alapján számítaná és írná ki az eredményt. Ez annyit jelent, hogy a program megírásakor nem ismerjük meg a változó értékét, csak futás közben derül ki, tehát nem is adhatjuk meg előre az értékadó utasítással. Rá kell venni a programot, hogy kérdezzen, várja meg a választ, és a kapott választ tárolja el egy változóban. Erre valók a Read és ReadLn eljárások. Használatuk:

Read(változónev) vagy

ReadLn(változónev)

ahol a változónev annak a változónak az azonosítója, melyikben a bekért értéket eltároljuk. A két eljárás között az a különbség, hogy a ReadLn a bekérés után sort emel (hasonlóan a Write, WriteLn eljárásokhoz). Az utasítás használatát a következő példa mutatja meg.

23. példa: Írjunk programot, amely dollárt vált át forintra!

Megoldás: Rávesszük a programot arra, hogy kérdezzen a ReadLn eljárás segítségével.

⁹ Célszerű közbenső változó bevezetése, így a megoldáskor először a közbenső változó veszi fel az első változó értékét, majd az első változó a másodikét, végül a második a közbensőét. Ezzel a módszerrel nem veszik el menet közben adatait.

```

Program PenzValtasBekeressel;
Uses Crt;
Var
    forint : integer;
    dollar : real;
Begin
    ClrScr;
    Readln(forint);
    dollar:=forint/220;
    WriteLn(forint,'Ft ',dollar:6:2,'$-nak felel
meg.');
```

Ha lefuttatjuk a programot, akkor azt látjuk, hogy a letörölt képernyő bal felső sarkában villog a kurzor. Most ért a program az Readln-hez, tehát most várakozik a válaszra. Adjuk meg az 1200-at, és a gépelést zárjuk le az ENTER-rel! Most megint megkapjuk az előző eredményt! Most futtassuk újra a programot, de most 5800-at adjunk meg! Erre a számítás eredménye is más lett. Próbáljuk ki, hogy mi történik akkor, ha nem számot válaszolsz a kérdésre! Futtassuk újra a programot, és adjuk azt a választ, hogy „ezer” (szöveggel)! A programunk úgynevezett futási hibával leáll, visszakapjuk a szerkesztő képernyőt, és piros alapon láthatjuk a hibaüzenetet: „helytelen számforma”. A kurzor pedig jelzi azt a programsort, amelyikben a hiba történt (ez jelen esetben a Readln). Az történt, hogy a **forint** változónk integer típusú, mi pedig szöveggel válaszoltunk. Most futtassuk újra a programot, és adjunk meg 60000-t! Az eredmény nagyon furcsa: azt kaptuk meg, hogy „-5536 Ft -25.16\$-t ér”. Hogy lehet ez? Úgy, hogy a **forint** változónk integer típusú. Az integer típus értékhátára -32768-tól 32767-ig terjed. Mi ennél nagyobb értéket adtunk meg, ezért a furcsa eredmény. A jelenséget túlcsoportulásnak nevezzük (vagyis azt, hogy egy változó értéke kicsúszik a típus által meghatározott értelmezési tartományból). Hogy ha egy változó egy kifejezés értékét kapja értékül, akkor nem elég a kifejezés végeredményének beleesnie a megadott értékhátárok közé, hanem minden részsámítás eredményének is bele kell esni! Megoldás lehet más típusok alkalmazása: deklaráljuk a **forint** változót is **real**, vagy **longint** típusúra.

Elérhetjük tehát, hogy kommunikáljunk a programunkkal. Igen ám, de micsoda beszélgetés ez? A programunk nem tesz fel kérdést, csak várja a választ. Az, hogy a programunk feltesz egy kérdést azt jelenti, hogy kiír a képernyőre egy kérdést, ezt pedig már jól ismerjük: Write vagy WriteLn.

24. példa: Írjuk át a fenti programot úgy, hogy kérdezzen! Javítsuk ki a programunk hibáját is!

Megoldás: Egy új sort kell beírni a Readln elé: a Write-ot. azért a Write-ot, mert a választ a kérdés után közvetlenül célszerű bekérni, nem a következő sorban.

```

Program PenzValtasBekeressel2;
Uses Crt;
Var
    forint, dollar : real;
Begin
    ClrScr;
    Write('Hány forintod van? Megmondom mennyi dol-
lart ér! ');
    Readln(forint);
    dollar:=forint/220;
    WriteLn(forint:10:0,'Ft ',dollar:6:2,'$-nak felel
meg.');
```

A kérdés feltevésénél nagyon ügyeljünk arra, hogy az egyszerű, könnyen érthető és egyértelmű legyen. Pontosán magyarázza meg, hogy mit is akarunk a felhasználótól, de ne legyen terjengős. Nem szabad sajnálni a fáradságot egy-egy jó kiírás megfogalmazására, mert bár ez nem befolyásolja a program működését, de nagyon fontos. Ha a program saját magunknak készül, és tudjuk, hogy mi a kérdés, akkor is ki kell írni, hiszen két hónap múlva nem biztos, hogy még mindig emlékszünk rá. Ha másnak készítjük, akkor felesleges is, hogy kihangsúlyozzuk a fontosságát, ez magától értetődik. A Write eljárásban a kiíratandó szöveg végén néhány szóköz van. Ennek az a szerepe, hogy elválassza a kiíratott szöveget a begépe-
lendő értéktől.

25. példa Készítsünk programot, amelyben bekérünk két számot és kiírjuk az összegüket!

Megoldás: Most nem egy, hanem két számot kell bekérnünk, ehhez pedig két különböző változóra van szükség két különböző azonosítóval.

```

Program Osszead;
Uses Crt;
Var
    a,b,osszeg : integer;
Begin
    ClrScr;
    Write('Kérek két számot, összeadom őket! ');
    Readln(a,b);
    osszeg:=a+b;
    WriteLn('A két szám összege: ',osszeg);
End.
```


A program megoldható az **összeg** változó használata nélkül is, ekkor a kiíratásnál közvetlenül az **a+b** értéket kell kiírni: `WriteLn('A két szám összege: ',a+b);`. A `ReadLn(a,b)` sorban egyszerre két változó értékét kérjük be. Ekkor szóközzel, vagy ENTER-rel elválasztva kell a két számot beírni a futtatás után. Általában a módszert nem javaslom, helyette inkább a több gépeléssel járó, de biztosabb és szebb eredményt garantáló külön-külön adatbekérést ajánlom:

Megoldás2:

```
Program Osszead2;
Uses Crt;
Var
    a,b,osszeg : integer;
Begin
    ClrScr;
    WriteLn('Adj meg két számot, összeadom őket!');
    Write('Kérem az első számot: ');
    ReadLn(a);
    Write('Kérem a második számot: ');
    ReadLn(b);
    osszeg:=a+b;
    WriteLn('A két szám összege: ',osszeg);
End.
```

Feladatok (a feladatokat adatbekéréssel oldja meg!):

39. Készítsen programot, amely dollárt vált át forintra! Az árfolyam értékét is kérje be!
40. Készítsen programot, amely egy winchester Mbyte-ban megadott tárolókapacitását írja ki bájt-okban!
41. Készítsen programot, amely egy ember cm-ben megadott magasságát mm-ben adja meg!
42. Készítsen programot, amely egy földdarab m^2 -ben megadott területét átszámítja hektárra!
43. Készítsen programot, amely kiszámítja egy négyzet területét és területét!
44. Készítsen programot, amely kiszámítja egy kör területét és területét! Az eredményt cm^2 -ben adja meg!
45. Készítsen programot, amely kiszámítja egy téglalap területét és területét!
46. Készítsen programot, amely egy ember lábban és hüvelykben megadott magasságát centiméterre számítja át (láb=30,48 cm; hüvelyk: 2,54 cm)!

47. Készítsen programot, amely a pontos időből (óra, perc) kiszámítja, hogy ez hányadik perce a napnak!
48. Készítsen programot, amely a dátumból (hónap, nap) kiszámítja, hogy ez hányadik napja az évnek (az egyszerűség kedvéért legyen minden hónap 30 napos)!
49. Módosítsa az előző feladatot úgy, hogy az órát is megkérdezze, és azt adja eredményül, hogy ez az év hányadik órája!
50. Készítsen programot, amely a tőke összegéből és a kamat mértékéből kiszámítja, hogy mennyi pénzünk is lesz egy év múltán!
51. Készítsen programot, amely benzinköltséget számol a megtett út, a kocsi fogyasztása és a benzín ára ismeretében!
52. Készítsen programot, amely két változóban tárolt bekért értéket felcseréli!
53. Kérjen be egy szöveget, például egy nevet, és írassa ki a tiszta képernyő közepére!
54. Kérjen be egy szöveget (pl. egy nevet) és egy szint! A bekért szöveget a bekért színnel jelenítse meg a képernyő közepén!
55. Kérjen be egy szöveget, és azt, hogy a szöveg a képernyő hányadik sorának hányadik oszlopába kerüljön! A megadott helyre írassa ki a bekért szöveget!
56. Kérjen be két számot, majd írassa ki az összegüket, különbségüket, szorzatukat és hányadosukat!
57. Módosítsa a névjegy-készítő programot úgy, hogy ne a saját névjegyet készítse el a képernyőn, hanem azét, aki a gép elé ül! (Kérdezze meg a nevét, címét, stb., és ezeket a szövegeket írassa ki a képernyőre. A névjegy legyen esztétikus, alkalmazzon színeket is!)

2.3.6.1. Az adatbekérés egy speciális alkalmazása: várakozás

Látható volt az előző példákban, hogy a `ReadLn` eljárásához érve a program várakozik, egészen addig, amíg a választ be nem gépeljük és le nem nyomjuk az ENTER-t. A `ReadLn`-t ez a tulajdonsága alkalmassá teszi arra, hogy a programunkat egy adott helyen addig várakoztassuk, amíg le nem nyomjuk az ENTER-t.

Valószínűleg már elég idegesítő lehetett az, hogy a program lefutása után minden esetben vissza kellett nézni az ALT-F5-tel a program eredményét. Használjuk ki tehát a `ReadLn` fentebb leírt tulajdonságát, és várakoztassuk a programot addig, amíg már kellően kigyönyörködtük magunkat a munkánk eredményében, és lenyomunk egy ENTER-t.

26. példa Írjuk ki a nevünket a képernyő közepére, majd várakoztassuk a programot egy ENTER lenyomásáig!

Megoldás: Adjuk meg a Readln-t a program legutolsó sorában!

```
Program Varakozas;
Uses Crt;
Begin
  ClrScr;
  GotoXY(34,12);
  WriteLn('Zseni Alfonz');
  GotoXY(1,25);
  Write('Nyomj egy ENTER-t, ha befejezted');
  Readln;
End.
```

A programban a név kiírása után a kurzort a legalsó sor elejére helyezzük, kiírunk egy magyarázó szöveget, hogy mit várunk el a felhasználótól, és végül várakozunk egy ENTER-re.

Ivaslom tehát, hogy a program utolsó sora egy paraméter nélkül kiadott Readln eljárás legyen. Házi használatra még magyarázó szöveg nélkül is alkalmazható, „rendes” programban azonban ezt nem szabad lespórolni.

2.3.7. Állandók (konstansok) használata

Az állandók nagyon hasonlítanak a változókra, csak éppen a program futása alatt nem változtathatják meg az értéküket. Ugyanúgy deklarálni kell őket, és ugyanúgy részt vehetnek kifejezésekben, mint a változók. Nem állhatnak azonban egy értékadás bal oldalán, vagyis nem kaphatnak értéket!

Minden olyan adatot, amelyet nem akarunk változtatni a program futása közben, célszerű állandókban tárolni. Olyan számok, színek, stb. tárolására is alkalmazhatók, amelyeket sokszor használunk fel a programunkban. Azért célszerű a használata ez utóbbi esetben, mert ha ezeken az értékeken módosítani kell, akkor sok helyen kell átírni ugyanazt az értéket az új értékre, míg állandó alkalmazásával csak egy helyen, a program elején.

Az állandók alapvetően kétféleképpen lehetnek: számok vagy szövegek. A számok lehetnek egészek vagy valósak. Az egész konstansok longint, a valósak real, a szövegek string típusúak, pontosabban ezekkel kompatibilisek. A számkonstansok értelmezési tartománya is ezen típusokéval egyezik meg.

A konstansokat a **Const** kulcsszó után adhatjuk meg a deklarációs részben. Használata:

```
Const
  konstans-azonosító = érték;
```

Néhány példa konstansok deklarálására:

```
Const
  a = 2234;
  p = 3.1415;
  s = 'kiskutya';
```

27. példa Írjuk át forintot dollárra átváltó programot úgy, hogy az árfolyamot állandóban tároljuk!

Megoldás: A deklarációs részben meg kell adni a használni kívánt konstans (legyen ez **arfolyam**), majd a programban tetszőleges helyen felhasználhatjuk, ugyanúgy, ahogy egy változót.

```
Program Allando;
Uses Crt;
Const
  arfolyam = 250;
Var
  forint, dollar : real;
Begin
  ClrScr;
  WriteLn('A dollár árfolyama jelenleg:');
  l$=' ', arfolyam, 'Ft');
  Write('Hány forintod van? Megmondom mennyi dollárt ér! ');
  Readln(forint);
  dollar:=forint/arfolyam;
  WriteLn(forint:10:0, 'Ft ', dollar:6:2, '$-nak felel meg.');
```

A programban kétszer használtuk fel az **arfolyam** konstans értékét: először egy kiírásban, másodszor pedig egy kifejezésben. Változtassuk meg a konstans értékét mondjuk 300-ra, és futtassuk újra a programot! Mindkét helyen a 300 került felhasználásra. Ha nem alkalmaztunk volna konstans, akkor egy ilyen átírás során két helyen kellett volna módosítanunk a fenti programot.

A deklarációs részben a **Const** és **Var** kulcsszavak sorrendje tetszőleges. Lehetőségünk van mindkettőből többet is szerepeltetni, szintén tetszőleges sorrendben.

2.3.7.1. Beépített konstansok

Az egyes unitokban eljárásokon és függvényeken kívül lehetnek konstansok is (és persze még sok egyéb más is, de ezekről majd a maga idejében). Így például a már megismert színkezelésnél az egyes színek helyett használható angol elnevezések valójában a Crt unit beépített konstansai. Ezek használatához természetesen ugyanúgy kell a unitot deklarálni a `Uses` kulcsszóval, mint eljárások használata esetén.

2.4. Számlálós ciklusok

2.4.1. A ciklus

Egy-egy tevékenység leírásakor gyakran kerülünk olyan helyzetbe, hogy bizonyos utasításokat vagy utasítássorozatokat többször kellene végrehajtani egymás után. Ilyenkor az egyik megoldás lehet az, ha sokszor egymás után leírjuk az ismétlendő részt, de ez unalmas is, felesleges is, és ha a végrehajtások száma elég sok, akkor bizony elég sokáig is tart. Egyszerűbb megoldás az, ha azt mondom: itt van ez az utasítás-sor, ezt hajtssd végre 10-szer, vagy 3-szor, vagy ahányszor kell. Ezt a fajta szerkezetet ciklusnak nevezzük, azt a tevékenységsort pedig, amelyet ismételni kell, ciklusmagnak.

2.4.2. A számlálós ciklus

A feladatok egy részénél tudjuk előre, hogy a ciklusunknak hányszor kell végrehajtani azt a bizonyos utasítássorozatot, más részüknél azonban nem – például ha a ciklusnak egy feltétel teljesülése esetén kell leállnia. Az előbbi ciklust számlálós ciklusnak, az utóbbit tesztelős ciklusnak nevezzük. Ebben a fejezetben a számlálós ciklusokkal foglalkozunk, a tesztelős ciklusokat egy kicsit későbbre hagyjuk.

A számlálós ciklusok szerkezete a következő lenne a leírtak alapján:

```
Ismételd n-szer:
    Ciklusmag utasításai
Ciklus vége
```

Ennek a megoldásnak az a hibája, hogy a végrehajtás során nem tudjuk, hogy éppen hányadiknál járunk, pedig a feladatok zöménél erre feltétlenül szükségünk van. Ezért aztán a programozás során nem ezt alkalmazzuk, hanem definiálunk egy ún. ciklusváltozót, amely számlálja a végrehajtásokat (innen az elnevezés), egy adott kezdőértéktől elkezdve egészen a végértékig. A szerkezet tehát a következőképpen néz ki:

```
Ciklus ciklusváltozó:=kezdőérték-től végérték-ig (ez a ciklusfej)
    Ciklusmag
Ciklus vége
```

Ilyenkor a végrehajtások számát úgy kapjuk meg, hogy a végértékből kivonjuk a kezdőértéket és hozzáadunk egyet (hiszen a kezdőértéket magát is bele kell számítanunk). A Turbo Pascalban a számlálós ciklus a következőképpen néz ki:

1. Ha a ciklusmag egyetlen utasításból áll, vagyis csak egyetlen utasítást kell ismételni

```
For ciklusváltozó:=kezdőérték to végérték do utasítás;
    vagy
For ciklusváltozó:=kezdőérték to végérték do utasítás;
```

2. Ha a ciklusmag több utasításból áll, akkor az utasításokat a **begin** és **end** szavak közé kell tenni, mint a teljes programot:

```
For ciklusváltozó:=kezdőérték to végérték do
    begin
        utasítás1;
        utasítás2;
        ...
    end;
```

Hogyan működik egy ilyen ciklus? A ciklusfejhez érve a ciklusváltozó felveszi a kezdőértéket. Majd végrehajtnak a ciklusmag utasításai. A ciklus végéhez érve a ciklusváltozó értéke 1-gyel nő. Ha ez az érték nagyobb a végértéknél, akkor a végrehajtás a ciklus utáni első programson folytatódik, ha pedig még nem nagyobb, akkor a ciklusmag hajtnak végre ismét, majd a ciklusvéghez érve a ciklusváltozó értéke megint 1-gyel nő, és újra csak elágazik a program attól függően, hogy a túlhaladta-e már ez az érték a végértéket, vagy sem.

A ciklusváltozó bármilyen sorszámozott típus lehet, tehát egész, karakter vagy logikai (a gyakorlatban persze legtöbbször egész szám), és természetesen ugyanúgy deklarálni kell, mint bármely más változót. A kezdő- és végérték típusának kompatibilisnek kell lennie a ciklusváltozó típusával, de nem csak konkrét érték lehet, hanem változó, konstans vagy kifejezés is.

Figyelni kell arra, hogy a ciklusváltozó a kezdőértéktől a végértékig egyesével számol el, mégpedig úgy, hogy a ciklusváltozó előző értékéhez minden végrehajtás után 1-et hozzáad; ezért a kezdőértéknek feltétlenül nagyobbnak kell lennie a végértéknek! Ha a kezdőérték egyenlő a végértékkel, akkor a ciklusmag utasításai egyszer hajtnak végre, ha pedig a végérték a kisebb, akkor egyszer sem.

Amennyiben a ciklusváltozó értéke nem nő, hanem csökken (persze ugyanúgy egyesével), akkor a **to** szócska helyett a **downto** szócskát kell alkalmazni.

A ciklusmagban szereplő sorokat beljebb írtuk, de ezt nem a Turbo Pascal követeli meg tőlünk, hanem saját magunknak szól. A **For** jelzi a ciklus elejét, és mivel a ciklusmaghoz tartozó sorokat egy kicsit beljebb írjuk, könnyen megtalálható a ciklus után következő programsor. Úgy nevezzük ezt, hogy a programot strukturáltan írjuk le. A továbbiakban ezt a fajta leírást következetesen alkalmazni fogjuk.

2.4.3. Egyszerű ciklusok

Most pedig lássunk néhány egyszerű példát a ciklusokra!

28. példa: Írassuk ki a nevünket tízszer egymás alá!

Megoldás: A kiíratás az a tevékenység, amit ismételni kell, tehát ez kerül a ciklusmagba. Mivel ez csak egyetlen utasítás, ezért a **begin-end** szópár elhagyható. A ciklusváltozót (**i**) deklarálni kell. A nevet egy konstansban tároljuk (persze erre nem feltétlenül van szükség, csak a gyakorlás végett).

```
Program Ciklus1;
Uses Crt;
Var
  i : byte;
Const
  nev = 'Zseni Alfonz';
Begin
  ClrScr;
  For i:=1 to 10 do
    WriteLn(nev);
  ReadLn;
End.
```

A ciklusváltozónak is célszerű „beszédes” nevet választani. Ha nincs jelentősége a névnek (mint ebben a példában is), akkor bármilyen azonosító megfelel. Hagyományosan a ciklusváltozó az **i**, **j**, stb. szokott lenni, amittől persze el lehet térni.

29. példa Írjuk ki a nevünket többször egymás alá a képernyőre! A kiírások számát a billentyűzetről kérjük be!

Megoldás: Az előző kis programunkat módosítjuk. Szükségünk van egy új változóra, amely a kiírások számát tárolja (**db**). Ennek értékét bekérjük, majd a ciklusban felhasználjuk.

```
Program Ciklus2;
Uses Crt;
Var
  i, db : integer;
Const
  nev = 'Zseni Alfonz';
Begin
  ClrScr;
  Write('Hányszor írjam ki a nevet? ');
  ReadLn(db);
  ClrScr;
  For i:=1 to db do
    WriteLn(nev);
  ReadLn;
End.
```

Próbáljuk ki a programot különböző értékekkel! Csak arra vigyázzunk, hogy a képernyőn kell maradni, azaz 1 és 25 közötti számot célszerű megadni. Erre esetleg a kiírásban lehet is figyelmeztetni: „Hányszor írjam ki a neved? (1-25) „. Persze ez még nem garancia arra, hogy a program, valóban jó értéket is kap, de a későbbiekben erre is megtanuljuk az orvosságot.

Feladatok:

58. Írassa ki a nevét egymás alá 5-ször a 6. sorban kezdve!
59. Írassa ki a keresztnévét a 20. sorba egymás mellé 3-szor!
60. Írassa tele a képernyő 10. sorát „@” jelekkel!
61. Kérjen be egy számot 1 és 24 között (ReadLn), és a bekért számnak megfelelő sort írassa tele „@” jelekkel!
62. Írasson ki a képernyő 12. sorában a 30. oszlopnál kezdve 20 db „@” jelet!
63. Írassa ki a keresztnévét a 12. sorba egymás mellé annyiszor, ahányszor kifer!

2.4.4. A ciklusváltozó felhasználása a ciklusmagban

Az eddigi példáinkban és feladatainkban nem láttuk hasznát annak, hogy a ciklusváltozó számlálja, hogy a ciklusmagot hányszor hajtotta már végre, csupán arra volt szükségünk, hogy összesen 80-szor, vagy 20-szor, vagy valahányszor ismétlje meg. A ciklusváltozó azonban a ciklusmagban felhasználható, és ezzel nagyon hasznossá tudja magát tenni. A konkrét feladatok nagy részében erre szükségünk is van.

Egy fontos figyelmeztetés: a ciklusváltozó értéke felhasználható a ciklusmagban, de nem célszerű megváltoztatni, mert akkor a ciklus lefutásainak a számát befolyásoljuk! Nézzünk egy példát:

30. példa: Írassuk ki egymás alá a számokat növekvő sorrendben 1-től 20-ig!

Megoldás: A ciklusmagban most egy szám kiírása szerepel, de ez a szám folyamatosan változik. A ciklusváltozó szintén folyamatosan változik, az értéke mindig azt mutatja, hogy most éppen hányadszor hajtottuk végre a ciklusmag utasításait. A feladatunk az, hogy először ($i=1$) az 1-es számot, másodszor ($i=2$) a 2-es számot, ... 20-adszor ($i=20$) pedig a 20-as számot kell kiíratnunk. Figyeljük meg, hogy az i változónak minden esetben pontosan azonos az értéke a kiíratandó számmal! A megoldás tehát: írassuk ki az i változó értékét!

```
Program CiklusValt1;
Uses Crt;
Var
  i : integer;
Begin
  ClrScr;
  For i:=1 to 20 do
    Writeln('i=', i);
  Readln;
End.
```

31. példa Módosítsuk az előző példát úgy, hogy a számokat visszafelé írjuk ki (tehát 20-tól 1-ig)!

1. megoldás: Most visszafelé számlálunk a ciklusfejben, tehát a **to** helyett a **downto** alkalmazandó. Persze ekkor nem 1-től 20-ig, hanem 20-tól 1-ig kell számlálni.

```
Program CiklusValt2;
Uses Crt;
Var
  i : integer;
Begin
  ClrScr;
  For i:=20 downto 1 do
    Writeln(i);
  Readln;
End.
```

2. Megoldás: Ha a ciklusfejhez nem nyúlunk hozzá, akkor a kiírásban lehet valamit változtatni. Az i értékét valahogyan át kell alakítani. Pontosán a kívánt eredményt adja a $21-i$ képlet. (Ha az $i=1$, akkor a $21-i=20$; ha az $i=2$, akkor a $21-i=19$, stb.)

```
Program CiklusValt3;
Uses Crt;
Var
  i : integer;
Begin
  ClrScr;
  For i:=1 to 20 do
    Writeln(21-i);
  Readln;
End.
```

32. példa: Írassunk ki a képernyőre 20 db „#” karaktert a bal felső sarokból indulva átlósan (tehát az első # jel az 1. sor 1. oszlopában, a második # jel a 2. sor második oszlopában legyen, és így tovább)!

Megoldás: Most a ciklusváltozó értékét a kiírás helyének meghatározására (GotoXY) használjuk fel, hiszen minden alkalommal a sor és az oszlop sorszáma megegyezik a ciklusváltozó értékével. Mivel a ciklusmag most már két sorból áll, ezért a **begin-end** szavak közé kell tennünk.

```
Program CiklusValt4;
Uses Crt;
Var
  i : integer;
Begin
  ClrScr;
  For i:=1 to 20 do
    begin
      GotoXY(i, i);
      Writeln('#');
    end;
  Readln;
End.
```

33. példa Írjuk ki a számokat 50-től 60-ig egymás alá!

Megoldás: nem törvényszerű, hogy a ciklusváltozónak 1-től kell számolnia ...

```
Program CiklusValt5;
Uses Crt;
Var
  i : integer;
Begin
  ClrScr;
  For i:=50 to 60 do
    Writeln(i);
  Readln;
End.
```

Feladatok:

64. Írassa ki a számokat 1-től 20-ig egymás alá, de a 40. oszlopba!
65. Módosítsa az előző feladatot úgy, hogy a számokat 20-tól 1-ig visszafelé írassa ki!
66. Írassa ki a számokat 50-től 60-ig egymás alá, de a 20. oszlopba!
67. Módosítsa az előző feladatot úgy, hogy a számokat 60-tól 50-ig visszafelé írassa ki!
68. Módosítsa az előző feladatot úgy, hogy a kiíratás az 5. sorban kezdődjön!
69. Írasson ki a képernyőre 20 db „@” karaktert a jobb felső sarokból átlósan lefelé és befelé!
70. Írasson ki a képernyőre 20 db „@” karaktert a bal alsó sarokból átlósan felfelé és befelé!
71. Írasson ki a képernyőre 20 db „@” karaktert a jobb alsó sarokból átlósan felfelé és befelé!
72. Írassa ki a számokat 1-től 20-ig a bal felső sarokból indulva átlósan lefele és befelé!
73. Írassa ki a számokat 1-től 20-ig a jobb felső sarokból indulva átlósan lefele és befelé!
74. Írassa ki a számokat 1-től 20-ig a bal alsó sarokból indulva átlósan felfele és befelé!
75. Írassa ki a számokat 1-től 20-ig a jobb alsó sarokból indulva átlósan felfele és befelé!
76. Írassa ki a számokat 50-től 60-ig a bal felső sarokból indulva átlósan lefele és befelé!
77. Írassa ki a számokat 50-től 60-ig a jobb felső sarokból indulva átlósan lefele és befelé!
78. Írassa ki a számokat 50-től 60-ig a bal alsó sarokból indulva átlósan felfele és befelé!
79. Írassa ki a számokat 50-től 60-ig a jobb alsó sarokból indulva átlósan felfele és befelé!
80. Írassa ki a számokat 100-től 80-ig visszafelé a bal felső sarokból indulva átlósan lefele és befelé!
81. Írassa ki a számokat 100-től 80-ig visszafelé a jobb felső sarokból indulva átlósan lefele és befelé!
82. Írassa ki a számokat 100-től 80-ig visszafelé a bal alsó sarokból indulva átlósan felfele és befelé!
83. Írassa ki a számokat 100-től 80-ig visszafelé a jobb alsó sarokból indulva átlósan felfele és befelé!

34. példa Írassuk ki a képernyő 20. oszlopába a számokat 1-től 20-ig egymás alá, és minden szám mellé írassuk ki a kétszeresét is!

Megoldás: A feladatnak két értelmes megoldása lehetséges, attól függően, hogy az oszlopokat vagy a sorokat íratjuk-e ki először.

1. megoldás: Először írassuk ki a számokat, majd a második lépésben a kétszeresüket! Egy oszlop megjelenítéséhez egy ciklus kell, két oszlophoz tehát két ciklus egymás után.

```

Program CiklusValt6;
Uses Crt;
Var
    i : integer;
Begin
    ClrScr;
    For i:=1 to 20 do
        WriteLn(i);
    For i:=1 to 20 do
        begin
            GotoXY(10,i);
            WriteLn(i*2);
        end;
    ReadLn;
End.

```

Az első ciklus íratja ki a számokat egymás alá az 1. oszlopba. A második ciklus végzi a számok kétszeresének kiíratását a 20. oszlopba. A két ciklusban ugyanazt a ciklusváltozót használtuk (i). Ez nem okoz gondot, hiszen az első ciklus lefutása után az értékére már nincs szükségünk.

2. megoldás: Egy lépésben íratjuk ki a számot, és mellé a kétszeresét. Ekkor csak egy ciklus kell. Persze a két érték között egy kis helyet kell hagyni.

```

Program CiklusValt7;
Uses Crt;
Var
    i : integer;
Begin
    ClrScr;
    For i:=1 to 20 do
        WriteLn(i, '    ', i*2);
    ReadLn;
End.

```

Látható, hogy ez a megoldás tömörebb, egyszerűbb, de ugyanazt a megoldást adja, mint az előző. A feladatok nagy része többféleképpen is megoldható, és amennyiben a jó eredményt kapjuk, akkor a megoldás jónak tekinthető. Azonban mégis vannak különbségek az egyes programok között végrehajtási időben, bonyolultságban, stb. Egy kész program esetén lehet gondolkodni egy esetleges jobb, hatékonyabb megoldáson.

Feladatok:

84. Írassa ki 20-tól 30-ig a számokat, majd melléjük a számok kétszeresét, valamint a náluk 5-tel kisebb számot (pl. 25, 50, 20)!
85. Írassa ki 10-től 30-ig a számok 3-szorosát!
86. Írassa ki 1-től 20-ig a számok kétszeresénél 1-gyel kisebb számot! Milyen számsorozatot kapott?
87. Készítsen programot, amely kiírja az 5-ös számra vonatkozó szorzótábla-részletet 20-ig a következő formában:
 $1 * 5 = 5$
 $2 * 5 = 10$
...
 $20 * 5 = 100!$

A ciklusváltozó segítségével komolyabb számításokat is tudunk végezni. Nézzünk erre is egy példát.

35. példa: Adjuk össze a számokat egytől tízig!

Megoldás: A feladat az $1+2+3+4+5+6+7+8+9+10$ összeg kiszámítása. Ehhez szükség lesz egy változóra, ami az összeget tárolja (legyen ez az **összeg**). A feladatot felbontjuk elemi lépésekre:

```
összeg = 1
összeg = 1+2 = 3
összeg = 1+2+3 = 3+3 = 6
összeg = 1+2+3+4 = 6+4 = 10 ...stb.
```

Egy lépésben csak egy összeadást végzünk el. A ciklusunk 1-től 10-ig fog menni, tehát a ciklusváltozó értéke pontosan azokon a számokon megy végig, amelyeket az addigi összeghez hozzá kell adni.

```
Program Osszeg;
Uses Crt;
Var
  i, osszeg : integer;
```

```
Begin
  Clrscr;
  osszeg:=0;
  For i:=1 to 10 do
    osszeg:=összeg+1;
  WriteLn('1-10-ig a számok összege: ',összeg);
  Readln;
End.
```

A programban először az **összeg** változó kezdőértéket kap (ha ez elmarad, a Turbo Pascal automatikusan a 0 (nulla) kezdőértéket adja a változónak). A számítást az első látásra kissé nehezen emészthető **összeg:=összeg+i** sor végzi el. A sor jelentése a következő: az **összeg** változó előző értékéhez adjuk hozzá az **i** változó értékét, és ezt az új értéket tároljuk mostantól az **összeg** változóban. Természetesen nem feledkezünk el az eredmény kiírásáról sem (**WriteLn**), ez a ciklus után helyezkedik el, hiszen csak a végösszegre vagyunk kíváncsiak, a részeredményekre nem. Ha esetleg nagyobb számokkal is számolunk, akkor már érdemes az **összeg** változót **longint**, vagy még inkább **real** típusúra deklarálni, hiszen az **integer** típussal leírt legnagyobb szám csak 32767 lehet.

Feladatok:

88. Készítsen programot, amely 50-től 150-ig adja össze a számokat!
89. Készítsen programot, amely összeszorozza 1-től 10-ig a számokat! Vigyázzon a kezdőértékre!
90. Adja össze az első N természetes számot (az N értékét kérje be)!
91. Készítsen programot, amelyben kérjen be egy tetszőleges számot, és ettől a számtól kezdve adjon össze tíz számot!
92. Készítsen programot, amely egy tetszőleges kezdő- és végérték között adja össze a számokat (a kezdő- és végértéket egyaránt kérje be)!

2.4.5. Lépésköz kezelése

Az eddigi példáinkban a ciklusváltozó értéke mindig 1-gyel nőtt vagy csökkent a végrehajtások során. Sok esetben azonban szükségünk lehet arra, hogy a ciklusváltozó valamilyen lépésközzel változtassa az értékét. Erre a Turbo Pascal programozási nyelv nem kínál kényelmes megoldást, mint például a Basic nyelvek többsége¹⁰, úgyhogy némi ravaszkodásra van szükség. Általában jó megoldás, ha a ciklus pontosan annyiszor fut le, ahányszor kell neki, a lépésközt pedig egy segédváltozó kezeli le. Nézzük a következő példát:

¹⁰ A Basic-ben a ciklusfejtében elhelyezett STEP szócska után megadhatjuk a lépésközt.

36. példa Írassuk ki a képernyőre 10-től 100-ig a számokat 10-esével egymás alá!

Megoldás: Nekünk az lenne kényelmes, ha a ciklusváltozó végigszaladna a 10, 20, ..., 100 számokon tízesével. Erre nincs lehetőség, tehát más megoldást keresünk. A megadott számokon nem a ciklusváltozó, hanem egy másik változó (legyen ez *j*) megy végig. Összesen tíz darab számot kell kiíratni, tehát a ciklus 10-szer fut le (1-től 10-ig). A *j* változó értékét pedig a ciklusváltozó értékéből állítjuk elő a ciklusmagban.

```
Program Lepeskozi1;
Uses Crt;
Var
  i, j : integer;
Begin
  ClrScr;
  For i:=1 to 10 do
    begin
      j:=i*10;
      WriteLn(j);
    end;
  ReadLn;
End.
```

*Persze megtehettük volna, hogy a *j* változó bevezetése helyett közvetlenül az *i**10 értéket íratjuk ki, mint ahogyan az előző fejezetben is láttunk erre példát, de ez a megoldás általánosabb. Ha az *i**10 értéket többször kell felhasználni, akkor célszerűbb csak egyszer kiszámolni, eltárolni egy változóban az értéket, és ahol szükség van rá, ott a változót használni.*

37. példa Írassuk ki minden második sorba a nevünket a képernyő aljáig, az első sorban kezdve!

Megoldás: Mivel 25 sor van, ezért a nevet 12-szer kell kiíratni (azaz az *i* értéke 1-től 12-ig megy). A kiíratás tényleges helyét egy segédváltozó (*j*) adja meg. Most azonban nem elég az, ha az *i* értékét kettővel szorozzuk, mert akkor a kiíratás a második sorban kezdődne, tehát az értékéből még egyet le is kell vonni.

```
Program Lepeskoz2;
Uses Crt;
Var
  i, j : integer;
```

```
Begin
  ClrScr;
  For i:=1 to 12 do
    begin
      j:=i*2-1;
      GotoXY(1, j);
      WriteLn('Zseni Alfonz');
    end;
  ReadLn;
End.
```

Egy másik jó megoldás az lehetne, ha kihagyjuk a játékból a segédváltozót. Ekkor a ciklusmagban először kiíratjuk a nevet, majd egy üres sort:

```
For i:=1 to 12 do
  begin
    WriteLn('Zseni Alfonz');
    WriteLn;
  end;
```

Ez a megoldás azonban csak akkor lehet célravezető, ha a lépésköz elég kicsi (jelen példánkban ez 2).

Feladatok:

93. Készítsen programot, amely az első sorban minden második oszlopba ír egy „@” karaktert!
94. Készítsen programot, amely a képernyőre minden harmadik sorba kiírja a „Hello, te zseni!” szöveget!
95. Írassa ki a 100, 200, 300, stb. számokat egymás alá 2000-ig a 60. oszlopba!
96. Írassa ki a 25, 50, 75 stb. számokat egymás alá a 20. oszlopba (500-ig), és minden szám mellett írassa ki a kétszeresénél 1-gyel nagyobb számot is!
97. Írassa ki 66-tól csökkenő sorrendben az összes pozitív, 3-mal osztható számot!
98. Írassa ki 100-tól 50-ig visszafelé az összes 5-tel osztható szám kétszeresét!

2.4.6. Egymásba ágyazott ciklusok

Egymásba ágyazott ciklusokról akkor beszélünk, ha egy ciklus magjában („belsejében”) egy vagy több újabb ciklus található. A ciklusmagban lévő ciklust belső ciklusnak, míg a befoglaló ciklust külső ciklusnak nevezzük.

38. példa Írjuk tele a képernyő első 20 sorát „#” karakterekkel!

Megoldás: Egy sor teleírását már megoldottuk egy ciklus segítségével. Most ezt kell megismételni 20-szor, akkor tehát ez is egy ciklus. A belső ciklus egy sor kiíratását végzi, ennek változója tehát egy soron belül az oszlopokon szalad végig. A külső ciklus változója mutatja tehát a sorokat.

```
Program Ciklusok1;
Uses Crt;
Var
    sor, oszl : integer;
Begin
    Clrscr;
    For sor:=1 to 20 do
        For oszl:=1 to 80 do
            begin
                GotoXY(oszl, sor);
                Write('#');
            end;
        Readln;
    End.
```

A külső ciklusban azért nem kell használni a **begin-end** párt, mert egy ciklus csak egyetlen utasításnak számít. A belső ciklusban minden egyes „#” jel kiíratása előtt a kurzort a megfelelő helyre pozícionáljuk, ami tulajdonképpen felesleges, hiszen a **Write** alkalmazásával a kurzor úgyis ott marad az utoljára kiíratott jel mögött. Eleget csak minden egyes sor kiíratása előtt a megfelelő helyre tenni a kurzort.

2. megoldás:

```
Program Ciklusok2;
Uses Crt;
Var
    sor, oszl : integer;
Begin
    Clrscr;
    For sor:=1 to 20 do
        begin
            GotoXY(1, sor);
            For oszl:=1 to 80 do
                Write('#');
            end;
        Readln;
    End.
```

Most a külső ciklus már több utasításból áll, ezért a **begin-end** szavak közé kell tennünk. A belső ciklus viszont csak a kiíratást tartalmazza, ott nem kell használni.

39. példa Módosítsuk az előző programot úgy, hogy nyomon követhető legyen a kiíratás sorrendje! Ehhez minden egyes „#” kiíratása után várakozzunk egy kicsit!

Megoldás: A program végrehajtását egy időre fel lehet függeszteni a **Delay** eljárás használatával. Alkalmazása:

Delay (ms)

ahol az ms milliszekundumot (ezredmásodpercet) jelent. A **Delay** eljárás a **Crt** unitban van, tehát a program a **Delay** sorhoz érve a paraméterben megadott ezredmásodpercig ott várakozik.

```
Program Ciklusok3;
Uses Crt;
Var
    sor, oszl : integer;
Begin
    Clrscr;
    For sor:=1 to 20 do
        begin
            GotoXY(1, sor);
            For oszl:=1 to 80 do
                begin
                    Write('#');
                    Delay(10);
                end;
            end;
        Readln;
    End.
```

Most már nyomon követhető a kiíratás sorrendje: látható, hogy sorfolytonosan íratunk ki, azaz először egy sort írnak tele, majd a következőt, és így tovább. Ebben a példában mind a külső, mind a belső ciklus maga több utasításból áll, ezért mind a kettőnél szerepel a **begin-end** szópár. Felhívom a figyelmet arra, hogy a két **end** közül az első a belső ciklust zárja le, a második pedig a külsőt. Általában is igaz: mindig le kell zárni a belső ciklust, mielőtt a külsőt lezárnánk!

A legutolsó példában már háromszor szerepel a **begin** és az **end**. A program strukturált begépelésének haszna most látszik igazán: pontosan felismerhető, hogy melyik **begin**-hez melyik **end** tartozik, vagyis

hogyan hol van a struktúra eleje és vége. Egy jó tanács: a program gépelésekor a **begin** begépelése után közvetlenül az **end** következzen, azután majd lehet közéjük írni a megfelelő utasításokat, így biztosan nem felejtünk el egy-két **end**-et beírni a programba (ez egyébként meglehetősen gyakori hiba).

Feladatok:

99. Írja tele a képernyő első 20 sorát „@” jelekkel, de oszlopolymonosan (azaz először az első oszlopot íród tele majd a következőt, ...stb.)!
100. Írja tele a képernyő közepét (a 10. és 15. sorok, illetve a 30. és 50. oszlopok által határolt területet) „@” jelekkel!
101. Készítsen programot, amely a képernyő minden második sorát teleírja „@” karakterekkel!
102. Módosítsa az előző feladatot úgy, hogy az egyes sorokban csak minden harmadik „@” kerüljön kiíratásra!

Végezetül nézzünk egy olyan példát, amely nagyon látványos, amellet nagyon egyszerű, és persze az egymásba ágyazott ciklusokat is nagyon jól példázza.

40. példa Készítsünk a képernyő közepére egy működő digitális órát, ami 5 percig jár!

Megoldás: Lássuk csak, hogyan is kell működni: végig kell számolnunk 0-tól 59-ig (másodpercek), ez egy perc. Az órának 5 percig kell járnia, tehát ezt a számlálást 5-ször kell végrehajtani. A belső ciklus tehát a másodperceket számlálja, a külső a perceket. Az időket természetesen másodpercenként ki kell írni, mindig a képernyő ugyanazon helyére. Azt, hogy a másodpercek tényleg egy másodpercenként kövessék egymást, arról a már megismert `Delay(1000)` eljárás segítségével tudunk gondoskodni.

```
Program Ora;
Uses Crt;
Var
    mp,perc : integer;
Begin
    Clrscr;
    For perc:=0 to 4 do
        For mp:=0 to 59 do
```

```
begin
    GotoXY(37,12);
    Writeln(perc,' : ',mp:2);
    Delay(1000);
end;
```

End.

A programcska különösebb magyarázatra nem szorul. Talán csak annyi, hogy a valós számok kiíratásánál megismert formázott kiíratás egészek esetén is alkalmazható, csak itt persze nem adható meg a tizedesek száma. Az mp:2 forma azt mutatja, hogy írjuk ki az mp értékét kettő karakterhelyen.

Elképzelhető, hogy az előző program futását alig győztük kivárni, vagy már az elején nyilvánvaló, hogy a program rosszul működik. Ilyenkor a program futását meg lehet szakítani a CTRL-Break billentyűkombináció ismételt lenyomásával.

Feladatok:

103. Módosítsa az órák programot úgy, hogy ne csak percet és másodpercet számoljon, hanem órákat is! (Ötlet: ehhez egy harmadik, legkülső ciklus kell. A programot kipróbálhatjuk, ha a Delay értékét csökkented.)
104. Készítsen számjegyes számlálót a képernyőre (amilyen a villanyórákban vagy kilométerórákban is van)! A számláló először két számjegyes legyen (tehát 00-tól 99-ig számlál), majd bővítsé fokozatosan hat számjegyesig!
105. Készítsen szorzótáblát a képernyőre (1-től 10-ig)!

2.5. Elágazások

A programunk írása során gyakran kerülünk olyan helyzetbe, hogy egy feltételtől függően különböző tevékenységeket kell csinálnunk, azaz a programot különböző helyeken kell folytatni. Ezt a fajta szerkezetet elágazásnak nevezzük.

41. példa Készítsünk programot, amelyben bekérünk két számot és kiíratjuk a hányadosukat!

```
Program Osztas;
Uses Crt;
Var
    a,b : integer;
    c : real;
```

```

Begin
  Clrscr;
  Write('Kérem az első számot: ');
  Readln(a);
  Write('Kérem a második számot: ');
  Readln(b);
  WriteLn;
  c:=a/b;
  Writeln('A két szám hányadosa: ',c:5:2);
  Readln;
End.

```

Futtassuk le a programot, és próbáljuk ki a 12 és 0 számokkal! A program a 200. futási hibával áll le: nullával való osztás. Persze, hiszen az osztás művelet nem mindig végezhető el, nullával való osztást nem értelmezzük.

Mi lehetne a fenti problémára a megoldás? Az osztás előtt meg kellene vizsgálni, hogy az elvégezhető-e. Ha igen, akkor végezzük el, ha nem, akkor pedig írassuk ki, hogy „nullával való osztás értelmetlen”. A programot el kell ágaztatni.

2.5.1. Elágazás kétfelé

A kétfelé ágaztatás sémája a következő:

HA feltétel igaz, AKKOR utasítás(ok) KÜLÖNBEN utasítás(ok)

Vagyis, ha a feltétel teljesül (igaz), akkor azok az utasítások hajtódnak végre, amelyeket az AKKOR után írtunk, ha pedig a feltétel nem teljesül, akkor azok, amelyeket a KÜLÖNBEN után írtunk.

A feltételben valamilyen logikai kifejezésnek kell szerepelnie, azaz olyanoknak, amelynek az eredménye IGAZ vagy HAMIS (az eldöntendő kérdésre hasonlít). A feltételben használható relációk és logikai műveletek a változókról szóló fejezetben részletesen ismertetésre kerültek.

A Turbo Pascal-ban az elágazást az **IF-THEN-ELSE** (HA-AKKOR-KÜLÖNBEN) szavakkal képezzük:

```

If feltétel
then utasítás(ok)
else utasítás(ok);

```

Amennyiben akár a **then**, akár az **else** után több utasítás szerepel, akkor a **begin-end** szópár között kell szerepeltetnünk azokat, a ciklusoknál már megismert módon.

Ha a feladat olyan, hogy a feltétel nem teljesülése esetén semmilyen tevékenységet nem kell végezni, akkor az **else**-ág elmaradhat.

42. példa Írjuk át az előző példaprogramot úgy, hogy az osztás csak akkor hajtódjon végre, ha a második szám nem nulla!

Megoldás: Használjuk az előbb megismert szerkezetet!

```

Program Osztas;
Uses Crt;
Var
  a,b : integer;
  c : real;
Begin
  Clrscr;
  Write('Kérem az első számot: ');
  Readln(a);
  Write('Kérem a második számot: ');
  Readln(b);
  Writeln;
  If b=0
  then Writeln('Nullával nem osztunk!!!')
  else
    begin
      c:=a/b;
      Writeln('A két szám hányadosa: ',c:5:2);
    end;
  Readln;
End.

```

Az elágazást nagyon sokféleképpen lehet strukturálni, ki-ki megtalálhatja azt, ami neki a legmegfelelőbb és a legjobban tetszik. A könyvben a fenti strukturálást fogjuk követni. Az elágazásban az **else** előtti utolsó sor mögé nem szabad kitenni a pontosvesszőt, mert az az elágazás lezárását jelentené!

Egy másik megoldás: Ha a különben (azaz **else**) nem állna a rendelkezésünkre, akkor a feladatot **If-then** szerkezetek ismételt alkalmazásával is megoldhatjuk az alábbi módon (csak az elágazás változik, tehát a program többi részét nem írom le újra):

```

If b=0 then Writeln('Nullával nem osztunk!!!');
If b<>0 then
begin
  c:=a/b;
  Writeln('A két szám hányadosa: ',c:5:2);
end;

```

Most nem egy, hanem két elágazást alkalmazunk. A megoldás hátránya, hogy a második feltétel kiértékelése teljesen feleslegesen történik meg, hiszen akkor teljesül, amikor az első feltétel nem; úgy-

hogy a két megoldás közül mindenképpen az első a jobb. Vannak azonban olyan helyzetek, amikor ez a legegyszerűbb. A programrészlet leírásánál most egy kicsit másképp strukturáltuk, csak hogy bemutassunk néhány más lehetőséget is.

A programok írása során az elágazásokat úgy kell kezelni, hogy minden lehetséges esetet – minden „utat”, ami felé elágaztathatunk –, figyelembe kell venni. Érdekes először a számunkra rossz esetet kezelni, hiszen az nem vezet sehová, a program eredménye szempontjából zsákcát jelent.

43. példa Kérjünk be egy számot, és döntsük el róla, hogy 1 és 10 között van-e!

Megoldás: Már nem elég egy egyszerű feltételt megfogalmazni, összetett feltételre van szükség

```
Program Elagazas;
Uses Crt;
Var
  a : integer;
Begin
  Clrscr;
  Write('Kérek egy számot: ');
  Readln(a);
  WriteIn;
  If (a>=1) and (a<=10)
    then WriteIn('A szám 1 és 10 köze esik')
    else WriteIn('A szám nem 1 és 10 köze esik');
  Readln;
End.
```

Az összetett feltételben azt mondtuk ki, hogy ha a szám nagyobb 1-nél és kisebb 10-nél, akkor ... A zárójeleket azért kellett használni, mert a precedencia-sorban az ÉS művelet előbb van, mint a relációk, tehát előbb is hajtódna végre. (Összetett feltételekben a zárójel hiánya elég gyakori hiba a Turbo Pascal-ban!)

Feladatok:

106. Kérjen be egy számot, és ha negatív, akkor szorozza meg (-1)-gyel, majd írja ki az eredményt!
107. Kérjen be egy számot, és írja ki, hogy a szám pozitív-e (pozitív, nem pozitív)!
108. Kérjen be egy számot, és döntse el, hogy kisebb-e 10-nél (kisebb, nem kisebb)!
109. Kérjen be egy számot, és döntse el, hogy 100 és 200 között van-e (közötte van, nincs közöttte)!

110. Kérjen be egy számot, és írja ki, hogy kétjegyű-e (kétjegyű, nem kétjegyű)!
111. Kérjen be egy számot, és döntse el, hogy lehet-e érdemjegy!
112. Kérjen be egy számot, és döntse el, hogy lehet-e ez egy szín kódja (0-15) (lehet, nem lehet)!
113. Kérjen be két számot, és írassa ki a nagyobbikat!
114. Kérjen be két számot, és a nagyobbikból vonja ki a kisebbiket!
115. Kérjen be két számot, és írja ki, hogy egyenlők-e (egyenlők, nem egyenlők)!

2.5.2. Elágazás többfelé

Néhány esetben a lehetséges esetek száma nem kettő, hanem három vagy több; ekkor többfelé kell elágaztatnunk a programot.

44. példa Kérjünk be két számot, és írassuk ki, hogy melyik a nagyobb! Három lehetséges eredmény van: 1. az első kisebb, 2. az első nagyobb, 3. egyenlők.

1. megoldás: Három felé kell elágaztatni a programot, ezt pedig a legegyszerűbben három elágazással valósíthatjuk meg.

```
Program Nagyobb1,
Uses Crt;
Var
  a,b : integer;
Begin
  Clrscr;
  Write('Kérem az első számot: ');
  Readln(a);
  Write('Kérem a második számot: ');
  Readln(b);
  WriteIn;
  If a<b then WriteIn('A második szám a nagyobb');
  If a>b then WriteIn('Az első szám a nagyobb');
  If a=b then WriteIn('A két szám egyenlő');
  Readln;
End.
```

A megoldásról a következő hátrányok írhatók le: ha az első feltétel ($a < b$) igaznak bizonyult, akkor a másik kettőről eleve tudjuk, hogy csak hamis lehet, és ezért feleslegesen vizsgálja meg a program. Ha pedig az első két feltétel egyaránt hamis, akkor a harmadik vizsgálatra nincs is szükség.

2. megoldás: Használjuk az **else** kulcsszót! Mivel most három-felé kell ágaztatni, az **else**-ágon egy további elágazásra van szükség. (csak az elágazásos részt írjuk újra)

```

If a<b
then WriteIn('A második szám a nagyobb')
else
  If a>b
  then WriteIn('Az első szám a nagyobb')
  else WriteIn('A két szám egyenlő');

```

A megoldás kis hátránya, hogy a programot bonyolulttá, nehezebben áttekinthetővé teszi. Mégis ennek a használatát ajánljuk, mert ez a megoldás elegánsabb és hatékonyabb. Ilyen módon természetesen többfelé is elágaztathatunk, mint három.

Feladatok:

116. Kérjen be egy számot, és írassa ki, hogy a szám pozitív, negatív vagy nulla!
117. Kérjen be két számot, és adja meg a köztük lévő relációt (az első szám a nagyobb, a második szám a nagyobb vagy egyenlő!)

2.5.3. Elágazás többfelé esetszétválasztással

Amennyiben egy változó vagy kifejezés lehetséges értékei szerint kell elágaztatni (általában többfelé) akkor érdemes a **Case** utasítást használni. Az utasítás formája:

```

Case szelektor-kifejezés of
  konstans1 : utasítás(ok);
  konstans2 : utasítás(ok);
  ...
  else utasítás(ok);
end;

```

A szelektor-kifejezés (vagyis az a kifejezés vagy változó, aminek lehetséges eseteit vizsgáljuk) csak sorszámozott típusú lehet, azaz egész, karakter vagy logikai. Fontos még az is, hogy longint típus használatakor sem léphetünk túl az integer típus értelmezési tartományán. A case-konstansok típusának egyeznie kell a szelektor típusával. Nem feltétlenül csak egy érték szerepelhet itt, hanem lehet felsorolás (az egyes értékeket vesszővel kell elválasztani), illetve megadható tartomány is (ekkor a „kezdőérték..végérték” formát kell alkalmazni).

Ha az elágazás egyes ágain több utasítás szerepel, akkor azt a **begin-end** szópár közé kell tenni.

Az **else**-ágon azokat az utasításokat kell szerepeltetni, amelyeket akkor akarunk végrehajtatni, ha egyik eset sem teljesült a felsoroltak közül. Az **else**-ág el is maradhat, nem kötelező a használata.

A **Case**-szerkezetet az **end;** zárja le minden esetben (**begin** nincs!).

45. példa Kérjünk be egy osztályzatot (pl. matekjegyet), és a kapott eredményt értékeljük szövegesen!

Megoldás: A feladatot a **Case** szerkezettel adjuk meg.

```

Program Matek;
Uses Crt;
Var
  jegy : integer;
Begin
  Clrscr;
  Write('Hányasod volt matekból év végén? ');
  Readln(jegy);
  WriteIn;
  Case jegy of
    1 : WriteIn('Csak nem megbuktál?');
    2..4 : WriteIn('Matekból bármilyen jegy jó...');
    5 : WriteIn('Gratulálok! Te ki zseni!');
    0,6 : WriteIn('Mellényomtasd a számot?');
  else WriteIn('Érdekesen osztályoz a tanarod.');
```

```

end;
Readln;
End.

```

A feladatnak van egy általánosítható tanulsága: az elágazásban nem csak azokat az értékeket kell lekezelni, amelyeket józan ésszel várhatók, hanem minden lehetséges értéket!

Feladatok:

118. Kérjen be egy számot, és döntse el, hogy lehet-e életkor! Ha igen, akkor adja meg a korcsoportot, amibe tartozik (gyerek–fiatal–felnőtt–idős), egyébként a program adjon hibaüzenetet!
119. Kérdezze meg, hogy a hét hányadik napja van, és adja meg ezt a neki megfelelő névvel (hétfő, kedd, stb.)! Ne feledkezzen meg azokról az esetekről sem, amikor nem 1 és 7 közötti számot kap a program!
120. Sorszámaival megadott hónapot adja meg a nevével!
121. Kérjen be egy számjegyet, és adja meg római számmal! Ha a szám nem számjegy, akkor adjon hibaüzenetet!

122. Kérjen be egy számot 1 és 999 999 között! Eredményül adja meg, hogy a szám hány jegyű! Ha nem esik a megadott értéktartományba, azt is jelezze ki!

2.6. Feltételes ugrások

46. példa Írjunk programot, amelyik eloszt egymással két számot

Megoldás: A programot már elkészítettük egyszer, úgyhogy semmi nehézséget nem okozhat a felírása:

```
Program Osztas;
Uses Crt;
Var
  a,b   integer;
  c : real;
Begin
  Clrscr;
  Write('Kérem az első számot: ');
  Readln(a);
  Write('Kérem a második számot: ');
  Readln(b);
  Writein;
  If b=0
  then Writein('Nullával nem osztunk!!!')
  else
    begin
      c:=a/b;
      Writein('A két szám hányadosa: ',c:5:2);
    end;
  Readln;
End.
```

A megoldás egy kicsit suta, hiszen amennyiben a második szám nulla, akkor csak egy hibaüzenetet adunk, hogy az osztás nem elvégezhető. Jó lenne újra megkérdezni a második számot. Csakhogy mi van akkor, ha ez is nulla lesz? Nem tudjuk, hogy hányszor kell újra és újra megkérdezni, mikor kegyeskedik végre a felhasználó valamilyen nullától különböző értéket gépelni. Azt kellene tehát elérni, hogy a vezérlés visszakérüljön arra a sorra, melyiktől kezdve ismételni akarunk! Erre szolgál a **Goto** (ugorj) utasítás. Használata:

Goto címke

ahol a címke azonosítja azt a sort, amelyikre ugrani akarunk. A címkét ugyanúgy deklarálni kell, ahogyan a változókat és konstansokat. A címkék deklarációja a **Label** kulcsszó után következhet.

47. példa Módosítsuk az osztásos feladatot úgy, hogy addig kérje be a számot újra meg újra, amíg csak nullától különböző értéket nem kapunk!

Megoldás: Az elágazásban adunk egy hibaüzenetet, majd visszaugrunk a második adatbekérés elé. Ehhez egy **ujra** nevű címkét deklarálunk. Ebben az esetben viszont az **else**-ág feleslegessé válik, kikerülhet az elágazásból, hiszen a kritikus $c:=a/b$ sorra csak akkor kerülhet a vezérlés, ha **b** értéke nem nulla (hiszen ha nulla, akkor újra bekérjük).

```
Program Osztas;
Uses Crt;
Label ujra;
Var
  a,b   integer;
  c : real;
Begin
  Clrscr;
  Write('Kérem az első számot: ');
  Readln(a);
  ujra:
  Write('Kérem a második számot: ');
  Readln(b);
  Writein;
  If b=0
  then
    begin
      Writein('Nullával nem osztunk! Adj meg másikat!');
      Goto ujra;
    end;
  c:=a/b;
  Writein('A két szám hányadosa: ',c:5:2);
  Readln;
End.
```

Mit is csinál most ez a programocska? Bekéri az első számot, majd bekéri a másodikat is. Ez utóbbit megvizsgálja, hogy nem nulla-e, és ha igen, akkor hibaüzenetet ad, és visszaugrik a második szám bekéréséhez (azt a sor azonosítottuk az **ujra** nevű címkével). Ha a második szám nem nulla, akkor a program végre túljut az elágazáson, és elvégezheti valódi feladatát, az osztást. Fontos, hogy a címke után mindig kettőspontot (:) kell tenni, a Turbo Pascal innen ismer rá, hogy ez egy címke. A címkét nem feltétlenül kell külön sorba tenni, állhat egy sor elején, az utasítás előtt is. A címke egyébként bármilyen azonosítót kaphat (eleje, vege, pityuka, stb.), de mindazok a szabályok vonatkoznak rá, amelyek a változók azonosítóira is vonatkoznak.

Figyeljük meg, hogy az a szerkezet, amelyet a **Goto** segítségével valósítottunk meg, az valójában egy ciklus! A ciklusmagban a második szám bekérése van, a ciklus pedig addig-addig hajtódik végre, amíg csak egy feltétel ($b=0$) hamissá nem válik. Ez azonban nem számlálós ciklus – nem tudjuk megadni előre, hogy a ciklusmagot hányszor kell végrehajtani –, hanem ún. tesztelős ciklus. Ezek szervezését tárgyalja a következő fejezet.

A feltételes ugrás azonban egy kicsit egyszerűbb, természetesebb, kezdőknek mindenképpen ajánlatos a megértése, mielőtt belemerül-nénk a tesztelős ciklusokkal való programozásba, bár hosszú távon ezek alkalmazása az ajánlatosabb.

Fontos megjegyzés: a **Goto** utasítás alkalmazása csak feltételhez kötött – tehát egy elágazás **then**- vagy **else**-ágán – engedhető meg. Bár a Turbo Pascal a használatát nem korlátozza, de a feltétel nélküli ugrások (vagyis egy önálló **Goto** utasítás) a programot áttekinthetetlen-né teszik, hiszen a logikus sorrend felborul, ha össze-vissza ugrálunk a programban.

Még egy megjegyzés: a fenti program alkalmas lehet bármilyen adatbekérés tesztelésére (értékhatarok ellenőrzésére).

A programjaink eddig úgy működtek, hogy a futás végeztével szépen maguktól leálltak. De mi van akkor, ha szeretnénk esetleg többször egymás után használni? Le kell futtatni többször is, ez azonban nem elég kényelmes és elegáns.

48. példa Kérjünk be két számot, irassuk ki a különbségüket, majd kérdezzük meg a felhasználót, hogy akar-e még számolni – ha igen, akkor kezdjük a programot előlről.

Megoldás: Most a program végén lesz szükség egy feltételes ugrásra. Megkérdezzük: „Akarsz még számolni?“, és az igen válasza visszaugrunk a program elejére. Az igenlő választ egy „i” vagy „I” karakter lenyomása fogja jelenteni (miután a két karakter a számítógép szempontjából különböző, mind a kettővel kell foglalkozni!). Ió lenne tehát egy olyan adatbekérési lehetőség, amelyik egyetlen karaktert vár csak a billentyűzetről. Van is ilyen, a **ReadKey** függvény. Használata:

```
karateres_változó:=ReadKey;
```

A program ehhez a sorhoz érve vár mindaddig, amíg le nem nyomunk egy billentyűt. A karakteres változó (char típusú) tárolja a lenyomott billentyűt. A **ReadKey** függvény a **Crt** unitban van.

```
Program Kilepes;
Uses Crt;
Label eleje;
Var
  a,b,k : integer;
  c : char;
Begin
  eleje:
    Clrscr;
    Write('Kérem az első számot: ');
    Readln(a);
    Write('Kérem a második számot: ');
    Readln(b);
    WriteLn;
    k:=a-b;
    WriteLn('A két szám különbsége: ',k);
    GotoXY(1,24);
    Write('Akarsz még egyet számolni? (i/n) ');
    c:=ReadKey;
    If (c='i') or (c='I') then Goto eleje;
End.
```

Figyelje meg, hogy a lenyomott billentyű után nem kell az **ENTER**-t is lenyomni! Fontos, hogy a kérdést úgy tegyük fel, hogy a rá adandó válasz egyértelmű legyen a felhasználó számára. Példánkban a ki-íratásban szerepeltettük, hogy az **i** vagy **n** billentyűre várunk. Bármilyen egyéb billentyű hatására a program befejeződik. Most már felesleges lett az eddig a program végére tett **Readln**; sor. Természetesen módosíthatnánk a működését úgy, hogy csak az **i** vagy **n** karaktereket fogadja el a program – ezt egy további feltételes ugrás valósítaná meg. Ekkor a program vége a következőképpen nézne ki:

```
Write('Akarsz még egyet számolni? (i/n) ');
ujrabill;
c:=ReadKey;
If (c<>'i') and (c<>'I') and (c<>'n') and
(c<>'N')
then Goto ujabill;
If (c='i') or (c='I')
then Goto eleje;
End.
```

Ez a programcska már csak az **i**, **I**, **n**, **N** karaktereket fogadja el a billentyűzetről, és addig várja újra meg újra a leütést, amíg csak meg

nem kapja valamelyiket. Látható, hogy a programba még egy címke bekerült, ennek azonban más a neve, mint az előzőnek. Általános szabály, hogy két azonos nevű azonosító nem szerepelhet a programban.

Feladatok:

123. Kérjen be egy 1 és 10 közötti számot (tesztelje, hogy tényleg 1 és 10 közötti-e, és csak ekkor fogadja el!), és írassa ki a szám négyzetét (önmagával való szorzatát)!
124. Módosítsa az előző feladatot úgy, hogy a végén kérdezze meg, hogy vége legyen-e már!
125. Kérjen be egy negatív számot (tesztelje!), majd írassa ki a szám -5-szörösét!
126. Módosítsa az előző feladatot úgy, hogy a végén kérdezze meg, hogy vége legyen-e már!
127. Kérjen be egy szöveget és egy számot, ami egy szín kódja, és a bekért számot jelenítse meg a bekért színnel (természetesen tesztelje, hogy létező színkódot (0-15) kap-e)!
128. Módosítsa az előző feladatot úgy, hogy a végén kérdezze meg, hogy vége legyen-e már!
129. Kérjen be egy számot 20 és 30 között, majd egy másikat 100 és 200 között (mindkettőt tesztelje)! A nagyobbikból vonja ki a kisebbet, és közölje az eredményt!
130. Módosítsa az előző feladatot úgy, hogy a végén kérdezze meg, hogy vége legyen-e már!

Megjegyzendő, hogy a Goto utasítást célszerű kerülni, mert könnyen áttekinthetetlenné teszi programjainkat, s a strukturált programozáshoz sem illik. Lehetőség szerint használjunk más megoldásokat.

2.7. Tesztelős ciklusok

Az előző fejezetben már szóltam arról, hogy a feltételes ugrások valójában ciklusokat valósítottak meg. Volt szó arról is, hogy ezek a ciklusok nem lehetnek számlálós ciklusok, hiszen nem tudtuk megmondani, hogy hányszor futtassák le a ciklusmagban szereplő utasításokat. A ciklusok minden esetben addig futottak, amíg valamilyen feltétel igaz (vagy hamis) nem lett. A feltételek teljesülését természetesen minden egyes végrehajtás előtt vagy után vizsgálni (tesztelni) kell. Ha a

ciklusmag előtt tesztelünk, akkor előtesztelős, ha a ciklusmag után, akkor hátulatesztelős ciklusokról beszélünk.

Természetesen elől- és hátulatesztelős ciklusok is ágyazhatóak egymásba, sőt bármilyen típusú ciklus magjába kerülhet más ciklus.

A tesztelős ciklusok is rendelkezhetnek ciklusváltozóval. Ebben az esetben a ciklusváltózó kezdőértékéről, illetve a ciklusban való megváltoztatásáról nekünk kell gondoskodnunk.

Még egy megjegyzés: a tesztelős ciklusok közül bármelyik bármelyikre átirható a feltétel megváltoztatásával. A számlálós ciklus is megadható tesztelős ciklusok segítségével.

2.7.1. Előtesztelős ciklusok

Előtesztelős ciklusokat a következőképpen szervezhetünk:

```
While feltétel do utasítás;
```

vagy (ha a ciklusmagban több utasítás szerepel):

```
While feltétel do
begin
utasítások;
end;
```

A ciklusmagban szereplő utasítások addig hajtódnak végre, amíg a feltétel IGAZ, vagyis akkor ér véget a ciklus, amikor a feltétel HAMISsá válik.

2.7.2. Hátulatesztelős ciklusok

Hátulatesztelős ciklusokat a következőképpen szervezhetünk:

```
Repeat
utasítások;
Until feltétel;
```

A ciklusmagban szereplő utasítások addig hajtódnak végre, amíg a feltétel HAMIS, vagyis akkor ér véget a ciklus, amikor a feltétel IGAZZá válik.

2.7.3. Példa a tesztelős ciklusokra

49. példa Írassuk ki a számokat 20-tól 40-ig a képernyőre egymás alá tesztelős ciklus alkalmazásával!

Emlékeztető: A programot már megírtuk számlálós ciklussal (csak akkor 50-től 60-ig mentünk). Nézzük most ezt emlékeztetőül:

```
Program Ciklus1;
Uses Crt;
Var
  i : integer;
Begin
  ClrScr;
  For i:=20 to 40 do
    WriteLn(i);
  ReadLn;
End.
```

1. megoldás: Most előtesztelős ciklussal valósítjuk meg a feladatot.

```
Program Ciklus2;
Uses Crt;
Var
  i : integer;
Begin
  ClrScr;
  i:=19;
  While i<40 do
    begin
      i:=i+1;
      WriteLn(i);
    end;
  ReadLn;
End.
```

2. megoldás: Most hátulatesztelős ciklussal valósítjuk meg a feladatot.

```
Program Ciklus3;
Uses Crt;
Var
  i : integer;
Begin
  ClrScr;
  i:=20;
  Repeat
    WriteLn(i);
    i:=i+1;
  Until i>40;
  ReadLn;
End.
```

A példában szereplő három programocska tökéletesen ugyanazt hajtja végre. A tesztelős ciklusoknál a különbség a ciklusfeltételben van (egymásnak pont az ellentettét kell felírunk). Általában is igaz, hogy a háromféle ciklus közül bármely feladathoz bármelyik alkalmazható. Azt, hogy az egyes esetekben melyik ajánlható, azt a következő szempontok döntenek el:

- Ha a program írásakor előre tudom, hogy a ciklusnak hányszor kell lefutnia, akkor a számlálós ciklust célszerű alkalmazni, egyébként tesztelős ciklust.
- A kétféle tesztelős ciklus között végrehajtási szempontból egyetlen lényeges különbség van: a hátulatesztelős ciklus magja egyszer biztosan lefut (hiszen csak ez után tesztelek, hogy ki kell-e lépni a ciklusból), míg az előtesztelős ciklusoknál előfordulhat, hogy a feltétel már a legelején sem igaz, ezért a ciklusmag egyszer sem fut le. Tehát ha biztosak vagyunk benne, hogy a ciklusmagot egyszer biztosan le kell futtatni, akkor mindenképpen érdemes a hátulatesztelős ciklust használni, míg ellenkező esetben az előtesztelős célszerűbb.

A fenti szempontok azonban csak ajánlások, hiszen mint említettük, a legtöbb feladat megoldható a három ciklus bármelyikével.

2.7.4. Feladatok tesztelős ciklusokra

Tesztelős ciklusoknak gyakori alkalmazási területe a bekért adatok tesztelése (értékhatar-helyesség ellenőrzése). Ilyen típusú feladatot már oldottunk meg a **Goto** segítségével, de ciklussal elegánsabb (ha csak tehetjük, kerüljük a **Goto** alkalmazását).

50. példa Kérjünk be egy számot 1 és 10 között, és írassuk ki a képernyő közepére!

Megoldás: A tesztelős ciklust az adatbekérésre alkalmazzuk, hiszen addig kell újra meg újra bekérni az adatot, amíg csak helyes értéket nem kapunk. A ciklusmagban tehát az adatbekérés szerepel. Elöl- vagy hátulatesztelős ciklust alkalmazzunk? A kérdés az dönti el, hogy a ciklusmagot végre kell-e hajtani legalább egyszer. A válasz **IGEN**, tehát célszerűbb hátulatesztelős ciklus alkalmazása. (Természetesen előtesztelős ciklussal is megoldható a feladat.)

```

Program Tesztelese;
Uses Crt;
Var
    a : integer;
Begin
    Clrscr;
    Repeat
        Write('Mondj egy számot 1 és 10 között! ');
        Readln(a);
    Until (a>=1) and (a<=10);
    Clrscr;
    GotoXY(35,12);
    WriteIn('A számod: ',a);
    Readln;
End.

```

A bekérést megvalósító Repeat-Until ciklus akkor ér véget, amikor a feltétel ((a>=1) and (a<=10)) igazgá nem válik, vagyis amikor 1 és 10 közti számot kapunk

2. megoldás: Kicsit elegánsabb a megoldás, ha rossz érték esetén hibaüzenetet is adunk.

```

Program Tesztelese2;
Uses Crt;
Var
    a : integer;
Begin
    Clrscr;
    Write('Mondj egy számot 1 és 10 között! ');
    Repeat
        Readln(a);
        If a<1 then Write('Ez túl kicsi! Mondj másikat: ');
        If a>10 then Write('Ez túl nagy! Mondj másikat: ');
    Until (a>=1) and (a<=10);
    Clrscr;
    GotoXY(35,12);
    WriteIn('A számod: ',a);
    Readln;
End.

```

Ennek a megoldásnak apró hátránya, hogy a teszteléseket többször végezzük el, de mivel a különböző hibák esetén különböző hibaüzeneteket adunk, ez elkerülhetetlen. Ha egyszerűen csak a hiba tényét szeretnénk jelezni, akkor előltesztelési ciklussal ez is elérhető:

3. megoldás: Elöltesztelési ciklussal oldjuk meg a feladatot.

```

Program Tesztelese3;
Uses Crt;
Var
    a : integer;
Begin
    Clrscr;
    Write('Mondj egy számot 1 és 10 között! ');
    Readln(a);
    While (a<1) or (a>10) do
        begin
            Write('1. és 10 közötti számot kértél! Mondj másikat: ');
            Readln(a);
        end;
    Clrscr;
    GotoXY(35,12);
    WriteIn('A számod: ',a);
    Readln;
End.

```

Ebben a megoldásban a ciklus csak az ismételt bekérésekkel foglalkozik, tehát az első bekérést a ciklus előtt külön meg kell adni. Természetesen az is megoldható lenne, hogy a ciklusmagban (a 2. megoldáshoz hasonlóan) teszteljük a különböző hibalehetőségeket, és külön hibaüzenetet adunk mindenféle hibára.

A tesztelési ciklusok egy másik gyakori felhasználási formája az, ha a programból való kilépést szabályozzuk vele. Azaz: addig hajtjuk végre az egész programot, amíg a felhasználó azt nem mondja, hogy elég volt. Ilyen típusú feladatot is oldottunk már meg a Goto segítségével, azonban itt is elmondom, hogy ha csak tehetjük, kerüljük a Goto alkalmazását.

51. példa Kérjünk be két tetszőleges számot, adjuk össze őket, majd a végén kérdezzük meg, hogy legyen-e vége!

Megoldás: A feladatot már megoldottuk a feltételes ugrások segítségével, de most inkább tesztelési ciklust kellene alkalmazni. A teljes program egy hátultesztelési ciklus magjába kerül.

```

Program Kilepes;
Uses Crt;
Var
    a,b,k : integer;
    c : char;
Begin

```

```

Repeat
  Clrscr;
  Write('Kérem az első számot: ');
  Readln(a);
  Write('Kérem a második számot: ');
  Readln(b);
  WriteIn;
  k:=a-b;
  WriteIn('A két szám különbsége: ',k);
  GotoXY(1,24);
  Write('Akarsz még egyet számolni? (i/n) ');
  c:=ReadKey;
Until (c<>'i') and (c<>'I');
End.

```

Vagyis a programból akkor lépünk ki, ha a leütött billentyű nem *i* vagy *I*. Persze ezt a programot is módosíthatjuk úgy, hogy csak az *i*, *I*, *n*, *N* billentyűt fogadjuk el.

2. megoldás: Ehhez egy újabb hátultesztelési ciklusra van szükség. (A programnak csak a végét írjuk le.) Ekkor a program végét is egyszerűbben tesztelhetjük.

```

...
GotoXY(1,24);
Write('Akarsz még egyet számolni? (i/n) ');
Repeat
  c:=ReadKey;
Until (c='i') or (c='I') or (c='n') or (c='N');
Until (c='n') or (c='N');
End.

```

52. példa Írjunk programot, amely egy tanuló jegyeinek átlagát számítja ki! Az érdemjegyeket egyesével kérjük be, a bekérést a 0 (nulla) zárja le!

Megoldás: Nem tudjuk, hány jegyet kérünk be, de azt tudjuk, hogy egészen addig, amíg nullát nem kapunk. A feladatot tesztelési ciklussal kell tehát megoldani. Kell két segédváltozó is, amelyek az addig bekért jegyek darabszámát (*db*), valamint összegét (*s*) tartalmazzák. Természetesen arról, hogy az érdemjegy valóban 1 és 5 közötti szám, egy tesztelési ciklus segítségével gondoskodunk.

```

Program Jegyek;
Uses Crt;
Var
  jegy, db, s : integer;
  atlag : real;

```

```

Begin
  ClrScr;
  WriteIn('Kérem a jegyeidet egyesével, ha vege,
    akkor a nullát üsd le!');
  s:=0;
  db:=0;
  Repeat
    db:=db+1;
    Repeat
      Write('Kérem a(z) ',db,'. jegyet: ');
      Readln(jegy);
    Until (jegy>=0) and (jegy<=5);
    s:=s+jegy;
  Until jegy=0;
  db:=db-1;
  If db=0
  then WriteIn('0 darab jegynek nincs átlaga!')
  else
    begin
      atlag:=s/db;
      WriteIn('Az átlagod: ',atlag:5:2);
    end;
  Readln;
End.

```

A ciklus utáni *db:=db-1*; utasításra azért van szükség, mert az utolsó jegy (ami a 0 volt) esetén is növeltük a *db* értékét, tehát a jegyek száma valóságosan *db-1*. A program végén pedig előfordulhat, hogy már elsőre a nullát kaptuk, vagyis nulla darab jegyet ír be valaki, ezért van szükség az elágazásra.

Feladatok:

A **Feltételes ugrások** fejezet feladatait írja meg tesztelési ciklusok alkalmazásával:

- Kérjen be egy 1 és 10 közötti számot (tesztelje!), és írassa ki a szám négyzetét (önmagával való szorzatát)!
- Módosítsa az előző feladatot úgy, hogy a végén kérdezze meg, hogy vége legyen-e már!
- Kérjen be egy negatív számot (tesztelje!), majd írassa ki a szám -5-szörösét!
- Módosítsa az előző feladatot úgy, hogy a végén kérdezze meg, hogy vége legyen-e már!
- Kérjen be egy szöveget és egy számot, ami egy szín kódja, és a bekért számot jelenítse meg a bekért színnel (természetesen tesztelje, hogy létező színek kódját kap-e)!

136. Módosítsa az előző feladatot úgy, hogy a végén kérdezze meg, hogy vége legyen-e már!
137. Kérjen be egy számot 20 és 30 között, majd egy másikat 100 és 200 között (mindkettőt tesztelje!!) A nagyobbikból vonja ki a kisebbet, és közölje az eredményt!
138. Módosítsa az előző feladatot úgy, hogy a végén kérdezze meg, hogy vége legyen-e már!
139. Adja össze a számokat 1-től 100-ig tesztelős ciklus segítségével!
140. Szorozza össze az egész számokat 1-től 5-ig tesztelős ciklus segítségével!

2.7.5. Végtelen ciklusok

Végtelen ciklusnak nevezzük az olyan ciklusokat, amelyben a feltélt – persze hibásan – úgy fogalmaztuk meg, hogy az soha ne tegye lehetővé a ciklusból való kilépést. Iól működő programokban a végtelen ciklusoknak természetesen nincs helyük, de elég gyakori hiba, főleg kezdő programozóknál, ezért érdemes külön foglalkozni velük. Néhány gyakoribb hibaforrást mutatok be a következőkben (minden példában csak a ciklus szerepel):

53. példa Kérjünk be egy számot 1 és 10 között!

Hibaforrás: Valamilyen meggondolatlanág vagy egyszerű elgépelés miatt olyan feltételt fogalmaztunk meg, ami egyetlen értékre sem teljesülhet soha. Pl. a feltételben a relációs jelek „véletlenül megfordultak”.

Helyes:

Repeat

Read (a);

Until (a>=1) and (a<=10).

Helytelen:

Repeat

Read (a);

Until (a<=1) and (a>=10);

54. példa Kérjünk be egy számot 1 és 10 között!

Hibaforrás: Véletlenül a ciklus elé tettük az adatbekérést, és a ciklus üres. Ez akkor okoz gondot, ha az elsőnek begépelt szám rossz.

Helyes:

Repeat

Read (a);

Until (a>=1) and (a<=10).

Helytelen:

Read (a);

Repeat

Until (a>=1) and (a<=10);

55. példa Adjuk össze a számokat 1-től 10-ig tesztelős ciklus segítségével!

Hibaforrás: A ciklusváltozó kezdőérték-adása véletlenül a cikluson belülre került.

Helyes:

i:=1;

s:=0;

While i<=10 **do**

begin

s:=s+1;

i:=i+1;

end;

Helytelen:

s:=0;

While i<=10 **do**

begin

i:=1;

s:=s+1;

i:=i+1;

end;

56. példa Adjuk össze a számokat 1-től 10-ig tesztelős ciklus segítségével!

Hibaforrás: A ciklusváltozó értékét elfelejtjük növelni a ciklus belsejében (nagyon gyakori hiba!).

Helyes:

i:=0;

s:=0;

Repeat

i:=i+1;

s:=s+i;

Until i=10;

Helytelen:

s:=0;

i:=0;

Repeat

s:=s+i;

Until i=10;

2.8. Alprogramok

Az alprogramok a programtól függetlenül megírt, de ott felhasználott, önálló programrészek. Akkor alkalmazhatók jól, ha

- a főprogram egy részét, amely önállóan is megállja a helyét, az áttekinthetőség kedvéért alprogramban írjuk meg, vagy
- olyan programrészeknél, amelyek sokszor ismétlődnek, és a többszöri megírás helyett az adott részt csak egyszer készítjük el, és többször meghívjuk.

Az alprogramok kétféleképpen lehetnek: eljárások és függvények.

Az eljárásokat és függvényeket először el kell készíteni, majd a program megfelelő helyéről le kell őket futtatni, vagyis meg kell őket hívni. Az egyes alprogramok egymásból is hívhatók.

Az alprogramok pontosan úgy épülnek fel, mint a főprogram: lehet deklarációs része, és a főprogramhoz hasonlóan magát a programot a **Begin-End** szópárnak kell közrefognia.

Az alprogramokat a deklarációs rész után, és a főprogram elé kell tenni. Az alprogramok egymás közötti helyzete (vagyis hogy melyik van előbb, és melyik később) lényegtelen, csupán egyetlen szabály van: még azelőtt kell szerepelniük, hogy valamelyik másik eljárásban hivatkozunk rájuk.

2.8.1. Eljárások

Egy eljárás általános szerkezete a következő:

```
Procedure Eljarasnev;
deklarációk
Begin
utasítások;
End;
```

Az eljárásokat a nevükkel tudjuk lefuttatni, vagyis a programban ott, ahol az adott eljárást szeretnénk lefuttatni, a nevét kell szerepeltetni.

57. példa Készítsünk programot, amely két beolvasott számot összead!

Megoldás: Ilyen programot írtunk már, de most az adatbekerést és az eredmény kiírását egy-egy eljárás végzi el. A jobb áttekinthetőség érdekében a programot üres sorokkal tagoljuk, és a főprogram elkülönítésére a főprogram **BEGIN-END**-jét nagybetűkkel írjuk.

```
Program Osszead;
Uses Crt;
Var
    a,b,c : integer;

Procedure AdatBekeres;
Begin
    Clrscr;
    Write('Kérem az első számot: ');
    Readln(a);
    Write('Kérem a második számot: ');
    Readln(b);
End;

Procedure Eredmeny;
Begin
    WriteLn;
    WriteLn('A két szám összege: ',c);
End;
```

```
BEGIN
AdatBekeres;
c:=a+b;
Eredmeny;
Readln;
END.
```

A főprogram ezzel áttekinthetőbbé vált, hiszen rövidebb. (Hosszabb és bonyolultabb programok főprogramjai sokszor semmi mást nem tartalmaznak, csak eljárások hívását.)

2.8.2. Paraméterátadás

Az eljárások egy jó része igényel valamilyen bemenő adatot ahhoz, hogy elvégezzen egy műveletet. Ekkor ezt a bemenő adatot, amit az eljárás paraméterének nevezünk, át kell adni az eljárás hívásakor, illetve az eljárás definíciójában át kell azt venni. Ezt a folyamatot nevezzük paraméterátadásnak.

58. példa Készítsünk programot, amely a képernyő tetszőleges sorát teleírja „#” karakterekkel! A sor teleírását egy eljárás végezzel!

Megoldás: Először elkészítünk egy eljárást, amelyik az *x*. sort írja tele. Majd beleírjuk egy programba, amelyik semmi mást nem tesz, mint meghívja ezt az eljárást.

```
Program ParameterAtadas;
Uses Crt;
Var
    i : integer;

Procedure Teleir(x:byte);
Begin
    GotoXY(1,x);
    For i:=1 to 80 do
        Write('#');
    End;

BEGIN
    ClrScr;
    Teleir(6);
    Readln;
END.
```

Az eljárás fejében zárójelek közt jelezzük, hogy az eljárás milyen paramétert vár ahhoz, hogy a feladatát elvégezhesse. Amikor pedig az eljárást hívjuk, akkor szintén zárójelek között meg kell adni azt a konkrét kifejezést, amely a program bemenő paramétere lehet. Ez a kis program így a 6. sort írja tele, de ha megváltoztatjuk ezt az értéket pl. 12-re, akkor a 12. sort írja tele a programocská.

Az eljárás definíciójában az átvett paraméter nevét és típusát kell megadni úgy, ahogy az az előbbi példából látszik. Természetesen egy eljárás több paraméterrel is dolgozhat, ekkor ezeket a következőképpen kell megadni:

```
Procedure Eljnev(p1,p2,...pn:típus1; r1,r2,...rn:típus2;...);
```

ahol a p1, p2, stb. paraméterek „típus1” típusúak, míg r1, r2, stb. paraméterek „típus2” típusúak. Az azonos típushoz tartozó paramétereket egymástól vesszővel választjuk el, míg az egyes típusokat pontosvesszővel.

Az eljárás fejében felsorolt paraméterlista az ún. formális paramétereket sorolja fel. Az eljárás belsejében az egyes értékekre a formális paraméterrel hivatkozunk. Ezeket az eljárás belsejében úgy használhatjuk, mint egy bármilyen változót, de az eljáráson kívül használva fordítási hibát kapunk. A formális paramétereket nem kell, sőt, nem szabad a deklarációs részben a **Var** kulcsszó után megadni.

Az eljárás hívásakor az ún. aktuális paramétereket adjuk át. Ezek számának és sorrendjének meg kell felelnie a formális paraméterlistában megadottakkal. Az eljárás hívásakor az történik, hogy a formális paraméter felveszi kezdőértékként a neki megfelelő aktuális paraméter értékét. Az aktuális paraméterek tetszőleges kifejezések lehetnek, amelyek típusa megegyezik (vagy legalább kompatibilis) a neki megfelelő formális paraméter típusával. Az aktuális paraméter gyakran egy változó, illetve annak az értéke.

59. példa Módosítsuk az előző példát úgy, hogy annak a sornak a számát bekérjük, amelyiket tele kell írunk!

Megoldás: Szükségünk van egy új változóra, és ki kell egészíteni a programot az adatbekéréssel. Az eljárás hívásakor pedig ennek a változónak az értékét kell átadnunk.

```
Program ParameterAtadas2;
Uses Crt;
Var
  s : byte;
  i : integer;

Procedure Teleir(x:byte);
Begin
  Clrscr;
  GotoXY(1,x);
  For i:=1 to 80 do
    Write('#');
  End;
```

```
BEGIN
  ClrScr;
  Write('Hányadik sort írjam tele? ');
  Readln(s);
  Teleir(s);
  Readln;
END.
```

Futtassuk le a programot, és adjuk meg a 15-öt! Az adatbekérésnél az s változó felveszi a 15-öt értékként. Az eljárás hívásakor az s értékét (15) adjuk át (ez az aktuális paraméter). Az eljárás ezt az értéket a saját formális paraméterében, az x változóban tárolja el, és az eljárásban az x-et használja.

A fenti példák az ún. érték szerinti paraméterátadást valósították meg. Ez azt jelenti, hogy a formális paraméter elszakad az aktuális paramétertől, csak annak értékét kapja meg. A paraméterátadás másik lehetséges módja az ún. cím szerinti paraméterátadás. Ekkor az aktuális paraméter egy változó, amelynek értéke folyamatosan együtt változik a hozzá tartozó formális paraméter értékével, amikor az az eljárásban módosul. Vegyük elő azt a példát, amelyben összeadtunk két számot!

60. példa Módosítsuk az összeadás feladatát úgy, hogy a kiíró eljárás paraméterként kapja a kiíratandó értéket, és az összeadás műveletet is egy eljárás végezze!

Megoldás:

```
Program Osszead2;
Uses Crt;
Var
  a,b,c : integer;

Procedure AdatBekeres;
Begin
  Clrscr;
  Write('Kérem az első számot: ');
  Readln(a);
  Write('Kérem a második számot: ');
  Readln(b);
End;

Procedure Eredmeny(x:integer);
Begin
  Writeln;
  Writeln('A két szám összege: ',x);
End;

Procedure Osszead(var x:integer);
Begin
  x:=a+b;
End;
```

```

BEGIN
AdatBekeres;
Osszead(c);
Eredmeny(c);
Readln;
END.

```

Az összeadó eljárás az *s* változót kapja paraméterül az *x* formális paraméterben. Az *x* értékét változtatjuk az eljárásban, és látható, hogy a kiírás során az *s* változó értéke vele változott.

A cím szerinti paraméterátadás során nem a változó értéke, hanem a memóriacíme adódik át, és erre a címre definiálják az eljárás a formális paramétert. Ebből következik, hogy az értékük együtt változik. A cím szerinti paraméterátadásról a formális paraméterlistában a **var** szócska gondoskodik. Az elé a formális paraméter elé kell tenni, amelyhez tartozó aktuális paramétert együtt szeretnénk változtatni vele.

2.8.3. Függvények

Az alprogramok másik típusa a függvény. A függvények abban különböznek az eljárásoktól, hogy mindig van egy visszaadott érték; az, amit a függvény számol. A függvények szervezése pontosan ugyanúgy megy végbe, mint az eljárásoké, csak a **Procedure** kulcsszó helyett a **Function** kulcsszót kell alkalmazni. A másik – de annál lényegesebb – különbség pedig az, hogy a függvény definíciójában szerepelnie kell a **Függvénynév=visszaadott érték** sornak, illetve a függvény definíciója során a függvény fejében meg kell adni az eredmény típusát is:

Function Függvénynev(formális paraméterek):eredmény_típusa;

61. példa Kérjünk be két tetszőleges számot, és irassuk ki a nagyobbát! A nagyobbik szám megadásához használjunk függvényt!

Megoldás: Készítünk egy *Max* nevű függvényt két bemenő paraméterrel! A kimenő érték a nagyobb szám lesz.

```

Program Nagyobb;
Uses Crt;
Var
  a,b,c : integer;

Function Max(x,y:integer):integer;
Begin
  If x>y
  then Max:=x
  else Max:=y;
End;

```

```

BEGIN
Clrscr;
WriteLn('Mondj két számot, megmondom melyik a nagyobb');
Write('az első szám: ');
Readln(a);
Write('a második szám: ');
Readln(b);
c:=Max(a,b);
WriteLn;
WriteLn('A nagyobb szám: ',c);
Readln;
END.

```

Az általunk elkészített függvények – ugyanúgy, ahogyan a Turbo Pascal saját függvényei – nem alkothatnak önálló programsort, hanem csak egy kifejezésben szerepelhetnek egy értékadás jobb oldalán, vagy egy kiíratásban!

Feladatok:

141. Készítsen programot, amelyben függvények segítségével számolja ki egy téglalap kerületét és területét!
142. Készítsen függvényt, amely átvált mondjuk dollárt forintra! A függvény bemenő paraméterei legyenek a dollárösszeg, illetve az ártípus, kimenő értéke a forintösszeg!
143. Készítsen programot, amely egy függvény segítségével kiszámítja egy bekért szám négyzetét!

2.8.4. Lokális és globális változók

Az alprogramok önálló deklarációs résszel rendelkezhetnek, amit az eddigi példáinkban nem használtunk ki. Deklarálhatunk változókat, konstansokat, címkéket: mindazt, amit a főprogramban is megadhatunk. Az alprogramban deklarált azonosítók (persze ezek leggyakrabban változók) csak az adott eljárásban vagy függvényben használhatók. Ezeket lokális változóknak, míg a főprogramban deklaráltakat globális változóknak nevezzük.

A lokális változók a főprogramban nem használhatók, ha megpróbáljuk fordítási hibát kapunk.

Egy lokális változó azonosítója megegyezhet egy globális változó azonosítójával (pl. lehet *i* változó a főprogramban is és az alprogramban is), és ilyenkor a Turbo Pascal két külön változónak is kezeli őket. A főprogrambeli változó aktuális értékét az alprogram változója nem változtatja meg, a főprogramba visszakérülve a változó visszakapja eredeti értékét. Ennek demonstrálására tanulmányozzuk gondosan a

következő programot, amely nem csinál semmi értelmet, csak a lokális és globális változók használatát hivatott bemutatni.

62. példa

```

Program LokalisEsGlobalisValtozok;
Uses Crt;
Var
    a,b : integer;
    s : string;

Procedure Procl;
Var
    a:integer;
    s:string;
Begin
    a:=21;
    s:='Tyutyuka';
    WriteIn(a,' ',b,' ',s);
End;

Procedure Proc2;
Var
    b:integer;
    s:string;
Begin
    b:=1200;
    s:='Gőznajó';
    WriteIn(a,' ',b,' ',s);
End;

BEGIN
    Clrscr;
    a:=10;
    b:=55;
    s:='Hello';
    WriteIn(a,' ',b,' ',s);
    Procl;
    WriteIn(a,' ',b,' ',s);
    Proc2;
    WriteIn(a,' ',b,' ',s);
    Readln;
END.

```

A program eredménye a következő:

10	55	Hello	(a, b, s globális értékei)
-21	55	Tyutyuka	(b globális értéke; a, s lokális értékei)
10	55	Hello	(a, b, s globális értékei újra)
10	1200	Gőznajó	(a globális értéke; b, s lokális értékei)
10	55	Hello	(a, b, s globális értékei újra)

Egy jó tanács: ha nem vagyunk biztosak benne, hogy használhatunk-e azonos nevű változót, akkor inkább ne használjunk!

2.9. Összetett adattípusok

Az eddigiekben megismert adattípusok nem alkalmasak arra, hogy nagyobb mennyiségű adatot kezeljünk egyszerre. Ehhez már olyan struktúrák kellene, amelyek egyszerű adattípusokból állnak, és sok adat tárolására alkalmasak.

2.9.1. Vektorok

A vektorok talán a leggyakrabban előforduló adatszerkezetek. A vektor nem más, mint azonos típusú adatok sora, amelyet a következő ábrával szemléltethetnénk:



Itt minden egyes cellában egy-egy azonos típusú érték van. Minden egyes érték ezen kívül rendelkezik még egy indexszel is. Az index mutatja meg, hogy a vektorban hanyadik elemről is van szó, vagyis ez nem más, mint az elem sorszáma. Azért nem nevezzük sorszámnak, mert a hétköznapi életben a sorszámok mindig 1-től (esetleg 0-tól) kezdődnek, de az indexek akárhol kezdődhetnek, és akárhol végződhetnek. Az indexek bármilyen sorszámozott típusú konstansok lehetnek, nem lehetnek azonban változók.

A vektor típusú változót is természetesen deklarálni kell a **Var** kulcsszó után a következőképpen:

```

Var
    v : array[kezdőindex..végindex] of elemek_típusa;

```

A kezdő- és végindexeket két darab pont választja el egymástól.

Néhány példa vektorok megadására:

v1 : array[1..100] of integer;	(100 db egész szám)
v2 : array[0..9] of byte;	(10 db bájt 0-tól 9-ig indexelve)
v3 : array[1..7] of string[10];	(a hét napjait tárolhatjuk például ilyen módon)
v4 : array[-10..10] of word;	(itt 21 db számot tárolunk, -10-től 10-ig indexelve)
v5 : array['a'..'z'] of real;	(az index bármilyen sorszámozott típus lehet, tehát a karakter is – más kérdés, hogy a gyakorlatban ritkán fordul elő, de azért időnként hasznos lehet)

Egy adott vektor egy bizonyos elemére a következőképpen hivatkozhatunk:

`v[index]`, vagyis a `v` vektor **index**-edik eleme.

Mivel a vektorok sok elemet tartalmazhatnak, a vektorokon végzett műveleteket általában ciklusok segítségével valósítjuk meg, mégpedig számlálós ciklusokkal, hiszen az indexek értéktartományát mindig ismerjük, vagyis tudjuk, hogy a vektor hány elemet tartalmaz.

63. példa Kérjünk be egy 10 elemű vektorba 10 számot, és adjuk meg az átlagukat!

Megoldás: Az adatok bekérését és a számítást egy-egy eljárás, illetve függvény végzi majd. A vektort deklarálni kell, és az egyes alprogramokban egy-egy számlálós ciklus segítségével oldhatjuk meg a feladatot.

```
Program AtlagSzamitas;
Uses Crt;
Var
  i : integer;
  a : real;
  v : array[1..10] of integer;

Procedure AdatBekeres;
Begin
  For i:=1 to 10 do
    begin
      Write('Mi a(z) ', i, '. szám? ');
      Readln(v[i]);
    end;
End;

Function Atlag:real;
Var osszeg : longint;
Begin
  osszeg:=0;
  For i:=1 to 10 do
    osszeg:=osszeg+v[i];
  Atlag:=osszeg/10;
End;

BEGIN
  Clrscr;
  Writein('Kérek 10 számot, megmondom az átlagukat');
  Adatbekeres;
  a:=Atlag;
  Writein('A számok átlaga: ',a:5:2);
  Readln;
END.
```

Feladatok:

144. Készítsen eljárást, amelyik egy vektor adott elemét megcseréli az utána-következővel! (Használjon közbenső változót!)
145. Készítsen eljárást, amelyik egy vektor két adott elemét megcseréli egymással!
146. Készítsen eljárást, amely egy vektor elemeit ciklikusan eggyel jobbra mozgatja úgy, hogy az utolsó elem az első helyen lépjen be!
147. Készítsen eljárást, amelyik egy vektor elemeit ciklikusan egy megadott értékkel mozgatja jobbra!
148. Készítsen eljárást, amelyik egy vektor első elemét törli úgy, hogy a többi elemet eggyel balra mozgatja!
149. Készítsen eljárást, amelyik egy vektor utolsó elemét törli úgy, hogy a többi elemet eggyel jobbra mozgatja!
150. Készítsen eljárást, amelyik egy vektor egy tetszőleges értékét törli (a törlendő elem indexét paraméterben kapja az eljárás)!
151. Készítsen eljárást, amely egy vektorba egy új elemet szűr be az első helyre (a többi adatot eggyel jobbra kell mozgatni)!

2.9.2. Tömbök

2.9.2.1. Kétdimenziós tömbök

A kétdimenziós tömbök (másképpen mátrixok) adatok táblázataként képzelhető el:

Deklarációjuk és alkalmazásuk nagyon hasonlít a vektorokéhoz, hiszen a vektor nem más, mint egy egydimenziós tömb. A mátrixnak azonban már két indexre van szüksége: egy sor- és egy oszlopindexre. Deklarációja tehát a következőképpen néz ki:

```
t : array[i1..i2,j1..j2] of elemtipus;
```

Itt az `i1` és `i2` jelenti a sorindexek kezdő- és végértékét, a `j1` és a `j2` pedig az oszlopindexek kezdő- és végértékét.

A tömb egy adott elemére pedig a $t[i, j]$ formulával tudunk; az i . sor j . elemét adva meg így.

Természetesen a mátrixok és vektorok hasonlóságából következően a mátrixok kezelésének is a ciklusok lesznek az eszközei, csak itt többnyire két, egymásba ágyazott ciklust alkalmazunk. Minden más, amit a vektoroknál leírtunk, a tömbökre is érvényes.

Feladatok:

152. Készítsen eljárást, amelyik egy numerikus tömb minden elemét megnöveli eggyel!
153. Készítsen eljárást, amelyik egy tömb egy adott sorát felcseréli a következővel!
154. Készítsen eljárást, amelyik egy tömb egy adott oszlopát felcseréli a következővel!
155. Készítsen eljárást, amelyik egy tömb két adott sorát felcseréli egymással!
156. Készítsen eljárást, amelyik egy tömb két adott oszlopát felcseréli egymással!

2.9.2.2. Három- és többdimenziós tömbök

A mátrixokhoz nagyon hasonlóan lehetőség van három- vagy többdimenziós tömbök készítésére és használatára, ekkor több index-tartományt kell megadnunk, illetve hivatkozáskor is három, vagy több indexet sorolunk fel a szögletes zárójelek között. A gyakorlatban a háromdimenziós tömb alkalmazására ritkán, több dimenziós tömbök alkalmazására szinte soha nem kerül sor, ezentúl a többdimenziós tömbök memóriaigényei is igen nagy.

2.9.3. Saját típusok létrehozása

2.9.3.1. Típusdeklaráció

Az eddigi példáinkban azokat az adattípusokat használtuk, amelyeket a Turbo Pascal a rendelkezésünkre bocsátott. Lehetőségünk van azonban saját típusok megadására és használatára is. Ezeket a **Type** kulcsszó után kell megadni a deklarációs részben:

```
Type
    típusnév = típusleírás;
```

Az így megadott saját típust ugyanúgy lehet használni pl. változók deklarálására, mint a Turbo Pascal beépített típusait.

Megtehetjük, hogy a meglévő típusokból egy újat készítünk (ami persze csak a nevében lesz új), például:

```
Type
    egész = integer;
    valós = real;
    szöveg = string[100];
```

2.9.3.2. Tömbök típusmegadása

A fenti példákkal ellentétben a tömbök esetén már kézzel fogható hasznót hajtanak a típusdefiníciók. Nézzünk egy példát!

```
Var
    t : array[1..100] of integer;
```

illetve

```
Type
    tomb = array[1..100] of integer;
Var
    t : tomb;
```

A fenti két deklaráció csaknem egyenértékű, különbség csak a használhatóságukban van, ugyanis eljárás vagy függvény paraméterlistájában csak olyan változó szerepelhet, amelynek a típusa deklarált! Vagyis a **Procedure Elj1**(t:array[1..100] of integer) helytelen, míg a **Procedure Elj2**(t:tomb) helyes megadást jelent. Általában célszerű a vektorok és tömbök típusát így módon deklarálni.

2.9.4. Rekordok

A rekord olyan adatszerkezet, amely több, általában különböző típusú, de tartalmilag-logikailag összetartozó adatot kapcsol egybe. Az egyes adatok a rekordban ún. mezőkben vannak elhelyezve. Rekord például egy személyi lap, amely tartalmazza a név, cím, életkor, házasság, stb. adatokat. A mező például a név vagy a cím. Lássuk, hogyan is kell deklarálni a rekordot:

```
Type
    típusazonosító = record
        mező11,mező12,... : típus1;
        mező21,mező22,... : típus2;
        ...
    end;
Var
    változoazonosító : típusazonosító;
```

A fentebb említett személyi lap valahogy így nézne ki:

```

Type
  szemlaptip = record
    nev, cim : string;
    kor : byte;
    hazas : boolean;
  end;
Var
  szemlap : szemlaptip;

```

Egy adott rekord egy adott mezőjére a változóazonosító.mezőnév formulával hivatkozhatunk. A példában a **szemlap** rekord típusú változó **nev** mezőjére a **szemlap.nev** hivatkozást tehetjük.

A rekordokat nagyon gyakran nem önmagában, hanem sokadmagukban alkalmazzuk oly módon, hogy egy vektor elemeit deklaráljuk rekord típusúra. A fenti példában a személyi lapokból szervezzünk egy 100 elemű vektort:

```

Var
  szemlapok : array[1..100] of szemlaptip;

```

64. példa Készítsünk programot, amelyben a fenti személyi lapot létrehozuk, és feltöltjük adatokkal; végül ezeket az adatokat kiírjuk a képernyő közepére!

Megoldás: Egy rekordban tároljuk a fent leírt módon az adatokat. Az adatok bekérését egy eljárás végzi majd.

```

Program SzemelyiLap;
Uses Crt;
Type
  szemlaptip = record
    nev, cim : string;
    kor : byte;
    hazas : boolean;
  end;
Var
  i : byte;
  szemlap : szemlaptip;

Procedure Adatbekeres;
Begin
  Writein('Kérem az adatokat:');
  Write('   Név: ');
  Readln(szemlap.nev);
  Write('   Cim: ');
  Readln(szemlap.cim);
  Write('   Életkor: ');

```

```

  Readln(szemlap.kor);
  Write('   Házas? (I:1; N:0)');
  Readln(i);
  If i=0
    then szemlap.hazas:=true
    else szemlap.hazas:=false;
End;

Procedure Kiiratas;
Begin
  Clrscr;
  Writein('Az adatok:');
  Writein('   Név: ', szemlap.nev);
  Writein('   Cim: ', szemlap.cim);
  Writein('   Életkor: ', szemlap.kor);
  If szemlap.hazas
    then Writein('   Házas? IGEN')
    else Writein('   Házas? NEM');
End;

BEGIN
  Clrscr;
  Adatbekeres;
  Kiiratas;
  Readln;
END.

```

65. példa Egy vektorban tároljuk tíz tanuló adatait. Ezek az adatok a következők: név, osztály, eredmény. Készítsük el azt a programot, amely feltölti adatokkal a rekordokat, majd kiszámítja az elsősök átlagát!

Megoldás: Egy rekordban tároljuk egy tanuló adatait, és a rekordokat egy vektorban tároljuk. Egy eljárás gondoskodik majd az adatok bekéréséről, és egy függvény az átlag kiszámításáról.

```

Program Iskola;
Uses Crt;
Type
  tanulotip = record
    nev : string;
    oszt : byte;
    jegy : byte;
  end;
Var
  i : byte;
  atlag : real;
  tanulok : array[1..10] of tanulotip;

```

```

Procedure Adatbekeres;
Begin
For i:=1 to 10 do
begin
WriteLn('Az ', i, ' tanulo adatai:');
Write('   Név: ');
ReadLn(tanulok[i].nev);
Write('   Osztály: ');
ReadLn(tanulok[i].oszt);
Write('   Érdemjegy: ');
ReadLn(tanulok[i].jegy);
end;
End;

Function Atlagszamitas:real;
Var s,db:integer;
Begin
s:=0;
db:=0;
For i:=1 to 10 do
If tanulok[i].oszt=1
then begin
s:=s+tanulok[i].jegy;
db:=db+1;
end;
If db=0
then Atlagszamitas:=0
else Atlagszamitas:=s/db;
End;

BEGIN
Clrscr;
Adatbekeres;
atlag:=Atlagszamitas;
Clrscr;
WriteLn('A 10 tanulo atlaga: ',atlag:5:2);
ReadLn;
END.

```

2.9.4.1. A rekordkezelés megkönnyítése: a With utasítás

A rekordok kezelése egyszerűsíthető oly módon, hogy a **With** utasítással mintegy kinyitjuk a rekordot, és innentől kezdve, ameddig a **With** hatálya tart – ezt egy **begin**–**end** szópár jelöli –, elég lesz a mezőnévvel hivatkozni, a rekordváltozó neve elmaradhat. Használata:

```

With rekordváltozó do
begin
...
end;

```

A személyi lapos példát elővéve a rekord egyes elemei a következőképpen kaphatnak értéket:

```

szemlap.nev:='Zseni Alfonz';
szemlap.cim:='1234. Budapest, Fő tér 2');
szemlap.kor:=35;
szemlap.hazas:=true;

```

A **With** utasítás alkalmazásával ez a programrészlet a következőképpen néz ki:

```

With szemlap do
begin
nev:='Zseni Alfonz';
cim:='1234. Budapest, Fő tér 2');
kor:=35;
hazas:=true;
end;

```

Látható, hogy a **With** használatával a programrészlet áttekinthetőbbé, egyszerűbbé vált.

Egy **With** utasítás több rekordot is „kinyithat”, ekkor ezeket vesszővel ellátva fel kell sorolni. Azoknak a rekordoknak a mezőire azonban, amelyeket nem szerepeltetünk az utasítás fejében, továbbra is a teljes névvel kell hivatkozni.

2.9.5. Fájlok

A fájl típus egy azonos típusú adatokból álló szekvenciális adatszerkezet, általában a háttértárolón. Itt most csak a deklarálással ismerkedünk meg, a fájl-ok kezelésével egy későbbi fejezet foglalkozik.

```

Var
fajlnev : file of adattípus;

```

Leggyakrabban egy fájl rekord típusú adatokat tartalmaz, de bármilyen, a Turbo Pascal-ban előre definiált, illetve a programozó által megadott típus alkalmazható.

A **fajlnev** változó egy ún. logikai fájl-változó, mielőtt bármire is használhatnánk, el kell végezni a hozzárendelését a háttértárolón található fizikai állománnyal.

Használhatunk olyan fájlokat is, amelyekhez nem határozzuk meg az adatok típusát, ezek az ún. nem tipizált fájlok. Megadásuk:

```

Var
fajlnev : file;

```

Speciális fájl-típus az, amelyik csak szöveget tartalmaz, ez az ún. szövegfájl. Megadása:

```
Var
    fajlnev : text;
```

2.9.6. Objektumok

Ha elkészítünk egy rekordot, akkor ahhoz meg kell írunk olyan eljárásokat és függvényeket, amelyek az adott rekord kezelését végzik el (adatbekérés, keresés, rendezés, stb.). Ha a rekordot megváltoztatjuk, a rekordot kezelő programokat is meg kell változtatni. Valójában tehát az adatszerkezethez szorosan hozzátartoznak az azt kezelő alprogramok is. Ebből a felismerésből született meg az objektum típus.

Az objektumot olyan speciális rekordnak tekinthetjük, amelynek nem csak adatmezői vannak, hanem az ezekkel az adatokkal dolgozó eljárásai és függvényei is, amelyeket itt metódusoknak nevezünk. Az objektumok rendelkeznek néhány olyan tulajdonsággal, amelyek a programozást megkönnyíthetik, így használatuk indokolt lehet. Alkalmazásuk azonban csak gyakorlottabb programíróknak ajánlható.

2.9.7. Halmazok

A halmazok a vektorhoz hasonló adatszerkezetek abból a szempontból, hogy több, azonos típusú adatot tartalmazhatnak. A fő különbség az, hogy a halmaz esetében az elemek sorrendje nem kötött, mint a vektornál. Úgy képzelhetjük el, mint egy bugyrot, amelybe bedobálunk különböző dolgokat. A halmaz megadása:

```
Type
    halmaznev    set of adattípus;
```

A halmaznak maximum 256 eleme lehet. Az egyes adatok típusa csak sorszámozott típus lehet, tehát egész, karakter vagy logikai.

Egy halmazba elemet tenni az értékadás során lehet, ekkor az elemeket szögletes zárójelek között kell felsorolni, vesszővel elválasztva. Használható még a .. (két pont), egy tartomány leírására.

Halmazba adatot beilleszteni az Include, belőle adatot kivenni az Exclude eljárással lehet. Használatuk:

```
Procedure Include(var h:set of T; elem:T);
Procedure Exclude(var h:set of T; elem:T);
```

Mindkét eljárásban a *T* a halmaz elemeinek típusát jelenti. A eljárások a 7.0-ás verzió újdonságai közé tartoznak, korábbi

verziókból hiányoztak. Halmazba adatot illeszteni a $H := H + \{e\}$, halmazból elemet törölni a $H := H - \{e\}$ formulával lehetett (ezek persze most is használhatók, de az eljárások hatékonyabban működnek).

Speciális művelet a halmazok esetén az *in*. Használata: „adat *in* halmaz”, eredménye pedig IGAZ, ha az adat eleme a halmaznak, HAMIS, ha nem.

66. példa A tesztelős ciklusokról szóló fejezetben szerepelt egy olyan példa, amelyben egy billentyű lenyomását várjuk:

```
...
GotoXY(1,24);
Write('Akarsz még egyet számolni? (i/n) ');
Repeat
    c:=ReadKey;
Until (c='1') or (c='I') or (c='n') or (c='N')
Until (c='n') or (c='N');
End.
```

Látható, hogy elég bonyolult tesztelést kellett alkalmaznunk annak érdekében, hogy csak az általunk megadott billentyűket fogadja el a program. Képzeljük el, milyen hosszú lenne ez a sor, ha nem csak két, hanem teszem azt öt különböző betűt fogadhatunk el. Alkalmaznunk egy halmazt!

Megoldás: Azokat a karaktereket fogjuk a halmazban elhelyezni, amelyeket szeretnénk elfogadni. A tesztelés során pedig csak azt kell eldönteni, hogy a lenyomott billentyű benne van-e a halmazban (azaz elfogadható), vagy nincsen benne. Erre az *in* operátort alkalmazhatjuk.

```
Program Halmaz;
...
Var
    kar : set of char;
...
GotoXY(1,24);
Write('Akarsz még egyet számolni? (i/n) ');
kar:={'i','I','n','N'};
Repeat
    c:=ReadKey;
Until c in kar;
Until (c='n') or (c='N');
End.
```

2.9.8. Egyéb sorszámozott típusok

2.9.8.1. Résztartomány típus

A már meglévő típusokból levezethetünk új típust is, ez az ún. résztartomány típus. A résztartományon bármely sorszámozott típus valamely részsorozatát értjük. Megadásakor a részsorozat első és utolsó elemét kell megadnunk, a tömbindexeknél megismert módon, vagyis két ponttal elválasztva. Fontos, hogy az első érték kisebb legyen a másodiknál! Nézzünk néhány példát:

Type

```
erdemjegy = 1..5;
szamjegy = '0'..'9';
nagybetu = 'A'..'Z';
ketjegy = 10..99;
```

A résztartomány típus memóriabeli helyfoglalását az elemszáma határozza meg. Az alábbi táblázat ezt ismerteti:

Elemszám	Helyfoglalás
1 .. 2^8 (=256)	1 bájt
257 .. 2^{16} (=65536)	2 bájt
65537 .. 2^{32}	4 bájt

Ennél több eleme a résztartomány típusnak nem lehet.

2.9.8.2. Felsorolás típus

A Turbo Pascal beépített sorszámozott típusaihoz hasonlókat magunk is létrehozhatunk úgy, hogy a típus elemeit egyszerűen felsoroljuk (ezért az elnevezés). Lássunk egy példát:

Type

```
evszak=(tavasz,nyar,osz,tele);
```

A felsorolásban nem szövegek vagy számok, hanem azonosítók szerepelnek, amelyeket a programon (vagy alprogramon) belül másra használni nem szabad (tehát nem lehet pl. „tavasz” nevű változónk). A felsorolás sorrendje egyben az értékek sorrendjét is jelenti, vagyis egy relációban, pl. a nyar<osz igaz lesz.

A felsorolás típus jelentősége nem túl nagy, de például tömbök indexelésére jól használható. Pl.:

Var

```
etomb : array[evszaki] of byte;
```

2.9.9. Tipizált konstansok

Nem csak a változók rendelkezhetnek típussal, hanem az állandók is. A konstansokról szóló fejezetben már említettem, hogy az egész állandók típusa longint, a valószínű real és a szövegé string. Ha ettől eltérő típusú konstans értéket szeretnénk használni, akkor a konstansnak is meg kell adni a típusát. Formája:

Const

```
azonosító : típus = érték;
```

A típus bármilyen egyszerű vagy összetett típus lehet, akár a Turbo Pascal beépített típusa, akár a felhasználó által definiált saját típus. Az egyes típusokhoz tartozó tipizált konstansok deklarálását a következő példákban jól lehet látni:

Egyszerű típussal rendelkező konstansok:

Const

```
napok : byte = 7;
uzenet : string[7] = 'Nem jó!';
pi : single = 3.14;
```

Vektor típusú konstans:

Type

```
jegytip : array[1..5] of byte;
```

Const

```
jegyek : jegytip = (1,2,3,4,5);
```

Kétdimenziós tömb típusú konstans:

Type

```
tomb : array[1..3,1..2] of byte;
```

Const

```
t : tomb = ((1,2), (3,4), (5,6));
```

Rekord típusú konstans:

Type

```
szemlep = record
    nev : string;
    kor : byte;
end;
```

Const

```
zseni : szemlep = (nev:'Zseni Alfonz';kor:35);
```

Halmaz típusú konstans:

Type

```
nalmaz = set of char;
```

Const

```
betuk : nalmaz = ['a'..'z','A'..'Z'];
```

2.10. Összefoglalás

Ebben az elég hosszúra nyúlt fejezetben a programozás alapjaival ismerkedhettünk meg. Megtanultuk, hogy melyek azok az elemek, amelyekből a programjainkat összeállíthatjuk. Minden programban a két alapvető dolog az adatok kezelése, illetve a program megírása.

A program nem más, mint egy olyan tevékenységsorozat, amely a bemenő adatokból kimenő adatot ad, ezért nagyon fontos az adatokat a feladatnak leginkább megfelelő adatszerkezetben tárolni. A legfontosabb adatszerkezetek (típusok) a következők:

Egyszerű adattípusok

Égészek (byte, shortint, word, integer, longint, comp)
Valósak (single, real, double, extended)
Szövegek (char, string)
Logikai típus (boolean)

Összetett adattípusok

Vektor (array)
Tömb (array)
Rekord (record)
Fájl (file, text)
Halmaz (set)

Az egyes típusokat változók, illetve konstansok deklarálására lehet felhasználni.

A programok utasítások sorozatából épül fel. A Turbo Pascalban használható utasítások a következők:

- Blokk eleje, vége (begin, end)
- Értékadás (:=)
- Eljárás hívása
- Ugrás (Goto)
- Elágazás kétfelé (If .. then .. else ..)
- Elágazás esetszétválasztással (Case .. of .. else .. end)
- Számlálós ciklus (For .. to/downto .. do ..)
- Elöltesztelős ciklus (While .. do ..)
- Hátultesztelős ciklus (Repeat .. Until ..)
- Rekord megnyitása (With .. do ..)

A program írása során azt kell különös tekintettel figyelembe vennünk, hogy mi az egyes programsorok (utasítások) végrehajtási sorrendje. Alapvetően a processzor szépen sorban, egymás után hajtja

végre az utasításainkat (ezt nevezzük szekvenciának). Amennyiben ettől a jól meghatározott végrehajtási sorrendtől el kívánunk térni, lehetőségünk van ún. vezérlési szerkezetek alkalmazására. A vezérlési szerkezetek – amelyek általában a Turbo Pascal utasításaival egyeznek meg – a következők:

Szekvencia

Elágazások

Elágazás kétfelé (If .. then .. else ..)
Elágazás többfelé esetszétválasztással (Case .. of .. else .. end)

Ciklusok

Számlálós ciklusok (For .. to/downto .. do ..)
Elöltesztelős ciklusok (While .. do ..)
Hátultesztelős ciklusok (Repeat .. Until ..)

Alprogramok

Eljárások (Procedure)
Függvények (Function)

A Turbo Pascal eddig megtanult eljárásait és függvényeit itt nem összegezzük, ezek megtalálhatók a **Mellékletben**.

3. Algoritmusok készítése

Ebben a fejezetben egy kicsit elszakadunk a Turbo Pascal nyelvtől, és a programok tervezésével foglalkozunk.

Az eddigiekben megismerkedtünk a Turbo Pascal, és ezzel együtt a programozás alapjaival. A programjainkat azokból az építőkövekből állítottuk össze, amelyeket megtanultunk, de az alkalmazásuk, és egyben a programok elkészítése nem gondos tervezésen alapult, hanem csak általános megfontolásokon. Ez az ún. ösztönös programozás. Komolyabb programok már csak eléggé kinkeservesen készíthetők el így, hiszen sok-sok adatot, sok és összetett vezérlési szerkezetet kell alkalmazni. Itt már a feladatot célszerű felbontani apróbb lépésekre (alprogramokra), és gondosan meg kell tervezni az egyes részek kapcsolatát, az alkalmazni kívánt adatszerkezeteket, az azokat kezelő eljárásokat és függvényeket, illetve az egyes alprogramokban használni kívánt vezérlési szerkezeteket és azok egymáshoz való viszonyát. Ezt a fajta programozást nevezzük strukturált programozásnak, és ezzel ismerkedünk az alábbiakban.

3.1. A programkészítés folyamata

A program elkészítése nem ott kezdődik, hogy nosza, ugorjunk neki a Turbo Pascalnak, aztán írjuk, és nem is végződik ott, hogy kész a program. A program készítése egy jóval összetettebb feladat, amelynek lépései a következők:

1. A feladat definiálása, meghatározása
2. A program tervezése
3. Kódolás (egy programozási nyelvre való átültetés)
4. Tesztelés, helyességvizsgálat, hibajavítás
5. Hatékonyságelemzés, a program hatékonyabbá tétele (hatékonyságon sok mindent érthetünk: gyorsaságot, kevesebb helyfoglalást, ill. a program bonyolultságát. Általában a hatékonyabb program a gyakorlatban a gyorsabb programot jelenti.)

6. Dokumentálás (a felhasználó számára szánt ún. felhasználói dokumentáció, és az esetleges továbbfejlesztéshez szükséges ún. fejlesztői dokumentáció elkészítése)

Mi eddig az első három lépést hajtottuk végre, bár nem elkülönülten, hanem egyszerre tervezve és kódolva. Ez az eljárás egyszerű gyakorlati programoknál még alkalmazható, de komolyabb, hosszabb programot már meg kell tervezni. A továbbiakban ezzel foglalkozunk, vagyis a programtervvel. A tesztelés, hatékonyságvizsgálat és dokumentálás a későbbiekben kerül bemutatásra.

3.2. A program tervezése

A program megtervezése során elvileg nem kell azzal foglalkoznunk, hogy milyen programozási nyelven fogjuk majd az adott programot elkészíteni (kódolni), bár a gyakorlatban nem tekinthetünk el tőle. A program megtervezése azt jelenti, hogy a feladat megoldását valamilyen formában leírjuk. Ez a leírás véges számú lépésből álló egyértelmű utasítások sorozata. Az ilyen leírást algoritmusnak nevezzük.

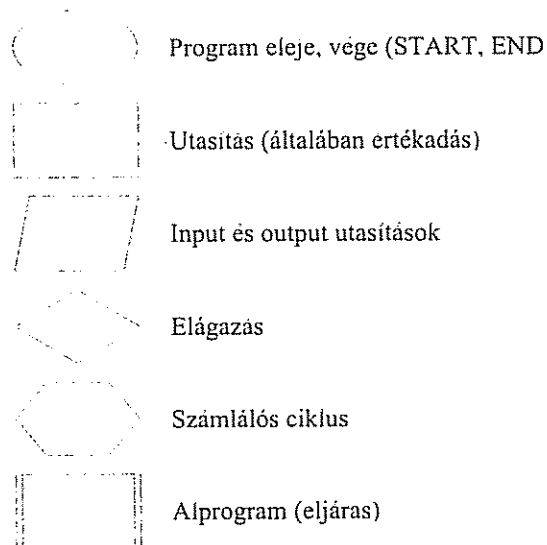
3.3. Algoritmus-leíró eszközök

Az algoritmusok használható – tömör, de mégis egyértelmű – leírása létfontosságú. Három jól használható algoritmus-leírási lehetőség áll a rendelkezésünkre: a csaknem mindenki által ismert folyamatábra, az algoritmus magyar mondatokkal való leírása, illetve a kettő elegyítéséből származó ún. struktogram.

3.3.1. Folyamatábra

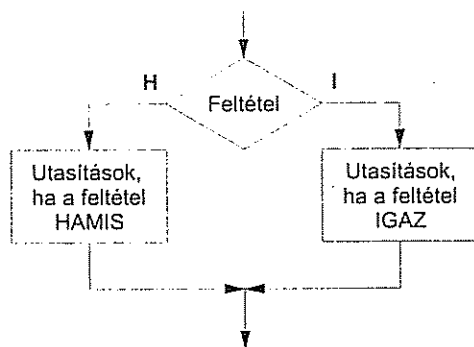
A folyamatábra a feladat ábrászerű leírását teszi lehetővé. Geometriai idomokból (amelyek az egyes utasításokat ábrázolják utasítástípusonként eltérő síkidommal), és az ezeket összekötő nyílakból áll. Előnye, hogy nagyon könnyen áttekinthető, de a rajzos jellegből következően eléggé szűkszavú leírást tesz csak lehetővé. További hátránya, hogy összetettebb programok leírásához nagyon nagy papírra van szükség. A gyors áttekinthetőség azonban olyan előny, amellyel e hátrányok nemigen versenyezhetnek, és ez a fő oka annak, hogy ez a legelterjedtebb, és leginkább ismert algoritmus-leíró eszköz.

A folyamatábra jelei a következők:

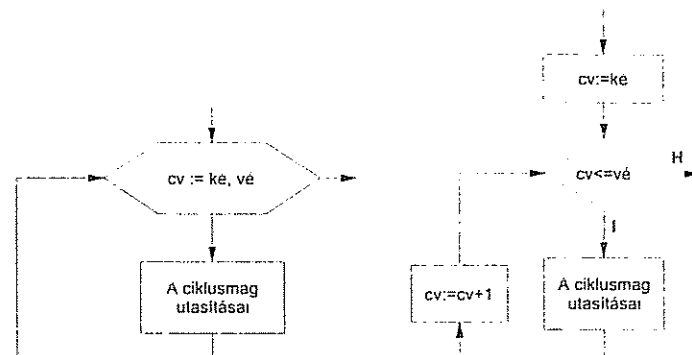


A különböző szakirodalmakban ezektől eltérő jeleket is alkalmaznak (pl. lehet külön jele a bemenetnek és a kimenetnek), de a fenti jelekkel gyakorlatilag minden leírható. Lássuk, hogy hogyan lehet a különböző vezérlési szerkezeteket folyamatábrával ábrázolni:

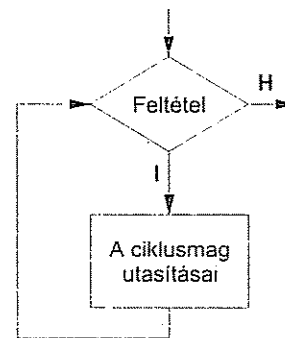
Elágazás kétféle:



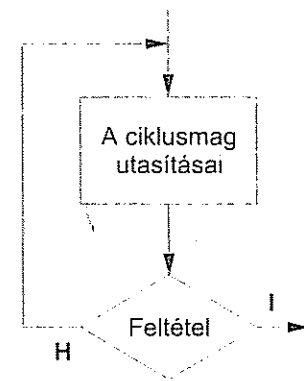
Számlálós ciklus (kétféle változatban):
(cv: ciklusváltozó; ké: kezdőérték; vé: végérték)



Elöltesztelési ciklus:

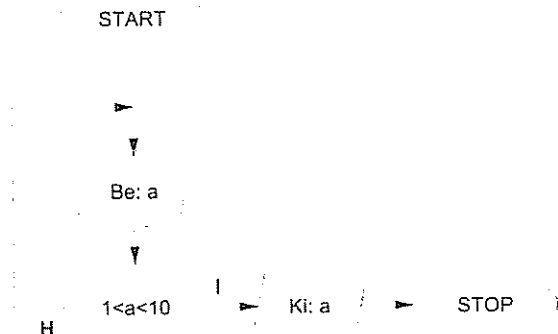


Hátultesztelési ciklus:



Végül nézzünk egy példát!

67. példa Készítsük el az **50. példa** folyamatábráját! Emlékeztetőül a példa a következő volt: Kérjünk be egy számot 1 és 10 között, majd írassuk ki a képernyő közepére!



3.3.2. Mondatszerű leírás

Az algoritmust itt egymást követő mondatokkal, illetve mondatyszerűen leírt szerkezetekkel írhatjuk le. Nagy előnye, hogy bőbeszédűbb lehet, mint a folyamatábra, közelebb is áll a gondolkodásunkhoz. Még a hosszabb programok is viszonylag röviden leírhatók a segítségével. Emellett ezek a mondatyszerű szerkezetek hasonlítanak a magas szintű programozási nyelvekben megszokott szerkezetekhez, ezért a kódolás sokkal egyszerűbb. A megismert szerkezetek alkalmazása pedig alkalmassá teszi a strukturált programok tervének elkészítésére.

A könyv további részében a mondatyszerű leírást fogjuk alkalmazni az algoritmusok leírására, és általában is ezt tartjuk hosszabb lélegzetű programok leírására a legalkalmasabb eszköznek, ezért meg lehetőséget tesz a részletességgel ismertetjük.

A mondatyszerű leírásnál a következő szerkezeteket alkalmazhatjuk:

A program eleje és vége:

```

Program Programnév
... utasítások
Program vége
  
```

Input és output utasítások (adatbekérés és kiírás):

```

Be: változó [a bekért adatra vonatkozó esetleges megjegyzés, általában értékhatár]
Ki: kifejezés [a kiírás formájára vonatkozó megjegyzés]
  
```

Elágazás kétféle:

```

Ha feltétel akkor utasítások
különben utasítások
  
```

Elágazás esetszétválasztással:

```

Elágazás
feltétel1 esetén: utasítások
feltétel2 esetén: utasítások
...
különben: utasítások
Elágazás vége
  
```

Számlálós ciklus:

```

Ciklus cv:=ké-től vé-ig
ciklusmag utasításai
Ciklus vége
  
```

Elöltesztelési ciklus:

```

Ciklus, amíg feltétel
ciklusmag utasításai
Ciklus vége
  
```

Hátultesztelési ciklus:

```

Ciklus
ciklusmag utasításai
mindaddig, amíg feltétel
Ciklus vége
  
```

Eljárás vagy függvény:

```

Eljárás Eljárásnév
... az eljárás utasításai
Eljárás vége
  
```

Megjegyzés: ezek a szavak csak ajánlások, ezektől el lehet térni (pl. **Be** helyett **Beolvas** vagy **Olvas** is lehetne), de a struktúrákat meg kell tartani.

Végül lássuk az előző fejezetben folyamatábrával megadott program algoritmusát mondatyszerű leírással:

68. példa

```

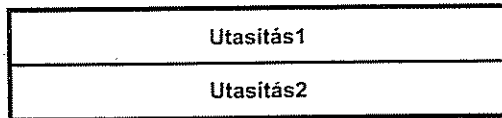
Program EgyjegyűSzám
Be: a [1 < a < 10]
Ki: a [középre]
Program vége
  
```

A bekérés leírásánál csak jeleztük az elfogadott értékhatárt (ezt persze megtehettük volna a folyamatábránál is), a bekérést végrehajtó ciklust a kódoláskor készítjük el (ez tulajdonképpen rutinleíradat). Megadjuk most a program bővebb algoritmusát a ciklussal együtt:

Program EgyjegűSzám
 Ciklus
 Be: a
 mindaddig, amíg $1 < a < 10$
 Ciklus vége
 Ki: a [középre]
 Program vége

3.3.3. Struktogram

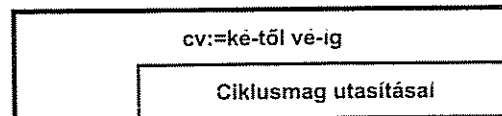
A struktogram egy kísérlet arra, hogy egyesítsük a folyamatábra képszerűségét és áttekinthetőségét a mondatszerű leírás egyértelműbb és bővebb leírásaival. A struktogram is alapvetően ábra jellegű ábrázolás, de nagyobb teret enged a leírásoknak. A programot egy téglalapba írjuk, amelyben az egyes utasítások kisebb téglalapokba kerülnek. A különböző szerkezetek másféle téglalapokba kerülnek a következőképpen:



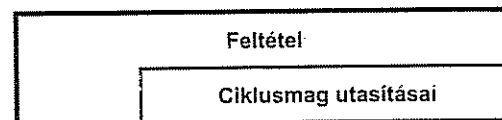
Elágazás:



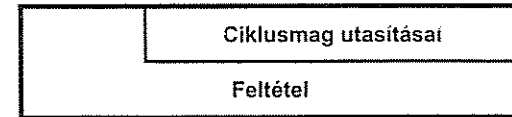
Számlálós ciklus:



Elöltesztelés ciklus:

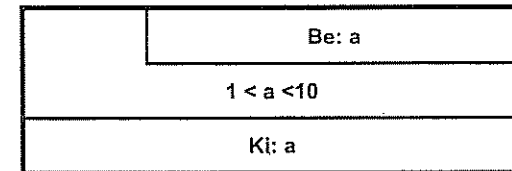


Hátultesztelés ciklus:



Végezetül lássuk, hogyan fest struktogrammal a már ismert példa!

69. példa



4. Függvények

Az eddig használt kifejezéseink eléggé egyszerűek voltak, és függvényeket talán nem is alkalmaztunk. Bonyolultabb számítások vagy átalakítások azonban már nem nélkülözhetik a függvényeket. A Turbo Pascal elég széles skáláját biztosítja a beépített függvényeknek, ezek közül a leggyakoribb matematikai függvényeket már felsoroltuk a változók ismertetésénél, most pedig megismerkedünk még néhányval (persze a teljesség igénye nélkül).

Az egyes függvényeket és eljárásokat a pontos paraméterlistával együtt fogjuk közölni, megadjuk, hogy mit csinál, majd egy példán keresztül bemutatjuk a működését.

Megjegyzés: A fejezetben általában a példák nem tartalmazzák a teljes programot, csak az adott feladatnak megfelelő eljárást vagy függvényt. Így többnyire nem szerepel bennük az adatbekérés, az adatellenőrzés, és az eredmény kiírása sem. Az egyes eljárások bemenő paraméterként fogják megkapni az adatokat, és változó paraméterekben (cím szerinti paraméterátadás) adják vissza az eredményt. A bekérésről és az eredmény kiírásáról külön kell gondoskodni, hiszen ki sem tudjuk próbálni ezeket az eljárásokat, ha nem írjuk meg a programot köréjük.

4.1. Matematikai függvények

4.1.1. Alapvető matematikai függvények

Function Sqr(x)

– négyzetre emelés

A függvény eredménye a paraméter négyzete lesz, azaz önmagával való szorzata (pl. $\text{Sqr}(5)$ értéke 25). A paraméter bármilyen egész vagy valós típusú lehet, a függvény eredménye a paraméterével megegyező típusú lesz.

Function Sqrt(x:real):real

– négyzetgyökvonás

A függvény eredménye a paraméter négyzetgyöke (pl. $\text{Sqrt}(25)$ értéke 5).

70. példa Készítsünk programot a másodfokú egyenlet megoldására! (Ez egy klasszikus példa, szinte egyetlen programozással foglalkozó könyvből sem hiányzik.)

Megoldás: A másodfokú egyenlet megoldóképlete:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Ehhez a már megismert matematikai műveleteken túl a négyzetre emelés (Sqr) és négyzetgyökvonás (Sqrt) függvényeket kell alkalmazni. Az eljárás bemenő paraméterei az együtthatók (a , b és c), kimenő paramétere a két gyök (x_1 és x_2). Ahhoz, hogy az eljárásban le tudjuk kezelni azt az esetet is, amikor a diszkrimináns (a gyök alatti kifejezés) negatív, szükségünk van még egy kimenő változóra, amely a gyökök számát tárolja (ennek lehetséges értékei a diszkrimináns (D) értékétől függnek: ha a D negatív, akkor nincsen megoldása az egyenletnek; ha D nulla, akkor egyetlen gyök van; és végül, ha D pozitív, akkor két különböző gyöke van az egyenletnek).

```

Procedure Masodfoku(a,b,c:real;var x1,x2:real;var
gyok:byte);
Var
  d : real;
Begin
  d:=Sqr(b)-4*a*c;
  If d<0
  then gyok:=0
  else
    begin
      x1:=(-b+Sqrt(d))/(2*a);
      If d=0
      then
        begin
          x2:=x1;
          gyok:=1;
        end
      else

```

```
begin
x2:=(-b-Sqrt(d))/(2*a);
gyok:=2;
end;
```

end;

End;

A kifejezésekben a négyzetre emelés függvény kiváltható az érték vagy kifejezés önmagával való szorzatával. Vagyis a példában az $\text{Sqr}(b)$ kifejezés ugyanazt jelenti, mint $b*b$.

Function Abs(x)

– abszolút érték

A függvény eredménye a paraméter abszolút értéke lesz (pl. $\text{Abs}(5)$ értéke 5 marad, míg $\text{Abs}(-5)$ értéke szintén 5 lesz).

A paraméter bármilyen egész vagy valós típusú lehet, a függvény eredménye a paraméterével megegyező típusú lesz.

Function Pi:real

– π

A függvény eredménye a π konstans értéke lesz, amely 3.1415926536. Ha használunk segédprocesszort, akkor a Pi értéke extended típusú lesz, és értéke pontosabban definiált: 3.141592653658979324.

71. példa Készítsünk függvényt, amely kiszámítja egy kör területét!

Megoldás: A kör területe a $2*r*\pi$ képlettel írható le, ahol r a kör sugarát jelenti. Mivel csak pozitív sugarú körök léteznek, ezért a bejövő paraméter (r) abszolút értékével számolunk majd, így a bemenő paraméter vizsgálata elmaradhat.

```
Function Kerulet(r:real):real;
Begin
Kerulet:=2*Abs(r)*Pi;
End;
```

4.1.2. Kerekítés

Function Int(x:real):real

– egészrész

A függvény eredménye a paraméter egészrésze lesz, amelyet úgy kapunk, hogy a törtrészt egyszerűen „levágjuk” róla (pl. $\text{Int}(5.34)$ értéke 5, és $\text{Int}(-5.34)$ értéke -5). Vigyázat, az eredmény nem egész, hanem valós típusú!

Function Trunc(x:real):longint

– egészrész

A függvény eredménye a paraméter egészrésze lesz, amelyet úgy kapunk, hogy a törtrészt egyszerűen „levágjuk” róla (pl. $\text{Trunc}(5.34)$ értéke 5, és $\text{Trunc}(-5.34)$ értéke -5). A függvény tehát ugyanazt az eredményt adja, mint az Int függvény, de az eredménye **egész** típusú lesz. Vigyázni kell, hogy a paraméter beleessen a longint értelmezési tartományába!

Function Round(x:real):longint

– kerekítés

A függvény eredménye a paraméter egészre kerekített értéke lesz, azaz a számhoz legközelebb eső egész szám (pl. $\text{Round}(5.34)$ értéke 5, és $\text{Round}(5.74)$ értéke 6). Vigyázni kell, hogy a paraméter beleessen a longint értelmezési tartományába!

Function Frac(x:real):real

– törtrész

A függvény eredménye a paraméter törtrésze lesz, amelyet úgy kapunk, hogy az egészrészt egyszerűen le hagyjuk (pl. $\text{Int}(5.34)$ értéke 0.34, és $\text{Int}(-5.34)$ értéke szintén 0.34). Az egészrész és a törtrész összege maga a szám: $x = \text{Int}(x) + \text{Frac}(x)$.

72. példa Készítsünk függvényt, amely a paraméter értékét két tizedes jegy pontosságra kerekíti!

Megoldás: Bármely számot kerekíthetünk két tizedes jegyre, ha a számot szorozzuk százszal, kerekítjük, majd visszaoosztjuk 100-zal ...

```
Function Kerekit2(x:real):real;
Begin
Kerekit2:=Round(x*100)/100;
End;
```

4.1.3. Véletlenszám generálása

Szimulációs programoknál, játékoknál, és még sok egyéb programnál jó hasznát vehetjük véletlenszerű számoknak. A Turbo Pascal – más programozási nyelvekhez hasonlóan – egy nulla és egy közé eső valós típusú számot állít elő a Random függvény segítségével.

Function Random:real

A függvény értéke nulla lehet, de 1 nem. Formálisan:

$0 \leq \text{Random} < 1$

Amennyiben két tetszőleges szám között kell véletlenszámot előállítani, akkor a függvény értékével el kell végezni néhány egyszerű műveletet.

A függvénynek van egy másik alakja is, amely egy nulla és tetszőleges word típusú szám között állít elő word típusú véletlenszámot:

Function Random(x:word):word

A függvény az előbbiekhez hasonlóan egy 0 és x közötti számot készít úgy, hogy értéke 0 lehet, de x nem. Formálisan:

$0 \leq \text{Random}(x) < x$

A leggyakrabban egy 1 és „valami” közötti egész számot kell „sorsolnunk” (pl. lottószámok esetén 1-90, kockánál 1-6, stb.). Ezekben az esetekben jön jól a függvénynek ez a második alakja, bár az eredeti formához egyet hozzá kell még adnunk.

Igy a $\text{Random}(90)+1$ függvény egy 1 és 90 közötti számot állít elő véletlenszerűen.

Ez a fajta számítógépen előállított véletlenszerű szám természetesen nem valódi véletlenszám, hiszen a gép nem tud egy kalapból húzni, vagy urnából sorsolni. Ezek ún. álvéletlen számok, vagy pszeudo-véletlen számok. Ez annyit jelent, hogy a gép egy matematikai kifejezés segítségével számítja az értékét, mindig az előző szám alapján. Ennek az a következménye, hogy a programot akárhányszor is futtatjuk le, mindig ugyanazt az öt lottószámot kapnánk. Ezen úgy lehet segíteni, hogy a véletlenszám-generátor kezdőértékét véletlenszerűen állítjuk be. Erre szolgál a Randomize eljárás.

Procedure Randomize;

Minden olyan programban, amelyik véletlenszámokkal dolgozik, meg kell hívni a Randomize eljárást (hacsak nem áll szándékunkban a „véletlent” megismételni).

73. példa Rendezzünk lottósorsolást! Írassunk ki a képernyőre öt véletlenszerű számot 1 és 90 között!

Megoldás:

```
Program LOTTO;
Uses Crt;
Var
    a,i    word;
Begin
    ClrScr;
    Randomize;
    For i:=1 to 5 do
        begin
            a:=Random(90)+1;
            WriteLn(a);
        end;
    End.
```

(A könyv végén szereplő mintafeladatban találkozunk még véletlenszámokkal.)

Feladatok:

157. Készítsen függvényt, amelyik két – paraméterben kapott – szám közötti véletlenszámot generál!

4.1.4. Trigonometriai függvények

Function Sin(x:real):real

– szinusz

A függvény eredménye a paraméter szinusza lesz. A paramétert radiánban kell megadni!

Function Cos(x:real):real

– koszinusz

A függvény eredménye a paraméter koszinusza lesz. A paramétert radiánban kell megadni!

Function ArcTan(x:real):real

– arkusz tangens

A függvény eredménye a paraméter arkusz tangense lesz, vagyis azt kapjuk meg, hogy melyik az a szög, amelyiknek a tangense a paraméter. Az eredményt radiánban kapjuk!

74. példa Készítsünk függvényt, amely fokot radiánba számít át!

Megoldás: A fok és radián közötti összefüggés alapján a függvény könnyen elkészíthető.

```
Function Rad(x:real):real;
Begin
    Rad:=x*Pi/180;
End;
```

75. példa A beépített trigonometriai függvények közül hiányzik a tangens. Készítsük ell

Megoldás: A tangens függvény a $\sin(x)/\cos(x)$ képlettel megadható. A függvény hívásakor ügyelni kell rá, hogy $\cos(x)$ ne legyen nulla, mert a program futási hibával le fog állni! (Esetleg a függvényben is lehet kezelni a problémát, de ezt már az olvasóra bízom.)

```
Function Tan(x:real):real;
Begin
  Tan:=Sin(x)/Cos(x);
End;
```

4.1.5. Hatvány, logaritmus

Function Ln(x:real):real

– természetes alapú logaritmus

A paraméter természetes alapú logaritmusát adja (a logaritmus alapja az e konstans, amelynek értéke kb. $e = 2,7182818285\dots$). A paraméter csak pozitív lehet!

Function Exp(x:real):real

– az Ln függvény inverze: az e -t hatványozza. A paraméter egy kitevő, a függvény értéke pedig az e^x .

76. példa A beépített függvények közül hiányzik a hatványozás. Készítsünk függvényt, amely tetszőleges számot tetszőleges hatványra emel!

Megoldás: A matematikai bizonyítás nem tartozik e könyv keretei közé, ha valaki ismeri a logaritmust, akkor ügyis megérti a képletet, ha nem, akkor pedig túl hosszasan kellene itt magyarázni. Ebben az esetben kipróbálhatjuk, és meggyőződünk róla, hogy a képlet helyes. A függvény az x^y értékét számítja ki. Az x értéke nem lehet negatív, erről a függvény hívásakor kell gondoskodni!

```
Function Hatvany(x,y:real):real;
Begin
  If x=0
  then Hatvany:=0
  else Hatvany:=Exp(y*Ln(x));
End;
```

77. példa Készítsünk tízes alapú logaritmus függvényt!

Megoldás: A különböző alapú logaritmusok közötti átváltás ismert képletét alkalmazzuk a függvényben.

```
Function Lg(x:real):real;
Begin
  Lg:=Ln(x)/Ln(10);
End;
```

Feladatok:

158. Készítsen függvényt **Kerekit** néven, amely kiírja a π értékét egészre kerekítve mindhárom beépített függvény segítségével! Adja meg a a törtrészt is!
159. Készítsen függvényt, amelyik egy valós számot tetszőleges tízedes jegyre tud kerekíteni! A függvény paraméterei a szám és a tízedesek száma legyenek!
160. Készítse el azt a függvényt, amelyik radiánból számol át fokba!
161. Készítse el a kotangens függvényt!
162. Készítsen függvényt, amely kettes alapú logaritmust számít!
163. Készítsen függvényt, amely tetszőleges alapú logaritmust számít (a logaritmus alapját is paraméterként kapja a függvény)!

4.2. Szövegek kezelése

A számok után most nézzük meg, hogy hogyan kezelhetjük a szövegeinket. A típusokról szóló részben már szó esett arról, hogy a szöveges változókat a **string** típusal azonosíthatjuk. Egy **string** típusú változó karakterek sorozatát tartalmazza, maximálisan 255 lehet belőlük. A műveleteknél megismertük az összefűzés (konkatenáció) műveletet, amely szövegek egymás után írását jelenti, és az összeadásjellel kellett elvégezni¹¹. Vigyázat, az összefűzött szöveg együttes hossza sem haladhatja meg a 255 karaktert, mert ilyenkor a 255. utáni karaktereket a Turbo Pascal egyszerűen elhagyja!

A szövegek kezelésére a Turbo Pascal több eljárást és függvényt is tartalmaz. Lássuk most ezek közül a hasznosabbakat!

4.2.1. A szöveg hossza

Function Length(s:string):integer – szöveg hossza

A paraméterben megadott szöveg hosszát adja meg, vagyis azt, hogy a szöveg hány karakterből áll.

¹¹ Ugyanerre a feladatra alkalmas a **Concat** függvény is.

78. példa Írjunk eljárást **KozepreKiir** néven, amely a paraméterben megadott szöveget pontosan a sor közepére írja ki!

Megoldás: Ahhoz, hogy a képernyő közepére írassunk, ismerünk kell a szöveg hosszát, ezt adja nekünk a **Length** függvény. A kiíratást pontosan annyiival kell a képernyő közepétől balra elkezdeni, amennyi a szóban forgó szöveg hosszának a fele.

```
Procedure KozepreKiir(s:string; sor:integer);
Var
    hossz : integer;
    kezd : integer;
Begin
    hossz:=Length(s);
    kezd:=40-(hossz div 2);
    GotoXY(kezd, sor);
    Write(s);
End;
```

A programban a **hossz** nevű változóba tesszük el a szöveg hosszát, amelyet a **Length** függvény ad meg. A **kezd** változó tartalmazza annak azt a helyet (oszlop), amelyen a kiíratást el kell kezdeni. Ebben a képernyő közepétől (40) visszaszámoljuk a szöveg hosszának a felét. Az osztást az ún. egészosztással végeztük el (**div**), amelynek az eredménye a helyes eredmény egészrésze. Erre azért volt szükség, mert ha a szöveg páratlan számú karaktert tartalmaz, akkor a fele nem lenne egész szám.

4.2.2. Szöveg kezelése karakterenként

Egy **string** típusú változót tekinthetjük úgy is, mint egy karakterekből álló tömböt. Így az **s** szöveg 3. karakterét az **s[3]** hivatkozással adhatjuk meg.

79. példa Készítsünk eljárást, amely egy szöveget kiírat a képernyő közepére karakterenként, minden karakter után egy kis szünetet tartva!

Megoldás: A karakterenkénti kiíratás hatásos eszköz lehet a kiíratások színesítésében. A megoldásához egy ciklus kell, amely a szövegen végigmegy. A várakoztatásról a már ismert **Delay** utasítás segítségével gondoskodhatunk.

```
Procedure KarakterenkentKiir(s:string);
Var
    i : integer;
    hossz : integer;
Begin
    ClrScr;
    hossz:=Length(s);
    GotoXY(40-hossz div 2, 12);
    For i:=1 to hossz do
        begin
            Write(s[i]);
            Delay(500);
        end;
    End;
```

80. példa Készítsünk függvényt, amely egy szövegben megszámolja a szóközöket!

Megoldás: A szövegen végig kell menni karakterenként, és egy **db** nevű változóban számolni a szóközöket.

```
Function SzokozSzam(s:string):integer;
Var
    i, db : integer;
Begin
    db:=0;
    For i:=1 to Length(s) do
        If s[i]=' ' then db:=db+1;
    SzokozSzam:=db;
    End;
```

81. példa Készítsünk függvényt, amely egy szövegben megszámolja a kisbetűket!

Megoldás: A megoldás majdnem ugyanaz, mint az előző példában, de most egy tartományt kell tesztelni. A szövegekre ugyanúgy alkalmazhatók a relációs jelek, mint a számokra, mégpedig az ASCII-táblában elfoglalt helyük szerint.

```
Function KisbetuSzam(s:string):integer;
Var
    i, db : integer;
Begin
    db:=0;
    For i:=1 to Length(s) do
        If (s[i]>='a') and (s[i]<='z') then db:=db+1;
    KisbetuSzam:=db;
    End;
```


4.2.3. Keresés a szövegben

Function Pos(s1,s:string):byte

– keresés a szövegben

A függvény az *s* szövegben keresi az *s1* szövegrészt. Visszatérési értéke az első olyan pozíció, ahol a szövegben megtalálta a keresett szöveget. Amennyiben nem talált ilyet, akkor a visszatérési érték 0 (nulla).

82. példa Készítsünk függvényt, amely megadja, hogy egy adott szövegben van-e szóköz!

Megoldás: A függvény logikai értékkel tér vissza; ha van szóköz a szövegben, akkor visszatérési értéke TRUE, egyébként FALSE.

```
Function VanSzokoz(s:string):boolean;
Begin
If Pos(' ',s)>0
then VanSzokoz:=true
else VanSzokoz:=false;
End;
```

Az elágazás helyett alkalmazhattuk volna a rövidebb, de nehezebben érthető `VanSzokoz:=(Pos(' ',s)>0)` formulát is. Azt jelenti, hogy a `VanSzokoz` felveszi a `Pos(' ',s)>0` logikai kifejezés értékét.

4.2.4. A szöveg alakítása

Sokszor előfordul, hogy a szövegünknek csak egy részére van szükségünk az adott pillanatban, vagy pedig a meglevő szöveget meg kellene változtatni. A következő eljárások segítségével „manipulálhatjuk” a szövegeinket:

Function Copy(s:string; kezd,hossz:integer):string

– szöveg egy része

A függvény eredménye az a szöveg, amely az *s* szöveg kezd pozíciójától kezdve *hossz* darab karaktert tartalmaz. Pl.: `Copy('baleset',3,3)` eredménye a 'les' szöveg.

83. példa Készítsünk eljárást, amely egy paraméterként kapott nevet vezetéknevre és keresztnévre bont!

Megoldás: Az eljárás változó paraméterben adja vissza a két szöveget. Először megkeressük, hogy a szöveg hányadik helyén kezdődik a keresztnév (keressük a szóközt), majd a nevet részekre bontjuk.

```
Procedure NevFelbont(nev:string;var
vnev,knev:string);
Var
hol:integer;
Begin
hol:=Pos(' ',nev);
vnev:=Copy(nev,1,hol-1);
knev:=Copy(nev,hol+1,Length(nev)-hol);
End;
```

Az *hol* változó tartalmazza a névben a szóköz pozícióját, ebből adjuk meg a keresztnévet és a vezetéknevet. Próbáljuk ki a programot úgy, hogy a beadott névben nincs szóköz (pl. Anonymus), és figyeljük meg, hogy ezt a programcskánk vezeté- vagy keresztnévnek veszi-e!

Procedure Insert(s1:string;var s:string;kezd:integer)

– beszúrás szövegbe

Az eljárás beszúrja az *s* szövegbe, annak *kezd* pozíciójától kezdve az *s1* szöveget. Ha a *kezd* értéke nagyobb, mint az *s* hossza, akkor az eljárás összefűzést jelent. Pl.: `Insert('Vitéz ','Csokonai Mihály',10)` eredményeként az *s* felveszi a 'Csokonai Vitéz Mihály' értéket.

Procedure Delete(var s:string;kezd,hossz:integer)

– törlés szövegből

Az eljárás töröl az *s* szövegből, annak *kezd* pozíciójától kezdve *hossz* darab karaktert. Pl.: `Delete('Csokonai Vitéz Mihály',10,6)` eredményeként az *s* felveszi a 'Csokonai Mihály' értéket.

84. példa Készítsünk eljárást, amely egy „ABC123” formában begépelte rendszámot „ABC-123” formátumúra alakít!

Megoldás: Csupán be kell szűrni egy kötőjelet a rendszám harmadik karaktere után.

```
Procedure Rendszam(var rsz:string);
Begin
Insert('-',rsz,4);
End;
```

Function UpCase(c:char):char

– karakter nagybetűvé alakítása

A függvény a paraméterben megadott karaktert – amennyiben az egy kisbetű – átalakítja a neki megfelelő nagybetűre. Ha a karakter nem egy kisbetű, akkor a függvény hatástalan. Vigyázat, csak az angol ABC betűire működik helyesen!

85. példa Egy programban jelszót szeretnénk alkalmazni. A jelszó legyen a „PASCAL” szó. A megoldás során azonban azzal a problémával kell szembenéznünk, hogy a jelszót a felhasználó szinte akárhogyan is begépelheti: pl. „Pascal” vagy „pascal” formában is. Ezeket is jó lenne elfogadni, mint helyes jelszót. Megoldást jelenthet a problémára, ha a jelszót nagybetűs alakban tároljuk el, és a begépelte jelszót nagybetűssé alakítva hasonlítjuk össze az eredetivel. Készítsünk eljárást, amely egy jelszót nagybetűssé alakít!

Megoldás: Az eljárás változó paraméterként kapja meg a jelszót. Karakterenként végigmegy a szövegen, és minden egyes betűt átalakít nagybetűre.

```
Procedure Nagybetu (var pw:string);
Var i:integer;
Begin
  For i:=1 to Length(pw) do
    pw[i]:=UpCase(pw[i]);
  End;
```

A *pw* változó tartalmazza a begépelte jelszót (*password=jelszó*).

Feladatok:

164. Készítsen függvényt, amely megfordít egy tetszőleges – paraméterben kapott – szó betűit, és a fordított szöveget adja eredményül!
165. Készítsen egy eljárást, amely egy szót ír ki a képernyő 20. oszlopába függőlegesen!
166. Készítsen eljárást, amelyik egy paraméterben kapott mondatot szavanként ír ki egymás alá!
167. Készítsen egy függvényt, amelyik két szót kap paraméterként, és a feladata az, hogy megszámolja, hogy a két szóban hány helyen vannak azonos karakterek (pl. alma és akar esetén csak a kezdőbetű egyezik, tehát 1 lesz a függvény értéke!).
168. A diákok év-vége naponta felírják a táblára a VAKÁCIÓ szó egy újabb betűjét. Készítsen eljárást, amelyik sorban, egymás alá felírja a képernyőre az Ó, IO, CIÓ, ... , VAKÁCIÓ szavakat! Minden újabb kiírás egy billentyű lenyomására történjen!
169. Készítsen függvényt, amely egy szöveg középső karakterét adja vissza!
170. Készítsen eljárást, amelyik egy szöveget úgy alakít át, hogy minden harmadik karaktert kicseréli a @ karakterre!
171. Készítsen eljárást, amelyik a paraméterben megadott teljes név alapján kiírja a monogramot!

4.3. Sorszámozott típusok függvényei és eljárásai

A sorszámozott típusok (egészek, karakter, felsorolás, stb.) speciális jellegükből adódóan használhatunk néhány olyan eljárást és függvényt, amely kihasználja a sorszámozott típus azon tulajdonságát, hogy az elemek fix sorszámmal rendelkeznek, azaz megadható, hogy melyik elemnek mi a sorszáma a típuson belül. Ezeket a függvényeket példa nélkül, csak egy-egy rövid megjegyzéssel soroljuk fel az alábbiakban:

Procedure Inc (n; {m:longint})

– paraméter értékének növelése

Az eljárás az *n* sorszámozott típusú változó értékét növeli meg, mégpedig az *m* változó értékével. Ha a második paramétert elhagyjuk, akkor az eljárás az *n* értékét 1-gyel növeli. Pl.: Inc (n) megfelel az *n:=n+1* utasításnak. Inc (n, 5) megfelel az *n:=n+5* utasításnak. stb. Az eljárás egy kicsit gyorsabb, mint az értékadás, mert csak egyszer kell a memóriából a változót kiolvasni, míg értékadásnál a bal és jobb oldal is szerepel, ezért kétszer fordul a program a memóriához.

Procedure Dec (n; {m:longint})

– paraméter értékének csökkentése

Az eljárás az *n* sorszámozott típusú változó értékét csökkenti, mégpedig az *m* változó értékével. Ha a második paramétert elhagyjuk, akkor az eljárás az *n* értékét 1-gyel csökkenti. Pl.: Dec (n) megfelel az *n:=n-1* utasításnak. Dec (n, 5) megfelel az *n:=n-5* utasításnak. stb. Az eljárás az Inc eljárás ellentettje, minden egyéb igaz itt is, amit ott említettünk.

Function Ord (n):longint

– a paraméter sorszáma a típuson belül

Az eljárás az *n* sorszámozott típusú változó típuson belüli sorszámát adja meg. A sorszámzás minden ilyen típus esetén 0-val kezdődik. A függvény karakter típusú paraméter esetén annak ASCII-kódjával tér vissza!

Function Pred (n)

– a típus előző értékét adja

Function Succ (n)

– a típus következő értékét adja

A két függvény sorszámozott típusok esetén az *n* változó előtti, illetve utáni értéket adja vissza (azaz azt az értéket, amelynek a típuson belüli sorszáma eggyel kisebb, illetve nagyobb a paraméter sorszámánál).

Pl.: Pred ('b') = 'a' és Succ ('b') = 'c'.

Function Odd(n):boolean

– a paraméter párosságát adja

A paraméter bármilyen egész típusú változó lehet, az eredmény TRUE (IGAZ), ha *n* értéke páratlan, FALSE (HAMIS), ha páros.

4.4. Típuskonverziós függvények

Azok a függvények tartoznak ide, amelyek egy adott típusú adatot más típusúvá alakítanak át¹²:

Procedure Str(n;var s:string)

– számból szöveg

Az eljárás az *n* numerikus változó értékét **szöveggé** konvertálja az *s* változóba. Az *n* tetszőleges numerikus típus lehet (egész vagy valós), és a Write-nál megismert módon adható meg – valós típus esetén – a tizedesek száma. A későbbiekben látni fogjuk, hogy a grafikus képernyőre csak szövegek írhatók ki, tehát ha számot kell kiírni, akkor azt előbb szöveggé kell konvertálni.

Procedure Val(s:string;var szam;var kod:integer)

– szövegből szám

Az eljárás az *s* szöveget számmá konvertálja, és a *szam* változóban helyezi el. Persze a konverzió csak akkor megy végbe, ha az *s* string valamilyen numerikus adatot tartalmaz: pontosabban, ha a szövegben leírt szám szigorúan megfelel a numerikus konstansok szintaktikai szabályainak (azaz nem tartalmazhat szóközt, betűt, stb.). Az átalakítás sikeréről a harmadik változó, a *kod* értéke tájékoztat: ha értéke 0 (nulla), akkor az átalakítás sikeres volt.

A Val eljárás jól használható numerikus adatok bekérésénél. Ugyanis itt problémát okozhat az, hogy a felhasználó nem mindig azt írja be egy kérdésre, ami logikusan elvárható lenne tőle. Például, hiába mondom én, hogy „gépelj be egy számot 1 és 10 között”, ez még korántsem garancia arra, hogy tényleg azt is kapok. Ha számot kapok ugyan, de nem esik bele a megadott értelmezési tartományba, akkor legfeljebb újra bekérem a számot (erre láttunk már példát a korábbiakban). A probléma ott kezdődik, ha nem számot kapunk (véletlenül vagy szándékosan), mert ilyenkor a programunk futási hibával fog leállni,

¹² A már megismert függvények között is találunk olyanokat, amelyek típus-átalakítást végeznek. Pl. a Round és a Trunc függvények, amelyek valós értéket konvertálnak egész típusúvá (kerekítés), vagy az Ord, amely sorszámozott típusú adatot alakít longint típusúvá.

ami elég „cikis”. A megoldás az lehet, ha mindenképpen szöveget kérünk be (persze erről a gép előtt ülő felhasználó mit sem tud), majd számmá konvertáljuk azt.

86. példa Kérünk be egy 1 és 10 közötti egész számot a billentyűzetről!

Megoldás: Egy univerzális adatbeolvasó eljárást készítünk a Val eljárás segítségével. Célszerű ezt a kissé hosszadalmas, ámde megbízható procedúrát minden adatbekérésnél végrehajtani, mert így a programunk – legalábbis az adatbekérések terén – „bolondbiztossá” válik¹³.

Program Adatbekeres;

Uses Crt;

Var

a : real;

s : string;

k : integer;

jo : boolean;

Begin

Clrscr;

Repeat

jo:=true;

Write('Kérek egy számot 1 és 10 között!');

Readln(s);

Val(s,a,k);

If k<>0

then begin

Writein('Ez nem szám! Próbáld újra!');

jo:=false;

end

else

If (a<1) or (a>10)

then begin

Writein('Ez nem 1 és 10 között van!');

jo:=false;

end

else

If a<>Trunc(a)

then begin

Writein('Ez nem egész szám!');

jo:=false;

end

Until jo;

{itt már biztosak lehetünk abban, hogy a bekért adat valóban 1 és 10 közötti egész szám}

End.

¹³ Egy programnak soha sem szabad futási hibával leállnia! Minden elképzelhető és elképzelhetetlen problémára fel kell készülnünk, és lehetőség szerint meg kell oldanunk őket. Az így megírt programot nevezzük „bolondbiztos”-nak, mert egy szándékosan rosszindulatú, vagy bolond felhasználó sem képes „kiakasztani”

A programban a **jo** változó mutatja meg, hogy jó-e a begépelte adat. Először megvizsgáljuk, hogy szám-e egyáltalán. Ha nem akkor hibaüzenetet adunk, és a **jo** értékét **hamisra** állítjuk, hogy a ciklus ne érhesen véget. Ha a begépelte adat szám, akkor megvizsgáljuk, hogy megfelelő-e a megadott értéktartománynak. Ha nem, akkor megint csak hibaüzenetet adunk, és a **jo** értékét **hamisra** változtatjuk. Ha igen, akkor már csak azt kell megállapítani, hogy a szám egész-e. Ezt úgy tehetjük meg a legegyszerűbben, hogy a számot összehasonlítjuk a saját egészrészével.

Function Chr(n):char

– egészből karakter

A függvény azt a karaktert adja vissza értéként, amelynek az **n** egész típusú változó az ASCII-kódja. Pl.: Chr(65)='a'. Amennyiben a paraméter kiesik a 0..255 értelmezési tartományból, akkor a függvény először az **n** 256-os modulusát (osztási maradékát) képz, és ezzel számol.

Pl.: Chr(321)=Chr(65)='a'.

Feladatok:

172. Készítsen függvényt, amely egy tetszőleges 10 karakteres szöveget generál – karakterenként véletlenszerűen! Vigyázzon, hogy csak betűk legyenek benne!

5. Fájlok kezelése

Az eddigi programokban az adataink csak addig álltak a rendelkezésünkre, amíg a programot használtuk. Az adatainkat azonban sokszor el kellene tenni későbbi felhasználásra. Ekkor ezek háttértárolóra (floppy-ra, winchesterre) kerülnek, fájlok (állományok) formájában. Az alábbiakban áttekintjük a fájlok kezelésének legfontosabb tudnivalóit.

A fájlok alapvetően szekvenciális adatszerkezetek, azaz a benne foglalt adatokhoz csak sorban, egymás után férhetünk hozzá. A Turbo Pascal lehetőséget biztosít a fájlok direkte kezelésére is, vagyis arra, hogy tetszőleges rekord elérhető legyen.

A fájlokra a Turbo Pascal-ban egy fájl-változó segítségével hivatkozhatunk, és első lépésben a fájl-változót kell összerendelnünk a háttértáron lévő fizikai állománnyal. Ezután nyithatjuk meg az állományt írásra vagy olvasásra, elvégezhetjük a megfelelő tevékenységeket, majd a fájlt be kell zárni (ezzel az összerendelés nem vesz el). Használata előtt a fájl-változót természetesen deklarálni kell, ennek módját korábban már bemutattam.

A fájlok leggyakrabban azonos típusú adatok sorozatát tartalmazzák, leggyakrabban rekordokat, de egyszerű típust is hozzárendelhetünk egy fájlhoz. Az ilyen típusú fájl típizált fájlnak nevezzük. Ekkor a deklaráció során meg kell adnunk a fájl típusát. Használhatunk olyan fájlokat is, amelyek típusa nem meghatározott, ekkor nem típizált fájlokról beszélünk (pl. ha egyetlen állományba szeretnénk menteni a programom több, de különböző adatát). Nem típizált fájlok esetén természetesen a programozó dolga, hogy az állományt pontosan kezelje.

Különleges fájl típus a szövegfájl. Ez olyan típizált fájl, amelyben szövegek vannak. Ezt a fájlt soronként kezelhetjük (a típizált fájl rekordonként), és csak szekvenciálisan férhetünk az adatokhoz. Ha egy olyan adatra van szükség, amelyik előbb van, mint a mutató, akkor a fájlt le kell zárni, újra megnyitni, és „elmenni” addig, amíg el nem érjük a kívánt adatot.

5.1. Deklaráció és hozzárendelés

A deklarációval már a korábbiakban megismerkedtünk, most mégis elevenítsük fel röviden:

```
Type
  tipfajl = file of adattipus;
  fajl = file;
  szovegfajl = text;
Var
  f1 : tipfile;
  f2 : fajl;
  f3 : szovegfile;
```

Az **f1** változó egy tipizált fájlt fog reprezentálni, az **f2** egy nem tipizált fájlt, míg **f3** szövegfájl lesz.

A deklaráció után a fájl-változót (logikai fájlnev) hozzá kell rendelni egy háttértárolón lévő fizikai fájlhoz az **Assign** eljárással.

```
Procedure Assign(var f;nev:string);
```

ahol **f** a fájl-változó, a **nev** pedig a fizikai fájlnev. Például érvényes hozzárendelések a következők:

```
Assign(f1,'alma.rec');
Assign(f2,'c:\kutya\csont.dat');
Assign(f3,'a:\nevek.txt');
```

5.2. Fájl megnyitása

Fájl megnyitására két lehetőségünk van: ha létező állományt akarunk megnyitni, akkor a **Reset** eljárást kell használni; ha pedig úgy akarunk megnyitni egy állományt, hogy egyben létre is hozzuk azt (új fájl), ezt a **Rewrite** eljárással tehetjük meg. Használatuk:

```
Procedure Reset(f); illetve
Procedure Rewrite(f);
```

Mindkét eljárás alkalmazása előtt meg kell történnie a fájl-változó hozzárendelésének a fizikai fájlhoz! Mindkét esetben egyaránt olvashatunk ki adatot a fájlból és írhatunk is bele. Ha egy létező állományt a **Rewrite**-tal nyitunk meg, az állomány felülíródik, azaz a régi tartalma elvész!

Egy időben több fájl is megnyitható (ezek maximális száma a **CONFIG.SYS** állományban van beállítva).

5.3. Fájl bezárása

Miután egy megnyitott állománnyal befejeztük a munkát, a fájlt be kell zárni a **Close** eljárással:

```
Procedure Close(f);
```

A bezárás nem jelenti a hozzárendelés megszűnését; későbbi megnyitás esetén nem kell az **Assign** eljárást ismételten használni.

5.4. A rekordmutató pozícionálása¹⁴

Már korábban említettük, hogy egy tipizált fájl rekordok (vagy egyszerű adatok) sorozatából áll. Azt, hogy melyik adattal végezhetjük a műveleteket, azt az ún. rekordmutató adja meg. Van tehát mindig egy aktuális rekord, erre mutat a rekordmutató. Mielőtt tehát adatot olvasnánk ki a fájlból (vagy írunk bele), pozícionálni kell a rekordmutatót arra a rekordra, amellyel szeretnénk foglalkozni. A rekordmutató pozícionálását a **Seek** eljárással végezhetjük el.

```
Procedure Seek(var f; n:longint);
```

Itt **f** a fájl-változó, az **n** pedig a rekord sorszáma. A Turbo Pascal a rekordokat nullától számozza.

Bármilyen műveletet is végezzünk az aktuális rekordon, a művelet végeztével a mutató automatikusan a következő rekordra mutat.

Az aktuális rekord sorszámát (vagyis a rekordmutató helyét) le is kérdezhetjük a **FilePos** függvénnyel:

```
Function FilePos(var f):longint;
```

Az állományban lévő rekordok számát pedig a **FileSize** függvény adja meg:

```
Function FileSize(var f):longint;
```

Vigyázat! A rekordok számozása nullától kezdődik, tehát az első rekord sorszáma 0, az utolsó pedig eggyel kisebb, mint a fájl mérete (vagyis a **FileSize** függvény által visszaadott érték).

¹⁴ A fejezet csak tipizált fájlokra vonatkozik, hiszen ez az utasítás teszi lehetővé a direkt hozzáférést, amelyre a szövegfájlok esetében nincs lehetőség.

5.5. Adat beolvasása fájlból

A rekordmutató által mutatott rekord beolvasása a már jól ismert Read eljárás segítségével történhet:

```
Procedure Read(var f; var a);
```

Itt *f* a fájl-változó, a második paraméter (*a*) pedig az a változó, amelybe az adat a beolvasás után kerül. Vigyázni kell arra, hogy az *a* változó típusa egyezzen meg a fájl komponenseinek típusával! Megadható több paraméter is, ekkor az egyes változókba szép sorjában olvassa be a Read az adatokat. (Természetesen a többi változó típusa is azonos kell hogy legyen a fájl komponenseinek típusával.)

Gyakori feladat, hogy a fájl összes elemét beolvassuk egy tömbbe. Ekkor egy ciklust kell szerveznünk, ahol minden lépésben egyetlen adatot olvasunk be. A ciklus lehet számlálós ciklus – ekkor 0-tól kell mennie a fájl méretéig, amelyet a *FileSize* függvénnyel meghatározhatunk; illetve alkalmazhatunk hátultesztelős ciklust is, amely addig fut, amíg a fájlban nincsen vége. Ez az *Eof* függvény segítségével határozható meg.

```
Function Eof(var f):boolean;
```

A függvény értéke TRUE (IGAZ), ha a fájl végén (vagyis az utolsó komponens után) áll a rekordmutató, különben hamis.

5.6. Adat írása fájlba

Az előzőek alapján már sejthető, hogy rekordot fájlba írni a Write eljárással lehet:

```
Procedure Write(var f; var a);
```

Minden, amit a Read eljárás kapcsán leírtunk, igaz itt is.

A fájl végére írni (azaz újabb rekordot a fájlba tenni) úgy lehet, hogy előtte a rekordmutatót a fájl végére pozícionáljuk (az utolsó rekord után), majd írunk a fájlba.

5.7. A fájl törlése

A fájl – ha már nincs rá szükség – a merevlemezről célszerű eltávolítani (letörölni). Ezt az Erase eljárással tehetjük meg:

```
Procedure Erase(var f);
```

Az eljárás törli az *f* változóhoz hozzárendelt fizikai fájl a háttértárolóról. A törölni kívánt fájl nem lehet megnyitva!

Lehetőségünk van arra is, hogy bizonyos rekordokat töröljünk a fájlból, de csak a végén levőket.

```
Procedure Truncate(var f);
```

Az eljárás törli a fájlból az összes rekordot a rekordmutatótól kezdődően (utána a mutató a fájl végére fog mutatni).

5.8. Nem tipizált fájlok

A nem tipizált fájlok esetén a már megismert eljárások és függvények többsége ugyanúgy alkalmazható, mint általános esetben. Kicsit eltérő a működése a Reset és Rewrite eljárásoknak, és nem alkalmazható a Read és a Write. Ezek helyett a jóval hatékonyabb (gyorsabb) BlockRead és BlockWrite eljárások alkalmazhatók. Ez a nagy sebességekülönbség indokolja, hogy nagy mennyiségű adat esetén tipizált fájl alkalmazása helyett nem tipizált fájl alkalmazása sokkal célszerűbb, bár a programozótól nagyobb odafigyelést igényel.

```
Procedure Reset(var f:file;n:word);
```

```
Procedure Rewrite(var f:file; n:word);
```

Nem tipizált fájlok esetén az eljárások egy második paramétert is kapnak, amely a fájl rekordmérete bájtokban. Általában célszerű ezt 1-nek megadni, és beolvasáskor, illetve kiíráskor foglalkozni az egyes adatok méreteivel.

```
Procedure BlockRead(var f:file;var r;n:word[;v:word]);
```

```
Procedure BlockWrite(var f:file;var r;n:word[;v:word]);
```

*Az eljárások az *s* értékének megfelelő számú rekordot mozgatnak az állomány és a memória, pontosabban az *r* változó között. Az *r* tetszőleges változó lehet. A *v* változó opcionális, ha megadjuk, akkor a ténylegesen mozgatott rekordszámot kapjuk vissza benne. Ha *v=s*, akkor a teljes átvitel megtörtént, ha *v<s*, akkor a fájl végére értünk már. Amennyiben megnyitáskor 1 bájtos rekordméretet adtunk meg, akkor az eljárások*

harmadik paramétere (*s*) pontosan az *r* változó mérete kell hogy legyen (bájtokban). Egy változó bájtokban vett méretét a `SizeOf` függvénnyel kaphatjuk meg a legegyszerűbben.

Function `SizeOf(x):integer;`

A függvény egyetlen paramétere tetszőleges típusú változó lehet (sőt, akár egy típus azonosítója is), értéke a paraméter bájtokban mért helyfoglalása.

5.9. Szövegfájlok kezelése

A szövegfájlok általában szöveges adatokat tartalmaznak, de tartalmazhatnak numerikus adatokat is. Leggyakrabban egy sorban egy adat van, de tehetünk egy sorba többet is, ekkor szóközzel vagy tabulátorral kell elválasztanunk őket egymástól. Épp ezért a használatuk kis mértékben eltér a hagyományos fájlok kezelésétől.

A szövegfájlok esetén is a fentebb már ismertetett eljárásokat alkalmazzuk. Nem szabad azonban használni a `Seek`, `FileSize`, `FilePos` és `Truncate` eljárásokat, kis mértékben eltérő a `Read` és `Write` használata, valamint a `Reset`-tel való megnyitás csak olvasásra jogosít, írásra nem. Ezenkívül van még néhány olyan eljárás, amely csak szövegfájlokra alkalmazható, ezek az `Append`, `Eoln`, `SeekEoln`, `SeekEof`, `Readln` és `Writeln`.

5.9.1. Hozzárendelés, megnyitás és bezárás

A tipizált fájlokhoz képest annyi az eltérés, hogy a `Reset` használatával a fájlt csak olvashatóra nyitjuk meg, azaz nem tudunk bele adatot írni. Létező szövegfájl írásra való megnyitására az `Append` szolgál.

Procedure `Append(var f:text);`

Az eljárás megnyitja a szövegfájlt, és a rekordmutatót annak végére állítja (vagyis az utolsó sor után). Ezzel az állomány továbbírására van lehetőségünk.

Az `Assign`, `Rewrite` és `Close` eljárások ugyanúgy alkalmazhatók, mint a tipizált fájlok esetében.

5.9.2. Beolvasás és írás

Adatot fájlba írni továbbra is lehet a `Read` segítségével, de a `Readln` is használható.

Procedure `Read(var f:text;var a1[,a2,...]);`

Procedure `Readln(var f:text;var a1[,a2,...]);`

A két eljárás az *f* szövegfájlból olvas be adatokat. Az *a1*, *a2*, stb. változók vagy szövegesek, vagy numerikusak lehetnek. Szöveges változó esetén annyi karakter beolvasása történik meg, ahány karakter hosszúságúra deklaráltuk az adott változót. Numerikus adat beolvasása esetén az adatnak meg kell felelnie a numerikus konstansok szintaktikai szabályainak (azaz a karaktersorozat számjegyet, és a következő jeleket tartalmazhatja: .. -, +; E, \$).

A két eljárás között csupán annyi a különbség, hogy a `Readln` alkalmazásával a mutató a következő sor elejére ugrik, függetlenül attól, hogy az adott sorban van-e még adat, vagy sem.

Szövegfájlba írni a `Write` és `Writeln` eljárásokkal lehet. Használatuk és paraméterezésük megegyezik a `Read` és `Readln` eljárásokéval.

Hasonlóan a tipizált fájlok kezeléséhez, ha a teljes fájl tartalmát szeretnénk beolvasni, akkor bizony egy tesztelős ciklust kell szerveznünk, amely addig megy, amíg a fájlban nincs vége. A fájl végét az `Eof` függvény IGAZ értéke jelzi. Ez szövegfájloknál is alkalmazható, de használhatjuk a `SeekEof` függvényt is, amely csak abban különbözik az `Eof`-tól, hogy a fájl végének jelzésekor nem veszi figyelembe az esetleges szóközöket és tabulátorokat.

Function `SeekEof(var f:text):boolean;`

A szövegfájlok sajátosságaiából adódik (vagyis abból, hogy a fájlt szekvenciálisan, soronként kezeli), hogy szükségünk lehet a fájl végének jelzésén túl a sor végének jelzésére is. Erre az `Eoln` és `SeekEoln` függvényeket alkalmazhatjuk. Mindkettő a sor végén IGAZ értéket vesz fel, egyébként HAMIS, és az `Eof`-hoz hasonlóan a `SeekEoln` csak abban tér el az `Eoln`-tól, hogy a sor végén nem veszi figyelembe a szóközöket és tabulátorokat.

Function `Eoln(var f:text):boolean;`

Function `SeekEoln(var f:text):boolean;`

5.10. Fájl átnevezése

A `Rename` eljárás segítségével új nevet adhatunk a fizikai fájlbanak.

Procedure `Rename(var f;ujnev:string);`

Az első paraméter a fájl-változó, a második pedig az új nevet adja meg, esetleg útvonallal együtt. Az átnevezni kívánt fájl nem lehet megnyitva!

5.11. Könyvtárak kezelése

Szigorúan véve ez a fejezet nem tartozik szorosan ide, de feltétlenül említést érdemel. Az állományok a merevlemezen ún. könyvtárakban vannak, tehát a könyvtárak és az állományok kezelése szorosan összetartozik. Anélkül, hogy mélyebben belemennénk a DOS ismeretekbe, tudni kell, hogy a könyvtárak között mindig van egy, amelyikben tartózkodunk, és ezt aktuális könyvtárnak nevezzük. Az aktuális könyvtár nevét a `GetDir` eljárás adja meg. Könyvtárat válthatunk a `ChDir` eljárással, és lehetőségünk van új könyvtárat létrehozni (`MkDir`) és törölni is (`Rmdir`). Az eljárásokat a következőképpen kell használni:

```
Procedure GetDir(m:byte;var s:string)
```

Az eljárás az s szöveges változóba teszi az aktuális könyvtár nevét. Az m változóval azt a meghajtót adjuk meg, amelyiken kíváncsiak vagyunk az aktuális könyvtárra. Ha m értéke 1, akkor ez az A meghajtó, 2 esetén a B, 3 esetén a C, és így tovább.

```
Procedure ChDir(s:string)
```

```
Procedure MkDir(s:string)
```

```
Procedure Rmdir(s:string)
```

Mindhárom eljárásnál a paraméterben a könyvtár nevét adjuk meg. Az Rmdir eljárás csak üres könyvtár törlésére alkalmas!¹⁵

5.12. A fájlkezelés hibáinak elhárítása

A programok írása során az egyik legkellemetlenebb feladat a bemenet és kimenet kezelése. Ilyenek például az adatbekérés a billentyűzetről, és a fájlkezelés is. Ezekben az esetekben ugyanis sok olyan körülményt kell figyelembe venni, ami nem a programozón múlik, hanem a program majdanai használóján. Hiába írjuk meg azt a programot tökéletesen, ami egy állományból olvassa be az adatokat, ha valaki azt az állományt letörli a winchesterről. Ekkor pedig a program futási hibával leáll, ami megengedhetetlen. Ezeket a hibalehetőségeket a program készítése során figyelembe kell venni, és lehetőleg helyesen le is kell kezelni.

Már a korábbiakban említettük a fordítási direktívákat. Ezek a fordítóprogramnak szóló utasítások. A Turbo Pascal alapértelmezésben olyan kódot generál (fordít), ami minden I/O műveletet ellenőriz, de ezt az I fordítási direktíva szabályozza. Bekapcsolt állapotában végez el-

lenőrzést, kikapcsolt állapotában nem. Ha nincs ellenőrzés, akkor a program sem áll le futási hibával, hanem csak felfüggeszti a futását mindaddig, amíg meg nem hívjuk az `IOResult` függvényt.

```
Function IOResult:boolean
```

A függvény a fellépő I/O hiba kódját adja értékül. Ha nem volt hiba, akkor értéke 0. Használatát az alábbi példa mutatja be:

87. példa Nyissunk meg egy fájlt úgy, hogy ellenőrizzük a megnyitás sikerességét!

Megoldás: A megnyitás előtt kikapcsoljuk az ellenőrzést, utána visszakapcsoljuk, és azonnal meghívjuk az `IOResult` függvényt.

```
Program IOHiba;
```

```
Var
```

```
  f : file of integer;
```

```
Begin
```

```
  Assign(f,'adatok.dat');
```

```
  {$I-} Reset(f); {$I+}
```

```
  If IOResult<>0
```

```
    then begin
```

```
      {Hibakezelő utasítások!}
```

```
    end
```

```
  else begin
```

```
    {Sikeres megnyitás esetén végrehajtandó
```

```
    programrész!}
```

```
    Close(f);
```

```
  end;
```

```
End.
```

A hiba kezelésénél adunk egy hibaüzenetet. Ha a program folytatását nem az elágazás `else`-ágára helyezzük el, hanem az elágazás után, akkor vagy le kell állítani a programot (nem ajánlott), vagy létre kell hozni egy üres fájlt, amivel tovább tudunk dolgozni. Esetleg egy `case` utasítás segítségével a lehetséges hibákat egyesével is kezelhetjük.

Az `IOResult` függvény csak egyszer hívható meg egy I/O művelet kapcsán, mert meghívása után elveszíti az értékét. Ha a hiba kódjára később is szükségünk van, akkor egy változóban kell tárolnunk azt.

Feladatok:

173. Készítsen eljárást, amelyik egy tipizált fájlba helyez 100 darab, 1 és 1000 között véletlenszerűen generált számot! Készítsen egy másikat is, amelyik képes ezeket a számokat beolvasni és megjeleníteni a képernyőn!

174. Oldja meg az előző feladatot szövegfájl alkalmazásával!

¹⁵ Ezek a Turbo Pascal eljárások rendre megfelelnek a CD, MD, illetve RD DOS parancsoknak.

6. Grafika a Turbo Pascal-ban

A mai programoknál a helyes és biztos működésén kívül alapkövetelmény a szép tálalás is. A jól kinéző képernyők iránti igény a hagyományos karakteres képernyőt alkalmazva szinte lehetetlen feladatot elé állít minket. Elkerülhetetlennek látszik a grafika használata. Ebben a fejezetben a grafikus lehetőségekről lesz szó.

A Turbo Pascal a Graph unit-tal egy sokoldalú és könnyen kezelhető eszközt ad a kezünkbe a grafikus képernyő kezelésére. A Graph unit 79 eljárást és függvényt tartalmaz, de ezeken a lapokon természetesen nem ismertetjük az összeset; csak azokat, amelyekre a leginkább szükség lehet.

Mivel minden grafikus eljárás a Graph unitban van, használatuk előtt a unitot deklarálni kell a már megismert módon:

```
Uses Graph;
```

6.1. A grafikus képernyő felépítése

A grafikus képernyő alapvető paraméterei a felbontás és az alkalmazható színek száma. A grafikus képernyő képpontokból áll, és a felbontás azt jelenti, hogy vízszintesen, illetve függőlegesen hány képpont fér a képernyőre. Mind a felbontást, mind a színek számát a számítógépünkben lévő grafikus kártya típusa határozza meg. Ma már annyira elterjedtek a VGA vagy SVGA kártyák, hogy a többivel nem is foglalkozunk. A VGA kártya alapvetően 640*480 képpontot jelenít meg a képernyőn, és minimum 16 színt használ. Általában lehetőségünk van a gép használata során (elsősorban a Windows-ban) ennél nagyobb felbontást (800*600, 1024*768, stb.) és több színt (256, 65 ezer, stb.) használni, de a Turbo Pascal „alapfelszerelésben” nem kínál többet¹⁶. Ez általában elég is.

¹⁶ A Turbo Pascal kezeli még a CGA (320*200 felbontás, 4 szín), MCGA (640*480, mono), EGA (640*350, 16 szín), Hercules (720*340, mono), IBM8514 (1024*768, 256 szín) és PC3270 (720*350, mono) grafikus kártyákat is. Minden egyébhez különleges meghajtóprogramok (driver-ek) szükségesek.

Az előbbieken említett felbontási értékek maximális értékek, ha ennél kisebb felbontást használunk, akkor akár több képernyőt is tudunk kezelni egy időben (pl. két képernyő kezelésénél az egyik látszik a monitoron, míg a másikra rajzolunk, és a kész rajzot azonnal meg tudjuk mutatni, aktívva téve a második képernyőt).

Az egyes grafikus kártyákhoz tartozó kezelőprogram a Turbo Pascal-ban .BGI kiterjesztésű állományokban van. Grafikát használó program nem nélkülözheti ezeket az állományokat (VGA monitor esetén az EGAVGA.BGI állományra van szükség), sem a fejlesztői környezetben való futtatáskor, sem akkor, ha már mint önálló programot (.EXE állományt) futtatjuk.

Mivel a képernyő képpontokból áll, nekünk is a képpontokat kell kezelnünk. Ismerkedjünk meg tehát a VGA képernyőn alkalmazott grafikus koordináta-rendszerrel! A képernyő sorai és oszlopai egyaránt 0-tól kezdődően vannak megszámozva; a sorok balról jobbra, az oszlopok felülről lefelé. Minden képpontot egy (x,y) koordináta-pár ír le, ahol az első koordináta az oszlop, a második a sor. A bal felső sarok koordinátája tehát (0,0), a jobb alsó saroké (VGA képernyőn!) (639,479).

6.2. Inicializálás

Láttuk már, hogy amikor elkezdünk dolgozni a Turbo Pascal-ban, akkor a karakteres képernyőn vagyunk. Ahhoz, hogy a képernyőt képpontonként tudjuk kezelni, azaz a grafikus képernyőre kerüljünk, inicializálni kell azt, majd a munka végén lezárni. A grafikus képernyő inicializálását az InitGraph eljárás végzi el.

```
Procedure InitGraph(var gd, gm: integer; s: string);
```

Az első paraméter (gd) a grafikus drivert, a második (gm) a grafikus módot specifikálja. Minden grafikus kártyához tartozik egy azonosító szám, amit a gd változónak kell tartalmazni: VGA esetén ez a szám a 9. A grafikus mód határozza meg a felbontást, a színek számát és a használható képernyők számát. A VGA három grafikus móddal rendelkezik, amelyet az alábbi táblázat mutat be:

Mód:	Érték:	Felbontás:	Színek:	Képernyők:
VGA _{Lo}	0	640*200	16	4
VGA _{Med}	1	640*350	16	2
VGA _{Hi}	2	640*480	16	1

Alapértelmezés szerint a VGA_{Hi} módot használjuk.

Az eljárás harmadik paramétere (s) a grafikus driver (egavga.bgi) elérési útját adja meg.

Amennyiben nem tudjuk, hogy milyen grafikus drivert és üzemmódot kell használnunk, a DetectGraph eljárással automatikusan detektáltathatjuk a hardvert. Azért is előnyös ez a fajta megoldás, mert akkor a programunk más gépeken is futhat anélkül, hogy egy esetleges inicializálási hiba miatt futási hibával leállna. Másrészt így módunk nyílik rá, hogy ellenőrizzük, hogy az adott gép rendelkezik-e VGA kártyával, hiszen a programunk csak azon fog futni, egyrészt mert az egavga.bgi állomány csak ezt kezeli (meg az EGA-t), másrészt pedig a programozás során kihasználjuk a teljes képernyőt. Persze írható olyan program is, amely az összes TP által kezelt grafikus monitoron fut, de ezekhez már hosszú és unalmas programozói munka szükséges (bár a kíváncsi természetesen ez lenne, és félreértés ne essék, senkit sem igyekszünk lebeszélni róla).

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

A paraméterek megegyeznek az InitGraph eljárásnál leírt paraméterekkel.

Miután kedvünkre kirajzoltuk magunkat, a grafikus képernyőt le kell zárni, és visszatérni a karakteres képernyőhöz. Erre szolgál a paraméterek nélküli CloseGraph eljárás.

```
Procedure CloseGraph;
```

88. példa Inicializáljuk a grafikus képernyőt!

Megoldás: Először detektáljuk a képernyőt, majd megnézzük, hogy VGA kártya van-e a gépben, és csak ebben az esetben inicializálunk. A munka végén természetesen lezárjuk a grafikus képernyőt.

```
Program Grafikal;
Uses Graph;
Var
    gd, gm : integer;
Begin
    DetectGraph(gd,gm);
    If gd=9
    then begin
        InitGraph(gd,gm,'c:\TP\BGI');
        (Munka a grafikus képernyőn)
        CloseGraph;
    end
```

```
else begin
```

```
    WriteLn('A grafikus képernyő nem iniciali-  
zálható');
```

```
end;
```

```
End.
```

Annak érdekében, hogy a programunk többféle grafikus kártya esetén is jól használható legyen, nem konkrét számokkal adjuk meg a koordinátákat, hanem konstansok és változók segítségével. A képernyő méretét két függvény adja meg nekünk:

```
Function GetMaxX:word; – Az utolsó oszlop sorszáma
```

```
Function GetMaxY:word; – Az utolsó sor sorszáma
```

VGA képernyő esetén GetMaxX=639, GetMaxY=479.

6.3. Színek használata

Ugyanúgy, ahogy a karakteres képernyőn, itt is van aktuális rajzoló-szín és aktuális háttérszín. Minden rajzoló eljárás a rajzot az aktív színnel készíti el. Alaphelyzetben ugyanazzal a 16 színnel gazdálkodhatunk, mint a karakteres képernyőn. Most pedig lássuk a színeket kezelő eljárásokat!

```
Procedure SetColor(szin:word);
```

– Az aktuális rajzoló szín megadása.

```
Procedure SetBkColor(szin:word);
```

– Az aktuális háttérszín megadása.

Az aktuális színek lekérdezhetőek is, ha valamiért nem vagyunk vele tisztában, hogy az adott eljáráshoz érve éppen mi volt beállítva:

```
Function GetColor:word; – Az aktuális rajzoló színt adja vissza
```

```
Function GetBkColor:word; – Az aktuális háttérszínt adja vissza
```

6.4. A grafikus képernyő törlése

A grafikus képernyő törlése – hasonlóan a karaktereséhez – az aktuális háttérszínre színezi a képernyőt, és mindent alaphelyzetbe állít vissza. Eljárása a ClearDevice.

```
Procedure ClearDevice;
```

6.5. CP – a grafikus kurzor

A grafikus képernyőn is van kurzor, ezt CP-nek (Current Point) nevezzük. A CP nem látható, és az `InitGraph`, illetve a `ClearDevice` eljárások után a bal felső sarokban van (0,0). A CP helye megadható és lekérdezhető az alábbi eljárások és függvények segítségével:

Procedure MoveTo(x,y:integer);

Procedure MoveRel(x,y:integer);

Mindkét eljárás a CP pozícionálását végzi, de míg a MoveTo abszolút koordinátákat használ, addig a MoveRel relatív koordinátákkal dolgozik. Vagyis a MoveTo(10,20) utasításra a CP a (10,20) koordinátájú pontra kerül, a MoveRel(10,20) esetén az aktuális pozíciójához képest 10 képponttal jobbra és 20 képponttal lejjebb kerül.

Function GetX:integer

Function GetY:integer

A két függvény a CP aktuális pozíciójának X, illetve Y koordinátáit adja.

6.6. Pontok

Általában nem rajzolunk képpontként, de időnként szükség lehet erre. A `PutPixel` eljárással kigyűjthetünk egy képpontot, a `GetPixel` függvény pedig egy adott pont színét adja vissza.

Procedure PutPixel(x,y:integer;szin:word);

Az eljárás az (x,y) koordinátájú pontot szin színnel gyűjti ki.

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

A függvény az (x,y) koordinátájú pont színét adja vissza.

Feladatok:

175. Készítsen programot, amely a képernyőt teleszórja véletlenszerűen pontokkal!

6.7. Vonalak

Vonalak rajzolása már sokkal gyakoribb feladat. Mielőtt azonban egy vonalat megrajzolnánk, meg kell határoznunk annak stílusát.

Procedure SetLineStyle(s,m,v:word)

Az eljárás első paramétere (s) a vonal stílusát határozza meg. Erre a célra a TP néhány előre definiált konstans-bocsát a rendelkezésünkre:

Konstans	Érték	Leírás
solidln	0	folytonos vonal
dottedln	1	pontozott vonal
centerln	2	egyenetlen szaggatott vonal
dashedln	3	egyenletes szaggatott vonal
userbitln	4	felhasználó által definiált vonal

A második paraméter (m) akkor adható meg, ha saját vonalstílust szeretnénk készíteni. Az érték egy bitminta definiál, az-az ahol a 16 bites számban egyes áll, ott pontot fogunk látni, ahol nulla, ott szünetet.

A harmadik paraméter (v) adja meg a vonal vastagságát. Sajnos csak kétféle vonalvastagság közül választhatunk:

Konstans	Érték	Leírás
normwidth	1	vékony vonal
thickwidth	3	vastag vonal

Alapértelmezés:

`SetLineStyle(solidln,0,normwidth);` azaz vékony folytonos vonal.

Procedure SetWriteMode(m:word)

Az eljárás a vonal kirajzolásának módját határozza meg. Ez kétféle lehet:

Konstans	Érték
copyput	0
XORput	1

Mindkét konstans kettes számrendszerbeli bitműveletet valósít meg. A lényeg az, hogy a copyput alkalmazásával a vonal a megadott rajzoló színnel jelenik meg (ez az alapértelmezés), míg a XORput alkalmazásával a háttérszín (pontosabban azon képpontok színe, amelyre a vonal pontjai kerülnek) és a rajzolószín közötti XOR művelet adja meg a vonal színét. Ez többek között azért érdekes, mert egy vonal kétszeri kirajzolása az eredeti színt állítja vissza.

Most már nekiláthatunk vonalat rajzolni, ehhez három eljárás áll a rendelkezésünkre:

Procedure Line(x1,y1,x2,y2:integer);

Az eljárás vonalat rajzol a képernyőre, amelynek kezdőpontja (x1,y1), végpontja pedig (x2,y2). A vonal az aktuális színnel, stílussal és vastagsággal kerül a képernyőre.

Procedure LineTo(x,y:integer);

Az eljárás vonalat rajzol a képernyőre, amelynek kezdőpontja a CP, végpontja pedig (x,y).

Procedure LineRel(x,y:integer);

Az eljárás vonalat rajzol a képernyőre, amelynek kezdőpontja a CP, végpontja pedig a CP-től (x,y)-nal arrébb lévő pont. A használata tehát a MoveRel-hez hasonló: relatív a végpont megadása.

Feladatok:

176. Készítsen eljárást, amelyik egy nagy csúcsára állított háromszöget rajzol a képernyőre!
177. Készítsen eljárást, amelyik egy házikót rajzol a képernyőre!

6.8. Alakzatok

A Turbo Pascal számos olyan eljárást kínál, amelyek különféle síkidomokat rajzolnak. Mindegyikre igaz, hogy a körvonalat az aktuális színnel, vonalstílussal és -vastagsággal rajzolja. Lássuk a legfontosabbakat!

Procedure RectTangle(x1,y1,x2,y2:integer);

Az eljárás egy téglalapot rajzol, amely a bal felső és jobb alsó sarkának koordinátáit adjuk meg ((x1,y1), illetve (x2,y2)).

Procedure Circle(x,y:integer;r:word);

Az eljárás kört rajzol a képernyőre, amelynek középpontja az (x,y) koordinátájú pont, sugara pedig r.

Procedure Arc(x,y:integer;ksz,vsz,r:word);

Az eljárás körívet rajzol a képernyőre, amelynek középpontja az (x,y) koordinátájú pont, sugara pedig r. A körívet egy szögtartomány segítségével definiálhatjuk, amelynek kezdő szöge a ksz, végződése pedig a vsz változókkal megadott. A szögeértékeket – a trigonometriai függvényekkel ellentétben – itt fokok-

ban kell megadni. A teljes kör egy [0..360] fokos tartományba esik, ahol a 0 fok a 3 órának megfelelő irányban van, és a szögek az óramutató járásával ellentétes irányba növekednek (azaz 12 óra 90 foknak, 9 óra 180 foknak, 6 óra pedig 270 foknak felel meg). Az itt leírtak szerint használja a szögeket a Graph unit összes szögeket használó modulja is.

Procedure Ellipse(x,y:integer;ksz,vsz,a,b:word);

Az eljárás ellipszist, vagy egy ellipszisívet rajzol a képernyőre, amelynek középpontja az (x,y) koordinátájú pont, tengelyei pedig a és b hosszúságúak. A ksz és vsz változók az Arc eljárshoz hasonlóan az ellipszisív kezdő és végszögét adják meg.

A fenti eljárások előre meghatározott alakzatot képesek rajzolni. Tetszőleges pontsorozat, vagy zárt alakzat rajzolásához a DrawPoly eljárás használható.

Procedure DrawPoly(n:word;var pontok);

Az eljárás első paramétere a pontok számát, míg második paramétere magukat a pontokat tartalmazza. Ez utóbbi típusa nem definiált előre. Célszerű használatát a következő példa program mutatja be.

89. példa Rajzoljunk egy háromszöget!

Megoldás:

```

Program Haromszog;
Uses Graph;
Var
    gd,gm : integer;
Const
    pontok : array[1..4] of PointType =
        ((x:10; y:10), (x:630; y:10), (x:320;
y:470), (x:10; y:10));
Begin
    DetectGraph(gd,gm);
    InitGraph(gd,gm,'C:\TP\BGI');
    DrawPoly(4,pontok);
    Readln;
    CloseGraph;
End.

```

A program egy nagyméretű, csúcsán álló, egyenlő szárú háromszöget rajzol. A PointType típus a Graph unit-ban van deklarálva a következőképpen:

```

Type
  PointType = record
    x,y : integer;
  end;

```

A **pontok** egy tipizált konstans, amely a háromszög pontjait tartalmazza. Figyeljük meg, hogy nem három, hanem négy pontot tartalmaz (a kezdőpontot kétszer)! Ez azért kell, hogy az eljárás a pontsorozatot bezárja.

Feladatok:

178. Készítsen programot, amely minél szebb olimpiai ötkarikát rajzol a képernyőre!
179. Készítsen programot, amely véletlenszerű paraméterekkel 100 téglalapot rajzol a képernyőre!
180. Készítsen programot, amely véletlenszerű paraméterekkel 100 kört rajzol a képernyőre!
181. Készítsen programot, amely véletlenszerű paraméterekkel 100 véletlenül választott alakzatot rajzol a képernyőre!

6.9. Kitöltött alakzatok

A kitöltés azt jelenti, hogy valamilyen színnel, illetve mintával az alakzat belsejét befestjük. Mielőtt kitöltött alakzatot rajzolnánk, meg kell adnunk a kitöltés módját.

```

Procedure SetFillStyle(minta,szin:word);

```

Az eljárás első paramétere definiálja a kitöltő mintát, a második paraméter pedig a kitöltő minta színét adja meg. A lehetséges előre definiált mintákat az alábbi táblázat mutatja:

Konstans	Érték	Leírás
EmptyFill	0	Háttérszínű kitöltés
SolidFill	1	Rajzoló színű kitöltés
LineFill	2	Vízszintes csíkozás
LtSlashFill	3	Ferde vékony csíkozás (/)
SlashFill	4	Ferde vastag csíkozás (/)
BkSlashFill	5	Ferde vastag csíkozás (\)
LtBkSlashFill	6	Ferde vonalas minta (\)
HatchFill	7	Kockás minta
XHatchFill	8	Ferde kockás minta

Konstans	Érték	Leírás
InterleaveFill	9	Egyenletes tónus
WideDotFill	10	Pontozott kitöltés
CloseDotFill	11	Sűrű pontozott kitöltés
UserFill	12	Felhasználó által definiált minta ¹⁷

Kitöltött alakzatok a SetFillStyle eljárásban definiált kitöltéssel készülhetnek az alábbiakban bemutatott eljárásokkal.

```

Procedure FloodFill(x,y,szin:word);

```

Az eljárás kitölti a képernyő egy területét. Ha az (x,y) pont egy **szin** színű vonalakkal határolt alakzat belsejébe esik, akkor kitölti azt a SetFillStyle eljárásban definiált mintával és színnel (a **szin** paraméter tehát nem a kitöltés színét adja!). Ha az alakzat nem zárt, akkor a teljes képernyőt befestí!

```

Procedure Bar(x1,y1,x2,y2:integer);

```

Az eljárás egy kitöltött téglalapot rajzol, amely a bal felső és jobb alsó sarkának koordinátáit adjuk meg ((x1,y1), illetve (x2,y2)).

```

Procedure Bar3D(x1,y1,x2,y2:integer;m:word;b:boolean);

```

Az eljárás egy kitöltött téglalapot rajzol, amelynél az előlap bal felső és jobb alsó sarkának koordinátáit adjuk meg ((x1,y1), illetve (x2,y2)). Az **m** változó a téglalapot mélységét adja meg képpontokban. Ha a **b** változó értéke TRUE (IGAZ), akkor az eljárás megrajzolja a felső éleket is, ellenkező esetben nem (pl. egy másik téglalapot rajzolható így az előző tetejére). Az eljárás nem perspektivikus, hanem axonometrikus képet ad (a távolodó élek szétartóak).

```

Procedure PieSlice(x,y:integer;ksz,vsz,r:word);

```

Az eljárás körcíkket rajzol, paraméterezése megegyezik az Arc eljárásnál bemutatottakkal. Ha **ksz**=0 és **vsz**=359, akkor kitöltött kört kapunk.

```

Procedure FillEllipse(x,y:integer;a,b:word);

```

Az eljárás kitöltött ellipszist rajzol a képernyőre, amelynek középpontja az (x,y) koordinátájú pont, tengelyei pedig **a** és **b** hosszúságúak.

¹⁷ Saját minta a SetFillPattern eljárással készíthető; ennek ismertetésére e könyv keretei között nem kerül sor.

Procedure Sector(*x,y:integer;ksz,vsz,a,b:word*);

Az eljárás kitöltött ellipszisszeletet rajzol a képernyőre, amelynek középpontja az (*x,y*) koordinátájú pont, tengelyei pedig *a* és *b* hosszúságúak. A *ksz* és *vsz* változók az Arc eljáráshoz hasonlóan az ellipszisív kezdő és végszögét adja meg.

Procedure FillPoly(*n:word;var pontok*);

Az eljárás egy pontjaival definiált kitöltött sokszöget rajzol. Paraméterezése és működése megegyezik a DrawPoly eljárással.

Feladatok:

182. Készítsen programot, amely véletlenszerű paraméterekkel 100 kitöltött téglalapot rajzol a képernyőre!
183. Készítsen programot, amely véletlenszerű paraméterekkel 100 kitöltött kört rajzol a képernyőre!
184. Készítsen programot, amely véletlenszerű paraméterekkel 100 véletlenül választott kitöltött alakzatot rajzol a képernyőre!

6.10. Szövegek

A grafikus képernyőre sokkal szebb szövegeket készíthetünk, mint karakteres képernyőre. Hasonlóan a vonalakhoz és alakzatokhoz, mielőtt nekilátnánk a képernyőre írni, meg kell adnunk a kiírás paramétereit.

Procedure SetTextStyle(*font,irany,meret:word*);

Az eljárás első paramétere a betűtípust adja meg. Az alábbi táblázatban látható a választék:

Konstans	Érték
DefaultFont	0
TriplexFont	1
SmallFont	2
SansSerifFont	3
GothicFont	4

Fontos: az ékezetes betűket ezek a betűtípusok általában nem tartalmazzák, tehát nagyon kell vigyázni a magyar nyelvű feliratok helyességére!

A második paraméter a kiírás irányát adja:

Konstans	Érték	Leírás
HorizDir	0	Vízszintes szöveg
VerDir	1	Függőleges szöveg

A harmadik paraméter a szöveg méretét definiálja. Egy karakter egy 8*8 képpontból álló négyzetet foglal el. A **meret** paraméter valójában egy szorzószám, amely megadja, hogy a karakter eredeti méretének hányszorosát foglalja el (pl. ha a méret 4, akkor a karakter 32*32 képpontot foglal el). A paraméter maximális értéke 10. (A SmallFont esetén a méretezés eltérő.)

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

Az eljárás a majdan kiíratandó szöveg pontos helyét képes megadni. A kiírás során meg kell adnunk egy képpontot, ahová a szöveget írjuk. Ez az eljárás definiálja, hogy a szövegünk melyik részét szeretnénk a megadott pontra igazítani. A *v* paraméter a vízszintes, az *f* paraméter a függőleges igazítást adja meg. Az alábbi táblázat mutatja a lehetőségeket:

Konstans	Érték	Leírás
Vízszintesen:		
LeftText	0	A szöveg baloldala
CenterText	1	A szöveg közepe
RightText	2	A szöveg jobb oldala
Függőlegesen:		
BottomText	0	A szöveg alja
CenterText	1	A szöveg közepe
TopText	2	A szöveg teteje

Miután a beállításokat megtettük, jöhet a kiírás:

Procedure OutText(*s:string*);

Az eljárás az *s* szöveget írja ki a képernyőre, mégpedig a CP pozíciójára.

Procedure OutTextXY(*x,y:integer;s:string*);

Az eljárás az *s* szöveget írja ki a képernyőre, mégpedig az (*x,y*) koordinátájú pontra.

Kis nehézséget jelent a kiíró eljárásoknál, hogy csak szöveg kiírására alkalmasak. Ha számot szeretnénk kiíratni, akkor azt előbb át kell alakítani szöveggé az Str eljárással.

90. példa Írassuk ki a nevünket a képernyő közepére, majd alá az életkorunkat kisebb betűkkel!

Megoldás:

```
Program GrafikusSzoveg;
Uses Graph;
Const
  nev = 'Zseni Alfonz';
  kor = 32;
Var
  gd,gm : integer;
  korszoveg : string;
Begin
  DetectGraph(gd,gm);
  InitGraph(gd,gm,'C:\TP\BGI');
  SetTextJustify(centerText,centerText);
  SetTextStyle(gothicfont,horizdir,4);
  OutTextXY(320,240,nev);
  Str(kor,korszoveg);
  SetTextStyle(gothicfont,horizdir,2);
  OutTextXY(320,270,korszoveg+' éves');
  Readln;
  CloseGraph;
End.
```

7. Saját unitok készítése

A Turbo Pascal-ban a nyelv gerincét alkotó eljárások és függvények programkönyvtárakban – ún. unitokban – vannak. Nagyobb lélegzetű programnál célszerű, ha nem egyetlen állományban van a teljes forráskód, hanem több részre osztva – ezáltal áttekinthetőbb és könnyebben módosítható lesz a programunk. Lehetőségünk van arra, hogy saját unitokat készítsünk, és eljárásainkat, függvényeinket, változóinkat, típusainkat, konstansainkat, stb. ezekben helyezzük el.

Egy unit ugyanúgy épül fel, ahogyan egy program is. Ugyanúgy PAS kiterjesztésű állományba is kerül. Lefordítva azonban TPU kiterjesztést kap (Turbo Pascal Unit). Egy unit felépítése a következőképpen néz ki:

```
Unit Unitnév;
Interface
  itt a deklarációs rész
Implementation
  implementációs (megvalósítási) rész
[Begin]
  [inicializációs rész]
End.
```

A unit neve, és az elmentett fizikai állomány neve nem kell, hogy azonos legyen, de célszerű, és mindenképpen javaslom. Ezen a néven kell majd hivatkozni rá a főprogramban, vagy egy másik unitban a **USES** kulcsszó után.

Az interface rész tartalmazza azokat a globális deklarációkat, amelyek a unitot hívó programból is elérhetők lesznek. Az implementációs rész tartalmazza a privát deklarációkat, vagyis azokat, amelyeket csak a unit maga használ fel, és az interface részben deklarált eljárások és függvények megvalósítását. Az inicializációs részben foglalt eljárások a unit első hívásakor futnak le, de nem kötelező ide bármit is írni. A **Begin** csak akkor kell, ha van inicializációs rész, de nem hiba a megadása egyébként sem.

Egy unit leggyakrabban eljárások és függvények tárolására szolgál. Eltérvően a hagyományos programoktól, itt ezeket is deklarálni kell – amelyiknél ezt elmulasztjuk, az nem lesz elérhető a unitot hívó programból. A unitok használatát egy példán keresztül mutatjuk be:

91. példa Készítsünk programot, amely egy tetszőleges (bekért) szöveget ír ki a képernyőre egy véletlenszerűen választott sor közepére! A kiírást és az adatbekérést egy-egy eljárás, a véletlen szám generálását, és a kiíratás kezdetét egy-egy függvény adja meg. Az alprogramokat egy SEGED névű unitban helyezzük el. A megoldásban először a unitot készítjük el, és csak aztán a főprogramot.

UNIT: Gépeljük be az alábbi unitot!

```
UNIT Seged;

INTERFACE
Uses Crt;
Var
    sor, oszl : integer;
    szoveg : string;
Procedure Beker(var sz:string);
Procedure Kiir;

IMPLEMENTATION

Function VeletlenSor:integer;
Begin
    VeletlenSor:=Random(24)+1;
End;

Function HolKezd(sz:string):integer;
Begin
    HolKezd:=40-(Length(sz) div 2);
End;

Procedure Beker;
Begin
    ClrScr;
    WriteLn('Milyen szöveget irjak ki?');
    ReadLn(sz);
End;

Procedure Kiir;
Begin
    ClrScr;
    sor:=VeletlenSor;
    oszl:=HolKezd(szoveg);
    GotoXY(oszl, sor);
    WriteLn(szoveg);
End;

BEGIN
    Randomize;
END.
```

Most mentsük el **SEGED** néven az elkészült unitot!

Egy másik modul (program vagy unit) a **Kiir** és **Beker** eljárásokat el fogja tudni érni, mert deklaráltuk őket az interface részben, míg a két függvény esetén ez nem történt meg, így csak a unit saját eljárásaiból érhetők el. Az alprogramok megvalósításánál a paraméterlista már elhagyható. Ha mégis megadjuk azt az implementációs részben is, akkor egyeznie kell a deklarált paraméterlistával! Az inicializációs részben szerepeltetjük a Randomize eljárást, hogy a unitban szereplő véletlenszám-generátorok kezdőértéke is véletlenszerű legyen.

Főprogram: Most kérjünk egy új lapot (File/New), és gépeljük be a főprogramot is!

```
Program KozepreKiir;
Uses Seged;
Begin
    Beker(szoveg);
    Kiir;
    ReadLn;
End.
```

A főprogramot is mentsük el! Most két állományunk van, két külön ablakban. Ha a unitban kell valamit módosítani, akkor a Window menü segítségével válthatunk aktív ablakot. A teljes programot lefordítani és futtatni úgy kell, hogy a főprogram ablaka legyen aktív.

Ha az eddigiekben használt Compile/Compile (ALT-F9) paranccsal fordítunk, az csak az aktív modult fogja fordítani. A Compile menüben található még a Make (F9) és Build lehetőség is. A Make-kel fordítva a programot, a fordítás a főprogramra, és minden olyan unitra terjed ki, amelyik az utolsó fordítás óta módosult (munka közben ez ajánlható leginkább). A Build lefordít minden modult.

8. Dinamikus adatszerkezetek

A Turbo Pascal egyik kellemetlen tulajdonsága az, hogy nem használhatunk akármennyi adatot. A programunk által használt teljes adattömegnek bele kell férnie egyetlen adatszegmensbe, azaz nem haladhatjuk túl a 64 kbyte-ot. Ez elsőre nem tűnik kevésnek, de például egy 15000 valós típusú adatot tartalmazó vektor már önmagában is meghaladta ezt a határértéket, és akkor a program többi változója és konstansai már sehová sem fér. Ilyenkor nem marad más választásunk, mint az adatokat fájlban tárolni.

A problémát (többek között) az okozza, hogy az adatszerkezeiteink számára már fordítási időben le kell foglalni a megfelelő méretű helyet a memóriában, és ez a hely a program futása alatt végig lefoglalt is marad. Az ilyen típusú adatokat statikus adatoknak nevezzük.

A statikus adatokkal ellentétben a dinamikus adatok számára csak futási időben foglaljuk le a helyet, és ha már nincs rá szükség, rögtön fel is szabadíthatjuk azt¹⁸. Például egy tömböt, amire csak átmenetileg van szükség, mert valamilyen számítási segédletnek használjuk, célszerű dinamikusan kezelni, hogy a használat végeztével megszabadulhassunk tőle.

Dinamikus változók használatánál saját magunknak kell végeznünk a memória kezelését, és ehhez meg kell ismerkednünk egy új típusal, a mutatóval (pointer). A mutató típus nem más, mint egy memóriacím (4 bájt).

8.1. Dinamikus adatok deklarálása

Dinamikus adattípus deklarálásánál a típus már ismert megadásán túl egy mutatót is definiálnunk kell, amely az adott adattípusra mutat a memóriában.

```
Type
  mutató = ^T;
  T = Típus;
```

¹⁸ Persze így sem foglalhatunk le 64 Kb-ajtnál nagyobb területet!

A T típust hagyományos módon deklaráljuk, és megadunk egy mutatót is, amely a T típusra fog mutatni. A deklarációt a ^ (kalap) karakter segítségével végezhetjük.

8.2. Értékadás = memóriaterület lefoglalása

A mutató típusú változó nem kap a hagyományos értelemben értéket, tehát a := értékadás itt nem fog működni. Helyette a memóriában helyet kell foglalnunk az általa mutatott adatszerkezet számára. Ezt a New vagy a GetMem eljárással tehetjük meg.

```
Procedure New(var p:pointer);
Procedure GetMem(var p:pointer; m:word);
```

A két eljárás között az a különbség, hogy míg a New automatikusan foglal le annyi helyet, amennyire az adott változónak szüksége van, addig a GetMem esetén az m paraméterben nekünk kell megadni a terület nagyságát.

A dinamikus változók a memória ún. HEAP területére kerülnek. A HEAP-ben a szabad, tehát lefoglalható területek nagyságáról a MemAvail és a MaxAvail függvények tájékoztathatnak bennünket.

```
Function MemAvail:longint;
```

A függvény a szabad memóriablokkok teljes méretét adja meg.

```
Function MaxAvail:longint;
```

A függvény a legnagyobb szabad memóriablokk méretét adja meg.

Miután lefoglaltunk helyet, az adatainkra a **mutató**[^] formával hivatkozhatunk.

8.3. A lefoglalt terület felszabadítása

A feleslegessé vált adatszerkezet által elfoglalt memóriaterület felszabadítását a Dispose, illetve FreeMem eljárásokkal végezhetjük.

```
Procedure Dispose(var p:pointer);
Procedure FreeMem(var p:pointer;m:word);
```

A New a Dispose¹⁹, míg a GetMem a FreeMem eljárással alkalmazható együtt.

¹⁹ A New-val lefoglalt memóriaterület felszabadítható még a Mark és Release eljárások segítségével is. A legoptimálisabb memóriakezelést azonban a Dispose végzi.

8.4. Egy egyszerű példa

92. példa Egy 100 elemű tömböt deklaráljunk dinamikusán, foglaljunk le helyet a számára, töltsük fel véletlenszerűen adatokkal, majd szabadítsuk fel a területet! A programcskának persze ebben a formájában különösebben sok értelme nincsen, csak a dinamikus adatkezelést szeretnénk bemutatni a segítségével.

1. megoldás: A New és Dispose eljárások alkalmazásával:

```
Program Dinamikus1;
Type
  tombmut = ^tomb;
  tomb = array[1..100] of integer;
Var
  tm : tombmut;
  i : integer;
Begin
  New(tm);
  For i:=1 to 100 do
    tm^[i]:=Random(32000);
  Dispose(tm);
End.
```

2. megoldás: A GetMem és FreeMem eljárások alkalmazásával:

```
Program Dinamikus2;
Type
  tombmut = ^tomb;
  tomb = array[1..100] of integer;
Var
  tm : tombmut;
  meret, i : word;
Begin
  meret:=SizeOf(tomb);
  GetMem(tm,meret);
  For i:=1 to 100 do
    tm^[i]:=Random(32000);
  FreeMem(tm,meret);
End.
```

9. Alkalmazások

Ebben a fejezetben néhány olyan programozási példát mutatunk be, amelyek gyakran fordulnak elő a munkánk során.

9.1. Gyakori feladatok vektorokra

A vektorok és a tömbök talán a leggyakoribb adattároló szerkezetek, amelyeket a programjainkban alkalmazunk. Érdemes végignézni néhány olyan tipikus feladatot, amely ezeket az adatszerkezeteket kezeli.

Szakirodalomban ezeket elemi algoritmusok vagy programozási tételek néven találjuk meg. Itt – szakítva a hagyományos, általános tárgyalási móddal – nem az ún. absztrakt algoritmusokat tárgyaljuk²⁰, hanem egy-egy gyakorlati példát mutatok be.

A példák nem teljes programok, csak egyes eljárások lesznek. Ezen kívül csak a vektor deklarációját adjuk meg.

9.1.1. Összegzés

93. példa Egy 12 elemű vektor tartalmazza egy kisvállalkozás havi bevételeit. Készítsünk eljárást, amely megadja az éves bevételt!

Megoldás: Egy olyan eljárást kell készíteni, amely végigmegy a vektor elemein, és közben összeadogatja azokat. Ehhez szükségünk lesz egy segédváltozóra (s) is.

```
Type
  BevetelTomb : array[1..12] of longint;

Procedure EvesBevetel (bev:BevetelTomb;var
  eredm:longint);
Var i,s:longint;
```

²⁰ Általános tárgyalásra lehet, hogy alkalmasabb lenne az absztrakt algoritmus, de a példákon keresztül bemutatva jobban érthető. Ráadásul egy kis gondolkodásra is készítheti az olvasót.

```

Begin
s:=0;
For i:=1 to 12 do
    s:=s+bev[i];
eredm:=s;
End;

```

Eljárás paramétereként csak deklarált típussal rendelkező formális változó állhat, ezért kellett a **BevetelTomb** típust megadni. Az eljárás az *s* változóban számlálja az összeget, végül az **eredm** kimenő változó kapja meg a végösszeget értékül (persze használhattuk volna a rész-számításokra is az **eredm** változót az *s* helyett).

Az eljárás alkalmas lehet arra, hogy kis átalakításokkal minden hasonló jellegű – vagyis amikor egy vektor elemeit összegezni (szorozni, átlagolni, stb.) kell. Használható az algoritmus arra az esetre is, amikor nem egy vektor elemeivel számolunk, hanem például egész számokkal (pl. adjuk össze a számokat 1-től 100-ig).

Feladatok:

185. Készítsen függvényt, amely összeadja az egész számokat 1-től N-ig (N bemenő paraméter)!
186. Készítsen függvényt, amely összeadja az egész számokat M-től N-ig (M és N bemenő paraméter)!
187. Alakítsa át az előbbi példát úgy, hogy a havi bevételek átlagát kapjuk eredményül!

9.1.2. Keresés

Ebben a feladattípusban a vektor elemei között egy adott tulajdonságú elemet keresünk. Lehet a feladat eredménye az, hogy van-e ilyen elem; lehet eredménye a keresett elem (az indexe vagy maga az elem); valamint lehet eredménye az, hogy hány darab ilyen elem van.

94. példa Az előző példában deklarált havi bevételeket tartalmazó vektor alapján írjunk függvényt, amely eredményül azt adja, hogy volt-e hónap, amikor 100 000 Ft-nál több volt a bevétel!

Megoldás: Itt nem feltétlenül kell a vektor minden elemét végignézni, hiszen ha találtunk ilyen elemet, a kérdést már meg tudjuk válaszolni.

```

Function VanSzazezer (bev: BevetelTomb): boolean;
Var i: integer;
Begin
i:=1;

```

```

While (bev[i]<100000) and (i<=12) do
    Inc(i);
VanSzazezer:=(i<=12);
End;

```

A függvény egy előtesztelési ciklus²¹ segítségével állapítja meg, hogy van-e ilyen eleme a vektornak. Ha az egész sorozaton végig kellett menni (*i*>12), az azt jelenti, hogy nincs ilyen szám, különben (*i*<=12) talált ilyen számot. Ennek mintájára már nem okozhat gondot megírni azt a programot, ami nem a „van-e” kérdésre felel, hanem a „melyik az”, vagy a „hányadik” kérdésre. Ekkor célszerű egy vagy több kimenő (**var**) paraméterben megadni a kívánt információt.

95. példa Módosítsuk a függvényt úgy, hogy ha talált 100 000 Ft-nál nagyobb bevételt, akkor azt is megadja, hogy ez a vektor hányadik eleme, és azt is, hogy ez mennyi!

Megoldás: Két kimenő paraméterben adjuk meg a kívánt információt.

```

Function Bevetel (bev: BevetelTomb; var index, ertek: longint): boolean;
Var i: integer;
Begin
i:=1;
While (bev[i]<100000) and (i<=12) do
    Inc(i);
If i<=12
    then begin
        index:=i;
        ertek:=bev[i];
    end;
Bevetel:=(i<=12);
End;

```

96. példa Módosítsuk úgy a függvényt, hogy az eredmény az legyen, hogy hány darab 100 000 Ft-nál nagyobb összeg van!

Megoldás: Most már biztosan végig kell menni minden adaton, tehát számlálós ciklust alkalmazunk.

```

Function Megszamol (bev: BevetelTomb): longint;
Var
    i: integer;
    s: longint;
Begin
s:=0;

```

²¹ Azért előtesztelési, mert lehet, hogy már az első elem is megfelel a feltételeknek, és a ciklusmagot egyszer sem kell végrehajtani. Természetesen megoldható a feladat hátultesztelési ciklussal is, csak akkor a ciklusváltozónak más kezdőértéket kell adni, és a ciklusfeltételt az ellentettjére kell átírni.

```

For i:=1 to 12 do
  If bev[i]>100000 then Inc(s);
Megszamol:=s;
End;

```

9.1.3. Maximum-kiválasztás

A feladat az, hogy egy vektor legnagyobb elemét keressük meg. Eredményül a legnagyobb elemet, és az indexét szeretnénk megkapni. Itt a vektor minden elemét végig kell nézni (ebben különbözik a kereséstől).

97. példa Az eddig is alkalmazott havi bevételeket tartalmazó vektorban keressük meg a „legjobb” hónapot, és azt is adjuk meg, hogy mennyi volt itt a bevétel!

```

Procedure Maximum (bev: BevetelTomb; var
max, ind: longint);
Var i: integer;
Begin
  i:=1;
  max:=bev[i];
  For i:=2 to 12 do
    If bev[i]>max
      then begin
        ind:=i;
        max:=bev[i];
      end;
  End;

```

Feladatok:

188. Módosítsa a fenti példát úgy, hogy az adatsor legkisebb elemét keresse!

9.1.4. Rendezés

Nagyon gyakori feladat, hogy az értékeinket valamilyen jól meghatározott sorrend szerint kell kezelnünk, ehhez pedig az adatainkat rendezni kell tudnunk. A következőkben három rendezési módszert mutatunk be az egyszerűbbek közül.

9.1.4.1. Rendezés minimum-kiválasztással

A rendezési eljárás lényege az, hogy az első menetben megkeresem a legkisebb elemet, majd ezt megcserélem az első elemmel. Ekkor a legkisebb elem már a helyén van. A továbbiakban ezt ismétlem egyre

rövidebb szakaszokra, hiszen egyre nő a vektor elején a rendezett szakasz hossza. Nézzük meg példán keresztül. Az eredeti vektor:

23	14	12	13	9	1	20	15	7	19
----	----	----	----	---	---	----	----	---	----

Az első lépésben megkeresem a legkisebb elemet (1), és megcserélem az elsővel:

1	14	12	13	9	23	20	15	7	19
---	----	----	----	---	----	----	----	---	----

A második lépésben megkeresem a vektor hátralévő részének legkisebb elemét (7), és megcserélem a másodikikkal:

1	7	12	13	9	23	20	15	14	19
---	---	----	----	---	----	----	----	----	----

A további lépések, most már magyarázat nélkül (a rendezett részt kiemelttem):

1	7	9	13	12	23	20	15	14	19
1	7	9	12	13	23	20	15	14	19
1	7	9	12	13	23	20	15	14	19
1	7	9	12	13	14	20	15	23	19
1	7	9	12	13	14	15	20	23	19
1	7	9	12	13	14	15	19	23	20
1	7	9	12	13	14	15	19	20	23

98. példa Készítsük el azt az eljárást, amely a fentiekben már alkalmazott **bev** vektort rendezzi (emlékeztetőül: a vektor 12 db, longint típusú elemet tartalmaz)!

Megoldás:

```

Procedure Rendezes1;
Var
  i, j, index: integer;
  min: longint;
Begin
  For i:=1 to 11 do begin
    index:=1;
    min:=bev[i];
    For j:=i+1 to 12 do
      If bev[j]<min
        then begin
          index:=j;
          min:=bev[j];
        end;
    bev[index]:=bev[i];
    bev[i]:=min;
  end;
End;

```

Reméljük, hogy az előzetes magyarázkodás után a példa teljesen érthető. A külső ciklus magjának egy-egy végrehajtása során egyre hosszabb rendezett szakaszokat kapunk. A magban lévő blokk az előző fejezetben megismert minimum-kiválasztás, majd a két elem cseréje következik.

9.1.4.2. Buborékos rendezés

A buborékos rendezés lényege, hogy az egymás melletti elemek folyamatos cseréje alakítja ki az eljárás végén a rendezett sorozatot. Az első menetben a vektor végétől kezdve folyamatosan összehasonlítjuk az egymás melletti elemeket, és ha rossz sorrendben vannak (azaz a nagyobb van előbb), akkor megcseréljük őket. A menet végére a legkisebb elem a tömb elejére kerül. Az eljárást addig ismételtetjük egyre rövidebb szakaszokra, amíg az egész sorozat rendezetté válik. Az előző példa alapján az egyes lépések a következők:

23	14	12	13	9	1	20	15	7	19
1	23	14	12	13	9	7	20	15	19
1	7	23	14	12	13	9	15	20	19
1	7	9	23	14	12	13	15	19	20
1	7	9	12	23	14	13	15	19	20
1	7	9	12	13	23	14	15	19	20
1	7	9	12	13	14	23	15	19	20
1	7	9	12	13	14	15	23	19	20
1	7	9	12	13	14	15	19	23	20
1	7	9	12	13	14	15	19	20	23

99. példa Készítsünk eljárást, amely a bev vektort rendezi a buborékos módszerrel!

Megoldás:

```

Procedure Rendezes2;
Var
  i, j: integer;
  e: longint;
Begin
  For i:=2 to 12 do
    For j:=12 downto i do
      If bev[j]<bev[j-1]
        then begin
          e:=bev[j];
          bev[j]:=bev[j-1];
          bev[j-1]:=e;
        end;
    end;
End;
```

A külső ciklus egy-egy végrehajtása során egyre hosszabb rendezett szakaszt kapunk. A belső ciklus végigmegy a vektor végétől a rendezett szakaszig, és elvégzi az összehasonlításokat és a cseréket. A cseré során egy segédváltozóra (e) is szükség van.

9.1.4.3. Beillesztéses rendezés

A módszer lényege, hogy a sorozatot úgy rendezzük, mintha kártyalapokat fognánk a kezünkbe; egyesével vesszük fel az asztalról a lapokat, és helyezzük a kezünkben már ott lévő lapok közé a helyére.

Első menetben a második számot illesztem a helyére, második menetben a harmadikat, stb. A már ismert példán keresztül nézzük meg a módszer működését!

23	14	12	13	9	1	20	15	7	19
14	23	12	13	9	1	20	15	7	19
12	14	23	13	9	1	20	15	7	19
12	13	14	23	9	1	20	15	7	19
9	12	13	14	23	1	20	15	7	19
1	9	12	13	14	23	20	15	7	19
1	9	12	13	14	20	23	15	7	19
1	9	12	13	14	15	20	23	7	19
1	7	9	12	13	14	15	20	23	19
1	7	9	12	13	14	15	19	20	23

100. példa Rendezzük most a bev vektort a beillesztéses rendezéssel!

Megoldás:

```

Procedure Rendezes3;
Var
  i, j: integer;
  e: longint;
Begin
  For i:=2 to 12 do
    begin
      j:=i-1;
      e:=bev[i];
      While (e<bev[j]) and (j>0) do
        begin
          bev[j+1]:=bev[j];
          Dec(j);
        end;
      bev[j+1]:=e;
    end;
End;
```

A belső ciklusban tehát addig tesszük egyre hátrébb az elemeket, amíg el nem érjük a kiválasztott elem helyét. Ekkor beillesztjük azt.

9.1.5. Logaritmikus keresés

A korábbiakban bemutatott keresési eljárás a vektor eleje felől elindul, és addig megy szépen sorban egyesével végig az elemeken, amíg csak meg nem találja a keresett elemet. Ez az ún. lineáris keresés. Amennyiben azonban a vektorunk már rendezett, akkor azt érdemes kihasználni, és egy jóval hatékonyabb keresési eljárást alkalmazni: a logaritmikus keresést.

A módszer lényege az, hogy először megnézem a középső elemet, és összehasonlítom a keresett értékkel. Most három eset lehetséges:

1. megtaláltam a keresett elemet,
2. a keresett elem a vizsgált elem előtt van,
3. a keresett elem a vizsgált elem mögött van.

Ha megtaláltam, akkor vége a keresésnek, de ha nem, akkor is leshűkítettem kb. a felére azt az intervallumot, amelyben tovább kell keresni. A megmaradt részen megismételjük az eljárást egészen addig, amíg megtaláltuk a keresett elemet, vagy egyértelműen megmondható, hogy a keresett elem nincs a vektorban.

A logaritmikus keresés elnevezés onnan származik, hogy a lépések száma, ami alatt a keresett elem megtalálható, N elemű vektor esetén kb. $\log N$, ami azt jelenti, hogy egy 1000 elemű vektorban ezzel az eljárással csupán kb. 10 lépésben megtalálható bármely elem.

101. példa Miután az előző példákban többféleképpen is rendezhettük a bevételeket tartalmazó vektort, semmi akadálya, hogy egy keresést hajtsunk végre! Keressük meg azt a hónapot, amikor a bevétel 100 000 Ft volt!

Megoldás: A logaritmikus keresést alkalmazzuk a **bev** vektorra. A keresett elem a **ke** paraméterben van. Az éppen vizsgálandó intervallum elejét és végét az **e** és **v** változók jelölik. Az intervallum középső elemének indexe a **k** változóban lesz. Természetesen egy tesztelési ciklust alkalmazunk, amely addig megy, amíg meg nem találjuk az elemet, vagy pedig az aktuális intervallum „elfogy”.

```

Procedure LogKeres (bev: BevetelTomb; ke: longint; var
ind: integer; var van: boolean);
Var
    e, v, k : integer;
Begin
    e:=1;
    v:=12;

```

```

Repeat
    k:=(e+v) div 2;
    If ke<bev[k] then v:=k-1;
    If ke>bev[k] then e:=k+1;
Until (ke=bev[k]) or (e>v);
    van:=(e<=v);
    If van then ind:=k;
End;

```

Feladatok:

189. Készítsen programot (játékot), amelyben a feladat a gép által „gondolt” szám kitalálása! A program állítson elő egy véletlen számot, mondjuk 1 és 1000 között! A játékos feladata, hogy ezt a számot kitalálja. Találghat, és a gép minden találgatás után megmondja, hogy a gondolt szám kisebb, vagy nagyobb a tippnél. (A játékos által követendő nyerő stratégia éppen a logaritmikus keresés módszer!)
 190. Ha az előző játékoska jól működik, és a játék során a nyerő stratégiát sikerült elsajátítani, akkor megpróbálkozhatunk a játék fordítottjával: a játékos által gondolt számot találja ki a program!

9.2. Menük készítése

Menüről beszélünk akkor, ha programunk több lehetőséget kínál fel, amelyből – legtöbbször egy billentyű lenyomásával – választhatunk, majd a választás alapján más és más programrészeket hajt végre.

9.2.1. Egyszerű menü

A menük általános sémája a következő:

1. Menüpontok kiírása a képernyőre
2. Várakozás egy billentyű lenyomására, majd a lenyomott billentyű vizsgálata: csak a felkínált lehetőségeket fogadjuk el válaszként!
3. A lenyomott billentyű alapján (ez leggyakrabban egy sorszám vagy egy kezdőbetű, bár sokszor alkalmazzuk a funkcióbillentyűket is) elágazás szervezése, ahol az elágazás egy-egy ágán készítjük el a végrehajtandó programrészeket (célszerű eljárásokat készíteni).

Az alábbi példa jól szemlélteti a menük készítését.

102. példa Készítsünk programot, amely két bekért számmal alapműveleteket végez! A bekérés után kérdezzük meg, hogy melyik műveletet is szeretnénk végezni (menü), majd a kiválasztott műveletet végzük is el!

Megoldás: A program terve a következő:

```

Program Menu
Be: A, B
Menüpontok kiírása (választási lehetőségek: 1-4)
Ciklus
    Be: c (a lenyomott billentyű)
    amíg '1' <= c <= '4'
    ciklus vége (a ciklus gondoskodik arról, hogy csak a lehetséges értékek legyenek
    választhatók)
Elágazás
    c='1' esetén Osszead
    c='2' esetén Kivon
    c='3' esetén Szoroz
    c='4' esetén Oszt
Elágazás vége
Program vége

```

A programlista (a jobb érthetőség kedvéért a főprogramot elláttuk megjegyzésekkel):

```

Program Menu;
Uses Crt;
Var
    a, b : integer;
    c : char;

Procedure Osszead;
Begin
    WriteLn('A számok összege: ', a+b);
    ReadLn;
End;

Procedure Kivon;
Begin
    WriteLn('A számok különbsége: ', a-b);
    ReadLn;
End;

Procedure Szoroz;
Begin
    WriteLn('A számok szorzata: ', a*b);
    ReadLn;
End;

Procedure Oszt;
Begin
    If b=0
    then WriteLn('Nullával nem osztunk!!')
    else WriteLn('A számok hányadosa: ', a/b:5:2);
    ReadLn;
End;

```

```

BEGIN
{Adatbekérés;}
ClrScr;
Write('Mi az első szám?');
ReadLn(a);
Write('Mi a második szám?');
ReadLn(b);
ClrScr;
{Menü kiírása;}
WriteLn('1. összeadás');
WriteLn('2. kivonás');
WriteLn('3. szorzás');
WriteLn('4. osztás');
WriteLn;
WriteLn('Melyiket választod? (1-4)');
{Várakozás billentyűre;}
Repeat
    c:=ReadKey
Until (c>='1') and (c<='4');
{Elágazás;}
ClrScr;
Case c of
    '1' : Osszead;
    '2' : Kivon;
    '3' : Szoroz;
    '4' : Oszt;
end;
END.

```

Az adatbekérés során karakteres változót célszerű alkalmazni numerikus helyett, mert így elkerülhető sok, az amúgy is kritikus adatbekérés során előforduló hiba (hiába tudjuk, hogy számot várunk, ha a felhasználó szöveget gépel be, és erre az eshetőségre a programot fel kell készíteni).

Az előbbi példa nem eléggé elegáns, hiszen a választást csak egyszer kínálja fel a program, majd a kiválasztott tevékenység végrehajtása után leáll. A „valódi”, menüt tartalmazó programokban mindig el kell helyeznünk egy „Kilépés” menüpontot, amelyet kiválasztva léphetünk ki „elegánsan” a programunkból. Máskülönben, a kiválasztott tevékenység végrehajtása után a menüt kell újból felkínálnunk.

A megoldás az lesz, hogy az előbbi programot egy ciklus belsejében helyezzük el...

103. példa Módosítsuk az előző példaprogramot úgy, hogy kezelje a kilépést is!

Terv:

```

Program Menu2
Be: A, B
Ciklus
  Menüpontok kiírása (választási lehetőségek: 1-4; + a 0 lesz a kilépés)
  Ciklus
    Be: c (a lenyomott billentyű)
    amíg '0' <= c <= '4'
    ciklus vége (a ciklus gondoskodik arról, hogy csak a lehetséges értékek le-
gyenek választhatók)
    Elágazás
      c='1' esetén Osszead
      c='2' esetén Kivon
      c='3' esetén Szoroz
      c='4' esetén Oszl
    Elágazás vége
  amíg c='0'
  Ciklus vége
Program vége

```

A programlista (az eljárások nélkül):

```

Program Menu2;
Uses Crt;
Var
  a, b : integer;
  c : char;

BEGIN
  (Adatbekérés:);
  ClrScr;
  Write('Mi az első szám?');
  Readln(a);
  Write('Mi a második szám?');
  Readln(b);
  Repeat
    ClrScr;
    (Menü kiírása:);
    WriteLn('1. összeadás');
    WriteLn('2. kivonás');
    WriteLn('3. szorzás');
    WriteLn('4. osztás');
    WriteLn;
    WriteLn('0. Kilépés');
    WriteLn;
    WriteLn('Melyiket választod? (1-4)');
    (Várakozás billentyűre:);
    Repeat
      c:=ReadKey
    Until (c>='0') and (c<='4');
    (Elágazás:);
    ClrScr;
    Case c of
      '1' : Osszead;
      '2' : Kivon;

```

```

      '3' : Szoroz;
      '4' : Oszl;
    end;
  Until c='0'
END.

```

Feladatok:

191. Készítse el a fenti példa szebbített változatát (szövegek elhelyezése, színek, stb.)!
192. Módosítsa az előző példát úgy, hogy a számok bekérése is egy menüpont legyen!
193. Készítsen egy valutaváltó programot! Menüből lehessen választani, hogy a bekért forint-értéket milyen valutára váltsa a program (pl. dollár, márka, font, stb.)!

9.2.2. Speciális billentyűk kezelése

Az elágazás során azoknak a billentyűknek a kezelése, amelyek megtalálhatók a billentyűzeten – a betűk, a számok és a jelek – nem okoznak gondot. Más a helyzet azokkal, amelyeknek csak az ASCII-kódja ismert. Ilyenek az ENTER, az ESC, a funkcióbillentyűk és a billentyű-kombinációk.

Az ASCII-táblázat első 32 eleme (0-tól 31-ig) vezérlőkaraktereket tartalmaz. Ezek a billentyűzetről a Ctrl+betű billentyűkombinációval gépelhetők be²², de itt van az ENTER, az ESC, a TAB és a BACKSPACE is. Az alábbi táblázat mutatja be a kódjaikat:

Billentyű	Kód
BACKSPACE	8
TAB	9
ENTER	13
ESCAPE	27

A funkcióbillentyűkkel azonban más a helyzet. Az ő kódjuk nem egy, hanem két számból áll, és az első szám a 0. A második értékeket mutatja az alábbi táblázat²³:

Funkcióbillentyű		Shift+	Ctrl+	Alt+
F1	59	84	94	104
F2	60	85	95	105

²² A Ctrl-A kódja 1, a Ctrl-B kódja 2, ... a Ctrl-Z kódja a 26.

²³ A lista nem teljes, csak a fontosabb kódokat írtuk le.

Funkcióbillentyű		Shift+	Ctrl+	Alt+
F3	61	86	96	106
F4	62	87	97	107
F5	63	88	98	108
F6	64	89	99	109
F7	65	90	100	110
F8	66	91	101	111
F9	67	92	102	112
F10	68	93	103	113
Insert	82	–	–	–
Delete	83	–	–	–
Home	71	–	119	–
End	79	–	117	–
PageUp	73	–	132	–
PageDown	81	–	118	–
←	75	–	115	–
→	77	–	116	–
↑	72	–	–	–
↓	80	–	–	–

Vagyis pl. az **F1** kódja 0+59, a **shift-F1** kódja 0+84, a **ctrl-F1** kódja 0+94 és az **alt-F1** kódja 0+104. A dupla kódú billentyűk kezelését mutatja be az alábbi példaprogram.

104. példa Készítsünk programot, amely egy csillagot mozgat a képernyőn! A csillagot a felhasználó mozgatja a nyilakkal, a programból kilépni az ESC-pel lehet. A csillag aktuális pozícióját egy **sor** és egy **oszl** nevű változó tárolja majd. A mozgatásnál vigyázni kell arra, hogy a csillag a képernyőn maradjon, és hogy az újbóli kirajzolásnál az előzőleg kirajzolt csillagot el kell tüntetni. A dupla kódú billentyűket (mint a nyilak is) úgy kell kezelni, hogy kétszer egymás után meg kell hívni a ReadKey függvényt.

Terv:

```

Program Csillag
sor:=12; oszl:=40 {alapbeállítás a képernyő közepére}
Ki(sor,oszl); '*'
Ciklus
  Be: c
  Ki(sor,oszl); szököz {előző csillag törlése}
  Elágazás
    c='←' esetén: Ha oszl>1 akkor oszl:=oszl-1
    c='→' esetén: Ha oszl<79 akkor oszl:=oszl+1

```

```

c='↑' esetén: Ha sor>1 akkor sor:=sor-1
c='↓' esetén: Ha sor<24 akkor sor:=sor+1
Elágazás vége
Ki(sor,oszl); '*'
amíg c=ESC
Ciklus vége
Program vége

```

A programlista:

```

Program Csillag;
Uses Crt;
Var
  c : char;
  sor, oszl : integer;
Begin
  ClrScr;
  sor:=12;
  oszl:=40;
  GotoXY(oszl,sor);
  Write('*');
  GotoXY(1,25);
  Repeat
    c:=ReadKey;
    If Ord(c)=0
    then begin
      c:=ReadKey;
      GotoXY(oszl,sor);
      Write(' ');
      Case Ord(c) of
        75 : If oszl>1 then Dec(oszl);
        77 : If oszl<80 then Inc(oszl);
        72 : If sor>1 then Dec(sor);
        80 : If sor<24 then Inc(sor);
      end;
      GotoXY(oszl,sor);
      Write('*');
      GotoXY(1,25);
    end;
  Until c=#27;
End.

```

A program egy lépésben először eltüntet a csillagot, majd megváltoztatja a lenyomott billentyű alapján a **sor** és **oszl** változók értékeit, végül kirajkja a csillagot az aktuális új helyre. Látható, hogy az elágazásban a **c** változóban tárolt karakter kódját (Ord(c)) vizsgáljuk. A hátultesztelés ciklusban a **c=#27** forma megegyezik az Ord(c)=27 formával (egy tetszőleges karakterre hivatkozhatunk a # jel és a kódja segítségével a példában leírt módon).

Feladatok:

194. A fenti példát próbálja meg kibővíteni úgy, hogy a * ne álljon meg a képernyő szélén, hanem a tulsó oldalon lépjen be!
195. Módosítsa a példát úgy, hogy a HOME lenyomására az adott sor elejére, END-re a végére ugorjon a csillag!
196. A példa még számtalan egyéb lehetőséget rejt, amit érdemes kipróbálni: változtathatjuk a csillag színét, változtatható a kirajzolt karakter, megadható a lépésköz, stb. Próbáljon meg kiegészíteni a programot ezekkel a lehetőségekkel, és akár továbbiakkal is, ha valami érdekes jut az eszébe! (Készíthet akár egy bekerítő kigyó játékot is, két játékoskal, bár ez már egy kicsit bonyolultabb.)
197. A fenti példa az előbbi módosításokkal együtt grafikus képernyőre is áttűtethető. Ha nem egy csillagot, hanem egy pontot mozgatunk, és az előző pontot nem töröljük ki, akkor egy kis rajzoló-programot lehet készíteni. Érdemes ezzel is egy kicsit eljátszogatni!

9.3. Rekurzio

A rekurzio egy nehezebben emészthető, de sok esetben nagyon jól használható módszer. Függvényeket írhatunk meg rekurzív módon, aminek az a lényege, hogy a függvény a végrehajtás során önmagát hívja meg. A matematikából ismerhetünk rekurzív definíciókat, a legismertebb közülük a Fibonacci-sorozat. A sortozatot a következőképpen adjuk meg ($F(n)$ jelöli a sorozat n . elemét):

```
F(1):=1
F(2):=1;
F(n):=F(n-2)+F(n-1)
```

Vagyis a sorozat minden eleme az előző két elem összege legyen. Másik ismert példa a faktoriális kiszámítása. Az n szám faktoriálisa alatt az $1*2*3*...*(n-1)*n$ számot értjük. A definíció itt is megadható rekurzív módon:

```
F(1):=1
F(n):=F(n-1)*n
```

Vagyis az n szám faktoriálisa az $(n-1)$ szám faktoriálisának n -szerese.

- 105. példa** Készítsünk függvényt, amely faktoriálisan számol rekurzív módon!

Megoldás:

```
Function Fakt(x:word):longint;
Begin
  If x=1
  then Fakt:=1
  else Fakt:=Fakt(x-1)*x;
End;
```

- 106. példa** Készítsünk függvényt, amely a Fibonacci-sorozat x -edik elemét számolja ki!

Megoldás:

```
Function Fibo(x:word):longint;
Begin
  If (x=1) or (x=2)
  then Fibo:=1
  else Fibo:=Fibo(x-2)+Fibo(x-1);
End;
```

Megjegyzés: minden rekurzio átírható iterációra (ciklusokra). A rekurzio jól alkalmazható például fraktálok esetén, és bizonyos grafikai feladatoknál. (Például a LOGO programozási nyelv egyik legalapvetőbb eszköze a rekurzio.) Klasszikus példa a rekurziora a Hanoi tornyai játék is.

Vigyázzunk azonban a rekurzioral, mert hibás program esetén könnyen „végtelen ciklusba” keveredhetünk, és akkor nem tudunk kilépni a programból.

10. Egy komolyabb program elkészítése

Ebben a fejezetben egy kis programot írunk meg, végigkövetve a programírási folyamatát.

10.1. A feladat

Készítsünk egy kis kaszinót! A kaszinóban kockázni lehet. A kockajáték során a gép ellen játszunk. Mindenkinek van valahány dobása (ez legyen megadható), akinek az összeredménye jobb, az nyer. Addig lehet játszani, amíg azt nem mondjuk, hogy ki akarunk lépni. Mindaddig, amíg ki nem lépünk, a program tárolja az eredményeket, és kilépkor kiírja. Egy menüt kell tehát készíteni, amelynek a pontjai:

1. A dobások száma
2. Játék
3. Kilépés

A dobások száma menüpontot választva lehet beállítani, hogy egy játékban hány dobása van a játékosoknak (1, 3, 5). Az alapértelmezett beállítás legyen, mondjuk 3 dobás. (A játéknak ez a változata nem ismeri a döntetlen fogalmát. Azonos kockadobás esetén a gép nyer. Ezért kell páratlan számú dobást játszani egy menetben.)

A játék menüpont választásával lehet egy játékot játszani. Ennek során mi is dobunk annyit, amennyit beállítottunk, és a gép is. Akinek az összeredménye jobb, az nyer. A nyerőt a gép kiírja, majd az eredményt hozzászámolja az összeredményhez. Az összeredményt meg fogjuk jeleníteni a menüpontok kiírása során is.

Annak érdekében, hogy a játékos dobása ne csak a géptől függjön, a TV-ben látott módon a gép addig sorsolja a számokat, amíg meg nem állítjuk, például a szóköz billentyűvel.

10.2. A programterv

Milyen adatszerkezetekre van szükségünk? Szerencsére a program nagyon kevés adatot kezel, néhány változó is elég lesz. Kell egy, amelyik tartalmazza a dobások számát (**dobas**); kell kettő, amelyik a nyertes és vesztes játszmákat számlálja (**nyert**, **vesztett**); kell egy karakteres változó, amely a kiválasztott menüpontot tárolja (**mp**); és végül kell néhány egyéb változó, pl. ciklusváltozók.

A program tervét modulonként készítjük el, a mondatyszerű leírást alkalmazva. Először tehát a menü, vagyis a főprogram:

```

Program KockaJatek
dobas:=3
nyert:=0
vesztett:=0
Ciklus
    Menupontok {eljáráshívás}
    EredmenyKiir {eljáráshívás}
    Be: mp (1, 2, 3)
    Elágazás
        mp='1' esetén DobasSzam {Eljáráshívás}
        mp='2' esetén Jatek {Eljáráshívás}
    Elágazás vége
    amig mp='3'
    Ciklus vége
Program vége

```

Most lássuk az egyes eljárásokat. Először a menüpontok kiíratását:

```

Eljárás Menupontok
Képernyőtörítés
Ki: „1. Dobások számának beállítása”
Ki: „2. Játék”
Ki: „3. Kilépés”
Eljárás vége

```

Az aktuális eredmény kiíratását az EredmenyKiir eljárás végzi:

```

Eljárás EredmenyKiir
Ki: nyert, vesztett
Eljárás vége

```

A menüből választható első menüpont a dobások számának megváltoztatása.

```

Eljárás DobasSzam
Képernyőtörítés
Ki: „A dobások aktuális száma:”, dobas

```

Ki: „Hány dobás legyen egy játékban?”

Be: **ujdobas** (1,3, 5)

dobas:=ujdobas

Eljárás vége

És végül maga a játék:

Eljárás Jatek

Képernyőtörítés

gny:=0

jny:=0

Ciklus i:=1-től dobas-ig

Ki: i, „. dobás:”

gd:=véletlen(6)

Ciklus

jd:=véletlen(6)

amíg le nem nyomjuk a szóközt

ciklus vége

Elágazás

gd>=jd esetén: Ki: „Most a gép nyert”

gny:=gny+1

gd<jd esetén: Ki: „Most Te nyertél”

jny:=jny+1

Elágazás vége

Ciklus vége

Ha gny>jny akkor Ki: „Ezt a játékot a gép nyerte”

vesztett:=vesztett+1

különben Ki: „Ezt a játékot Te nyerted”

nyert:=nyert+1

Eljárás vége.

Az eljárásban **gd** változó tartalmazza a gép által dobott számot, **jd** pedig a játékos dobásait. A **gny** és **jny** változók számolják egy meneten belül a gép, illetve a játékos nyert játszmáinak számát. Miután csak páratlan számú dobást engedünk, ezért döntetlen eredmény nem születhet, nem is kell kezelnünk a végén a nyert és veszített játékok számolásánál.

Feladatok:

198. Próbálja meg elkészíteni a programtervet folyamatábra és struktogram segítségével is!

10.3. Kódolás

A kódolás azt jelenti, hogy a megtervezett programot lefordítjuk a Turbo Pascal nyelvére. Egy jó tanács: gépelés közben a program mindig fordítható állapotban legyen! Ha begépeljük egy ciklus fejét, cél-

szerű mindjárt a végét is beírni, és majd ezek után megírni a ciklusmagot. Amikor egy-egy modullal készen vagyunk, célszerű lefuttatni, és kipróbálni, hogy jól működik-e. Készítsük el először a főprogramot, és a fordíthatóság érdekében egyelőre az eljáráshívásokat tegyük kapcsos zárójelek közé (mintha kommentek lennének). Amikor majd egy-egy eljárást elkészítettünk, akkor az eljáráshívásnál kivesszük a kapcsos zárójeleket.

Javasolt, hogy próbáljuk meg önállóan elkészíteni az egyes modulokat, és a végén hasonlítsuk csak össze az itt leírt megoldással, akár működik, akár nem.

Program KockaJatek;

Uses Crt;

Var

dobas,nyert,vesztett : integer;
mp : char;

BEGIN

dobas:=3;

nyert:=0;

vesztett:=0;

Repeat

{Menupontok;}

{EredmenyKiir;}

Repeat

mp:=ReadKey

Until mp in ['1'..'3'];

Case mp of

'1' : {DobasSzam};

'2' : {Jatek};

end;

Until mp='3'

END.

A következő lépés a Menüpontok eljárás kódolása. (Az eljárásokat a főprogram elé kell elhelyezni!)

Procedure Menupontok;

Begin

ClrScr;

WriteLn('1. Dobások számának beállítása');

WriteLn('2. Játék');

WriteLn('3. Kilépés');

WriteLn;

WriteLn('Melyiket választod? (1,2,3)');

End;

A program most kiírja a menüpontokat, és tetszőleges billentyűt le nyomhatunk, nem történik semmi, csak a hármast nyomva látunk

eredményt: kilépünk a programból; azaz eddig minden tökéletes. Következhet az EredmenyKiir eljárás:

```

Procedure EredmenyKiir;
Begin
  GotoXY(50,20);
  Write('eddig nyertél ',nyert,' játszmát');
  GotoXY(50,21);
  Write('és vesztettél ',vesztett,' játszmát');
End;

```

A képernyő jobb alsó sarkába került ezzel az eddigi eredmény. Lehet, hogy a képernyő kinézete még nem tökéletes, de erre ráérünk később is. Most folytassuk a program kódolását a DobasSzam eljárással:

```

Procedure DobasSzam;
Var
  ujdobas,c : char;
  kod : integer;

Begin
  ClrScr;
  WriteIn('A dobások aktuális száma: ',dobas);
  Write('Hány dobás legyen egy játékban? (1,3,5)');
  Repeat
    ujdobas:=ReadKey;
    If not (ujdobas in ['1','3','5'])
      then begin
        Sound(880);
        Delay(100);
        Nosound;
      end;
  Until ujdobas in ['1','3','5'];
  Val(ujdobas,dobas,kod);
  WriteIn;
  WriteIn('A dobások új száma: ',dobas);
  GotoXY(1,25);
  Write('Nyomj egy billentyűt, és visszatérhetsz a menühöz');
  c:=ReadKey;
End;

```

A programlista egy kicsit hosszabb lett, mint a terv, de a tervben nem térünk ki olyasmire, hogy hogyan kérünk be egy számot, ami csak 1, 3, vagy 5 lehet. A bekérésnél egy karakter típusú változót olvasunk be (így nem kell az ENTER-re várni, és elkerülhetjük azokat a bosszantó futási hibákat, amelyek más típusú adat begépelésekor keletkeznek). A ciklus belsejében teszteljük, hogy a leütött billentyű megfelel-e az elvárásoknak, egyébként egy rövidet sípolunk²⁴, így

²⁴ A Sound eljárás paramétereként a hangfrekvenciát kell megadni (a normál A hang 440 Hz), és az így bekapcsolt hang addig szól, amíg a Nosound eljárással ki nem kapcsoljuk azt. A Delay a sípolás hosszát adja meg milliszekundumban. Mindhárom eljárás a Crt unitban van.

figyelmeztetve a játékos, hogy a leütött billentyű nem megfelelő. A beolvasás után a karakteres változót át kell alakítani numerikussá, ezt végzi a Val eljárás.

Végül jöhet a lényeg, a játék elkészítése:

```

Procedure Jatek;
Var
  i,gd,jd,gny,jny : integer;
  c : char;

Begin
  Randomize;
  ClrScr;
  gny:=0;
  jny:=0;
  For i:=1 to dobas do begin
    c:=#0;
    gd:=Random(6)+1;
    GotoXY(1,i*2);
    Write(i,' dob som: ',gd);
    GotoXY(20,i*2);
    Write(i,' dob sod: ');
    Repeat
      jd:=Random(6)+1;
      GotoXY(33,i*2);
      Write(jd);
      If KeyPressed then c:=ReadKey;
      Delay(50);
    Until c=' ';
    If gd>=jd
      then begin
        GotoXY(40,i*2);
        WriteIn('A gép nyert');
        Inc(gny);
      end
    else begin
      GotoXY(40,i*2);
      WriteIn('Te nyertél');
      Inc(jny);
    end;
  end;
  If gny>jny
    then begin
      GotoXY(1,15);
      Write('Ezt a játékot sajnos ');
      Write(gny,':',jny);
      WriteIn(' arányban a gép nyerte');
      Inc(vesztett);
    end
  else begin
    GotoXY(1,15);
    Write('Gratulálok! Ezt a játékot ');
    Write(jny,':',gny);

```

```

        WriteLn(' aranyban Te nyerted');
        Inc(nyert);
    end;
GotoXY(1,25);
Write('Nyomj egy billentyűt, és visszatérhetsz a menü-
      nöz');
c:=ReadKey;
End;

```

Az eljárásban a *KeyPressed* függvényt alkalmaztuk, amelyet nem írtuk le eddig. A függvény értéke IGAZ, ha történt billentyűlenyomás, különben hamis. Ugyanítt a várakozás (*Delay*) csak azért kell, hogy egy kicsit látványosabb legyen a játék. Az eljárás többi része (reméljük) érthető.

10.4. Hibajavítás, tesztelés

Befejeztük a program elkészítését, és a programunk elég jól működik. A bejelentkező képernyő viszont elég csúnyára sikerült, ráadásul a kurzor is rossz helyen villog, és ez meglehetősen zavaró. Többféle megoldást találhatunk a problémára. Legegyszerűbb, ha az EredményKiír eljárást a Menüpontok eljárás előtt hívjuk meg. Cseréljük fel a két eljárást! Most az lett a probléma, hogy az eredmények nem látszanak a képernyőn, mert a MenüKiír eljárásban egy képernyőtörlés van. Az eredmények kiírását ez után kellene tenni, tehát az EredményKiír eljárás hívását helyezzük át a főprogramból a MenüKiír eljárásba, közvetlenül a képernyőtörlés után! Most fordításkor az **Error 3: Unknown identifier** hibaüzenetet kapjuk ebben a sorban. Azért van, mert a meghívott eljárás definíciójának meg kell előznie a programban az őt hívó eljárás leírását. Tehát az egész EredményKiír eljárást át kell helyezni a MenüKiír eljárás elé!

Most már fut a program, de még csúnyább a képernyő képe! Próbáljuk meg a kiírások helyét megváltoztatni úgy, hogy a menüpontok a képernyő közepére kerüljenek, az aktuális eredmény pedig a jobb felső sarokba! Most már sokkal szebb!

Esetleg zavaró lehet az, hogy nem látjuk mindig, hogy hány dobás az alapértelmezett. Ez kiírható például a „2. láték” menüpont után zárójelben. Módosítsuk így a programot!

Futtassuk sokszor a programot, állítsunk be különböző dobásszámokat, játszunk vele! Találtunk hibát? Ha igen, próbáljuk javítani!

Feladatok:

199. A fentiek szellemében Módosítsa a programterv érintett részeit is!

10.5. Hatékonyságvizsgálat

Egy program hatékonyságának vizsgálata és javítása akár egy külön kötetet is megtölthetne. Most csak a legfontosabbakat említjük.

A hatékonyság többféle dolgot jelenthet. Leggyakrabban a futási sebességet értjük ez alatt, de a használt adatok memóriabeli helyfoglalása, vagy a program bonyolultsága is ide tartozó kérdés. Általában ki kell választani, hogy melyik szempont a legfontosabb az adott feladattal, mert a felsorolt három tényező általában csak egymás rovására javítható.

Jelen programcskánk esetében a helyfoglalás nemigen csökkenthető, és olyan kevés változónk van, hogy nem is érdemes foglalkozni vele²⁵. A program sebessége sem számottevő kérdés, bármilyen gép esetén is szépen fog futni²⁶.

A program bonyolultságának problémaköre azonban minden program esetén előkerül: meg kell próbálni egyszerűsíteni a kódot. Többször alkalmazott programrészeket csak egyszer kell elkészíteni, és egy eljárásban célszerű megadni azt. Olyan tevékenységeket, amelyek egy elágazás minden ágán szerepelnek, ki lehet hozni az elágazás elé (ilyen szerepel is a programunkban). Olyan programrészeket, amelyeket nem kell többször végrehajtani, nem kell a ciklus magjába tenni; ki lehet hozni a ciklus elé.

Szóba kerülhet még a program kódjának az olyan megszerkesztése is, hogy az a későbbiekben minél egyszerűbben legyen módosítható vagy továbbfejleszhető. Ennek érdekében célszerű a programban megjegyzéseket elhelyezni; többször használt adatot konstansba tenni, mert akkor csak egy helyen kell változtatni rajta később, ha szükséges (pl. színek, képernyő-pozíciók, kiírt szövegek, stb.).

²⁵ Olyan programoknál, ahol sok adat van, ott kell foglalkozni ezzel a kérdéssel, mert a Turbo Pascal programok adatai által lefoglalható memóriaméret véges: mindössze egyetlen szegmens, azaz 64 KB területet foglalhatnak le. Ez sokszor elég kevés. Például egy személyi lapokat tartalmazó tömb esetén, ahol a nevet 20, és a címet 40 karakteren írjuk le, valamint az egyéb adatokra rászánunk még 10 bájt (ez összesen 70 bájt); nos itt a tömb nem tartalmazhat többet, mint 930 rekordot, ami elég kevés (és az egyéb változóinkról még szó se volt). Ilyenkor az adatokat fájlban kell tárolni (ez persze a sebesség rovására megy, és a program leírása is bonyolultabb lesz).

²⁶ Olyan programoknál, ahol bonyolult számításokat végzünk el nagyon sokszor (ciklus), ott ez már égetőbb probléma. Meg kell próbálni a sebességet növelni. Néhány módszer: számítások helyett adatokat eltárolni (nő a helyfoglalás!); ugyanazokat a kepleteket csak egyszer kiszámolni, és az eredményt változókból tárolni (ugyancsak nő a helyfoglalás, és a program is bonyolultabb!); egyszerűsíteni a számításokon, vagy próbáljuk a végrehajtások számát minimalizálni (bonyolultabb a program!), de ne feledkezzünk meg a hatékonyabb eljárásokról (pl. logaritmus keresés) sem. Ez persze csak izelőd.

Feladatok:

200. Mindkét fő eljárásunk végén van egy ismétlődő programrészlet (várakozás egy billentyűre a menühöz való visszatérés miatt). Helyezze el ezt egy külön eljárásban!
201. Utaltunk rá, hogy előfordul a programban olyan, hogy egy elágazás minden ágán szerepel ugyanaz a programrészlet! Hozza ki az elágazás elé!

10.6. Dokumentáció

Miután egy programot késznek minősítünk, el kell készíteni hozzá a dokumentációt. Kétféle dokumentációt kell írunk: felhasználói és fejlesztői dokumentációt. A felhasználói dokumentációnak gyakorlatilag azt kell tartalmaznia, amit egy majdan felhasználónak kell tudnia a program működéséről. A legfontosabb szempontok a következők:

- A program hardver és szoftver igénye.
- A program telepítése.
- A program használata.
- Teszteredmények.

A fejlesztői dokumentáció kiegészül még ezeken kívül a következőkkel:

- A program továbbfejlesztési lehetőségei.
- A programterv (algoritmus).
- A forráskód kinyomtatva.

Mi most egy egyszerű felhasználói dokumentációt készítünk el:

**Felhasználói dokumentáció
a KOCKAJATEK nevű programhoz**

1. A programnak különösebb hardverigénye nincsen, bármilyen IBM PC kompatibilis számítógépen fut. DOS operációs rendszerre van szüksége.
2. A program nem igényel telepítést, egyszerűen a `kocka.exe` állományt másolja be egy tetszőleges könyvtárba (vagy akár egy floppy-ra, bár ekkor a program futása lassabb lesz). A programot a prompt után gépelt `kocka` paranccsal indíthatja el abból a könyvtárból, ahová masolta.

3. A program használata

A program egy kis szerencsejáték, amit kockával játszanak. Egy játszmban Ön is, és a gép is dob egy bizonyos számú kockával. A menetet az nyeri, aki többször dobott nagyobb (azonos dobások esetén a gép a nyerő!). Amíg ki nem lépünk, a program számolja a nyert és veszített játszmákat, és erről folyamatosan tájékoztat is bennünket. Az egyes menetekben a dobások száma megváltoztatható, akár játszmánként is.

A program indítása után egy menüt látunk magunk előtt, amelyből a felkinált irányokba mehetünk tovább. A dobások számának megváltoztatásához az 1-es, a játékhoz a 2-es, míg a kilépéshez a 3-as billentyűt kell lenyomni. Ugyanitt láthatjuk az eddig lejátszott játszmáink alakulását is.

A dobások számának megváltoztatásánál (1. menüpont) az 1, 3 és 5 a lehetőségek. Bármely más billentyű lenyomása egy figyelmeztető sipjelet von maga után. Miután leütöttük a megfelelő billentyűt, egy tetszőleges gomb lenyomásával visszatérhetünk a menühöz, ahol már látjuk is az újonnan beállított dobásszámot a Játék menüpont után.

A Játék-ot (2) választva következhetnek a dobások. Először a gép dob, majd mi dobhatunk. A „kocka pörgését” a szóköz billentyűvel Ön állíthatja meg. Jobb oldalon látható, hogy az adott dobást ki nyerte, majd az összes dobás után a gép kiírja, hogy ki nyerte a játszmát. Egy tetszőleges billentyűvel térhetünk vissza a menühöz, ahol már láthatjuk is a nyert és veszített játszmák számanak megváltozott, új állását.

A Kilépés menüpontot (3) választva a játékot befejeztük, és a programból kiléptünk.

Jó szórakozást:

Zseni Alfonz

10.7. A teljes programlista

Végezetül íme a teljes programlista a változtatásokkal együtt:

```

Program KockaJatek;
Uses Crt;
Var
    dobas,nyert,vesztett : integer;
    mp : char;

Procedure EredmenyKiir;
Begin
    GotoXY(50,1);
    Write('eddig nyertél ',nyert,' játszmát');
    GotoXY(50,2);
    Write('és vesztettél ',vesztett,' játszmát');
End;

Procedure Menupontok;
Begin
    ClrScr;
    EredmenyKiir;
    GotoXY(20,9);
    WriteLn('1. Dobások számának beállítása');
    GotoXY(20,11);
    WriteLn('2. Játék (dobások száma: ',dobas,')');
    GotoXY(20,13);
    WriteLn('3. Kilépés');
    GotoXY(20,16);
    Write('Melyiket választod? (1,2,3)');
End;

Procedure Varakozas;
Var
    c : char;

Begin
    GotoXY(1,25);
    Write('Nyomj egy billentyűt, és visszatérhetsz a menühöz');
    c:=ReadKey;
End;

Procedure DobasSzam;
Var
    ujdobas : char;
    kod : integer;

Begin
    ClrScr;
    WriteLn('A dobások aktuális száma: ',dobas);
    Write('Hány dobás legyen egy játékban? (1,3,5)');
    Repeat
        ujdobas:=ReadKey;
        If not (ujdobas in ['1','3','5'])

```

```

        then begin
            Sound(880);
            Delay(100);
            Nosound;
        end;
    Until ujdobas in ['1','3','5'];
    Val(ujdobas,dobas,kod);
    WriteLn;
    WriteLn('A dobások új száma: ',dobas);
    Varakozas;
End;

Procedure Jatek;
Var
    i,gd,jd,gny,jny : integer;
    c : char;

Begin
    Randomize;
    ClrScr;
    gny:=0;
    jny:=0;
    For i:=1 to dobas do begin
        c:=#0;
        gd:=Random(6)+1;
        GotoXY(1,i*2);
        Write(i,'. dobásom: ',gd);
        GotoXY(20,i*2);
        Write(i,'. dobásod: ');
        Repeat
            jd:=Random(6)+1;
            GotoXY(33,i*2);
            Write(jd);
            If KeyPressed then c:=ReadKey;
            Delay(50);
        Until c=' ';
        GotoXY(40,i*2);
        If gd>=jd
            then begin
                WriteLn('A gép nyert');
                Inc(gny);
            end
            else begin
                WriteLn('Te nyertél');
                Inc(jny);
            end;
    end;
    end;
    GotoXY(1,15);
    If gny>jny
        then begin
            Write('Ezt a játékot sajnos ');
            Write(gny,'',jny);
            WriteLn('arányban a gép nyerte');
            Inc(vesztett);
        end

```



```

else begin
    Write('Gratulálok! Ezt a játékot ');
    Write(jny,':',gny);
    WriteLn(' arányban Te nyerted');
    Inc(nyert);
end;
Varakozas;
End;

BEGIN
dobas:=3;
nyert:=0;
vesztett:=0;
Repeat
    Menupontok;
    Repeat
        mp:=ReadKey
    Until mp in ['1'..'3'];
    Case mp of
        '1' : DobasSzam;
        '2' : Jatek;
    end;
Until mp='3'
END.

```

10.8. Feladatok

A példaprogram rengeteg továbbfejlesztési lehetőséget kínál, amelyek nagyon jó és élvezetes programozási gyakorlatra adnak lehetőséget. Fontos, hogy mindig a programterv módosításával kezdjük, és csak utána Módosítsuk magát a programot!

Feladatok:

202. Készítse el a példaprogram szebbített változatát! Elsősorban a színek alkalmazásával, de lehet változtatni a kiírásokon is.
203. Készítse el a program grafikus képernyős változatát is!
204. Építsen be egy új menüpontot, amely a játék leírását jeleníti meg!
205. Tegye lehetővé a döntetlen játszmákat úgy, hogy számolja ezeket is! (Ekkor már nincs értelme a dobások számának csak a páratlan számokra való leszűkítésének sem.)
206. Tegye lehetővé a döntetlen játszmákat úgy, hogy ha egy játszma döntetlenre végződött, akkor addig dobál újra, amíg csak el nem dől!
207. Módosítsa a programot úgy, hogy egyszerre két kockával folyjon a játék!
208. Készítsen el egy új játékot a fentiek felhasználásával, saját ötlete alapján! (Lehet ez a kigyós játék is, vagy bármi más.)

11. Mellékletek

11.1. Az ASCII kódtábla

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0	1	2	3	4	5	6	7	8	9	:	<	=	>	?	
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
p	q	r	s	t	u	v	w	x	y	z	{		}	~	
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
Ç	ü	é	ä	å	ä	ç	ê	ë	ë	ì	í	î	ï	À	Á
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
É	æ	Æ	ö	ä	ö	ü	ü	ý	ÿ	ÿ	ç	£	¥	€	ƒ
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
ı	ó	ü	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ	ñ
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
L	ı	T	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
α	β	Γ	π	Σ	σ	μ	τ	φ	θ	Ω	δ	ω	ø	ε	ñ
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

11.2. A Turbo Pascal védett szavai

A védett szavak olyan azonosítók, amelyeket a programozó nem használhat saját azonosítóként. A védett szavak a programlistában eltérő színnel jelennek meg (alapértelmezés szerint a sárga színnel írt utasítások közül fehérrel van kiemelve). A könyvben félkövér betűkkel vannak szedve. A védett szavak a következők:

and	goto	program
asm	if	record
array	implementation	repeat
begin	in	set
case	inherited	shl
const	inline	shr
constructor	interface	string
destructor	label	then
div	library	to
do	mod	type
downto	nil	unit
else	not	until
end	object	uses
exports	of	var
file	or	while
for	packed	with
function	procedure	xor

11.3. A Turbo Pascal kulcsszavai és utasításai

Az alábbi lista tartalmazza a Turbo Pascal nyelv kulcsszavait és utasításait. A felsorolás ABC rendben van, és az objektum nevén kívül feltünteti a típusát (K – kulcsszó, U – utasítás), a leírását, valamint azt az oldalszámot, az ismertetése található.

Megnevezés	Típus	Leírás	Oldal-szám
Array .. of	K	Vektor vagy tömb típus deklarálása	121
Begin	U	Egy szakasz kezdetét jelzi	45
Case .. of .. else .. end	U	Elágazás esetszétválasztással	98
Const	K	Állandók deklarálása	77
End	U	Egy szakasz végét jelzi	45
File of	K	Fájl típus deklarációja	129
For .. to/downto .. do	U	Számlálós ciklus	79
Function	K	Függvény megadása	118
Goto	U	Ugrás a programban	100
If .. then .. else	U	Elágazás	94
Implementation	K	Unitban az implementációs rész kezdete	183
Interface	K	Unitban a deklarációs rész kezdetét jelzi	183
Label	K	Címkék deklarálása	100
Object	K	Objektum deklarációja	–
Procedure	K	Eljárás megadása	114
Program	K	A program kezdetét jelzi (elhagyható)	45
Record	K	Rekord deklarációja	125
Repeat .. Until	U	Hátultesztelős ciklus	105
Set of	K	Halmaz deklarációja	130
Type	K	Típusdeklaráció	124
Unit	K	Unit kezdetét jelzi	183
Uses	K	Felhasználandó unitok megadása	42
Var	K	Változó deklarálása	66
While .. do	U	Előtesztelős ciklus	105
With	U	Rekordok „kinyitása”	128

11.4. A Turbo Pascal nyelv eljárásai és függvényei

Az alábbi lista tartalmazza a Turbo Pascal nyelv azon eljárásait és függvényeit, amelyek a könyvben szerepelnek, és sok olyan is, amelyről a könyvben nincs bővebb leírás, de a tárgyalt témákhoz hozzátartozó kiegészítésnek lehet tekinteni. A felsorolás ABC rendben van, és az eljárás vagy függvény nevén kívül feltüntetni a típusát (E – eljárás, F – függvény), a leírását, azt, hogy melyik unitban található, valamint azt az oldalszámot, ahol az illető alprogram ismertetése található. Azok az eljárások, ahol a unitot nem adtam meg, a System unitba tartoznak.

Megnevezés	Típus	Leírás	Unit	Oldal
Abs	F	Abszolút érték		146
Append	E	Szövegfájl megnyitása		166
Arc	E	Körív rajzolása	Graph	176
ArcTan	F	Arkusz tangens függvény		149
Assign	E	Fájl-változó hozzárendelése fizikai fájlhoz		162
AssignCrt	E	Szövegfájl hozzárendelése a képernyőhöz	Crt	–
Bar	E	Kitöltött téglalap rajzolása	Graph	179
Bar3D	E	Téglatest rajzolása	Graph	179
BlockRead	E	Olvasás nem tipizált fájlból		165
BlockWrite	E	Írás nem tipizált fájlba		165
ChDir	E	Könyvtárváltás		168
Chr	F	ASCII-kód alapján karakter		160
Circle	E	Kör rajzolása	Graph	
ClearDevice	E	Grafikus képernyő törlése	Graph	173
ClearViewport	E	Aktív ablak törlése	Graph	–
Close	E	Fájl bezárása		163
CloseGraph	E	Grafikus mód lezárása	Graph	172
ClrEol	E	Sor törlése a kurzortól kezdve	Crt	–
ClrScr	E	Képernyőtörlés	Crt	50
Concat	E	Szövegek összefűzése		–
Copy	F	Szöveg másolása		154
Cos	F	Koszinusz függvény		149
Dec	E	Értéksökkentés		157
Delay	E	Várakozás	Crt	91
Delete	E	Törlés a szövegből		155

Megnevezés	Típus	Leírás	Unit	Oldal
DelLine	E	Sor törlése	Crt	–
DetectGraph	E	Grafikus driver és mód megállapítása	Graph	172
Dispose	E	Dinamikus változó lefoglalása		187
DrawPoly	E	Poligon rajzolása	Graph	177
Ellipse	E	Ellipszis rajzolása	Graph	177
Eof	F	Fájlvég-teszt		164
Eoln	F	Sorvég-teszt		167
Erase	E	Fájl törlése		165
Exclude	E	Új elem beillesztése halmazba		130
Exit	E	Blokkból való kilépést tesz lehetővé		–
Exp	F	Exponenciális függvény		150
FilePos	F	Rekordmutató pozícióját adja		163
FileSize	F	Fájl komponenseinek számát adja		163
FillEllipse	E	Kitöltött ellipszis rajzolása	Graph	179
FillPoly	E	Kitöltött poligon rajzolása	Graph	180
FloodFill	E	Terület kifestése	Graph	179
Frac	F	Törtresz függvény		147
FreeMem	E	Dinamikus változó felszabadítása		187
GetArcCoords	F	A legutolsó kör vagy körív adatait adja vissza	Graph	–
GetBkColor	F	Aktuális háttérszín lekérdezése	Graph	173
GetColor	F	Aktuális rajzoló szín lekérdezése	Graph	173
GetDir	F	Aktuális könyvtár lekérdezése		168
GetDriverName	F	A grafikus eszköz nevét adja vissza	Graph	–
GetFillPattern	F	Az aktuális felhasználói mintát adja vissza	Graph	–
GetFillSettings	F	Az aktuális kitöltő szint és mintát adja vissza	Graph	–
GetGraphMode	F	A grafikus mód kódját adja vissza	Graph	–
GetLineSettings	F	Az aktuális vonalstílust és vastagságot adja	Graph	–
GetMaxColor	F	A legmagasabb kódú szint adja meg	Graph	–
GetMaxMode	F	A lehetséges grafikus módok száma	Graph	–
GetMaxX	F	A képernyő utolsó oszlopának sorszáma	Graph	173
GetMaxY	F	A képernyő utolsó sorának sorszáma	Graph	173
GetMem	F	Dinamikus változó lefoglalása		187
GetModeName	F	A grafikus mód nevét adja vissza	Graph	–

Megnevezés	Típus	Leírás	Unit	Oldal
GetPixel	E	Képpont színét adja	Graph	174
GetTextSettings	F	Az aktuális szövegbeállítást adja vissza	Graph	–
GetViewSettings	F	Az aktuális ablak adatait adja vissza	Graph	–
GetX	F	CP pozíciójának X koordinátája	Graph	174
GetY	F	CP pozíciójának Y koordinátája	Graph	174
GotoXY	E	A kurzort pozicionálja a megadott koordinátákra	Crt	50
GraphDefaults	E	Újrainicializálja a grafikus rendszert	Graph	–
Halt	E	Program befejezése	–	–
High	F	A típus maximális értékét adja	–	–
HighVideo	E	Intenzív kírátás	Crt	–
Inc	E	Érték növelése	–	157
Include	E	Elem törlése halmazból	–	130
InitGraph	E	Grafikus képernyő inicializálása	Graph	171
Insert	E	Beszúrás szövegbe	–	155
InsLine	E	Sor beszúrása	Crt	–
Int	F	Egészrész függvény	–	146
IOResult	F	I/O hiba kódját adja	–	169
KeyPressed	F	Billentyű lenyomásának érzékelése	Crt	–
Length	F	Szöveg hossza	–	151
Line	E	Vonal rajzolása	Graph	176
LineRel	E	Vonal rajzolása relatív koordinátákkal	Graph	176
LineTo	E	Vonal rajzolása a CP-től kezdve	Graph	176
Ln	F	Természetes alapú logaritmus függvény	–	150
Low	F	Típus minimális értékét adja	–	–
LowVideo	E	Alacsony intenzitású kírátás	Crt	–
Mark	E	Heap állapotának elmentése	–	–
MaxAvail	F	Legnagyobb szabad blokk mérete	–	187
MemAvail	E	Szabad terület nagysága a Heap-ben	–	187
MkDir	E	Könyvtár létrehozása	–	168
MoveRel	E	CP pozíciójának megadása	Graph	174
MoveTo	E	CP pozíciójának relatív megadása	Graph	174
New	E	Dinamikus változó lefoglalása	–	187
NormVideo	E	Normál intenzitású kírátás	Crt	–
NoSound	E	Hangszóró kikapcsolása	Crt	–
Odd	F	Páros vagy páratlan?	–	158
Ord	F	Sorszámozott típus sorszámát adja	–	157

Megnevezés	Típus	Leírás	Unit	Oldal
OutText	E	Szöveg írása	Graph	181
OutTextXY	E	Szöveg írása adott helyre	Graph	181
Pi	F	π konstans, értéke: 3.14159265...	–	146
PieSlice	E	Körcikk rajzolása	Graph	179
Pos	F	Keresés szövegben	–	154
Pred	F	Előző érték	–	157
PutPixel	E	Képpont kigyújtása	Graph	174
Random	F	Véletlenszám-generátor	–	148
Randomize	E	Véletlenszám-generátor kezdőértékének beáll.	–	148
Read	E	Adatbekérés	–	71
Read	E	Olvasás fájlból	–	164
ReadKey	F	Egyetlen karakter bekérése a billentyűzetről	Crt	102
ReadLn	E	Adatbekérés soremeléssel	–	71
ReadLn	E	Beolvasás szövegfájlból	–	167
RecTangle	E	Téglalap rajzolása	Graph	176
Release	E	Heap terület felszabadítása	–	–
Rename	E	Fájl átnevezése	–	167
Reset	E	Fájl megnyitása	–	162
Rewrite	E	Új fájl megnyitása	–	162
RmDir	E	Könyvtár törlése	–	168
Round	F	Egészre kerekítés	–	147
Sector	E	Ellipsziscikkely rajzolása	Graph	180
Seek	E	Rekordmutató pozicionálása	–	163
SeekEof	F	Fájlvég-teszt szövegfájlban	–	167
SeekEoln	F	Sorvég-teszt szövegfájlban	–	167
SetActivePage	E	Aktív ablak kijelölése	Graph	–
SetBkColor	E	Aktuális háttérszín megadása	Graph	173
SetColor	E	Aktuális rajzoló szín megadása	Graph	173
SetFillPattern	E	Saját minta definiálása	Graph	–
SetfillStyle	E	Kitöltő minta és szín megadása	Graph	178
SetGraphMode	E	Grafikus mód megváltoztatása	Graph	–
SetLineStyle	E	Vonalstílus és -vastagság beállítása	Graph	175
SetTextJustify	E	Szöveg igazításának megadása	Graph	181
SetTextStyle	E	Szöveg stílusának és méretének a megadása	Graph	180
SetUserCharSize	E	Az aktuális betűtípus méretaránya-	Graph	–

Megnevezés	Típus	Leírás	Unit	Oldal
		nak megváltoztatása		
SetViewport	E	Aktív ablak definiálása	Graph	–
SetVisualPage	E	Látható képernyő megadása	Graph	–
SetWrtMode	E	Egyenesek rajzolás módját adja meg	Graph	175
Sin	F	Színusz függvény		149
SizeOf	F	Változó vagy típus bájtokban mért helyfoglalása		166
Sound	E	Hanggenerátor	Crt	–
Sqr	F	Négyzetre emelés		144
Sqrt	F	Négyzetgyökvonás		145
Str	E	Szám átalakítása szöveggé		158
Succ	F	Következő érték		157
TextBackground	E	Karakteres képernyőn háttérszín megadása	Crt	56
TextColor	E	Karakteres képernyőn karakterszín megadása	Crt	56
TextHeight	F	Szöveg képpontokban mért magasságát adja	Graph	–
TextMode	E	Szöveges mód beállítása	Crt	–
TextWidth	F	Szöveg képpontokban mért hosszát adja	Graph	–
Trunc	F	Egészrész függvény		147
Truncate	E	Fájl végének levágása		165
UpCase	F	Nagybetűssé alakít		155
Val	E	Szöveg átalakítása számmá		158
WhereX	F	Kurzor X koordinátája	Crt	–
WhereY	F	Kurzor Y koordinátája	Crt	–
Window	E	Aktív ablak definiálása karakteres képernyőn	Crt	–
Write	E	Képernyőre ír		45
Write	E	Fájlba írás		164
Writeln	E	Képernyőre ír, majd sort emel		48
Writeln	E	Szöveg fájlba írás		167

11.5. Fordítási hibák jegyzéke

1. Betelt a memória
2. Azonosító hiányzik
3. Ismeretlen azonosító (az azonosító nem deklarált)
4. Dupla azonosító (két azonos nevű azonosítót deklaráltunk)
5. Szintaktikai hiba (helytelen karakter használata)
6. Hiba a valós állandóban
7. Hiba az egész állandóban (pl. olyan értéket adunk a konstansnak, ami már kívül esik a típus értelmezési tartományán)
8. Túl hosszú string állandó (maximum 255 karakter lehet)
10. Váratlan programvég. (A begin-ek és end-ek száma nem azonos.)
11. Túl hosszú sor a programban. (maximum 127 karakter lehet)
12. Típusazonosító hiányzik
13. Túl sok fájl van megnyitva (az maximális megnyitható állományok számát a CONFIG.SYS állományban a FILES=n sorban adhatjuk meg)
14. Helytelen fájlnev
15. A fájl nem létezik
16. A lemez tele van
17. Helytelen fordító direktíva
18. Túl sok fájl használunk a fordításnál
19. Nem definiált típus a mutató (pointer) definíciójában
20. A változó azonosítója hiányzik
21. Hibás típusdeklaráció
22. Túl nagy méretű struktúra (leggyakrabban a tömböt túl nagyra deklaráltuk; maximális helyfoglalása 65520 bájt lehet)
23. Egyszerű típus túlcordulása
24. Fájl komponense nem lehet fájl vagy objektum típusú
25. Helytelen szöveg hossz a deklarációban (max 255 lehet)
26. Típus inkompatibilitás kifejezésekben, értékadásnál vagy paraméterátadásnál
27. Helytelen résztartomány típus
28. Az alsó határ nagyobb, mint a felső
29. Sorszámozott típus hiányzik (pl. ciklusváltozó csak sorszámozott típusú lehet)
30. Egész típusú állandó hiányzik
31. Állandó hiányzik
32. Egész vagy valós típusú állandó hiányzik
33. Mutató típusú azonosító hiányzik
34. Helytelen a függvény eredményének (visszatérési értékének) típusa
35. Címke azonosító hiányzik
36. BEGIN hiányzik
37. END hiányzik
38. Egész típusú kifejezés hiányzik
39. Sorszámozott típusú kifejezés hiányzik
40. Logikai típusú kifejezés hiányzik
41. Az operandus típusa helytelen (nem felel meg az operátornak)
42. Hiba a kifejezésben

43. Illegális hozzáférések
44. Rekordokban mezőazonosító hiányzik
45. Object fájl túl hosszú (maximum 64K lehet)
46. Nem definiált külső (external) eljárás
47. Helytelen rekord az object fájlban
48. Kódsegmens (vagyis a program) túl hosszú (maximum 65520 bájtt lehet).
Unitok használatával a hiba elhárítható
49. Adatszegmens túl nagy (vagyis túl sok az adataink helyfoglalása)
(maximum 65520 bájtt lehet). Dinamikus változók használatával a probléma megoldható
50. DO hiányzik
51. Helytelen PUBLIC definíció
52. Helytelen EXTERN definíció
53. Túl sok EXTERN definíció
54. OF hiányzik
55. INTERFACE hiányzik
56. Helytelen hivatkozás
57. THEN hiányzik
58. TO vagy DOWNTON hiányzik
59. Nem definiált FORWARD
- 60.
61. Helytelen típuskijelölés
62. Nullával való osztás
63. Helytelen fájl típus
64. Nem lehet változót a fájlba írni, vagy a fájlból olvasni
65. Pointer típusú változó hiányzik
66. String típusú változó hiányzik
67. Szöveges kifejezés hiányzik
68. Körkörös hivatkozás unit-ok között
69. Helytelen unitnév
70. Helytelen unit-változat: az utolsó fordítás óta a unit tartalma változott; újra kell fordítani
71. Verem túlsordulás (túl sok az egymásba ágyazott eljárás vagy függvényhívás)
72. A unit fájl formátuma hibás
73. IMPLEMENTATION hiányzik
74. CASE szerkezetekben a változó típusa és a konstans típusa nem kompatibilis
75. Rekord vagy objektum típusú változó hiányzik
76. A konstans értéke túlsordult a típus értelmezési tartományából
77. Fájl típusú változó hiányzik
78. Pointer típusú kifejezés hiányzik
79. Egész vagy valós típusú kifejezés hiányzik
80. A hivatkozott címke nincs az aktuális blokkban
81. Címkedefiníció már létezik
82. Nem definiált címke
83. Helytelen a @ operátor argumentuma

84. UNIT hiányzik
85. „:” hiányzik
86. „:” hiányzik
87. „:” hiányzik
88. „(” hiányzik
89. „)” hiányzik
90. „=” hiányzik
91. „:=” hiányzik
92. „(” vagy „(.” hiányzik
93. „)” vagy „.)” hiányzik
94. „.” hiányzik
95. „...” hiányzik
96. Túl sok változó (a változók helyfoglalása maximum 64K lehet)
97. Helytelen ciklusváltozó (csak sorszámozott típus lehet)
98. Egész típusú változó hiányzik
99. Tipizált konstans nem lehet fájl vagy eljárás típusú
100. Helytelen stringhossz
101. Helytelen a mezők sorrendje
102. String típusú állandó hiányzik
103. Egész vagy valós típusú változó hiányzik
104. Sorszámozott típusú változó hiányzik
105. INLINE hiba
106. Karakteres kifejezés hiányzik
107. Túl nagy a program helyfoglalása (egy EXE-fájl a Turbo Pascal-ban max. 64K lehet)
108. Túlsordulás az aritmetikai műveletben
109. A Break vagy Continue eljárások nem használhatók a cikluson kívül
110. Nyomkövetéshez szükséges információs tábla túlsordult
- 111.
112. CASE konstans kívül esik a lehetséges értékhatáron
113. Hiba az utasításban
114. Nem lehet meghívni a megszakítást
- 115.
116. Koprocesszor szükséges (és persze az {\$N+} fordítási direktíva) a fordításhoz
117. A megadott cím nem létezik a Search/Find error használata esetén
118. Include (beszárt) fájl itt nem állhat
119. Inherited kulcsszó helytelen alkalmazása
- 120.
121. Helytelen változóhasználat (nem tömb típus indexelése, nem rekord típus esetén mezőhivatkozás vagy olyan mutató használata, amely nem mutató típusú)
122. Helytelen változó-hivatkozás
123. Túl sok szimbólum (maximum 64K lehet)
124. A végrehajtási rész túl nagy (maximum 24K lehet)
- 125.
126. Fájl paraméter csak változó lehet
127. Túl sok feltételes szimbólum

128. Feltételes fordítási direktíva rossz helyen van
129. ENDIF direktíva hiányzik
130. Hiba a feltételes direktíva inicializálásában
131. Eljárás vagy függvény feje különbözik a deklarálttól
- 132.
133. A kifejezés kiértékelhetetlen
134. A kifejezés helytelenül végződik
135. Helytelen formátum megadás
136. Helytelen indirekt hivatkozás
137. Összetett típusú változó itt nem állhat
138. A kifejezés kiértékelése lehetetlen a Sytem unit nélkül
139. A szimbólum nem hozzáférhető
140. Helytelen lebegőpontos művelet
141. Overlay program nem fordítható memóriába
142. Pointer vagy eljárás típusú változó hiányzik
143. Helytelen eljárás vagy függvény hivatkozás
144. A unit nem átlapolható
145. Túl sok az egymásba ágyazott elem
146. Fájl nem hozzáférhető (pl. csak olvasható fájlba akartunk írni)
147. Objektum típus hiányzik
148. Lokális objektum típus nem állhat itt, vagy az objektumot eljárás és függvény nélkül definiáltuk
149. VIRTUAL hiányzik
150. Metódus azonosító hiányzik
151. A Constructor nem lehet virtuális
152. Constructor azonosító hiányzik
153. Destructor azonosító hiányzik
154. A Fail eljárás csak konstruktorból hívható
155. Assembler kódban helytelen az operandusok kombinációja
156. Memóriahivatkozás hiányzik
- 157.
158. Helytelen regiszter kombináció
159. A 286-os kód használatához a [C+] fordító direktíva szükséges
160. Helytelen szimbólumhivatkozás
161. Kódgenerálási hiba
162. ASM hiányzik
163. A dinamikus metódus indexe már létezik egy másik metódusnál
164. Duplikált forrás-azonosító
165. Duplikált vagy helytelen kimeneti index (1-32767 lehet)
166. Eljárás vagy függvény azonosítója hiányzik
167. A szimbólum nem exportálható
168. Ilyen névvel már létezik export
169. A futtatható fájl fejléce túl hosszú
170. Túl sok szegmens (csak védett módban)

11.6. Futási hibák jegyzéke

Futási hiba esetén a program végrehajtása megszakad, és a képernyőn megjelenik a hibaüzenet: „Run-time error N at xxxx:yyyy”, ahol N a hiba kódja, xxxx:yyyy pedig a hiba helye (szegmenscim:offsztetcm).

- | | |
|---|--|
| 1. Nem létező DOS függvény hívása | 200. Nullával való osztás |
| 2. Fájl nem létezik | 201. Túlcsordulás |
| 3. Útvonal (könyvtár) nem létezik | 202. Verem (STACK) túlcsordulás |
| 4. Túl sok az egyszerre megnyitott fájl | 203. HEAP túlcsordulás |
| 5. A fájl nem hozzáférhető | 204. Helytelen pointer művelet |
| 6. Helytelen fájlkezelés | 205. Lebegőpontos túlcsordulás |
| 12. Helytelen fájl-hozzáférési kód | 206. Lebegőpontos alulcsordulás |
| 15. Helytelen meghajtó-szám | 207. Helytelen lebegőpontos művelet |
| 16. Az aktuális könyvtár nem törölhető | 208. Az átlapolás-kezelő nincs installálva |
| 17. Az állomány nem átnevezhető | 209. Olvasási hiba átlapolt fájlban |
| 18. Nincs több fájl | 210. Az objektum nem inicializált |
| 100. Lemezolvasási hiba | 211. Absztrakt metódus hívása |
| 101. Lemezírási hiba | 212. Stream-regisztrációs hiba |
| 102. A fájl nincs hozzárendelve logikai fájlhoz | 213. Kollekción-index túlcsordulása |
| 103. A fájl nincs megnyitva | 214. Kollekción túlcsordulása |
| 104. A fájl nincs megnyitva olvasásra | 215. Aritmetikus túlcsordulás |
| 105. A fájl nincs megnyitva írásra | |
| 106. Helytelen numerikus formátum | |
| 150. Írsvédett lemez | |
| 151. Eszközvezérlő belső hibája | |
| 152. A meghajtó nem üzemképes | |
| 154. CRC hiba az adatokban | |
| 156. Fájl-pozicionálási hiba a lemezen | |
| 157. Ismeretlen típusú háttértároló | |
| 158. Nem létező szektor | |
| 159. A nyomtatóból kifogyott a papír | |
| 160. Írasi hiba az eszközön | |
| 161. Olvasási hiba az eszközön | |
| 162. Hardver meghibásodás | |



