SZÁMÍTÁSTECHNIKA I.

A projekt címe: „Egységesített Jármű- és mobilgépek képzés- és tananyagfejlesztés”

A megvalósítás érdekében létrehozott konzorcium résztvevői:

KECSKEMÉTI FŐISKOLA

BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM

AIPA ALFÖLDI IPARFEJLESZTÉSI NONPROFIT KÖZHASZNÚ KFT.

Fővállalkozó: TELVICE KFT.

TELVICE

Budapesti Műszaki és Gazdaságtudományi Egyetem
Közlekedésmérnöki Kar

Írta:
ARADI SZILÁRD

BÉCSI TAMÁS

GYENES KÁROLY

PÉTER TAMÁS

Lektorálta:
VARGA BALÁZS

SZÁMÍTÁSTECHNIKA I.

Egyetemi tananyag

2011

COPYRIGHT: © 2011-2016, Aradi Szilárd, Dr. Bécsi Tamás, Dr. Gyenes Károly, Dr. Péter Tamás, Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedés-mérnöki Kar

LEKTORÁLTA: Varga Balázs

Creative Commons NonCommercial-NoDerivs 3.0 (CC BY-NC-ND 3.0)
A szerző nevének feltüntetése mellett nem kereskedelmi céllal szabadon másolható, terjeszthető, megjelentethető és előadható, de nem módosítható.

ISBN 978-963-279-594-2
KÉSZÜLT: a Typotex Kiadó gondozásában
FELELŐS VEZETŐ: Votisky Zsuzsa

TÁMOGATÁS:
Készült a TÁMOP-4.1.2.A/2-10/1-2010-0018 számú, „Egységesített jármű- és mobil-gépek képzés- és tananyagfejlesztés” című projekt keretében.

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638

A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

KULCSSZAVAK:
Pascal, Object Pascal, algoritmus, Turbo Pascal, programtervezés, programnyelvek, szintaktika, szemantika, OOP

ÖSSZEFOGLALÁS:
E jegyzet célja, hogy a leendő mérnökök ne csak a számítástechnikai eszközök felhasználói, de azok működésének értői is legyenek. Ha a hallgató belelát a számítógép belső felépítésébe, működésének alapvető ismeretében hatékony programok készítésére lesz képes.
Elsőként a nyelv-független algoritmus készítéssel foglalkozunk, majd áttérünk egy jól felhasználható algoritmikus nyelv – az Object Pascal - ismertetésére.
A jegyzet mérnökhallgatók számára készült. A számítógépek programozása nagymértékben gyakorlati tárgy. Ezért jegyzetünk az előadásokhoz igazodva egy elméleti, majd egy jelentős gyakorlati részt tartalmaz. Ez a második rész kidolgozott példák gyűjteménye, amely a tényleges programírási gyakorlatot hivatott segíteni a Pascal nyelv szintaktikája alapján.
Az egyes fejezetek végén az elméleti anyaghoz szorosan kapcsolódó feladatok, és a tanulást segítő ellenőrző kérdések találhatók.

Tartalomjegyzék

TARTALOMJEGYZÉK.....1

1 SZÁMÍTÁSTECHNIKAI ALAPOK.....8

1.1 ADATOK MEGJELENÉSI FORMÁI A SZÁMÍTÓGÉPBEN.....8
1.1.1 Egész típusú adatok.....8
1.1.2 Valós típusú adatok.....13
1.1.3 Karakterek megjelenési formái.....18
1.1.4 Szöveges adatok formái.....22
1.2 BINÁRIS MŰVELETEK.....24
1.2.1 Additív műveletek bináris kódban.....24
1.2.2 Additív műveletek BCD kódban.....26
1.2.3 Bináris szorzás.....28
1.2.4 Bináris osztás.....29
1.2.5 Az adatok védelme.....31
1.2.6 Paritás ellenőrzés.....31
1.2.7 Ellenőrző összeg.....32
1.2.8 Polinomos adatvédelem.....33
1.2.9 Modulo 2 algebra.....34
1.2.10 Bináris mező és polinom.....35
1.2.11 Adatvédelmi algoritmus.....36
1.3 AZ ARITMETIKAI MŰVELET ELLENŐRZÉSE.....39

2 PROGRAMNYELVEK OSZTÁLYOZÁSA.....42

2.1 GÉPI KÓD.....42
2.2 ASSEMBLER NYELV.....42
2.3 MAGAS SZINTŰ PROGRAMNYELV.....43
2.4 SZIMBOLIKUS NYELV.....44

3 A PROGRAMKÉSZÍTÉS LÉPÉSEI.....46

3.1 A PROBLÉMA MATEMATIKAI MEGFOGALMAZÁSA.....46
3.2 ALGORITMUSSZERKESZTÉS.....48
3.2.1 Folyamatábra szimbólumok.....50
3.2.2 Elágazásos programok.....52
3.2.3 Egyszeres ciklus.....53
3.2.4 Kétszeres ciklus.....54

4 A PASCAL NYELV TÖRTÉNETE.....56

4.1 A PASCAL NYELV GENERÁCIÓI.....56
4.2 A PASCAL KÜLÖNBÖZŐ SZINTJEI.....57
4.2.1 Hivatkozási szint.....57
4.2.2 Publikációs szint.....57
4.2.3 Gépi reprezentáció.....57
4.2.4 Fejlesztői környezet.....58

5 A PASCAL NYELV STRUKTURÁLIS FELÉPÍTÉSE 60
5.1 ÉPÍTŐELEMEK 60
5.1.1 Számok 60
5.1.2 Betűk 60
5.1.3 Szimbólumok 60
5.1.4 Karakterek 61
5.1.5 Azonosítók 61
5.1.6 Adattípusok 62
5.1.7 Konstansok 64
5.1.8 Standard függvények 65
5.1.9 Címke 66
5.2 KIFEJEZÉSEK 67
5.3 UTASÍTÁSOK 71
5.3.1 Értékadó utasítás 71
5.3.2 Deklaráció utasítás 71
5.3.3 Kommentár utasítás 73
5.3.4 Feltételes utasítás 73
5.3.5 Vezérlésátadó utasítás 76
5.3.6 Típus definíció 76
5.3.7 Ciklus utasítások 77
5.3.8 Case utasítás 80
5.3.9 Zárójelek használata 83
5.3.10 Rekord adattípus használata 84
5.3.11 With utasítás 87
5.3.12 Input-Output utasítások 88
5.4 PROGRAM SZEGMENTÁCIÓ 90
5.4.1 Függvények 91
5.4.2 Függvény típusú változók 95
5.4.3 Eljárások 96
5.4.4 Lokális szegmens használata. 102
5.5 HALMAZOK 105
5.6 MUTATÓK, DINAMIKUS ADATOK 110
5.7 FILE KEZELÉS A PASCAL NYELVBEN 118
5.7.1 Típusos file 122
5.7.2 Text file 131
5.7.3 Típusnélküli file 135
5.8 GRAFIKA HASZNÁLATA A PASCAL KÖRNYEZETBEN 138
5.9 OBJEKTUMOK 139
5.9.1 Egyszerű statikus objektum 140
5.9.2 Dinamikus objektum 146
5.9.3 Objektumok öröklési tulajdonsága 150
5.10 A PASCAL UNIT 154
5.10.1 Standard unit 154
5.10.2 Unitok definiálása és használata 154
5.10.3 A változók, modulok hatásköre 157
5.11 KÜLSŐ MODULOK (DLL) 162
5.11.1 DLL létrehozása 162

6 FELADATGYŰJTEMÉNY 166
6.1 PÉLDÁK AZ ALGORITMUS SZERKESZTÉSRE 166
6.1.1 Elágazásos algoritmusok 166
6.1.2 Egyszeres ciklust használó feladatok 170
6.1.3 Kétszeres ciklust használó feladatok 175
6.2 PÉLDÁK A PASCAL ADATTÍPUSOK GYAKORLÁSÁRA 180
6.3 KIFEJEZÉSEK GYAKORLÁSA 183
6.4 VÁLTOZÓK AKTUÁLIS ÉRTÉKÉNEK MEGHATÁROZÁSA 185
6.5 PÉLDÁK KÜLÖNBÖZŐ ÉRTÉKADÁSRA 186
6.6 FELTÉTELES UTASÍTÁSOK GYAKORLÁSA 186
6.7 PROGRAMKAPCSOLÓ (CASE UTASÍTÁS) GYAKORLÁSA 188
6.8 PÉLDÁK CIKLUSOK HASZNÁLATÁRA 189
6.8.1 for – to – do ciklusok 189
6.8.2 while – do ciklusok 193
6.8.3 Repeat - until ciklusok 197
6.9 REKORD ADATTÍPUS GYAKORLÁSA 199
6.10 PÉLDÁK FÜGGVÉNY HASZNÁLATÁRA 203
6.11 PÉLDÁK ELJÁRÁS HASZNÁLATÁRA 206
6.12 HALMAZOK GYAKORLÁSA 210
6.13 FILE KEZELÉS GYAKORLÁSA 212
6.13.1 Skalár típusú file (egész, valós, karakter) 212
6.13.2 Tömb és rekord típusú file 214
6.13.3 Random file kezelés 218
6.13.4 Szöveg file kezelése 220
6.13.5 Típus nélküli file kezelése 221
6.14 POINTEREK HASZNÁLATA 221
6.15 OBJEKTUMOK KÉSZÍTÉSE 230
6.15.1 Dinamikus objektum 232
6.15.2 Ős- és leszármazott objektum 234

TÁBLÁZATJEGYZÉK 243

IRODALOMJEGYZÉK 245

1 Számítástechnikai alapok

E fejezet célja, hogy a leendő mérnökök ne csak a számítástechnikai eszközök felhasználói, de azok működésének értői is legyenek. Ha a hallgató belelát a számítógép belső felépítésébe, működésének alapvető ismeretében hatékony programok készítésére lesz képes.

A jegyzet mérnökhallgatók számára készült. A számítógépek programozása nagymértékben gyakorlati tárgy. Ezért jegyzetünk az előadásokhoz igazodva egy elméleti, majd egy jelentős gyakorlati részt tartalmaz. Ez a második rész kidolgozott példák gyűjteménye, amely a tényleges programírási gyakorlatot hivatott segíteni.

Az egyes fejezetek végén az elméleti anyaghoz szorosan kapcsolódó feladatok, és a tanulást segítő ellenőrző kérdések találhatók.

1.1 Adatok megjelenési formái a számítógépben

A feldolgozásra szánt mennyiségek a mindennapi életben és a technikában használatos analóg mennyiségek (hőmérséklet, nyomás, feszültség, áram, sebesség, stb.). A számítógéppel való feldolgozhatóság érdekében ezeket a fizikai mennyiségeket a gép számára elfogadható formába kell alakítani. Ezt a folyamatot digitalizálásnak nevezzük. E célra a számos analóg-digitál átalakító szolgál. Ezekkel a berendezésekkel más tárgy (pl. Jelfeldolgozás a közlekedésben) foglalkozik.

A már rendelkezésre álló számszerű adatokat tovább kell alakítani valamilyen alkalmas kettes számrendszerű (bináris digitális) formába. Jelen fejezet ezekkel az átalakításokkal foglalkozik.

1.1.1 Egész típusú adatok

A decimális egészszámok valahány decimális karakterből állnak előjellel vagy anélkül. Ezeket a számítógép egésztípusú adatként tárolja. Az átalakításra az osztó algoritmus használatos. Az osztó algoritmus működését az alábbi példán mutatjuk be:

Legyen a decimális egészszám (n) 649. Ezt osztogatjuk 2-vel, a mindenkori maradék adja a bináris alak bit-jeit (bit = binary digit).

 © Aradi, Bécsi, Gyenes, Péter, BME

649	1
324	0
162	0
81	1
40	0
20	0
10	0
5	1
2	0
1	1
0	

1. ábra: Osztó algoritmus

A kapott bináris szám legnagyobb helyiértékű bitje a legalsó. Így

$$649_{10} = 1010001001_{2}$$

A kapott eredmény helyességéről meggyőződhetünk:

$$2^{0} + 2^{3} + 2^{7} + 2^{9} = 1 + 8 + 128 + 512 = 649$$

Ezen szám gépben való megjelenése függ az egészadat típusától, azaz attól, hány bitet használ a gép a szám tárolására. Az alábbiakban felsorolunk néhány egész adattípust (ez a típus függ az adott számítógép típusától – azaz a gépi reprezentációtól – és a használt programozási nyelvtől). Mi a példáinkban a Pascal és a Delphi nyelv típusait használjuk, és az Intel x86 architektúrát használó PC felépítését vesszük alapul.

Egészadat típusa	Hossz bitekben
byte	8
word	16
longword	32

1. táblázat: Az előjel nélküli egész típusok hosszai

A példában szereplő 649 tehát word típusban 0000 0010 1000 1001 lesz. Ez a szám byte típusban nem férne el, míg a longword típus sok felesleges bitet foglalna el.

Már ezen egyszerű példa is mutatja, hogy a programozó felelőssége a megfelelő adattípus kiválasztása az általa készített programban.

Az alábbi táblázat a fenti típusokkal reprezentálható maximális számokat mutatja:

Egészadat típusa	Delphi	Pascal
byte	255	255
word	65535	65535
longword	4294967295	nincs

2. táblázat: Az előjel nélküli egész típusok maximális értékei

A negatív egészszámokat a gépek kettes komplemens kódban jelenítik meg. Ez a kód egy – az ábrázolástól függő – pozitív egész értékből vonja ki a decimális szám abszolút értékét.

$N_{compl} = 2^{k+1} - |N|$, ahol k az ábrázolásra szánt bitek száma.

Nézzük pl. a -649 komplementálását k=11 esetére.

$$N_{compl} = 2^{12} - 649 = 4096 - 649 = 3447$$

 © Aradi, Bécsi, Gyenes, Péter, BME

Ez a rendelkezésre álló 11 biten ***1***10101110111 alakú lesz. A legelső dőlt bit lesz az előjel. Ez negatív egészszámok esetén mindig 1 lesz.

A gépek a kettes komplemens előállítását egyszerűbben bit negálással végzik. Kiindulási alap a szám bináris alakja (példánkban 1010001001). Ezt az alakot a legkisebb helyiértéktől (jobbról) az első egyesig a biteket másolva, ettől kezdve bitenként negálva megkapjuk a kettes komplemens számot: ***1***10101110111.

A negatív egészszámokat is tárolni képes adattípusokat az alábbi táblázat mutatja:

Egészadat típusa	Delphi	Pascal
shortint	8	u.a.
smallint	16	u.a.
integer	32	16
longint	32	u.a.
int64	64	nincs

3. táblázat: A Pascal nyelv előjeles egész típusainak hosszai

A fenti típusok számábrázolási tartománya:

Egészadat típusa	Delphi	Pascal
shortint	-128 - 127	u.a.
smallint	-32768 - 32767	nincs
integer	-2147483648 - 2147483647	-32768 - 32767
longint	-2147483648 - 2147483647	u.a.
int64	$-2^{63}..2^{63}-1$	nincs

4. táblázat: A Pascal nyelv előjeles egész típusainak intervallumai

Az egész adattípusok megismerése után nézzünk néhány példát a különböző egészszámok ábrázolására.

Példa 1: Ábrázoljuk a 387 számot integer típus esetén.

$387^{10} = 110000011^{\underline{2}}$ (Osztó algoritmussal)

integer alak: 0000 0000 0000 0000 0000 0001 1000 0011 (32 biten)

A számítástechnikában a hosszú bináris számok kiolvasása a hexadecimális (16 alapú) számrendszerben történik az alábbiak szerint:

Decimális szám	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimális szám	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

5. táblázat: A hexadecimális számrendszer értékkészlete

Ennek figyelembevételével az előbbi integer alak:

$00000183^{\underline{16}}$

Megfigyelhetjük, hogy 4 bit (1 bináris tetrád) ad egy hexadecimális digitet.

Példa 2: Ábrázoljuk a -58 decimális számot shortint adattípusban (8 bit).

$5810 = 111010^{\underline{2}}$ (Osztó algoritmussal)

komplementálva: -58 = ***1*** 100 0110 (A legelső bit az előjel) = $C6^{\underline{16}}$

Látható, hogy az abszolút értékre csak 7 hasznos bit marad, ez indokolja a -128 .. 127 számábrázolási intervallumot.

<u>Gyakorló feladatok:</u>

F01. Ábrázoljuk a jelenlegi évszámot a legalkalmasabb típusnak megfelelő formában, és adjuk meg hexa alakban is.

F02. Ábrázoljuk a - 941 decimális számot a legalkalmasabb típusnak megfelelő formában, és adjuk meg hexa alakban is.

Ellenőrző kérdések:

K01. Hogyan váltunk át egy decimális egészszámot bináris alakba?

K02. Milyen egész adattípusokat használ a Pascal nyelv?

K03. Mire való a 16-os számrendszer?

1.1.2 Valós típusú adatok

Valós típusba tartoznak a törtrészt és/vagy kitevőt tartalmazó számok. Ezek ábrázolására a számítógépek a lebegőpontos normál alakot használják. Az ábrázolás fontos adata itt is a hasznos bitek száma.

Valós adat típusa	Delphi	Pascal
single	32	u.a.
Real48	48	nincs
real	64	48
double	64	u.a.
comp	64	nincs
currency	64	nincs
extended	80	u.a.

6. táblázat: A valós adattípusok méretei bitben

A decimális számok normál alakja:

$N = m * 10^{\pm k}$, ahol m a mantissza és k a karakterisztika.

A mantissza szokásos értéktartománya: 1 > m < 10.

Például: 12345 egészszám decimális normál alakja: $1.2345 * 10^4$

A bináris normál alak:

$N = m * 2^{\pm k}$, ahol m a mantissza és k a karakterisztika.

A mantissza értéktartománya: 0 > m < 1.

Például: 231 egészszám bináris normál alakja: $0.11100111 * 2^8$

Itt kell foglalkozni az egynél kisebb decimális számok bináris formába való átalakításával. Erre a célra a gép a szorzó algoritmust használja. A működését ismét példán mutatjuk be. Legyen az átalakítandó decimális szám 0.625

.625	
.25	1
.5	0
.0	1

A törtrészt ismételten kettővel szorozva a biteket az egészrész adja.

A fentieknek megfelelően:

$0.625^{\underline{10}} = 0.101^{\underline{2}}$, ahol a legelső bit a legnagyobb helyiértékű: 2^{-1}

Az eredmény helyességét könnyen ellenőrizhetjük:

$$2^{-1} + 2^{-3} = \frac{1}{2} + \frac{1}{8} = 0.625$$

Fontos megjegyezni, hogy míg az egészszámok átalakítása pontos és maradéktalan, addig a decimális törtek csak ritkán adnak véges bináris törtet.

Egy általános valós szám bináris normál alakúvá konvertálása a gép számára több lépést kíván.

Vegyük az alábbi példát: $N = 3.24125 * 10^2$

Eltüntetjük a kitevőt:

$$N = 324.125$$

Az egészrészt átalakítjuk bináris formába:

$324^{\underline{10}} = 101000100^{\underline{2}}$ (osztó algoritmus)

Az törtrészt átalakítjuk bináris formába:

$0.125^{\underline{10}} = 0.001^{\underline{2}}$ (szorzó algoritmus)

 © Aradi, Bécsi, Gyenes, Péter, BME

Egyesítjük a két értéket:

$$324.125_{10} = 101000100.001_{2}$$

A bináris normál alak:

$$0.101000100001 * 2^{9}$$

Ez a lebegőpontos számábrázoláshoz szükséges forma, azaz

$$m = 0.101000100001 \text{ és } k = 9.$$

Most nézzük, hogy a valós számokat miként ábrázolja a gép. Jelenleg a legszélesebb körben elterjedt ábrázolási módot az IEEE 754 nemzetközi szabvány rögzíti. Az alapelve mindegyik típusnál ugyanaz, csak a karakterisztika és mantissza, valamint az eltolás mértékében van különbség. Így az itt bemutatott példából a további típusok ábrázolása is levezethető.

Válasszuk a single 32 bites adatformátumot, amely 32 bitet (4 byte) használ. Az egyes bitek az alábbi ábrán láthatók.

31. bit	30-23. bit	22-0. bit
Ej. (1 bit)	Karakterisztika (8 bit)	Mantissza (23 bit)

A szám előjele a legelső bit (1: negatív, 0 : pozitív valós szám).

A bináris alappontot a mantissza elé képzeljük, mivel a mantissza 1-nél kisebb. Ennek az alappontnak a tényleges helyét azonban a karakterisztika adja, ezért nevezzük ezt az ábrázolási módot lebegőpontos számformátumnak.

A karakterisztikára tehát 8 bit szolgál, ebbe kell beférni előjelesen a ± k értéknek. A karakterisztikát eltolt nullpontú számábrázolással jeleníti meg a gép. Ez azt jelenti, hogy a rendelkezésre álló 8 biten ábrázolható -127..128 tartományt eltoljuk 127-el, így kapjuk meg az ábrázolt karakterisztikát. Azonban a két szélsőérték (-127 és 128) nem használatos, mivel azok speciális értékek tárolására vannak fenntartva (Lásd később!), így a használható tartomány -126..127.

Példánkban k = 9 szerepel, azonban az IEEE 754 szabvány szerint a mantisszát egyes normalizált, explicit bites alakban ábrázoljuk, ahol az egyes helyiértéken lévő egyest a mantissza nem tartalmazza, mivel értéke egyértelmű.

Ezek után a fenti példánk a következőként alakul:

m = 1.01000100001 és k= 8

Az eltolt karakterisztika k'=k+127=135, azaz binárisan 10000111, míg a mantissza az egyes helyiérték elhagyása után m'=01000100001.

Ezek után már meghatározhatjuk miként alakul a 324.125 decimális szám lebegőpontos normál alakja single típusú (32 bit) adatként:

0	10000111	01000100001000000000000
Ej. (1 bit)	Karakterisztika (8 bit)	Mantissza (23 bit)

$0100\ 0011\ 1010\ 0010\ 0001\ 0000\ 0000\ 0000_{2} = 43\ A2\ 10\ 00\ 00_{16}$

Gyakorlásképpen nézzünk egy másik példát is. Ábrázoljuk a -0.34375 negatív számot double (64 bit) típusú számként.

A szám abszolút értékének bináris alakja: $0.34375_{10} = 0.01011_{2}$

A bináris normál alak: $0.1011 * 2^{-3}$

Az eltolt karakterisztika k' = k + 1023 = 1020_{10} = $011\ 1111\ 1100_{2}$ (11 bit) A double alak tehát:

Előjel: 1

k = 011 1111 1100

m= 0110000000000000000000000000000000000000000000000000

Hexadecimálisan: BF C6 00 00 00 00 00 00

Amint az fent említésre került az IEEE 754 szabvány lehetőséget nyújt speciális értékek tárolására is, melyet az alábbi táblázat foglal össze.

 © Aradi, Bécsi, Gyenes, Péter, BME

Típus	Karakterisztika	Mantissza
± Nulla	0	0
±Végtelen	2^k-1	0
Nem szám (NaN)	2^k-1	Nem nulla

7. táblázat: A valós számábrázolás speciális értékei

A következő táblázatban a 32 biten ábrázolt speciális értékek láthatók.

Típus	Érték
Nulla	0 00000000 00000000000000000000000
Negatív nulla	1 00000000 00000000000000000000000
Pozitív végtelen	0 11111111 00000000000000000000000
Negatív végtelen	1 11111111 00000000000000000000000
Nem szám (NaN)	0/1 11111111 nem nulla

8. táblázat: A valós számábrázolás speciális értékei 32 biten

Fontos tudni az egyes lebegőpontos számformátumok számábrázolási tartományát. Ezt az alábbi táblázatból olvashatjuk ki.

Valós adat típusa	Delphi	Pascal
single	-1.5 * 10^{45} .. 3.4 * 10^{38}	u.a.
real	-5.0 * 10^{324} .. 1.7 * 10^{308}	-2.9 * 10^{39} .. 1.7 * 10^{38}
double	-5.0 * 10^{324} .. 1.7 * 10^{308}	u.a.
extended	-3.6 * 10^{4951} .. 1.1 * 10^{4932}	u.a.

9. táblázat: A valós adattípusok számábrázolási tartományai

A real típust a Pascal nyelv használja nagyobb pontosságú számábrázolásra, a Delphiben azonban már megegyezik a double típussal. A Delphiben kompatibilitási okokból létrehozták a real48 típust, amely a megfelel a Pascal real típusának..

Az extended típus extrém nagy számokhoz van tervezve. Használata csak kivételes esetekben indokolt.

Gyakorló feladatok

F01. Ábrázoljuk lebegőpontos normál alakban a $4.625 * 10^{-2}$ decimális számot 32 biten!

F02. Adjuk meg hexa számokkal a 95.5 decimális szám real (6 byte) alakját!

Ellenőrző kérdések

K01. Mi a lebegőpontos bináris normál alakja egy számnak?

K02. Adja meg a 0.1 decimális tört bináris megfelelőjét 16 bit esetére?

K03. Miként ábrázolja a gép a karakterisztikát?

1.1.3 Karakterek megjelenési formái

A billentyűzetről az adatokat karakterenként olvassa be a számítógép. Minden karakternek van egy bináris kódja, amelyet a billentyűzetben levő processzor sorosan továbbít a számítógép felé.

PC környezetben általánosan használt az ASCII (American Standard Code of Information Interchange) 7 bites kódrendszer. 7 biten 27 = 128 kódszó alkotható, így ez lehetővé teszi a számok, kis-és nagybetűk, a legfontosabb szimbólumok és vezérlő jelek megjelenítését. Eredetileg a 8. bitet adatvédelmi célokra használták (paritás bit), azonban a PC-n belül az adat sérülésének alacsony valószínűsége miatt újabban ezt a nyolcadik bitet is felhasználják karakterek kódolására.

Ez a kiterjesztett ASCII kódrendszer így 256 kódszót használ, ebbe az ékezetes betűk és számos különleges jel (szemigrafikus karakterek) kódja is helyet kapott. Az alap és kiterjesztett kódrendszert a 8. és 9. táblázat mutatja.

(Fontos megjegyezni, hogy a PC és billentyűzet kommunikációja – a különböző nyelvű billentyűzetek elterjedése miatt – napjainkban már egy speciális kódolással történik, amelyből nem származtatható egyértelműen a karakter ASCII kódja!)

 © Aradi, Bécsi, Gyenes, Péter, BME

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	Null character	32		64	@	96	`
1	Start of Header	33	!	65	A	97	a
2	Start of Text	34	"	66	B	98	b
3	End of Text	35	#	67	C	99	c
4	End of Transmission	36	$	68	D	100	d
5	Enquiry	37	%	69	E	101	e
6	Acknowledgement	38	&	70	F	102	f
7	Bell	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Horizontal Tab	41	)	73	I	105	i
10	Line feed	42	*	74	J	106	j
11	Vertical Tab	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	Shift Out	46	.	78	N	110	n
15	Shift In	47	/	79	O	111	o
16	Data link escape	48	0	80	P	112	p
17	Device control 1	49	1	81	Q	113	q
18	Device control 2	50	2	82	R	114	r
19	Device control 3	51	3	83	S	115	s
20	Device control 4	52	4	84	T	116	t
21	Negative acknowledgement	53	5	85	U	117	u
22	Synchronous idle	54	6	86	V	118	v
23	End of transmission block	55	7	87	W	119	w
24	Cancel	56	8	88	X	120	x
25	End of medium	57	9	89	Y	121	y
26	Substitute	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	File separator	60	<	92	\	124	|
29	Group separator	61	=	93	]	125	}
30	Record separator	62	>	94	^	126	~
31	Unit separator	63	?	95	_	127	Del

10. táblázat: ASCII 7 bites kódtábla

Dec	Char	Dec	Char	Dec	Char	Dec	Char
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

11. táblázat: Kiterjesztett ASCII 8 bites kódtábla

 © Aradi, Bécsi, Gyenes, Péter, BME

A karakterkódolás különbözik az előző fejezetekben bemutatott szám kódolásoktól.

Tekintsük például a 461 szám karakter kódját:

$$461_{10} = 00110100 \quad 00110110 \quad 00110001_{ASCII}$$

$$'4' \longrightarrow 34 \quad '6' \longrightarrow 36 \quad '1' \longrightarrow 31$$

Összehasonlításul álljon itt ugyanezen szám bináris ekvivalense:

$$461_{10} = 111001101_2$$

Láthatjuk, hogy a karakter kód bitjeinek száma a decimális digitek számának 8-szorosa, míg a bináris alak bitjeinek száma a decimális szám 2 alapú logaritmusával arányos (pontosabban a szám kettes alapú logaritmusának egészrésze +1).

A számítógépben a karakter típusú adatokat a Pascal nyelvben char típusjelzéssel azonosítjuk és a gép 8 biten tárolja ezeket.

A táblázatból kitűnik, hogy a decimális számjegyek kódjának felső 4 bitje 0011. Az alsó 4 bit egy általánosan használt számjegykódot ad. Ezt BCD (Binary Coded Decimal) kódnak nevezzük.

Álljon itt a BCD kódtábla, amely a decimális digitek 4 bites megfelelőjét mutatja.

DEC	BCD			
	2^3	2^2	2^1	2^0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

2. ábra: BCD kódtábla

Tekintsük például a 84 decimális szám BCD kódú alakját:

$$84^{\underline{10}} = 1000\ 0100^{\underline{BCD}} \text{ (ugyanez binárisan : } 84^{\underline{10}} = 101\ 0100^{\underline{2}}\text{)}$$

Adjuk meg az y = 4*x képlet karakter kódját (hexadecimálisan):

79 3D 34 2A 78 .

Gyakorló feladatok

F01. Ábrázoljuk BCD kódban 5381 decimális számot.

F02. Adjuk meg -95.5 decimális szám karakter kód formáját.

Ellenőrző kérdések

K01. Hány bitet igényel egy 6 digites decimális szám a BCD kódrendszerben?

K02. Hol használja a számítógép a CHAR adattípust?

K03. Mi a karakter kód típusának neve?

1.1.4 Szöveges adatok formái

A több karakterből álló, összefüggő szöveg gyakran előfordul az adatfeldolgozás során. Ezek a szövegek kezelhetők az előbb megismert karakter típus felhasználásával is. Mindamellett ilyenkor a feldolgozás kissé nehézkes – egyenként kell kezelni a szöveg karaktereit – ezért a programnyelvek e szövegek tárolására és kezelésére külön adattípusokat definiálnak. Ezek a Delphi környezetben az alábbiak, melyet kiegészítettünk a Pascal nyelv különbségeivel.

Típusa	Pascal	Mező hossza	Karakterek száma
ShortString		max. 256 byte	255
AnsiString		max. 2 GB*	max. 2^{31}
String	= ShortString	= AnsiString	
WideString		max. 2 GB	max. 2^{30}
PChar**		max. 2 GB	max. 2^{31}

12. táblázat: Szöveg tárolására alkalmas adattípusok

 © Aradi, Bécsi, Gyenes, Péter, BME

* 1 GB (GigaByte) = 1024 MB (MegaByte) = 2^{30} byte

** A PChar a más nyelveknél (pl. C) használatos bináris 0 végű string forma

A ShortString a Pascalban került definiálásra, előre megadott karakterszámú szöveg tárolására.

Pl. Tekintsük az ’Ali Baba’ szöveg ShortString formáját:

Pozíció	0	1	2	3	4	5	6	7	8
Érték	8	65	108	105	32	66	92	98	97
Karakter	Hossz	A	l	i	_	B	a	b	a

13. táblázat: Szöveg tárolására alkalmas adattípusok

A legelső byte a string hossza, a többi a szöveg egyes karaktereinek ASCII kódját tartalmazza. A Pascal számon tartja, hogy ezek nem egészszámok, hanem karakter kódok.

Az AnsiString típusú mezőt a nyelv dinamikusan kezeli (dinamikus adatokról később az 5.6. fejezetben lesz szó). Ehelyütt elegendő annyit tudni, hogy a mező automatikusan felveszi a rátöltött szöveg karaktereit, és a vonatkozó eljárások megfelelő módon kezelik azokat.

A PChar szintén dinamikus string forma, amelyet ’NULL’ karakter terminál. Az egyes string típusok egymásba átalakíthatók konverzióval.

A string típusú mező egyes karaktereihez indexeléssel férhetünk hozzá, például a fenti string 3. eleme a kis i betű.

Gyakorló feladatok

F01. Határozzuk meg a ’pi = 3.14’ képlet ShortString formátumát.

Ellenőrző kérdések

K01. Mi a szöveges adatábrázolás előnye?

K02. Mekkora lehet a ShortString és az AnsiString hossza?

1.2 Bináris műveletek

A számítógépekben (és a kalkulátorokban is) minden műveletvégzés kettes számrendszerben történik. Ennek oka, hogy ezt a bináris rendszert a legkönnyebb realizálni, mindössze kétállapotú eszközök használatát igénylik. Ilyen kétállapotú eszközök a tranzisztorok és a bistabil multivibrátorok. Ezek az elemek alkotják a számítógépek aritmetikai egységét és tárolóit (regisztereit), amiben a műveletvégzés történik. Ebben a fejezetben megvizsgáljuk, hogy az előzőkben bemutatott adatokkal miként működnek az alapvető műveletek.

1.2.1 Additív műveletek bináris kódban

Additív művelet alatt az összeadás és kivonás végrehajtását értjük. Az egész számok esetében ezt a két műveletet az operandusok komplemens kódjával végzi az aritmetikai egység. A komplemens kód alkalmazásával nincs szükség kivonásra, csak összeadást kell realizálni.

Tekintsük az alábbi példát.

A feladat a C = A + B összefüggés számítása. Feltételezzük, hogy mindhárom adat shortint típusú.

Amint korábban láttuk ez a típus 7 hasznos bitet használ, 8. bit az előjel.

Elsőként legyen A = 23 és B = 11

Ekkor A alakja a gépben: 0 0010111

B alakja a gépben: 0 0001011

Az összeadás a bináris algebra szabályai szerint történik, amely 2 bit esetén az alábbi igazságtáblázattal írható le (bejövő átvitel nélküli félösszeadó):

A	0	1	0	1
B	0	0	1	1
A+B (Sum)	0	1	1	0
Átvitel (Carry)	0	0	0	1

14. táblázat: Félösszeadó igazságtáblázata

 © Aradi, Bécsi, Gyenes, Péter, BME

Ha az adott (i-ik) helyiértéken figyelembe vesszük az előző (i-1-ik) helyiértékről érkező átvitelt (Ci-1) is, akkor az egy bites teljes összeadó igazságtáblázatát kapjuk.

A_i	0	1	0	1	0	1	0	1
B_i	0	0	1	1	0	0	1	1
C_{i-1}	0	0	0	0	1	1	1	1
A+B (S_i)	0	1	1	0	1	0	0	1
Átvitel (C_i)	0	0	0	1	0	1	1	1

15. táblázat: Teljes összeadó igazságtáblázata

Ezzel az összeadás (kiemeléssel az előjel bitet jelöltük):

A	**0**	0	0	1	0	1	1	1
B	**0**	0	0	0	1	0	1	1
S	**0**	0	1	0	0	0	1	0
C		0	0	1	1	1	1	1

Az eredmény 00100010 = $2^1 + 2^5 = 2 + 32 = 34$ a helyes összeget adta.

Másodjára legyen A = 23 és B = - 11

Ekkor A alakja a gépben: 00010111 (mint az előbbi esetben)

B alakja a gépben: 11110101 (-11 komplemens kódban)

Az összeg: 00001100 értéke + 12, tehát helyes

Harmadjára legyen A = -23 és B = 11

Ekkor A alakja a gépben: 1 1101001 (-23 komplemens kódban)

B alakja a gépben: 0 0001011

Az összeg: 1 1110100, negatív szám, amely komplemens kódban jelentkezett, értéke - 12, tehát helyes

A bemutatott példák igazolják, hogy az additív műveletek az operandusok komplemens kódjának használatával csak összeadásként elvégezhetők.

1.2.2 Additív műveletek BCD kódban

Egésztípusú adatok összeadásakor a művelet BCD kódban is elvégezhető, ha az operandusokat decimális jegyenként adjuk össze.

Természetesen az összeadóval szemben támasztott követelmény, hogy ha az operandusok BCD kódban adottak, akkor az összeg is BCD kódban jelenjen meg.

Képletben: $A_{BCD} + B_{BCD} = (A+B)_{BCD}$ kell legyen.

Ez nem minden esetben teljesül a kódolás természetéből adódóan. Három esetet vizsgálunk.

$A + B < 10$, vagyis a két decimális számjegy összege kisebb, mint 10.

Legyen,

$$\begin{aligned} A &= 6 \longrightarrow A_{BCD} = 0110 \\ B &= 3 \longrightarrow B_{BCD} = \underline{0011} \\ & \qquad\qquad\qquad\quad 1001 \end{aligned}$$

Az összeg BCD kódú, értéke 9, tehát helyes.

$9 < A + B < 16$, vagyis

a két decimális számjegy összege nagyobb mint 9, de kisebb mint 16.

Legyen,

$$\begin{aligned} A &= 6 \longrightarrow A_{BCD} = 0110 \\ B &= 7 \longrightarrow B_{BCD} = \underline{0111} \\ & \qquad\qquad\qquad\quad 1101 \end{aligned}$$

Az összeg nem BCD kódú, a helyes összeg 0001 0011 (13) lenne.

$15 < A + B$, vagyis a két decimális számjegy összege nagyobb mint 15.

$$A = 8 \longrightarrow A_{BCD} = 1000$$
$$B = 9 \longrightarrow B_{BCD} = \underline{1001}$$
$$000\ 10001$$

Az összeg BCD kódú, értéke 11, de a helyes összeg 0001 0111 (17) lenne.

Látható, hogy a b, és c, esetben még hibátlan összeadás elvégzése esetén is hibás eredmény adódik. A javítás egy +6 korrekció, amit a hamis összeghez adunk hozzá, ha az operandusok decimális összege > 9.

Nézzünk egy példát, amely a korrekcióval immár a helyes összeget adja.

Legyen A = 587 és B = 94

A	0101	1000	0111
B	0000	1001	0100
Hamis összeg	0110	0010	1011
Decimális átvitel		1	1
Korrekció	0000	0110	0110
Helyes összeg	0110	0110	0001
	6	8	1

Gyakorló feladatok

F01. Végezzük el az összeadást binárisan, ha A = 126 és B = -39 .

F02. Összeadás BCD kódban: 178 + 39

Ellenőrző kérdések

K01. Miért használják a számítógépek a komplemens kódrendszert?

K02. Mely esetben nem ad helyes eredményt 2 BCD kódú számjegy összeadása?

K03. Hogyan kell korrigálni a BCD hamis összeget?

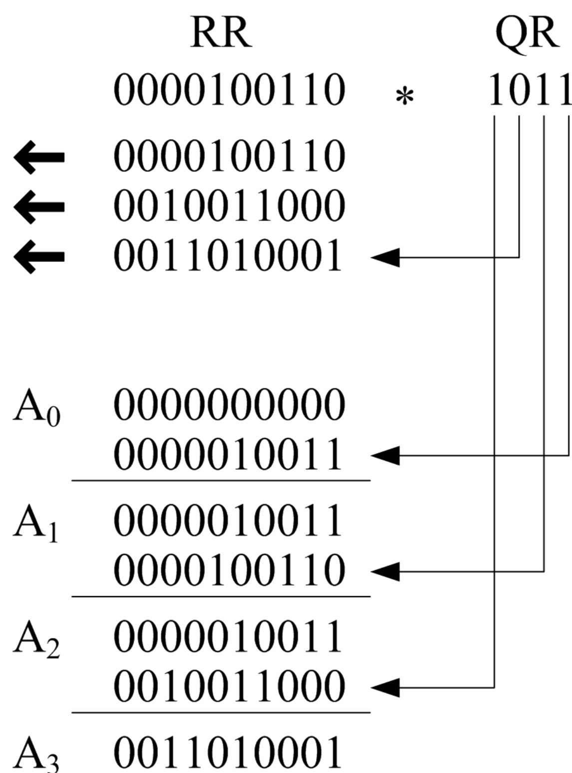
1.2.3 Bináris szorzás

A négy alapművelet egyike a szorzás. Ezt a műveletet a számítógép aritmetika ismételt összeadások sorozatával végzi el. Számos algoritmust dolgoztak ki a géptervezők és a matematikusok. Közös vonásuk a bináris összeadás és léptetés (shiftelés).

Az operandusok speciális rekeszekbe, regiszterekbe kerülnek. A regiszter olyan tároló rekesz, amelyben a bináris adat párhuzamosan és sorosan is betölthető, jobbra, illetve balra léptethető, sőt a mindenkori értékét külön jelző bitek (FLAG) mutatják.

A szorzáshoz 3 regiszter szükséges (vannak olyan módszerek, amelyeknél több), ezek az AR (Accumulator Register), RR (Reserved Register) és QR (Quotient Register). A bináris szorzást számpéldán mutatjuk be.

Számítandó a 19 * 11 szorzat.



3. ábra: Szorzás balra léptetéssel

 © Aradi, Bécsi, Gyenes, Péter, BME

A szorzandót (19) az RR-be, a szorzót (11) QR-be töltjük. Az AR kezdeti tartalma 0 legyen. A szorzás folyamán a QR bitjeit sorban vesszük, és ha az 1, akkor az RR tartalmát az AR-hez adjuk. Eztán RR tartalmát eggyel balra léptetve az eljárást a szorzó bitszámának megfelelően ismételjük. Ha a szorzó hossza k, akkor a szorzat k lépésben (összeadás + léptetés) adódik.

A fenti módszer egészszámokra való. A valós számok szorzása esetén az aritmetika a mantisszákat ily módon összeszorozza, míg a karakterisztikával összeadás műveletet végez.

1.2.4 Bináris osztás

Amiként a szorzás műveletet a számítógép ismételt összeadások és léptetések (shift) sorozatával valósítja meg, az összeadást is ismételt kivonásokkal és léptetésekkel oldja meg. Miután a kivonás nem más, mint komplemens kódú összeadás, valójában az osztás során is összeadások realizálódnak.

Mivel a hatványozás ismételt szorzás, valójában a számítógép minden aritmetikai műveletet sorozatos összeadásokkal valósít meg.

Nézzük az osztást megint csak példában. (A példában az egyszerűség kedvéért a kivonást nem komplemens kódú összeadásként mutatjuk, hanem valódi bináris kivonást végzünk.)

Számítandó a 11/16 hányados. A számlálót (11) az AR-be, a nevezőt (16) RR-be töltjük. A hányados jegyei QR-ben keletkeznek. Az osztás folyamán az alábbi lépések ismétlődnek:

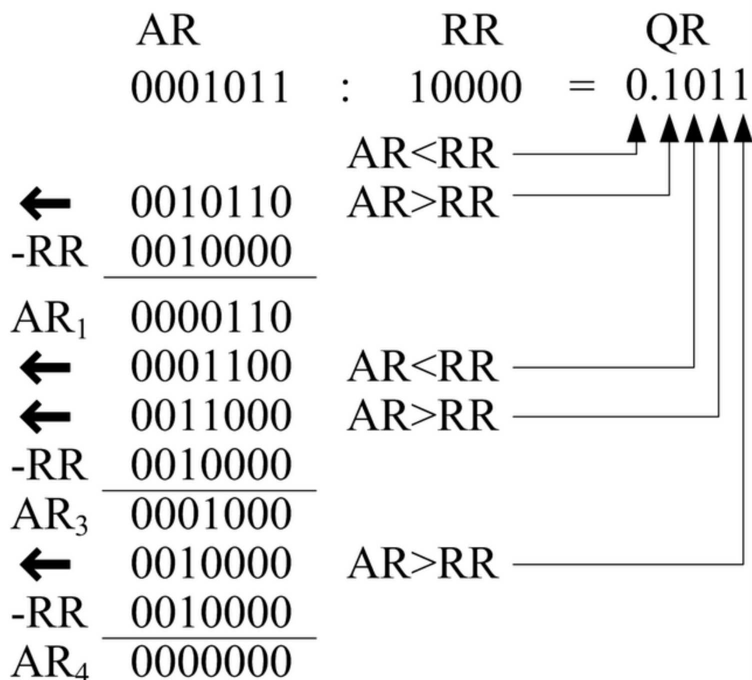
1.lépés: Megvizsgálandó, hogy az AR >= RR.

2.lépés: Ha igen, akkor a hányados aktuális bitje 1 lesz, és AR-ből levonjuk RR tartalmát, ha nem, akkor a hányados aktuális bitje 0 lesz nem végezzük el a kivonást.

3.lépés: Az AR tartalmát balra léptetjük.

4.lépés: Ha AR <> 0 vagy lépésszám < n, akkor visszatérünk az 1. lépéshez, egyébként az osztás véget ért.

A hányados QR-ben keletkezik, a maradék (ha van) AR-ben.



4. ábra: Osztás ismételt kivonással

Az itt bemutatott osztásmódszer csak egy a gyártók által kidolgozott sokféle megoldás közül. Lényegében mind hasonló elven, azaz ismételt összeadásra visszavezetve működik.

Gyakorló feladatok

F01. Végezzük el a 23 * 13 bináris szorzást.

F02. Végezzük el a 17/32 bináris osztást.

Ellenőrző kérdések

K01. Miként szoroz az aritmetika?

K02. Hogyan hatványoz a számítógép?

K03. Hány regisztert igényel a bináris szorzás, és melyek ezek?

1.2.5 Az adatok védelme

A bináris rendszer információhordozó képessége kicsi (mindössze 2 állapot). Ennek következtében a bináris alakú információ sok bitet tartalmaz. A bináris információt elektronikus eszközök tárolják. Ezek a villamos zavarokra (hálózati zavar, villámlás, induktív behatás) érzékenyek. Tekintve, hogy az újabb számítógép típusok egyre alacsonyabb tápfeszültséggel működnek (néhány volt feszültséggel) az ’1’ és ’0’ bitet reprezentáló feszültségszintek távolsága (zajtávolság) is egyre kisebb. Ezért annak valószínűsége, hogy egy zavarimpulzus hatására bit tévesztés jön létre egyre nagyobb. Ez indokolja a bináris adatok védelmének szükségességét.

1.2.6 Paritás ellenőrzés

A védelemnek különböző szintjei vannak. A számítógépen belül elegendő egy alacsonyabb szintű védelem (kis vezetékhossz, fémdobozos kivitel) is. Ez általában a paritásbit alkalmazásával oldható meg.

Az adatvédelem általános elve, hogy az adathordozó biteket (kódszó) redundáns (járulékos) bitekkel egészítve tároljuk. Ezek a járulékos bitek a kódszó valamely tulajdonságát jelzik.

A paritás bit a kódszóban lévő egyesek számát párosra (páros paritás bit) vagy páratlanra egészítik (páratlan paritás bit) ki. A védett kódszó hossza ezzel növekszik.

Tekintsük pl. a 7 bites ASCII kódot páros paritás bittel kiegészítve.

Legyen a karakter ’C’, amelynek ASCII kódja decimálisan 67, hexadecimálisan 43. A gépben ez természetesen binárisan jelenik meg : 100 0011. Az egyesek száma 3, így a páros paritásbit 1 lesz.

Ezzel az új (immár 8 bites) kódszó 1 100 0011 lesz.

Hiba felfedése céljából a számítógép adat betöltésekor, vagy olvasásakor ellenőrzi a paritásbitet. Ha valamelyik bit (beleértve a paritásbitet is) megsérül, akkor ez hibát (Parity error) okoz, azaz a hiba felfedhető.

Nézzük az alábbi esetet :

Hibátlan kódszó: 1 100 0011

Meghibásodott kódszó: 1 101 0011

E hibás kódszónál az egyesek száma páratlan lett, így a hiba felfedhető.

Megjegyezzük, hogy az egy kódszóban megjelenő többes (páros számú) hibát ez az alacsony szintű védelem nem fedi fel.

A páros paritás ellenőrzést a számítógép bitenkénti antivalencia (kizáró vagy) művelettel határozza meg.

Az antivalencia (jele: $\oplus$) logikai művelet, amelynek igazságtáblázata 2 bitre az alábbi:

A	0	1	0	1
B	0	0	1	1
$A \oplus B$	0	1	1	0

16. táblázat: Az antivalencia művelet igazságtáblázata

A logikai algebra szabálya szerint $A \oplus B = AB + AB$

Legyen pl. a kódszó : 110101

Ekkor az antivalencia ismételt alkalmazásával megkapjuk a páros paritásbitet.

$$1 \oplus 1 = 0 \oplus 0 = 0 \oplus 1 = 1 \oplus 0 = 1 \oplus 1 = 0$$

A kapott paritásbit tehát 0, így a védett kódszó : 1101010 lesz.

1.2.7 Ellenőrző összeg

Nagyobb védelmet ad az ellenőrző összeget (checksum) alkalmazó védelem. Ez a védelmi rendszer az adathordozók (hard disk, pen-drive) adatblokkjainak védelmére használatos. Az eljárás során az adatblokk byte-jainak előjel nélküli (word típusú) összegét képezik 16 bit hosszban, majd ennek komplemensét csatolják az adatblokkhoz.

Blokk olvasásakor az adatbyte-ok összegének az ellenőrzőszámot is beleértve 0-t kell adni.

Legyen pl. az adatblokk: 3E F9 63 AB (4 hexa byte)

Ezen byte-ok összege: 0245^{16}

Az ellenőrzőszám 1 0000^{16} - 0245^{16} = $0DBB^{16}$

Így az ellenőrzőszámmal kiegészített adatmező: 3E F9 63 AB 0D BB

Felhasználáskor a mező byte-jait összeadva:

3E + F9 + 63 + AB + 0DBB = 0000.

Természetesen ha az adatblokk bitjei sérülnek, akkor az ellenőrző összeg nem lesz 0, így a hiba felfedhető (checksum error).

Gyakorló feladatok

F01. Határozzuk meg az 11101001 kódszó páros és páratlan paritásbitjét.

F02. Számítsuk ki a ’Pascal’ szó ellenőrző számát.

Ellenőrző kérdések

K01. Miért szükséges a bináris adatok védelme ?

K02. Mekkora védelmet ad a paritásbit ?

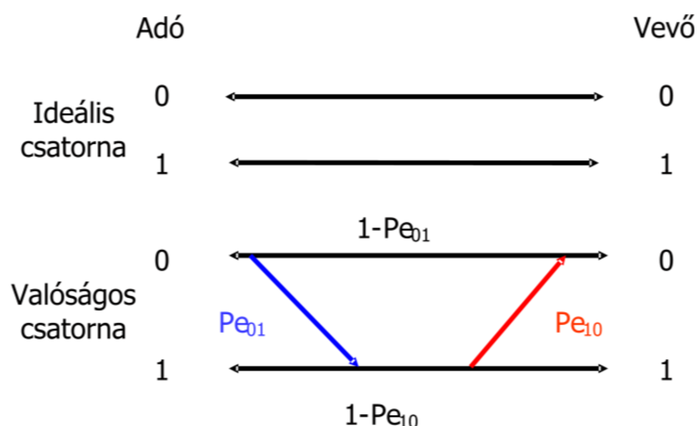
K03. Hol használjuk az ellenőrző számos adatvédelmet ?

1.2.8 Polinomos adatvédelem

Adatátvitel esetén az információ nagyobb sérülésveszélynek van kitéve, mint a számítógépen belül, ezért ekkor magasabb szintű védelem szükséges. Az átvitel általában sorosan (bitenként) történik két pont között. Az átvitelre a bináris csatorna szolgál.

A bináris csatorna legegyszerűbb modellje az 1.3.1. ábra szerinti lehet. Pe_{01} annak valószínűsége, hogy az adó által forgalmazott 0 a vevő oldalon a zavar miatt 1 lesz, hasonlóképpen a Pe_{10} pedig az 1 helyett 0 észlelésének valószínűsége (probability).

Nagy (esetleg földrészeket is áthidaló) távolságok esetén ezek a hibavalószínűségek számottevők lehetnek, és jelentős mértékben rontják az adatátvitel minőségét. Sok esetben ez (pl.légiközlekedési, vasúti automatikai, közúti vezérlések területén) biztonsági kockázatot jelent.



5. ábra: A bináris csatorna modellje

Az adatvédelem másik fontos szempontja az illetéktelen felhasználók kizárása (pl. banki, államigazgatási rendszerek). Erre is megoldást adnak az adatvédelmi (titkosítási) módszerek. Ebben a fejezetben egy viszonylag egyszerűen realizálható, mégis magas védelmi szintet biztosító módszert, a polinomos adatvédelmi eljárást mutatjuk meg.

A tárgyaláshoz szükségünk lesz néhány fogalom és eljárás bevezetésére.

1.2.9 Modulo 2 algebra

Az adatvédelem nem igényli a szokásos algebra használatát. Egyszerűbben kivitelezhető a modulo 2 algebra, amelynek szabályrendszere az alábbi tábla alapján érthető meg.

A lehetséges értékei	0 0 1 1
B lehetséges értékei	0 1 0 1
A+B	0 1 1 0
A-B	0 1 1 0
A*B	0 0 0 1
A/B	- 0 - 1

17. táblázat: A modulo 2 algebra szabályrendszere

Megfigyelhetjük, hogy az összeadás és kivonás azonos eredményt ad (átvitel nincs). A szorzás a logikai algebra szabálya szerinti, a 0-val való osztás (mint az algebrában) nincs értelmezve.

Ezen algebra használatát a későbbiek során számpéldán mutatjuk be.

1.2.10 Bináris mező és polinom

A bináris mező k elemű bitsorozat, amelyen lineáris műveleteket végzünk. A bináris polinom a bináris mező elemeiből, mint együtthatókból képzett algebrai kifejezés. Egy k-ad fokú bináris polinom általános alakja:

$$P_k = \sum_{i=0}^{k} m_i * x^i$$

ahol m_i a bináris mező i-ik bitje, x pedig formális változó (bináris polinom esetén értéke 2).

Tekintsük például az 1011001 mezőhöz tartozó bináris polinomot:

$$P_6 = 1*x^6 + 0*x^5 + 1*x^4 + 1*x^3 + 0*x^2 + 0*x + 1 = x^6 + x^4 + x^3 + 1$$

A legkisebb helyiértékkel a mező jobboldali eleme rendelkezik.

Az előbbi pontban definiált modulo 2 műveletek szemléltetését példán mutatjuk be.

Legyen az A mező 1001011 és a B mező 10111.

Az A+B összeg (ugyanaz mint A-B különbség):

$$\begin{array}{r} 1001011 \\ \pm \quad 10111 \\ \hline 1011100 \end{array}$$

Polinomként felírva:

$$(x^6 + x^3 + x + 1) + (x^4 + x^2 + x + 1) = x^6 + x^4 + x^3 + x^2$$

Szorozzuk össze az 10101 mezőt az 1010 mezővel:

$$(x^4 + x^2 + 1)*(x^3 + x) = (x^7 + x) \text{ mező formában: } 10000010.$$

Végül tekintsük az 1100011: 10101 osztást:

$$(x^6 + x^5 + x + 1) : (x^4 + x^2 + 1) = x^2 + x + 1$$

$$\begin{array}{l}
x^6 + x^4 + x^2 \\
\hline
x^5 + x^4 + x^2 + x + 1 \\
x^5 + x^3 + x \\
\hline
x^4 + x^3 + x^2 + 1 \\
x^4 + x^2 + 1 \\
\hline
x^3 \quad \text{maradék}
\end{array}$$

1.2.11 Adatvédelmi algoritmus

A fentiek figyelembe vételével lássuk a polinomos adatvédelem működését.

A továbbítandó polinom alakú információ jele legyen u(x). Az információ polinomját megszorozzuk egy generátor polinommal, amelynek jele legyen g(x). Így a továbbítandó kódblokk:

$$b(x) = u(x) * g(x)$$

Ha u(x) n bitet, míg g(x) k bitet foglal el, akkor b(x) polinom hossza n+k lesz, azaz a generátor polinom fokszáma adja a redundáns bitek számát.

 © Aradi, Bécsi, Gyenes, Péter, BME

A következő táblázatban néhány gyakori generátor polinomot mutatunk be.

CRC név	Felhasználó	Polinom
CRC-15-CAN	Járműipar, CAN hálózat	$x^{15} + x^{14} + x^{10} + x^{8} + x^{7} + x^{4} + x^{3} + 1$
CRC-16-IBM	USB, Modbus	$x^{16} + x^{15} + x^{2} + 1$
CRC-16-CCITT	X.25, Bluetooth, SD kártya	$x^{16} + x^{12} + x^{5} + 1$
CRC-24	Járműipar, Flexray	$x^{24} + x^{22} + x^{20} + x^{19} + x^{18} + x^{16} + x^{14} +$ $x^{13} + x^{11} + x^{10} + x^{8} + x^{7} + x^{6} + x^{3} + x + 1$
CRC-32Q	Repülőipar, AIXM	$^{32} + x^{31} + x^{24} + x^{22} + x^{16} + x^{14} +$ $x^{8} + x^{7} + x^{5} + x^{3} + x + 1$
CRC-32-IEEE 802.3	V.42, MPEG-2, PNG	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} +$ $x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$

18. táblázat: Példák generátor polinomra

Tekintsünk egy konkrét példát a kódolás/dekódolás megvalósítására.

Legyen az u(x) = 110101 mezővel adott, és használjuk a CRC-16 generátort.

$$b(x) = u(x)*g(x) = (x^{5} + x^{4} + x^{2}+1) * (x^{16} + x^{15} + x^{2} + 1) =$$

$$= x^{21} + x^{19} + x^{18} + x^{17} + x^{16} + x^{15} + x^{7} + x^{6} + x^{5} + 1$$

A kódszó tehát 1011111000000011100001

Ez kerül az adatátvitel során továbbításra. A vevő oldalon végzik el a dekódolást az u(x) = b(x)/g(x) hányados kiszámításával. Az adatátvitel hibátlannak minősül, ha az osztás során a maradék 0-nak adódik. A hányados a hasznos információ, azaz az eredeti üzenet.

$$(x^{21}+x^{19}+x^{18}+x^{17}+x^{16}+x^{15}+x^7+x^6+x^5+1):(x^{16}+x^{15}+x^2+1) = x^5+x^4+x^2+1$$

$$\begin{array}{l}
x^{21}+x^{20}+x^7+x^5 \\
\hline
x^{20}+x^{19}+x^{18}+x^{17}+x^{16}+x^{15}+x^6+1 \\
x^{20}+x^{19}+x^6+x^4 \\
\hline
x^{18}+x^{17}+x^{16}+x^{15}+x^4+1 \\
x^{18}+x^{17}+x^4+x^2 \\
\hline
x^{16}+x^{15}+x^2+1 \\
x^{16}+x^{15}+x^2+1 \\
\hline
0
\end{array}$$

Nézzük mi történik adatátviteli hiba esetén.

Ha az átvitel hibás, akkor a b(x) forgalmazott kódblokk helyett r(x) polinom adódik, amelyet

$$r(x) = b(x) + e(x)$$

alakban írhatunk fel.

Képletünkben e(x) a hibapolinom (vagy hibaszindróma), amelynek maximális fokszáma n+k -1 lehet. A vevő hibátlannak tekinti az átvitelt, ha az osztás elvégzése után a maradék = 0. A maradék akkor lesz 0, ha e(x) = 0 (ez a hibamentes átvitel esete), vagy ha e(x) = k*g(x), azaz a hiba polinom a generátor polinom egészszámú többszöröse (ez a fel nem fedett hibák esete).

Számítsuk ki a fel nem fedett hiba előfordulási valószínűségét. Feltételezzük az egyenletes hibaeloszlást, másszóval a hibák korreláció mentességét.

Ekkor

$$hv(x) = \frac{p}{2n + k - 1}$$

ahol p azon esetek száma, amikor

© Aradi, Bécsi, Gyenes, Péter, BME

$$\frac{r(x)}{g(x)} = i \cdot g(x) \quad (0 < i < n)$$

a nevező pedig az összes hibás esetek száma.

Mivel r(x) fokszáma n+k, g(x) fokszáma k, így p = n-1.

Ezzel:

$$hv(x) = \frac{n-1}{2n+k-1}$$

Fenti példánk adataival, ahol n =5, k=16 volt, a fel nem fedett hiba valószínűsége

$$hv(x) = \frac{4}{221-1} = 0{,}0000019073$$

A felfedett hibák kijavítására többféle stratégia létezik. Ezekkel részletesebben a Jelfeldolgozás a közlekedésben c. tárgy keretében foglalkozunk.

1.3 Az aritmetikai művelet ellenőrzése

Amint a korábbi fejezetekben láttuk, a műveletek alapja a bináris összeadás, így az abban fellépő hiba valószínűségét a lehető legkisebb értékre kell szorítani. Az alábbiakban egyszerű módszert mutatunk be a helyes működés ellenőrzésére. A védelem alapja a paritásbit képzése és speciális ellenőrzés végzése.

Rendeljünk az operandusokhoz páros paritásbitet a következőképpen:

$$P_A = A_1 \oplus A_2 \oplus A_3 \oplus \ldots \oplus A_n$$

$$P_B = B_1 \oplus B_2 \oplus B_3 \oplus \ldots \oplus B_n$$

Hasonlóképpen rendeljünk az összeghez és az átvitelhez is paritásbitet:

$$P_S = S_1 \oplus S_2 \oplus S_3 \oplus \ldots \oplus S_n$$

$$P_C = C_0 \oplus C_1 \oplus C_2 \oplus \ldots \oplus C_{n-1}$$

Az átvitel bitjei az összeadás természete szerint eltolt indexet használnak.

Az összeadó logikai egyenlete szerint az i. helyiértéken az összeget (S_i) az

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

ismert képlet adja (egy bites teljes összeadó).

Helyettesítsük be ezt az összefüggést a PS képletébe:

$$P_S = A_1 \oplus B_2 \oplus C_0 \oplus A_2 \oplus B_2 \oplus C_1 \oplus \ldots \oplus A_i \oplus B_i \oplus C_{i-1}$$

Figyelembe véve, hogy az antivalencia lineáris művelet, így asszociatív, kommutatív, disztributív tulajdonságokkal rendelkezik. Rendezzük át képletünket :

$$P_S = A_1 \oplus A_2 \oplus \ldots \oplus A_n \oplus B_1 \oplus B_2 \oplus \ldots \oplus B_n \oplus C_0 \oplus C_1 \oplus \ldots \oplus C_{n-1}$$

$$P_S = P_A \oplus P_B \oplus P_C$$

A képletben P_S az elméletileg kiszámított paritásbit. Az összeadás elvégzése után előállított paritásbit a tényleges, amely az összeadó hibázása esetén eltérhet az elméletileg helyes paritástól.

Az összeadás hibás volt, ha az elméleti és tényleges, paritás különbözik, azaz antivalens:

$$Error = P_{Selm} \oplus P_{Stényl} = P_A \oplus P_B \oplus P_C \oplus P_S$$

Ezen megfontolás alapján mindössze 4 paritásbit antivalenciájának vizsgálatával megoldható az összeadó ellenőrzése.

A módszer szemléltetésére nézzünk egy számszerű példát először a hibátlan esetre:

Változó	Érték	Paritás
A	01101	**1**
B	01011	**1**
S	11000	**0**
C	01111	**0**

Error = 0

 © Aradi, Bécsi, Gyenes, Péter, BME

Most feltételezzünk egy bithibát az összegben:

Változó	Érték	Paritás	
A	01101	**1**	
B	01011	**1**	Error = 1
S	11100	**1**	
C	01111	**0**	

Látható, hogy az ellenőrzés felfedte a hibát.

Gyakorló feladatok

F01. Számítsuk ki az 11101001 kódszó CRC-16 generátorral való szorzatát.

F02. Számítsuk ki 111001 / 101 hányadost.

Ellenőrző kérdések

K01. Miért szükséges a polinomos adatvédelem?

K02. Mi a hibaszindróma?

K03. Hogyan ellenőrzik az aritmetikai összeadót?

2 Programnyelvek osztályozása

A programnyelvek a számítógépek számára végrehajtandó utasítások leírását definiálják.

2.1 Gépi kód

A számítógép processzora saját kódolású, bináris számsorozatot értelmez. Ezt gépi kódnak nevezzük. Ezek elemei a gépi kódú utasítások (más néven makro utasítások). Ezek felépítése nagyban függ a számítógép processzorának típusától. Ezek alkotják a programnyelvek legalacsonyabb szintjét.

Példaként legyen az y = c + b parancs fiktív gépi kódú formája:

101101	01110001	10001011
+	c	b

A programozó számára manapság ez a nyelv nem használatos. A számítógépes technika kezdetén még csak ez a nyelv létezett.

2.2 Assembler nyelv

A következő, magasabb nyelvi szint az assembler (halmozott). Ez a nyelv az egyes utasításokat emlékeztetőkkel (mnemonik) helyettesíti: pl. az összeadás ADD, osztás DIV, töltés MOVE, stb.

Az assembler már alkalmas nagy hatásfokú, kisméretű, gyors programok készítésére. Főként mikroprocesszoros vezérlők programozására használatos.

Minden processzorcsaládnak a gyártója saját assembler nyelvet definiál. Például sokáig népszerű volt a 8 bites Z80 assembler, vagy a Motorola 6800 családhoz készült assembler. A PC gépeknek is létezik az Intel cég által fejlesztett x86 assembler programnyelve.

Ezeket a nyelveket a processzorok számára le kell fordítani gépi kódú alakba a végrehajtás végett. Ezt a fordítást a megfelelő assembler fordító (compiler) tudja elvégezni.

Szemléltetésképpen bemutatunk egy assembler program részletet, amely az Intel 8051processzorához íródott:

```
loop:  mov   a,P4        ; Get Switch
       anl   a,#0x01     ; Test lowest bit
       jz    even        ; If 0 even
       anl   P5,#0xEF    ; LED1 Off
       orl   P5,#0x20    ; LED2 On
even:  anl   P5,#0xEF    ; LED2 Off
       orl   P5,#0x10;   ;LED1 On
       sjmp  loop
```

2.3 Magas szintű programnyelv

A számítógépek fejlődése, teljesítményük növekedésével járt. A fejlődést az integrált áramköri technológia megjelenése lendítette fel az 1960-70 évektől kezdve. Így lehetővé vált a felhasználók minél magasabb szintű kiszolgálása: megjelentek az operációs rendszerek. Az operációs rendszer (OS: Operating System) olyan rendszerprogramok összessége, amely a számítógép hardware (a gép szerkezeti részei) vezérlését és a felhasználóval való magas szintű kommunikációt végzik.

Az igény a programfejlesztők részéről a minél kényelmesebb, a matematikában megszokott szimbólumokkal operáló programnyelv használata lett. Így jöttek létre sorra a magas szintű programnyelvek. A teljesség igénye nélkül álljon itt néhány fontosabb nyelv:

- BASIC: Beginners All-Purpose Symbolic Instruction Cod, kalkulátorokhoz, korai otthoni számítógépekhez: ZX-Spectrum, Commodore 64, stb. Továbbfejlesztett változatai ma is léteznek, mint például a Microsoft Visual Basic.
- PASCAL: Általános nyelv mérnöki, matematikai feladatok programozására.
- Object Pascal:
- C: Tömör szintaktikájú nyelv gépközeli programozáshoz.
- C++: C-n alapuló, általános célú objektumorientált programnyelv.

- C#: a Microsoft által a .NET keretrendszer részeként kifejlesztett objektum-orientált programozási nyelv
- .NET keretrendszer: A Microsoft által készített gyors alkalmazásfejlesztést (RAD), platformfüggetlenséget és hálózati átlátszóságot támogató szoftverfejlesztői platform
- Java: objektumorientált, platformfüggetlen programozási nyelv, amelyet a Sun Microsystems fejleszt a 90-es évek elejétől kezdve napjainkig. Szintaxisát főleg a C és C++ programnyelvektől örökölte, viszont sokkal egyszerűbb objektummodellel dolgozik, mint a C++.

Ezen nyelvek mindegyike alkalmas a mérnöki feladatok megoldására, de valamely szempontból egyik vagy másik előnyösebb lehet egy szakma, vagy egy iskola számára.

A nyelvek mindegyikének saját fordító programja van, amely az adott nyelv szintaktikáját (nyelvtanát) ismeri. A nyelvek kidolgozói és forgalmazói gyakran a fordítót integrált fejlesztői környezetbe : IDE (Integrated Development Environment) ágyazzák. Ezek a fejlesztő rendszerek sokrétű szolgáltatásaikkal (editor, debugger, teszter, interpreter, GUI desginer) megkönnyítik a programozást és a hibakeresést.

2.4 Szimbolikus nyelv

Az egyes szakmák saját gyártmányaik fejlesztésének segítésére speciális nyelveket alkalmaznak.

Ezek a számítógéppel támogatott tervező: CAD (Computer Aided Design), vagy gyártást segítő : CAM (Computer Aided Manufacturing) programok már nem utasításokból, hanem eljárásokat megvalósító blokkokból épülnek fel. Ezeket a blokkokat grafikus szimbólumok: ikonok (képecske) azonosítják. A program készítője ezeket az ikonokat helyezi el valamilyen Form (megjelenési kép) felszínén, és a programkészítés során ezen modulok (objektumok, processek) tulajdonságait állítja be. Végül az alkalmazás ezek meghatározott sorrendben való végrehajtásából áll.

A teljesség igénye nélkül néhány szakma szimbolikus nyelvét említjük:

Architect: építészek tervezői munkáját segítő programrendszer

Orcad és Protel: villamos tervező

AutoCAD: gépészeti tervező

MatLab: matematikai problémák szimulációs rendszere

Oracle: adatfeldolgozó

Delphi: eseményvezérelt mérnöki alkalmazás fejlesztő

3 A programkészítés lépései

A programkészítés nem egyszerű feladat. Átgondolt, szisztematikus, figyelmet és szakértelmet igényel. Gondoljunk arra, hogy a műszaki gyakorlatban az elkészített programok az esetek nagy részében műszaki-gazdasági folyamatokat irányítanak, esetenként nagy értékű berendezéseket vezérelnek. A hibás program nagy kárt okozhat akár, ha a közlekedési folyamatokra gondolunk (vasúti automatikai vezérlők, légi közlekedési irányító rendszerek, jármű fedélzeti számítógépek programjai, stb.)

Ugyancsak nagy felelősséget hordoznak az adatfeldolgozó alkalmazások, amelyek eredményei alapján jelentős – vállalati, vagy akár nemzetgazdasági döntések születhetnek.

Fontosak a pénzpiaci, banki alkalmazások is, nagy kárt okozhat egy hibás tőzsdei, vagy árfolyamszámító program.

Ezen megfontolások alapján jó megtanulni már a legelső egyszerű (iskolai) programjaink készítésekor a precíz, figyelmes munkát. Célszerűen az alábbi lépések szerint haladjuk programtervező munkánkban.

3.1 A probléma matematikai megfogalmazása

A számítógép, amint korábban láttuk csak egyszerű alapműveleteket végez (összeadás, logikai alapműveletek). Bármilyen területen használjuk is fel, a megoldandó problémát valamilyen matematikai formában kell megfogalmaznunk. Ezt a lépést modellezésnek nevezzük. E művelet során valamely zárt alakú képletekbe, összefüggésekbe kell konvertálni a problémát.

Nézzünk a modellezésre két példát!

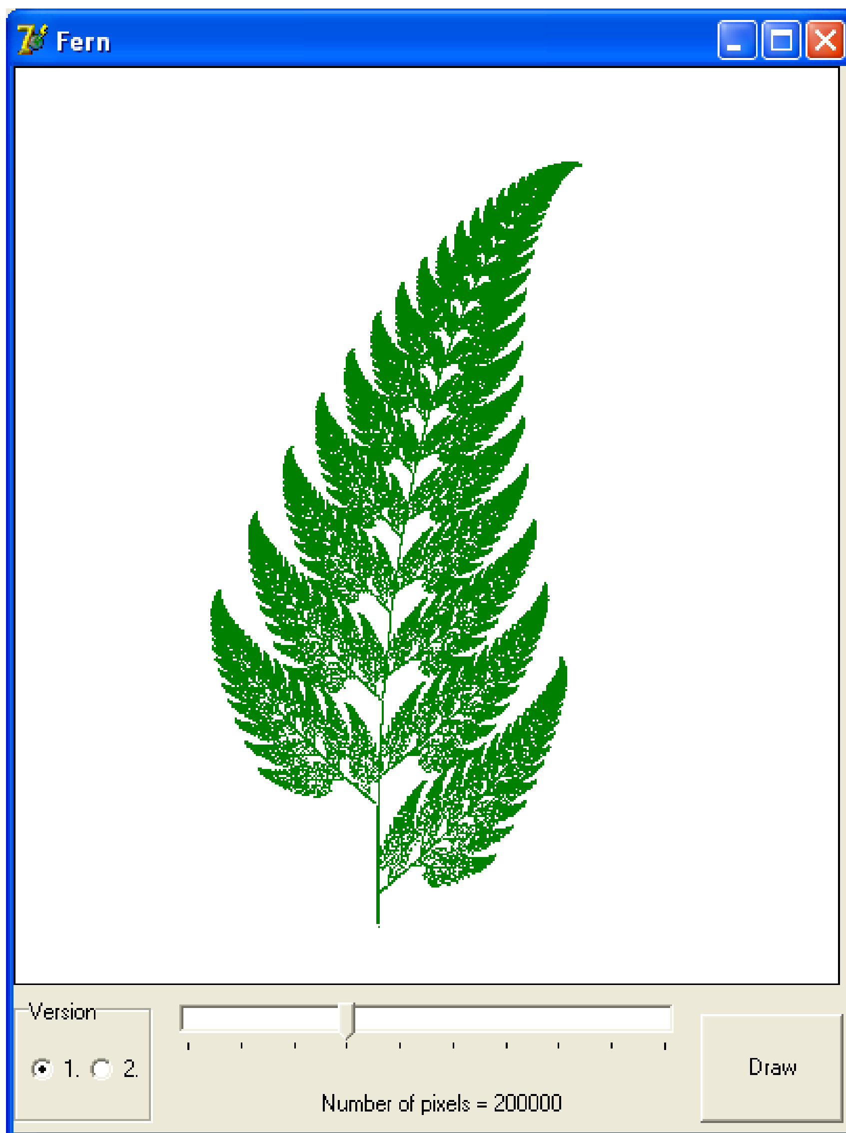
Biológusok kíváncsiak arra, hogy milyen szabályok alapján növekszik egy páfrány. Miként alakítja csipkés, szabályos leveleit. Matematikához is értő kutatók felállítottak két egyszerű képletet:

$$x_{n+1} = a * x_n + b * y_n + e$$

$$y_{n+1} = c * x_n + d * y_n + f$$

A képletben x és y egy síkbeli pont koordinátái, a,b,c,d,e,f pedig alkalmas konstansok.

A képletek alapján számolt pontsorozat az alábbi képet adja.

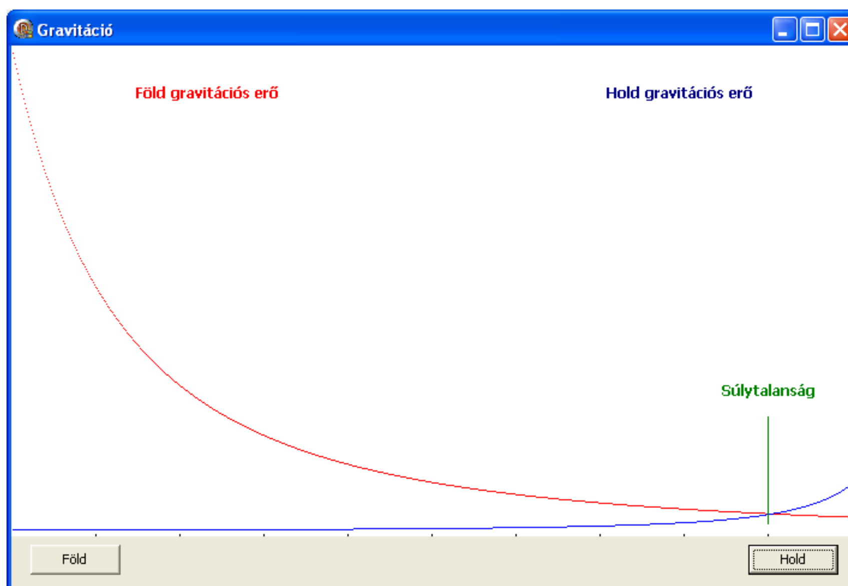


6. ábra: A páfrány levelét megrajzoló program képernyőképe

Egy másik példa: Tudni szeretnénk, hogy a Föld-Hold tengelyen a Földtől milyen távolságra következik be a súlytalansági állapot. Erre egy egyenletet állíthatunk fel:

$$g_f = g_h \ ahol \ g_f = \frac{k_f}{x^2} \ és \ g_h = \frac{k_h}{x^2}$$

A súlytalanság a piros és kék görbe metszéspontjában következik be.



7. ábra: A gravitációt modellező program képernyőképe

Láthatjuk, hogy két külön tudományterületről vett problémához miként lehet matematikai modellt felállítani.

3.2 Algoritmusszerkesztés

Az „algoritmus" kifejezés a IX. századbeli Ibn Musza Al Chowarizmi híres arab matematikus nevének latin változatából származik. Az „algebra" szó Al-jebr w'al mukabalah munkájának címéből származik. A cím annyit jelent, mint az „átalakítás és visszavezetés módszere" amely az egyenlet tagjainak, az egyik oldalról a másikra való átvitelére és az egynemű tagok összevonására vonatkozik. Ő vezette be az ismeretlen mennyiségek betűkkel és szimbólumokkal való jelölését és az ilyen szimbólumokkal történő számításokat és először foglalta szabályokba, hogy a tízes számrendszerben hogyan lehet a négy alapműveletet elvégezni.

Az algoritmus legáltalánosabb értelemben nem más, mint tervszerűség, röviden tervkészítés. Ha egy elvégzendő cselekvéssorozatot lépésről lépésre előre átgondolunk, megtervezünk, úgy is mondhatjuk, hogy algoritmust adunk meg egy

 © Aradi, Bécsi, Gyenes, Péter, BME

bizonyos cél elérésére, (arra, hogy milyen lépéseket fogunk végrehajtani, esetleg a korábbi lépések eredményétől függően).

A számítógépek az algoritmusok alkalmazhatóságát sokszorosára növelték. A gépek megjelenése megváltoztatta a helyzetet: sokkal nagyobb, bonyolultabb, számítási, adatfeldolgozási feladatokat tudnak megoldani, mint az ember magában, de ehhez pontosan, egyértelműen megfogalmazott programra, algoritmusra van szükségük. Ezek az algoritmusok a gépek nélkül gyakorlatilag használhatatlanok, ezért korábban a megfelelő feladatok fel sem vetődtek.

Az algoritmus tulajdonságai:

általános: az algoritmusnak nemcsak egy feladatot kell megoldania, hanem az összes feladatot az adott feladatkörből; (például az $ax^2+bx+c=0$ másodfokú egyenlet gyökeit kiszámító algoritmus bármilyen a, b, c értékekre helyes eredményeket kell szolgáltasson.)

véges*: a kezdeti információt véges számú transzformációs lépések alakítják át a megoldást jelentő véginformációvá, vagyis az algoritmusnak, mikor egy konkrét feladatra alkalmazzuk, véges számú lépés után be kell fejeződnie;

egyértelmű: minden közbeeső transzformációt, ami az előző állapotot a következővé alakítja át, az algoritmus szabályai egyértelműen határoznak meg;

megvalósítható: a feladat megoldása csak a rendelkezésre álló eszközöket igényli; (számítási eszköz lehet az egyszerű logarléc, vagy a legbonyolultabb számítógép).

A modell bonyolultságától függően a feladat megoldását elemi lépésekre kell bontani. A program elemi utasítások sorozata. Azt az eljárást, amelynek során a számítási feladatot mozzanatokra bontjuk, algoritmusszerkesztésnek nevezzük. Az algoritmus mozzanatait, amelyek egy-egy gépi utasításnak, vagy utasítás csoportnak felelnek meg négy csoportba oszthatjuk, és konvenció szerint meghatározott szimbólumokkal jelöljük.

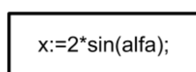
* A matematikai algoritmusoknál ez a megszorítás nem szükséges. Pl. az analízisben számtalan olyan algoritmus van, amely a határérték fogalmát használja, tehát elvben végtelen sok lépést követel (ha az elméleti pontosság elérésére törekednénk).

3.2.1 Folyamatábra szimbólumok

Technikai jellegű utasításoknak nevezzük az úgynevezett nem végrehajtható utasításokat. Ilyenek a program indítása, definíciók, deklarációk, program leállítása, stb. E típusú utasítás szimbóluma az ellipszis.

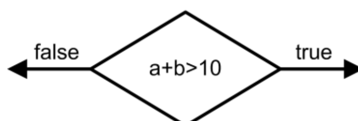
8. ábra: A program indítását jelző szimbólum

Aritmetikai utasítások során a program valamely érték számítását végzi, például az x:=2sinα érték kiszámítását az alábbi szimbólum jelöli.



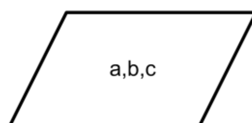
9. ábra: Aritmetikai utasítást jelző szimbólum

A harmadik csoportba a logikai utasítások tartoznak, amelyek a programban egy döntés nyomán elágazást valósítanak meg. Ezen logikai utasítások kimenete igaz (true) vagy hamis (false) lehet. Pl. az $a + b > 10$ logikai mennyiséget program elágazásra (branch) használó szimbólum az alábbi.



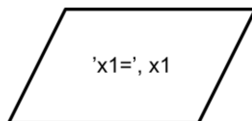
10. ábra: Elágazást jelző szimbólum

Negyedik csoportba az adat beviteli (Input) illetve adatkiviteli (Output) utasítások tartoznak. Ezek lehetnek billentyűzet, képernyő, vagy lemezegység.



11. ábra: Példa az adatbevitel jelölésére

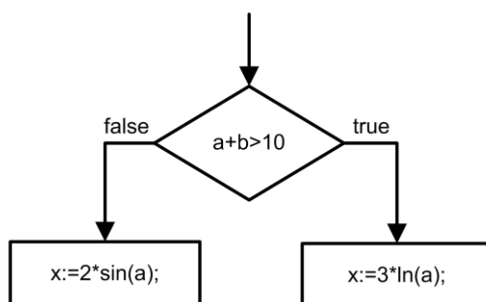
 © Aradi, Bécsi, Gyenes, Péter, BME



12. ábra: Példa az adatkivitel jelölésére

Az adat be- illetve kivitelt jelölhetjük még trapézzal, valamint a képernyőn történő kijelzésnél találkozhatunk egy speciális jelöléssel, amelyre (többek között) a 14. ábraán láthatunk példát.

Az egyes utasítások sorrendjét nyíllal jelöljük.



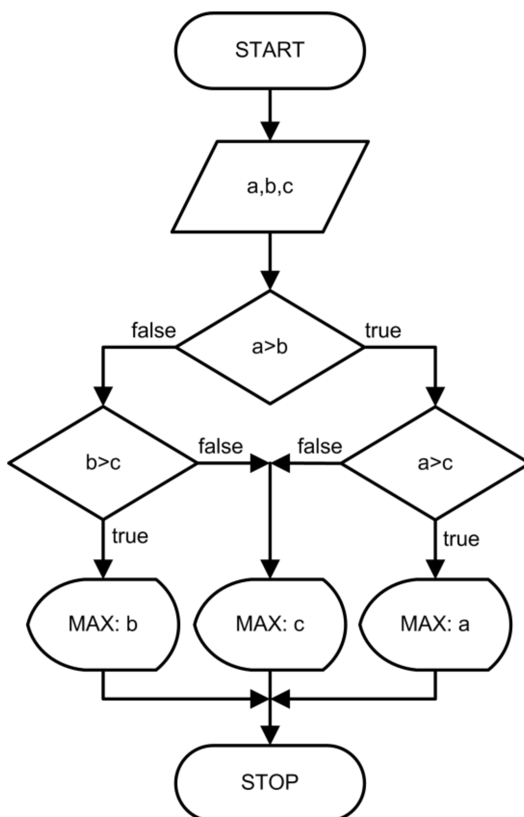
13. ábra: Példa a nyilak használatára

Ezekkel a szimbólumokkal megszerkesztett programot folyamatábrának (flowchart) nevezzük.

Az így megszerkesztett folyamatok programnyelvtől függetlenek, mi jegyzetünkben didaktikai okból mégis a Pascal nyelv szintaktikáját fogjuk követni.

3.2.2 Elágazásos programok

Példaként készítsünk folyamatábrát, amely három beolvasott szám közül kiválasztja a legnagyobbat.



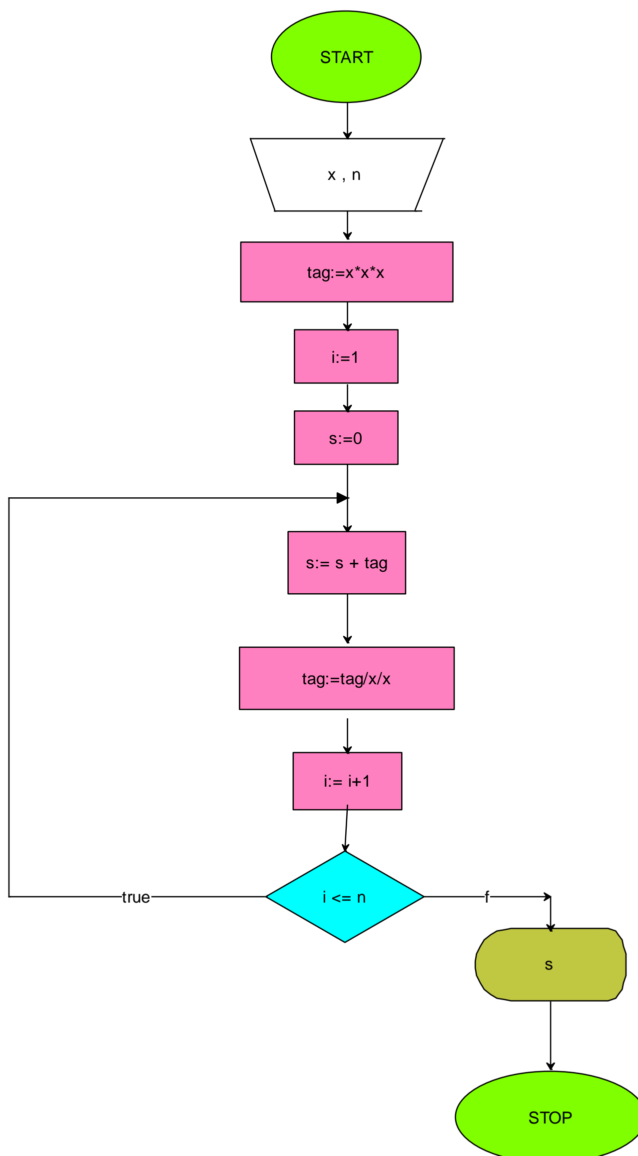
14. ábra: Három szám maximumának meghatározása

 © Aradi, Bécsi, Gyenes, Péter, BME

3.2.3 Egyszeres ciklus

Készítsünk folyamatábrát, amely alapján cos(x) értéke ε pontossággal számítható. A cos(x) értékét Taylor sorával számíthatjuk ki:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k!} \cdot x^{2k}$$



15. ábra: Cos(x) Taylor-sorba fejtésének folyamatábrája

3.2.4 Kétszeres ciklus

A kétszeres ciklus alatt az egymásba ágyazott cikluspárt értjük. Ez azt jelenti, hogy a külső ciklus minden egyes lépésére a belső ciklus lépéseit végre kell hajtani. Leggyakoribb eset a kétméretű tömbök feldolgozása, ahol n sor, és minden sorban m elem található. Ekkor a ciklusmag utasítás végrehajtásának száma n*m lesz.

Tekintsünk erre egy példát. Legyen egy tanulókörben n hallgató. Minden hallgató m tárgyból szerzett jegyet. Számítandó az egyes hallgatók tanulmányi átlaga. Az adatok az alábbi formában adottak:

Jegyek tömbje (a)	Tárgy 1	Tárgy 2	Tárgy j	Tárgy m
Hallgató 1	Jegy 1,1	Jegy 1,2	Jegy 1,j	Jegy 1,m
Hallgató 2	Jegy 2,1	Jegy 2,2	Jegy 2,j	Jegy 2,m
Hallgató i	Jegy i,1	Jegy i,2	Jegy i,j	Jegy i,m
Hallgató n	Jegy n,1	Jegy n,2	Jegy n,j	Jegy n,m

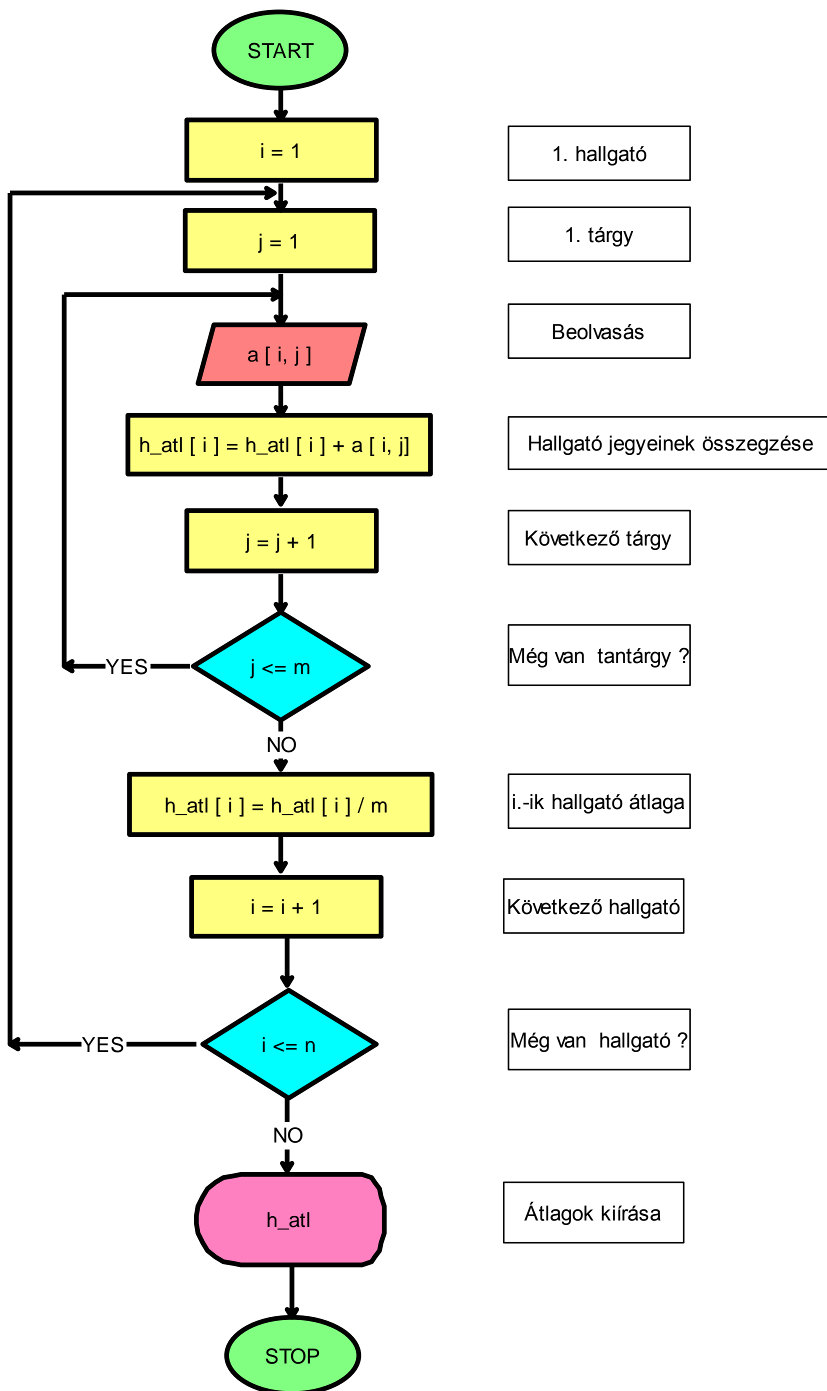
19. táblázat: A jegyek tömbje táblázatos formában

Megfigyelhetjük, hogy a sorindexet i jelöli, ahol i = 1..n, míg az oszlopindex jele j = 1..n. Az általános elem a i,j . Hallgatói átlagból n érték adódik. Ez egyindexes vektor lesz : h_atl i, ahol i = 1..n .

A tényleges programban az átlagok kezdeti értékét 0-ra kell állítani. Láthatjuk, hogy a beolvasást összesen n*m esetben hajtja végre a program. Az átlagok kiíratása egyszeres ciklussal történik, jelen folyamatábrában az egyszerűség kedvéért ez sincs feltüntetve.

A feladat megoldásának algoritmusa az alábbi lehet.

 © Aradi, Bécsi, Gyenes, Péter, BME



16. ábra: A hallgatók tanulmányi átlagát számító algoritmus folyamatábrája

4 A Pascal nyelv története

A Pascal magas szintű programozási nyelv, amelyet N. Wirth alkotott meg Zürichben az 1960-as és 70-es évek fordulóján. A nyelvet Blaise Pascalról, a 17. századi híres francia matematikusról és filozófusról nevezte el. Az Algol (Algorithmic Language) logikájából kiindulva, azokat - főleg az adatszervezés terén - új elemekkel bővítve, remek programnyelvet alkotott. A Standard Pascal meglehetősen puritán nyelv volt, csak a legszükségesebb parancsokat és utasításokat tartalmazta, a programozónak meglehetősen sok munkája volt. A programnyelv azonban valóban jól használható, közel áll az emberi gondolkodáshoz és könnyen elsajátítható és jól tanítható.

Rohamtempóban terjedt el a világban - főleg a személyi számítógépek megjelenése óta. Szoftvercégek sora adta ki saját változatát, kiegészítve újabb parancsokkal, függvényekkel, miközben a kezelését egyre kényelmesebbé tették, integrált keretrendszert alkottak hozzá. Az egymással konkuráló szoftvercégek Pascal változatai közül kétségtelenül a Borland cégé lett a legsikeresebb. Az évek során a Turbo Pascal verziók között ma a 8.0-nál tartanak, de már évekkel ezelőtt megjelent a Windows-os változat is.

4.1 A Pascal nyelv generációi

Az első változat 1968-ban jelent meg az ALGOL60-ra alapozva. Az első használható fordító 1970-ben készült el. A forrásnyelvű program ekkor még soreditorral vagy más szövegszerkesztővel készült.

Később 1973-ban jelent meg egy, a tapasztalatok alapján továbbfejlesztett változat. 1983-ban pontos szabvány definiálta a Pascal nyelvet. Ezt szintén a nyelv atyja Niklaus Wirth publikálta Zürichben, a Pascal szintaktika alapjait később más programnyelvek is átvették (Ada, Chill, Delphi).

Az első változat még második generációs központi számítógépen futott. A programozó csak külső adathordozóval (lyukszalag) érintkezett a számítógéppel, amit operátor kezelt. A kimeneti eszköz eleinte konzol írógép volt, amit később felváltott a sorszervezésű karakteres monitor.

A személyi számítógép megjelenése és elterjedése eleinte a Basic nyelv használatát tette lehetővé. Amint azonban az Intel és a Motorola kifejlesztette 16 bites processzorait (AT286, ill. Motorola 68000) már elegendő erőforrás állt rendelkezésre a Pascal nyelv használatához.

Népszerű lett a Turbo Pascal, amely a korábbinál jelentősen gyorsabb fordítást, és futást eredményezett. Ehhez sok kiegészítő külső modul, un. Unit (CRT,DOS,GRAPH, PRINTER, stb) íródott, amelyet könnyen lehetett a programhoz csatolni.

Az Object Pascal az objektum orientált programozás (OOP) eszköze, amellyel a valóságot legjobban leíró objektumok használata vált lehetővé.

A későbben megjelenő Delphi programnyelvbe is be van építve a Pascal (Console Application) a nyelv szintaktikájának elsajátítása érdekében.

4.2 A Pascal különböző szintjei

A nyelv kedvező fogadtatása, és jó használhatósága miatt az évek során új és új verziók láttak napvilágot. A felhasználók igényeit kiszolgáló lehetőségek beépítésével jól strukturált nyelv jött létre. Alapvetően 3 szintet különböztethetünk meg.

4.2.1 Hivatkozási szint

Ez a szabványban rögzített forma. Az átlag programozónak ezt nem kell ismernie. A fordító programok készítői használják ezt a matematikában szokásos szimbólumrendszerrel definiált szabályrendszert. Vitás kérdésekben a hivatkozási szintű leírás a kérdés eldöntésének alapja.

4.2.2 Publikációs szint

A publikációs szint a nyelv szintaktikai szabályrendszerét tartalmazza közérthető formában. Itt az utasítások formái vannak leírva, amit a programozó az alkalmazás készítésekor használ. Ez a publikációs szint a Pascal oktatás alapja.

4.2.3 Gépi reprezentáció

A gépi reprezentáció a Pascal nyelv egy adott számítógépen futó változata. Ez nagyban eltérhet attól függően, hogy un. nagyszámítógépen (korábban RMT-286, IBM370, ICL-4, Honeywell), vagy PC-n (IBM-PC vagy Macintosh) fut. A gépi reprezentáció függ a használt operációs rendszertől is (pl. DOS, Unix, Windows, Linux). Az egyes forgalmazók szintén kialakítják a saját változatukat. A programozónak ismernie kell az általa használt gép jellemzőit, és az ahhoz illeszkedő program változatot.

Sok nehézséget okoz, ha egy alkalmazásnak több, különféle környezetben kell működnie.

4.2.4 Fejlesztői környezet

Fontos, hogy már tanulásunk legelején használni tudjuk a számítógépünket Pascal környezetben. Ez lehetővé teszi, hogy a jegyzetben található mintapéldákat és feladatokat rögtön (on-line módon) kipróbálhassuk.

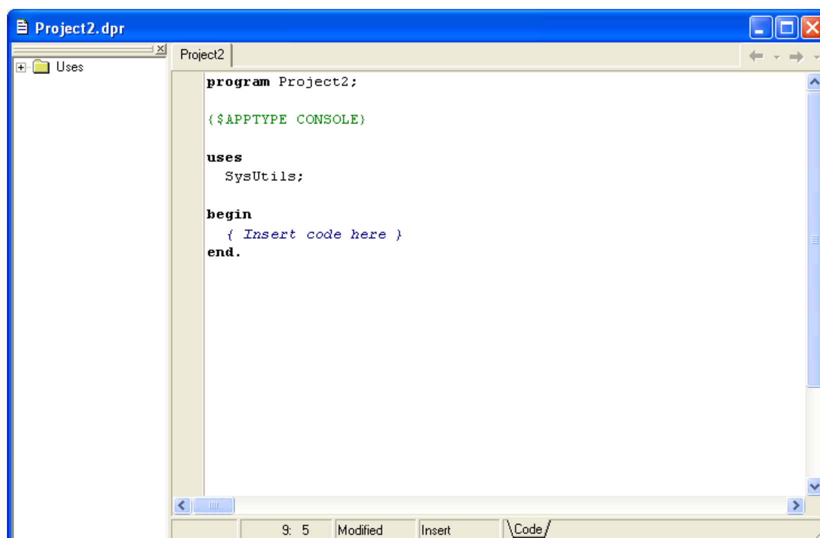
Javasolhatjuk bármely Borland cég által forgalmazott, és ingyenes Turbo Pascal változatot. Ezek a programok az újabb Windows rendszerek alatt nehézkesen használhatók, mivel DOS operációs rendszer alá készültek.

Jobb megoldás lehet az ugyancsak Borland cég által forgalmazott Delphi, vagy Turbo Delphi használata.

Jelen jegyzetben tárgyalt Számítástechnika 1. c. tantárgy az algoritmikus programozás alapjainak és a Pascal nyelv szintaktikájának elsajátítását tűzte ki célul.

Ehhez a Delphi Console Application alkalmazás-fejlesztői környezetét használjuk. A Delphi indítása után ezt a lehetőséget a File / New / Console Application menüpont alatt találjuk meg.

Az alábbi futási képet látjuk. A főprogram elejét a **begin** kulcsszó, míg végét az **end**. jelzi.



17. ábra: A Console Application projekttípus kezdőképe

Fontos, hogy a {$APPTYPE CONSOLE} fordító direktívát és a **uses** SysUtils; parancsot ne töröljük ki.

A Pascal adott gépi reprezentációjának megismerését segíti a Delphi Help menü, ahol angol nyelvű tájékoztatót és némely esetben mintapéldát is találunk.

5 A Pascal nyelv strukturális felépítése

A nyelv szintaktikájának könnyebb elsajátíthatósága végett különböző részekre bontjuk a nyelvet. A legegyszerűbb elemektől haladunk a bonyolultabb struktúrák felé. Ezt a felosztást több évtizedes oktatási tapasztalatunk alapján alakítottuk ki. Az egyes részeket szinteknek fogjuk nevezni, melyet ne tévesszünk össze a nyelvi formációk szintjeivel (4.2. fejezet).

5.1 Építőelemek

A nyelv legalsó szintjén az építőelemeket találjuk. Ezek alkotják a nyelv alapját. Minden magasabb szint ezekből építkezik.

5.1.1 Számok

A számok a decimális számrendszer digitjei: 0,1, ..9

A Pascal értelmezi a Hexadecimális számjegyeket is: 0,1, .. 9, A,B,C,D,E,F ($ prefixum használatával).

5.1.2 Betűk

Az angol ABC kis- és nagy betűi: a..z és A..Z. A Pascal nyelvtanilag nem különbözteti meg a kis- és nagy betűket. A Pascal a betűk közé sorol még két karaktert is : @ (at karakter) és _ (underscore).

5.1.3 Szimbólumok

A szimbólumok a betűk és számok mellett építik fel a nyelv magasabb szintű elemeit:

Egykarakteres szimbólumok:

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

Kétkarakteres szimbólumok:

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

Különleges szimbólumok:

% ? \ ! " (double quotation mark) | (pipe) ~ (tilde).

© Aradi, Bécsi, Gyenes, Péter, BME

5.1.4 Karakterek

Az ASCII kódtáblában megjelenő többi karakter, amelyet a Pascal nyelvtanilag nem használ.

Pl. ␣ (szóköz : space), amely nyomtatásban nem látható

á β ` stb.

5.1.5 Azonosítók

Az azonosító (Identifier) szimbolikus név, azaz alfanumerikus karaktersorozat, amely alfabetikus karakterrel kezdődik, változók, típusok, konstansok, stb. neveit jelöli. A név maximális hossza a gépi reprezentáció függvénye, a Pascal 63, de a Delphi 255 karaktert enged meg.

Példák a helyes azonosító megadásra:

alfa3 ZOLI summa_4 _cimke

Példák a hibás azonosító megadásra:

4xy (számmal nem kezdődhet)

Jolly joker (szóközt nem tartalmazhat)

Begin (foglalt szó nem lehet)

Az alábbi felsorolás tartalmazza a Pascal védett (reserved) szavait abc sorrendben.

and	array	as	asm	begin
case	class	const	constructor	destructor
div	do	downto	else	end
except	file	finalization	finally	for
if	implementation	in	inherited	inline
interface	is	label	library	mod
nil	not	object	of	or
out	packed	procedure	program	property

© Aradi, Bécsi, Gyenes, Péter, BME

raise	record	repeat	set	shl
shr	string	then	threadvar	to
try	type	unit	until	uses
var	while	with		

5.1.6 Adattípusok

A Pascal számos adattípust használ. Ezeket többféleképpen csoportosíthatjuk.

5.1.6.1 Standard, skalár adattípusok

A standard típusokat a nyelv előre definiálja, és felkínálja a programozónak. A skalár (simple) típusnál egy azonosító egyetlen értéket tárol.

Itt csak a leggyakrabban használt standard (szabványosított) típusokat vesszük számba.

a) <u>Egésztípusok</u>

Típus	Tartomány
byte	0..255
shortint	-128 - 127
word	0..65535
smallint	-32768 - 32767
integer	-32768 – 32767 (Delphiben eltérő)
longint	-2147483648 - 2147483647
int64	$-2^{63}..2^{63}-1$

20. táblázat: Egész adattípusok

Fontos megjegyezni, hogy az integer típus tartománya a fordítótól és az operációs rendszertől függően különböző lehet. Így Delphiben (Win32) az integer 32 bit hosszú, azaz a tartománya -2147483648-tól 2147483647-ig terjed. Továbbá a Delphiben elérhető egy 64 bites típus is, int64 néven, melynek tartománya: $-2^{63}..2^{63}-1$.

 © Aradi, Bécsi, Gyenes, Péter, BME

b) Valós típusok

Típusa	Tartomány
single	$-1.5 * 10^{45}$ .. $3.4 * 10^{38}$
real	$-2.9 * 10^{39}$ .. $1.7 * 10^{38}$
double	$-5.0 * 10^{324}$ .. $1.7 * 10^{308}$
extended	$-3.6 * 10^{4951}$ .. $1.1 * 10^{4932}$

21. táblázat: Valós adattípusok

A valós típusnál szintén van különbség az egyes fordítók és operációs rendszerek esetén, Delphiben (Win32) a real típus megegyezik a double típussal.

c) Karakter típus

char: Az ASCII karakter kódtábla bármely eleme.

d) Logikai típus

boolean: Csak a két logikai érték (true, false) valamelyikét veheti fel.

5.1.6.2 Nem standard, skalár adattípusok

Ezeket a típusokat a programozó határozza meg ún. típus definícióval.

a) Felsorolásos (enumerated) típus

A programkészítő azonosítók felsorolásával definiál típust. Például a színek típusa lehet: red, green, blue, yellow, vagy napok típusa: mon, tue, wed, thu, fri, sat, sun.

b) Intervallum típus

A programozó egy korábbi megszámlálható típus részhalmazát definiálja. Például a pozitív egészek típusa lehet a 0..100 intervallum, vagy a kisbetű típus: ’a’ .. ’z’. Figyeljük meg, hogy a .. szimbólum jelentése tól – ig.

5.1.6.3 Összetett adattípusok

Ezek a típusok egy azonosítóval az értékek egy csoportját jelölik.

a) Tömb típus

Azonos típusú értékek közös néven történő kezelésére való. Például a *vector* néven létrehozott **array [1..10] of integer** típus 10 db egészszámot tartalmaz. Az egyes elemekre indexes azonosítóval hivatkozhatunk, például a *vector[3]* a tömb 3. elemét jelöli.

b) String típus

Maximálisan 255 karaktert tartalmazó csoport közös néven való kezelésére való. Például a *name* azonosítójú **string[25]** típusú változó maximum 25 karakterből álló nevet jelöl. E típus egyes karaktereire indexes azonosítóval is hivatkozhatunk, például a *name[6]* a string 6. karakterét jelöli.

Fontos megjegyezni, hogy Delphiben a méret nélkül deklarált string típusú változó mérete jóval nagyobb is lehet, amennyiben a memória mérete lehetővé teszi, akár 2 GB is lehet.

c) Rekord típus

Különböző típusú elemek közös néven való kezelésére való. Például legyen a *student* nevű rekord típus az alábbi :

```
record
  name : string[30];
  age  : byte;
end;
```

A rekord elemeit (name, age) mezőnek (field) nevezzük. A mezőkre való hivatkozás eszköze a pont. Például a *student.age* a második mezőt azonosítja.

Az összetett adattípusokkal a későbbiek során gyakran fogunk találkozni.

5.1.7 Konstansok

A Pascal azokat az azonosítókat nevezi konstansoknak, amelyek a program futása során értéküket nem változtatják meg. (Erről ezen adatok elhelyezése gondoskodik: a fordító a konstansokat a kódszegmenesbe – ld. szegmens kiosztás - helyezi el.)

 © Aradi, Bécsi, Gyenes, Péter, BME

A konstans lehet explicit (tipizált) és implicit (nem tipizált) is. Az explicit konstans típusát a programozó határozza meg. Például a

num : **integer** = 12

alakban megadott konstans integer lesz és értéke 12. Az implicit konstans definíciója esetében a típust a fordító határozza meg. Például a

yes = true

estén a *yes* azonosító *boolean* típusú lesz..

A logikai mennyiségekhez 2 előredefiniált konstans a true és a false tartozik.

5.1.8 Standard függvények

A Pascal számos előre definiált (standard) függvényt kínál a programozónak. Mi jegyzetünkben ezeket itt, az építőelemek között tárgyaljuk (a függvényekről a későbbiekben részletesen tanulunk az 5.4.1. pontban). Az alábbiakban felsoroljuk a legelemibb függvényeket.

5.1.8.1 Aritmetikai függvények

Abs(x): x abszolút értékét adja.

Exp(x): ex értékét számolja, e a természetes logaritmus alapja (Euler konstans), e = 2,718 281.

Frac(x): x valós szám törtrészét adja.

Int(x): x valós szám egészrészét adja valós típusban.

Ln(x): x természetes logaritmusát adja.

Pi: π értékét adja (3.1415926535897932385).

Round(x): x kerekített értékét adja egésztípusra konvertálva.

Sqr(x): x^2 értékét adja.

Sqrt(x): x négyzetgyökét adja.

Trunc(x): x egészrészét adja egésztípusra konvertálva.

5.1.8.2 Karakter függvények

Chr(x): x egészszámot ASCII kódként értelmezve annak karakter megfelelőjét adja.

Ord(ch): ch karakter ASCII kódját adja.

UpCase(ch): ch kisbetű karaktert nagybetűre konvertálja.

5.1.8.3 Ordinális függvények

Inc(x): x értékét eggyel növeli (x = x +1) (vigyázat ez eljárás, ld. 5.4.2. pont).

Inc(x,n): x értékét n-el növeli (x = x +n) (vigyázat ez eljárás, ld. 5.4.2. pont).

Odd(x): logikai függvény, értéke true, ha x páratlan.

Ord(x): x sorszámát adja abban a megszámlálható halmazban, amelynek eleme.

Pred(x): x -et megelőző elemet adja egy megszámlálható halmazban.

Succ(x): x-et követő elemet adja egy megszámlálható halmazban.

5.1.8.4 Trigonometrikus függvények:

Cos(x): x radiánban értelmezett szög cos értékét számolja.

Sin(x): x radiánban értelmezett szög sin értékét számolja.

5.1.9 Címke

A nyelv programhelyek megjelölésére címkét (label) használ. Ez a címke lehet azonosító, vagy egészszám is. A címke után mindig kettőspont áll, például *ide:*.

A későbbiek során még számos standard és nem standard típussal találkozunk.

Gyakorló feladatok

F01. Tervezzünk rekord típust egy gépkocsi jellemzőinek kezelésére.

F02. Készítsünk alkalmas típust az év hónapjaihoz.

Ellenőrző kérdések

K01. Melyek a valós típusok?

K02. Mi a rekord adattípus meghatározása?

K03. Mit értünk tömb típus alatt?

5.2 Kifejezések

A nyelv második szintjén a kifejezések találhatók. A kifejezés (expression) definíciója következő.

Kifejezés : építőelemek és operátorok szintaktikus kombinációja.

Az operátorok műveletet írnak elő kifejezések számára. Végrehajtásuk meghatározott sorrendben (precedencia vagy prioritás) történik. Ha egy kifejezésben több operátor van, először a legmagasabb prioritású operátor lesz végrehajtva. Ezért fontos, hogy ismerjük az operátorok prioritás sorrendjét.

a) Elsődleges operátorok

A legmagasabb prioritású a not operátor, amely elsősorban logikai típusú mennyiségekre vonatkozik, hatására a logikai érték ellenkezőre vált (true - false és false - true). A Pascalban ugyancsak elsődleges a + vagy - előjel (sign), amelyet a számérték követ.

b) Másodlagos operátorok

Ide tartoznak a multiplikatív operátorok, melyek egymással azonos prioritásúak:

*** / div mod and**

A * operátor (asterix) a szorzás műveletét írja elő mint a matematikában, de itt általánosabb felhasználása lesz, mivel a halmazok metszetének meghatározására is használja a Pascal.

A / operátor (slash) valós típusú osztást ír elő aritmetikai mennyiségek között, hatása megegyezik a matematikában szokásos osztás művelettel.

A **div** operátor jelentése egésztípusú osztás, amely csak egésztípusú mennyiségekre van értelmezve.

A **mod** operátor (modulo) jelentése : maradék számító osztás. A művelet eredménye az egészosztás elvégzése utáni maradék.

Az **and** operátor logikai szorzás (ÉS művelet), amely logikai típusú mennyiségekre vonatkozik. Két logikai mennyiség esetén igazságtáblázata az alábbi.

A	false	true	false	true
B	false	false	true	true
A **and** B	false	false	false	true

22. táblázat: Az ÉS művelet igazságtáblázata

c) Harmadlagos operátorok

Ide tartoznak az additív operátorok.

+ - or xor

A + és - operátorok a matematikában megszokott módon összeadást és kivonást végeznek. A Pascal ezeket az operátorokat használja halmazokkal végzett műveleteknél is (ld. halmaz típusú adatok).

Az **or** operátor logikai összeadás (VAGY művelet), amely logikai típusú mennyiségekre vonatkozik Két logikai mennyiség esetén igazságtáblázata az alábbi.

A	false	true	false	true
B	false	false	true	true
A **or** B	false	true	true	true

23. táblázat: A VAGY művelet igazságtáblázata

A **xor** operátor logikai antivalencia (kizáró vagy művelet), amely logikai típusú mennyiségekre vonatkozik Két logikai mennyiség esetén igazságtáblázata az alábbi.

A	false	true	false	true
B	false	false	true	true
A **xor** B	false	true	true	false

24. táblázat: Az ANTIVALENCIA művelet igazságtáblázata

Ez a művelet az ekvivalencia negált művelete. A Pascal a logikai algebrával szemben nem alkalmaz külön ekvivalencia operátort mert ezt a **not** és **xor** alkalmazásával oldja meg.

d) Negyedleges operátorok

Ide tartoznak a relációs operátorok. Ezek állnak a prioritási sor végén. A relációs operátorok végrehajtása minden esetben logikai értéket (true v. false) eredményez.

= <> < > <= >= in

Az = operátor egyenlőséget (equality) vizsgál két azonos típusú mennyiség között.

A <> operátor az ún. nem egyenlő szimbólum, az előbbi ellentett művelete.

A < operátor (less than) és a > (greater than) használata a matematikában szokásos módon történik.

A <= operátor (less or equal) és a >= (greater or equal) operátor használata is a matematikában szokásos módon történik.

Az **in** logikai operátor azt vizsgálja, hogy egy érték benne van-e egy adott halmazban. Például a *8* ***in*** *[3..12]* kifejezés értéke true:

A teljesség kedvéért megemlítjük a bitmanipulációs operátorokat is, amelyek egy vagy két egészadatot bitenként módosítanak. Ezek használatát csak speciális feladatok igénylik.

not: egészszám bitenkénti negálása.

and: két egészszám bitenkénti szorzata (and szabály szerint).

or : két egészszám bitenkénti összeadása (or szabály szerint).

xor: két egészszám bitenkénti antivalenciája (xor szabály szerint).

shl: egészszám balra tolása.

shr: egészszám jobbra tolása.

Fontos szabályok a kifejezések használatához!

Szabály 1: Az operátorok két oldalán csak azonos típusú, vagy azonosra konvertálható építőelem állhat.

Szabály 2: Az egy kifejezésben előforduló több azonos prioritású operátort a leírás szerint balról jobbkéz felé értékeli ki a Pascal. Ez a precedencia szabály.

Szabály 3: Egy kifejezés, vagy annak része elsődlegessé tehető kerek zárójelbe téve.

Szabály 4: Kifejezés a programban önállóan nem állhat.

Gyakorló feladatok

F01. Határozzuk meg az alábbi kifejezések aktuális értékét!

Succ(9), **Pred**(True), **Ord**('A'), **Round**(4.68), **Trunc**(4.68), 13 **div** 4,
13 **mod** 4, 4 **mod** 13

F02. Egy felsorolásos típus elemei (piros, kek, zold, sarga), számítsuk ki az alábbi értékeket!

Ord(kek), **Succ**(zold), **Pred**(piros).

F03. Keressük meg az alábbi kifejezésekben lévő hibákat!

Chr(300), 6 + 4.2 * **tan**, 3 < x < 10, (z = 2) **and** (z < 'z')

F04. Írjuk fel helyesen az F03. hibás kifejezéseit!

Ellenőrző kérdések

K01. Mi a Pascal második szintje és mi annak definíciója?

K02. Melyek a multiplikatív operátorok?

K03. Mely operátoroknak van a legalacsonyabb prioritása?

K04. Mi a precedencia szabály?

5.3 Utasítások

Utasítás (statement) a programnyelv legkisebb önálló egysége.

Definíció: Az utasítás a kifejezések és Pascal parancsszavak szintaktikai kombinációja, amely a program számára valamilyen műveletvégzést ír elő. A Pascal az utasításokat pontosvessző (semicolon) zárja.

Számos utasítás létezik, először az egyszerűbbeket tárgyaljuk. Azokat az utasításokat vesszük előre, amelyek az első, egyszerűbb alkalmazások készítéséhez feltétlenül szükségesek.

5.3.1 Értékadó utasítás

Az értékadó utasítás (assignment statement) értéket rendel egy azonosítóhoz. Alakja

```
azonosító := kifejezés;
```

Az utasítás baloldalán csak változónév (építőelem) vagy rekord mezőleíró állhat. Az értékadást a : = kétkarakteres szimbólum (legyen egyenlő) jelzi. Az értékadás jobb oldalán kifejezés áll, amelyet a program kiszámít és a kapott értéket a változóba tölti, például

```
y:= 3*x; st[4] := chr(23); b:= x = y; betu := 'M';
```

Fontos szabályok!

Szabály 1: Az értékadás 2 oldalán azonos típusú, vagy azonosra konvertálható mennyiségek állhatnak.

Szabály 2: Többszörös értékadás a Pascalban nincs (x := y := 2; hibás értékadás).

5.3.2 Deklaráció utasítás

A deklaráció ún. nem végrehajtható utasítás. Ez a fordító számára jelent feladatot: memória terület lefoglalását a bemutatott változók számára. A változó deklaráció kulcsszava a **var** (variables), szintaktikája a következő.

```
var azonosító(k) : típus ;
```

Egy utasítással egy, vagy több azonos típusú változó mutatható be. A Pascalban minden változót deklarálni kell.

Skalár változók deklarálása:

var x,y : single;	x és y valós típusú változók bemutatása
var n : byte;	n egésztípusú változó bemutatása
var ch: char;	ch karakter típusú változó bemutatása
var yes, b1: boolean;	yes és b1 logikai változók bemutatása
var inter : 10 .. 20;	inter nevű intervallum típusú változó bemutatása

Összetett adattípusú változók deklarálása

var vec : **array** [1..5] **of integer**;	vec 5 egésztípusú adat (elemei: vec[i] , ahol i = 1..5 lehet)
var matrix **array** [1..4, 1..6] **of char** ;	matrix 4 sorba és 6 oszlopba rendezett 24 karakter elemű tömb.
var sorok : **array** [boolean] **of single**;	sorok 2 valós elemet tartalmaz (sorok[false] és sorok[true])
var name : string [20];	name változó max 20 karakteres nevet tárol.

Sok más típus (pl. mutató, halmaz, rekord) is deklarálható, ezekkel később találkozunk.

A Pascalban a címkét is deklarálni kell. Ennek módja:

label ide,oda;

Gyakorló feladatok

F01. Deklaráljuk alkalmas változót, amellyel egy max. 20 hallgatós csoport Neptun kódjait kezelhetjük.

F02. Deklaráljunk alkalmas változót, amely egy 5-ös lotto szelvény tippelt számait tárolja.

Ellenőrző kérdések

K01. Mi a deklaráció utasítás feladata?

K02. Melyek az értékadás szabályai?

K03. Hogyan deklarálunk logikai változókat?

5.3.3 Kommentár utasítás

Célja : informatív megjegyzések beírása a programba, amiket a fordító nem vesz figyelembe. Nagyon ajánlott ezek használata a program érthetőségét segíti a programozó számára.

Az utasítás után írt // jeltől a sor végéig a programozó megjegyzése lesz például:

```
y := 2 * sqr (y - 4);                    // ez egy parabola egyenlete
```

Más módszerrel egész sorok tehetők megjegyzéssé:

```
{ ezt a fordító negligálja }             kapcsos zárójel (brace) pár
(* ezt a fordító negligálja *)           kerek zárójel (brackets) pár
```

Egész programrészek is kommentárrá tehetők (ifdef endif fordító direktíva):

```
{$ifdef nincs }
  ezt a sort a fordító negligálja
  meg ezt is
{$endif }
```

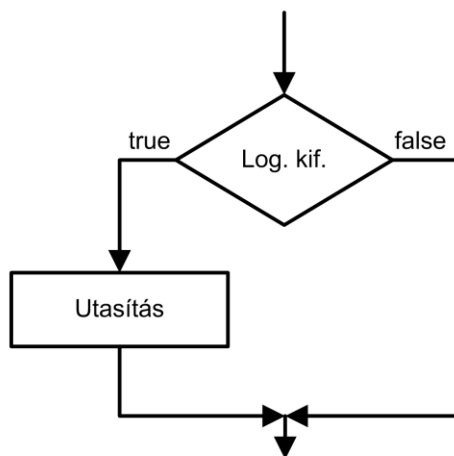
5.3.4 Feltételes utasítás

A feltételes utasítás (Conditional Statement) az utasításba beépített feltétel aktuális értéke szerint elágazást valósít meg. Két fajtája van, az egyszerűbb felépítése az alábbi:

if logikai kifejezés **then** utasítás;

Ha a logikai kifejezés aktuális értéke **true**, akkor az utasítás végrehajtása megtörténik, különben nem.

Ezen utasítás folyamatábra megfelelője az alábbi.



18. ábra: Az if-then utasítás folyamatábrája

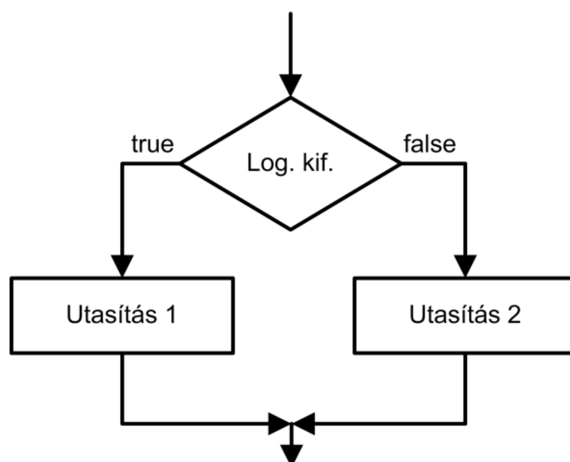
Például:

if a >= 2 **then** x := 5;

A másik fajta feltételes utasítás szintaktikai felépítése:

if logikai kifejezés **then** utasítás 1 **else** utasítás 2;

Ha a logikai kifejezés aktuális értéke **true**, akkor az utasítás 1, egyébként az utasítás 2 végrehajtása történik. Folyamatábra megfelelője az alábbi.



19. ábra: Az if-then-else utasítás folyamatábrája

 © Aradi, Bécsi, Gyenes, Péter, BME

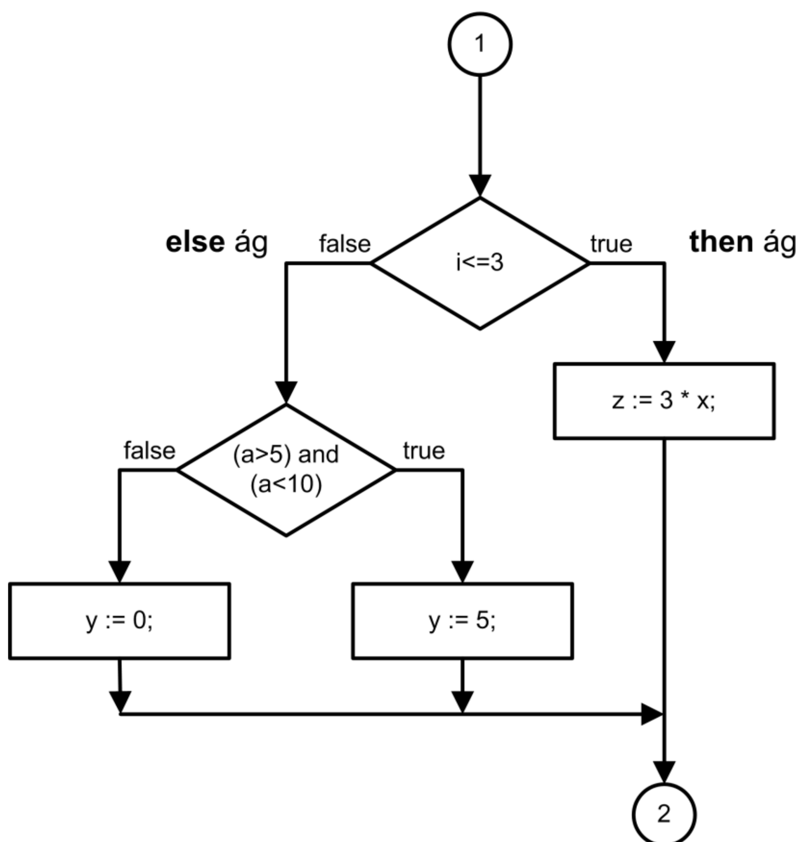
Például:

```
if a <> 0 then y := sqrt(x) else y:=13.5 ;
```

A **then** és **else** kulcsszavak mögött tetszőleges utasítás állhat, lehet újabb feltételes is. Erre az esetre mutatunk egy példát:

```
if  i <= 3 then z := 3*x
           else if (a>5) and (a<10) then y:=2-a
                                    else y:=0;
```

Gyakorlásképpen megadjuk ennek az utasításnak a folyamatábra megfelelőjét.



20. ábra: Példa az összetett feltételes utasításra

Gyakorló feladatok

F01. Írjuk fel Pascal alakban a következő összefüggést!

$$y = \begin{cases} \lg(x-2) & ha \;\; x > 2 \\ 18 & ha \;\; x = 0 \\ 3 * \cos(x) & ha \;\; x < 0 \end{cases}$$

F02. Számítsuk ki s értékét, ha a = 2 és b = 5.

If 3 + b **mod** a > 4 **then** s : = **sqrt** (a) **else** s := **sqr** (b);

Ellenőrző kérdések

K01. Mi a feltételes utasítás szintaktikája?

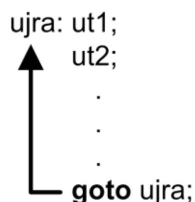
K02. Milyen operátorok használhatók a feltétel megadásához?

5.3.5 Vezérlésátadó utasítás

A vezérlésátadó utasítás (Control Statement) célja, hogy a program egy címkével megjelölt helyének adja át a vezérlést (végrehajtást). Gyakran ugró utasításnak is nevezzük, az assembler nyelvekben jump (vagy jmp) mnemonikkal jelöljük.

Szintaktika: **goto** címke;

Például:



Ne felejtsük el, hogy a címkét deklarálni kell (**label** ujra;)

5.3.6 Típus definíció

A típus definíció (type definition) nem végrehajtható utasítás, célja a programban a feladat adatainak leginkább megfelelő típus megadása (természetesen csak a nyelv szabályainak betartásával).

 © Aradi, Bécsi, Gyenes, Péter, BME

Szintaktika: **type** *típusnév = típusmegadás* **;**

A típusnév szimbolikus név lehet, amit célszerű a változóktól megkülönböztetni. Mi a típusnevet mindig ***t_nev*** alakban fogjuk használni.

Például:

type t_kisbetu = ’a’ .. ’z’ **;**	Kisbetűk típusa
t_pozitiv = 0 .. 10000**;**	Pozitív számok típusa amelyek < 10000
t_vektor = **array[**1 **..** 20**] of char;**	max 20 elemű vektor típus, minden eleme karakter.
t_vektor = **array**[1 .. 20] **of char**;	max 20 elemű vektor típus, minden eleme karakter.

5.3.7 Ciklus utasítások

A ciklus utasítások (loop statements) egy adott utasítást (vagy utasítás csoportot) ismételnek meg az előirt érték, vagy valamilyen feltétel függvényében. Jelentőségük nagy, mivel a gépi adatfeldolgozás egyik alapvető célja, hogy a sok ismétlődő (esetleg bonyolult) számítás terhét átvegye az embertől. Ezért a feladat jobb kiszolgálása érdekében több fajtát kínál a Pascal nyelv.

5.3.7.1 For ciklus

Ezt a ciklusutasítást akkor használjuk, ha a program írásának idején meg tudjuk határozni az ismétlések számát (valamely konstans, vagy változó aktuális értéke).

Szintaktika: **for** *cv* **:=** *kezd* **to** *vég* **do** *utasítás* **;**

A *cv* a ciklusváltozót jelöli, ami megszámlálható (ordinal) típusú mennyiség. A ciklusváltozó a kezdő értéktől lépked a típusnak megfelelő elemeken egyesével, míg a végértéket eléri, miközben mindannyiszor végrehajtja a **do** (tedd) parancsszó után megadott utasítást (vagy utasításcsoportot).

Fontos szabályok!

Szabály 1: A ciklusváltozó csak skalár, ordinal típusú (char, integer, byte, boolean, felsorolásos, interval) lehet.

Szabály 2: A ciklusváltozó a kijelölt zárt intervallum elemein egyesével lépked.

Szabály 3: A ciklusváltozó aktuális értékét a számítógép a ciklusmag végrehajtása előtt ellenőrzi.

Szabály 4: A ciklus lejárta után a ciklusváltozó értéke nem definiált.

Például

x:=5;	
for i :=10 **to** 5 **do** x :=2*x;	esetén az utasítás nem lesz végrehajtva, x = 5 marad.
sum:=0;	
for i:=2 **to** 6 **do** sum:=sum + i;	ciklusban az utasítás 5-ször lesz végrehajtva sum = 20 lesz.

Gyakorta szükség lehet arra, hogy a ciklusváltozó visszafelé lépkedjen az intervallumon. Erre egy módosított for utasítás szolgál.

Szintaktika: **for** *cv* **: =** *vég* **downto** *kezd* **do** *utasítás***;**

Például:

n:=1;	
for i := 4 **downto** 1 **do** n:=n*i;	ciklus lefutása után n=24 lesz.

5.3.7.2 While ciklus

Általános ciklusutasítás, amelynél nincs ciklusváltozó, az ismétlések számát feltétel határozza meg.

Szintaktika: **while** *logikai kifejezés* **do** *utasítás* ;

A ciklusmag (az utasítás, vagy utasítás csoport) a logikai kifejezés **true** értékénél ismételten végrehajtásra kerül.

Például:

x:=5;	
while x < 20 **do** x:=3*x;	végrehajtása után x = 45 lesz.

A while típusú ciklusutasítást akkor választjuk, ha a lépések számát nem ismerjük, vagy a ciklusváltozó valós típusú.

Például:

x:= 0.1;	
while x < 1.1 **do** x := x + 0.1;	itt x a 0.1 .. 1.0 intervallumon lépked 0.1 lépésenként
fract := 1;	

 © Aradi, Bécsi, Gyenes, Péter, BME

```
while fract > 1E-3 do fract:= fract / 2;
```
a ciklus leáll, ha fract ≤ 10^{-3} a lépésszám nem ismert.

5.3.7.3 Repeat ciklus

Általános ciklusutasítás, amelynél nincs ciklusváltozó, az ismétlések számát feltétel határozza meg.

Szintaktika:

```
repeat
  utasítás_1;
  utasítás_2;
  utasítás_n;
until logikai kifejezés ;
```

Az utasítássorozat legalább egyszer végre lesz hajtva, majd mindaddig ismételten, míg a logikai kifejezés **false** értékű nem lesz. A számítógép a logikai kifejezést a ciklusmag végrehajtása után ellenőrzi.

Például:

```
x:=5;
repeat
  x:=3 * x;
until x>50;
```
ciklus végrehajtása után x = 135 lesz.

Érdekességképpen megjegyezzük, hogy a ciklus szerkezeteket goto utasítással is megírhattuk volna (Basic és assembler programoknál szokásos módon).

```
      x :=5;
loop: x:=3 * x;
      if x <= 50 then goto loop;
```

Gyakorló feladatok

F01. Adott k db valós érték. Írjunk alkalmas ciklus utasítást, amely megkeresi ezek közül a legnagyobbat.

F02. Hányszor lesz végrehajtva az alábbi ciklus, és mik a változók által felvett értékek a végrehajtás folyamán?

```
i := 1; j := 3;
while j > 0 do
  begin
    j := (j + 10) div (2 * i);
    inc (i);
  end;
```

F03. Hányszor lesz végrehajtva az alábbi ciklus, és mik a változók által felvett értékek a végrehajtás folyamán?

```
c :=’a’;
repeat
  if odd (ord(c)) then c := succ (succ (c))
                  else c := pred (c);
until c > chr (99);
```

Ellenőrző kérdések

K01. Melyek a különböző ciklus utasítások, és mikor melyiket használjuk?

K02. Mi a címke, és milyen szabály érvényes a használatára?

5.3.8 Case utasítás

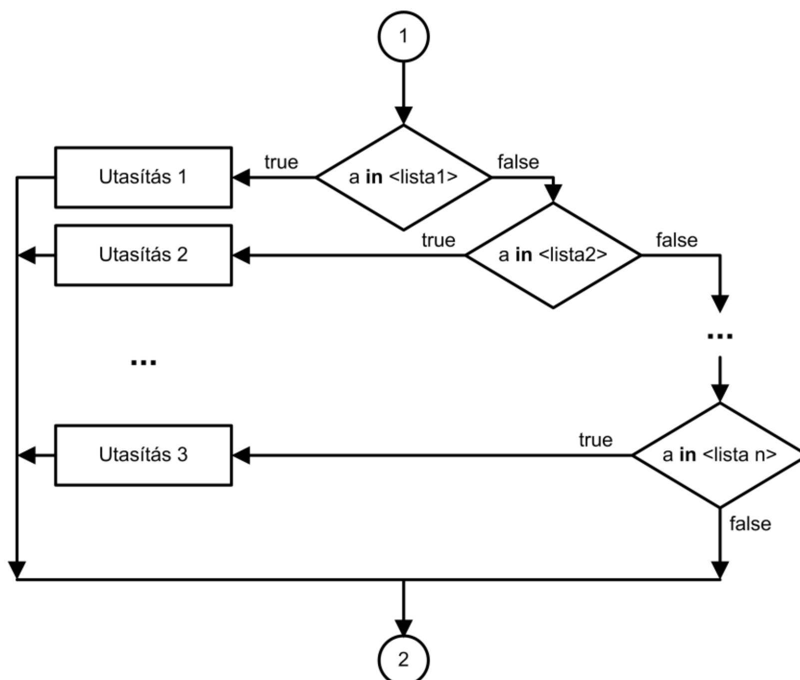
A sok egymásba ágyazott feltételes utasítás gyakran áttekinthetetlen program struktúrát eredményez. Ezért a Pascal a **case** utasítást (Program Switch) definiálja. Ennek is két változata létezik.

Szintaktika:

```
case azonosító of
        lista1 : utasítás1;
        lista2 : utasítás2;
        …
end;
```

A számítógép az *azonosító* lehetséges értékei (amik a *lista1*, *lista2*, .. felsorolásban jelennek meg) szerint a megfelelő utasítást (illetve utasítás csoportot) hajtja végre.

Az első **case** változat folyamatábra ekvivalense a következő.



21. ábra: A case utasítás folyamatábrája

Fontos szabályok!

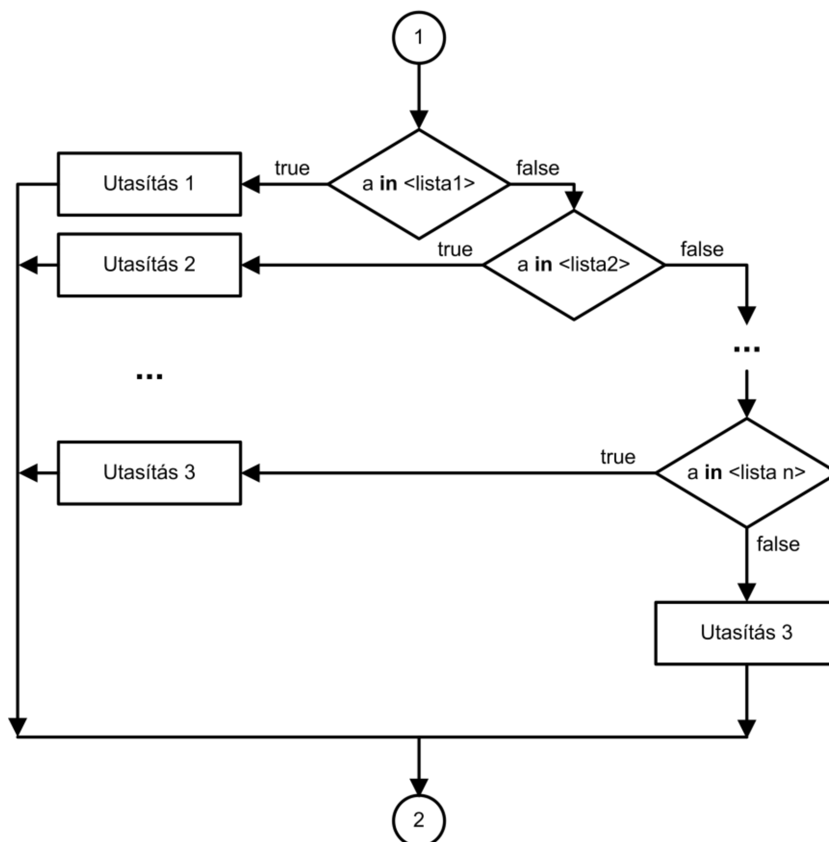
Szabály 1: Az azonosító csak ordinal (megszámlálható) típusú lehet.

Szabály 2: A lista egyetlen érték, felsorolás, vagy intervallum lehet.

A második, vagy teljes változat szintaktikája:

case *azonosító* **of**
  lista1 : *utasítás1*;
  lista2 : *utasítás2*;
  …
else
  utasítás;
end;

A második **case** változat folyamatábra ekvivalense a következő.



22. ábra: A teljes case utasítás folyamatábrája

Álljon itt egy példa e második változatra.

```
case x of
  3          : y:=4 * x;            // lista egy érték
  7..9       : y:=abs(sqr(x));      // lista intervallum
  2,0,11     : y:=sqrt(5 * x);      // felsorolás
  1,12..15   : y:=0;                // vegyes lista
else   y := arctan(x+3);
end;
```

Gyakorló feladatok

F01. Hajtsuk végre az alábbi programrészletet, és készítsünk listát a változó felvett értékeiről.

```
x := chr ( 66 );
repeat
```

```
    case x of
      'B', 'C', 'E' : x := chr ( ord ( x ) + 2 );
      'A', 'F'..'H' : x := pred ( x );
      else pred (x);
    end;
  until x > 'D';
```

F02. Adjuk meg a változók értékeit minden lépésben:

```
a : = -3;
for b := 1 to 5 do
  case a of
    -3 . . 0 : a := a + b;
           3 : a := a - b;
  end;
```

Ellenőrző kérdések

K01. Mikor célszerű a case utasítás használata?

K02. Mik lehetnek a lista elemei?

5.3.9 Zárójelek használata

A Pascal négyféle zárójel párt használ.

a) Kifejezéseknél a kerek zárójel használatos: pl. 3* (x+2*y). A kerek zárójeles kifejezés elsődlegessé válik.

b) Index jelölésére a szögletes zárójel párt [(kódja 91)] (kódja 93) használjuk. A PC billentyűn ASCII kódjával bármely karaktert kiválaszthatjuk, ha az ALT billentyű nyomva tartásával a NUM-PAD (oldalsó szám mezőn) begépeljük a kódot. Pl. x [i + 2] , matrix [3,5].

A szögletes zárójelpárt a halmaz típus jelölésére is használjuk (ld. 5.5. pont).

c) Kommentár jelölésére a kapcsos zárójel párt { (kódja 123) } (kódja 125) használjuk. Pl.: i:= i +1; { i változó növelése }

d) Utasítás zárójel a **begin-end** pár, amely az utasításokat köti egy csoportba. Erre nagyon gyakran szükség van a programozási gyakorlatban. A Pascal az utasítás zárójeles csoportot összetett utasításnak (Compound statements) nevezi.

Álljon itt egy példa az utasítás zárójel használatára.

```
i := 2;  s := 0;
while i < 5 do
  begin                    // az alábbi 2 utasítás minden lépésben végrehajtódik
   s := s + 3 * i ;
   i: = i + 2 ;
  end;
```

5.3.10 Rekord adattípus használata

A számítástechnika egyik fontos feladata a valóságos fizikai folyamatok modellezése. A skalár típusú adatok csak egy jellemzőt (pl. hőmérséklet, nyomás, feszültség, stb.) tudnak megjeleníteni. A valóságos jelenségeket azonban több, különböző típusú adat jellemzi. Egy mozgó tárgynak van tömege, pozíciója, sebessége, gyorsulása. Ha tehát pl. egy gépkocsit akarunk modellezni, akkor ezeket a mennyiségeket lehetőleg egy adattípusba kell integrálni, hiszen ugyanarra a mozgó tárgyra vonatkoznak.

A Pascal (és a többi magas szintű nyelvek is) erre a feladatra kínálja a **record** adattípust.

Definíció: A **record** adattípus közös típusnév alatt kezelt különböző típusú adatok összessége.

Ezt a típust a programozó típusdefinícióval adja meg (ld. 5.3.6. pont).

Szintaktika:

```
type azonosító = record
                   mező1 : títpus1;
                   mező2 : títpus2;
                 end;
```

Egy hallgatói nyilvántartás számára az alábbi rekordot definiálhatjuk.

```
type t_student = record
                   name :  string[40];        // név
                   birth  :  integer;         // születési dátum
                   sex    :  boolean;         // nem
                   addr  :  string[30];       // lakcím
                   mark  : single;            // tanulmányi átlag
                 end;
```

A fenti rekord típus helyfoglalását a memóriában az alábbi táblázat mutatja.

	Mező	Méret
t_student	**name**	41 byte
	birth	4 byte
	sex	1 byte
	addr	31 byte
	mark	4 byte

25. táblázat: A t_student rekord helyfoglalása

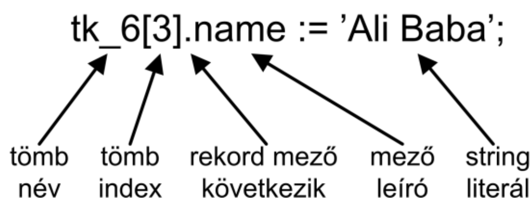
Egy hallgató rekordja 81 byte helyet foglal. A rekord típusdefiniálása után deklaráljuk a szükséges változókat:

```
var hallg : t_student;                  // egy hallgató
    tk_6 : array[1..40] of t_student;   // egy tanulókör (pl. 6. tanulókör)
```

A fenti deklaráció következtében a *hallg* változó 81, míg a *tk_6* változó 40*81=3240 byte memóriaterületet foglal el.

Nézzük miként adunk értéket a rekord típusú változó egyes mezőinek.

Legyen a 6. tanulókör 3. hallgatójának neve Ali Baba, és a tizedik hallgató átlaga 4.25.



Az átlag megadása:

```
tk_6[10].mark := 4.25;
```

A string típusú adatot (*name* mező) karakterenként is kezelhetjük. Kiolvashatjuk a 3. hallgató nevének 5. karakterét egy *letter* nevű, karakter típusú változóba:

```
letter := tk_6 [3] . name [5];          //a letter változó értéke ' B' lesz.
```

Előfordulhat, hogy egy rekord struktúra egyes elemeinek a közös vonásokon kívül specifikus jellemzői is vannak. Ekkor alkalmazzuk a változó rekord szer-

kezetet. A fenti példánk szerint különböztessük meg a fővárosi és vidéki hallgatókat (pl. a szociális támogatás miatt). A vidéki hallgatóknak legyen a kollégiumi elhelyezés költsége jellemző, míg a fővárosiaknak az egy főre jutó jövedelem.

Ekkor a rekord definició az alábbi lehet:

```
type t_student = record
                    name :  string[40];                // név
                    birth  :  integer;                 // születési dátum
                    case citizen: boolean of
                      true : college_price : integer;  // vidéki
                      false: huf_per_person : single;  // fővárosi
                 end;
```

A program futásakor ha a *citizen* mező aktuális értéke **true**, akkor a *college_price* mező, míg **false** értéke esetén a *huf_per_person* mező releváns.

Nézzünk egy másik példát, amely síkidomok jellemző adatait modellezi.

Legyen egy felsorolásos (enumerated) típus az alábbi.

```
type t_sikidom = (rectangle, triangle, circle, ellipse);
```

Ehhez az alábbi változó rekordot rendelhetjük.

```
type t_adatok = record
                   case fajta : t_sikidom of
                     rectangle: (height, width: single); // téglalapnál mag., szélesség
                     triangle   : (side1, side2, angle: single);
                                      // háromszögnél 2 oldal    és a közbezárt szög
                     circle      : (radius: single);          // Körnél a sugár
                     ellipse     : (axis1, axis2 : single);   // Ellipszisnél tengelyek
                 end;
```

A megfelelő változó deklaráció az alábbi lehet.

```
var idom : t_adatok;
```

A megfelelő értékadás pedig a következő.

```
if idom.fajta = circle then idom.radius := 12.34;
```

Gyakorló feladatok

F01. Tervezzünk rekord típust különböző járművek paramétereink tárolására. A járművek fajtái lehetnek: Hajó, repülő, vonat autó. Mindegyik típushoz más jellemzők tartoznak. Ezeket változó rekord szerkezetben jelenítsük meg.

F02. Adott az alábbi rekord szerkezet. Adjunk értéket a *staff* változó összes mezőjének.

```
type t_color = (red, brown, black);
     t_worker = record
                   name, fam_name : string[20];
                   case sex : (man, woman) of
                     man : weight, age : integer;
                     woman : hair : t_color;
                  end;
var staff : array[1..2] of t_worker;
```

Ellenőrző kérdések

K01. Mikor használjuk a rekord adattípust?

K02. Hogy nevezzük a rekord típus elemeit?

K03. Milyen szimbólummal hivatkozunk egy rekord típusú változó valamelyik elemére?

5.3.11 With utasítás

Az összetett adattípusok kényelmesebb kezelésére szolgál a **with** utasítás.

Használatát az alábbi példával szemléltetjük. Adjuk meg a 6. tanulókör 12. tagjának adatait!

```
tk_6[12].name := 'Kis Péter' ;
tk_6[12].birth  := 19870513 ;
tk_6[12].sex    := true ;
tk_6[12].addr  := 'Kör utca 4.' ;
tk_6[12].mark := 3.78 ;
```

Az öt értékadásban a tk_6 [12] változatlan, ezért kiemelhető az alábbi módon.

```
with tk_6 [12] do begin
                name := 'Kis Péter';
                birth  := 19870513;
                sex    := true;
                addr  := 'Kör utca 4.';
                mark := 3.78;
              end;
```

Láthatjuk, hogy a **with** belsejében a mezőleírók változóként vannak felhasználva.

5.3.12 Input-Output utasítások

Pascal környezetben a számítógép alapértelmezett (default) bemeneti eszköze a billentyűzet (keyboard), míg kimeneti eszköze a monitor képernyő (display).

Már legelső egyszerű programjainknak szüksége lehet bemenő adatokra. Ezeket a billentyűzetről tudjuk bevinni programunkba a

```
readln (x );
```

input utasítással. Az x változónév, amely a begépelt értéket (szám, betű, string) tárolni fogja.

Legyen egy *nr* nevű változó deklarálva!

```
var nr : integer;
```

Ha ennek az adott futási helyzetben 1234 értéket kívánunk adni, akkor a következő utasítást adjuk meg.

```
readln (nr);                                // Billentyűn begépelni: 1234 + ENTER
```

Hasonlóképpen ha a képernyőn szeretnénk valamilyen eredményt kiíratni, akkor a

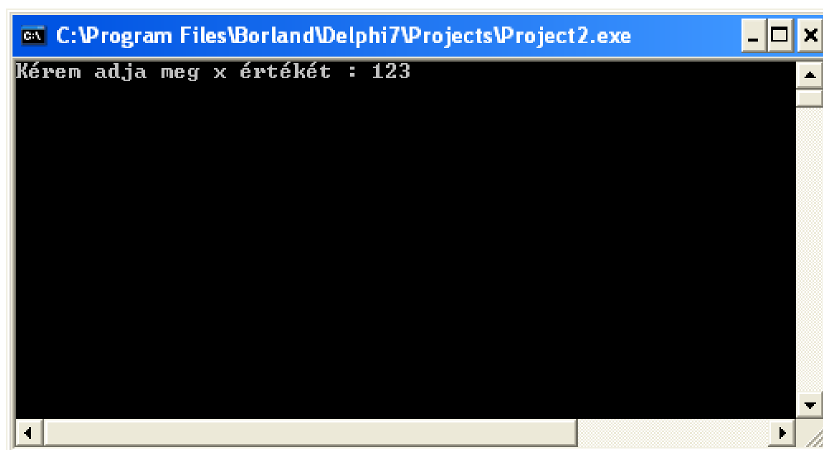
```
writeln( y );
```

utasítás kiadásával tehetjük meg.

A beolvasást gyakran megelőzi egy kiíratás, amellyel a program használóját tájékoztatjuk a beolvasandó mennyiségről.

```
write (' Kérem adja meg x értékét : ');
readln(x);
```

utasítás pár esetén a képernyőn az alábbiakat látjuk.



23. ábra: Példa az adatbeolvasásra

A **writeln** használatakor megadhatjuk a kiírás formátumát is az alábbi módon.

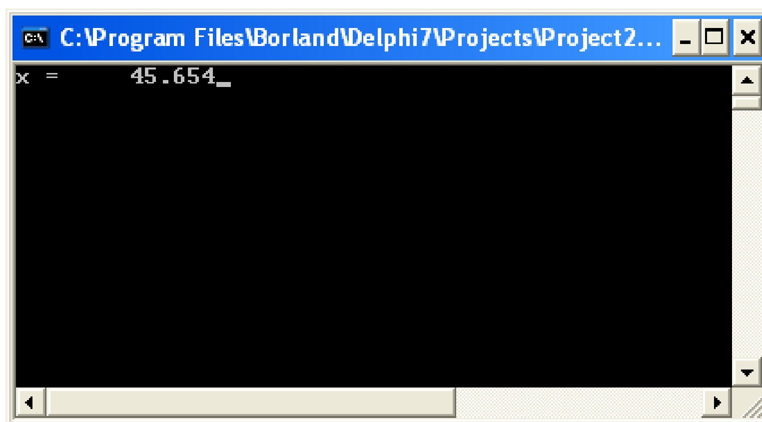
writeln (x : k : t);

ahol x a változó azonosítója, k a felhasznált pozíciók száma és t a tizedes jegyek száma.

Például, ha az x aktuális értéke 45.654321, akkor a

writeln (’x = ’, x : 10 : 3);

utasítás hatására az alábbi kijelzést látjuk (10 helyre 3 tizedes jeggyel).



24. ábra: Példa a formázott kiíratásra

Gyakorló feladatok

F01. Adjuk meg a 2 * pi / 3 kifejezés 4 tizedes pontos kiíratására való utasítást.

F02. Miként olvassuk be egy háromszög oldalait?

Ellenőrző kérdések

K01. Hogyan emelünk sort a képernyőn?

5.4 Program szegmentáció

A strukturált programozás eszköze a program újra felhasználható részekre (szegmensekre vagy más néven modulokra) való bontása. Ezt a lehetőséget a Pascal 4. szintje biztosítja. A szegmens fajtái:

- függvény (**function**)
- eljárás (**procedure**)
- egység (**unit**).

Az első kettő belső (az aktuális program fejében deklarált), a harmadik külső (könyvtári) szegmens.

A szegmensek használatához ismernünk kell a Pascal program szerkezeti felépítését.

A Pascal program 2 fő részből áll: program fej és törzs. A programfejben helyezzük el a már ismert nem végrehajtható utasításokat: definíciók, deklarációk. Itt kell deklarálni a program szegmenseket is.

A program törzsébe a tanult végrehajtható utasításokat, valamint a szegmensek aktivizálását írjuk.

A program törzs elejét és végét a **begin end.** pár jelzi. A Pascal program szerkezete az alábbi ábrán látható.

 © Aradi, Bécsi, Gyenes, Péter, BME

Program fej

program név;
Globális típus definíciók (**type**)
Globális konstans definíciók (**const**)
Globális változó deklarációk (**var**)
Eljárás deklarációk (**procedure**)
Függvény deklarációk (**function**)

begin

Program törzs

utasítások
függvény hívások
eljárás hívások

end.

25. ábra: A Pascal program szerkezete

5.4.1 Függvények

A függvények valamely típusú érték kiszámítását végzik. A Pascal kétféle függvényt használ:

Standard (ezeket az építőelemek között tárgyaltuk) és a programozó által definiált függvények. Ebben a fejezetben ezekkel foglalkozunk.

A Pascal függvény szerkezete az ábrán látható:

Függvény fej

function fnam (formális paraméterek) : típus ;
Lokális típus definíciók (**type**)
Lokális konstans definíciók (**const**)
Lokális változó deklarációk (**var**)
Lokális eljárás deklarációk (**procedure**)
Lokális függvény deklarációk (**function**)

begin
utasítások

Függvény törzs

fnam := kifejezés; **a fv. neve értéket kap !**

end;

26. ábra: A Pascal függvény szerkezete

A programstruktúrához hasonlóan a függvény is két részből áll. A fej tartalmazza a nem végrehajtható utasításokat: definíciók, deklarációk. Ezek lokálisak, azaz az itt bemutatott elemek csak a függvény belsejében érvényesek.

A függvény elején a fejsor áll. Ez tartalmazza a függvény nevét, a formális paraméter listát (ez opcionális), majd a függvény nevének típusmegjelölését. A formális paraméter lista a főprogrammal való adattranszfer eszköze. A bemenő paraméterek adatot vesznek át a programtól.

Példa a bemenő paraméterekre.

function szoroz (x , y : **single**) : **single**;

function check (c : **char** ; v : **integer**) : **boolean** ;

Az azonos típusú paramétereket vessző, míg különböző típusú paramétereket pontosvessző szimbólum választja el egymástól. A bemenő paramétereket a függvény érték szerint veszi át a programtól.

A kimenő paraméterek adatot adnak át a programnak, de egyben bemenő paraméterként is szolgálnak.

Példa a be/kimenő paraméterekre.

function modi (**var** z : **byte** ; **var** a : t_**tomb**) : **char**;

Látjuk, hogy az adatáramlás irányát a **var** kulcsszó mutatja.

A paraméter lista mindkét típust tartalmazhatja.

function does (m,c : **char** ; **var** r : **real**) : **boolean** ;

A példában *m* és *c* bemenő-, míg *r* kimenő paraméter.

A be/kimenő paramétereket a függvény cím szerint veszi át a programtól.

A paraméter lista után a fejsorban adott típusmegadás függvény nevére vonatkozik. A ***függvény neve*** a megadott típusnak megfelelő ***értéket hordoz.*** Ezért kell a függvény törzsében ezt az értéket értékadó utasítással megadni.

A függvény törzse **begin end;** között elhelyezett végrehajtható utasítások sorozata. Ezek között szerepel az is, amelyik a függvény nevének értéket ad.

A változók (típusok, konstansok, szegmensek) hatásköre:

a) A programfejben deklarált változók (típusok, konstansok, szegmensek) az egész programban érvényesek, még a szegmensek (függvények és eljárások) belsejében is. Ezért ezeket globális elemeknek (változók, típusok, konstansok,

 © Aradi, Bécsi, Gyenes, Péter, BME

szegmensek) nevezzük. Ezzel szemben a függvény fejben bemutatott mennyiségek csak a függvény belsejében, ezért ezeket lokális elemeknek nevezzük.

b) Előfordul, hogy valamely globális és lokális elem azonos névvel van jelölve. Ekkor a lokális elem érvényes, az azonos nevű globális nem érhető el a szegmens belsejében.

Függvény hívása a programban: a programfejben deklarált függvény kifejezésként használható, a programtörzsben ***önállóan ne használjuk !***

Példa a helyes függvényhívásra.

```
azonosító := fv_nam (aktuális paraméterek);
```

Az ***aktuális paraméterlista*** a függvény hívásakor a paraméterként használt adatok (literálok, változók) tényleges értékét, azonosítóit tartalmazza.

Fontos szabály, miszerint a formális és aktuális paraméterlista elemeinek darabszámban, típusban és sorrendben meg kell egyezni.

A 6. fejezetben számos példát találunk a függvények alkalmazására. A továbbiakban a paraméterátadás módjaival foglalkozunk.

Példa az érték szerinti paraméter átadásra

Legyen a programban az alábbi deklaráció.

```
var x : integer;
```

Ennek hatására a fordító lefoglal egy rekeszt x néven, melynek tartalma ismeretlen.

A programunk fejrészében az alábbi függvény található:

```
function szor( x :integer): integer;        // x formális paraméter
 begin
   x := 3 * x ;                             // nem a globális x !!!
   szor := x ;
 end;
```

A programunk törzsében az alábbi utasítások vannak:

```
begin
  x := 6;                // globális x rekesz:
  write (szor(x) );      // a fv átveszi a 6-ot, kiírja : 18 , x nem változik !
end.
```

© Aradi, Bécsi, Gyenes, Péter, BME

Példa a cím szerinti paraméter átadásra

Legyen a programban az alábbi deklaráció.

```
var x : integer;                // x = ?
```

A programunk fejrészében az alábbi függvény található.

```
function szor(var y:integer): integer;  // y formális ki/be paraméter
 begin
   y := 3 * y;
   szor := y;
 end;
```

A programunk törzsében az alábbi utasítások vannak:

```
begin
  x := 6;                     // globális     x : 6
  write (szor(x));            // kiírja : 18   x : 18
end.
```

Látható, hogy érték szerinti paraméterátadáskor a globális változó értéke nem változott, míg a cím szerinti paraméterátadáskor a globális változó felvette a függvényben kapott értéket.

Példa a globális és lokális változók hatáskörére.

```
program legitim;
  var x : byte;                      x: ?

  function add (y:byte): byte;
   var x : byte;                     x: ?
   begin
    x := 2 * y;                      // itt a kék x érvényes !
    add := y + x;                    // a piros x nem !
  end;

  begin                               // program törzs
   x:= 10;                            // piros x érvényes     x: 10
   write (add (x) );                  // kiírja : 30          x: 20
   write(x);                          // kiírja : 10
  end.
```

Gyakorlásul módosítsuk a függvény formális paraméterét be/kimenő típusura, és vizsgáljuk meg a program működését.

© Aradi, Bécsi, Gyenes, Péter, BME

Gyakorló feladatok

F01. Készítsünk függvényt k! (k faktoriálisának kiszámítására : k! = 1.2.3…k).

F02. Készítsünk boolean függvényt, amely egy pozitív egész számról eldönti, hogy az prímszám-e.

Ellenőrző kérdések

K01. Mi a programszegmens meghatározása?

K02. Rajzolja le a függvény szerkezetét!

K03. Mi az érték- és cím szerinti paraméterátadás?

K04. Mi az aktuális paraméter lista?

5.4.2 Függvény típusú változók

Némely alkalmazásban szükség lehet arra, hogy a szegmensnek átadott paraméter ne csak valamilyen típusú adat, hanem egy függvény legyen. Ily módon lehetővé válik, hogy ugyanaz a szegmens más és más paraméterezéssel más műveletsort végezzen el.

Definiáljunk például egy kétparaméteres függvény típust (x és y bemenő paraméterek).

```
type t_func = function (x,y : byte) : integer;
```

Ezután deklarálhatunk egy ilyen függvény (t_func) típusú változót.

```
var func_var : t_func;
```

Deklaráljunk egy függvényt (*operation*), amelynek formális paraméterei közé felvesszük a függvény típusú változót (természetesen csak formális néven: példánkban fv).

```
function operation (fv : t_func; a,b:byte) : integer;
 begin
  operation:= fv(a,b);            // Az eredmény a paraméterben átadott függvény-
                                  től függ
 end;
```

Még deklaráljunk két külön kétparaméteres függvényt is:

```
function add (c,d : byte): integer;
  begin
   add := c + d;                          // A két paraméter összege
  end;

function sub (u,v : byte): byte;
  begin
   sub:= u - v;                           // A két paraméter különbsége
  end;
```

A fenti deklarációk után főprogramunkban aktivizáljuk az *operation* függvényt:

```
begin
   writeln ( operation (add,5,8) );     // Aktuális paraméter add fv. neve; Kiírás : 13
   writeln ( operation (sub,10,4) );    // Aktuális paraméter sub fv. neve; Kiírás : 6

   func_var := add;                     // Közvetlen értékadás : a függvény típusú
                                        // változó felveszi az add függvényt.

    writeln (operation (func_var,13,4) );   // Aktuális paraméter func_var változó;
                                            // Kiírás : 17

    func_var := sub;                        // Közvetlen értékadás : a függvény típusú // vál-
                                            tozó felveszi a sub függvényt.
   writeln (operation (func_var,12,5) );    // Aktuális paraméter func_var változó;
                                            // Kiírás : 7
end.
```

5.4.3 Eljárások

Az eljárás (**procedure**) utasítások egy sorozatát fogja össze egyetlen újra felhasználható szegmensbe. Pascal kétféle eljárást használ.

Standard eljárások, amelyeket a nyelv előre definiálta, és használatra ajánlja a programozónak. Néhány fontosabb standard eljárás:

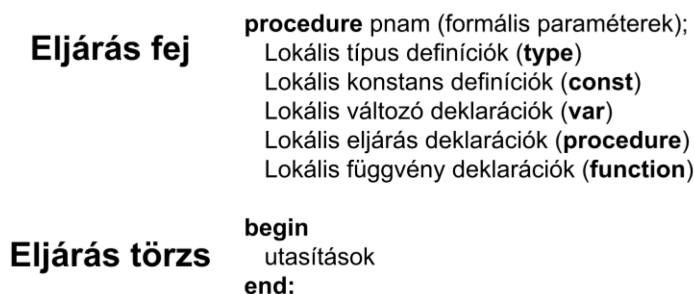
Append	Assign	BlockRead	BlockWrite	ChDir
Close	Dec	Delete	Dispose	Erase
Exec	Exit	FreeMem	GetDir	GetTime
Halt	Inc	Insert	MkDir	New
Randomize	Read	ReadLn	Rename	Reset

 © Aradi, Bécsi, Gyenes, Péter, BME

Rewrite	Seek	Str	Truncate	Val
Write	WriteLn			

E fejezetben a programozó által definiált eljárásokkal foglalkozunk.

A Pascal eljárás szerkezete az ábrán látható.



27. ábra: A Pascal eljárás szerkezete

A program és függvény struktúrához hasonlóan az eljárás is két részből áll. A fej tartalmazza a nem végrehajtható utasításokat, amelyek definíciók, deklarációk. Ezek lokálisak, azaz az itt bemutatott elemek csak az eljárás belsejében érvényesek.

Az eljárás elején a fejsor áll. Ez tartalmazza az eljárás nevét, a formális paraméter listát (ez opcionális), majd pontosvesszővel zárul.

A formális paraméter lista a főprogrammal való adat-transzfer eszköze. A paraméterek bemenő és kimenő paraméterek lehetnek (ld. a függvényeknél elmondottakat).

Az eljárás nevének ***nincs típusa***, értéket nem hordoz.

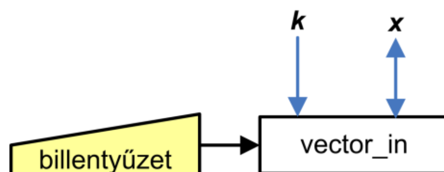
Az eljárás a programban csak önmagában állhat, szintaktikailag utasításként hívható meg.

Az eljárás törzsében az az utasításlista írandó, amelyet az eljárás aktivizálásakor végre akarunk hajtani.

Eljárás formában a többször felhasználni kívánt program részeket célszerű elkészíteni.

Nézzünk egy példát az eljárás deklarációra.

Gyakran szükséges adatok beolvasása a billentyűzetről, és azok elhelyezése egy alkalmas tömb típusú változóban. Tervezzük meg ezt a feladatot, megvalósító eljárást. Az adatok számát a *k* formális bemenő paraméter helyén veszi át a *vector_in* nevű eljárás, majd az adatokat, tároló vektort az *x* kimenő formális paraméter helyén adja át a programnak.



28. ábra: A beolvasó eljárás terve

Szükség van az adatokat tároló vektor típusának definiálására:

```
type t_vector = array [1..20] of single;
```

A tömb méretét meghatározó 20 szám becsült érték (az adott feladat függvényében változhat) . Ennyi single típusú rekeszt foglal el egy ilyen vektor. A fejsort ezután az alábbi formában írhatjuk meg:

```
procedure vector_in (k : byte ; var x : t_vector);
```

A beolvasás ciklusutasítással történik amihez szükség van egy ciklus változóra. Fontos szabályként jegyezzük meg, hogy egy szegmensben ***a ciklusváltozó kötelezően lokális kell legyen***.

A beolvasás 2 utasítás ismételgetéséből áll: kiíratunk (**write** utasítás) egy üzenetet (mit vár a program) majd beolvasunk (**read** utasítás) egy értéket. Ezek előrebocsátása után beolvasó eljárásunk utasításlistája az alábbi lesz.

```
procedure vector_in (k : byte ; var x : t_vector);
 var i : byte;                         // lokális ciklus változó
 begin                                 // eljárás törzs kezdete
  for i := 1 to k do
   begin                               // nyitó utasítás zárójel
    write ( 'x[' , i, ']= ');          // kérjük a tömb i.-ik elemét
    readln ( x[ i ] );                 // a tömb i.-ik elemének beolvasása
   end;                                // csukó utasítás zárójel
  end;                                 // eljárás törzs vége
```

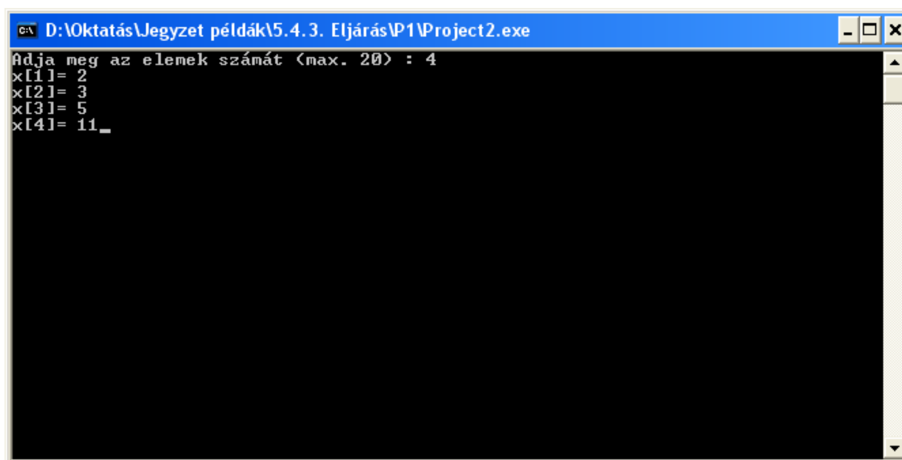
Mintaprogramjainkban a Pascal foglalt kulcsszavait vastagon szedve írjuk.

Fő programunk utasításlistája az alábbi lesz.

```
// Vektor beolvasó mintaprogram. Írta Ali Baba, 2005.11.03.

program vektor_be;

type t_vector = array [1..20] of single;     // Típus definíció

procedure vector_in (k : byte ; var x : t_vector);// Eljárás deklaráció

    --- A fenti eljárás bemásolása ---
  end;

var adatok : t_vector;                        // adatok nevű tömb deklarációja
          n  : byte;                          // elemek számának változója

begin
  write ('Adja meg az elemek számát (max. 20) : ');
  readln (n);                                 // elemek aktuális számának beolvasása
  vector_in ( n, adatok );                    // eljárás hívása n, és adatok aktuális
                                              // paraméterekkel
end.
```

Ez a program a beolvasáson kívül mást nem csinál, értelme az eljárás készítésének bemutatása.

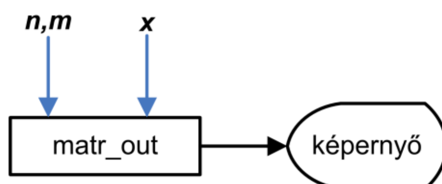
A program futási képe a következő.



29. ábra: A beolvasó program futási képe

Hasonlóképpen hasznos lehet eljárást készíteni a tömbök képernyőre való kiíratására is. A paraméterezéshez először is szükséges a tömb típusának definiálása. Ez például az alábbi lehet:

```
type t_matrix = array[1..10,1..10] of single;
```



30. ábra: A kiíró eljárás terve

Formális paraméterlistában meg kell adni a sor és oszlopszámot (n és m) és a mátrixot szimbolizáló szimbolikus nevet (x). Ezek mindegyike bemenő paraméter.

Ezután már megírhatjuk az eljárást, amelynek neve *matr_out* lehet.

```
procedure matr_out (n,m : byte ; x : t_matrix);
 var    i,j : byte;                        // lokális ciklusváltozók
 begin                                     // eljárás törzse
  for i :=1 to n do
   begin
    writeln;                               // új sor kezdés
    for j:=1 to m do  write (x[ i , j ]:8:2); // elemek kiíratása
   end;                                    // egymás mellé
 end;                                      // eljárás törzs vége
```

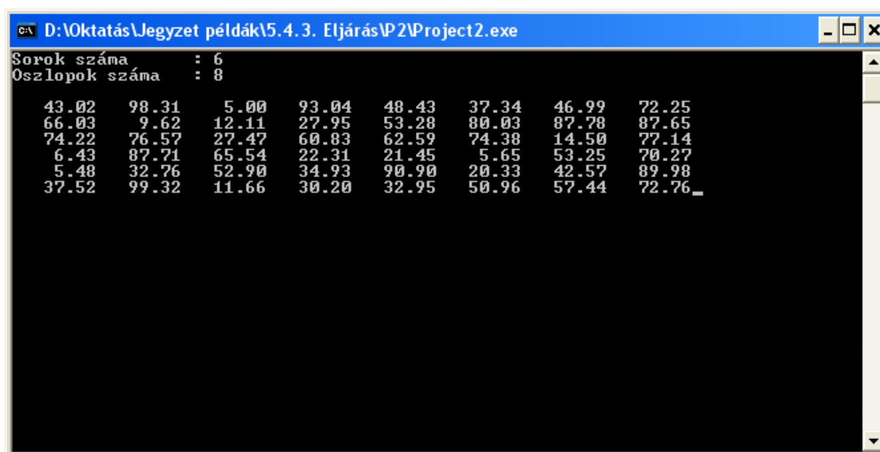
Az eljárás aktivizálása kipróbálás céljára az alábbi program listával történhet.

```
// Mátrix kiíratása próbaprogram
// Nagy Benő 2010.09.05.

program matrix_display;

type t_matrix = array[1..10,1..10] of single;

procedure matr_out (n,m : byte ; x : t_matrix);
 begin < az eljárás bemásolása > end;

var               a  : t_matrix ;
       i,j,sor,oszl  : byte;
begin                                      // program törzs eleje
 write('Sorok száma     : '); readln(sor);
 write('Oszlopok száma : '); readln(oszl);
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
    randomize;                 // véletlen szám generátor inicializálása
    for i := 1 to sor do       // sorok
     for j := 1 to oszl do     // oszlopok
       a[i,j]:=100*random;     // valós típusú elemek generálása 0 és 100 // kö-
                               zött
    matr_out(sor,oszl,a);      // Az eljárás hívása az aktuális
                               // paraméterekkel
   end;
```

A program futási képe az alábbi ábrán látható.



31. ábra: A mátrix kiíró program képernyőképe

Gyakorló feladatok

F01. Deklaráljunk *rendez* néven eljárást, amely paraméterben átvesz egy valós típusú vektort annak méretével együtt, és a vektor elemeit csökkenő sorrendbe rendezve adja vissza.

F02. Deklaráljunk *search* nevű eljárást, amely paraméterben átvesz egy szöveget (max 255 karakter) és egy szót. Megkeresi a szövegben a szó előfordulásainak számát, és e számot paraméterben visszaadja.

Ellenőrző kérdések

K01. Adja meg az eljárás szerkezeti felépítését.

K02. Mik a legfontosabb különbségek az eljárás és függvény között ?

K03. Mikor alkalmazzuk az eljárást ?

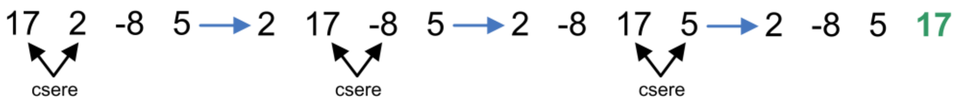
5.4.4 Lokális szegmens használata.

A programozási gyakorlatban előfordul, hogy egy szegmens belsejében saját (lokális) eljárás, vagy függvény használata célszerű. Az alábbiakban erre mutatunk példát.

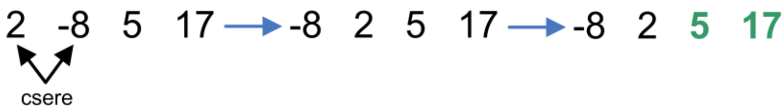
A program beolvas neveket, majd azokat ABC sorrendbe rendezi, és kiírja. A neveket egy **string** típusú tömbbe fogjuk beolvasni és tárolni.

A rendezéshez a buborék algoritmust használjuk. Ezen módszer szerint a szomszédos elemeket kell csak figyelni, és szükség esetén felcserélni.

A buborék algoritmus működését az alábbi 4 szám rendezésével szemléltetjük.



A számok még nincsenek sorban, de a legnagyobb szám biztosan a helyén (leghátul) van.



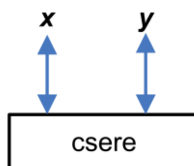
Most már a 2. legnagyobb szám is a helyén van. Újra elölről kezdjük a vizsgálatot, de már az utolsó 2 elemet nem vizsgáljuk.

A sorbarendezés befejeződött, a számok sorban vannak.

Feladatunkban ugyanezt tesszük a nevek betűivel. A nevek betűinek rendezését hátulról kezdjük a legutolsó betűvel.

A csere eljárás terve a következő ábrán látható.

 © Aradi, Bécsi, Gyenes, Péter, BME



32. ábra: A „csere” eljárás terve

Az eljárás kicseréli az x és y formális paraméterek helyén átvett mennyiségéket és ugyanazon helyen visszaadja.

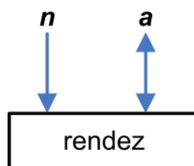
A feladat megoldásához szükséges egy alkalmas típus definiálása, amely egy nevet tárol. Erre az alábbi utasítás megfelel.

```
type t_name  = string [30];              // név típus, max. 30 karakter
```

Ezzel megírhatjuk eljárásunkat.

```
procedure csere( var x,y : t_name );     // x és y cím szerint átadott paraméterek
  var z : t_name;                        // lokális segédváltozó a cseréhez
  begin
    z := x;  x := y;  y := z;            // a két mennyiség cseréje
  end;
```

Most megírhatjuk a rendező eljárást, melynek terve a következő ábrán látható.



33. ábra: A „rendez” eljárás terve

Az első formális paraméter (n) a nevek száma. Ezt érték szerint veszi át az eljárás (bemenő paraméter).

A második formális paraméter (a) a nevek tömbje, amely bemenetkor rendezetlen, kimenetkor ABC sorrendben lesznek. Ez tehát cím szerint átadott be/ki menő paraméter.

```
procedure  rendez (n: byte; var a : t_tomb);      // Az eljárás fejsora
  var i, j, k : byte;                    // lokális ciklus változók

 procedure  csere( var x,y : t_name);    // lokális eljárás
```

```
  var z : t_name;
   begin
    z := x;  x := y;  y := z;
   end;

begin                                   // A rendez eljárás törzse
  for i:=30 downto 1 do                 // Visszafelé lépked a név betűin
    for j:=n downto 2 do                // Egy vizsgálati ciklus
      for k:=1 to j-1 do                // mindig eggyel rövidebb
        if a[k,i]> a[k+1,i] then csere(a[k] , a[k+1] );// Lokális eljárás hívása, ha szükséges
 end;
```

A *csere* eljárás csak a rendezéshez szükséges, ezért helyeztük el a *rendez* eljárás fej részében.

Végül a teljes program utasításlistája.

```
// ABC sorba rendező program.
// Okos Kata 2010.03.15.

program nevek;
const nr        = 25;                   // Nevek maximális száma
type t_name  = string [30];             // Név típus
       t_tomb   = array [1..nr] of t_name;  // Nevek tömbjének típusa
var nevek     : t_tomb;                 // Nevek tömbje
    i,n          : byte;                // Globális változók

procedure beolvas(var n: byte; var a : t_tomb); // Neveket beolvasó eljárás
  var i    : byte;                      // lokális ciklus változó
  ch  : char;                           // karakter típusú lokális segédváltozó
  begin                                 // eljárás törzs
   for i:=1 to nr do a [ i ] := '';     // tömb törlése
    n := 0;                             // számláló törlése
    repeat
      n := n + 1;                       // számláló növelése
      write('Név : '); readln (a[ n ] );   // az n.-k név beolvasása
      writeln('Tovább ( I / N ) ? : ');
      readln(ch);                       // van még beolvasandó név ?
until upcase(ch) = 'N';                 // válasz karakter konvertálása nagybetűssé
  end;                                  // beolvasó eljárás vége

procedure rendez (n: byte; var a : t_tomb);        // A rendez eljárás bemásolása
   begin  … end;

begin                                   // főprogram eleje
  beolvas(n,nevek);                     // beolvasó eljárás hívása
  rendez (n,nevek);                     // rendező eljárás hívása
  writeln('A nevek ABC sorrendben : ');
  for i:=1 to n do writeln(nevek[ i ]);    // rendezett névsor kiíratása
```

```
    readln;                          // képernyő váltás megakadályozása
  end.
```

A 6. fejezetben még sok példát találunk az eljárások készítésére.

5.5 Halmazok

A halmaz maximálisan 256 elemű adat típus, amelyet megszámlálható, skalár elemek alkotnak. Számos feladatban jól használható eszköz ez a különleges adattípus. A halmaz típusú változó maximálisan 32 byte (256 bit) hosszú mezőben foglal helyet. A halmaz minden eleméhez egy bit tartozik. A bit értéke 1, ha az adott érték eleme a halmaz típusú változónak, illetve 0 ha nem.

A halmaz kezelését lépésenként mutatjuk be.

a) A halmaz ***definiálása*** mindig típus definícióval történik a **set** kulcsszó segítségével.

```
type t_karakter =  set of char;          // Halmaz típus definíció
     t_kisbetu  = 'a' .. 'z';            // Intervallum típus
     t_kis      = set of t_kisbetu;      // Kisbetű halmaz
     t_vitamin = (a, b1, b2, b3, b6, c, d, e);     // Felsorolásos típus
     t_vit      = set of t_vitamin;      // Halmaz típus
```

b) A halmaz típusú változók ***deklarálása***.

```
var  betuk  : t_karakter;           // Halmaz típusú változó
     kis1   : t_kisbetu;            // Intervallum típusú változó
     kicsik : t_kis;                // Halmaz típusú változó
     krumpli, citrom, zoldseg, cigi : t_vit;  // Halmaz típusú változók
```

c) A halmaz típusú változók értékének megadása.

```
kis1     := ' m ';                    // Nem standard (intervallum) skalár érték
kicsik   := [ 'c'..'f ' , 'm' , 'x'..'z' ];  // Kisbetűkből álló halmaz megadása
krumpli := [ a, b2..b6 ];             // Felsorolásos elemek halmazának megadása
citrom   := [c];                      // Egyetlen eleme van ennek a halmaznak
zoldseg := krumpli + citrom;          // Két halmaz összege
cigi     := [ ];                      // Üres halmaz
```

Néhány érdekességet megfigyelhetünk. A halmaz érték megadásához az index kifejezéseknél már megismert **[ ]** zárójel pár használatos. A halmazok esetében

a Pascal értelmezi a + , - és * műveleteket. Jelentésük ezúttal eltér a matematikában megszokottól.

d) Műveletek halmazokkal.

Az alábbiakban néhány halmazokkal végezhető logikai műveletet mutatunk be.

```
['k','m'] <> ['k'..'m']              // a kifejezés értéke true
['k','m'] < ['k'..'m']               // a kifejezés értéke true
['l'..'n'] < ['l','m','n']           // a kifejezés értéke false
[] < ['a']                           // a kifejezés értéke true
['g'] + ['a'..'e'] = ['a'..'g'] - ['f']    // a kifejezés értéke true
[ ] - ['m'] < []                     // a kifejezés értéke false
'k' in ['f'..'m','x']                // a kifejezés értéke true
```

Láthatjuk, hogy egy halmaz aktuális értéke megadható az elemek felsorolásával (vesszővel elválasztva), vagy a folyamatos elemsor esetén intervallumként is (a .. szimbólum segítségével).

A halmaz típusú változók jobb megértését segíti, ha ismerjük a memóriában való elhelyezésüket. Legyen egy intervallum típus az alábbi.

```
type t_naturals = 1 .. 5;            // Intervallum típus
```

Deklarálunk egy halmaz típusú változót.

```
var x : set of t_naturals;           // Halmaz típus 5 elemmel

x := [ ] ;          // üres halmaz
```

0	0	0	0	0
1	2	3	4	5

```
x := [ 2 .. 4 ] ;   // 3 elemű halmaz
```

0	1	1	1	0
1	2	3	4	5

Az *x* halmaz típusú változó 5 bitet használ. Ha a halmaz üres, akkor minden bitje 0 értékű. A második esetben a halmaz típusú változó felveszi a 2,3 és 4 értéket, a megfelelő 3 bit értéke tehát 1 lesz.

 © Aradi, Bécsi, Gyenes, Péter, BME

A halmaz típusú változó viselkedése eltér az eddig megismert skalár típusoktól. A halmaz típusú változó egyidejűleg több értéket (vagy egyet sem) felvehet, míg a skalár változónak mindig egy értéke van.

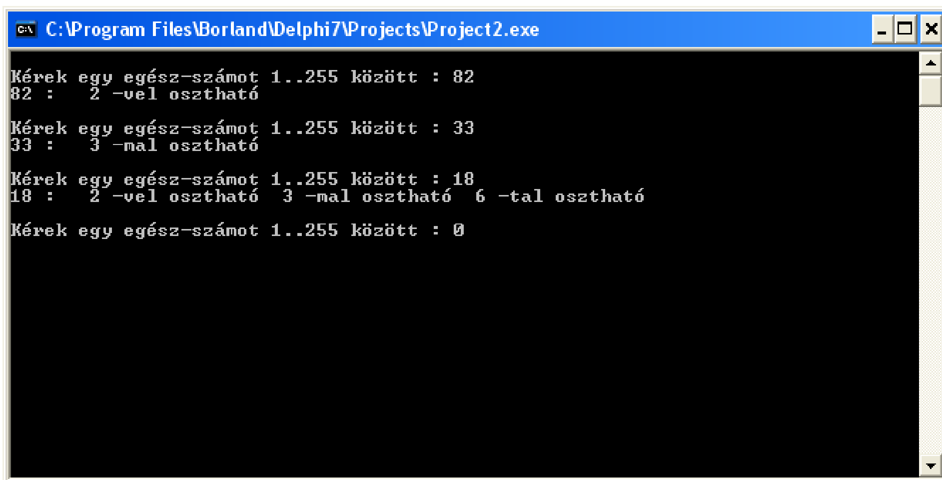
Nézzük a halmaz típusú változó használatát egy konkrét feladaton keresztül.

Eldöntendő 255-nél kisebb egészszámokról, hogy 2-vel, 3-mal és 6-tal osztha-tók-e. A feladat megoldása során képezzük a páros számok, és a 3-mal osztható számok halmazát. Ezen két halmaz közös elemei adják a 6-tal osztható számo-kat.

A programban vizsgáljuk, hogy a beolvasott szám melyik halmaz eleme.

```
// Halmaz típusú változók kezelése
// Zord Benő 2010.03.15.

program halmazok;

type  t_nr          = 1 .. 255;                // intervallum típus definíció
var    ket, har, hat : set of t_nr;            // halmaz típusú változó deklaráció
       x , i         : t_nr;                   // skalár változó deklaráció
       c             : char;                   // skalár segédváltozó
begin                                          // Főprogram törzse
  ket := [] ;  har := [];                      // Üres halmaz
  for i:= 1 to 127 do ket := ket + [2 * i];    // 2-vel osztható számok halmaza
  for i:= 1 to  85 do har :=  har + [3 * i];   // 3-al osztható számok halmaza
  hat:= ket * har;                             // 6-al osztható számok halmaza (konjunkció)
  repeat
   write('Kérek egy egész-számot 1..255 között : ');
   readln( x );                                // Szám bekérése
   write(x,' : ');
   if x in ket then write ('  2 -vel osztható');
   if x in har then write ('  3 -mal osztható');
   if x in hat then write ('  6 -tal osztható');
   writeln;                                    // Sor emelés
  until not (x in [1..255]);                   // Leállás ha  255 < x < 1
end.
```

A program egy futási képe az alábbi ábrán látható.



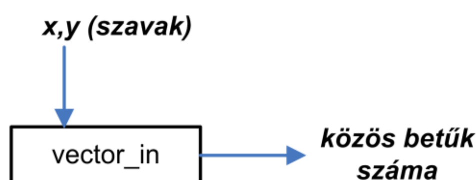
34. ábra: A halmaz típusú változókat kezelő program képernyőképe

A halmaz típusú adatok használatára nézzünk egy másik példát.

A program a halmaz használatának előnyét mutatja be. Írjunk Pascal függvényt, amely két szó közös betűinek számát adja meg a helytől és ismétléstől függetlenül (a kis és nagybetűt nem különböztetjük meg).

A feladat megoldása során képezzük a szavakat alkotó betűk halmazát. Ezen két halmaz közös elemei adják a közös betűk számát.

Tervezzük meg a függvényt.



35. ábra: A „common" függvény terve

Utasításlista készítés.

```
// Halmaz típusú változó
// Kis Imre 2010.06.05.

program kozosbetu;

type   t_betu   = 'A'..'Z';                    // nagybetű típus
```

```
  t_kozos  = set of t_betu;              // halmaz típus a nagybetűkből
  st20     = string [20];                // szavak típusa

var   w1,w2   : st20;                    // szavak globális változói

function common( x, y : st20 ) : byte;
 var   k           : byte;               // lokális segédváltozó
       c           : char;
       b1, b2, bc : t_kozos;             // halmaz típusú segédváltozók
 begin                                   // függvény törzse
  b1:=[ ]; b2:=[ ];                      // változók ürítése
  for k:=1 to length (x) do b1 := b1 + [ upcase ( x [ k ] ) ];      // Első szó betűi
                                         // ből készített halmaz
  for k:=1 to length (y) do b2 := b2 + [ upcase ( y [ k ] ) ];      // Második szó
                                         // betűiből álló halmaz
  bc := b1 * b2;                         // közös elemek: konjunktív halmaz
  k := 0;
  for c :='A' to 'Z' do if c in bc then inc(k);       // közös betűk számlálása
  common := k;                           // függvény értéket kap
 end;

begin                                    // a főprogram törzse
 write( ' Első szó        : '); readln ( w1 );
 write( ' Második szó   : '); readln ( w2 );
 writeln ( 'A közös betűk száma = ' , common( w1, w2 ) );      // Függvény hívása
 readln;
end.
```

Programunk futási képe az alábbi képen látható.

```
D:\Oktatás\Számtech1\EA-10\közösbetü\Project2.exe
 Elsô szó       : Pista
 Második szó    : Margitka

A közös betük száma = 3
_
```

36. ábra: A közös betűket kereső program képernyőképe

Gyakorló feladatok

F01. Készítsünk eljárást, amely 5 lottószámot sorsol ismétlés nélkül. Használjuk ki a halmazok tulajdonságát.

F02. Készítsünk programot, amely a 256-nál kisebb prímszámokat állítja elő Erathosztenész módszerével. (A módszer leírását megtaláljuk a http://hu.wikipedia.org/wiki/Eratoszthenesz címen.)

Ellenőrző kérdések

K01. Hány, és milyen eleme lehet egy halmaz típusnak?

K02. Mikor használjuk a halmaz adattípust?

K03. Mi a standard skalár és a halmaz típus alapvető különbsége?

5.6 Mutatók, dinamikus adatok

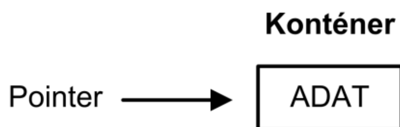
Az eddig használt mennyiségek (változók, szegmensek) mind statikusak voltak. Ez azt jelenti, hogy a programfejben meghatározott módon a Pascal fordító ezeknek a memóriában lefoglalta a deklarációban megadott helyet.

Például a **var** x : **array** [1..20] **of integer** ; utasítás hatására a fordító 20 egész-típusú (2 byte) rekeszt lefoglal az *x* változó részére.

Gyakran megtörténhet, hogy a tömb tényleges mérete (amely a program futása közben kerül meghatározásra, pl. beolvasás nyomán) ennél lényegesen kisebb. Ez természetesen rontja a memória kihasználást, sőt nagyméretű tömbök esetén már gondot okozhat. Ugyanilyen problémát okozhat a statikus **string** típusú változók használata.

Például a **var** nam : **string**[30] ; deklaráció esetén a *nam* változó számára a fordító 31 byte helyet foglal le, noha az aktuális értéke (Pl. 'Kis Ili' esetében) csak 7 byte hosszú.

A vázolt problémák, megoldására ajánlja fel a Pascal (és más nyelvek is) a dinamikus változók használatának lehetőségét. A dinamikus adatnak nincs neve, de van típusa. A ***pointer*** a dinamikus adat elérésének az eszköze.



37. ábra: A dinamikus adat felépítése

A dinamikus adatot tároló rekeszt (vagy rekeszcsoportot) konténernek nevezzük. A Pascal ezeket a konténereket a program futása közben hozza létre.

A továbbiakban lépésről lépésre mutatjuk be a dinamikus adatok használatát.

a) A pointerek típus definíciója

A Pascalban léteznek standard pointer típusok. A teljesség igénye nélkül ezek közül felsorolunk néhányat:

Pointer bármely típusú adatra mutathat:

PString	dinamikus stringre mutat
PByteArray	byte típusú tömbre mutat
PDouble, PExtended, PSingle	valós típusú adatra mutat
PInteger	integer típusra mutat, stb.

Mi azonban a programozó által definiált pointerekkel foglalkozunk. A pointer típust a ^ (karát) szimbólum jelzi.

```
type t_bptr   = ^byte;          // a pointer név nélküli byte típusú adatra mutat
     t_iptr   = ^integer;       // integer pointer típus
     t_ptrec  = ^number;        // record pointer típus
     number = record            // előre-definiált rekord
                nr   : word;    // Adatmező
                next : t_ptrec; // Pointer mező
              end;
```

b) A pointerek deklarálása

A fenti típusdefiníciók felhasználásával deklarálhatjuk az alábbi pointer változókat :

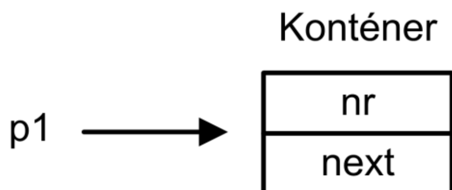
var p1,p2 : t_ptrec; // tipizált pointer deklaráció (rekordra mutat)
p3 : t_iptr; // tipizált pointer deklaráció (integer típusra mutat)
p4 : **^real**; // nem tipizált real pointer direkt deklarációja

Láthatjuk, hogy lehet előzetesen definícióval meghatározott típusú, és direkt, azaz típusdefiníció nélkül bemutatni pointer változókat.

c) Műveletek pointerekkel

A pointer típusú változó inicializálása:

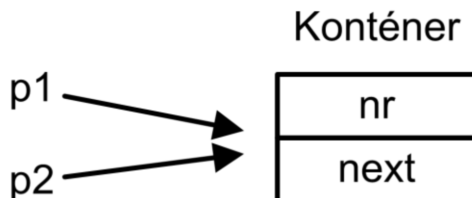
new (p1); A pointer egy új, a p1 által adott típusú dinamikus konténerre mutat. Ez példánkban a *number* típusú, névtelen rekord.



38. ábra: A p1 pointer inicializálás után

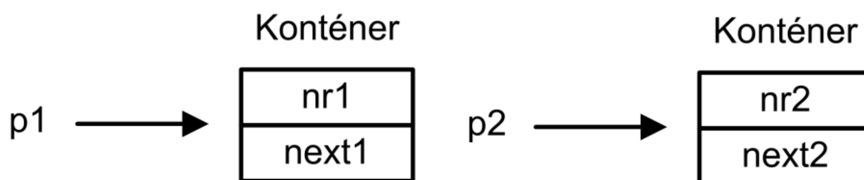
Az értékadó utasítások a következők lehetnek.

A p1:= p2; művelet végrehajtása után mindkét pointer azonos dinamikus konténerre mutat.



39. ábra: A p1:=p2; művelet eredménye

A p1^ := p2^ ; művelet végrehajtása után a két dinamikus konténer ***tartalma azonos*** lesz, a két pointer különböző marad.

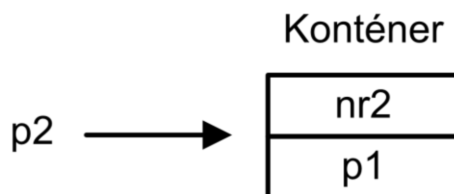


40. ábra: A p1^:=p2^; művelet eredménye

Figyeljük meg, hogy amíg a pointer típusdefiniálásakor és deklarálásakor a ^ jel a típus előtt van, addig az értékadáskor a változó neve mögé írjuk.

A *number* rekordtípus második mezője (next) szintén t_ptrec típusú, így a következő értékadás helyénvaló: p2^.next := p1;

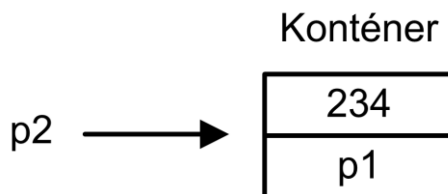
A művelet végrehajtása után a konténer tartalma az alábbi ábrán látható.



41. ábra: A p2^.next := p1; művelet eredménye

Hasonlóképpen adhatunk értéket a konténer másik mezőjének is :

p2^.nr := 234; melynek végrehajtása után a konténer tartalma a következő ábrán látható.



42. ábra: A p2^.nr := 234; művelet eredménye

A pointer típusú változó nullázása a p1 := **nil**; utasítással történhet. Ennek eredményeképpen a p1 nem mutat semmilyen címre. A **nil** a Pascalban védett kulcsszó.

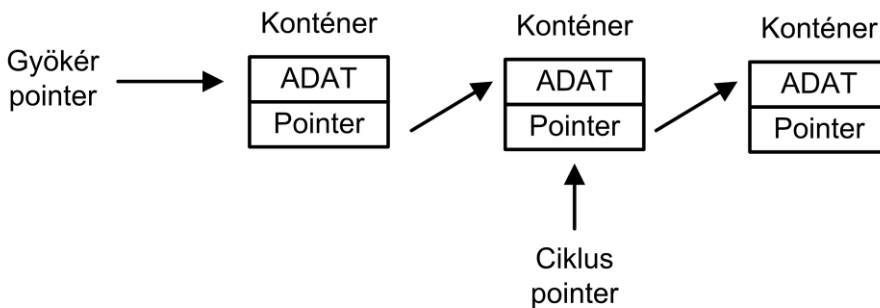
A dinamikus konténer által elfoglalt hely a program futása közben felszabadítható a **Dispose** (p1); utasítással. Ezután a p1 által jelölt konténer megszűnik (de a pointer nem).

d) Műveletek dinamikus adatokkal

p3^ :=125;	// p3 integer konténer felveszi a 125 értéket
p4^:= 0.01;	// p4 valós konténer felveszi a 0.01 értéket
inc (p3^ , 5);	// p3 konténer új értéke 130 lesz (előzőleg 125 volt)

e) Dinamikus adatokkal különféle stuktúrák építhetők fel.

– Előreláncolt dinamikus vektor. Ennek szerkezetét az alábbi ábra mutatja.



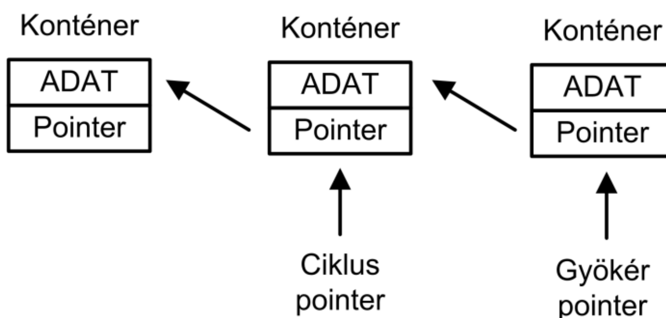
43. ábra: Előreláncolt dinamikus vektor

Az egész vektor egyetlen pointerrel (gyökér) megadható, amely a legelső konténerre mutat. Minden konténer tartalmazza a következő elemre mutató pointert. Az utolsó elem pointer mezője **nil**, ez jelzi a vektor végét.

– Hátraláncolt dinamikus vektor. Hasonló, mint az előző, de minden konténer pointere az előző elemre mutat.

–

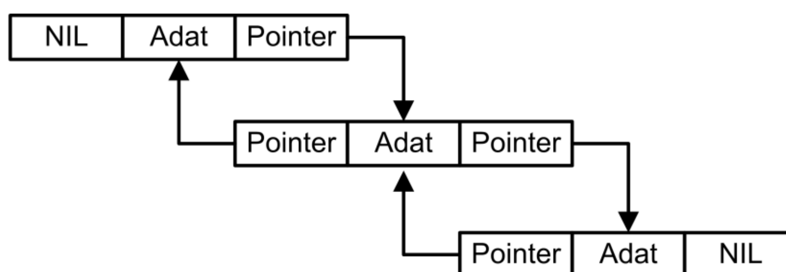
 © Aradi, Bécsi, Gyenes, Péter, BME



44. ábra: Hátraláncolt dinamikus vektor

A két típus között a felhasználás módja tesz különbséget. A bemutatott egyszeresen láncolt vektorok használatát nehezíti, hogy csak egy irányban lehet végigjárni a vektor elemeit. Ezt a hiányosságot küszöböli ki a következő típus.

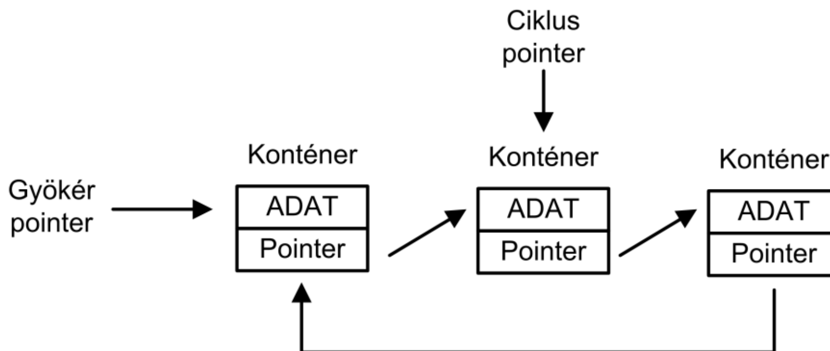
– Kétszeresen láncolt dinamikus vektor.



45. ábra: Kétszeresen láncolt dinamikus vektor

Amint az ábrán is látható, minden dinamikus konténer 2 pointert használ. Az egyik előre, a másik hátrafelé mutat. Ezzel a vektorban mindkét irányban lehet lépkedni. Az egypointeres vektor struktúra másik formája a következő.

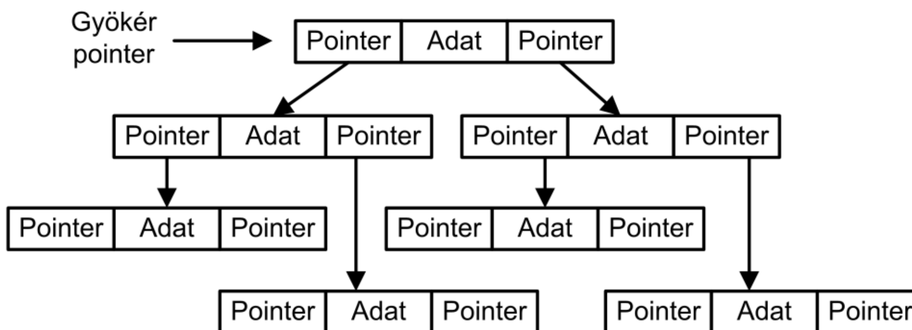
Egyszeresen láncolt dinamikus gyűrű.



46. ábra: Egyszeresen láncolt dinamikus vektor

A vektor utolsó elemének pointere az első elemre mutat. Ezzel a megoldással minden elem elérhető akkor is, ha csak a vektor egyetlen pointerét ismerjük. Létezik még a kétszeresen láncolt dinamikus gyűrű is. Ennek felépítése az utóbbi két típus egyesítésével szerkeszthető meg.

Különleges struktúra a dinamikus fa. Ennek felépítését az alábbi ábra mutatja.



47. ábra: Dinamikus fa

Ezekkel a speciális fa struktúrákkal gyors (rekurzív) dekódoló programok készíthetők. A példatárban találunk egy Morse dekódoló programot, amely ezt a fa struktúrát használja fel.

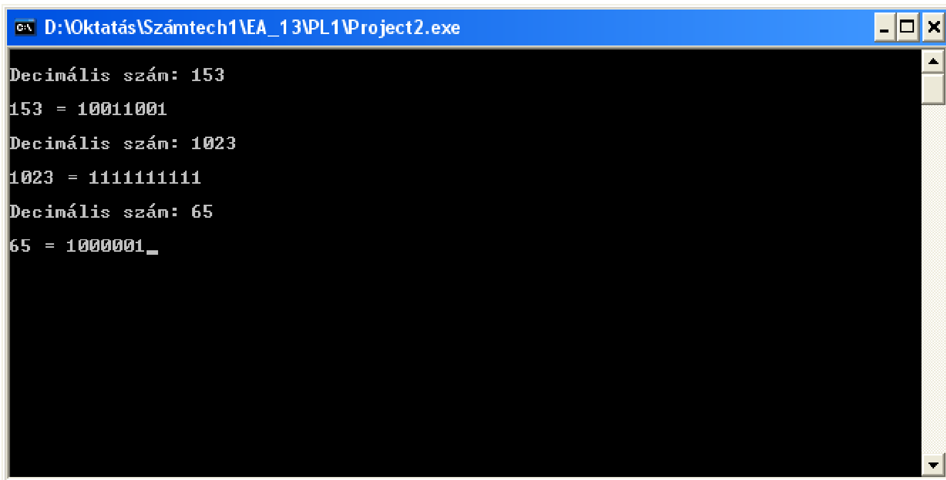
Nézzünk egy példát a dinamikus vektorok alkalmazására. A program egy decimális számot alakít át binárisba úgy, hogy a biteket dinamikus vektorba gyűjti. Minden dinamikus konténer pointere az előző rekordra mutat, a legelső pointe-

 © Aradi, Bécsi, Gyenes, Péter, BME

re = NIL. Az egyes biteket az osztó algoritmus alkalmazásával határozzuk meg (lásd 1.1.1 fejezet). A feladatot hátra láncolt dinamikus vektor felépítésével oldjuk meg. program utasítás listája a következő.

```
// Hátra láncolt dinamikus vektor
// Kis Pál 2010.01.01.

program dec_bin;

type     t_ptr = ^bit;                     // konténer típus definíció
          bit= record
                  num : 0..1;              // Adat mező : intervallum típusú
                  prev : t_ptr;            // Hátra mutató pointer
               end;

var     n,m      : longint;                // globális változók
      pold, p   : t_ptr;
begin
 write('Decimális szám: '); readln(n);     // Az átalakítandó decimális szám beolva//sása
 m := n;                                   // Elmentjük az eredeti számot
 pold := nil;                              // Nullázzuk a gyökér pointert
 while n>0 do                              // Amíg a szám 0 nem lesz
  begin                                    // Utasítás nyitó zárójel
   new(p);                                 // Új dinamikus konténer létrehozása
   p^.num := n mod 2;                      // Előállítjuk a soron következő bitet
                                           // (maradék)
   n := n div 2;                           // Leosztjuk kettővel a számot
   p^.prev := pold;                        // Az új konténer pointere az előző elemre //mutat
   pold:= p;                               // Aktuális pointer lesz a régi
  end;                                     // Utasítás csukó zárójel

{ Most kiíratjuk a dinamikus vektort visszafelé }

 writeln; write(m,' = ');                  // Az eredeti decimális szám
 while p <> nil do                         // Amíg a vektor elejére ér ismétel
  begin                                    // Utasítás nyitó zárójel
   write (p^.num);                         // Az aktuális bit kiírása
   p := p^.prev;                           // A pointer visszalép
  end;                                     // Utasítás csukó zárójel
 readln;
end.
```

Programunk futási képe a következő.



48. ábra: A kettes számrendszerbe átváltó program képernyőképe

Gyakorló feladatok

F01. Készítsünk *make_din* néven eljárást, amely paraméterben átvesz egy egész számot (n), majd létrehoz egy n elemű előreláncolt dinamikus vektort 10 és 50 közötti véletlen számokból. A gyökérpointert kimenő paraméterben adja át.

F02. Készítsünk Pascal programot, amely az F01. feladatban leírt eljárást aktivizálja, és annak pointerével a dinamikus vektor elemeit a képernyőre listázza.

Ellenőrző kérdések

K01. Mi a dinamikus adat meghatározása ?

K02. Milyen dinamikus struktúrákat ismer ?

K03. Hogyan deklarálhatunk egy dinamikus konténerre mutató pointert ?

5.7 File kezelés a Pascal nyelvben

A file (állomány) logikailag összefüggő adatok rendezett halmaza. Mi a tárgyalásunk során szűkebb értelemben a háttértárolón (lemezegység, CD, külső tárolón) elhelyezett adathalmazként kezeljük a file-okat.

A file kezelést a Pascal 5.szintjeként értelmezzük. Ismétlésképpen álljon itt a korábbi 4 szint felsorolása :

– 1. szint → Építőelemek
– 2. szint → Kifejezések
– 3. szint → Utasítások
– 4. szint → Program szegmentáció

Korábbi módszerünket alkalmazva, a file kezelést is lépésenként mutatjuk be.

a) A file azonosítása

A lemezen teljes, vagy fizikai névvel: ***drive : <útvonal> név.kiterjesztés***, ahol

- a drive (meghajtó) a lemezegység logikai betűjele,
- az elérési út (path) a könyvtárszerkezet elemeiből áll,
- a név a file tényleges neve, amely lehetőleg szimbolikus név legyen, mivel egyébként némely fordító hibaüzenetet ad,
- végül a kiterjesztés a file típusára utal. Sok általánosan elterjedt kiterjesztés van használatban, ezeket ne használjuk más célra.

Gyakori kiterjesztések:

– pas: Pascal forrás program
– dpr: Delphi project
– txt: Szöveges (text) állomány
– doc: Dokumentáció
– xls: Excel állomány
– ddd: Diagram designer (folyamatábra tervező)
– dat: Adatok állománya, stb.

A file neve és kiterjesztése közé pontot teszünk. Egy fizikai file név lehet a következő az alábbi.

D:\work\tp7\matrix.dat

A Pascal programban a file azonosítása szimbolikus névvel (file változó) történik,

például: f1,f2.

© Aradi, Bécsi, Gyenes, Péter, BME

Hasznos, ha a programozó bizonyos konvenciók szem előtt tartásával választja meg a változó nevét. E konvenció szerint a file változó mindig ***f*** betűvel kezdődik.

b) A file típusa

A Pascal három típusba sorolja a file-okat:

– típusos file (bármely standard v. nem standard típus)
– típus nélküli file
– text típusú file

A file kezelése típusonként eltér.

c) A file mérete

A file méretét a benne tárolt azonos típusú rekordok száma határozza meg. A rekordok számát az egésztípusú **filesize**(f) standard függvény adja meg. Az egyes rekordok hosszát a **sizeof** (elem) függvény adja meg byte-okban. Így a file teljes helyfoglalása a **filesize** (f) * **sizeof** (elem) szorzattal adódik.

d) A file változó

A programban deklarált szimbolikus file névhez (pl. f) a string típusú tényleges file nevet (pl. nev) az **assign** (f,nev) standard eljárás rendeli hozzá.

e) A file kezelése

– - szekvenciális (a rekordok elérése sorosan –egymás után –történik)
– - random (bármely rekord közvetlen elérhető)

f) A file deklarálása

Más változókhoz hasonlóan, a file változót is deklaráljuk.

– típusos
  var f : **file of** <típus>; például.: **var** f : **file of byte**;
– típus nélküli
  var f : **file**;
– text
  var f : **text**;

 © Aradi, Bécsi, Gyenes, Péter, BME

g) A file megnyitása

A file-t használata előtt meg kell nyitni. Ezt kétféleképpen tehetjük:

– olvasásra

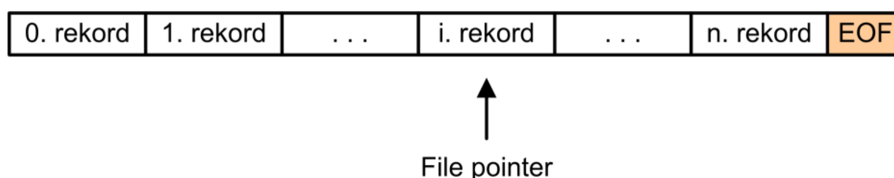
reset (f); Típusos file esetén írni is lehet a file-ba, szöveges file esetén csak olvasni.

– írásra

rewrite (f); Létező fájlnál törli a file tartalmát!
append (f); Csak szöveges file esetén működik, a file végéhez lehet hozzáírni.

A ***file pointer*** a file egy rekordjára mutat. Megnyitáskor az első fizikai, de 0.-k számú rekordra mutat.

A file szerkezetét az alábbi ábra szemlélteti. A file végén az EOF (End of File) marker található, amely logikai mennyiségként kezelhető : EOF (f) mindaddig **true** , amíg a file pointer $\leq$ n .



49. ábra: A típusos file szerkezet

Ne tévesszük össze a file rekordját a korábban megismert **record** adattípussal. A file pointer is csak névrokona a dinamikus adatra mutató pointer típusoknak.

h) Adat olvasása file-ból

A már megnyitott file egy rekordjának olvasása a **read** (f, *adat*) kétparaméteres eljárással történik. Az adatáramlás iránya : f → *adat.* Minden olvasás után a file pointer értéke eggyel nő. Fontos szabály, hogy az *adat* változó típusa korrelál (típusos file estén azonos) a file típusával.

i) Adat írása file-ba

Az *adat* változót a már megnyitott file egy rekordjára a **write** (f, *adat*) kétparaméteres eljárással írhatjuk fel. Az adatáramlás iránya : *adat* → f . Minden írás után a file pointer értéke eggyel nő.

j) A file lezárása

A file-t használatának végén a **close** (f) eljárással le kell zárni. A Pascal programból való kilépés minden megnyitott file lezárását eredményezi.

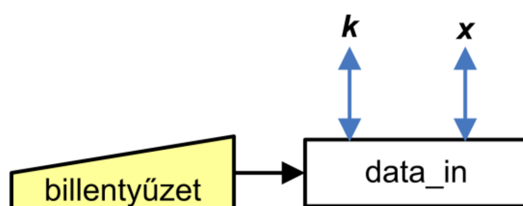
A későbbiekben még további file kezelő eljárásokat is megismerünk.

5.7.1 Típusos file

Nézzünk most két minta példát a típusos file kezelésére.

Olvassunk be egész-számokat billentyűről az *a* tömbbe, írjuk fel a tömböt a lemezre egy egésztípusú állományba, majd olvassuk vissza a lemezről az adatokat a *b* tömbbe és hasonlítsuk össze *a* -val. A feladat megoldása során készítsünk eljárást a beolvasásra, a file-ba való felírásra és a visszaolvasásra.

A beolvasó eljárás terve a következő. A beolvasott adatok számát a *k* formális paraméter helyén, míg az adatok tömbjét az *x* formális paraméter helyén adja át az eljárás. A *k* egésztípusú, míg *x* alkalmas tömb típusú, mindkettő kimenő paraméter.



50. ábra: A beolvasó eljárás terve

Az adatok tömbjének típusát a

```
type t_tomb  = array[1..10] of integer;
```

definíció adja meg.

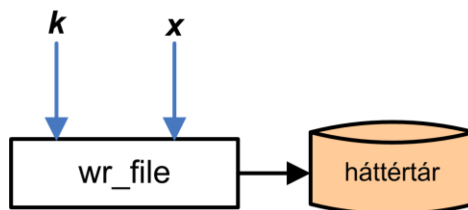
A beolvasó eljárás a következő.

```
procedure data_in (var k: byte ; var x: t_tomb);
   var i : byte;                           // lokális ciklusváltozó
   ch : char;                              // segédváltozó
   begin
     i:=1;
     repeat
       write('Egész szám : ');
```

```
      readln( x[i] );                 // Az i.-ik adat bekérése
      inc( i );                       // Ciklusváltozó növelése
      write('Tovább (I/N) ? : ');
      readln( ch );                   // Folytatja ?
    until  ch in ['n' , 'N'] ;        // N,n esetén leáll
    k:= i – 1;                        // k = adatok darabszáma
  end;
```

Az eljárás mindaddig ismétli a beolvasást, amíg a Tovább kérdésre a beolvasott karakter kis-, vagy nagy N betű.

Második eljárásunk az *x* paraméter helyén átvett *k* darab adatot a lemezre írja. Ezek megfelelő típusú bemenő paraméterek. Az eljárás feltételezi, hogy a file fizikai neve már hozzá van rendelve a file változóhoz. Ezt az utasítást – **assign** (f, fnam) – célszerűen a főprogram törzs elején adjuk ki, mivel több eljárás is használja a file-t.

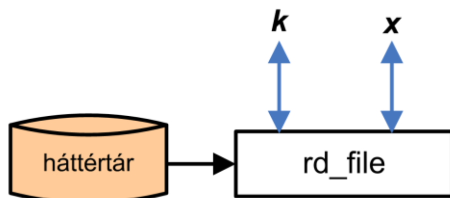


51. ábra: A file felíró eljárás terve

Az eljárás utasításlistája a következő.

```
procedure wr_file ( k: byte ; x: t_tomb );
  var i : byte;                       // lokális ciklusváltozó
  begin
    rewrite ( f );                    // file megnyitása írásra
    for i:=1 to k do write (f , x[i] );   // k db elem felírása
    close ( f );                      // file lezárása
  end;
```

Harmadik eljárásunk a file tartalmát olvassa fel egy tömbbe. A formális paraméterek a *data_in* eljáráséhoz hasonló módon kimenő típusúak.

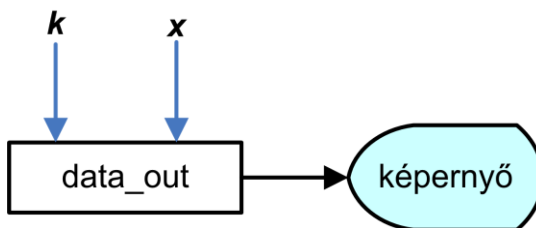


52. ábra: A file olvasó eljárás terve

Az eljárás utasításlistája a következő.

```
procedure rd_file (var k: byte ; var x: t_tomb);
  var i : byte;                              // lokális ciklusváltozó
  begin
    reset ( f );                             // file megnyitása olvasásra
    k := filesize ( f );                     // A file rekordjainak száma
    if k > 0 then                            // Ha a file nem üres
      for i:=1 to k do read (f , x[i] );     // Beolvasás
    close ( f );                             // file lezárása
  end;
```

Végül megírjuk a tömb adatait kiíró eljárást.



53. ábra: A kiíró eljárás terve

```
procedure data_out (k: byte ; x: t_tomb);
  var i : byte;                              // lokális ciklusváltozó
  begin
    writeln;                                 // Soremelés
    for  i:=1 to k do write(x [i] : 4 );     // A számok kiírása 4 karakterhelyre
  end;
```

Ezek után megírhatjuk a teljes programot az alábbi módon.

```
// Egésztípusú file kezelése
```

```
// Kis Aliz 2010.05.01.

program wr_rd_integer_file;

type t_tomb  = array[1..10] of integer;
var     f      : file of integer;
        fnam : string [20];             // A file fizikai nevének változója
        n      : byte;                  // Darabszám
        a, b   : t_tomb;                // Adattároló tömbök

< a megírt 4 eljárás bemásolása >

begin                                   // Fő program törzse
  write('A file neve: '); readln( fnam ); // A file fizikai nevének bekérése
  assign(f, fnam );                     // Nevek egymáshoz  rendelése
  data_in ( n, a );                     // Adatok bekérése a –ba
  wr_file ( n, a );                     // Felírás file-ba
  rd_file ( n, b );                     // Visszaolvasás b – be
  data_out ( n, a );                    // Kiíratás :  a
  data_out ( n, b );                    // Kiíratás :  b
  readln;
end.
```

Programunk futási képe az alábbi lesz.

```
D:\Oktatás\Számtech1\EA-8\wr_rd_int\Project2.exe
A file neve: integers.dat
Egész szám : 12
Tovább (I/N) ? : i
Egész szám : 23
Tovább (I/N) ? : i
Egész szám : 345
Tovább (I/N) ? : i
Egész szám : -12
Tovább (I/N) ? : n

  12  23 345 -12
  12  23 345 -12
```

54. ábra: Az egész típusú file-t kezelő program futási képe

Gyakorta szükség van összetett adattípusok file-ban tárolására és kezelésére. A továbbiakban erre adunk egy példát.

Az összetett adattípus egy dolgozó nevét, születési dátumát és órabérét tartalmazza. Ehhez az alábbi típusdefiníciót készítjük.

```
type  t_worker = record
                    nam  : string [30];
                    birth  : longword;       // ÉÉÉÉHHNN
                    wage : single;           // EUR
                 end;
```

A mezőleírókhoz az angol elnevezéseket azért használjuk, mert a magyar nyelvű változat nem megengedett ékezetes betűket kívánna (magyarul: *nev, ber, szul* lenne). A dolgozók tömbjének típusa a következő.

```
type t_staff = array[1..10] of  t_worker;
```

Példánkban csak 10 dolgozónak foglalunk helyet statikus tömbbe, ha ezzel jól működik a program, akkor igény szerint növelhető a tömb mérete, vagy nagy dolgozó számhoz dinamikus tömböt célszerű használni.

Először a dolgozók adatainak bevitelére készítünk eljárást. A terv hasonló az előző feladatban használt *data_in* eljárás tervéhez. Az eljárás utasításlistája eltér az egyszerű egész értékeket beolvasó szegmenstől.

```
procedure input (var k : byte ; var x : t_staff );
 var i : byte;                            // lokális ciklusváltozó
 begin
  write('Dolgozók száma : ')
  readln(k) ;                             // A dolgozók számának megadása
  for i:=1 to k do                        // ismétlés k - szor
   with x [i] do                          // Az i.-ik dolgozó
    begin                                 // Utasítás zárójel
     write ('Név                : ');
     readln (nam) ;                       // Név bekérése
     write ('Születés dátuma : ');
     readln (birth) ;                     // Születés dátuma
     write ('Órabér             : ');
     readln(wage) ;                       // Órabér
    end;                                  // Utasítás zárójel
 end;
```

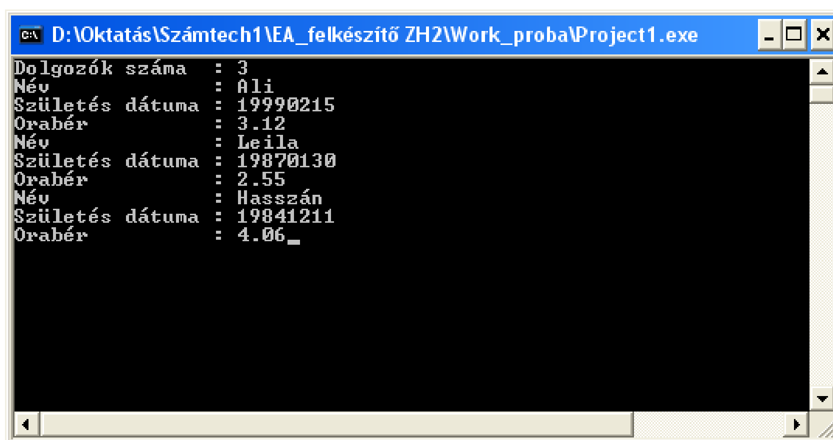
Az adatok tömbjét a file-ba író eljárás terve és paraméterezése is hasonló az előző feladatban megismert *wr_file* eljáráshoz.

```
procedure save ( k : byte ; x : t_staff );
 var i : byte;                            // lokális ciklusváltozó
 begin
   rewrite ( f );                         // file megnyitása írásra
   for i:=1 to k do write (f , x[i] );    // k db elem felírása
   close ( f );                           // file lezárása
 end;
```

Az eddig megírt programot célszerű kipróbálni: aktivizálni kell az *input* és *save* eljárásokat ! Erre az alábbi programot írjuk meg.

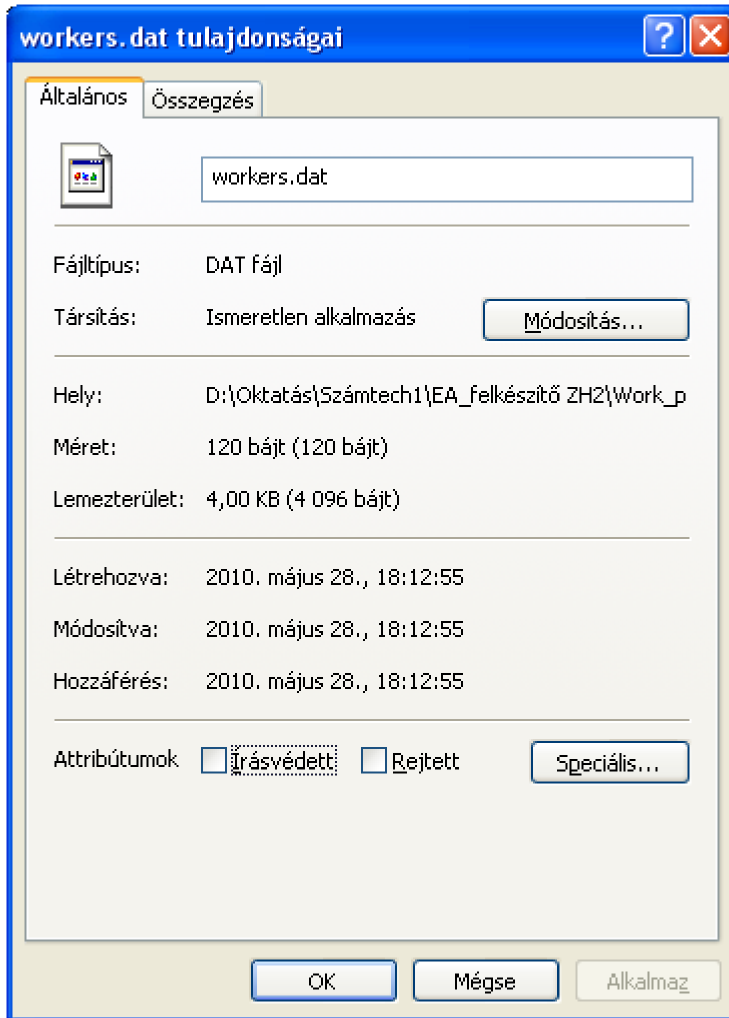
```
// Rekordok file-ba írása
// Barna Pál 2009.05.06.

program workers;

type  t_worker = record
                    nam  : string [30];
                    birth  : longword;      // ÉÉÉÉHHNN
                    wage : single;           / EUR
                  end;

         t_staff = array[1..10] of  t_worker;

const fnam = ’workers.dat’;                  // A file neve

var   adatok : t_staff ;
              n  : byte;
              f   : file of t_worker;        // Rekord típusú file

< az input és save eljárások bemásolása >

begin
  input (n,adatok);                          // Dolgozók adatainak bekérése
  assign (f, fnam);
  save (n, adatok);
end.
```

Eddig megírt próba programunk futási képe a következő.



55. ábra: Az record típusú file-ba író program futási képe

Ellenőrizni kell a ***workers.dat*** file meglétét és méretét.



56. ábra: A workers.dat file tulajdonságai

A file a könyvtárban létezik. Tudjuk, hogy 3 rekord van benne. Egy rekord helyfoglalását a **sizeof** (t_worker) függvény adja meg. A rekord első mezője (nam) 31 byte, a második (birth) 4 byte, míg a harmadik (wage) 4 byte hosszú. Így egy dolgozó rekordja összesen 39, kerekítve 40 byte, 3 dolgozóé 120 byte.

Megjegyezzük, hogy a Pascal 16 bites PC rendszerre íródott, az alapfogalmak fejezetben megadott adattípusok 32 bites gépi reprezentációra (Delphi) vonat-

 © Aradi, Bécsi, Gyenes, Péter, BME

koznak. Ezért az ott megadott mezőhosszak eltérnek a Pascal által használtak-tól.

A továbbiakban programot készítünk a *workers.dat* file-ban rögzített adatok használatára. Programunk a *load* eljárás segítségével felolvassa a dolgozók adatait, a születési dátum szerint rendezve a képernyőre listázza. A file-ból való felolvasáshoz deklaráljunk *load* néven eljárást. Ez is hasonló a korábban bemutatott *rd_file* minta eljáráshoz.

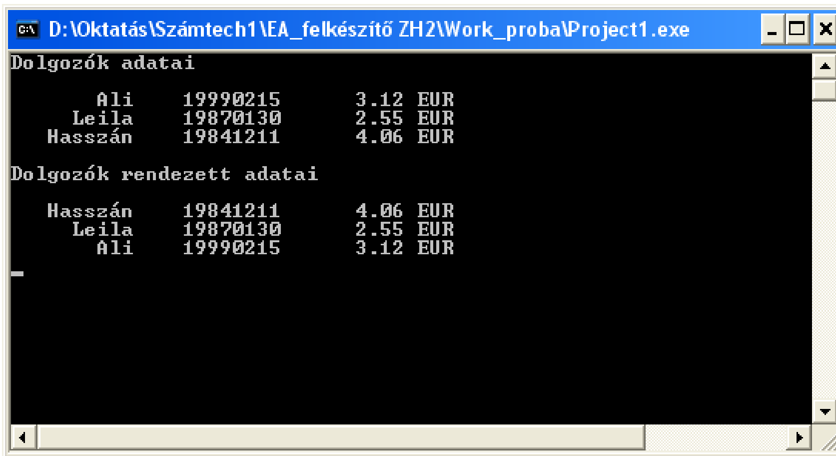
```
procedure load (var k : byte ; var x : t_staff );
 var i : byte;                          // lokális ciklusváltozó
   begin
    reset ( f );                        // file megnyitása olvasásra
    k := filesize ( f );                // A file rekordjainak száma
    if k > 0 then                       // Ha a file nem üres
      for i:=1 to k do read (f , x[i] ); // Beolvasás
    close ( f );                        // file lezárása
  end;
```

A születés dátuma szerinti sorba rendezést a már megismert buborék algoritmus alkalmazásával oldjuk meg. Programunk az alábbi lesz.

```
// Rekordok file-ból olvasása és rendezése
// Zöld Péter 2009.05.06.

program workers;

type  t_worker = record
                    nam  : string [30];
                    birth  : longword;
                    wage : single;
                  end;

        t_staff = array[1..10] of  t_worker;

const fnam = 'workers.dat';             // A file neve

var   adatok : t_staff ;
         n,i,j  : byte;
            f   : file of t_worker;     // Rekord típusú file
            z  : t_worker;              // Segédváltozó a cseréhez

< a load eljárás bemásolása >

begin
 assign (f, fnam);
 load (n, adatok);                      // file felolvasása
```

```
for i:=1 to n do                              // Listázás rendezés előtt
 with dolg[i] do
  writeln (nam:10, birth:12, wage:10:2,' EUR');

for i:=1 to n do                              // Rendezés kettős ciklussal
 for j:=1 to n-1 do
  if adatok[j].birth > adatok[j+1].birth then
   begin
     // Két rekord cseréje
    z:= adatok[j]; adatok[j]:=adatok[j+1]; adatok[j+1]:=z;
   end;

for i:=1 to n do                              // Listázás a rendezés után
 with dolg[i] do
  writeln (nam:10, birth:12, wage:10:2,' EUR');
readln;
end;
```

A program futási képe a következő.



57. ábra: Az record típusú file-ból olvasó program futási képe

Gyakorló feladatok

F01. Készítsünk programot, amely az első 5 természetes szám ismétlés nélküli permutációit állítja elő, listázza a képernyőre, majd *perm5.dat* néven a lemezre írja.

F02. Hozzunk létre *angle.dat* néven a lemezen olyan file-t, amely 100 db random generátorral sorsolt 0 és 180° közötti szögeket tartalmaz. Ol-

vassuk fel ezeket a file-ból és listázzuk a képernyőre (soronként 8 szöget 2 tizedes pontossággal).

Ellenőrző kérdések

K01. Milyen file típusokat ismer?

K02. Melyik a file méretét megadó utasítás?

K03. Mi jelzi a file végét a lemezen?

K04. Mik a file kezelés lépései?

5.7.2 Text file

A szöveges állományok kezelésére annak jelentősége miatt a Pascal számos lehetőséget (segédprogramokat, rutinokat) ajánl. A text file csak karaktereket tartalmazhat. Szerkezete és kezelése eltér az előző fejezetben megismert típusos file kezeléstől. A text file változó hosszúságú rekordokból áll, szemben a típusos file-al, amelynél a rekordhossz állandó. A text file szerkezetét az alábbi ábra mutatja.

58. ábra: Az text file szerkezete

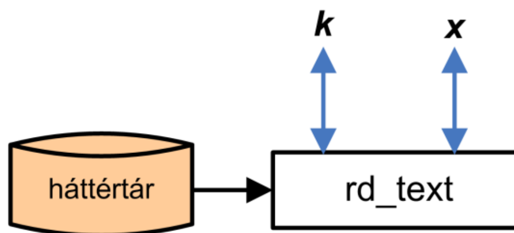
A text file minden rekordja egy szövegsor, amelyet string típusként kezelhetünk. A sorok végeit az EOL (End of Line) marker jelzi. A típusos file-nál erre nem volt szükség az ismert és állandó rekordméret miatt. Az egész állomány végét az EOF (End of File) marker jelzi.

Írás text file-ba leggyakrabban a **writeln** (f, sor); utasítással történhet. Az adatáramlás iránya : f → sor. Olvasás text file-ból a **readln** (f, sor); utasítással történhet. Ekkor az adatáramlás iránya : sor → f.

A file fizikai és logikai nevének egymáshoz rendelésére a már megismert **assign**(f, fnam); utasítást használjuk. A text file megnyitása és lezárása is a típusos file kezelésénél bemutatott módon történik. A text file írása és olvasásá-

ra javasoljuk a soronkénti műveletvégzést alkalmas string típusú változó felhasználásával.

A fentiek szemléltetésére nézzünk egy példát: Olvassunk be egy létező szöveges állományt a lemezről. Írjuk ki a képernyőre, konvertáljuk nagybetűsre, majd azonos néven, de *big* kiterjesztéssel írjuk fel a lemezre. Végül olvassuk vissza a lemezről és írjuk ki a képernyőre a módosított állományt. A megoldáshoz eljárásokat tervezünk.



59. ábra: A file olvasó eljárás terve

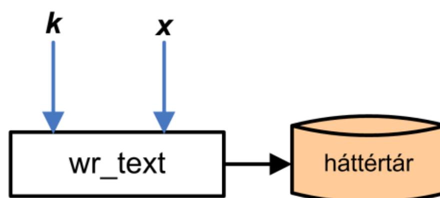
Az első formális (k) paraméter egésztípusú, míg a második a sorokat tároló tömb típus kell legyen.

```
type    t_sor = string[80];              // típus 1 sor szöveghez
                                         // (max. 80 karakter)
        t_txt = array[1..100] of t_sor;  // típus az egész állományhoz
```

A megtervezett eljárásunk utasítás listája a következő.

```
procedure rd_text (var k: byte ; var x: t_txt );
 var   ext : string[3];                  // A kiterjesztés 3 karakteres
 begin
  write(’File neve            : ');
  readln(fnam);                          // A text file neve kiterjesztés nélkül
  write(’File név kiterjesztése : ');
  readln(ext);                           // Külön kérjük a kiterjesztést beolvasni
  assign(f, fnam + ’.’ + ext);           // file teljes név előállítása
  reset(f);                              // Megnyitás olvasásra
  k := 0;                                // sor számláló (formális paraméter)
  while not eof(f) do
   begin
    inc(k);
    readln ( f, x [ k ] );               // k.-ik sor beolvasása
   end;
  close ( f );
 end;
```

Szükségünk lesz a módosított szöveg file-ra írásához a *wr_text* eljárásra**.**



60. ábra: A file felíró eljárás terve

Az eljárás utasítás listája a következő.

```
procedure wr_text (k: byte ; x: t_txt);
 var i : byte;
 begin
  assign(f,nev + '.big');               // Módosított kiterjesztés
  rewrite( f );                         // File megnyitás írásra
  for i:=1 to k do writeln(f,x[i]);     // k db sor felírása
  close(f);
 end;
```

Ezek után megírhatjuk az egész, működő programunkat.

```
// Text file kezelése
// Kék Sára 2009.05.06.

program text_file;

type    t_sor  = string[80];            // 1 sor szöveg
        t_txt  = array[1..50] of t_sor; // az egész állomány

var      f      : text;
       fnam     : string[20];
        n,i,j   : byte;
        msg     : t_txt;

procedure rd_text (var k: byte ; var x: t_txt );
 < eljárás bemásolása >

procedure wr_text (k: byte ; x: t_txt);
 < eljárás bemásolása >

begin                                   // főprogram
 rd_text(n,msg);                        // Text file beolvasása
 writeln('A text file : ');
 for i:=1 to k do writeln(x[i]);        // Listázás a képernyőre
 for i:=1 to n do                       // Soronként
```

```
  for j:=1 to length(msg[i]) do
    msg[i,j]:=upcase(msg[i,j]);        // minden betű nagybetűsre váltása
  wr_text(n,msg);                      // File ba írás
  writeln('A módosított text file : ');
  for i:=1 to k do writeln(msg[i]);    // Listázás a képernyőre
  readln;
end.
```

Figyeljük meg, hogy az egész szöveget tartalmazó *msg[i]* változó egy indexes formája egy egész sort (az i-ediket), míg két indexes formája: *msg[i,j]* egyetlen karakter (az i.-edik sor j.-edik karaktere) jelöli.

A program futási képe az alábbi.

```
D:\Oktatás\Jegyzet példák\pl_textfile\Project1.exe
Az eredeti szöveg :

A file neve:             row
A file kiterjesztése:    txt

A text file:

Minden text file változó hosszúságú rekordordokból áll.
Minden rkord 1 sort tartalmaz.
A sor hosszát a Length(sor) függvénnyel határozzuk meg.
A sorok végét az ELOLN marker jelzi.
Egy sor beolvasása a Readln(f,sor) eljárással történik.


A módosított szöveg :

A FILE NEVE:             ROW
A FILE KITERJESZTéSE:    TXT

A TEXT FILE:

MINDEN TEXT FILE VáLTOZó HOSSZúSáGú REKORDORDOKBóL áLL.
MINDEN RKORD 1 SORT TARTALMAZ.
A SOR HOSSZáT A LENGTH(SOR) FüGGVéNNYEL HATáROZZUK MEG.
A SOROK VéGéT AZ ELOLN MARKER JELZI.
EGY SOR BEOLVASáSA A READLN(F,SOR) ELJáRáSSAL TöRTéNIK.
```

61. ábra: Az text file-t kezelő program futási képe

Gyakorló feladatok

F01. Készítsünk programot, amely két text file hosszát vizsgálja. A végeredmény az alábbi 3 üzenet valamelyike: - a két file azonos hosszúságú; - az első file hosszabb; - a második file hosszabb.

F02. Készítsünk programot, amely két text file összeadásából egy harmadikat készít a lemezen.

Ellenőrző kérdések

K01. Miben különbözik a típusos és text file rekordja?

K02. Mely utasítással olvasunk text file-ból?

K03. Mi jelzi a sor végét text file esetében?

5.7.3 Típusnélküli file

Némely alkalmazásban szükség lehet típusnélküli, helyesebben ismeretlen típusú file kezelésére. Ez akkor fordul elő, ha nem a saját alkalmazásunk állította elő a vizsgálni, vagy használni kívánt állományt. Ilyen esetekben eldönthetjük, hogy mekkora rekordméretet használjunk. Emlékeztetőül az ismert típusú file esetében a rekord méretét a **sizeof** (tipus) függvény adta meg. A rekord méretét a file megnyitásakor adjuk meg. Olvasásra való megnyitáshoz: **reset** (f , bl); kétparaméteres standard eljárást használjuk, ahol *bl* adja a rekord (vagy blokk) hosszát byte egységben. A blokkhossz megválasztása megfontolást igényel, mivel értéke összefügg a file méretével. A file mérete a blokkhossz egészszámú többszöröse kell legyen. Minden esetben jó a bl=1 választás, ekkor azonban a típusnélküli file temérdek rekordként (mindegyik 1 byte) lesz kezelve.

Írásra való megnyitáshoz a **rewrite**(f, bl); szintén kétparaméteres standard eljárást használjuk.

A típusnélküli file olvasása a **blockread**(f, membuf, **sizeof**(membuf), db); négy paraméteres eljárással történik, ahol

– f: file változó,
– membuf: deklarált memória terület, ahova a file adatát olvassuk,
– **sizeof**(membuf): memória puffer mérete,
– db: a beolvasott rekordok száma (az eljárás adja kimenő értékkent).

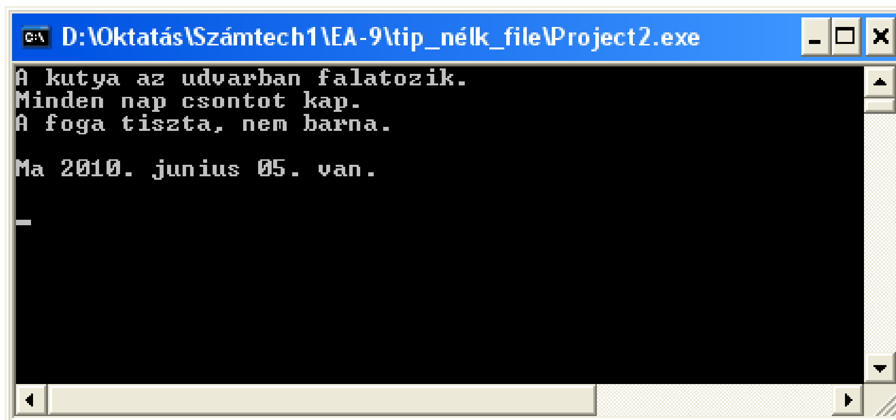
Hasonlóképpen írhatunk a file-ba a **blockwrite**(f,membuf,db); három paraméteres eljárással, ahol

– f: file változó,
– membuf: deklarált memória terület,
– db: írásra szánt rekordok száma (az eljárás bemenő paramétere).

Példaképpen olvassuk fel a 'dog' nevű ismeretlen típusú file-t, és listázzuk a képernyőre.

```
// Típusnélküli file kezelése
// Tasmán Ördög 2009.05.06.

program untyped_file;

var i,b : byte;
    db   : integer;
    buff : array [1..500] of byte;           // Memória puffer
    f    : file;                             // File változó
begin
 assign(f, 'dog');                           // Az ismeretlen file neve
 reset (f,1);
 repeat
  blockread(f, buff, 500, db);
 until eof(f);
 close(f);
 for i:=1 to db do write (chr(buff [i]));    // Kiíratás byte-onként
 readln;
end.
```

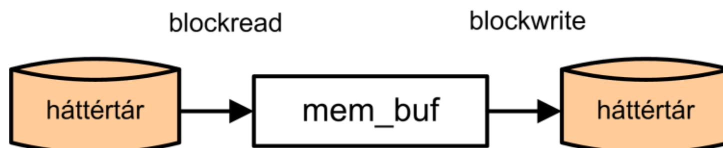
Programunk futási képe az alábbi.



62. ábra: A típus nélküli file-t kezelő program futási képe

Második példánk file-ok gyors másolását oldja meg. Programunkban 2 file-t kezelünk egyszerre. Az egyik a forrás file, ebből olvasunk, a másik a cél file, ebbe írunk. A file-ok típusa bármi lehet, ezért használjuk a típusnélküli file kezelési technikát. A másoláshoz egy memóriában elhelyezett tömböt (memória puffert) használunk. Programunk működésének terve a következő.

 © Aradi, Bécsi, Gyenes, Péter, BME

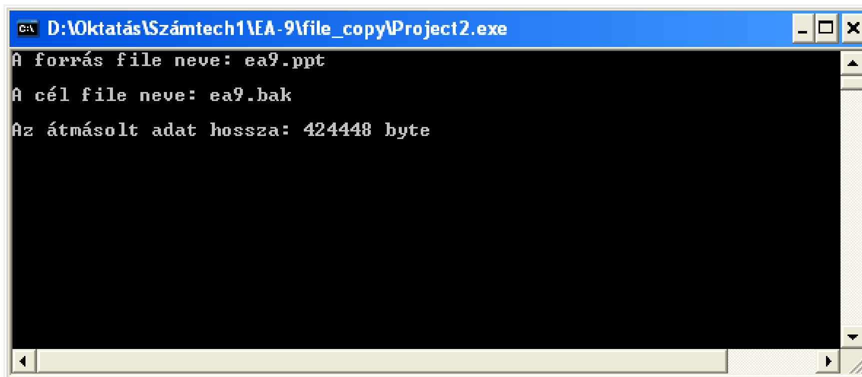


63. ábra: A file másoló program működési terve

A másolandó file hossza lehet nagyon nagy, amely nem fér el a *mem_buf* tömbbe. Ekkor több blokkművelet szükséges. Arra is figyelni kell, hogy az utolsó blokkmásolás a maradék adatot másolja, mivel a másolandó file mérete (az esetek nagy részében) nem lesz a *mem_buf* méretének egészszámú többszöröse. A *mem_buf* méretének megválasztása kompromisszum által adható meg. Ha kicsire választjuk, akkor sok blokkművelet kell, ami lassítja a nagy állományok másolását. Ha nagy méretet választunk, akkor memória pazarlást végzünk. Mi példánkban 16 KB-ot ($4000) választottunk. Programunk utasításlistája a következő.

```
// File gyorsmásolása
// Nagy Zsiga 2010.02.02.

program file_copy;
  var  sour,dest  : file;                  // Típusnélküli file változók
              cl  : longint;               // Számított hossz
               i  : integer;
          membuf  : array [0..$4000] of byte;       // 16 Kbyte memória puffer
             fnam : string [30];

begin
 write ('A forrás file neve: '); readln(fnam);
 assign (sour, fnam);
 reset (sour,1);                         // Megnyitjuk olvasásra rekordhossz=1
 write ('A cél file neve: '); readln(fnam);
 assign (dest, fnam);
 rewrite (dest,1);                       //Megnyitjuk írásra rekordhossz=1
 cl:=0;
 repeat
  blockread (sour, membuf, sizeof(membuf), i); // i byte számláló
  blockwrite (dest ,membuf ,i);
  inc(cl, i);                            // byte számláló növelése
 until eof (sour);                       // Forrás file végéig ismétel
 writeln ('Az átmásolt adat hossza: ', cl, ' byte');
 close(sour);
 close(dest);
 readln;
end.
```

Programunk futási képe az alábbi.



64. ábra: A file másoló program futási képe

Gyakorló feladat

F01. Készítsünk programot, amely felolvas egy ismeretlen típusú file-t, kivágja az abban található szöveges részeket, és a számadatokat egy *adatok.dat* nevű állományba menti.

Ellenőrző kérdések

K01. Mikor használjuk a típusnélküli file-okat ?

K02. Hogy olvasunk típusnélküli file-okat ?

K03. Mire szolgál a memória puffer ?

5.8 Grafika használata a Pascal környezetben

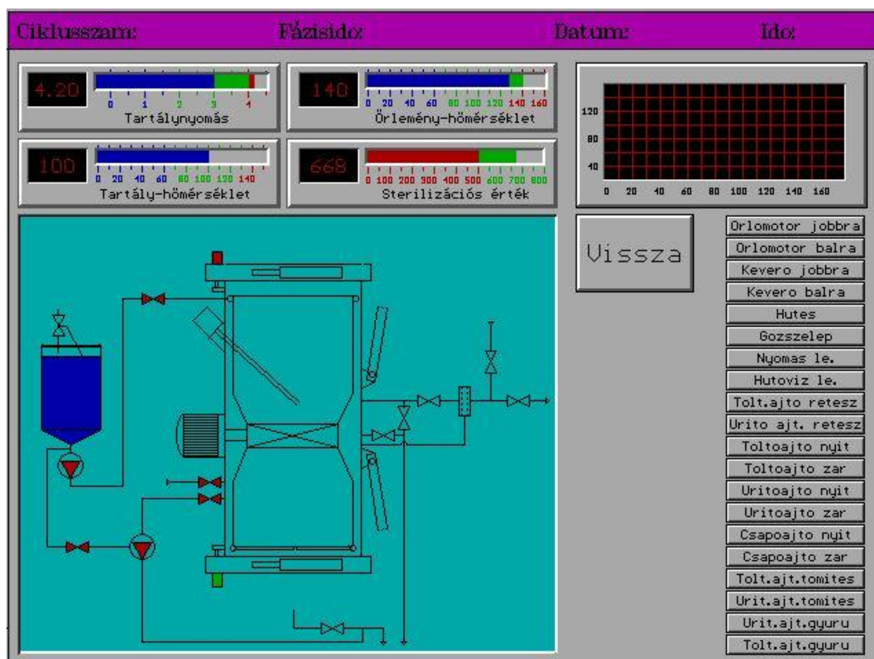
Pascal programmal készíthető látványos grafikai alkalmazás is. Erre azonban a Borland cég Turbo Pascal programjára van szükség. Ez a program DOS (Disk Operating System) környezetben fut. A grafikus rutinok a Pascal *Graph.tpu* unit (egység) részei.

Jelen jegyzetben csak a teljesség kedvéért szólunk röviden erről a lehetőségről. Ennek oka az, hogy magas színvonalú grafikus alkalmazásokra, a Pascal szintaktikára épülő Delphi programot fogjuk használni. Ezzel külön jegyzetünk, a Számítástechnika II. foglalkozik behatóan.

 © Aradi, Bécsi, Gyenes, Péter, BME

Windows operációs rendszer alatt, az általunk használt Delphi fejlesztő környezetben a Console Application csak szimulálja a Pascal programrendszert. A karakteres futási képernyő (25 sor és 80 oszlop) szintén a DOS képernyő szimulált változata. Ez a magyarázata, hogy Console Application alatt nem tudunk grafikai alkalmazást készíteni.

Álljon itt egy Turbo Pascalban készített vezérlő program futási képe, a program közlése nélkül, amely szemlélteti a grafikus alkalmazások lehetőségeit.



65. ábra: Példa a Turbo Pascal grafikus lehetőségeire

5.9 Objektumok

Az objektum olyan struktúra, amely adattípusokat és metódusokat integrál egyetlen struktúra szerkezetbe. A metódus előre definiált eljárás vagy függvény. Az objektum a valóságos fizikai, gazdasági, műszaki jelenségek modellezésének legmagasabb fokú eszköze. Nemcsak a jelenségek, folyamatok mennyiségi tulajdonságait, hanem azok viselkedéseit is egyetlen struktúrába (az objektum) integrálja. Ebben a fejezetben az objektum fajtáit, és készítésének lépéseit ismertetjük.

Az objektum felhasználó által definiált típus, ezért típusdefinícióval mutatjuk be a programnak. Az objektum definiálása a következő.

```
type obj_neve = object
                  <mezők>
                  <metódusok>
                end;
```

Az objektumot adattípus és szegmens integrálásával hozzuk létre.

5.9.1 Egyszerű statikus objektum

Elsőként egy statikus objektumot mutatunk be. Ez a legegyszerűbb objektum fajta. A létrehozandó objektumnak először az elemeit definiáljuk. Hozzunk létre egy dinamikus tömb típust, amely egészszámokat tárol.

```
type t_vec = array of integer;              // dinamikus tömb
```

A dinamikus tömb a deklaráláskor 0 méretű. Valóságos méretét a program futása közben kapja meg.

Másodjára deklaráljunk *fill* néven eljárást, amely feltölt 1 és 50 közötti véletlen egész-számokkal egy vektort.

```
procedure fill (var v : t_vec);
 var i : byte;
 begin
   randomize;
   for i:= Low(v) to High(v) do             // din. tömb index tartománya
     v[i]:=random (50)+1;
 end;
```

A ciklusban a **Low**(v) függvény a v formális paraméter tömb aktuális alsó indexét, míg a **High**(v) a dinamikus tömb felső indexét jelenti.

A véletlen generátort a programban egyetlen alkalommal inicializálni kell a **randomize** standard eljárás hívásával. A véletlen szám generátor egész számokat sorsol, ha paraméterrel aktivizáljuk, a **random**(n) függvény 0 és n-1 között sorsol egy véletlen egész számot. Példánkban a **random** (50) függvény visszatérési értéke 0 és 49 közötti véletlen szám. Ha ehhez adunk 1-et, akkor a kívánt 1..50 intervallumot kapjuk.

Deklaráljunk egy *print* nevű eljárást is, amely listázza egy dinamikus vektor elemeit.

```
procedure print (v : t_vec);
 var i : byte;
 begin
   for i:= Low(v) to High(v) do write(v [i] : 4);
 end;
```

 © Aradi, Bécsi, Gyenes, Péter, BME

Most már minden elem megvan egy objektum definiálásához, ezeket fogjuk integrálni objektum struktúrába:

```
type t_obj = object
            vec : t_vec;            // adat mező
            procedure fill;         // metódus
            procedure print;        // metódus
          end;
```

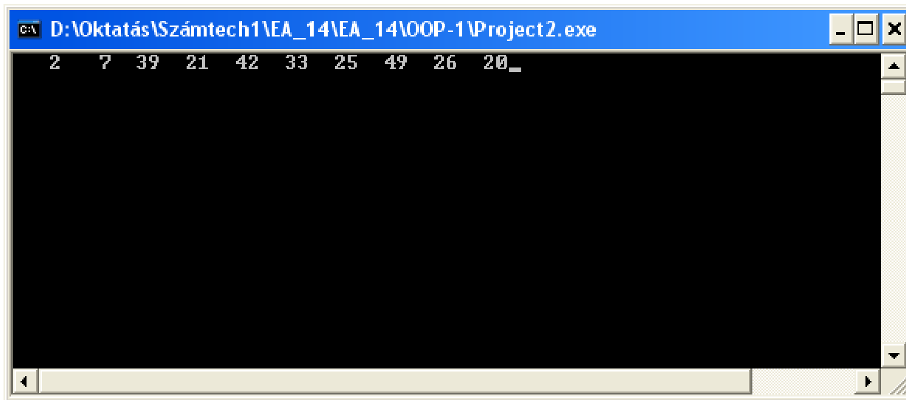
Amint látjuk objektumunkat egy adatmező (dinamikus vektor típus), és két metódus (fill, print) alkotja. Az objektum típus definíció után deklarálhatunk ilyen típusú változót a **var** obj : t_obj; utasítással.

Nézzük miként alkalmazzuk programban a fenti objektumot. Az objektumokkal való munkát OOP (Objectum Oriented Programming) technikának nevezzük. Programunk utasításlistája az alábbi lehet:

```
// Statikus objektum
// Erős Pál 2010.06.02.

program oop_1;

type     t_vec  = array of integer;
         t_obj  = object
                   vec : t_vec;           // adat mező
                   procedure fill;        // metódus prototípus
                   procedure print;               // metódus prototípus
                 end;

 procedure t_obj.fill;                    // Metódus definíció (objektum része)
   var i : byte;
    begin
     randomize;
     for i:= Low(vec) to High(vec) do     // vec az objektum saját adatmezője
         vec[i] := random(50) +1;
   end;

 procedure t_obj.print;                   // Metódus definíció (objektum része)
   var i : byte;
   begin
     for i:= Low(vec) to High(vec) do write(vec[i]:4);
   end;

var  ob : t_obj;                          // objektum típusú változó deklarációja

begin                                     // program törzs
 SetLength(ob.vec,10);                    // az objektum vektorának mérete = 10
 ob.fill;                                 // fill metódus hívása
```

```
    ob.print;                              // print metódus hívása
    readln;
  end.
```

Figyeljük meg, hogy az objektum elemeit a rekord típusnál már használt módon, pont jelöléssel használjuk : *ob.vec* az adatmező, míg *ob.fill* a metódus hívása. Teszteljük első OOP programunkat, amelynek egy futási képe az alábbi lehet.



66. ábra: Az OOP példaprogram képernyőképe

Gyakorlásképpen álljon itt egy másik példa is. Egy takarékbetét főbb jellemzői az alábbiak:

- betét dátuma
- betét összege
- lekötés (látra 1,2,3,6,12 hó)
- kamat %
- látra vagy fenntartásos.

Definiáljunk objektum típust, amely a betétösszeget a kamatjóváírással automatikusan tudja növelni hetente (látra szólónál), illetve megfelelő időnként a többinél.

```
  type t_deposit = object
                ev,ho,nap: word;        // betét dátuma
                sum       : longint;    // betétösszeg
                res       : byte;       // 0 = látra, 1,2,3,6,12
                kam       : single;     // interest = kamat %
                stat      : boolean;    // látra = false, bemutatóra = true
                function  tim:word      // felolvassa a dátumot, számolja az eltelt //napot
```

```
                procedure timmod;     // Módosítja a betét dátumát
                procedure kamat;      // A metódus a kamatot számolja
              end;
```

Az egyes metódusok kidolgozása.

```
function t_deposit.tim : word;            //  Két dátum különbsége napokban
  var  year,mont,day,dow  : word;         // lokális segédváltozók
  begin
   getdate (year,mont,day,dow);           // Az aktuális dátum bekérése
   tim := (year*365  + mont*31 + day ) - (ev * 365 + ho * 31 + nap);
  end;

procedure t_deposit.timmod;               // régi dátum módosítása az aktuálisra
  var  year,mont,day,dow  : word;
  begin
    getdate(year,mont,day,dow);
    ev := year;
    ho := mont;
    nap := day;
  end;
procedure t_deposit.kamat;                // Kamat számító metódus

  procedure make;                         // Lokális eljárás
    begin
      if res = 0 then sum := (1 + kam /100 /54 ) * sum      // Látra
                else  sum := (1+kam / 100 * int(res) /12) * sum;      // Lekötve
      timmod;                             // Objektum másik mtódusa
    end;

 begin                                    // Metódus törzs
    case res of                           // A különböző lekötési idők szerint
      0: if tim >= 7 then make;           // az eltelt napok figyelése
      1: if tim >= 31 then make;
      2: if tim >= 62 then make;
      3: if tim >= 93 then make;
      6: if tim >= 186 then make;
     12: if tim >= 365 then make;
     end;
  end;
```

A fentiek alapján megírhatjuk programunkat valahány betét kezelésére.

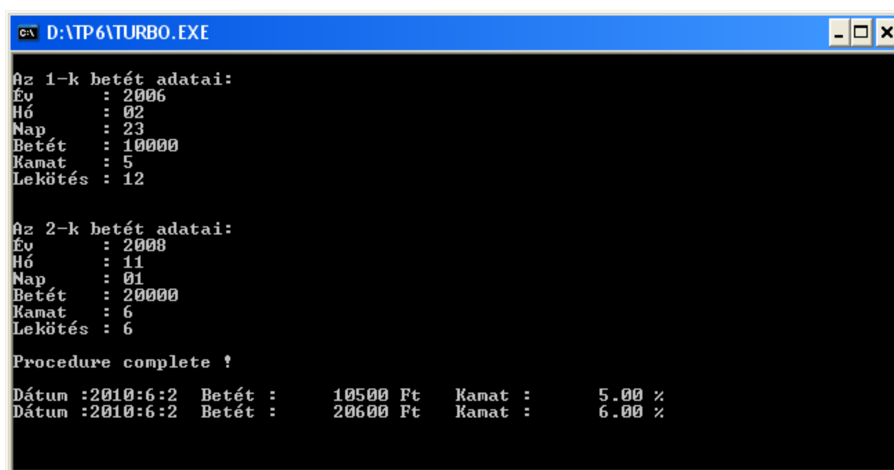
```
// Bankbetétek kezelése OOP_2
// Szép Virág 2009.12.11.

program otp;

const   nr = 2;                           // Példánkban 2 betétet kezelünk
```

```
type    t_deposit = object
                          ev,ho,nap: word;      // betét dátuma
                          sum       : single;   // betétösszeg
                          res       : byte;     // 0 = látra, 1,2,3,6,12
                          kam       : single;   // interest = kamat %
                          stat      : boolean;  // látra = false, bemutatóra = true
                          function  tim:word;   // felolvassa a dátumot, számolja az eltelt //napot
                          procedure timmod;     // Módosítja a betét dátumát
                          procedure kamat;      // A metódus a kamatot számolja
                        end;

   < Metódusok bemásolása >

var  bt : array[1..nr] of t_deposit;            // Betétek tömbje
     i  : byte;

procedure list;                                 // Listázó eljárás (nem része az
                                                // objektumnak)
  var    i: byte;
  begin
   for i:=1 to nr do
     with bt[i] do                              // A with utasítás érvényes az objektumra //is
      begin
       write ('Dátum  ' , ev , ' : ' , ho , ' : ' , nap);
       write(' Betét     : ' , sum:10, ' Ft ' );
       writeln(' Kamat    : ',kam:10:2, ' %' );
     end;
  end;

procedure init;                                 // Betét objektumok feltöltése
  var    i: byte;
  begin
   for i:=1 to nr do
    with bt[i] do
     begin
      writeln;
      writeln ('Az ' ,i, '-k betét adatai: ');
      write(Év         : '); readln(ev);
      write('Hó        : '); readln(ho);
      write('Nap       : '); readln(nap);
      write('Betét     : '); readln(sum);
      write('Kamat   : '); readln(kam);
      write('Lekötés : '); readln(res);
     end;
  end;

begin                                           // Főprogram
  init;
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
    for i:=1 to nr do
     with bt[i] do kamat;
     writeln('Procedure complete !');
     list;
    readln;
end.
```

Programunk futási képe az alábbi.



67. ábra: A betéteket kezelő OOP program képernyőképe

Példánkban a *bt* jelű tömbben jönnek létre az objektum példányok (itt nr = 2). Fontos tudni, hogy objektum példánynak saját adatmezői vannak, míg a metódusokból csak egyetlen példány készül, amelyet az objektum példányok közösen használnak.

Gyakorló feladat

F01. Készítsünk objektum típust, amelynek adatmezője valahány kockadobást (1..6) tárol. Az objektumnak legyen egy metódusa, amely sorsolja a kockadobásokat és eltárolja azokat saját adatvektorában. Másik metódusa a képernyőre listázza a vektort, harmadik pedig file-ba írja. Írjunk programot az objektum felhasználására.

Ellenőrző kérdések

K01. Mi az objektum definíciója ?

K02. Mik a metódusok ?

K03. *N* objektumpéldányt készítve hány adatmező és hány metódus készlet jön létre ?

5.9.2 Dinamikus objektum

Az előző pontban bemutatott statikus objektum példányok a deklaráció végrehajtásakor jönnek létre a memóriában. Ez nagyobb terjedelmű, sok adatmezővel dolgozó objektumok esetén jelentős memóriaterületet foglal el. Ezért célszerű az objektum példányokat a program futása közben létrehozni, és használatuk után az általuk foglalt helyet azonnal felszabadítani. Mivel egyidejűleg egy objektumpéldány aktív, így csak egy létrehozása szükséges. Az elmondottak megoldására alkalmazhatjuk a dinamikus objektumokat. Az eljárás hasonló az 5.6. pontban tárgyalt dinamikus adatok kezeléséhez. Az objektum típus kidolgozása után statikus pointert (vagy pointer tömböt) deklarálunk, amely (vagy amelyek) objektum típusú lesz(nek). Az egyes objektumpéldányokat futás közben a már megismert **new**(optr) eljárással hozzuk létre. Használat után a **dispose**(optr) eljárás fogja a memóriaterületet felszabadítani.

Mintaprogramunk egy dinamikus objektumot definiál. Ennek adatmezőjében egy vektor lesz, amely véletlen generátorral sorsolt bináris számokat tárol. Az objektum 3 metódust is használ, az első feltölti a vektort a véletlen számokkal, a második kiírja a képernyőre, a harmadik egy függvény, amely megszámolja a vektorban az egyesek számát, és nevében visszaadja az összes elemhez viszonyított százalékos arányát.

Először írjuk meg a típus definíciót.

```
type t_optr = ^ obj;                 // Objektumra mutató pointer típus
     obj    = object
                vec: array of 1..2;  // Adatmező : bináris vektor (dinamikus,
                                     //intervallum típusú)
                procedure fill;
                procedure print;
                function calc : single;
              end;
```

Az egyes metódusok kidolgozása.

 © Aradi, Bécsi, Gyenes, Péter, BME

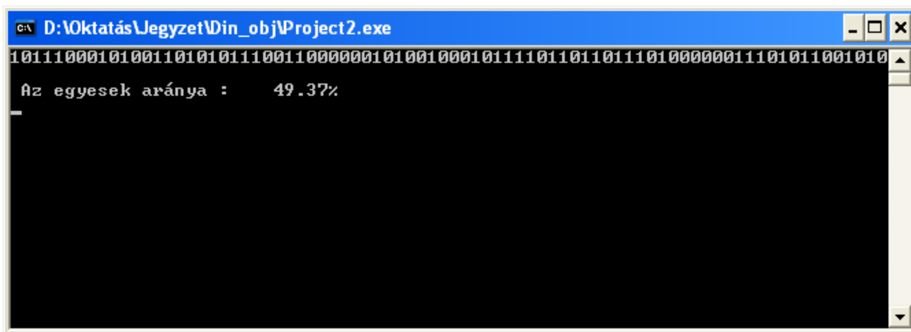
```
procedure obj.fill;
 var i : byte;
 begin
  randomize;
  for i:= Low(vec) to High(vec) do        // vec az objektum saját adatmezője
     vec[i] := random(1);                 // sorsol 0 és 1 értékeket
 end;

procedure obj.print;
 var i : byte;
 begin
   for i:= Low(vec) to High(vec) do write(vec[i] );
   writeln;
 end;

function obj.calc : single;
 var i,db : byte;
 begin
  db := 0;
  for i:= Low(vec) to High(vec) do
    if vec[i] = 1 then inc (db);          // Számolja az egyeseket
  calc := int(db) / int(High(vec) / 100.0;    // % számítása
 end;
```

Ezek után már megírhatjuk teljes programunkat.

```
// Dinamikus objektum OOP_3
// Kiss Éva 2009.10.10.

program bin_sor;

type t_optr = ^ obj;                      // Objektumra mutató pointer típus
     obj    = object
                vec: array of 1..2;       // Adatmező : bináris vektor (dinamikus,
                                          //intervallum típusú)
                procedure fill;
                procedure print;
                function calc : single;
              end;

< a megírt metódusok bemásolása >

var optr : t_optr ;                       // Objektum pointer változó

begin
 new (oprt);                              // Dinamikus objektum példány létrehozása
 setlength (optr ^. vec, 80);             // A vektor hossza legyen 80
 with optr ^ do
  begin
```

```
      fill;                              // A vektor feltöltése
      print;
       // Kiírás 8 helyre 2 tizedessel
      writeln (' Az egyesek aránya : ',calc : 8:2,'%');
    end;
  dispose (optr);                        // Objektumpéldány megszüntetése
 readln;
end.
```

Programunk futási képe az alábbi.



68. ábra: A bináris vektort kezelő OOP program képernyőképe

Vizsgáljuk meg azt az esetet is, amikor a dinamikus objektum adatmezői is dinamikus adatok. Ilyenkor ezen adatmezők konténereit a dinamikus objektumpéldány létrehozásával együtt készítjük el. Erre a Pascal a **constructor** nevű standard eljárást kínálja. Ez a szegmens inicializálja a dinamikus változókat, ezért célszerűen *init* nevet kap. Hasonlóképpen a dinamikus objektum megszüntetése előtt a dinamikus változókat is fel kell szabadítani, erre a célra a **destructor** eljárás (*done*) való. A dinamikus objektumot a konstruktorral együtt a kiterjesztett kétparaméteres **new**(optr,init) hozza létre, míg megszüntetését a kétparaméteres **dispose** (optr,done) végzi el.

Példánkban egy dinamikus objektum dinamikus adatmezője legyen egy téglalap két oldalát megadó számpáros. A *ter* és *ker* függvény típusú metódusok a téglalap területét és kerületét számítják ki. A típusdefiníciók az alábbiak lehetnek.

```
type t_optr  = ^ rect;                   // Objektumra mutató pointer típus
       rect  = object
                 a, b : ^ single;        // Dinamikus adatok pointer változói
                 constructor init (a0, b0 : single) ; // Létrehozza a konténereket
                 function ter : single;
                 function ker : single;
                 destructor done;
```

```
                    end;
```

Elsőként a konstruktort készítjük el.

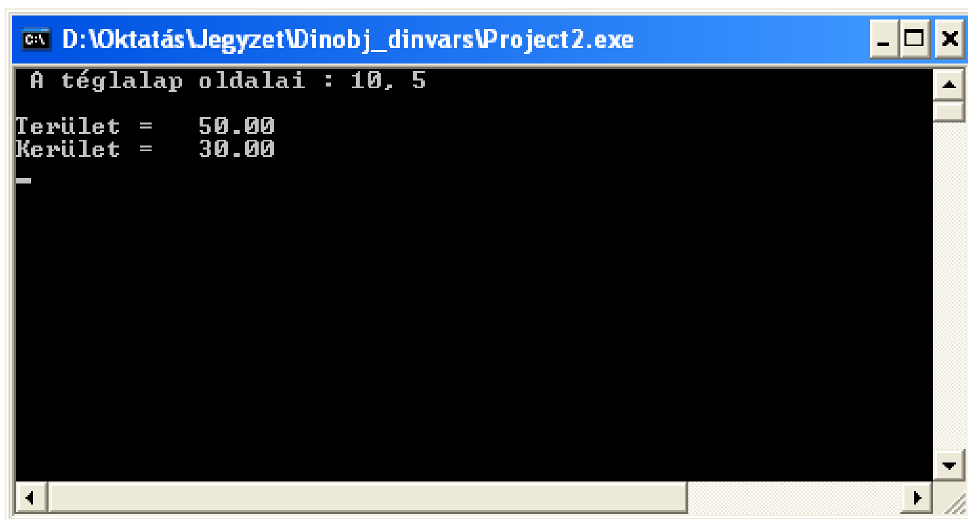
```
constructor rect.init (a0, b0 : single);
 begin
   new (a);                    // a konténere a heap memóriában
   new (b);                    // b konténere a heap memóriában
   a^ := a0;                   // felveszi a formális par.-ben átvett értéket
   b^ := b0;
 end;

function rect.ter : single;
 begin
  return (a^ * b^);            // visszatérési érték a terület
 end;

function rect.ker : single;
 begin
  return (2 * (a^ + b^ ) );    // visszatérési érték a kerület
 end;

destructor rect.done;
 begin
   dispose (a);                // Dinamikus konténerek felszabadítása
   dispose (b);
 end;
```

Ezek után programunk utasításlistája az alábbi lehet.

```
// Dinamikus objektum OOP_4
// Nagy Zoli 2009.10.14.

program din_rectangle;

type t_optr  = ^ rect;                      // Objektumra mutató pointer típus
        rect  = object
        a, b : ^ single;                    // Dinamikus adatok pointer változói
        constructor init (a0, b0 : single) ; // Létrehozza a konténereket
        function ter : single;
        function ker : single;
        destructor done;
       end;

 < a kidolgozott metódusok bemásolása >

var ptro : t_optr;                          // Objektum pointere

begin
```

```
    new (ptro, init (5, 10) );              // Objektum és változók létrehozása + értékadás
    writeln( 'Terület = ' , ptro ^ .ter : 7: 2);   // Terület kiírása 7 helyre 2 tizedessel
    writeln( 'Kerület = ' , ptro ^ .ker : 7: 2); // Kerület kiírása 7 helyre 2 tizedessel
    dispose (ptro, done);                    // Változók és objektum megszüntetése
    readln;
  end.
```

Programunk futási képe az alábbi.



69. ábra: A dinamikus objektumokat kezelő program képernyőképe

Gyakorló feladat

F01. Módosítsuk az előző fejezetben kidolgozott OOP_1 programot, hogy az abban szereplő statikus objektum típust váltsuk fel azonos feladatot ellátó dinamikus objektummal.

Ellenőrző kérdések

K01. Mi a dinamikus objektum használatának előnye?

K02. Miként hozunk létre és szüntetünk meg dinamikus objektum példányokat?

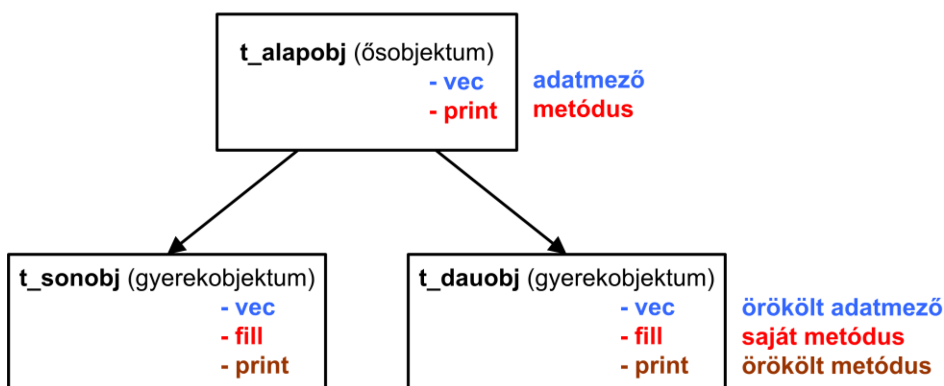
5.9.3 Objektumok öröklési tulajdonsága

Az objektum orientált programozás előnye valójában az öröklési tulajdonság (inheritance) kihasználásával mutatkozik meg. Ez azt jelenti, hogy készíthetünk egy meglévő objektumra épülve újabb objektumokat, amelyek saját adataik és

metódusaik mellett használni tudják (öröklik) a már elkészített objektum adatmezői és metódusait is. Ezzel a technikával az objektumok hierarchikus rendbe szervezhetők. A Pascal egy öröklési szintet enged a túl bonyolult struktúra elkerülése végett.

Az öröklés szempontjából beszélhetünk alap (vagy ős-) objektumról, illetve leszármazott (vagy gyerek-) objektumról. Az örökölt adatmezőkről másolat készül a származtatott objektum példányban, de az örökölt metódusról nem (kivétel a re-entrant = ráindítható, vagy többszörösen beléphető tulajdonsággal felruházott metódus). Az öröklési tulajdonság kihasználását példán mutatjuk be.

Programunk feltölt 2 különböző vektort véletlen számokkal, majd kiíratja azokat. Az első vektort 0..49, a másodikat 50..100-ig terjedő számokkal tölti ki. A kiíratásnál kihasználjuk az objektum öröklődési tulajdonságait. Először tervezzük meg az objektum struktúrát.



70. ábra: Az objektum struktúra terve

Az egyes mezőket színekkel kódoltuk a jobb megértés miatt.

Először készítünk egy alkalmas vektor típust.

```
type t_vec = array of word;
```

Másodszor elkészítjük az alap- vagy ős objektum típust.

```
t_alapobj  = object                        //  ős objektum
               vec   :  t_vec;
               procedure print;
             end;
```

Harmadjára a leszármaztatott objektum típusokat definiáljuk.

```
t_sonobj  = object (t_alapobj)              // leszármazott objektum
              procedure fill;               // saját metódus
             end;

t_dauobj  = object (t_alapobj)              //  leszármazott objektum
              procedure fill;               // saját metódus
             end;

procedure t_alapobj.print;                  // Ősobjektum metódus
 var i : byte;
 begin
   for i:= Low(vec) to High(vec) do write(vec[i]:4);
    writeln;
 end;

procedure t_sonobj.fill;                    // Metódus
 var i : byte;
   begin
    for i:= Low(vec) to High(vec) do        // vec örökölt adatmező
      vec[i]:= random(50);                  // 0..49
   end;

procedure t_dauobj.fill;                    // Metódus
 var i : byte;
 begin
   for i:= Low(vec) to High(vec) do         // vec örökölt adatmező
     vec[i]:= random(51) + 50;              // 50..100
 end;
```

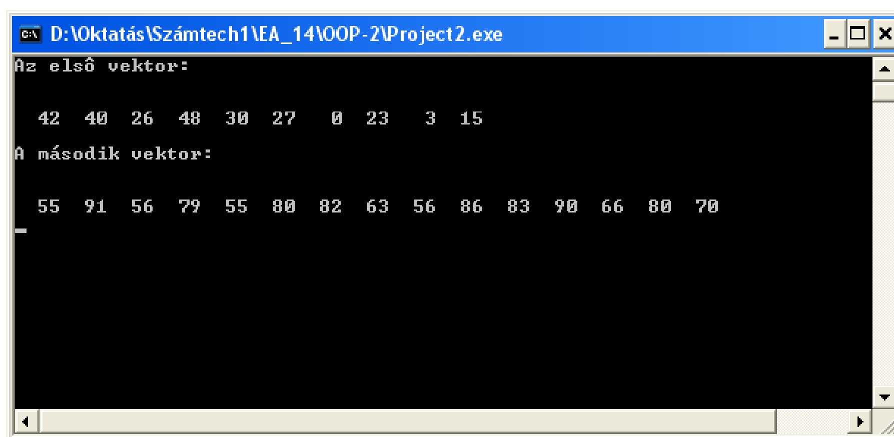
Ezen definíciók és deklarációk után megírhatjuk a programot.

```
// Objektum öröklés OOP_5
// Ék Albert  2010.10.14.
program oop_5;

type  t_vec        = array of word;

       t_alapobj  = object                  // ős objektum
                     vec   :  t_vec;
                     procedure print;
                    end;

   t_sonobj = object (t_alapobj)            // leszármazott objektum
                procedure fill;             // saját metódus
               end;

  t_dauobj  = object (t_alapobj)            // leszármazott objektum
                procedure fill;             // saját metódus
               end;
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
 < a metódusok bemásolása >
var fiuobj    : t_sonobj;                  // objektum deklaráció
       lanyobj : t_dauobj;

begin
  randomize;
  SetLength( fiuobj. vec,10 );             // Vektor mérete = 10
  fiuobj. fill;                            // Saját metódus
  writeln('Az első vektor:'); writeln;
  fiuobj.print;                            // Örökölt metódus

  with lanyobj do                          // with érvényes az objektumnál
    begin
      SetLength (vec,15);                  // Vektor mérete = 15
      fill;                                // Saját metódus
      writeln('A második vektor:'); writeln;
      print;                               // Örökölt metódus
    end;
  readln;
end.
```

Programunkban csak két objektumpéldányt deklaráltunk, az alapobjektum a vektor mezőt és a *print* metódust adta. A program egy futási képe az alábbi.



71. ábra: Az öröklést bemutató OOP program képernyőképe

Gyakorló feladat

F01. Írjunk OOP programot. Az alapobjektum *make* metódusa véletlen generátorral létrehoz n darab kockadobást imitáló számot és *store* nevű mezőjében tárolja. A *son* leszármazott objektum örökli ezeket a tulajdonságokat, majd saját *statistic* metódusával megszámolja az egyes dobá-

sok %-os előfordulását (összesen 6 adat). A *daughter* leszármazott objektum a dobásokat a *cube.dat* nevű file-ba írja.

Ellenőrző kérdések

K01. Mi az objektum inheritance tulajdonsága ?

K02. Hogyan jelöljük a leszármazott objektumot ?

K03. Hány generációs öröklést enged a Pascal ?

5.10 A Pascal unit

A szegmensek tárgyalásakor már említést tettünk a unitról, mint a Pascal külső szegmenséről (5.4. pont). A unit a Pascal önálló szegmense, vagy más szóval külső modulja, amely lefordított (tárgykódú) formában egészít ki valamely programot. A unit lehetővé teszi nagyméretű alkalmazások készítését PASCAL környezetben. Alapvetően kétféle unit van: standard unit, ami a nyelv részeként előre definiálva van, és a programozó által készített unit.

5.10.1 Standard unit

A standard unitok gyűjteménye a feltelepített fejlesztő rendszerben lefordított változatban a ***bgi*** könyvtárban rendelkezésre áll. A Turbo Pascal ezeket ***tpu*** (Turbo Pascal Unit) kiterjesztéssel azonosítja. A standard unitok használatához ismerni kell a benne lévő eljárások és függvények neveit, feladatukat, valamint paraméterezésük módját. Az utasításlistát a Pascal nem közli.

Delphi fejlesztői környezetben a standard unitok kiterjesztése ***dcu*** (Delphi Compiled Unit) és a ***lib*** könyvtárban találjuk meg ezeket forráskódjukkal (*pas* kiterjesztés) együtt. Ezek közül álljon itt néhány.

SysUtils amely a Consol Application alaprutinjait tartalmazza. Ez szükséges a Pascal szimulációhoz Delphi környezetben. Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, stb a Delphi alap unitjai, amelyeket minden alkalmazás indításakor felkínál. Ezekkel részletesen a Számítástechnika II. c. jegyzetben foglalkozunk.

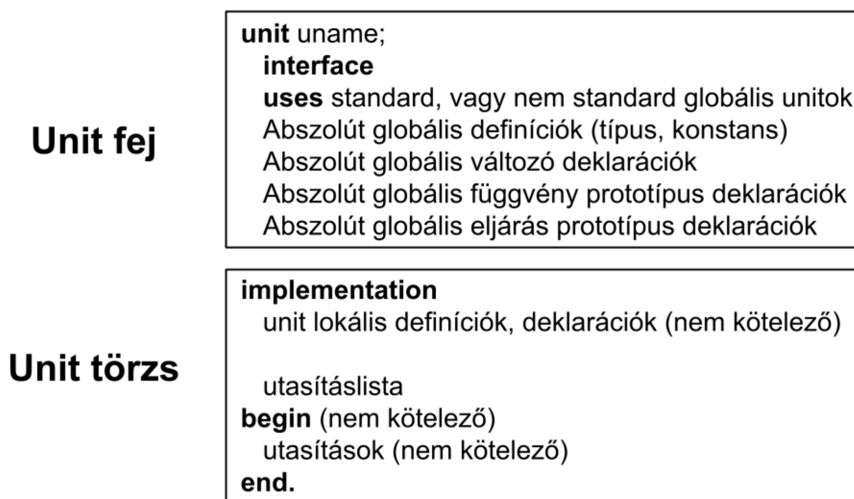
5.10.2 Unitok definiálása és használata

Számunkra most a saját unitunk készítése fontos. Ehhez elsőként a unit szerkezeti felépítését kell megismerni. Amint a többi szegmens (függvény, eljárás) és maga a program is, a unit is két fő részből áll: fej és törzs. A unit fejsorában a

 © Aradi, Bécsi, Gyenes, Péter, BME

unit kulcsszó és a név (szimbolikus név) áll. Az **interface** kulcsszó után, de még a fejrészben az abszolút globális deklarációk, és definíciókat találjuk. Az itt megadott mennyiségek nem csak jelen unitban, hanem más programokból is elérhetők, amelyek ezt a unitot használják. A unit törzsben az **implementation** kulcsszó után, de a **begin** előtt a unit lokális mennyiségeket adjuk meg. Ezek csak jelen unit belsejében érvényesek. A **begin** és az ezt követő utasítások nem kötelezőek. Az itt megírt programrészek a unit meghívásakor azonnal végrehajtásra kerülnek, s főként a unit változóinak inicializálására használatosak. A unit önmagában nem futtatható, kipróbálására segédprogramot készítünk. A unit létrehozásának lépései Delphi környezetben:

- Delphi: File/New/Unit kiválasztás
- Editálás (a unit megírása)
- Mentés lemezre, *azonos névvel* mint a unit név
- Fordítás F9 (ellenőrizzük a könyvtárat → ***nev.dcu***)
- File/New/Consol Application kiválasztása
- A főprogram megírása
- Uses unit ***nev*** beillesztése
- Főprogram futtatása.



72. ábra: A unit szerkezete

Első példánkban készítünk *small* néven unitot, amelynek eljárása egy stringet kisbetűsre konvertál. Unitunk fejrésze az alábbi lesz.

Unit fej

```
unit small;
  interface
  type t_str = string;                  // abszolút globális típus
  var str : t_str;                      // abszolút globális változó
  procedure lower (var s : t_str);      // abszolút globális eljárás
```

Törzsébe az alábbiakat írjuk.

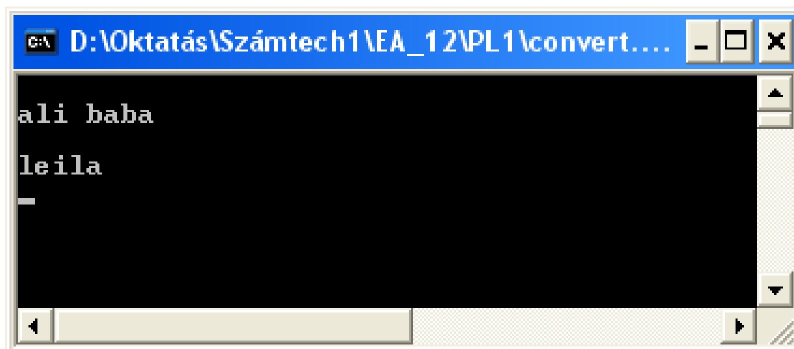
Unit törzs

```
implementation
  procedure lower;            // paraméterlista lehagyható
    var i : byte;
    begin
      for i:=1 to length(s) do
        if s[i] in ['A'..'Z'] then s[i]:=chr(ord(s[i])) or $20);
    end;
end.
```

Most elmentjük *small.pas* néven, és lefordítjuk a unitot. Ellenőrizzük, hogy könyvtárunkban megjelent a *small.dcu* file. Jelen fázisban csak szintaktikai ellenőrzés történik. Ezek után írjunk programot, amely két különböző idézetre alkalmazza a megírt, és unitban elhelyezett eljárást. A program utasításlistája a következő.

```
// Unit alkalmazása
// Víg Jenő 2009.04.23.

program convert;

uses small in ’small.pas’;          // A unit alkalmazása

var msg : t_str;                    // A unit abszolút globális típusának hasz//nálata

begin
  str := 'Ali BABA';                // Abszolút globális str változó használata
  lower (str);                      // A unitban megírt eljárás hívása
  writeln (str);                    // → ali baba
  msg := 'LeiLA';                   // A program globális msg változójának
                                    //használata
  lower(msg);
  writeln(msg);                     // → leila
  readln;
end.
```

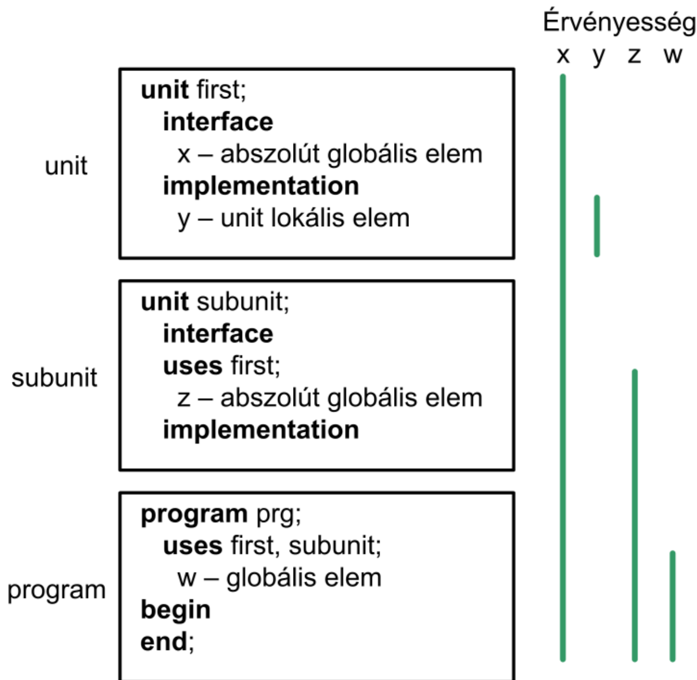
Programunk futási képe az alábbi.



73. ábra: A unitot használó példaprogram képernyőképe

5.10.3 A változók, modulok hatásköre

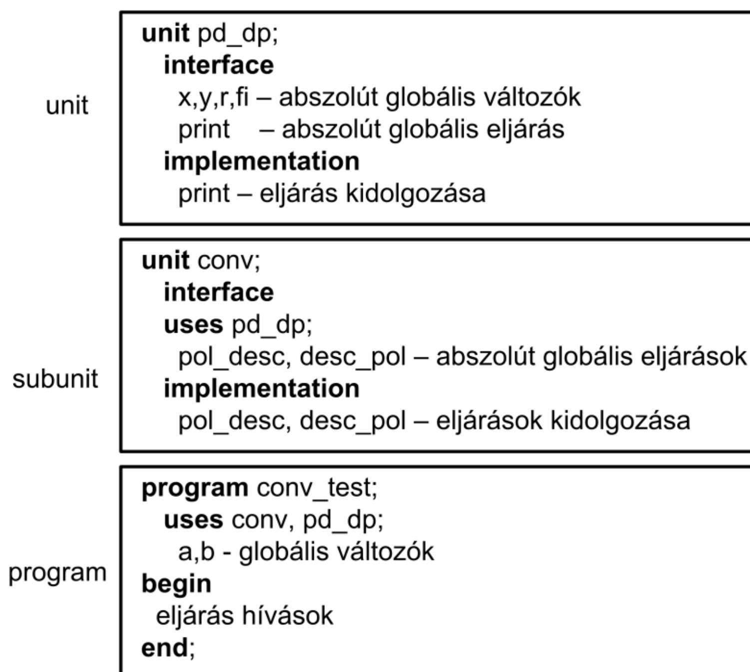
Mielőtt második mintapéldánkra rátérnénk ismertetjük, hogy a unitokat is használó elemek miként használhatók fel. Másszóval a láthatóságukat (visibility) vizsgáljuk. Ezt leginkább egy olyan struktúrán szemléltethetjük, amely két unitot is alkalmaz. Későbbi Delphi alkalmazásainknál ez gyakori eset lesz.



74. ábra: Az egyes program elemek érvényessége

A legelső, *first* nevű unitunkban deklarált *x* változó mindenütt látható, míg *y* csak a *first* belsejében. A második, subunit *z* változója a *first*-ben nem, máshol látható. A *prg* program w változója csak saját törzsében látható. Az elmondottak a konstansokra, eljárásokra és függvényekre is érvényesek.

Ezek után készítsünk programot, amely szögek átváltását (Descartes/Polár és Polár/Descartes) valósítja meg. A konvertáló eljárásokat unitban helyezzük el. Célszerű megtervezni programunk szerkezeti felépítését. Ezt az alábbi ábra szemlélteti.



75. ábra: A unitok és a program terve

Első unitunk utasítás listája a következő.

```
// Unit_dp_pd
// Vas Ádám 2008.11.01.

unit pd_dp;

var  x,y,r,fi: single;
procedure print(a,b : single; pddp : boolean);     // kiíratás

interface

procedure print;                               // Itt nem kell a paraméter listát ismételni
 begin
  if pddp then writeln ('x = ', a:8:3,'  y = ',b:8:3);          //  Descartes alak
             else  writeln ('r = ',a:8:3,  ' fi = ',b:8:3);          // Polar alak
 end;

 end.
```

Mentsük (*pd_dp.pa*), és fordítsuk le (*pd_dp.dcu*) unitunkat. Második, konvertáló unitunk utasítás listája a következő lesz.

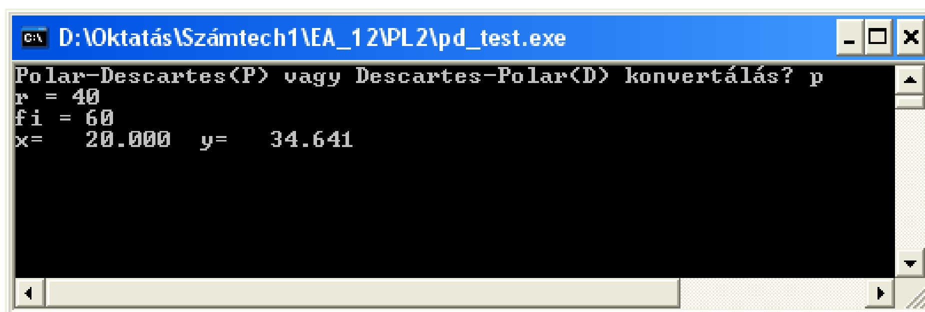
```
// Unit_conv
// Vas Ádám 2008.11.01.

unit conv;

interface
uses pd_dp;                                   // Fő Unit
procedure pol_desc(var a,b : single);         // Polár-Descartes
procedure desc_pol(var a,b : single);         // Descartes-Polár

implementation
  var t: single;                              // Unit lokális változó

 procedure pol_desc;
  begin
    t:=a*cos(b);                              // x koord.
    b:=a*sin(b);                              // y koord.
    a:=t;
  end;

procedure desc_pol;
  begin
    t:=sqrt(a*a + b*b);                       // absz. érték
    if a <> 0 then b:=arctan(b/a)*180/pi
              else b:=0;                      // szög
    a:=t;
  end;

end.                                          // Unit vége
```

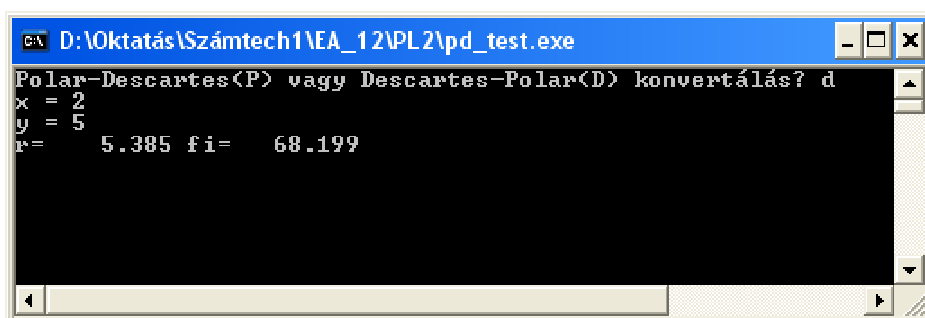
Ismét mentés (*conv.pas*)és fordítás (*conv.dcu*)következik, ezután a program.

```
// Unit_próba
// Vas Ádám 2008.11.01.

program pd_test;

uses pd_dp, conv;                             // A unitok alkalmazása

var   ch  : char;                             // Program globális változók
      a,b : single;
begin
  repeat
   write('Polar-Descartes(P) vagy Descartes-Polar(D) konvertálás? ');
   readln(ch);
   ch:=upcase(ch);                            // Válasz nagybetűsre váltása
  until ch in [‘P’,’D’];                      // Csak P vagy D lehet
  if ch = ‘P’ then
   begin                                      // Polár-Descartes
```

```
      write ('r = '); readln(r);             // abszolut érték bekérése
      write ('fi = '); readln(fi);           // szög bekérése fokban
      fi := fi*pi / 180;                     // fokból radián
      pol_desc (r,fi);                       // eljárás hívása r és fi paraméterekkel
      print (r,fi, true);                    // eredmény kiírása Descartes alakban
    end
   else
    begin
      write (‘x = '); readln (a);            // x koordináta bekérése a globális //változóba
      write (‘y = '); readln (b);            // y koordináta bekérése b globális //változóba
      desc_pol (a,b);                        // eljárás hívása a és b paraméterekkel
      print (a,b, false);                    // eredmény kiírása Polár alakban
    end;
    readln;
  end.
```

Programunk futási képe mindkét esetre az alábbiakban látható.





76. ábra: A Descartes<->polár konvertáló program képernyőképe

Gyakorló feladat

F01. Készítsünk unitot, amelyben a *tax* nevű függvény egy árukészlet eladása után fizetendő ÁFA összeget (25%) számítja. A program két különböző áru készletet olvas be 2 rekord típusú file-ból, majd a unit függvényét aktivizálja, és képernyőre írja a fizetendő ÁFA összegeket.

Ellenőrző kérdések

K01. Mik a unit fő részei?

K02. Mit értünk abszolút globális változó alatt ?

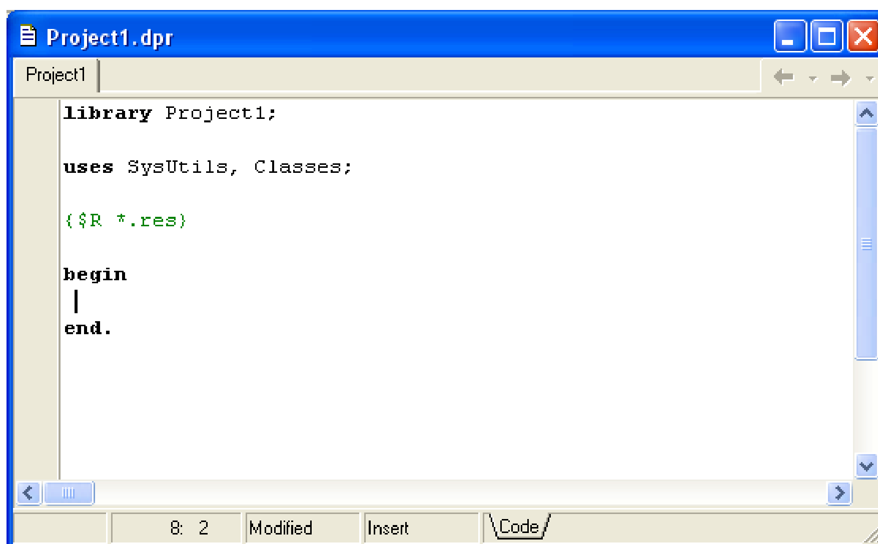
5.11 Külső modulok (DLL)

A unit a szegmentáció egyik eszköze. Ez a külső szegmens Pascal környezetben, Pascal nyelven készül. Használata Pascal, vagy Delphi alkalmazásokban lehetséges. Számos alkalmazás megkívánhatja, hogy olyan külső modulokat is használjunk, amelyek nem Pascal nyelven íródtak. Erre is van lehetőség, ha ezek a modulok DLL (Dinamic Link Library) formában állnak rendelkezésre. Ezeknél a moduloknál – mivel bármilyen programnyelvű alkalmazásba beilleszthetők – már nem ismerjük a forrásnyelvű alakot, de erre nincs is szükség. Ezért nevezzük ezeket dinamikusan becsatolható moduloknak. Használatukhoz ismernünk kell a benne lévő eljárások hívási alakját és paramétereit. A DLL a Microsoft fejlesztése, használata Windows és OS/2 operációs rendszerek alatt lehetséges.

5.11.1 DLL létrehozása

Az univerzálisan felhasználható DLL modult Pascal nyelven is készíthetünk. Ennek módját mutatjuk be.

A Delphi környezetben a File/New/DLL Wizard kiválasztása után a fejlesztő az alábbi ablakot nyitja meg.



77. ábra: A Delphi DLL projekt ablaka

A DLL modul szerkesztési neve library lesz. Ez az elnevezés arra utal, hogy a nyilvános könyvtár része készül. A fejrész itt is a deklarációkat tartalmazza. Készítsünk DLL könyvtári modult, amely 2 egészszám legnagyobb közös osztóját, és legkisebb közös többszörösét számító függvényeket tartalmazza. Az utasításlista a következő lesz.

```
// DLL modul
// Kő Pál  2010.05.22.

library dll_mat;

uses SysUtils, Math, Classes;          // Standard unitok használata

{$R *.res}                              // Fordító direktíva

// Két szám legnagyobb közös osztója
function lko( a,b: integer): integer; stdcall;
  var n: integer;
  begin
   n := min (a,b);                      // A kisebb számot adja
   while (a mod n > 0) or (b mod n > 0) do dec (n);
   result := n;                         // Visszatérési érték
  end;

// Két szám legkisebb közös többszöröse
function lkt( a,b: integer): integer; stdcall;
  var n: integer;
```

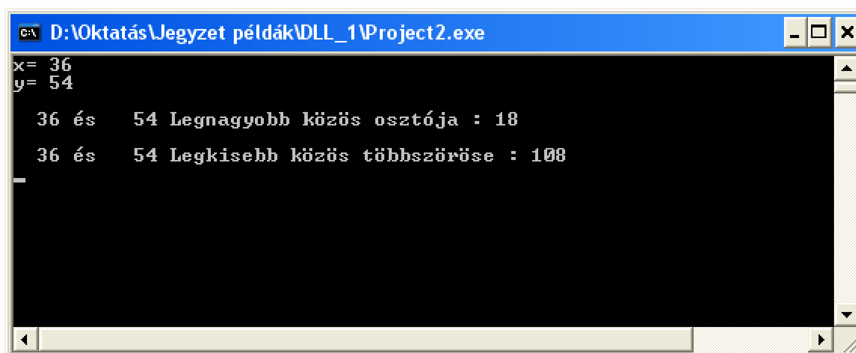
```
 begin
  n := max (a,b);                         // A nagyobb számot adja
  while (n mod a > 0) or (n mod b > 0) do inc (n);
  result := n;
 end;

exports  lko, lkt;                        // Exportálandó függvények

end.
```

Mentsük el programunkat, majd a Compile/ Build funkció kiválasztásával hozzuk létre és ellenőrizzük le a ***dll_mat.dll*** könyvtári modult. Ezután már megírhatjuk programunkat, amely felhasználja az elkészített modult.

```
// DLL modul
// Kő Pál  2010.05.22.

program dll_alk;

function lko (a,b: integer):integer; stdcall; external 'dll_mat.dll';

function lkt (a,b: integer):integer; stdcall; external 'dll_mat.dll';

var x,y: integer;

begin
  writeln ('x = '); readln (x);              // Egyik szám bekérése
  writeln ('y = '); readln (y);              // Egyik szám bekérése
  writeln ('Legnagyobb közös osztó       : ', lko (x,y) );
  writeln ('Legkisebb közös többszörös : ', lkt (x,y) );
  readln;
end.
```

Láthatjuk, hogy csak néhány kulcsszó szükséges a külső modulban lévő, már lefordított függvény használatához. A fordítónak ismernie kell a DLL modul elérési útvonalát is, ha az nem az aktuális könyvtárban van. Programunk futási képe az alábbi.

 © Aradi, Bécsi, Gyenes, Péter, BME



78. ábra: A DLL-t használó példaprogram képernyőképe

Gyakorló feladat

F01. Készítsünk DLL könyvtári modult, amely két komplex szám összegét, különbségét és szorzatát számítja. Írjunk programot, amely aktivizálja a megírt eljárásokat.

Ellenőrző kérdések

K01. Mi a DLL használatának előnye?

K02. Milyen kulcsszó használatos a DLL rutinok készítésekor?

K03. Hogyan hivatkozunk a programban a külső (DLL) rutinokra.

6 Feladatgyűjtemény

Jelen fejezet célja, hogy kidolgozott példákkal segítsük a tananyag megértését. Az előadásokon bemutatott anyagrészeket a hallgatók a laborban számítógépen gyakorolhatják Sok éves tapasztalatunk szerint a rendelkezésre álló idő (heti egy alkalommal) nem elegendő az önálló feladatmegoldó képesség kialakításához. Különösen azok számára okoz nehézséget a tantárgy követelményeinek teljesítése, akik előtanulmányaik során nem találkoztak program nyelvekkel. Példatárunk szerkezeti felépítése igazodik az elméletet tárgyaló fejezetek struktúrájához.

6.1 Példák az algoritmus szerkesztésre

A folyamatábra szerkesztése nehézséget okoz sok hallgatónak. Szeretnénk felhívni a figyelmet ennek a feladattípusnak a fontosságára. Mielőtt nekilátnánk a program megírásának, minden esetben célszerű a feladat megoldásának ilyen módon való átgondolása. A folyamatábra nagyobb, összetett feladatok megoldásánál nélkülözhetetlen. Nagy előnye, hogy áttekinthető, grafikus formában írja le a megoldást. A programok dokumentálásának és ismertetésének is fontos része. A folyamatábra készítéséhez használjuk fel az ingyenesen letölthető Diagram Designer programot a http://logicnet.dk/DiagramDesigner webhelyről.

6.1.1 Elágazásos algoritmusok

P1_1 Készítsünk algoritmikus programot másodfokú egyenlet megoldására. Az egyenlet általános alakja:

$$a * x^2 + b * x + c = 0$$

Az egyenletben a, b, c együtthatók a program bemenő adatai. Meg kell vizsgálni a lehetséges eseteket.

- Ha ***a*** =0, akkor az egyenlet b * x + c = 0 alakú lesz. Ez esetben a megoldás: x = -c / b

- Ha ***a*** =0 és ***b*** =0, akkor az egyenlet c = 0 lesz.

- Ha ***a*** =0 és ***b*** =0 és ***c*** =0 , akkor 0 = 0 azonosság adódik.

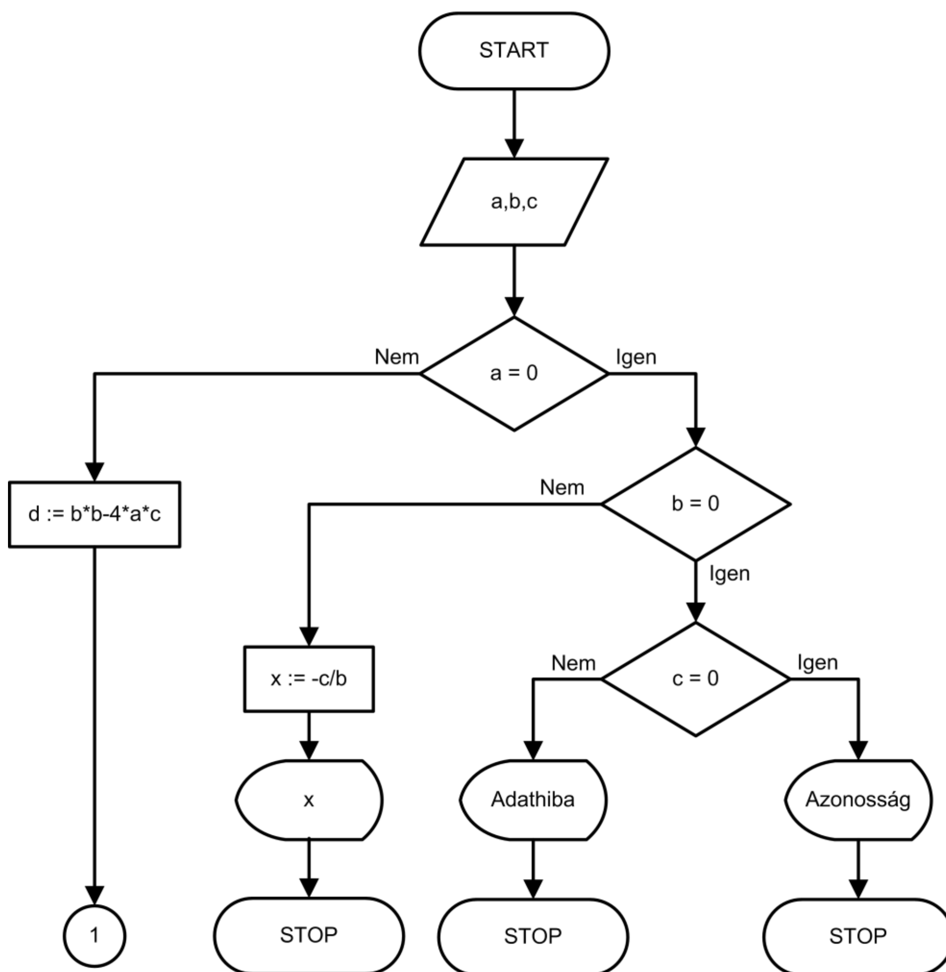
- Ha ***a*** =0 és ***b*** =0 és ***c*** <>0 , akkor c = 0 ellentmondás, azaz adathiba lép fel.

- Ha ***a*** <>0, akkor a megoldó képlet alkalmazható:

$$X_{1,2} = -b \pm SQRT (b*b - 4*a*c) / 2 / a .$$

- Ebben az esetben vizsgálni kell a diszkrimináns (d) értékét: Ha $d \geq 0$, akkor a megoldás valós értékű, míg ha $d < 0$, akkor komplex gyökök adódnak.

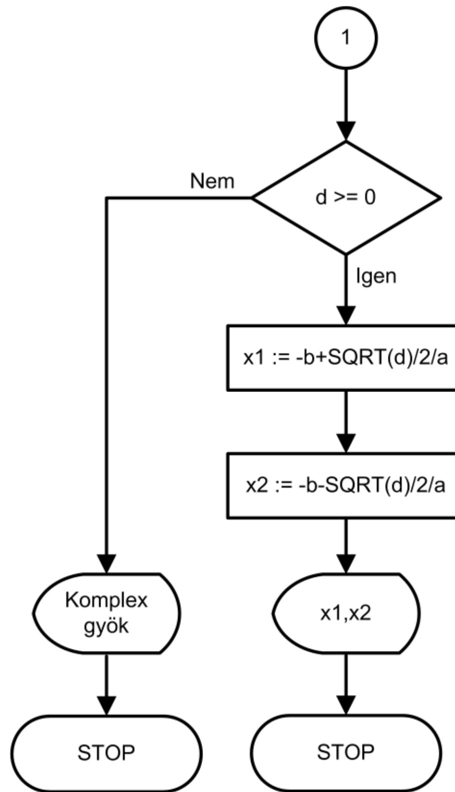
Ezen megfontolások alapján készítjük az algoritmust.



79. ábra: A másodfokú egyenlet megoldásának első része

Az ábra sok esetben nem fér el egy lapon. Ekkor lap kapcsolót (page connector) használunk. Jelen példánkban ez a körrel és számmal (1) jelölt szimbólum. Az ábra folytatása a következő lapon ugyanezzel a szimbólummal van jelölve. Fi-

gyeljük meg, hogy a négyzetre emelést egyszerű szorzásként realizáltuk. Feladatunk programmal való megoldását a 6.6. pontban találjuk meg.



80. ábra: A másodfokú egyenlet megoldásának második része

Példánkban a komplex gyökök kiszámításával nem foglalkoztunk.

P1_2 Második példánkban az alábbi kifejezés kiszámítására készítünk folyamatábrát:

$$y = \frac{\sqrt{\ln(b - 3 * c) - a}}{(9 - a^2)(4 - b^2)}$$

A képletben az alábbiakra kell tekintettel lenni:

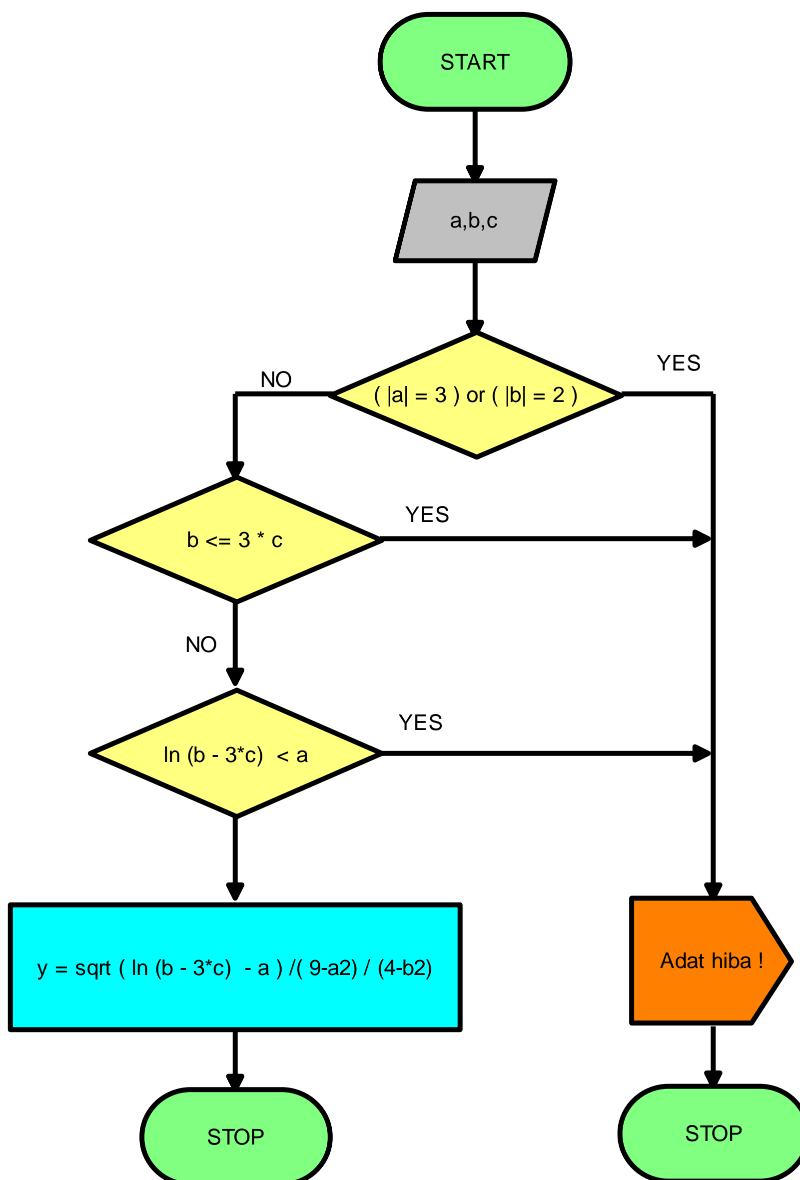
- A nevező ne legyen 0 értékű, azaz $|a| \neq 3$, és $|b| \neq 2$.

- A logaritmus argumentuma nem lehet ≤ 0, azaz $b > 3*c$ kell legyen

 © Aradi, Bécsi, Gyenes, Péter, BME

- A négyzetgyök alatt nem állhat negatív szám, azaz ln (b - 3*c) ≥ 0 kell legyen.

Ezek figyelembevételével készülhet az algoritmikus program.



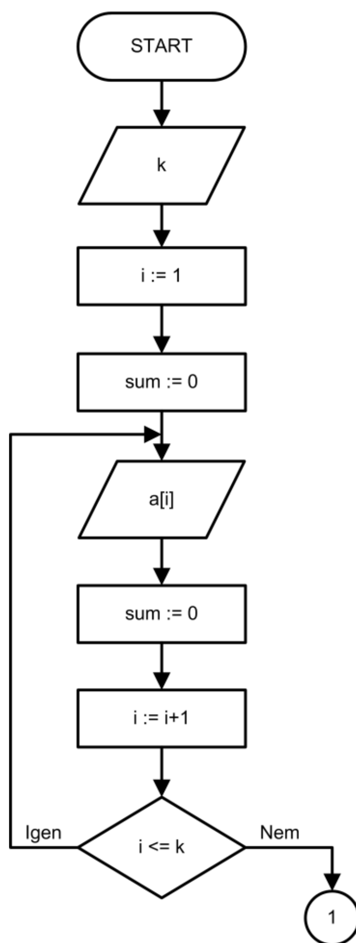
81. ábra: Összetett matematikai kifejezés vizsgálata

6.1.2 Egyszeres ciklust használó feladatok

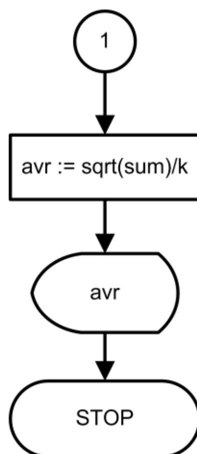
P2_1. Készítsünk folyamatábra programot, amely egy k elemű vektor négyzetes átlagát számítja ki.

$$avr = \frac{\sqrt{\sum_{i=1}^{k} a_i^2}}{k}$$

A feladat megoldása során rendre beolvassuk az elemeket. és egy változóban mindjárt összeadjuk a beolvasott értékek négyzetét. Végül a kiíratáskor négyzetgyököt vonunk az összegből és elosztjuk az elemek számával.



 © Aradi, Bécsi, Gyenes, Péter, BME



82. ábra: A négyzetes átlag megoldása

P2_2. Készítsünk folyamatábra programot az

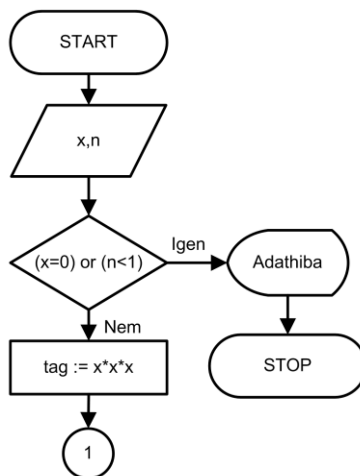
$$y = x^3 - x + 1/x - 1/x^3 + - \dots$$

n tagú összeg számítására.

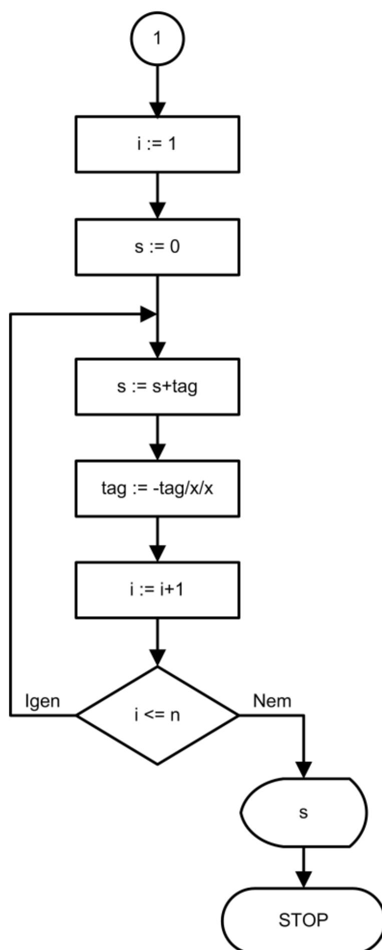
Elsőként n és x értékét kell beolvasni. A feladat akkor értelmes, ha $n > 0$ és $x \neq 0$. Észrevehetjük, hogy bármelyik tag az előzőből a

$$tag = -tag / x / x$$

összefüggéssel számítható. A legelső tag x^3.



© Aradi, Bécsi, Gyenes, Péter, BME

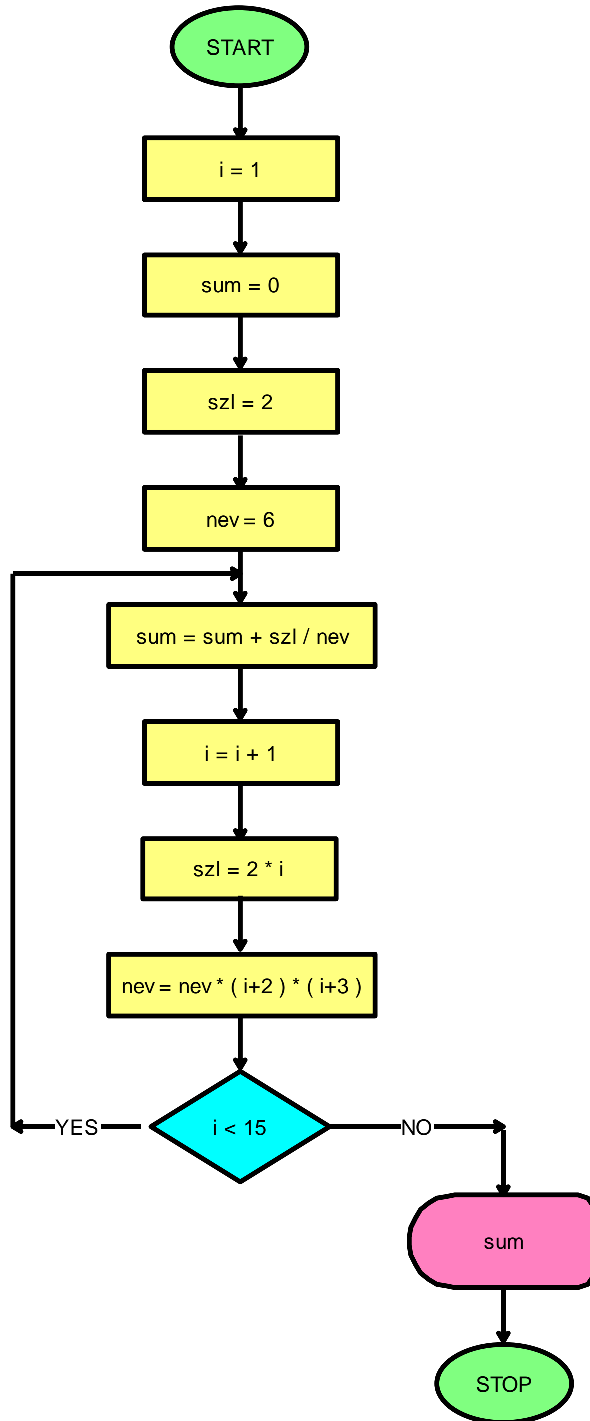


83. ábra: A P2_2 összeg kiszámítása

P2_3. Készítsünk folyamatábra programot z = 2/3! + 4/5! + 6/7! … 15 tagú összeg számítására. Sorszámozzuk meg a tagokat, legyen a sorszám változó neve ***i***. Az első sorszáma 1, a második 2, stb 15-ig. A számláló 2-ről indul és minden tagnál 2-vel nő, azaz a ***szl*** = 2*i alakban írható fel. A nevező induló értéke 3! = 1*2*3 = 6, és minden tag esetében a tag sorszám felhasználásával kifejezve

nev = ***nev****(i+2) * (i+3),

azaz a 2. tagnál ***nev*** = 6 * 4 * 5 = 5! = 120. Ezeket kell összegezni egy sum változóban. (A faktoriális művelet a Pascalban nincs értelmezve.) Programunkban nincs beolvasandó adat.



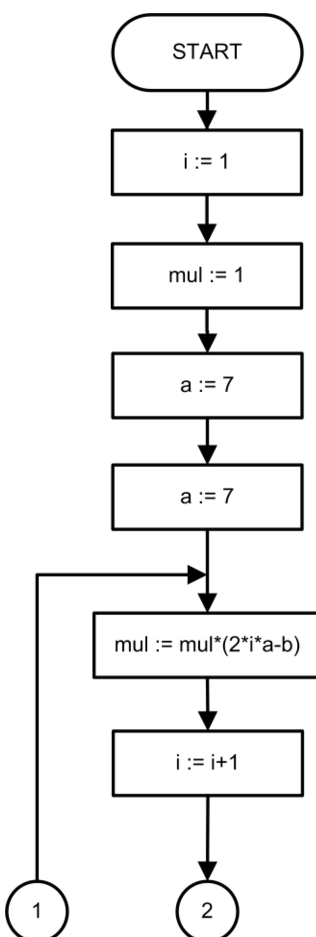
84. ábra: A P2_3 összeg kiszámítása

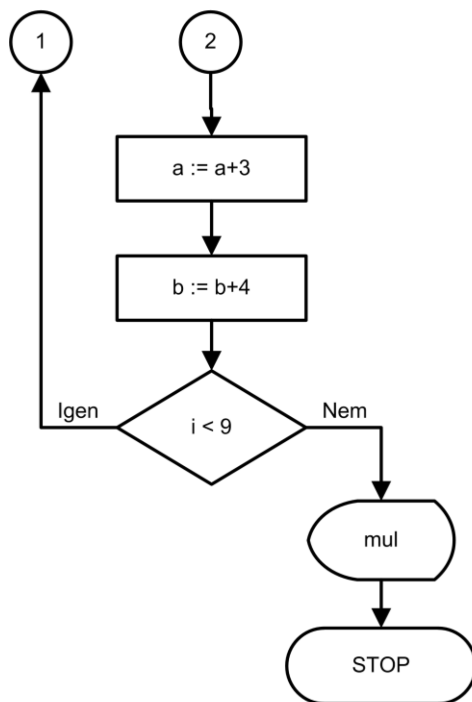
P2_4. Készítsünk folyamatábra programot a

$$mul = (2*7-13)(4*10 – 17)(6*13 - 21)… 9$$

tényezős szorzat számítására.

Az egyes tényezők sorszáma ezúttal is legyen ***i***-vel jelölve. Minden tényezőben egy szorzat és egy érték különbsége van. A szorzat felírható 2*i****a*** alakban, ahol ***a*** kezdő értéke 7, és minden újabb tagnál 3-al nő. A kivonandót jelöljük b-vel, ennek kezdő értéke 13, és minden újabb tagnál 4-el nő. Az egyes tagokat egy ***mul*** jelű változóban szorozzuk össze.





85. ábra: A P2_4 szorzat kiszámítása

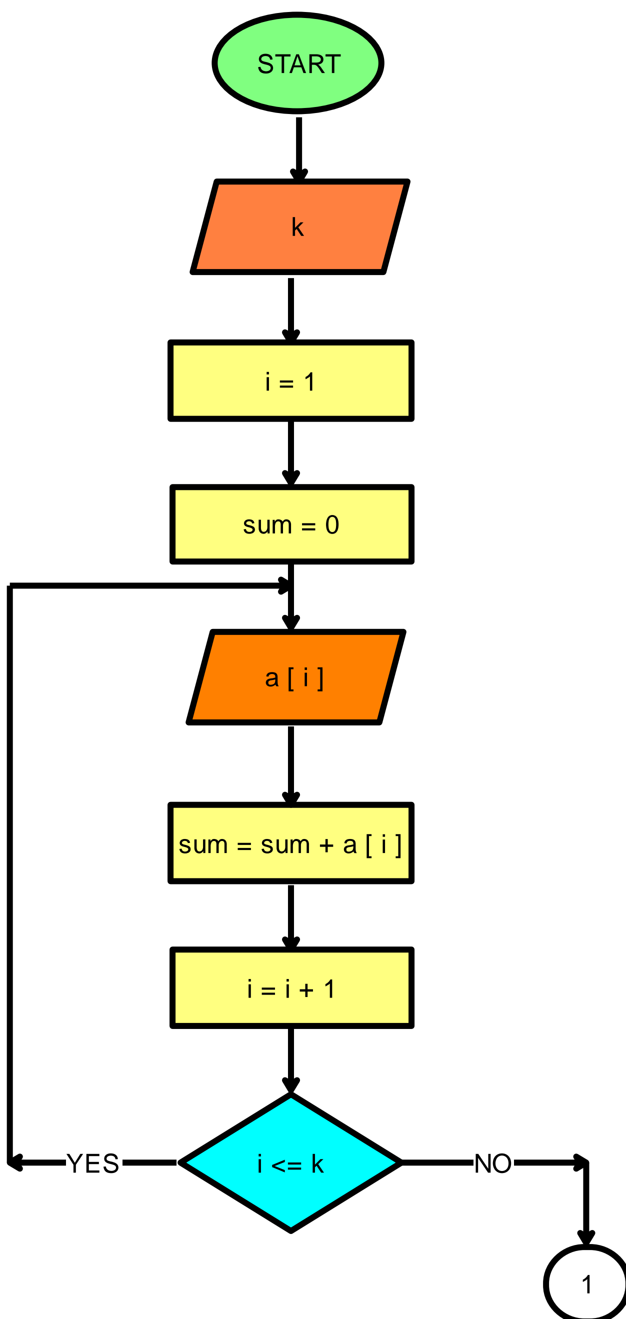
6.1.3 Kétszeres ciklust használó feladatok

Lényeges különbség van a két ciklust és a kettős ciklust igénylő programok között. A két ciklus végrehajtása esetén, ha az első n lépést, a második m lépést igényel, akkor a végrehajtások száma n+m lesz. Ezzel szemben a kettős ciklus (egymásba ágyazott → nested loop) esetén a végrehajtások száma n*m lesz. Nézzünk először egy példát a két ciklust igénylő programra.

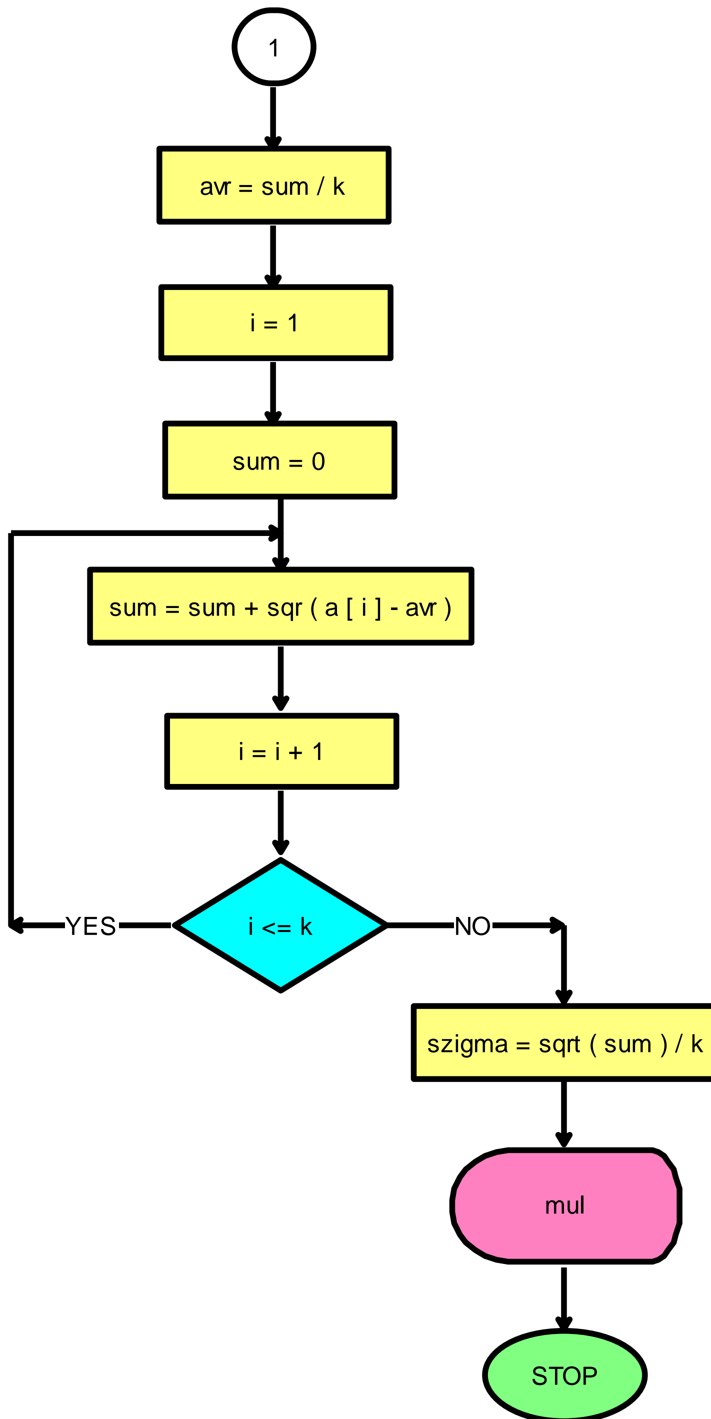
P3_1. Számítandó egy k elemű vektor négyzetes szórása. A számítás alapja a

$$\sigma = \frac{\sqrt{\sum_{i=1}^{k} (a_i - atl)^2}}{k}$$

képlet, azaz az egyes számok átlagtól való eltérésének négyzetátlaga. Ehhez először ki kell számolni a vektor átlagát, majd az elemeket újra használva a négyzetes eltéréseket számíthatjuk. Az első ciklusban a vektor elemeinek beolvasását és az átlag meghatározását, a másodikban a szórás számítását végezzük el.



 © Aradi, Bécsi, Gyenes, Péter, BME



86. ábra: Szórás kiszámítása

P3_2. Készítsünk folyamatábra programot Pascal háromszög első n sorának előállítására.

1.sor	1
2.sor	1 2 1
3.sor	1 3 3 1
4.sor	1 4 6 **4** 1
5.sor	1 5 10 10 5 1

Az i.-ik sor j.-ik elemét (az elem sorszáma 0-val kezdődik) az

$$\binom{i}{j}$$

(ejtsd: i alatta j) képlet (binomiális együttható) adja. Ennek programozható alakja

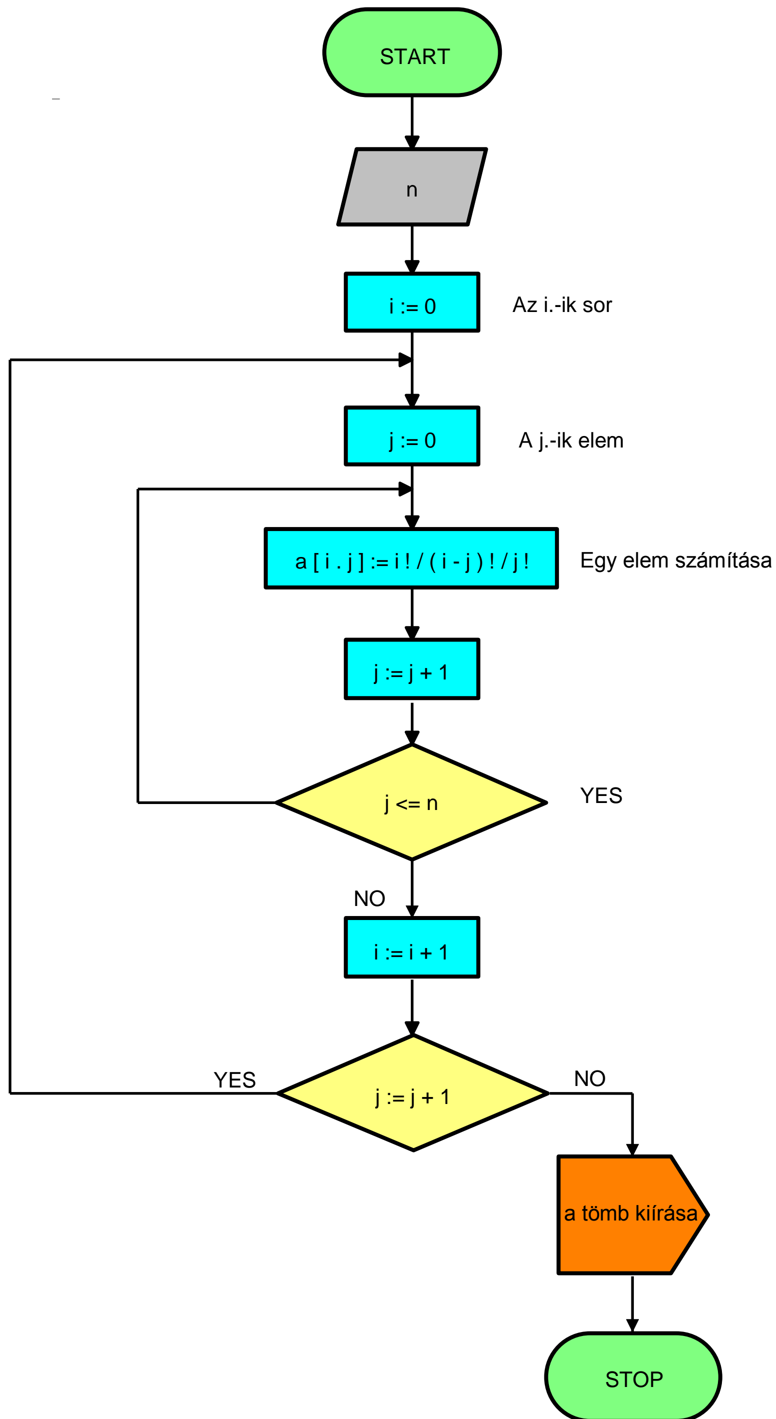
$$\binom{i}{j} = i!/(i-j)!/j!$$

lesz. Például a 4. sor 3 eleme (vastagon szedve)

$$\binom{i}{j} = 4!/(4-3)!/3! = 1*2*3*4/1/1*2*3 = 4$$

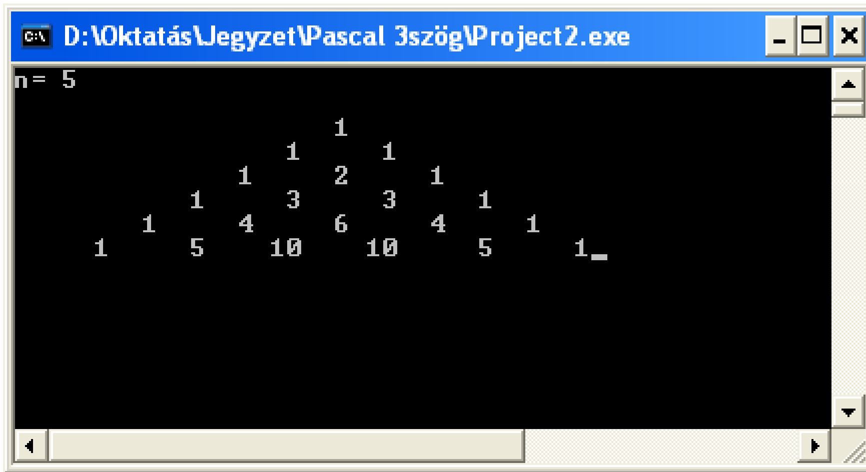
Programunkban lesz egy sor indexváltozó (i =0..n), és egy oszlopváltozó (j = 0..i) értékekkel. A faktoriális számításhoz kell egy segédváltozó (k). A háromszög elemeit egy n*(n+1) méretű tömbben helyezzük el, amelynek soronként egyre több eleme lesz.

Feladatunkban egyetlen beolvasandó adat, a sorok száma (n) lesz.



87. ábra: Pascal háromszög előállítása

Érdekességképpen bemutatjuk a diagram alapján megírt program futási képét (A program utasításlistáját későbbi fejezetben megmutatjuk).



88. ábra: Pascal háromszög előállító program képernyőképe

6.2 Példák a Pascal adattípusok gyakorlására

A jó program alapja a megfelelően megválasztott, a feladathoz leginkább illeszkedő, helykímélő adattípusok megválasztása. Ehhez elengedhetetlenül szükséges azok alapos ismerete. Ez a fejezet példái ehhez nyújtanak segítséget.

P4_1. Hőmérséklet mérések feldolgozásához válasszunk alkalmas adattípusokat. Adjuk ezeknek értéket. A hőmérséklet általában nem egész érték, ezért **single** adattípust választunk tárolásához:

```
type  t_ temp = single;
      t_tempek = array of t_temp;        // Adattömb
var   t1    : t_temp;                    // Egy adat tárolása
      tsok : t_tempek;                   // Adatsor tárolása
      f1    : t_temp;                    // file változó (minden rekord 1 adat)
      f sok : t_temp;                    // file változó (1 rekord 1 adatsor)
      t     : shortint;                  // segédváltozó
```

Értékadások:

```
t1 := 39.2;
t1 := int(20);                           // Valós konverzió
t1 := 25;                                // Automatikus típus konverzió
t  := -6;                                // Egésztípus
t1 := t;                                 // Automatikus egész / valós típuskonverzió
```

 © Aradi, Bécsi, Gyenes, Péter, BME

P4_2. Ciklusváltozó típusának megválasztása. A ciklusváltozó típusát a ciklus lépésszám határozza meg. Általában indexként használjuk, ezért (kivételes esetektől eltekintve) pozitív egész értékeket vesz fel. Ha a lépésszám nem haladja meg a 255 értéket, akkor **byte** típust választunk.

```
var i : byte;
```

Ha ez kevés lenne, akkor word típusú ciklusváltozót deklarálunk (max. 65535 lehet).

```
var i : word;
```

Ha valamely rendkívüli esetben ez sem volna elég, akkor a **longword** típust választjuk (max. 4294967295 lehet).

```
var i : longword;
```

Lehet olyan feladat, ahol a ciklus karakter értékeket vesz fel. Ilyenkor értelemszerűen típusa **char** lesz.

```
var ch : char;
```

Egy ilyen ciklusutasítás lehet az alábbi:

```
for ch:='a' to 'z' do <utasítás>;
```

A ciklusváltozó felvehet más, nem standard típust is. Tekintsük az alábbi típusdefiníciót:

```
type t_day = ( mon, tue, wed, thu, fri, sat, sun );   // Enumerated (felsorolásos típus)
var inap : t_day;
```

Ekkor a ciklusutasítás :

```
for inap:=mon to fri do <utasítás>;          // Munkanapok
```

P4_3. Válasszunk alkalmas típust oktális (8-as számrendszerű) számok feldolgozására.

```
type t_oktal = 0..7;                          // Intervallum típus
var                                           onr : t_oktal;
```

Ekkor az értékadások:

```
onr := 3;                                     // helyes
onr := 8;                                     // hibás utasítás, érték túllépés

onr := 10/3;                                  // hibás, a kifejezés valós típusú
onr := 10 div 3;                              // helyes
```

```
onr := round (11/ 3);        // helyes, onr = 4 lesz
onr := trunc (11/ 3);        // helyes, onr = 3 lesz

onr := sign (8);             // helyes, onr = 1 lesz (a sign – előjel- Math unit
                             része )
```

P4_3. Egy véleménykutató íven minden kérdésre 15 lehetőségből legfeljebb 4-et jelölhet meg a válaszadó. A megfelelő adattípus ekkor az alábbi lehet:

```
type t_resp = set of 1..15;  // Halmaz típus
var  voks : t_resp;          // Szavazat típusú változó

voks := [ 1,3,6 ];           // 1,3, és 6 válasz
voks := [ ];                 // nincs válasz
```

P4_4. Maximálisan 10 nagybetűből álló betűs szavak feldolgozásához alkalmas az alábbi típus:

```
type t_word = string [10];
var w1 : t_word;             // változó

w1:= 'Pára';                 // Értékadás
```

A w1 változó tartalma :

5	**P**	**á**	**r**	**a**	Q	/	f	,	$

Az első szám a **string** aktuális hossza (itt 5 byte). A maradék 3 byte nem definiált. A **string** típusú változó használat előtt célszerű „nullázni", azaz feltölteni szóköz (space) karakterrel.

```
w1 : = '          ';         // 10 db szóköz
```

A **string** típusú változók aktuális hossza értékadással módosítható.

```
w1:= 'Kukafej';              // Hossz = 7
writeln (w1);                // Kiírás : Kukafej
w1:= 'Zoli';                 // Hossz = 4
writeln (w1);                // Kiírás : Zoli
w1[0] := #7 ;                // Hossz átírása 7-re
writeln (w1);                // Kiírás : Zolifej
w1 := 'kukafejtető';
writeln (w1);                // Kiírás : Kukafejtet  ( levágja a 10 fölötti
                             //karaktereket )
```

A **string** egy eleme és **char** típus egymással kompatibilis.

 © Aradi, Bécsi, Gyenes, Péter, BME

```
var ch : char;
```

ekkor ch : = w1[2] ; hatására ch = 'u' lesz.

```
ch := chr (65);                  // 'A'  chr ( ) standard függvény
w1[5] := ch;                     // w1 = KukaAejtet
```

6.3 Kifejezések gyakorlása

A kifejezések helyes használatához elengedhetetlen az operátorok hierarchikus rendjének ismerete. Jelen fejezetben az operátorok használatát gyakoroljuk.

P5_1. Feltételezzük az alábbi deklarációkat

```
var i,j : shortint;
    b,c : boolean;
    ch  : char;

i := 3 - - 4;                    // i = 7 lesz, az első – jel kivonás, a //második ne-
                                 gatív előjel
j := 3 + 4 * 5;                  // j = 23, a szorzás magasabb prioritású
j := ( 3 + 4 ) * 5;              // j = 35, a zárójel miatt

i := 5;
j := 8;

writeln( i mod j + 9 div j ) ;            // Kiírás 6
writeln( i mod ( j + 9) div j ) ;  // Kiírás 0

writeln( i mod 3 + round ( 9 / i ) );  // Kiírás 4
j := 3 + 2 * succ(ord ('!') );   // j = 71, mert a '!' ASCII kódja 33
i := sign ( j – 20);             // i = 1

writeln ( i shl 4 );             // Kiírás 16, mert shl 4 balra léptet 4 //bináris he-
                                 lyi értéket, azaz 16-al szoroz
writeln ( j shr 2 );             // Kiírás 17, mert shr 2 jobbra léptet 2 //bináris
                                 helyi értéket, azaz 4-el oszt
```

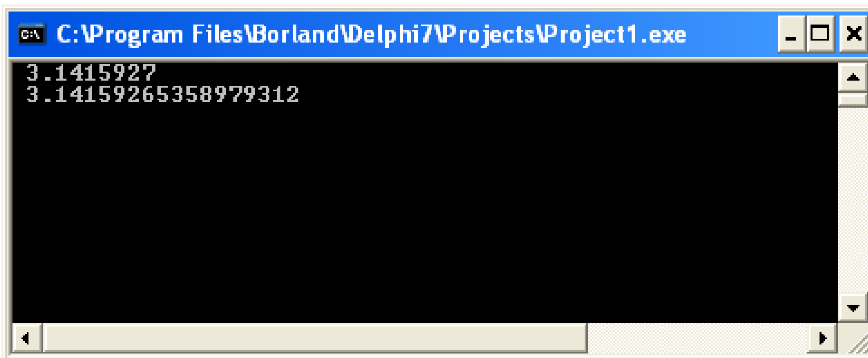
P5_2. Összetett kifejezések

```
const k    = 21;
      c    = '?';
var i,j   : integer;
    b,d  : boolean;
    ch   : char;

i  := 2* 27 – 9 div 4 * 3;       // i = 48
j  := 10 - trunc (sqrt ( k ) );  // j = 6
b := i = j ;                     // b = false
```

```
ch := pred ( c );                      // ch = ’>’
writeln ( ord (ch ) < i );             // Kiírás true, mert a ’>’ kódja 62
writeln ( not b or  (ch  > ’i’ );      // Kiírás true, mert a ’>’ kódja 62

d := c = chr (63);                     // d = true, mert chr(63) = ’?’, azaz d = c
writeln ( ( j in [2..6] ) and not d )  // Kiírás false, mert j = 6 benne van a 2..6
                                       //intervallumban
```

Pl 3. Kifejezések valós típusú változókkal

```
var x, y : single;
       d  : double;
x := sin (pi / 4) + 2;                 // x = 2.707107
writeln ( 2 * frac(x) );               // Kiírás 1.414214, mert x törtrésze = //0.707107
writeln ( 5 * int(x) );                // Kiírás 10.0, mert x egészrésze = 2.

y :=  tan (pi / 3);                    // y = 1.7321

writeln ( sqr (x) + sqrt (5*y) );      // Kiírás 10.27126
writeln (exp (y) );                    // Kiírás 5.65223
x := log10 (1000);                     // x = 3.0
writeln (x / 2);                       // Kiírás 1.5

y := pi;
d := pi;
writeln (y);                           // Kiírás 3.1415927
writeln (d);                           // Kiírás 3.141592765358979312 (dupla
                                       //pontosságú)
```



89. ábra: Valós típusokat kiíró program képernyőképe

Az alábbiakban néhány feladatot mutatunk be a matematikában szokásos kifejezések Pascal szintaktika szerinti átírására.

P6_1.

$$\frac{x^2 + y^2}{2ab} \rightarrow (x * x + y * y) / 2 / a / b \qquad // \; a \neq 0 \text{ esetén}$$

P6_2.

$$\frac{a - b}{\sqrt{2ax + b}} \rightarrow (a - b) / sqrt\ (2 * a * x + b) \qquad // \; 2ax+b > 0 \text{ esetén}$$

P6_3.

$$\sqrt{3 \sin \delta + \lg x^2} \rightarrow sqrt\ (3 * sin(\ delta\ ) + ln(\ x * x) / ln(\ 10)$$

P6_4.

$$x^7 + e^{2\alpha x} \rightarrow exp(\ 7 * ln(\ x)) + exp(\ 2 * x * alfa\ )$$

6.4 Változók aktuális értékének meghatározása

P7_1.

x := 2 + 19 **mod** 3 * 2; // x = 4

P7_2.

y := **sqr** (9) – 55; // y = 36

P7_3.

z := 3 + 4 * **trunc** (4.8) **div** 2; // z = 11

P7_4.

u := **pred** (**chr** (6*z)); // u = 'A' karakter

P7_5.

w := **ln**(100) / **ln**(10) = 2 ; // w = **true**, **boolean**

P7_6.

z := **not** w **or** (u < **chr** (80)); // z = **true**, **boolean**

6.5 Példák különböző értékadásra

Nullázzuk a típusnak megfelelő értékkel az alább meghatározott u,w és z változókat.

```
type t_intv   = 5 .. 10;                    // Intervallum típus
     t_halm = set of 10 .. 99 ;             // Halmaz típus
     t_arr   = array [char] of t_halm;      // Halmaz típusú tömb (indexe char lehet)

var u  : array [t_intv] of t_arr ;          // Halmaz típusú mátrix (indexe 5..10 lehet )
    w : array [ - 3 .. 4, 'B' .. 'E' ] of boolean;    // Logikai tömb
    z  : array [boolean] of char;
    i , j : shortint ;                      // ciklusváltotó
    ch   : char ;
```

P8_1. u tömb nullázása

```
for i := 5 to 10 do                         // sorok szerint
  for ch := #0 to #255 do                   // oszlopindex az összes karakter
    u [ i, ch ] := [ ];                     // üres halmaz
```

P8_2. w tömb nullázása

```
for i := -3 to 4 do                         // sorok szerint
  for ch := 'B' to 'E' do                   // oszlopindex karakter
    w [i, ch] := false;                     // logikai érték nullázása
```

P8_3. z kételemű char típusú tömb nullázása

```
z [false] := ' ';                           // karakter típusú nullázás szóközzel
z [true] := ' ';
```

6.6 Feltételes utasítások gyakorlása

P9_1. Írjuk át Pascal formába az alábbi értékadást

$$x = \begin{cases} 2 * \mathrm{tg}\,(i\pi\gamma) & \text{ha } i \text{ páratlan} \\ \sin^2(i\gamma) & \text{ha } i \text{ páros} \end{cases}$$

Megoldás:

```
if odd ( i ) then x := 2 * sin (i*pi*gamma) / cos (i*pi*gamma)
             else  x := sqr ( sin (i*gamma) ;
```

P9_2. Írjuk át Pascal formába az alábbi értékadást

$$Y = \begin{cases} 3 * \lg(2x) & \text{ha } x > 0 \\ 11.5 & \text{ha } x = 0 \\ \sqrt{-x/2} & \text{ha } x < 0 \end{cases}$$

Megoldás:

```
if  x > 0 then Y := 3 * ln ( 2 * x ) / ln (10)
          else if x < 0 then Y := sqrt (-x / 2 )
                     else Y := 11.5;
```

P9_3. Írjuk át Pascal formába az alábbi értékadást

$$s = \frac{|x|}{\ln(x) - 2.2}$$

Megoldás:

```
if (x > 0) and (ln (x) <> 2.2) then s := abs (x) / (ln (x) – 2.2);
```

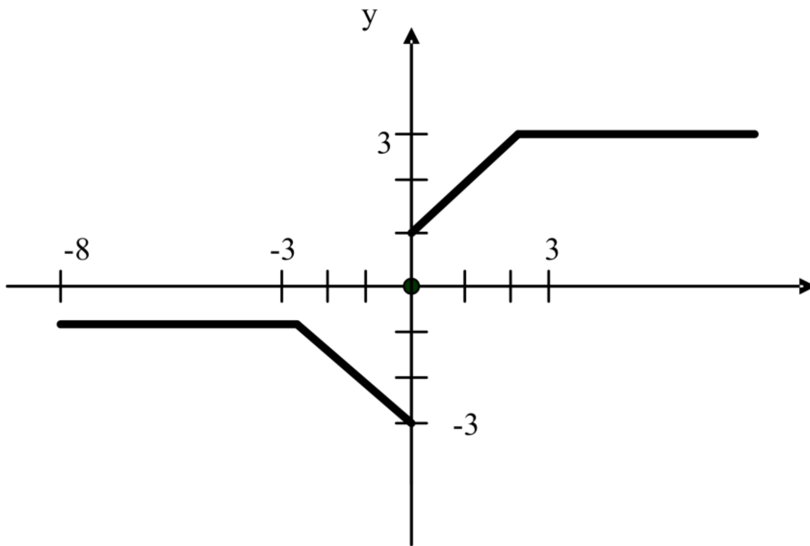
P9_4. Írjuk át Pascal formába az alábbi értékadást

$$Y = \begin{cases} 12n & \text{ha n osztható } 3-\text{mal} \\ 4n & \text{ha n osztható } 5-\text{tel} \\ 2n & \text{ha n osztható } 7-\text{tel} \\ n & \text{egyébként} \end{cases}$$

Megoldás:

```
if n mod 3 = 0 then m:= 12 * n
             else if n mod 5 = 0 then m:= 4*n
                     else if n mod 7 = 0 then m:= 2*n
                                 else m :=n;
```

Pl 5. Írjuk át Pascal formába az alábbi szakaszos függvényt



Megoldás:

```
if  (x < -3) and (x > -8) then y:= -1
                          else if x < 0 then  y:= -x - 3
                                        else if x < 3 then y:= x + 1
                                                      else y := 3;
```

6.7 Programkapcsoló (case utasítás) gyakorlása

P10_1. Számoljuk meg egy adott stringben előforduló kisbetűket, nagybetűket, számokat és jeleket.

Megoldás:

```
var st              : string;
    kb, nb, sz, jel : byte;
    i               : byte;

for i:= 1 to length (st ) do          // length (st) a szöveg karaktereinek száma
  case st[i] of                       // az i.-ik karakter vizsgálata
    'a'..'z'  : kb := kb  +1;         // kisbetű
    'A'..'Z'  : nb := nb  +1;         // nagybetű
    '0'..'9'  : sz := sz  +1;         // számjegy
    else jel := jel +1;                    // egyéb karakter
  end;
```

P10_2. Egy közvélemény-kutatás feldolgozásakor a lehetséges válaszok: A, N, J, K és 0. Számoljuk meg n kitöltött ív esetén az egyes válaszok %-os arányát.

 © Aradi, Bécsi, Gyenes, Péter, BME

Megoldás:

```
type t_resp = (A,N,J,K);                // Felsorolásos típus
var v                  : t_resp;
    i,va,vn,vj,vk,v0 : byte;

for i:= 1 to n do                       // az összes válaszok feldolgozása
  case v of                             // az i.-ik válasz vizsgálata
    A   : inc(va);                      // A válaszok számának növelése
    N   : inc(vn);                      // N válaszok számának növelése
    J   : inc(vj);                      // J válaszok számának növelése
    K   : inc(vk);                      // K válaszok számának növelése
    else inc(v0)                        // 0 válaszok számának növelése
  end;

writeln (100*va/n, 100*vn/n, 100*vj/n, 100*vk/n, 100*v0/n,);     // %-os arányok
```

6.8 Példák ciklusok használatára

A számítógépes programok leginkább sokszor ismétlődő számításokat végeznek. Ezzel sok mechanikus munkától kímélik meg az alkalmazót. Ezért nagy jelentősége van a ciklusutasítások használatának. A programozó számára fontos feladat annak eldöntése, melyik ciklus utasítást válassza. Általános irányelv, ha a lépések száma ismert, akkor legegyszerűbb a **for** típusú ciklus használata. Ha a ciklusváltozó nem megszámolható típusú (pl valós szám), vagy ha nem ismeretes a lépések száma, akkor a **while** típust választjuk. Ha a ciklusba való belépéskor a leállás feltétele nem áll rendelkezésre (a ciklus belsejében derül ki), akkor a **repeat** típust kell használni. Ezen ciklusutasítások mindegyikére mutatunk példákat komplett, működő program formában.

6.8.1 for – to – do ciklusok

P11_1. Eldöntendő egy k elemű vektorról, hogy számtani sorozatot alkot-e.

```
// Számtani-e K.J. 2010.06.02.

program pl_for_1;

var  i, k : byte;
     a    : array of single;            // dinamikus tömb
   yes : boolean;
   dif  : single;
begin
 write ('k = '); readln (k);            // Az elemek számának bekérése
 Setlength (a,k);                       // A vektor méretének beállítása : k
 for i:=1 to k do
   begin
```

```
      write ('A[' , i , ']= '); readln (a[i]);   // Az elemek bekérése
   end;
   yes := true;                                  // feltételezzük, hogy a válsz igen
   dif := a[2] – a[1];                           // differencia
   for i:=2 to k-1 do
     if a[i+1] – a[i] <> dif then
                          begin
                            yes:=false;          // már nem számtani sorozat
                            exit;                // Kiugrás a ciklusból
                          end;
   if yes then writeln ('Számtani sorozat !')
          else writeln ('Nem számtani sorozat !');
   readln;
 end.
```

Programunk futási képe az alábbi.

```
D:\Oktatás\Jegyzet példák\pl_for_1\Project1.exe
k = 5

Az elemek :
A[1]= -1
A[2]= 3
A[3]= 7
A[4]= 11
A[5]= 15

Számtani sorozat !
```

90. ábra: A számtani sorozat példaprogram képernyőképe

P11_2. Készítsünk programot, amely egy n*n méretű speciális mátrixot állít elő az alábbi szabály szerint.

$$A\ mátrix\ \ eleme = \begin{Bmatrix} 1 & ha\ a\ főátlóban\ \ van \\ 2 & ha\ a\ mellékátló\ \ ban\ \ van \\ 0 & egyébként \end{Bmatrix}$$

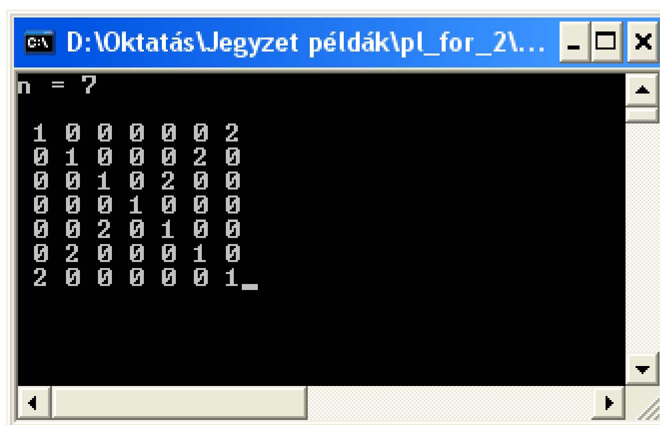
Megoldás:

```
// Spec. mátrix N.S. 2010.06.11.

program pl_for_2;
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
var   i,j, n : byte;
          a : array[1..10,1..10] of 0..2;      // statikus kétdimenziós tömb
begin
  write ('n = '); readln (n);                          // Az sorok számának bekérése
  for i:=1 to n do                             // Sorindex 1..n
    for j:=1 to n do                           // Oszlopindex 1..n
      if i =j then a[i , j] := 1               // Főátló
            else if i + j = n + 1 then  a[i , j] := 2     // Mellékátló
                              else a[i , j] := 0;      // Egyéb
  for i:=1 to n do                             // Kiírás mátrix formában
   begin
    writeln;                                   // Új sor
    for j:=1 to n do write (a[i , j]:2);       // Minden elem 2 pozícióra
   end;
  readln;
end.
```

Programunk futási képe az alábbi.



91. ábra: A mátrix előállító példaprogram képernyőképe

P11_3. A 6.1.3. pontnál P3_2-ben megoldottuk a Pascal háromszög előállítását végző program folyamatábráját. Most megmutatjuk a hozzátartozó utasításlistát.

```
// Pascal 3 szög S.N. 2009.05.06.
program pl_for_3;

{$APPTYPE CONSOLE}                             // Delphi környezetben
uses SysUtils;

var        n,i,j,k   : byte;
    faki,fakj,fakd : word;
                 a : array[0..10,0..10] of word;
```

```
begin
 write('n= '); readln(n);              // Sorok számának bekérése
 for i:= 0 to n do
  for j:= 0 to n do a[i,j]:= 0;        // Tömb nullázása

 for i:= 0 to n do                     // i.-ik sor
  for j:=0 to i do                     // j-ik elem
  begin
   faki :=1;
   for k:=1 to i do faki := faki * k;  // i!
   fakj :=1;
   for k:=1 to j do fakj := fakj * k;  // j!
   fakd :=1;
   for k:=1 to i-j do fakd := fakd * k; // (i-j)!
   a[i,j]:= round (faki / fakj / fakd);
  end;

 for i:= 0 to n do                     // Kiírás
  begin
   writeln;
   for j:=0 to n do
    if a[i,j]> 0 then write(a[i,j]:4); // Csak a nem 0 elemek
  end;
 readln;
end.
```

A program futási képe az alábbi.

92. ábra: A Pascal háromszög előállító példaprogram képernyőképe

Megjegyezzük, hogy e példa megoldása a faktoriális számító függvény alkalmazásával sokkal szebb.

 © Aradi, Bécsi, Gyenes, Péter, BME

P11_4. A jó programozói készség eléréséhez szükséges a már megírt program analízise is. Határozzuk meg, hogy az alábbi programrészlet miként működik, mik lesznek a benne szereplő változók értékei.

```
m:= 5;
for k := 1 to m do
  case m of
    -2,-1,0 : m := m + k;
    2,3,4,5 : m := m – k;
  end;
```

A megoldáshoz értéktáblázatot célszerű használni:

k		1	2	3	4	5
m	5	4	2	-1	3	-2

6.8.2 while – do ciklusok

Ezt a ciklusutasítást akkor használjuk, ha az ismétlések számát nem ismerjük. Az ismétlés feltétele a ciklusba való belépés előtt kerül ellenőrzésre, s ha a feltétel aktuális érétke **false**, akkor a ciklusmag nem lesz végrehajtva. Akkor is ezt a ciklusfajtát használjuk, ha a ciklusváltozó nem megszámlálható típusú.

P12_1. Írjunk Programot, amely szinusz táblázatot készít. Az x 0..30 fokig változik 0.1 fokonként.

```
// Szinusz tábla A.B. 2008.05.06.
program pl_while_1;

var x,y : single;
begin
  x := 0;  y := 0;
 writeln ('Szinusz áblázat :'); writeln;
 while x < 1 do begin
     write (x:5:2); x:=x+0.1;            // Tizedek tengelye
 end;
 writeln;
 x:=0;
 while x < 30 do                         // Egész fokok
   begin
     writeln; write (x:6:0,'  ');        // Egészek tengelye
     while y < 1 do                      // Tized fokok
       begin
         write (sin ( pi*(x+y ) / 180):5:2);   // Szinusz érték
         y := y + 0.1;
       end;
```

```
      x:= x + 1;
      y:=0;
    end;
  readln;
end.
```

A program futási képe az alábbi.

```
D:\Oktatás\Jegyzet példák\pl_while_1_sin_table\Project2.exe
Szinusz táblázat :

         0.00 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90

    0    0.00 0.00 0.00 0.01 0.01 0.01 0.01 0.01 0.01 0.02
    1    0.02 0.02 0.02 0.02 0.02 0.03 0.03 0.03 0.03 0.03
    2    0.03 0.04 0.04 0.04 0.04 0.04 0.05 0.05 0.05 0.05
    3    0.05 0.05 0.06 0.06 0.06 0.06 0.06 0.06 0.07 0.07
    4    0.07 0.07 0.07 0.07 0.08 0.08 0.08 0.08 0.08 0.09
    5    0.09 0.09 0.09 0.09 0.09 0.10 0.10 0.10 0.10 0.10
    6    0.10 0.11 0.11 0.11 0.11 0.11 0.11 0.12 0.12 0.12
    7    0.12 0.12 0.13 0.13 0.13 0.13 0.13 0.13 0.14 0.14
    8    0.14 0.14 0.14 0.14 0.15 0.15 0.15 0.15 0.15 0.15
    9    0.16 0.16 0.16 0.16 0.16 0.17 0.17 0.17 0.17 0.17
   10    0.17 0.18 0.18 0.18 0.18 0.18 0.18 0.19 0.19 0.19
   11    0.19 0.19 0.19 0.20 0.20 0.20 0.20 0.20 0.20 0.21
   12    0.21 0.21 0.21 0.21 0.21 0.22 0.22 0.22 0.22 0.22
   13    0.22 0.23 0.23 0.23 0.23 0.23 0.24 0.24 0.24 0.24
   14    0.24 0.24 0.25 0.25 0.25 0.25 0.25 0.25 0.26 0.26
   15    0.26 0.26 0.26 0.26 0.27 0.27 0.27 0.27 0.27 0.27
   16    0.28 0.28 0.28 0.28 0.28 0.28 0.29 0.29 0.29 0.29
   17    0.29 0.29 0.30 0.30 0.30 0.30 0.30 0.30 0.31 0.31
   18    0.31 0.31 0.31 0.31 0.32 0.32 0.32 0.32 0.32 0.32
   19    0.33 0.33 0.33 0.33 0.33 0.33 0.34 0.34 0.34 0.34
   20    0.34 0.34 0.35 0.35 0.35 0.35 0.35 0.35 0.36 0.36
   21    0.36 0.36 0.36 0.36 0.36 0.37 0.37 0.37 0.37 0.37
   22    0.37 0.38 0.38 0.38 0.38 0.38 0.38 0.39 0.39 0.39
   23    0.39 0.39 0.39 0.40 0.40 0.40 0.40 0.40 0.40 0.41
   24    0.41 0.41 0.41 0.41 0.41 0.41 0.42 0.42 0.42 0.42
   25    0.42 0.42 0.43 0.43 0.43 0.43 0.43 0.43 0.44 0.44
   26    0.44 0.44 0.44 0.44 0.44 0.45 0.45 0.45 0.45 0.45
   27    0.45 0.46 0.46 0.46 0.46 0.46 0.46 0.46 0.47 0.47
   28    0.47 0.47 0.47 0.47 0.48 0.48 0.48 0.48 0.48 0.48
   29    0.48 0.49 0.49 0.49 0.49 0.49 0.49 0.50 0.50 0.50
```

93. ábra: A szinusz táblázatot előállító példaprogram képernyőképe

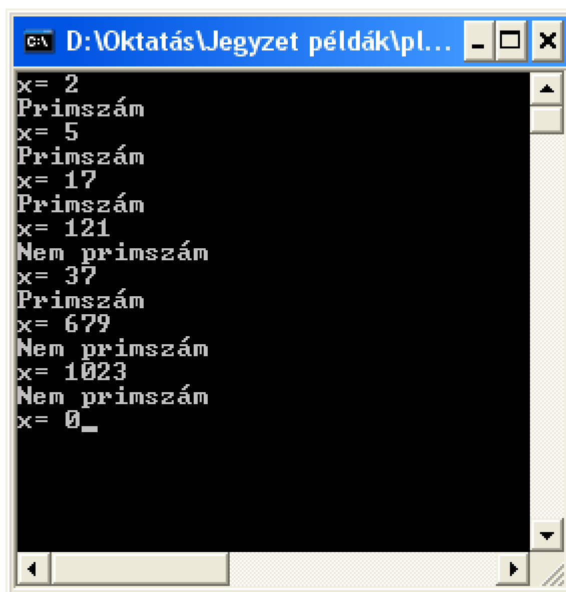
P12_2. Készítsünk programot, amely egy beolvasott számról eldönti, hogy prímszám-e.

```
// Prímvizsgálat A.Sz. 2010.05.06.
program pl_while_2;
var   x,i : integer;
      prim : boolean;
 begin
   repeat
      write('x= '); readln (x);          // Szám bekérése
      prim := true;                      // Feltételezzük, hogy prím
      if x mod 2 = 0 then prim:=false;   // Páros : nem prím
      i:=1;
      while (x > i * i) and prim do      // A szám gyökéig keresünk osztót
        begin
          if x mod (i*2 + 1) = 0 then prim:=false;    // Van osztója : nem prím
          inc(i);
        end;
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
        if x in [2,3,5] then prim:=true;          // 2,3,5 prím

        if prim then write ('Primszám')
                else write ('Nem primszám');
       writeln;
    until x <= 0;                                  // Ismétlés amíg pozitív
    readln;
  end.
```

Programunk futási képe az alábbi.



94. ábra: A prímszám vizsgáló példaprogram képernyőképe

P12_3. Készítsünk programot, amellyel cos(x) értéke ε pontossággal számítható. A cos(x) értékét Taylor sorával számíthatjuk ki.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k!} \cdot x^{2k}$$

Az ε a matematikában szokásos küszöbszám, értéke a program bemenő adata. A konvergáló sorozat i.-ik és (i +1).-ik tagjának különbségeként értelmezzük. Mivel példánkban az $x^{2i} / (2i)!$ tag i növelésével 0-hoz tart, ezért a tagok számítását és összegzését mindaddig folytatjuk, amíg az $x^{2i} / (2i)! > \varepsilon$ feltétel igaz. Az aktuális számláló előállítható az előző tag számlálójának x*x taggal való szorzá-

sával. Hasonlóképpen a mindenkori nevező a megelőző nevező i*(i+) taggal való szorzásaként adódik. A program utasításlistája az alábbi.

```
// Cos függvény Z.M. 2009.04.26.
program cosx;

var  x,eps, szl, cosx : single;
                nev,i : integer;
begin
  writeln ('A cos(x) számítása Taylor sorával'); writeln;
  write ('x (fokban) = '); readln (x);          // x bekérése
   x := pi * x / 180;                           // átváltás radiánra
  write ('pontosság (eps) = ');
  readln (eps);  writeln;                       // ε bekérése
  cosx := 1;                                    // A sorozat első tagja
  szl := -x * x;                                // A számláló kezdő értéke
  nev := 2;                                     // A nevező kezdő értéke
  i := 3;
  while abs ( szl / nev) > eps do
    begin
      cosx := cosx + szl / nev;
      szl := - szl * x * x;                     // előjelváltás, új számláló
      nev := nev * i * (i+1);                   // nevező új értéke : faktoriális
      i := i + 2;
    end;
   writeln ('A Taylor sorral számított érték : ',cosx:10:6);
   writeln ('A cos(x) standard függvény      : ',cos(x):10:6);
  readln;
end.
```

Programunk futási képe az alábbi.

```
D:\Oktatás\Jegyzet példák\pl_while_3_cosx\Project2.exe
A cos(x) számítása Taylor sorával

x(fokban)= 60
pontossβg (eps) = 0.00001

A Taylor sorral számított érték :    0.500000
A cos(x) standard függvény      :    0.500000
```

95. ábra: A cos(x) számító példaprogram képernyőképe

P12_4. Hányszor lesz végrehajtva az alábbi ciklus, és mi lesz a változók értéke lépésenként?

```
i := 1; j := 3;
 while j > 0 do
   begin
     j := ( j + 10 ) div ( 2 * i );
     i := i + 1;
   end;
```

Megoldás:

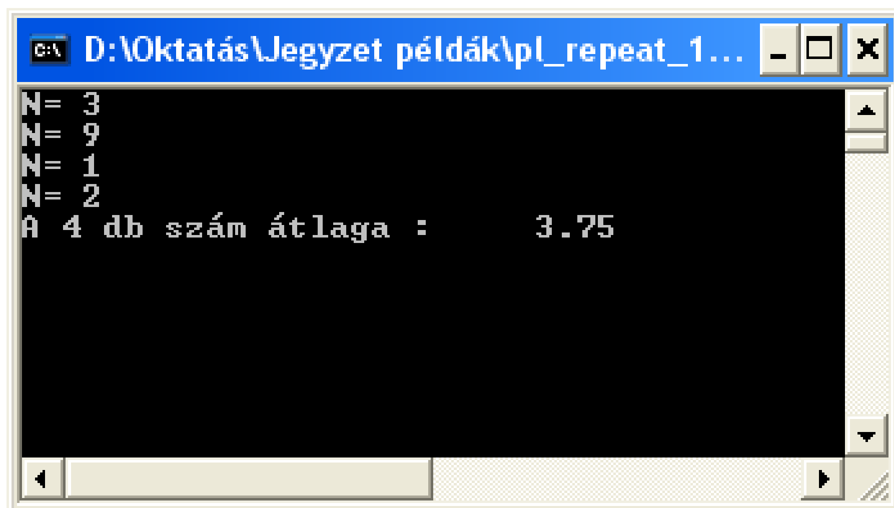
		ciklusban					
i	1	2	3	4	5	6	7
j	3	6	4	2	1	1	0

6.8.3 Repeat - until ciklusok

P13_1. Készítsünk programot, amellyel mindaddig olvasunk be egészszámokat, amíg az új és a megelőző szám különbsége 5-nél nagyobb. Ezután kiírjuk a beolvasott számok számtani átlagát. Itt nem ismerjük a lépések számát, így a **for** ciklus alkalmazása nem lehetséges.

```
// Repeat példa_1 K.I. 2009.01.26.
program repeat_pl;

var a,b,db,sum,dif : integer;
begin
 write ('N= '); readln(a);            // első szám bekérése
 sum := a;                            // mentés
 db := 1;                             // számláló
 repeat
  write ('N= '); readln (b);          // következő szám bekérése
  sum := sum + b;                     // mentés
  dif := abs (a-b);                   // különbség
  a := b;                             // az új legyen az előző
  inc (db);
 until dif <= 5;                      // leállás feltétele
 writeln ('A ',db,' db szám átlaga : ',sum/db:8:2);
 readln;
end.
```

Programunk futási képe az alábbi.



96. ábra: A P13_1 példaprogram képernyőképe

P13_2. Készítsünk programot, amely 45 számból 6 különböző számot sorsol véletlen generátorral és azt egy alkalmas tömbbe menti és kiírja.

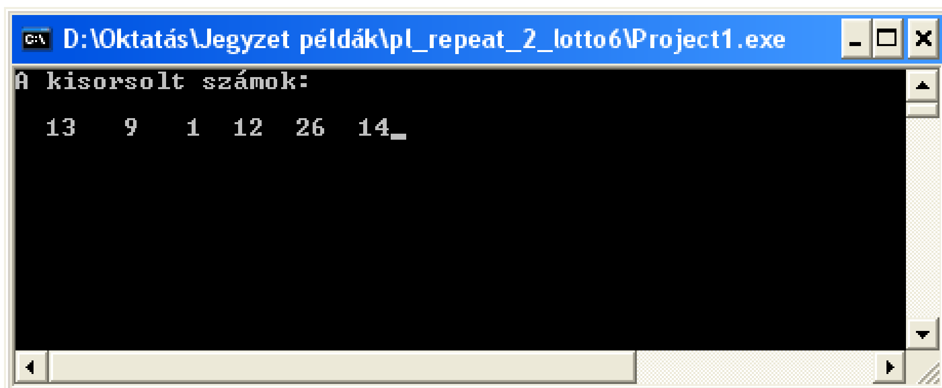
Egyszerű **for** ciklus itt sem jó, mivel lehet, hogy a 6 lépésben sorsolt számok között azonosak is lehetnek. Ezért mindaddig kell új számot sorsolni, amíg 6 különböző nem adódik.

```
// Repeat példa_2  K.I. 2009.01.26.
program lotto_6;

type t_num = 1..45;
     t_halm = set of  t_num;
var  sors    : t_halm;
     db,n    : byte;
begin
 randomize;
 sors := [ ];                        // üres halmaz
 db := 0;                            // számláló
 writeln ('A kisorsolt számok: '); writeln;
 repeat
  n := random (45) + 1;              // 1..45
  if not (n in sors) then            // ha még nincs benne a halmazban
                begin
                 sors := sors + [n]; // hozzáadjuk
                 write (n  :4);      // kiírjuk
                 inc (db);
               end;
 until db = 6;                       // ismétlés amíg 6 különböző lesz
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
  readln;
end.
```

Programunk futási képe az alábbi.



97. ábra: A hatos lottó sorsoló példaprogram képernyőképe

P13_3. Harmadik példánk legyen ezúttal is egy programrészlet elemzése. Hányszor fut le az alábbi ciklus, és mi lesz a változók értéke futás közben?

```
x := chr (66);
repeat
  case x of
    'B','C','E'  : x := chr ( ord (x) + 2);
    'A','F'.. 'H' : x := pred (x);
   else x := pred (x);
  end;
until x > 'E';
```

Használjuk az értéktáblát. x kezdeti érétke 'B' betű (kódja 66).

		ciklusban			
x	B	D	C	E	G

6.9 Rekord adattípus gyakorlása

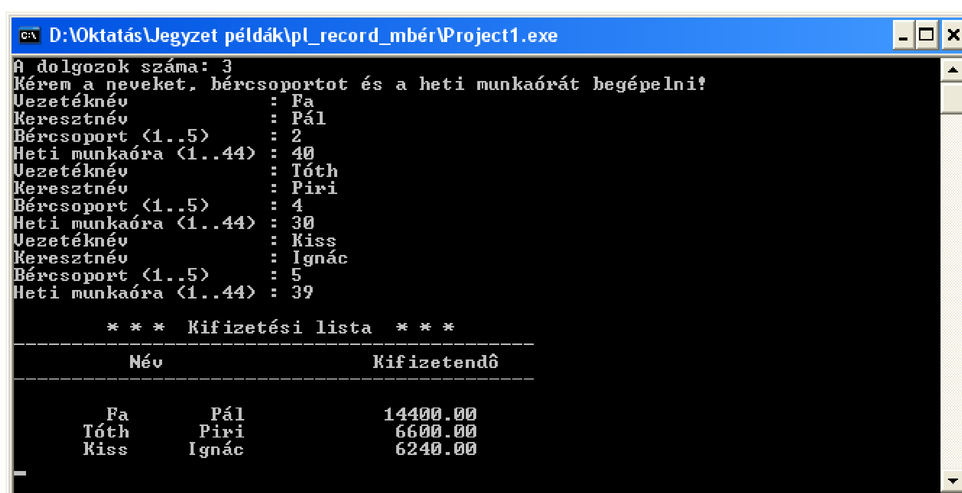
P14_2. Készítsünk programot, amellyel egy kisvállalat dolgozóinak heti munkabére számítható.

```
// Munkabér T.T. 2008.11.12.
program berkalk;

const dsz = 50;                          // a dolgozók maximális száma
type  t_st  = string [10];
        t_nev = record
                   elonev : t_st;
                   utonev : t_st;
                  end;

        t_mido = record
                    dolgozo: t_nev;
                    mora   : 0..44;
                   end;

        t_ber = record
                   dolgozo: t_nev;
                   bercsop: 1..5;
                   bere   : real;
                  end;

const bercs : array[1..5] of real = (450, 360, 290, 220, 160);        // Ötféle órabér

var     n,i  : byte;
        dolg  : t_nev;
        tdolg : array [1..dsz] of t_nev;         // dolgozok tömbje
       tmora : array [1..dsz] of byte;           // heti munkaórák tömbje
            q : array [1..dsz] of byte;          // besorolások tömbje
      mber : array [1..dsz] of t_ber;            // eredmények tömbje
begin
 write ('A dolgozok száma: '); readln(n);  // Tényleges dolgozószám
 writeln('Kérem a neveket, bércsoportot és a heti munkaórát begépelni!');
 for i:=1 to n do
  begin
   write ('Vezetéknév                : '); readln (dolg.elonev);
   write ('Keresztnév                : '); readln (dolg.utonev);
   tdolg[i] := dolg;                        // mentés a rekord mezőbe
   write ('Bercsoport (1..5)         :  '); readln(q [i] );
   write ('Heti munkaóra (1..44) : '); readln ( tmora[i] );
  end;

 for i:=1 to n do                         // a bér kiszámitása és az 'mber' tömb feltöltése
  with mber[i] do
   begin
    dolgozo := tdolg[i];                  // nevek áttöltése
    bercsop := q[i];                      // bércsoport áttöltése
    bere := tmora[i] * bercs[q[i] ];      // bér kiszámítása
   end;
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
    writeln;                                    // kifizetési lista készítés
    writeln ('      * * *  kifizetési lista  * * *      ');
    writeln ('--------------------------------------------');
    writeln ('        Név              Kifizetendő   ');
    writeln ('--------------------------------------------');
    writeln;
    for i:=1 to n do
      writeln (mber[i].dolgozo.elonev:10,
               mber[i].dolgozo.utonev:10,
               mber[i].bere:20:2);
   readln;
end.
```

Programunk futási képe az alábbi.



98. ábra: A munkabér számító példaprogram képernyőképe

P14_2. Készítsünk programot használt autók nyilvántartására. Egyszerűsített példánkban az autóknak csak 3 tulajdonságát vesszük számba. A 6.14.2 pontban e program továbbfejlesztett változatát megtaláljuk.

```
// Használt autó piac verzió_1. Kő Péter 2008.06.22.
program carmarket;

const   max  = 100;                        // Az autók maximális száma

type    t_st10 = string[10];
        t_car  = record
                   typ  : t_st10;
                   color: t_st10;
                   price: word;
```

```
                    end;
        t_cars  = array [1..max] of t_car;
var     n    : byte;
        one  : t_car;
        cars : t_cars;
        c    : char;

procedure w; begin writeln end;          // Soremelés eljárás

procedure datain(var cars : t_cars; var n : byte);          // Adat beolvasó eljárás
 var i : byte;
 begin
  write ('Autók száma : '); readln(n);
  for i:=1 to n do
   begin
    write ('Tí pus      : '); readln (cars[i].typ);
    write ('Szín        : '); readln (cars[i].color);
    write ('Ár (EUR) : '); readln (cars[i].price);
   end;
 end;

procedure list (cars:t_cars; n:word);       // Listázó eljárás
  var i:  byte;
  begin
    w;
    writeln ('   T'ípus          Szín          Ár (EUR)');
    writeln(' __________________________________________');
    for i:=1 to n do
    writeln (cars[i].typ:10, cars[i].color:14, cars[i].price:12);
  end;

begin                                    // Főprogram
 repeat
  repeat
    w;
    writeln ('Beolvasás........I');
    writeln ('Listázás……...L');
    writeln ('Kilépés...........E');
    readln (c);
    c := upcase(c);
  until c in ['I','L','E'];              // Csak ezt a 3 karaktert fogadja el
  case c of
    'I': datain (cars,n);
    'L': list (cars,n);
    'E': exit;
  end;
 until c = 'E';
end.
```

Programunk futási képe az alábbi.

```
D:\Oktatás\Jegyzet példák\pl_record_carmarket\Project2.exe
Beolvasás.....I
Listázás......L
Kilépés.......E
i
Autók száma : 3
Típus     : Skoda 120
Szín      : Kék
ár (EUR)  : 250
Típus     : BMW
Szín      : Ezüst
ár (EUR)  : 6000
Típus     : Toyota
Szín      : Fekete
ár (EUR)  : 3500

Beolvasás.....I
Listázás......L
Kilépés.......E
l

   Típus          Szín       ár (EUR)
 ------------------------------------
 Skoda 120          Kék          250
       BMW        Ezüst         6000
    Toyota       Fekete         3500

Beolvasás.....I
Listázás......L
Kilépés.......E
E_
```

99. ábra: A használtautó nyilvántartó példaprogram képernyőképe

6.10 Példák függvény használatára

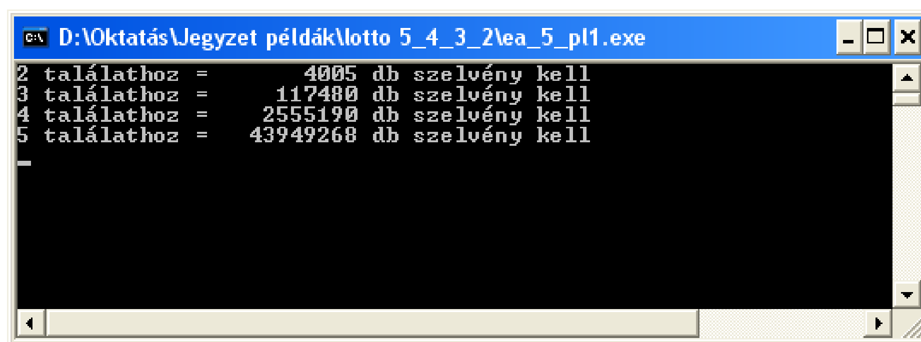
P15_1. Készítsünk faktoriális számító függvényt. Alkalmazzuk függvényünket ötös lottó esetében annak kiszámítására, hogy az 5,4,3 és 2 találat eléréséhez hány tippszelvényt szükséges szisztematikusan kitölteni. Ezeket a számokat a már megismert (Pascal háromszög példánál) binomiális együtthatók adják rendre:

$$\binom{90}{5} \quad \binom{90}{4} \quad \binom{90}{3} \quad \binom{90}{2}$$

```
// Lotto 5 Só Lilla 2007.05.16.
program faktorial;

function fak (k : byte) : extended;
  var f : extended;
      i : byte;
  begin
   f := 1;
   if k > 1 then
   for i:=1 to k do f := f * i;
   fak := f;
  end;
```

```
  var i: byte;

begin
  for i:=2 to 5 do
   writeln(i,'találathoz = ',fak(90)/fak(90-i)/fak(i):10:0,' db szelvény kell');
  readln;
end.
```

Programunk futási képe az alábbi.



100. ábra: A faktoriális számító példaprogram képernyőképe

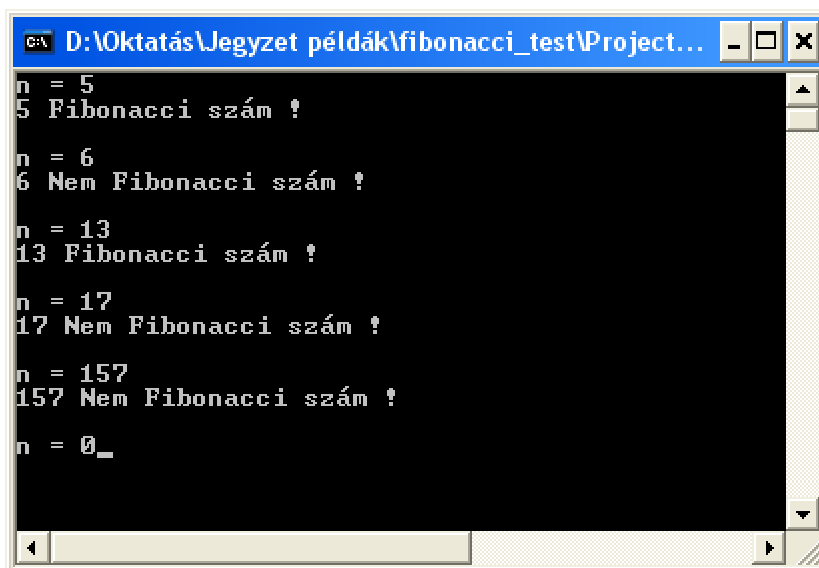
P1_2. A fibonacci számokat az alábbi sorozat elemei alkotják: 1, 1, 2, 3, 5, 8, 13, 21, stb. Képletben $Fib_i = fib_{i-1} + fib_{i+2}$. Készítsünk programot, amely egy beolvasott egészszámról eldönti, hogy eleget tesz-e a fibonacci kritériumnak. A vizsgálatot egy logikai függvénnyel oldjuk meg.

```
// Fibonacci  teszt  K.L. 1999.04.22.
program fibo;

var n: word;

function fib_test(x : word) : boolean;
 var f1, f2, f, i : word;
 begin
  if x > 2 then
   begin
     f1 := 1; f2 := 1; f := 2;
     repeat
      f := f1 + f2;
      f1 := f2;
      f2 := f;
    until f  >= x;
   end;
  fib_test := (x = f) or (x = 2);
 end;
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
begin
 repeat
  write ('n = '); readln (n);
  if fib_test (n) then writeln (n,' Fibonacci szám !')
                  else writeln (n,' Nem Fibonacci szám !');
  writeln;
 until n=0;
 readln;
end.
```

Programunk futási képe az alábbi.



101. ábra: A Fibonacci ellenőrző példaprogram képernyőképe

P15_3. Készítsünk programot, amely egy egészszám jegyeit adogatja össze mindaddig, amíg egyjegyű szám adódik. A megoldáshoz alkalmazzunk függvényt.

```
// Jegyadogató  M.S. 1999.06.21.

program figures;

type t_fig = 0..9;

function adogat(x : word): t_fig;
```

```
var   j : t_fig;
   sum : word;
begin
  repeat
   sum:=0;
   while x > 0 do
     begin
       j := x mod 10;                    // egyesek
      sum:=sum + j;
      x := x div 10;
    end;
   x := sum;                              // újra, ha az összeg > 9
  until sum < 10;
 adogat:=sum;
end;

var n : word;

begin
 repeat
  write ('n= '); readln (n);
  writeln (n,' jegyeinek összege : ', adogat(n) );
 until n = 0;
 readln;
end.
```

Programunk futási képe az alábbi.

```
D:\Oktatás\Jegyzet\Jegyzet példák\pl_func_adogat\Project...
n= 12
12 jegyeinek összege : 3
n= 89
89 jegyeinek összege : 8
n= 6593
6593 jegyeinek összege : 5
n= 0_
```

102. ábra: A számjegyeket összeadó példaprogram képernyőképe

6.11 Példák eljárás használatára

Eljárást akkor deklarálunk, ha a szegmensünknek nincs jól definiált visszatérési értéke. Utasításcsoportokat fogunk össze ily módon, ezzel programunk jól tagolttá, áttekinthetővé válik.

P16_1. Készítsünk programot, amellyel 2 mártix összegét, különbségét és szorzatát számíthatjuk ki. Az eredmények mátrix alakban való kiírásához deklaráljunk eljárást. Az egyszerűség kedvéért példánkban a mátrixok elemeit szintén eljárással generáljuk.

Két mátrix összege (illetve különbsége): $c_{i,j} = a_{i,j} \pm b_{i,j}$, ahol i = 1..n és j = 1..n. A két mátrix mérete megegyező kell legyen.

Kész mátrix szorzata a[n,m] és b[m,k] esetén $c_{i,j} = \Sigma\, a_i{*}b_i$ (a i.-k sorának és b i.-ik oszlopának skalár szorzata. Az a mátrix sorszáma kötelezően megegyezik b mátrix oszlopszámával.

```
// Mátrix algabra  H.S. 2005.06.21.
program matrix;

type t_matr = array [1..10,1..10] of single;
var i, j, k, n1, n2, m1, m2 : byte;
    ch     : char;
    a,b,c : t_matr;
    sum  : single;

procedure print (n,m : byte; x : t_matr);
 var i,j : byte;
 begin
  for i:=1 to n do
   begin
    writeln;
    for j:= 1 to m do write (x[i,j]:8:2);
   end;
  writeln; writeln;
 end;

procedure general(n,m : byte; var x : t_matr);     // x kimenő formális paraméter
 var i,j : byte;
 begin
  for i:=1 to n do
   for j:= 1 to m do x[i,j] := 10* random;  // 0-9.9
 end;


begin                                       // Főprogram
 randomize;
 write('n1= '); readln(n1);
 write('m1= '); readln(m1);
 general (n1,m1,a);                         // a matrix
 print (n1,m1,a);
 write ('n2= '); readln (n2);
```

```
write ('m2= '); readln (m2);
general (n2,m2,b);                    // b matrix
print (n2,m2,b);
write ('Összeadás (+)   Kivonás (-)    Szorzás (*) :');
readln (ch);
if ch in [ '+', '-', '*' ] then
 case ch of
  '+' : begin
          if (n1 = n2) and (m1 = m2) then
           begin
            for i:=1 to n1 do
              for j:= 1 to m1 do c[i,j] := a[i,j] + b[i,j];
            print (n1,m1,c);
           end
          else writeln ('Hibás méret !');
        end;
  '-' : begin
         if (n1 = n2) and (m1 = m2) then
          begin
           for i:=1 to n1 do
             for j:= 1 to m1 do c[i,j] := a[i,j] - b[i,j];
           print (n1,m1,c);
          end
        else writeln (Hibás méret!');
       end;
  '*' : begin
         if (m1 = n2) then
          begin
           for i:=1 to n1 do
             for j:= 1 to m2 do
               begin
                 sum:=0;
                 for k:=1 to m1 do sum := sum +  a[i,k] * b[k,j];
                 c[i,j] := sum;
               end;
            print(n1,m2,c);
          end
         else writeln ('Hibás méret!');
       end;
  end
else writeln('Hibás parancs !');
readln;
end.
```

Az egyes futási képek a különböző esetekre az alábbiak.

```
D:\Oktatás\Jegyzet példák\pl_eljárás_matrix\Project2.exe
n1= 3
m1= 4

  9.62  8.84  0.64  7.82
  6.12  9.37  5.37  1.24
  8.61  2.04  5.46  4.16

n2= 3
m2= 4

  0.05  4.65  2.79  8.18
  6.27  3.57  2.63  4.16
  2.26  4.43  4.04  9.01

összeadás (+) Kivonás (-) Szorzás (*) :-

  9.57  4.19 -2.15 -0.36
 -0.15  5.81  2.74 -2.93
  6.36 -2.39  1.42 -4.85
```

103. ábra: A mátrix kivonás eredménye

```
D:\Oktatás\Jegyzet példák\pl_eljárás_matrix\Project2.exe
n1= 3
m1= 4

    5.10    6.98    1.24    1.12
    7.96    4.30    3.74    7.77
    5.11    5.39    5.75    9.96

n2= 4
m2= 2

    2.56    4.47
    6.44    1.00
    1.78    0.76
    9.14    1.36

összeadás (+) Kivonás (-) Szorzás (*) :*

   70.49   32.22
  125.88   53.28
  149.13   46.12
```

104. ábra: A mátrix szorzás eredménye

```
D:\Oktatás\Jegyzet példák\pl_eljárás_matrix\Project2.exe
n1= 3
m1= 3

  7.94  1.61  9.38
  1.27  7.98  0.76
  8.35  5.07  6.33

n2= 3
m2= 3

  2.45  1.42  8.64
  9.96  3.15  6.75
  8.69  9.62  0.56

összeadás (+) Kivonás (-) Szorzás (*) :/
Hibás parancs !
```

105. ábra: A hibás parancs eredménye

```
D:\Oktatás\Jegyzet példák\pl_eljárás_matrix\Project2.exe
n1= 3
m1= 4

  4.83  8.58  3.36  6.71
  9.63  2.78  2.53  9.39
  7.62  7.89  4.29  1.84

n2= 3
m2= 5

  0.47  7.46  9.99  5.80  9.72
  0.78  1.74  8.12  8.09  1.56
  5.46  2.52  2.71  5.50  7.98

összeadás (+) Kivonás (-) Szorzás (*) :+
Hibás méret !
```

106. ábra: A hibás méret eredménye

6.12 Halmazok gyakorlása

Számos faladat van, amelynek megoldása halmaz típusú változó felhasználásával egyszerűen és látványosan megoldható. Nézzünk erre egy példát.

P17_1. Egy szállodában 20 szoba van:

- fürdőszobás az 1, 2, 5, 6, 7, 8, 15, 18, 19, 20 számú,
- parkra néz az 1-8 és 13-15,
- erkélyes az 1,2, 4-8, 14, 15,
- nincs szomszédja az 1, 7, 8,

– egy szomszédja a 2, 6, 9, 15, 16, 20-nak,
– 2 szomszédja a 3, 4, 5, 10-14, 17, 18,19-nek van.

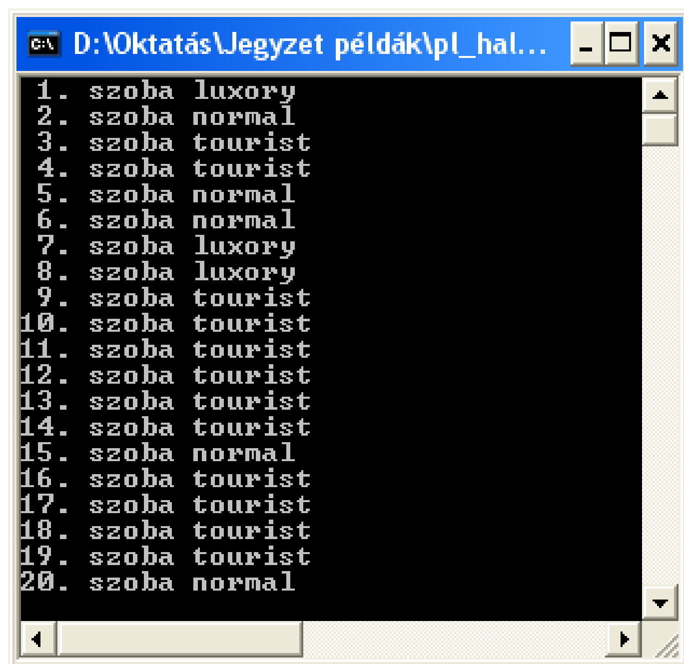
Sorolja be három kategóriába a szobákat az alábbi feltételek szerint:

– Luxus: erkélyes, fürdőszobás, parkra néz és nincs szomszédja.
– Normál: fürdőszobás és 1 szomszédja van, vagy erkélyes, fürdőszobás és 2 szomszédja van, vagy parkra néz, fürdőszobás és 2 szomszédja van.
– Turista: a többi szoba.

Írjunk programot, amely kilistázza a 20 szoba besorolását. A feladat igen bonyolultnak tűnik, főképpen, ha feltételes utasítások tömegével akarnánk megoldani. Ha azonban halmaz típusokkal dolgozunk, sokkal egyszerűbb a megoldás.

```
// Hotelszobák  I.G. 2009.03.21.
program hotel;

type t_szobak = 1..20;                  // A szállodában 20 szoba van
     t_komf   = set of t_szobak;        // Komfort típus
     st8      = string[8];
var  i          : byte;
     asznr      : t_szobak;             // Aktuális szoba szám
     szom0, szom1, szom2,               // 0..2 szomszédja van
     fszob,                             // fürdőszobásak
     pszob,                             // parkra néző
     eszob      : t_komf;               // erkélyes
     luxory,  normal, tourist : t_komf; // kategóriák

begin
 szom0 := [1,7,8];
 szom1 := [2,6,9,15..16,20];
 szom2 := [3..5,10..14,17..19];
 fszob   := [1,2,5..8,15,18..20];
 pszob  := [1..8,13..15];
 eszob  := [1,2,4..8,14,15];
 luxory := szom0 * fszob * pszob * eszob;
 normal:= szom1*fszob + szom2*fszob*eszob + szom2*fszob*pszob;
 tourist := [1..20] - luxory - normal;   // Az összesből levonjuk a számított szo//bákat
 for i:=1 to 20 do                       // Szobák besorolásának listázása
  begin
   write (i:2,'. szoba ');
    if i in luxory then writeln ('luxory');
     if i in normal then writeln ('normal');
     if i in tourist then writeln ('tourist');
  end;
  readln;
end.
```

Programunk futási képe az alábbi.



107. ábra: A szállodai példaprogram képernyőképe

6.13 File kezelés gyakorlása

Nagy súlyt fektetünk a file kezelés gyakorlására. Ezért típusonként adunk egy-egy mintapéldát.

6.13.1 Skalár típusú file (egész, valós, karakter)

A leggyakrabban ezt a file típust használjuk.

P18_1. Készítsünk programot, amely egy beolvasott egészszámot (max. 3 jegyű) prímtényezőire bont, majd az eredeti számot, és annak prímtényezőit fileba menti, majd a lemezről felolvasva a képernyőre listázza azokat. A szóba jöhető prímszámokat egy másik program generálta, jelen programunkban konstans tömbként definiáljuk.

 © Aradi, Bécsi, Gyenes, Péter, BME

```
D:\Oktatás\Számtech1\Gyakorlat\prim_list\Project2.exe
A= 1
B= 1000
   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67  71
  73  79  83  89  97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173
 179 181 191 193 197 199 211 223 227 229_
```

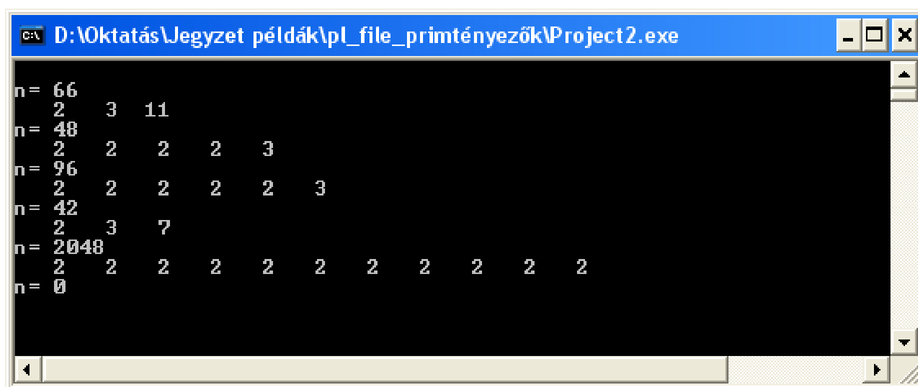
108. ábra: Prímgeneráló programunk futási képe

Jelen programunk utasítás listája az alábbi.

```
// Prímtényezők  S.A. 2010.03.20.
program integerfile;

const primek : array[1..12] of integer = (2,3,5,7,11,13,17,19,23,29,31,37);
var         n,i : integer;
             f   : file of integer;
begin
 assign (f, 'primtényezők.dat');         // File név bemutatása
 rewrite(f);                             // File megnyitása írásra
 write ('n= '); readln (n);
 write (f,n);                            // Eredeti szám felírása a file-ba
 i:=1;
 while (n > 0) and (i < 12) do           // Prímtényezők keresése
  begin
   if n mod primek[i] = 0 then           // Talált egy prímet
    begin
     write ( f, primek [i] );            // Prímtényező felírása a file-ba
     n := n div primek [i];              // A szám leosztása
    end
   else inc (i);                         // Következő prímszám
  end;
  close(f);                              // File bezárása
  reset(f);                              // File megnyitása olvasásra
  read (f,n);                            // Eredeti szám
  writeln;
  write(n,'= ');                         // Kiírása
  while not eof (f) do                   // File végéig
   begin
    read (f,n);                          // Prímtényező olvasása
    write (n:4);                         // Prímtényező kiírása  4 helyre
   end;
   close (f);                            // File bezárása
  readln;
end.
```

© Aradi, Bécsi, Gyenes, Péter, BME

Programunk futási képe az alábbi.



109. ábra: A skalár típusú file-t kezelő példaprogram képernyőképe

6.13.2 Tömb és rekord típusú file

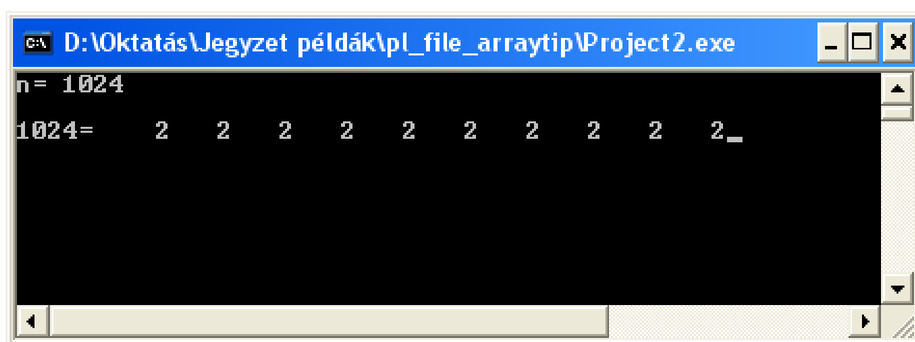
Tömb típusú file egy-egy rekordja nem egyetlen adatot, hanem egy azonos típusú elemekből álló tömböt tartalmaz.

P19_1. Módosítsuk előző feladatbeli programunkat oly módon, hogy az n számhoz tartozó prímtényezők egy rekordban legyenek elmentve.

```
// Prímtényezők  S.A. 2010.03.20.
program arrayfile;

const primek : array[1..12] of integer = (2,3,5,7,11,13,17,19,23,29,31,37);
type    t_vec = array[1..10] of integer;
var          n,i : integer;
       primvec : t_vec;                     // Prímtényezők tömbje
             f   : file of t_vec;
begin
 assign (f, 'primtényezők.dat');            // File név bemutatása
 rewrite(f);                                // File megnyitása írásra
 write ('n= '); readln (n);
 primvec[1] := n;                           // Szám tárolása a vektor 1. elemeként
 i:=2;
 while (n > 0) and (i < 12) do              // Prímtényezők keresése
  begin
   if n mod primek[i] = 0 then              // Talált egy prímet
    begin
     primvec[1] := primek [i];             // Prímtényező tárolása a vektor i. elemeként
     n := n div primek [i];                 // A szám leosztása
    end
   else inc (i);                            // Következő prímszám
```

```
    end;

    for j:=2 to i do write ( f, primvec [i] );   // Prímtényezők felírása a file-ba
    close(f);                                    // File bezárása
    reset(f);                                    // File megnyitása olvasásra
    read (f,n);                                  // Eredeti szám
    writeln;
    write(n,'= ');                               // Kiírása
    while not eof (f) do                         // File végéig
     begin
      read (f,n);                                // Prímtényező olvasása
      write (n:4);                               // Prímtényező kiírása  4 helyre
     end;
     close (f);                                  // File bezárása
    readln;
  end.
```

Programunk futási képe az alábbi.



110. ábra: A rekord típusú file-t kezelő példaprogram képernyőképe

Gyakorta mentünk file-ba rekord típusú adatokat. Az adatfeldolgozó-nyilvántartó programok nagy része megkívánja ezt a technikát. Alábbi programunk repülőjárat utasainak kockázati faktorát számítja (ez a szám az utasbiztosítási díj alapja lehet).

P19_2. Elsőként definiálunk *t_man* néven rekord típust az alábbi mezőkkel:

nam (string), sex (boolean), age (byte), risk (single)

Másodjára deklarálunk *make* nevű eljárást, amely létrehoz egy fenti típusú állományt (a file nevét paraméterben veszi át) billentyűről beolvasott adatokból (a rizikó faktor értéke 0 legyen).

Harmadszor deklarálunk *list* nevű függvényt, amely paraméterben átveszi az utasok állományának nevét, a képernyőre listázza azok adatait, alkalmas típusú paraméterben átadja az utasok rekordját, nevében pedig a file rekordjainak számát.

Végül a program számítja és listázza az utasok nevét és az alábbi táblázat alapján számolt rizikófaktorok értékét.

Életkor	Férfi	Nő
< 30	0.05	0.06
31-40	0.1	0.15
41-50	0.3	0.35
51-60	0.4	0.45
> 60	0.5	0.6

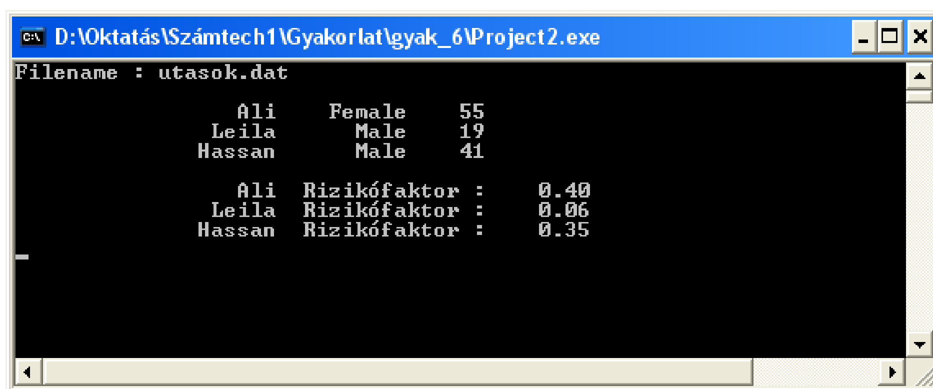
26. táblázat: Rizikófaktor táblázat

```
// Repülő utasbiztosítás HB 2008.11.11.
program riziko;

type    t_st20   = string [20];
t_man  = record
                  nam : t_st20 ;
                  sex  : boolean;
                  age  : byte;
                  risk : single;
              end;
         t_utasok = array[1..20] of t_man;

var    f         : file of  t_man;
       fname    : t_st20;
       utasok   : t_utasok;
       utasnr, i : byte;

procedure make (fnev:t_st20);               // Létrehozza a rekord típusú file-t
 var  fn : byte;
        c  : char;
    emb : t_man;
 begin
  assign (f, fnev);
```

```
  rewrite(f);
  repeat
   with emb do
    begin
     write ('Utas neve          :'); readln (nam);
     write('Neme 1:ffi / 2:nő  :'); readln (fn);       // Beolvasás számként
     sex := fn = 1;                               // Logikai mező érétket kap
     write ('Utas kora          :'); readln(age);
     risk:=0;
    end;
   write (f, emb);                                // 1 utas mentése file-ba
   write ('Tovább (i/n) ? '); readln (c);
  until c in ['n','N'];                           // n-re vége a beolvasásnak
  close (f);
  writeln ('File has written !');
 end;

// Felolvassa és listázza az utasok adatait
function list (fnev : t_st20; var u:t_utasok):byte;
 var i : byte;
 begin
  assign (f,fnev);
  reset (f);
  list := filesize (f);                           // File méret
  i := 1;
  repeat
   read (f,u[i]);
   with u[i] do
    begin
     write (nam:20);
     if sex then write('Female':10) else write('Male':10);
     writeln(age:6);
    end;
   inc(i);
  until eof (f);
  writeln;
  close(f);
 end;

begin                                             // Főprogram
 write ('Filename : '); readln(fname);            // File nevének bekérése
 make (fname);                                    // File készítése
 utasnr := list (fname,utasok);                   // Listázó fv. hívása
 for i:=1 to utasnr do
  with utasok[i] do
   begin
    write (nam:20);
    case age of                                   // rizikó számítása
      1..29: if sex then risk:=0.05 else risk:=0.06;
```

```
        30..39: if sex then risk:=0.1 else risk:=0.15;
        40..49: if sex then risk:=0.3 else risk:=0.35;
        50..59: if sex then risk:=0.4 else risk:=0.45;
        60..90: if sex then risk:=0.5 else risk:=0.6;
      end;
      writeln ('  Rizikófaktor : ',risk:7:2);
    end;
  readln;
end.
```

Programunk futási képe az alábbi.



111. ábra: A rizikófaktort számoló példaprogram képernyőképe

6.13.3 Random file kezelés

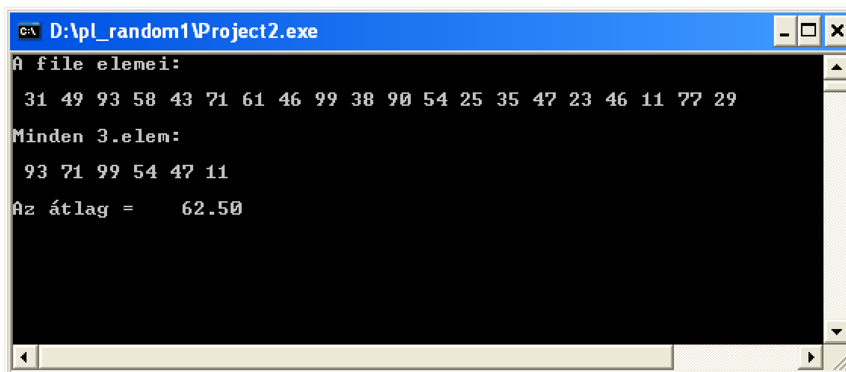
A random file kezelés lényege az, hogy eltérően a soros (szekvenciális) file kezeléstől az egyes rekordok elérése közvetlenül történik. A már megismert file pointer a file bármely rekordjára irányítható a **seek**(f,nr) standard eljárás segítségével. A random file kezelés fokozott figyelmet kíván meg a program készítőjétől, mert a hibás pointer állítás (és azt követő írás parancs) kárt tehet más állományokban.

P20_1. Készítsünk programot, amely egy egésztípusú file minden harmadik elemének számtani átlagát adja meg. Az adatállomány 20 db 10 és 99 közötti véletlen számból álljon.

```
// Random1 T.A. 2007.01.25.
program seekfile;
var                                   i,n : byte;
        sum : word;
          f   : file of byte;
begin
 randomize;
```

```
assign (f,' integers.dat');
rewrite(f);
for i:=1 to 20 do
 begin
  n:= random (90) + 10;                // 10..99
  write (f,n);                         // file létrehozása
 end;
close (f);

reset(f);                              // file megnyitás, pointer=0
writeln ('A file elemei:'); writeln;
for i:= 1 to 20 do
 begin
  read (f,n);
  write(n:3);
 end;
writeln;
sum:=0;
seek(f,0);                             // pointer = 0
i:=2;
writeln;writeln ('Minden 3.elem:'); writeln;
repeat
 seek (f,i);                           // pointer minden 3.elemre mutat
 read (f,n);                           // elem felolvasása
 write (n:3);
 sum := sum + n;
 inc (i,3);
until i > 19;
writeln;
writeln;writeln ('Az átlag = ', sum/6:8:2);
readln;
end.
```

Programunk futási képe az alábbi.



112. ábra: A random file kezelő példaprogram képernyőképe

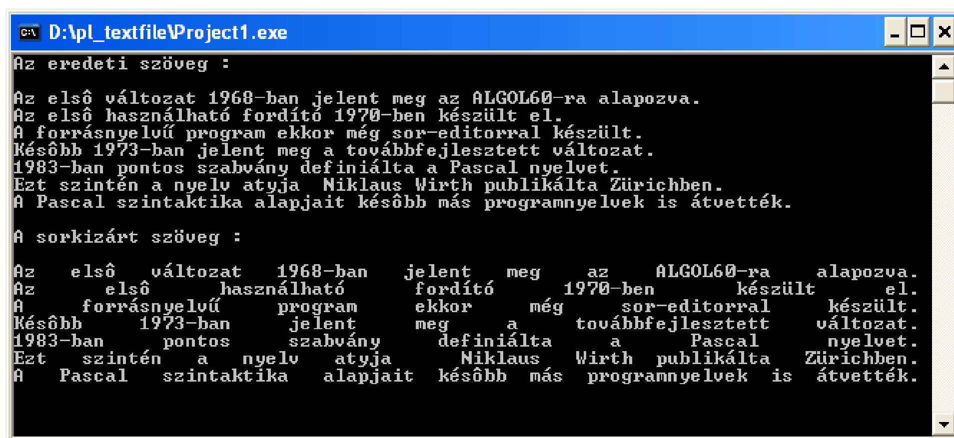
6.13.4 Szöveg file kezelése

Készítsünk programot, amely egy szövegfile sorait szóközökkel 80 karakteresre egészíti ki.

```
// Editálás sorkizárással. P.T. 2000.05.10.
program textfile;
type t_st80 = string [80];
var      f  : text;
        sor : t_st80;

procedure sor80 (var s : t_st80);            // 1 sort kiegészít
 var spnr,len,ins,i,j : byte;
                sorjav : t_st80;
 begin
  sorjav := '';
  len := length (sor);                         // A sor hossza
  spn r := 0;
  for i:=1 to len do                           // Szóköz számlálása
   if sor[i] = ' ' then inc (spnr);
  ins  := round ((80-len) / spnr);             // Beszúrandó sp. száma
  for i:=1 to len do                           // Szóközök beszúrása
   begin
    sorjav :=s orjav + sor[i];                 // Másolás sp-ig
    if sor[i] = ' ' then
     for j:=1 to ins do sorjav := sorjav + ' '; // Beszúrás
   end;
  sor := sorjav;
 end;

begin
 assign(f, 'pascal.txt');
 reset (f);
 writeln ('Az eredeti sz'#148'veg :'); writeln;
 while not eof (f) do
  begin
   readln (f,sor);
   writeln (sor);
  end;
 reset(f);                                     // Újra a file elejére
 writeln;writeln('A sorkizárt szöveg :'); writeln;
 repeat
  readln (f,sor);
  sor80 (sor);
  writeln(sor);
 until eof (f);
 close(f);
 readln;
end.
```

 © Aradi, Bécsi, Gyenes, Péter, BME

Programunk futási képe az alábbi.



113. ábra: A szöveges file kezelő példaprogram képernyőképe

6.13.5 Típus nélküli file kezelése

Ezzel a technikával bármely file kezelhető. Például a tömörítő programok amelyek a Huffmann-Maley algoritmust használják.

6.14 Pointerek használata

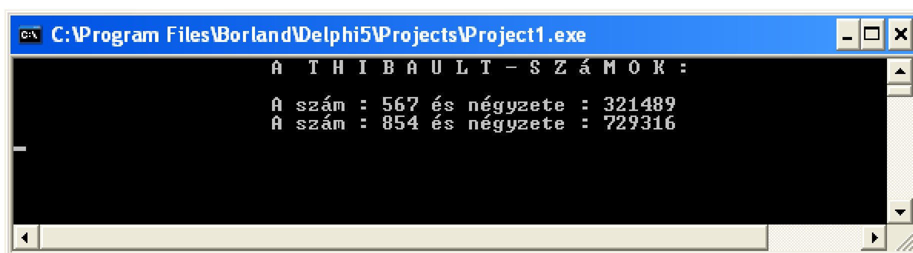
A dinamikus adatok használatával nagyméretű adatmezők kezelése válik lehetővé. Az alábbiakban különböző típusú pointerek használatát mutatjuk be.

P22_1. Első példánkban skalár típusú pointert használunk. Számítsuk ki a Thibault-féle számokat, amelyek négyzetükkel együtt mind a 9 számjegyet felhasználják. Ezek csak 3 jegyű számok lehetnek. A Thibault-féle számhoz **word** típusú pointert, míg négyzetéhez **longword** típusú pointert használunk.

```
// Thibault szám keresés K.K. 1999.11.05
program Thibault;

type  t_num = '1'..'9';
      t_halm = set of t_num;

var     i      : byte;
        j      : word;
        n      : ^word;              // Pointer
        nn     : ^longword;          // Pointer
          th   : t_halm;
      TH_str   : string[9];
        n_st   : string[3];
```

```
        nn_st   : string[6];
begin
 WriteLn(' A  T H I B A U L T - S Z Á M O K : ');
 WriteLn;
 new (n);                                // Konténer a Thibault számnak
 new (nn);                               // Konténer a Thibault szám négyzetének
 for j:=317 to 987 do                    // Csak ebben az intervallumban lehet
  begin
   TH_str:='        ';
   n^:=j;
   nn^:= n^ * n^;
   Str(n^,n_st);                         // String konverzió
   Str(nn^,nn_st);                       // String konverzió
   TH_str := n_st + nn_st;               // A két string összefűzése
   th :=[];                              // Halmaz nullázása
   for i:=1 to 9 do th := th + [TH_str[i]];   // Halmaz feltöltése
   if th = ['1'..'9'] then               // Ha minden szám benne van, akkor //Thibault
    writeLn('A szám : ', n^, ' és négyzete : ', nn^);
  end;
 readln;
end.
```

Programunk futási képe az alábbi.



114. ábra: A Thibault számokat bemutató program képernyőképe

P22_2. Rekord típusú pointer használata. Készítsünk programot telefonregiszter kezelésére. A program tegye lehetővé új név bevitelét, listázást és név szerint való keresést. Az egyes bejegyzések dinamikus konténerben legyenek elhelyez-ve.

```
// Telefonregiszter H.S. 200.04.15.
program tel_book;

type namptr =  ^nam;                     // rekord pointer típus
       nam    = record
                  neve : string[30];
                 szam : string[20];
                end;
```

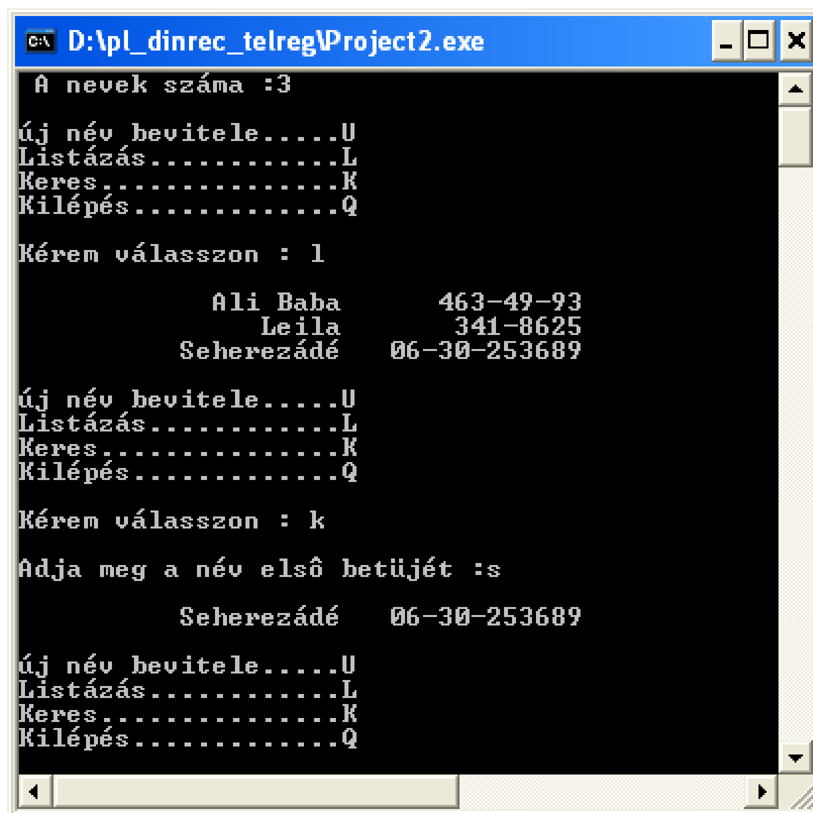
 © Aradi, Bécsi, Gyenes, Péter, BME

```
var    nevek  : array [1..100] of namptr;    // rekord pointerek tömbje
       len, i : byte;
       f      : file of nam;
       ch     : char;

procedure hozzaad;
 var i: byte;
     ad: nam;
 begin
  write('Név      :'); readln (ad.neve);
  write ('Tel. nr  :'); readln (ad.szam);
  reset (f);
  seek (f,len);
  write (f,ad);
  close (f);
  inc (len);
  nevek[len]^ := ad;
 end;

procedure lista;
 var i:byte;
 begin
  writeln;
  for i:=1 to len do writeln (nevek[i].neve:20,nevek[i].szam:15);
 end;

procedure keres;
 var i:byte;
 begin
  writeln;
  write ('Adja meg a név első betüjét :');
  readln (ch);
  writeln;
  for i:=1 to len do
   if upcase (ch) = nevek[i].neve[1] then
    writeln (nevek[i].neve:20,nevek[i].szam:15);
 end;

begin                                   // Főprogram
 assign (f,'telnr.dat');
 {$I-}                                  // IO ellenőrzés kikapcsolása
 reset (f);
 {$I+}                                  // IO ellenőrzés bekapcsolása
 if IOResult = 0 then
  begin                                 // File már létezik;
   len:= filesize(f);                   // nevek száma
   for i:=1 to len do
    begin
```

```
    new(nevek[i]);                    // Dinamikus konténer létrehozása
    read ( f, nevek[i]^ );
   end;
  close (f);
 end
else                                  // File még nem létezik
 rewrite (f);                         // Létre kell hozni
writeln (' A nevek száma :', len);
repeat
 writeln;
 writeln (Új név bevitele.....U');
 writeln ('Listázás............L');
 writeln ('Keres...............K');
 writeln ('Kilépés.............Q'); writeln;
 repeat
  write ('Kérem válasszon : ');
  readln(ch);
 until upcase (ch) in ['U','L','K','Q'];
 case upcase (ch) of
  'U': hozzaad;
  'L': lista;
  'K': keres;
  'Q': Exit;
 end;
until upcase (ch)='K';

readln;
end.
```

 © Aradi, Bécsi, Gyenes, Péter, BME

Programunk futási képe az alábbi.



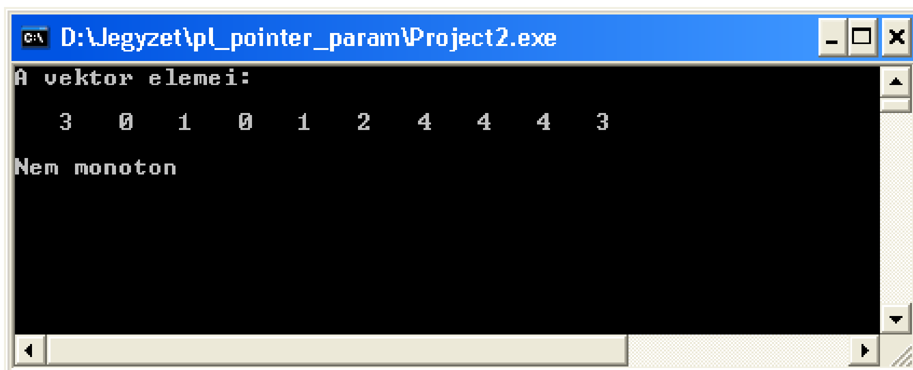
115. ábra: A telefonregiszter példaprogram képernyőképe

P22_3. Paraméterátadás pointer típussal. Az eljárások és függvények formális paramétereit érték szerinti paraméter átadásnál (nincs előtte var kulcsszó) a Pascal a stack szegmensben egy másolati példányként kezeli. Ha a paraméter egy, vagy több tömb (array), vagy összetett típusú (record), akkor hamar megtelik a stack memória (nagysága a Pascal környezetben 64 Kbyte). Ezért célszerű a vektorokra mutató pointert (aminek a helyfoglalása csak 4 byte) használni paraméterként. Megjegyezzük, hogy a C nyelv csak ezt a pointeres paraméter átadást engedi. Készítsünk logikai függvényt, amely egy vektorról eldönti, hogy elemei szigorúan monoton növekvők-e.

```
// Monoton B.K. 1998.11.11.
program point_param;
type t_vec = array of byte;
     t_ptr =  ^t_vec;                    // Tömb pointer típus
```

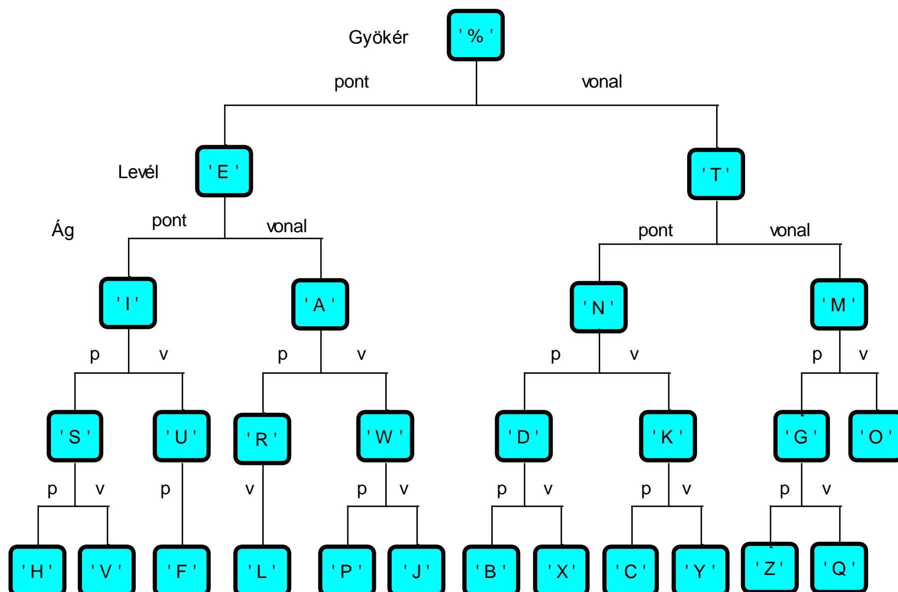
```
var     ptr : t_ptr;                          // Pointer változó
         i   : byte;

function monoton (p:t_ptr):boolean;
 var i:byte;
     yes:boolean;
 begin
  yes := true;
  for i:=1 to 9 do
   if p^[i] >= p^[i+1] then yes := false;     // Nem monoton
  monoton := yes;
 end;

begin
 randomize;
 new (ptr);                                   // Konténer
 Setlength (ptr^, 10);
 for i:=1 to 10 do ptr^[i] := random(5);      // 1..4
 writeln ('A vektor elemei:'); writeln;
 for i:=1 to 10 do write (ptr^[i]:4); writeln;writeln;        // Kiírás
 if monoton (ptr) then writeln ('Monoton')
                  else writeln ('Nem monoton');
 readln;
end.
```

Programunk futási képe az alábbi.



116. ábra: A vektor monotonitást vizsgáló program képernyőképe

P22_4. Morze dekódoló program. Mintaprogramunk egy morze üzenetet fogad /..-/---/.-/ alakban. Dinamikus fa struktúra felhasználásával gyors dekódolást valósít meg, és kiírja a dekódolt üzenetet. Először tervezzük meg a fa struktúrát. Ebben pont esetében balra, vonal esetében jobbra lépünk. A bejárt útvonal végén találjuk a keresett karaktert.



117. ábra: A morze dekódoló fa struktúra terve

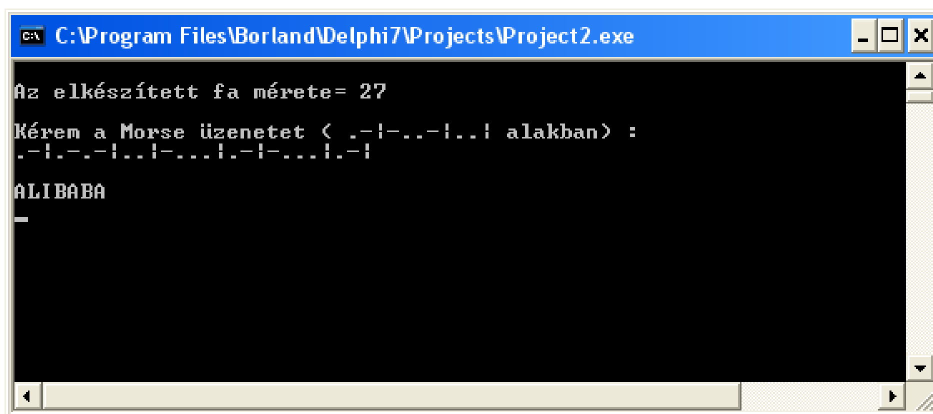
```
// Dinamikus fa struktúra
// GYK 2009.05.23.

program morze_dekoder;

type   t_msg  = string [80];
         t_ptr  = ^level;
       level    = record
                  betu  : char;
                  stock : byte;
                  pos    : byte;
                  pont,vonal   : t_ptr;
                end;

var     root     : t_ptr;                         // gyökér pointer
       adat      : char;
         msg     : t_msg;

// egy levelet készít 'leaf' pointerhez 'a' betüvel, 's' szintre 'p' oszlopba
procedure levelke(leaf:t_ptr; a:char; s, po:byte);
  begin
    with leaf^ do
      begin
        betu := a;
        stock := s;
        pos := po;
```

```
      pont := nil;
      vonal := nil;
    end;
  end;   { level }

 // A fil procedure kitölti a bináris fát a..z -ig 4 szinten
 // A gyökérre a 'root' mutat
procedure fill (var root : t_ptr);
 // ABC
 const   abc : array [1..26] of char = 'etianmsurwdkgohvflpjbxcyzq';
 var p, pb, pj, puj : t_ptr;
        i, j, index : byte;
        irany       : 0..1;              // A pont-vonal helyett egyszerűbb
        szint, po   : byte;
        adat,db     : byte;
 begin   { fill }
  szint := 1;                            // szint 1..4
  index := 0;                            // index 1..26
  repeat
   adat := 0;                            // adat minden szinten ujra indul
   db := round ( exp ( szint * ln(2) ) );   // db = 2^szint
    repeat
     inc (index);
     p := root;                          // keresés a gyökértől
     i := adat;
     for j:=1 to szint do
      begin
       irany := i and 1;
      i := i shr 1;
       if irany = 0 then                 // pont
             begin
              if p^.pont = nil then
                   begin
                    new (puj);
                    p^.pont := puj;
                    levelke( puj,abc[index],szint,col[index]);
                   end
                  else p := p^.pont;
             end
            else                         // vonal
             begin
              if p^.vonal = nil then
                    begin
                     new (puj);
                     p^.vonal := puj;
                     levelke (puj,abc[index],szint,col[index]);
                    end
                   else p := p^.vonal;
             end;
```

```
      end;
      inc(adat);                          // Következő betű
     until (adat = db) or (index = length(abc));      // 2^szint * szeresen

     inc (szint);                         // Következő szint
   until szint=5;                         // 4-szer
  end;  { fill}

// A rekurzív függvény megállapítja egy fa méretét
function sizefa ( root : t_ptr) : byte;
  begin
   if root = nil then sizefa:=0
                 else
                  // Magát hívja meg
                  sizefa := 1 + sizefa(root^.pont) + sizefa(root^.vonal);
  end;   { sizefa }

// Dekódolja a txt-ben megadott morze üzenetet
// Átveszi a gyökérpointert és a morse üzenetet
procedure dekod (p:t_ptr; txt:t_msg);
  var  i,j: byte;
       s  : string[4];                    // A pont-vonal sorozatot tárolja
  begin
    i:=1;
    repeat
     s:='';
     repeat                               // 1 betű
       s := s + txt [i];
       inc(i);
     until (txt[i] = '|') or (i>80);      // végjelig
     p := root;                           // pointer a gyökérre áll
     for j := length(s) downto 0 do       // változó hosszú
      begin
        if s[j] = '.' then p := p^.pont;
        if s[j] = '-' then p:=p^.vonal;
      end;
     write (p^.betu);
    until i = length (txt);
  end;    { dekod }

begin                                     // Főprogram
  new(root);                              // A fa gyökere 'root' pointernél
  levelke(root,'%',0,40);                 // elkészíti a gyökér csomópontot
  fill(root);
  writeln;
  writeln('Az elkészített fa mérete= ',sizefa (root));
  writeln('Kérem a morse üzenetet ( .-|-..-|..| alakban) :');
  readln(msg);
  dekod(root,msg);
```

```
  readln;
end.
```

A program futási képe az alábbi.



118. ábra: A morze dekódoló program képernyőképe

6.15 Objektumok készítése

Az objektum a legkomplexebb típus. Először egy egyszerű statikus objektumot mutatunk meg.

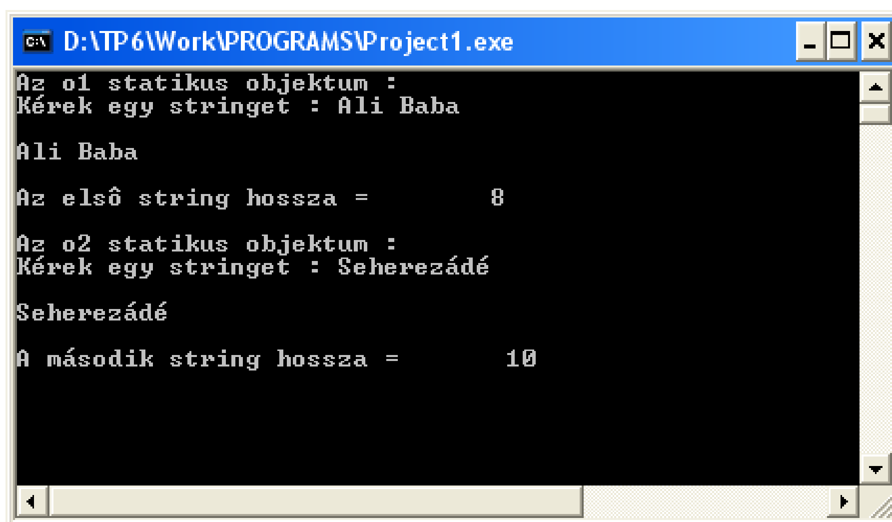
P23_1. Az objektum típus adatmezője egy **string**, metódusai pedig a **string** megadása, kiírása és hosszának számítására szolgálnak.

```
// Statikus objektum N.Sz. 2000,02.22.
program obj_stat;

type   st20    = string [20];
       obj_typ = object
                   s : st20;
                   procedure init;
                   procedure print;
                   function  hossz : byte;
                 end;

       procedure obj_typ.init;
        begin
         write ('Kérek egy stringet : '); readln (s);
        end;

       procedure obj_typ.print;
        begin
         writeln;
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
        writeln (s);
       end;

      function obj_typ.hossz : byte;
       begin
        hossz := length (s);
       end;

var     o1,o2 : obj_typ;                    // 2 objektum példány

begin
  writeln('Az o1 statikus objektum : ');
  with o1 do
   begin
    init;
    print;
    writeln;
    writeln ('Az első string hossza = ',o1.hossz:8);
   end;
   writeln;
   writeln ('Az o2 statikus objektum : ');
     o2.init;
     o2.print;
     writeln;
     writeln ('A második string hossza = ',o2.hossz:8);
   readln;
end.
```

A program futási képe az alábbi.



119. ábra: A string kezelő OOP példaprogram képernyőképe

6.15.1 Dinamikus objektum

P23_2. Most bemutatunk egy dinamikus objektumot használó programot. A program tanulók adatait olvassa be, majd név, és életkor szerint rendezve listázza azokat.

```
// Dinamikus objektum J.L. 2006,06.10.
program obj_d2;

const  max      =  50;                   // a tankör max. létszáma
type   st20     =  string [20];
       p_tan_typ = ^tanulo;              // 1 tanulóra mutató pointer típus
        tanulo  =  object
                    nev : st20;
                    age : byte;
                    constructor init (nev0:st20; age0:byte);
                   end;

       constructor tanulo.init (nev0:st20; age0:byte);
        begin
         nev := nev0;
         age := age0;
        end;

type  p_tk_typ = ^tankor;                // a tankör pointer típusa
       tankor  =  object
                    tk : array [1..max] of p_tan_typ;      // pointer tömb !
                    nr : byte;           // Tényleges tanuló szám
                    constructor init;
                    destructor done;
                    // Két tanuló felcserélése
                    procedure csere(var x,y : p_tan_typ);
                    procedure nevrend;
                    procedure korrend;
                    procedure print;
                   end;

       constructor tankor.init;          // Elkészíti a dinamikus tömböt
        var   i   : byte;
            nev0 : st20;
            age0 : byte;
        begin
        write ('A tanulók száma : '); readln (nr);
        for i:=1 to nr do
          begin
           write ('A hallgató neve : '); readln (nev0);
           write ('Kora             : '); readln (age0);
```

 © Aradi, Bécsi, Gyenes, Péter, BME

```
      tk[i] := new (p_tan_typ, init (nev0, age0) );        // Létrehozza a dinamikus konté-
                                                  nert
    end;
  end;

destructor tankor.done;                    // Felszabadítja a heap-ben a helyet
 var    i    : byte;
 begin
  for i:=1 to nr do
   dispose ( tk [i] );
 end;

procedure tankor.csere(var x,y : p_tan_typ);
 var  s     : st20;
       k    : byte;
 begin
  s := x^.nev;  k := x^.age;
 x^.nev := y^.nev;
 x^.age := y^.age;
 y^.nev := s; y^.age := k;
 end;

procedure tankor.nevrend;
 var    i,j : byte;
 begin
  for i:=1 to nr do
   for j:=1 to nr-1 do
    if tk[j]^.nev > tk[j+1]^.nev then csere (tk[j], tk[j+1]);
 end;

procedure tankor.korrend;
 var    i, j, k: byte;
          s    : st20;
 begin
  for i:=1 to nr do
   for j:=1 to nr-1 do
    if tk[j]^.age > tk[j+1]^.age then csere (tk[j], tk[j+1]);
 end;

procedure tankor.print;
 var    i,n : byte;
 begin
  for i:=1 to nr do
   begin
    n:= length ( tk[i]^.nev);
    writeln (tk[i]^.nev,' ':20-n, tk[i]^.age:3);
   end;
 end;
```

```
var   ptankor : p_tk_typ;                    // Tankör típusú pointer

begin
  ptankor:= new ( p_tk_typ,init );
  writeln; writeln ('Eredeti adatok : ');
  ptankor^.print;
  ptankor^.nevrend;
  writeln; writeln ('Név szerint rendezett adatok :');
  ptankor^.print; writeln;
  ptankor^.korrend;
  writeln ('Életkor szerint rendezett adatok :');
  ptankor^.print;
  dispose (ptankor,done);
  readln;
end.
```

Programunk futási képe az alábbi.

```
D:\Oktatás\Jegyzet példák\pl_din_obj\Project2.exe
A tanulók száma : 3
A hallgató neve : Soma
Kora            : 26
A hallgató neve : Leila
Kora            : 18
A hallgató neve : Hasszán
Kora            : 21

Eredeti adatok
Soma                 26
Leila                18
Hasszán              21

Név szerint rendezett adatok :
Hasszán              21
Leila                18
Soma                 26

Születési év szerint rendezett adatok :
Leila                18
Hasszán              21
Soma                 26
```

120. ábra: A tanulók adatait kezelő OOP program képernyőképe

6.15.2 Ős- és leszármazott objektum

Az objektumok öröklési tulajdonságának használatára mutatunk be egy példát.

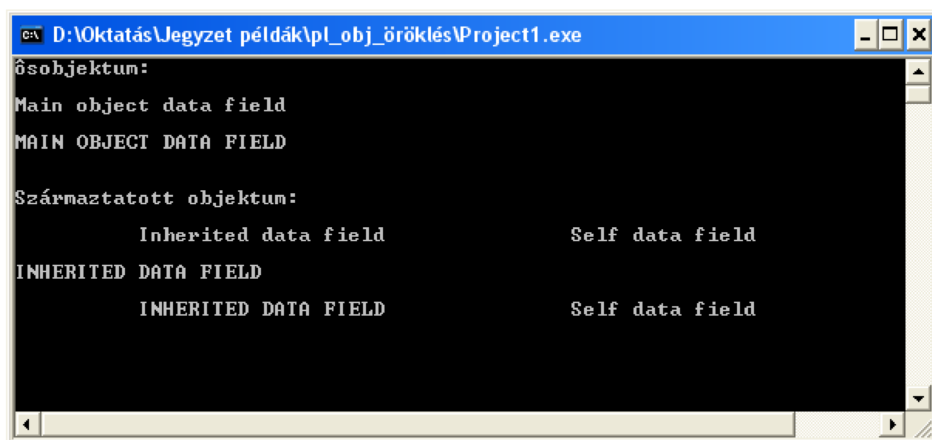
P23_3. Az alapobjektum egy adatmezőt (x : sting) és 2 metódust (print, upper) tartalmaz. A leszármazott objektum ezeket örökli, de saját adatmezővel (y) és saját print metódussal is rendelkezik. Ez utóbbi szándékosan azonos nevű, mint az örökölt print metódus. Kérdés, melyiket tudja a leszármazott objektum hasz-

nálni: az örököltet, a saját, vagy mindkettőt. Példaprogramunkban erre a kérdésre adjuk meg a választ.

```
// Objektum öröklés Gy.K. 2010.06.24.
type   st30      =  string [30];
       mainobjtip =  object
                      x : st30;
                      procedure print;
                      procedure upper;
                     end;

              procedure mainobjtip.print;
               begin
                writeln (x); writeln;          // Sor elejére ír
               end;

              procedure mainobjtip.upper;
               var i:byte;
               begin
                for i:=1 to length (x) do x[i] := upcase (x [i]);       // Nagybetűre vált
                print;                          // Saját metódus hívása
               end;

 type  sonobjtip  =  object (mainobjtip)
                      y : st30;
                      procedure print;
                     end;

              procedure sonobjtip.print;
               begin
                writeln (x:30, y:30);           // Sor közepére ír
                writeln;
               end;

 var pra : mainobjtip;
     prb : sonobjtip;

 begin
  writeln('Ősobjektum: '); writeln;
  pra.x := 'Main object data field';            // Ősobjektum adatmező
  pra.print;                                     // Kiír: Main object data field
  pra.upper;                                     // Kiír: MAIN OBJECT DATA FIELD
  writeln;writeln ('Származtatott objektum: '); writeln;
  with prb do
   begin
    x := 'Inherited data field';                // Örökölt adatmező
    y := 'Self data field';                     // Saját adatmező
    print;                                       // Saját print metódus
```

```
      upper;                    // Ősobjektum print metódusát használja
      print;                    // Saját print metódus
    end;
  readln;
end.
```

A futási kép az alábbi.. Figyeljük meg, hogy az ősobjektum print metódusa a képernyő bal szélére, míg a származtatott objektumé középre ír.



121. ábra: Az öröklődést bemutató OOP program képernyőképe

 © Aradi, Bécsi, Gyenes, Péter, BME

Ábrajegyzék

1. ábra: Osztó algoritmus 9

2. ábra: BCD kódtábla 21

3. ábra: Szorzás balra léptetéssel 28

4. ábra: Osztás ismételt kivonással 30

5. ábra: A bináris csatorna modellje 34

6. ábra: A páfrány levelét megrajzoló program képernyőképe 47

7. ábra: A gravitációt modellező program képernyőképe 48

8. ábra: A program indítását jelző szimbólum 50

9. ábra: Aritmetikai utasítást jelző szimbólum 50

10. ábra: Elágazást jelző szimbólum 50

11. ábra: Példa az adatbevitel jelölésére 50

12. ábra: Példa az adatkivitel jelölésére 51

13. ábra: Példa a nyilak használatára 51

14. ábra: Három szám maximumának meghatározása 52

15. ábra: Cos(x) Taylor-sorba fejtésének folyamatábrája 53

16. ábra: A hallgatók tanulmányi átlagát számító algoritmus folyamatábrája 55

17. ábra: A Console Application projekttípus kezdőképe 58

18. ábra: Az if-then utasítás folyamatábrája 74

19. ábra: Az if-then-else utasítás folyamatábrája 74

20. ábra: Példa az összetett feltételes utasításra 75

21. ábra: A case utasítás folyamatábrája 81

22. ábra: A teljes case utasítás folyamatábrája 82

23. ábra: Példa az adatbeolvasásra 89

24. ábra: Példa a formázott kiíratásra 89

25. ábra: A Pascal program szerkezete 91

26. ábra: A Pascal függvény szerkezete 91

27. ábra: A Pascal eljárás szerkezete 97

28. ábra: A beolvasó eljárás terve 98

29. ábra: A beolvasó program futási képe 99

30. ábra: A kiíró eljárás terve 100

31. ábra: A mátrix kiíró program képernyőképe 101

32. ábra: A „csere” eljárás terve 103

33. ábra: A „rendez” eljárás terve 103

34. ábra: A halmaz típusú változókat kezelő program képernyőképe 108

35. ábra: A „common” függvény terve 108

36. ábra: A közös betűket kereső program képernyőképe 109

37. ábra: A dinamikus adat felépítése 111

38. ábra: A p1 pointer inicializálás után 112

39. ábra: A p1:=p2; művelet eredménye 112

40. ábra: A p1^:=p2^; művelet eredménye 113

41. ábra: A p2^.next := p1; művelet eredménye 113

42. ábra: A p2^.nr := 234; művelet eredménye 113

 © Aradi, Bécsi, Gyenes, Péter, BME

43. ábra: Előreláncolt dinamikus vektor 114

44. ábra: Hátraláncolt dinamikus vektor 115

45. ábra: Kétszeresen láncolt dinamikus vektor 115

46. ábra: Egyszeresen láncolt dinamikus vektor 116

47. ábra: Dinamikus fa 116

48. ábra: A kettes számrendszerbe átváltó program képernyőképe 118

49. ábra: A típusos file szerkezet 121

50. ábra: A beolvasó eljárás terve 122

51. ábra: A file felíró eljárás terve 123

52. ábra: A file olvasó eljárás terve 124

53. ábra: A kiíró eljárás terve 124

54. ábra: Az egész típusú file-t kezelő program futási képe 125

55. ábra: Az record típusú file-ba író program futási képe 127

56. ábra: A workers.dat file tulajdonságai 128

57. ábra: Az record típusú file-ból olvasó program futási képe 130

58. ábra: Az text file szerkezete 131

59. ábra: A file olvasó eljárás terve 132

60. ábra: A file felíró eljárás terve 133

61. ábra: Az text file-t kezelő program futási képe 134

62. ábra: A típus nélküli file-t kezelő program futási képe 136

63. ábra: A file másoló program működési terve 137

64. ábra: A file másoló program futási képe 138

65. ábra: Példa a Turbo Pascal grafikus lehetőségeire 139

66. ábra: Az OOP példaprogram képernyőképe 142

67. ábra: A betéteket kezelő OOP program képernyőképe 145

68. ábra: A bináris vektort kezelő OOP program képernyőképe 148

69. ábra: A dinamikus objektumokat kezelő program képernyőképe 150

70. ábra: Az objektum struktúra terve 151

71. ábra: Az öröklést bemutató OOP program képernyőképe 153

72. ábra: A unit szerkezete 155

73. ábra: A unitot használó példaprogram képernyőképe 157

74. ábra: Az egyes program elemek érvényessége 158

75. ábra: A unitok és a program terve 159

76. ábra: A Descartes<->polár konvertáló program képernyőképe 161

77. ábra: A Delphi DLL projekt ablaka 163

78. ábra: A DLL-t használó példaprogram képernyőképe 165

79. ábra: A másodfokú egyenlet megoldásának első része 167

80. ábra: A másodfokú egyenlet megoldásának második része 168

81. ábra: Összetett matematikai kifejezés vizsgálata 169

82. ábra: A négyzetes átlag megoldása 171

83. ábra: A P2_2 összeg kiszámítása 172

84. ábra: A P2_3 összeg kiszámítása 173

85. ábra: A P2_4 szorzat kiszámítása 175

86. ábra: Szórás kiszámítása 177

87. ábra: Pascal háromszög előállítása 179

88. ábra: Pascal háromszög előállító program képernyőképe 180

89. ábra: Valós típusokat kiíró program képernyőképe 184

90. ábra: A számtani sorozat példaprogram képernyőképe 190

91. ábra: A mátrix előállító példaprogram képernyőképe 191

92. ábra: A Pascal háromszög előállító példaprogram képernyőképe 192

93. ábra: A szinusz táblázatot előállító példaprogram képernyőképe 194

94. ábra: A prímszám vizsgáló példaprogram képernyőképe 195

95. ábra: A cos(x) számító példaprogram képernyőképe 196

96. ábra: A P13_1 példaprogram képernyőképe 198

97. ábra: A hatos lottó sorsoló példaprogram képernyőképe 199

98. ábra: A munkabér számító példaprogram képernyőképe 201

99. ábra: A használtautó nyilvántartó példaprogram képernyőkép e 203

100. ábra: A faktoriális számító példaprogram képernyőképe 204

101. ábra: A Fibonacci ellenőrző példaprogram képernyőképe 205

102. ábra: A számjegyeket összeadó példaprogram képernyőképe 206

103. ábra: A mátrix kivonás eredménye 209

104. ábra: A mátrix szorzás eredménye 209

105. ábra: A hibás parancs eredménye 210

106. ábra: A hibás méret eredménye 210

107. ábra: A szállodai példaprogram képernyőképe 212

108. ábra: Prímgeneráló programunk futási képe 213

109. ábra: A skalár típusú file-t kezelő példaprogram képernyőképe 214

110. ábra: A rekord típusú file-t kezelő példaprogram képernyőképe 215

111. ábra: A rizikófaktort számoló példaprogram képernyőképe 218

112. ábra: A random file kezelő példaprogram képernyőképe 219

113. ábra: A szöveges file kezelő példaprogram képernyőképe 221

114. ábra: A Thibault számokat bemutató program képernyőképe 222

115. ábra: A telefonregiszter példaprogram képernyőképe 225

116. ábra: A vektor monotonitást vizsgáló program képernyőképe 226

117. ábra: A morze dekódoló fa struktúra terve 227

118. ábra: A morze dekódoló program képernyőképe 230

119. ábra: A string kezelő OOP példaprogram képernyőképe 231

120. ábra: A tanulók adatait kezelő OOP program képernyőképe 234

121. ábra: Az öröklődést bemutató OOP program képernyőképe 236

 © Aradi, Bécsi, Gyenes, Péter, BME

Táblázatjegyzék

1. táblázat: Az előjel nélküli egész típusok hosszai 10

2. táblázat: Az előjel nélküli egész típusok maximális értékei 10

3. táblázat: A Pascal nyelv előjeles egész típusainak hosszai 11

4. táblázat: A Pascal nyelv előjeles egész típusainak intervallumai 11

5. táblázat: A hexadecimális számrendszer értékkészlete 12

6. táblázat: A valós adattípusok méretei bitben 13

7. táblázat: A valós számábrázolás speciális értéke i 17

8. táblázat: A valós számábrázolás speciális értékei 32 biten 17

9. táblázat: A valós adattípusok számábrázolási tartományai 17

10. táblázat: ASCII 7 bites kódtábla 19

11. táblázat: Kiterjesztett ASCII 8 bites kódtábla 20

12. táblázat: Szöveg tárolására alkalmas adattípusok 22

13. táblázat: Szöveg tárolására alkalmas adattípusok 23

14. táblázat: Félösszeadó igazságtáblázata 24

15. táblázat: Teljes összeadó igazságtáblázata 25

16. táblázat: Az antivalencia művelet igazságtáblázata 32

17. táblázat: A modulo 2 algebra szabályrendszere 34

18. táblázat: Példák generátor polinomra 37

19. táblázat: A jegyek tömbje táblázatos formában 54

20. táblázat: Egész adattípusok 62

21. táblázat: Valós adattípusok 63

22. táblázat: Az ÉS művelet igazságtáblázata 68

23. táblázat: A VAGY művelet igazságtáblázata 68

24. táblázat: Az ANTIVALENCIA művelet igazságtáblázata 69

25. táblázat: A t_student rekord helyfoglalása 85

26. táblázat: Rizikófaktor táblázat 216

 © Aradi, Bécsi, Gyenes, Péter, BME

Irodalomjegyzék

Felhasznált irodalom

1. *Steven K.O'Brian: Turbo Pascal: The Complete Reference, McGrawHill, 2005.*
2. *Benkő T, Benkő L., Komócsin Z., Gyenes K.: Objektum Orientált programozás Turbo Pascal nyelven, ComputerBooks 2003.*

Ajánlott irodalom

3. *Benkő T, Benkő L., Komócsin Z., Gyenes K..: Objektum Orientált programozás Turbo Pascal nyelven, ComputerBooks, 2003.*
4. *Benkő Tiborné: Programozási feladatok és algoritmusok Delphi rendszerben, ComputerBooks, 2004.*