Ukrajna Oktatási és Tudományügyi Minisztériuma
Ungvári Nemzeti Egyetem
Ukrán-Magyar Oktatási-Tudományos Intézet
Fizika és Matematika Tanszék

Petki Katalin, Sáfrányos Miroszláv, Traski Viktor,
Traski Natália, Turóci - Sütő Jolán

Informatika és programozás

Egyetemi jegyzet

UNGVÁR 2023

UDC 004.43(076)
BBK В183.4я73
П-29

Petki Katalin, Sáfrányos Miroszláv, Traski Viktor, Traski Natália, Turóci-Sütő Jolán. Informatika és programozás: Egyetemi jegyzet – Ungvár: «AUTDOR-Shark», 2023. – 144 old.

RECENZENSEK:
Dr. Gecse Ferenc *a Technikai Tudományok doktora, professzor, a Fizika és Matematika Tanszék professzora.*
Dr. Mlávec Jurij *a Fizika és Matematika Tudományok kandidátusa, docens, a Kibernetika és Alkalmazott Matematika Tanszék docense*

Kiadását ajánlotta:

- **Fizika és Matematika Tanszék**
  (2023. június 23-i ülésén, 11.sz. jegyzőkönyv)
- **Ukrán-Magyar Oktatási-Tudományos Intézet módszertani bizottsága**
  (2023. június 27-i ülésén, 2.sz. jegyzőkönyv)
- **Ukrán-Magyar Oktatási-Tudományos Intézet Tudományos Tanácsa**
  (2023. június 30-i ülésén, 10.sz. jegyzőkönyv)

Міністерство освіти та науки України
ДВНЗ «Ужгородський національний університет»
Українсько-угорський навчально-науковий інститут
Кафедра фізико-математичних дисциплін

К.П. Петкі, М.І. Шафраньош, В.Б. Трошкі,

Н.В. Трошкі, Й.М. Туровці-Шютев,

Інформатика та програмування

Посібник

Ужгород 2023

УДК 004.43(076)
ББК В183.4я73
П-29

К.П. Петкі, М.І. Шафраньош, В.Б. Трошкі, Н.В.Трошкі, Й.М. Туровці-Шютев, Т-76 Інформатика та програмування: Посібник - Ужгород: «АУТДОР-Шарк», 2023. – 144 с.

РЕЦЕНЗЕНТИ:
Федір ГЕЧЕ – *доктор технічних наук, професор, професор кафедри фізико-математичних дисциплін.*
Юрій МЛАВЕЦЬ – *кандидат фіз.-мат. наук, доцент, доцент кафедри кібернетики та прикладної математики.*

Рекомендовано до друку:

- **Кафедрою фізико-математичних дисциплін** *(протокол No11 від 23 червня 2023р.)*
- **Науково-методичною комісією Українсько-угорського навчально-наукового інституту** *(протокол No2 від 27 червня 2023р.)*
- **Вченою радою Українсько-угорського навчально-наукового інституту** *(протокол No10 від 30 червня 2023р.)*

Tartalom

Bevezetés

Ez a jegyzet a Ukrán-Magyar Oktatási-Tudományos Intézet első évfolyamos fizika és informatika szakos hallgatói számára készült és a szak tematikáját követi.

A könyv két fejezetből épül fel, amelyek több témából állnak. A szerzők minden fejezetben példákon keresztül is igyekeznek bemutatni a új ismereteket. A példaprogramokat célszerű begépelni és a javasolt módokon kipróbálni. Az egyes fejezetek végén felsorolt kérdések lehetőséget adnak arra, hogy a diákok gyakorolhassák az adott témakört. A második fejezet végén pedig gyakorlati és pótfeladatok találhatóak, melyeknek segítségével fejleszthetik készségeiket és képességeiket egyes témakörökből.

Mindegyik témához fel vannak sorolva a főbb elméleti tudnivalók az önálló felkészüléshez, példák a feladatok megoldására és az önálló munkához ajánlott feladatsor húsz változata.

A jegyzet célja: segítséget nyújtani a matematika és informatika szakos hallgatók számára az önálló munkában a “Informatika és programozás” nevű tantárgy tanulmányozása során és a megfelelő modul dolgozatokra, szigorlatokra és vizsgákra való felkészülésben.

A Pascal programozási nyelv alapjait egy svájci tudós, Niklaus Wirth fejlesztette ki 1971-ben. Akkor még csak egy számítógéprendszeren futott Pascal fordítóprogram, 1974-ben már tízen, 1979-re már több, mint nyolcvanon. Ez a programozási nyelv olyan sikeresnek bizonyult, hogy gyorsan népszerűvé vált a programozók között, és a Borland Corporation jóvoltából egy nagy teljesítményű modern professzionális programozási rendszerré vált, melynek segítségével különböző feladatokat lehet megoldani a viszonylag egyszerű számítástechnikai programoktól egészen a komplex relációs adatbázis-kezelő rendszerek létrehozásáig.

A nyelv egyszerűségének és lenyűgöző képességeinek sikeres kombinációja az arra épülő erőteljes programozási rendszerek megjelenéséhez vezetett, mint például a Turbo Pascal, Delphi, Kylix.

A Pascal prgramozási nyelvet az teszi különösen vonzóvá, hogy ma azon kevés programozási nyelvek egyike, amely fejlett programozási eszközökkel rendelkezik számos elterjedt modern operációs rendszerhez, ami meglehetősen egyszerűvé teszi a Pascal nyelven írt programok portolását különböző operációs rendszerekre.

Ma valamennyi korszerű gépen rendelkezésre áll a nyelv; a mindenhová bevonult személyi számítógépek, az intelligens mukaterminálok elképzelhetetlenek a Pascal nyelv nélkül.

A Pascal magas szintű, általános célú, struktúrált programnyelv, az Algol és a Fortran legjobb elemeit egyesíti. Szigorú nyelv, egyszerű eszközrendszerrel, szintaktikai és szemantikai szabályokkal. A programnyelv jól használható, közel áll az emberi gondolkodáshoz és könnyen elsajátítható. Az oktatás kedvelt nyelve. Szabad formátumú nyelv, azaz a külalaknak (pl. sorhosszúság, bekezdések) csak a program áttekinthetősége szempontjából van jelentősége.

A Pascal program nyelvezete erősen emlékeztet a programstruktúra mondatszerű leírására. Az angolul tudók a parancs- és utasításnevek ismeretében - a nyilvánvaló kötöttségekkel - szinte szabályos mondatokat fogalmazva írhatják meg programjaikat. Természetesen az angol nyelvet nem ismerők számára sem jelent problémát a programozás Pascal nyelven.

A Delphi tervezői felület nyelvezete az Object Pascal. Fejlesztöi környezet vizualitása elösegiti a diákok projekt orientált programozási készségét. A Delphiben szerkesztett alkalmazások ösztönzik a logikai gondolkodás fejlödését és az algoritmusok programozási nyelven valo ábrázolás képességét.

Az “Informatika és programozás” című tantárgy elsajátítása elengedhetetlen több egyetemi kurzus sikeres elvégzéséhez, mint például a “Numerikus analízis”, “Informatika tanitás módszertana”, valamint a számítástechnikai gyakorlat eredményes leadásához.

A fentiekkel összefüggésben és az információs technológiák rohamos fejlődésére való tekintettel a Pascal és a Delphi nyelvek eszközeinek elsajátítása a szakképzett szakemberek alapképzésének szerves részévé vált, akik képesek hatékonyan kihasználni a modern számítástechnika minden lehetőségét a gyakorlati problémák megoldására.

I Fejezet

Értékadás, olvasás a billentyűzetről, kiírás a képernyőre

Értékadó utasítás

változó := *kifejezés*
függvény azonosító := *kifejezés* ld. xx fejezet

Példa:

```
x := 4 * ( x + y ) + sin(alfa);
```

A kifejezés kiértékelése során előálló érték kerül a változó címére a memóriába. A kifejezések formálisan operandusokból, műveleti jelekből (operátorok) és kerek zárójeleből épülnek fel. Operandusok konstansok (pl. 2, 'szöveg'), változók (pl. x, alfa) és függvényhívások (pl. sin(alfa)) lehetnek [1].

Kifejezések kiértékelése:

- Először a zárójelben szereplő kifejezések kerülnek kiértékelésre.
- A műveletek kiértékelésének a sorrendjét a precedenciájuk szabja meg. A precedencia szintek:

1. NOT, +, -, @ (egy operandusú műveletek)
2. *, /, DIV, MOD, AND, SHL, SHR
3. +, -, OR, XOR
4. <, >, <=, >=, <>, =, IN

- Azonos prioritás esetén a balról jobbra szabály lép érvénybe, amelyet a fordító felülbírálhat az optimális kód készítése érdekében [2].

Olvasás a billentyűzetről

A Pascal nyelvben nincs input utasítás, a billentyűzetről a változókba a Read és a ReadLn eljárások segítségével olvashatunk be.

```
Read(v1 [,v2...])

ReadLn(v1 [,v2...])
```

A változók numerikus, karaktes és string (karakterlánc) típusúak lehetnek. A részletesebb leírást ld. szöveges állományok.

Példa:

```
ReadLn(fizetes);
```

A program ennél a sornál megáll, a leütött karakterek a képernyőn is megjelennek, és az Enter leütéséig szerkeszthetők. Majd a változóba bekerül a megfelelő típusú érték. Ha a beírtak nem felelnek meg a változó típusának, akkor a program futási hibával leáll (I/O hiba lép fel).

Karakteres, karakterlánc típusú változók olvasásakor a Read használata kellemetlen félreértéseket okozhat, használjuk a ReadLn eljárást. A Read eljárás a

billentyűzet pufferből kiolvassa a változónak megfelelő értéket, de nem törli a puffert, míg a ReadLn eljárás olvasás után törli a puffert. Próbáljuk ki a következő programrészletet:

```
Read(a);
Read(b);
Read(c);
WriteLn(a);
WriteLn(b);
WriteLn(c);
```

Kiírás a képernyőre

A Pascal nyelvben nincs output utasítás, a képernyőre a Write és a WriteLn eljárások segítségével írhatunk.

Write(k1 [,k2...])

WriteLn(k1 [,k2...])

Az eljárások az aktuális pozíciótól kezdődően kiírják a kifejezések értékeit. A WriteLn eljárás ezután sort emel. A kifejezések numerikus, karaktes és string (karakterlánc) és logikai típusúak lehetnek. A kiírást módosíthatjuk, mezőszélességet illetve valós kifejezés esetén a tizedesjegyek számát adhatjuk meg: Write(k[:MezSzel[:Tized]]). A mezőben jobbra igazítva, illetve a megfelelő számú tizedesjegyre kerekítva jelenik meg a kifejezés értéke [3].

Példa:

```
Write('A dolgozó fizetése: ', fizetes:10);
```

A Pascal program felépítése

Egy Pascal programot három részre oszthatunk:

1. Programfej
2. Programblokk - deklarációs rész (vagy: leíró rész)
3. Programblokk - végrehajtandó rész (vagy: programtörzs)

Programfej:

[**PROGRAM** *azonosító***;**]
[**USES** *azonosító* [**,***azonosító*...]**;**]

A PROGRAM fenntartott[1] szó után álló azonosító[2] lesz a programunk neve. Ez célszerűen megegyezhet a forrásprogram[3], a lemezen tárolt PAS kiterjesztésű állomány nevével. Mindez elhagyható, de használata jaánlott [3].

A USES kulcsszó után a programunk által használt egységeket soroljuk fel. A System egység, amely a leggyakrabban használt deklarációkat - konstansokat, változókat, eljárásokat, függvényeket - tartalmazza automatikusan hozzászerkesztődik a programunkhoz[4].

Deklarációs rész:

[**LABEL** *címke* [**,***címke*...]**;**]
[**TYPE** *azonosító* **=** *típus***;**...]
[**CONST** *azonosító* [**:** *típus*] **=** *konstans***;**...]
[**VAR** *azonosító* **:** *típus***;**...]
[**PROCEDURE** *azonosító* [**(***formális paraméterlista***)**]**;**
eljárásblokk]
[**FUNCTION** *azonosító* [**(***formális paraméterlista***)**] **:** *típusazonosító***;**
függvényblokk]

A későbbiek során részletesebben tárgyaljuk a deklarációs rész egyes elemeit.

A **Var** kulcsszó után álló változódeklarációs szakaszban a programblokkban használt összes változót fel kell sorolni, és típusát megadni. A típusmegadás történhet áttételesen is, a **Type** utáni típusdeklaráció segítségével.

A konstansok[4] használata programozói munkánkat könnyítheti meg (**Const**).

Címkék[5] igen ritkán fordulnak elő egy Pascal programban (**Label**).

Végrehajtandó rész:

---

1 Fenntartott szó, kulcsszó. A programnyelv tulajdonít neki értelmet, amit nem lehet megváltoztatni, azaz ezeket az azonosítókat másra nem használhatjuk A Turbo Pascal kulcsszavai: AND, ASM, ARRAY, BEGIN, CASE, CONST, CONSTRUCTOR, DESTRUCTOR, DIV, END, FILE, FOR, FUNCTION, GOTO, IF, IMPLEMENTATION, IN, INLINE, NIL, NOT, OBJECT, OF, OR, PACKED, PROCEDURE, PROGRAM, RECORD, SHR, STRING, THEN, TO, TYPE, UNIT, UNTIL, USES, VAR.

2 Azonosító. A program egyes objektumait azonosítja (program, egység, eljárás, függvény, címke, konstans, típus, változó, rekordmező...). Kis- és nagybetűből, számjegyből és aláhúzásjelből állhat, de számjeggyel nem kezdődhet. A rendszer az első 63 karaktert különbözteti meg.

3 Forrásprogram. Egy magasszintű programnyelven megírt szöveg. A Turbo Pascal rendszerben kiterjesztése .pas. A forrásprogram fordítása után kapjuk a tárgykódot. A tárgykódú modulakat a szerkesztő (linker) állítja össze futtatható programmá. A Turbo Pascal renszerben a szerkesztés automatikus.

4 Konstansok.
Nevesített konstans: olyan programozói objektum, melynek neve címe, típusa, értéke van. Értéke állandó, a programban nem lehet megváltoztatni. Pl.: const max = 100;

5 Címke. Egy utasítást jelölhetünk meg vele, és egy másik utasításban a címkére, azaz a megjelölt utasításra hivatkozhatunk

BEGIN
[*utasítás* [**;** *utasítás*...]]
END.

A Pascal szabad formátumú nyelv, azaz több utasítás is szerepelhet egy sorban vagy akár egy utasítást több sorra is tördelhetünk. Célszerű a program olvashatóságára, áttekinthetőségére törekedni. Az egyes utasításokat **;** -vel választjuk el egymástól. (Szemben pl. a C nyelvvel, ahol minden utasítás végére ; -t kell írnunk.) [1]

A Pascal nyelv nem különbözteti meg a kis és nagy betűket (szemben a C nyelvvel).

Programunkban korlátlan hosszúságú megjegyzést[6] helyezhetünk el a { ... } illetve a (* ... *) jelek között.

A program szövegében használhatunk:

- fenntartott szavakat(pl. Program, Begin, If, Var ...)
- standard azonosítókat[7] (pl Sin, Integer...)
- azonosítókat.

Egyszerű adattípusok

Egész típusok

Egy típus jellemzésénél az alábbiakat kell figyelembe venni:

a. felvehető értékek halmaza (adatábrázolás)
b. konstansai
c. végezhető műveletek
d. szabványos eljárások, függvények

a. A Pascal nyelvben bőséges választék áll a rendelkezésünkre egész típusokból

Típus	*Értéktartomány*	*Tárolás*
Byte	0..255	1 byte
ShortInt	-128..127	1 byte
Word	0..65535	2 byte
Integer	-32768..32767	2 byte
LongInt	$-2*10^9..2*10^9$	4 byte

A Byte és a Word típus esetén 8 illetve 16 biten $2^8 = 256$ (00000000-tól 11111111-ig) illetve $2^{16} = 65536$ különböző számot ábrázolhatunk kettes

[6] Megjegyzés. A forrásprogram olvasójának szól, a fordító figyelmen kívűl hagyja. A program egyes részeit magyarázhatjuk, a későbbi könnyebb érthetőség érdekében.

[7] Standard azonosító. A programnyelv tulajdonít neki értelmet, de azt megváltoztathatjuk, átdefiniálhatjuk. Célszerű meghagyni az eredeti funkcióját. Ilyen például: integer, sin, abs...

számrendszerben. A ShortInt, az Integer és a LongInt típusokban negatív számokat is tárolhatunk. Itt az ábrázolási tartomány egyik fele kettes komplemens kódolással[8] negatív számokat jelent [4].

b. Egész típusú konstans

Decimális egészek, pl. 25, -123
Hexadecimális egészek, pl. $33, -$A2D6

c. Végezhető műveletek

Az eredmény is egész típusú:

+,- (előjel), *, +, -
div egész osztás, pl. 13 div 3 = 4
mod maradékképzés, pl. 13 mod 3 = 1
not bitenkénti negálás, pl not 28 = -29 ; (not 00011100 = 11100011)
and bitenkénti *és*, pl. 5 and 6 = 4; (00000101 and 00000110 = 00000100)
or bitenkénti *vagy*, pl. 5 or 6 = 7; (00000101 or 00000110 = 00000111)
xor bitenkénti *kizáró vagy*, pl. 5 xor 6 = 3; (00000101 and 00000110 = 00000011)
shl bitenkénti eltolás balra, pl. 5 shl 3= 40; (00000101 shl 3 = 00101000)
shr bitenkénti eltolás jobbra, pl. 5 shr 2= 1; (00000101 shr 2= 00000001)

Az eredmény kivezet az egész számok köréből:

/ az eredmény mindig valós (6/2-t már nem egész számként kezeli a rendszer)
<, >, <=, >=, =, <> relációs műveletek, az eredmény logikai típusú
in halmaz elemvizsgálat, logikai eredmény

d. Fontosabb szabványos eljárások, függvények

Függvények:

Abs - visszatérési értéke, az argumentummal azonos típusú, az argumentum abszolút értéke.
Sqr - visszatérési értéke, az argumentummal azonos típusú, az argumentum négyzete.
Trunc - egy valós értéket egésszé csonkít a törtrész levágásával.
Round - egy valós értéket egésszé kerekít.
Ord - egy sorszámozott típusú kifejezés sorszámával tér vissza.
Pred - a paraméterét megelőző értéket adja vissza.
Succ - a paraméterét követő értéket adja vissza.
Random - egy véletlenszámot állít elő.

---

8 Kettes komplemens kódolás. Negatív egész számok ábrázolására használják. Így a kivonás visszavezethető egy kettes komplemens képzésre (ami igen egyszerű) és egy összeadásra.
Egy b (egy bájtos) bináris szám kettes komplemense: (11111111 - b) + 1. A zárójelben lévő rész a b egyes komplemense.

Eljárások:

Inc - egy változót növel.
Dec – egy változót csökkent
Str - egy numerikus értéket karakterlánccá konvertál.
Val - egy karakterláncot numerikus értéké konvertál.
Randomize - inicializálja a beépített véletlenszám generátort.

Megj.
A függvények mindig egy értéket állítanak elő (visszatérési érték), kifejezésekben hívhatjuk meg őket, pl. egy értékadó utasítás jobb oldalán; a := Abs(a) + 2. A függvények paraméteri kifejezések lehetnek, pl. Sqr(a + Round(x)).
Az eljárásokat utasításszerűen hívjuk, pl. Inc(i).

Példaprogram [1]

```
program Oszt;
uses Crt;                              {A képernyő törlést tartalmazó unit}
var Osztando, Hanyados, Maradék: byte;
begin
  ClrScr;                              {Képernyő törlés}
  Write('Kérem az osztandót: ');
  ReadLn(Osztando);
  Hanyados := Osztando div 3;
  Maradek := Osztando mod 3;
  WriteLn('A hányados: ', Hanyados);
  WriteLn('A maradék: ', Maradek);
  ReadLn
end.
```

Valós típusok

a. Értéktartomány, számábrázolás

A Turbo Pascalban a következő valós típusokat definiálták: Real (6 byte), Single (4 byte), Double (8 byte), Extended (10 byte), Comp (8 byte), azonban a Real típus kivételével mindegyik használatához matematikai társprocesszorra, vagy annak emulálására van szükség [1].

A Real típus ábrázolása 6 bájton, lebegőpontosan történik. Az értéktartomány:

$S_{min} = 2{,}9*10^{-39}$
$S_{max} = 1{,}7*10^{38}$

A pontosság 11-12 decimális számjegy. (Ennyi értékes számjegye egy valós számnak, a többi jegyet nem ábrázolja a rendszer, pl. a 1234567890,1234567 számból a színes jegyek elvesznek.)

b. Konstansok

Pl. 5.12, -123.2313, 12.54E6, 21.12E-5

c. Műveletek

Az eredmény is valós típusú: +,- (előjel), *, /, +, -
Az eredmény logikai típusú: <, >, <=, >=, =, <>

d. Fontosabb szabványos eljárások, függvények

Függvények: **Abs, Sqr, Random, Round, Trunc**,

Sqrt – Visszatérési értéke az argumentum négyzetgyöke.
Sin – Visszatérési értéke az argumentum szinusza.
Cos – Visszatérési értéke az argumentum koszinusza.
ArcTan – Visszatérési értéke az argumentum arkusz tangense.
Exp – Visszatérési értéke: e^x, ahol *e* a természetes logaritmus alapja.
Ln – Visszatérési értéke az argumentum természetes alapú logaritmusa.
Int – Visszatérési értéke az argumentum egészrésze.
Frac – Visszatérési értéke az argumentum törtrésze.
Pi – A π értékével tér vissza.

Eljárások: **Str, Val, Randomize**

Példaprogram [1]

```
program Valos_Pelda;
uses Crt;
var alap, kitevo, hatvany: real;
begin
  ClrScr;
  Write('Kérem a hatvány alapját: ');
  ReadLn(alap);
  Write('Kérem a hatvány kitevőjét: ');
  ReadLn(kitevo);
  hatvany := Exp(kitevo * Ln(alap)); {ld. logaritmus definiciója, logaritmus
azonosságok}
  WriteLn('A hatvány értéke: ', hatvany:10:2);
  ReadLn
end.
```

A karakteres - Char típus

a. Értéktartomány, adatábrázolás

Egy bájtos típus, tehát $2^8 = 256$ különböző érték , az ASCII kódrendszer 256 elemének a tárolására képes. A karakter típusú változó egy ASCII kódot tartalmaz. Pl. ha a változóhoz tartozó memóriarekesz értéke 65, akkor mivel változónk típusa Char, ezt a rendszer 'A' betűként értelmezi [3].

b. Konstansok

Formája: 'A', '*', '4', #65 (ez utóbbi a 65-ös ASCII kodú karaktert, azaz 'A'-t jelenti).
Lehetnek: betűk, számjegyek, írásjelek, speciális karakterek (pl. '@', '#', '$', ...), vezérlő karakterek (pl. a gyakran használt #27 - Escape), egyéb karakterek (az ASCII kódtábla 128-255 része, pl. 'é').

c. Műveletek

Relációs műveletek: <, >, <=, >=, =, <> (az eredmény természetesen logikai típusú, a karakter ASCII kódja határozza meg. Pl. 'A' < 'B' igaz, 'A' = 'a' hamis.) **in** halmaz elemvizsgálat, logikai eredmény (ld. halmaz adattípus)

d. Fontosabb szabványos eljárások, függvények

Függvények: Ord, Chr, UpCase, Pred, Succ
Eljárások: Inc, Dec

Példaprogram [1]

```
program Kisbetu;
uses Crt;
var Nagy, Kis: char;
begin
  ClrScr;
  Write('Kérek egy nagybetűt: ');
  ReadLn(Nagy);
  Kis := Chr( Ord(Nagy) + 32 );          {egy nagybetűt kisbetűvé konvertál}
  WriteLn(Kis);
  ReadLn
end.
```

A logikai - Boolean típus

a. Értéktartomány, adatábrázolás

Egy logikai típusú változó két értéket vehet fel: *igaz* vagy *hamis*. Ábrázolására egy bájton történik (akár egy bit is elég lenne). Ha a bájt értéke 0, akkor a logikai típusúként értelmezett érték *hamis*, nullától eltérő érték esetén pedig *igaz*.

b. Konstansok

Két előredefiniált konstans: *True* (igaz), *False* (hamis)

c. Műveletek

Az eddig tanult típusokra értelmezett relációs operátorok (valamint az in halmazművelet) mindig logikai értéket állítanak elő [5].

Logikai műveletek: **not**, **and**, **or**, **xor** (az operandusok logikai típusúak). A műveletek igazságtáblái:

NOT		**AND**			**OR**			**XOR**		
A	**not** A	A	B	A **and** B	A	B	A **or** B	A	B	A **xor** B
False	*True*	*False*	*False*	*False*	*False*	*False*	*False*	*False*	*False*	*False*
True	*False*	*False*	*True*	*False*	*False*	*True*	*True*	*False*	*True*	*True*
		True	*False*	*False*	*True*	*False*	*True*	*True*	*False*	*True*
		True	*True*	*True*	*True*	*True*	*True*	*True*	*True*	*False*

Pl. egy logikai kifejezés: (x > 4) and (x < 10) or not b , ahol *b* egy Boolean változó.

d. Logikai értéket előállító függvények

Odd - az Odd (X) függvény értéke True, ha X páratlan szám.

Eof – ha az állomány mutató a fájl végén (az utolsó elem mögött) áll, akkor az értéke True.

Eoln - ha az állomány mutató a sor végén áll, akkor az értéke True.

Példaprogram [1]

```
program Logikai;
uses Crt;
var b: boolean;
   a: byte;
begin
  ClrScr;
  Write('Kérek egy számot: ');
  ReadLn(a);
  b := a mod 3 = 0;
  if b then
    WriteLn(a,' osztható 3-mal)
  else
    WriteLn(a,' nem osztható 3-mal);
  ReadLn
end.
```

Felsorolt típus

a. Értékek, adatábrázolás

A típus megadásakor fel kell sorolnom a lehetséges értékeit. Ezek csak azonosítók lehetnek.

A konstansok a felsorolás sorrendjében sorszámot kapnak 0-tól kezdődően. Tarolás a konstansok számától függően 1 vagy 2 bájton történik. [8]

Pl.

```
var tantargy: (magyar, matek, fizika, tesi);
...
tantargy := matek;
```

b. Konstansok

Konstansai a felsorolásban szereplő azonosítók.

c. Műveletek

Relációs műveletek:<, >, <=, >=, =, <> (az eredmény természetesen logikai típusú, a felsorolt típusú érték sorszáma határozza meg. Példánkban: *matek* < *tesi* igaz.) **in** halmaz elemvizsgálat, logikai eredmény (ld. halmaz adattípus)

d. Függvények, eljárások

A sorszámozás alapján értelmezhetőek az alábbi
függvények:**Ord, Pred, Succ**;
eljárások: **Inc, Dec**.

Intervallum típus

Egy már létező sorszámozott típus egy intervalluma. Szintén nekünk kell definiálnunk.

Az adatábrázolás, műveletek, függvények, eljárások megegyeznek az eredeti típuséval [11].

Pl.

```
var  nap: (hetfo, kedd, szerda, csutortok, pentek, szombat, vasarnap); {felsorolt
típus}
     munkanap: hetfo..pentek;
     betu: 'A'..'Z';
     szamjegy: '0'..'9';
```

A felsorolt típus használata a program tesztelésekor hasznos lehet. (A fenti példában *szamjegy := 'e'* fordítási hibát, *szamjegy* beolvasásakor az 'e' karakter pedig futási hibát eredményez ($R+ fordítási direktíva esetén).) [1]

Az egyszerű típusok csoportosítása

- Sorszámozott típusok
  - egészek
  - logikai
  - karakter
  - felsorolt
  - intervallum
- Valós típusok

Sorszámozott típusok: minden lehetséges értékhez hozzárendelhető egy sorszám, az értékek a sorszám szerint rendezettek.

Algoritmus elemek

Ciklusok - iterációk

Segítségével utasítás(ok) ismételten végrehajható(k). Részei az utasítást (utasításokat) tartalmazó ciklusmag és az ismételt folytatást vagy befejezést vezérlő rész.

A Pascal nyelv három ciklust definiál: két feltételes ciklust valamint egy előírt lépésszámú (más néven számláló vagy léptető) ciklust [12].

A WHILE ciklus

Az utasítás szintaktikája:

WHILE *feltétel* **DO** *utasítás*

Kezdőfeltételes ciklus. Amíg a feltétel igaz, addig ismétli az utasítást, ha hamis, akkor a program következő utasítására ugrik. A feltétel egy logikai (boolean) kifejezés. Ha több utasítás szerepel a ciklus magjában, akkor **Begin** - **End** ún. utasítás zárójelet kell alkalmaznunk. Ha a feltétel már először sem teljesül, akkor a ciklusmag egyszer sem kerül végrehajtásra (üres ciklus), ha pedig a feltétel soha nem vesz fel hamis értéket, akkor a program végtelen ciklusba kerül.

Tipikus használata: a ciklus végrehajtása attól függ, hogy van-e még feldolgozandó adat, az ismétlések számát előre nem ismerjük, akár 0 is lehet [5].

Példa: Képezzük a billentyűzetről érkező pozitív számok összegét, a számsorozat végét a 0 vagy egy negatív szám jelezze [1].

Megoldás

Az ilyen típusú feladatok általános megoldási sémája:

- *az első adat előállítása (pl. beolvasása)*
- *ciklusfej*
- *az adat feldolgozása*
- *a következő adat előállítása (pl. beolvasása)*
- *ciklus vége*

```
program Osszeg_szamitas;
uses Crt;
```

```
var adat, osszeg: integer;
begin
  ClrScr;
  osszeg := 0;
  ReadLn(adat);
  while adat > 0 do
    begin
      osszeg := osszeg + adat;
      ReadLn(adat)
    end;
  WriteLn('Az összeg: ',osszeg);
  ReadLn
end.
```

A REPEAT ciklus

Az utasítás szintaktikája:

REPEAT [*utasítás* [**;** *utasítás*...]] **UNTIL** *feltétel*

Végfeltételes ciklus. Az utasítás(ok) végrehajtását meg kell ismételni, ha a feltétel hamis. Ha a feltétel igaz, a program a ciklus utáni utasítással folytatódik.

A ciklusmag legalább egyszer végrehajtódik. A ciklusmagot a **Repeat** - **Until** kulcsszavak fogják közre, nem kell **Begin** - **End** utasítás zárójelet használnunk.

A While és a Repeat ciklust hasonló típusú feladatok megoldására használhatjuk. Esetleg az egyik egy kicsit kényelmesebb megoldást nyújt.

Az előző példa megoldása Repeat-Until ciklus alkalmazásával [1].

```
program Osszeg_szamitas;
uses Crt;
var adat, osszeg: integer;
begin
  ClrScr;
  osszeg := 0;
  ReadLn(adat);
  repeat
    osszeg := osszeg + adat;
    ReadLn(adat)
  until adat <= 0;
  WriteLn('Az összeg: ',osszeg);
  ReadLn
end.
```

A FOR ciklus

Az utasitás szintaktikája:

FOR *ciklusváltozó* **:=** *kezdőérték* **TO** / **DOWNTO** *végérték* **DO** *utasítás*

ahol a *ciklusváltozó* sorszámozott típusú változó hivatkozás, a *kezdőérték* és a *végérték* pedig sorszámozott típusú kifejezések.

A *ciklusváltozó* a *kezdőértéktől* a *végértékig* egyesével nő (**To**) vagy csökken (**DownTo**). Az *utasítás* (vagy a **Begin** - **End** utasítás zárójelek közé zárt utasításcsoport) a ciklusváltozó minden értékénél végrehajtódik. Elöltesztelő ciklus, így ha a *kezdőérték* > *végérték* (**To** esetén) vagy a *kezdőérték* < *végérték* (**DownTo** esetén), akkor a ciklusmag egyszer sem kerül végrehajtásra (üres ciklus).

A ciklusváltozó értékét a ciklusmagban felhasználhatjuk, de nem változtathatjuk meg [5].

Tipikus használata: az ismétlések száma a ciklusba való belépés előtt már ismert vagy kiszámítható.

Példák [1]:

1. Írjunk ki N darab csillagot a képernyőre!

```
program Csillag;
uses Crt;
var i, n: byte;
begin
  ClrScr;
  ReadLn(n);
  for i := 1 to n do
    Write('*');
  ReadLn
end.
```

2. Számoljuk ki N! értékét! [1]

```
Program Faktorialis;
Uses Crt;
Var i, n: Byte;
    fakt: LongInt;
Begin
  ClrScr;
  ReadLn(n);
  fakt := 1;
  For i := 2 to n do
    fakt := fakt * i;
```

```
  WriteLn(n,'! = ',fakt);
  ReadLn
End.
```

Elágazások - szelekciók

A ciklus mellett a másik alapvető algoritmus elem az elágazás vagy szelekció, amelyben lehetőség van több tevékenység közül egyet kiválasztani.

Az IF utasítás

Az utasítás szintaktikája:
IF *feltétel* **THEN** *utasítás1* [**ELSE** *utasítás2*]
ahol a *feltétel* egy logikai kifejezés.

Ha a *feltétel* igaz, akkor az *utasítás1* hajtódik végre, egyébként az *utasítás2*. Az **Else** ág elhagyható, ilyenkor az *utasítás1* kimarad, a program a következő utasítással folytatódik. Egy ágon több utasítás is végrehajtható a **Begin** - **End** utasítás zárójelek alkalmazásával. Vigyázzunk, az **Else** előtt a pontosvessző szintaktikai hiba, mivel azzal lezárjuk a teljes **If** utasítást! [12]

Példa:

```
  ReadLn(Oszto);
  ReadLn(Osztando);
  if Oszto <> 0 then
    Hanyados := Osztando / Oszto
  else
    WriteLn('0-val való osztás, nincs értelmezve.');
```

Akár az *utasítás1* vagy az *utasítás2* is lehet újabb **If** utasítás, és ezáltal többirányú elágazást is megvalósíthatunk.

If utasítások egymásba ágyazása [1], [6], [7]

```
if x > 0 then
  WriteLn('Pozitív)
else
  if x = 0 then
    WriteLn('Nulla')    {A három ág közül egy hajtódik végre.}
  else
    Writeln('Negatív');
{A problémát az alábbi módon is kódolhatjuk,
 mivel a három feltétel közül egyszerre csak egy áll fenn:}
if x > 0 then
  WriteLn('Pozitív);
if x = 0 then
  WriteLn('Nulla');
```

```
if x < 0 then
  Writeln('Negatív');
{Az alábbi viszont hibás megoldás:}
if x > 0 then
  WriteLn('Pozitív);
if x = 0 then
  WriteLn('Nulla')
else
  Writeln('Negatív');
{Vigyázzunk, az Else mindig a legutolsó Then párja!}
if a > 0 then
  if Not Odd(a) then
    Writeln('Pozitív, páros')
  else
    Writeln('Pozitív, páratlan');
if a > 0 then
 begin
  if Not Odd(a) then
    Writeln('Pozitív, páros')
 end
  else
    Writeln('Nem pozitív');
```

A CASE utasítás

Az utasítás szintaktikája:

CASE *szelektor* **OF**
állandó [**..***állandó*] [,*állandó*[**..***állandó*]...] **:** *utasítás*;
[*állandó* [**..***állandó*] [,*állandó*[**..***állandó*]...] **:** *utasítás*;
[**ELSE** *utasítás*]
END

ahol a *szelektor* egy sorszámozott típusú kifejezés. Abban az ágban lévő *utasítás* (vagy **Begin** - **End** közé zárt utasításcsoport) hajtódik végre, ahol a szelektor értéke megegyezik az egyik *állandóval* vagy beleesik az egyik megadott tartományba. Ha ez egyik esetre sem teljesül, akkor az **Else** ágra kerül a vezérlés. Ez utóbbi elhagyható [4].

Példaprogram [1]

```
program CasePl;
uses Crt;
var  Kar: char;
begin
```

```
  ClrScr;
  ReadLn(kar);
  case Kar of
    'A'..'Z', 'a'..'z': WriteLn('betű');
    '0'..'9':           WriteLn('számjegy');
    '.', ',', '!', '?': WriteLn('írásjel');
    else                WriteLn('egyéb karakter')
  end;
  ReadLn
end.
```

Karakterlánc típus valamint a strukturált (összetett) típusok

A karakterlánc - String típus

Deklarálása: **STRING*/[***maxhossz***]/***

ahol 1 <= maxhossz <= 255, ha elhagyjuk, maxhossz = 255. Dinamikus hosszúságú karaktersorozat: hossza futás közben változhat, elemei Char típusúak [1].

Karakterlánc konstans: 'aposztrófok közötti szöveg'

Hivatkozhatunk a karakterlánc egy elemére (amely Char típusú):

*azonosító***[***index***]**, pl. karlanc[5].

A karakterlánc típusú változó értékét megváltoztathatjuk értékadással, valamint beolvasással. Értékét kiírathatjuk a képernyőre.

Pl. karlanc := 'hahó'

Műveletek:

+ összefűzés, pl. file_spec := utvonal + file_nev;

Relációs műveletek: <, >, <=, >=, =, <> (az eredmény természetesen logikai típusú, az első nem egyenlő karakter ASCII kódja határozza meg. Pl. 'ATTILA' < 'ALFONZ' hamis.)

Adatábrázolás:

A rendszer a karakterek ASCII kódját tárolja *maxhossz*+1 bájton. A karakterlánc 0. bájtja egy tartalmazza a karakterlánc hosszát, **Ord**(karlanc[0]) = **Length**(karlanc).

Szabványos függvények, eljárások:
Függvények:
Length - a karakterlánc dinamikus hosszát adja vissza.
Pos - megkeresi a karakter első előfordulását a karakterláncban, és visszaküld egy egész értéket, amely az S. szorszáma.
Copy - visszaad egy sztring egy részét.
Concat – összekapcsol több sort.
Eljárások: **Str, Val**,

Delete – törli a sztring egy részét.
Insert - egy szöveget egy sztringbe helyez.

Lehetőleg a string műveleteket és függvényeket használjuk a karakterláncok kezelésére, a karakterenkénti hozzáférést körültekintően végezzük [4].

Példa [1]:
Olvassunk be egy modatot, és írjuk ki csupa nagybetűvel!

```
program Nagybetu;
uses Crt;
var s: string;
   i: byte;
begin
  ClrScr;
  ReadLn(s);
  for i := 1 to Length(s) do
    s[i] := Upcase(s[i]);
  WriteLn(s);
  ReadLn
end.
```

A tömb - Array típus

A tömb strukturált (összetett adattípus).

Deklarálása: **ARRAY[***indextípus [,indextípus...]***] OF** *elemtípus*

Pl.
vektor: array[1..10] of integer; {Tíz elemű egydimenziós tömb.}

matrix: array[1..5, 1..4] of integer; {5 sorból és 4 oszlopból álló kétdimenziós tömb, mátrix, elemei integer típusúak.}

A tömb olyan adatcsoport, melynek elemei azonos típusúak, az elemek száma rögzített (statikus méretű), sorrendjük kötött (indexelés), az elemekhez közvetlenül hozzáférhetünk. Meghatározása: név, dimenzió, elemeinek típusa, indexeinek típusa és tartománya [3].

Az indextípus csak sorszámozott típus lehet (kivéve Longint), többnyire intervallum típus,

pl.

array[1..10] of real {általában egész típus intervalluma}
array['a'..'z'] of real {ritkábban karakteres típus intervalluma}
array[byte] of real {még ritkábban egy előre definiált típus}.
Többdimenziós tömbök: a tömb elemei maguk is tömbök,

pl.
m: array[1..10, 1..20] of integer,
vagy
m: array[1..10] of array[1..20] of integer.
A hivatkozás a tömb egy elemére az index segítségével történik,

pl.
vektor[3] {a vektor nevű tömb 3. eleme, Integer típusú},

matrix[2, 3] vagy matrix[2][3] {a matrix nevű kétdimenziós tömb 2. sorának 3. eleme}.

Műveletek:

Két azonos típusú tömbre az értékadás és az egyenlőség vizsgálat megengedett. Egy vektor bemásolható a mátrix egy sorába.

A tömbökkel végzett műveletek során többnyire a for ciklust használjuk, úgy hogy a ciklusváltozót végigléptetjük a tömb indextartományán.

Pl. egy a vektor nevű tömb beolvasása a billentyűzetről:

```
for i := 1 to 10 do
  begin
    Write('Kérem a tömb ',i,'. elemét: ');
    ReadLn(t[i])
  end;
```

Tömb típusú konstans:

Tömb konstanst csak a **Const** deklarációs részben adhatunk meg tipizált konstansként. Az elemeket zárójelben, vesszővel elválasztva kell felsorolnunk.

Pl.
```
const T1 : array[1..3, 1..4] of byte = ( (1, 3, 4, 1), (2, 3, 4, 2), (1, 6, 3, 5) );
```

Adatábrázolás:

A rendszer a tömböt sorfolytonosan tárolja a memóriában. A foglalt terület az elemek száma szorozva egy elem méretével.

Tömb típus deklarálása:

Tömb típusú változók használatakor célszerű először a **Type** típusdeklerációs részben a megfelelő típusokat deklarálni, majd ezeket a saját típusainkat használhatjuk a változók deklarálásakor (a Var után). A Pascal nyelv logikája ezt az áttételes deklarálást támogatja, és bizonyos esetekben ez nem is kerülhető meg **[1]**.

Pl.
```
type VektorTip = array[1..4] of integer;
    MatrixTip = array[1..5] of VektorTip;
var  v1, v2: VektorTip;
    m1, m2: MatrixTip;
```

A fenti példában elvégezhető a következő értékadás: pl. m1[2] := v1.

Példák [1]:

1. Töltsünk fel egy 10 elemű integer tömböt. Számítsuk ki az elemek számtani átlagát.

```
program Atlagsz;
uses Crt;
type TombTip = array [1..10] of integer;
var       t: TombTip;
          atlag: real;
          osszeg, i: integer;
begin
  ClrScr;
  {beolvasás}
  for i := 1 to 10 do
    begin
      Write('Kérem a tömb ',i,'. elemét: ');
      ReadLn(t[i])
    end;
  {átlagszámítás}
  for i := 1 to 10 do
     osszeg := osszeg+t[i];
  atlag := osszeg/10;
  WriteLn('A tömb elemeinek számtani átlaga: ', atlag:10:2);
  ReadLn
end.
```

2. Állítsuk elő a Fibonacci sorozat (1, 1, 2, 3, 5, 8, 13...) első 20 elemét.

```
program Fibonacci;
uses Crt;
type TombTip = array [1..20] of integer;
var  t: TombTip ;
     i: integer;
begin
  t[1] := 1;
  t[2] := 1;
  for i := 3 to 20 do
     t[i] := t[i-2] + t[i-1];
  for i := 1 to 20 do
     Write(t[i], '  ');
  ReadLn
end.
```

A rekord - Record típus

A rekord strukturált adattípus, amelyben különböző típusú adatokat (mezőket) fogunk össze.

Deklarálása:

Deklarálásakor a Record és End kulcsszavak között fel kell sorolnunk az egyes mezők neveit valamint típusait. A rekord tartalmazhat egy változó részt is, amely a fix rész egy mezőjétől (szelektor mező) függően más-más adatokat tartalmazhat [5].

RECORD
[*mezőlista*;]
[**CASE** *szelektormező*: *sorszámozott típus* **OF**
állandók: (*mezőlista*)
[*állandók*: *(mezőlista)*...]]
END

ahol mezőlista:
mezőazonosító [*,mezőazonosító*...]: *típus*
[;*mezőazonosító* [,*mezőazonosító*...]: *típus*...]

Példák:

1.
```
type DolgozoTip = record
                    Nev: string;
                    Hazas: boolean;
                    Fizetes: real;
                  end;
var Dolgozok: array[1..50]of DolgozoTip;
   d1, d2: DolgozoTip;
```
2.
```
type RekTip = record
              Nev: string[70];
              Lakohely: string[70]
              case Nagykoru: boolean of
                True: (Gyerek_Szam: byte);
                False: (Anyja_Neve: string[70];
                        Apja_Neve: string[70]);
            end;
```

Hivatkozás a rekord egy mezőjére:

rekordazonosító.mezőazonosító

Pl.

d1.Nev
Dolgozok[5].Hazas

A With utasítás

Ha egy programrészletben gyakran hivatkozunk egy (vagy több) rekord mezőire, akkor a With utasítással ez leegyszerűsíthető, a rekordazonosító elhagyható.
Szintaktikája: **WITH** *rekordazonosító* [**,** *rekordazonosító*] **DO** *utasítás*

Pl.

```
with d1 do
  begin
    ReadLn(Nev);
    ReadLn(Fizetes)
  end;
```

Ha több rekordazonosítót sorolunk fel, akkor az utolsónak a legnagyobb a prioritása [12].

Rekord típusú konstans:

Hasonlóan a tömb konstanshoz csak tipizált konstans kezdőértékeként adhatjuk meg a **Const** deklarációs részben.

Pl.

```
const Origo: record
               x, y: integer;
             end = (x: 320; y: 240);
```

Adatábrázolás:

A memóriában elfoglalt hely egyszerűen a mezők helyfoglalásának az összege (fix rész + legnagyobb változó rész).

A halmaz - Set típus

A programozásban a halmaz azonos típusú különböző elemek összességét jelenti. A halmazt az elemek felsorolásával adhatjuk meg. Az elemek rendezetlenek. Az összetett adattípusokhoz soroljuk, bár a halmaz egy elemére nem tudunk hivatkozni [11].

Deklarálása: **SET OF** *alaptípus*

ahol az *alaptípus* csak olyan sorszámozott típus lehet, amelynek maximálisan 256 eleme van.

Halmaz típusú konstans:

Szögletes zárójelben a halmaz elemeit (az alaptípussal megegyező típusú konstansokat) vagy azok intervallumait felsoroljuk.

Pl.

```
const Betuk = ['A'..'Z', 'a'..'z'];
```

H1 := [1, 4, 6, 8..12]
H2 := [] {üres halmaz}

A programunkban egy halmaznak a halmaz konstanstól egy kicsit különböző **halmazkonstruktorral** is értéket adhatunk. Itt a szögletes zárójelben felsorolt kifejezések változókat is tartalmazhatnak.

Pl.
k := '?';
H := [k, '!','.'];

Műveletek:
* metszet
+ egyesítés
- különbség
Logikai típusú eredményt szolgáltatnak:
= egyenlőség
<> különbözőség
<=, >= tartalmazás (részhalmaz)
IN elemvizsgálat

Adatábrázolás:

Egy lehetséges halmazelemnek egy bit felel meg a memóriában. Ha az lehetséges elem benne van a halmazban, akkor a bit 1, ellenkező esetben 0. Gyakran az első és az utolsó bájt olyan biteket is tartalmazhat, amelyek nem vesznek részt a tárolásban.

Alprogramok

Az alprogram olyan utasítások csoport, amelyet a program bizonyos pontjairól aktivizálhatunk. Az alprogramokat a deklarációs részben kell megírnunk (**procedure, function** kulcsszavak után). Az alprogramok tartalmazhatnak újabb alprogramokat (egymásba ágyazás) [1].

Akkor használjuk őket, ha

- bizonyos tevékenység többször előfordul a programban,
- egy nagy programot tagolni, strukturálni szeretnénk [3].

Két fajtája van:

- eljárás (procedure): egy tevékenységcsoportot hajt végre, utasításszerűen hívjuk (ld. standard eljárások),
- függvény (function): feladata egy érték előállítása, hívásakor neve operandusként egy kifejezésben szerepelhet (ld. standard eljárások).

Már eddig is sok szabványos eljárást és függvényt használtunk, melyeket a különböző egységekben deklaráltak. Nézzük, hogyan készíthetünk saját alprogramokat!

Eljárás - Procedure

a, Szerkezete:

Hasonló a programéhoz.

Eljárás fej: **PROCEDURE** *azonosító* [**(** *formális paraméter lista* **)**]**;**

Label...
Deklarációs rész: **Const**...
Type...
Var...
Procedure...
Function...
Begin
Végrehajtandó rész: *utasítások*
End;

ahol, formális paraméter lista:
[**Var**] *azonosító* [**,** *azonosító*...] **:** *típusazonosító* [**;** [**Var**] *azonosító* [**,** *azonosító*...] **:** *típusazonosító*...]

Például:

```
procedure Teglalap(a, b: integer; var t, k: integer);
  begin
    t := a * b;
    k := 2 * (a + b)
  end;
```

b, Az eljárás hívása [12]:

azonosító [**(***aktuális paraméter lista***)**]

ahol az aktuális paraméter lista elemei kifejezések vagy változók lehetnek (ld. paraméterátadás) egymástól vesszővel elválasztva.

Pl. *Teglalap(5, 4, Ter, Ker)*

c, Az eljárások hatásköre [11]:

A Pascal nyelv befelé struktúrált, az alprogramokat egymásba ágyazhatjuk (az alprogram deklarációs részében is lehet alprogram). Ezért fontos tisztán látnunk, hogy hogy a főprogramból illetve egy alprogramból mely eljárásokat hívhatjuk meg:

- A program illetve egy eljárás meghívhatja (ismeri) azokat az alprogramokat, melyeket a program vagy az adott alprogram deklarációs részében deklaráltunk, de azok alprogramjait már nem. - Egy alprogram meghívhatja az ugyanazon deklarációs részben (, ahol őt deklaráltuk) korábban deklarált alprogramokat.
- Egy alprogram meghívhatja az őt tartalmazó eljárásokat.
- Egy alprogram meghívhatja saját magát.

d, Paraméterek [8]:

A paraméterek az eljárás és az őt hívó programrész közötti adatcserét, kommunikációt szolgálják. A formális paraméterekkel írjuk le az alprogram tevékenységét. Híváskor ezek helyére konkrét objektumokat, aktuális paramétereket írunk.

Az aktuális és a formális paramétereknek meg kell egyezniük számban, sorrendben és típusban.

Vigyázzunk, hogy a formális paraméterek típusának megadásakor csak típusazonosítót használhatunk, így pl. a következő eljárásfej hibás: *procedure elj (t: array[1..10] of real)*;

A paraméterátadás két féle módon történhet:

- *Érték szerinti paraméter átadás* (a deklarációban a formális paraméter előtt nincs **Var**)

Ekkor az aktuális paraméter értéke kerül át a formális paraméterbe. Az eljárás minden egyes hívásakor a rendszer tárterületet rendel a verem memóriában a formális paraméterekhez, és ide másolja be az aktuális paraméterek értékeit. Az eljárás végeztével ez a terület felszabadul. Az aktuális paraméter értékét az eljárás nem változtathatja meg, így ez csak *bemenő paraméter*.

Az aktuális paraméter kifejezés lehet.

- *Cím szerinti paraméter átadás* (a deklarációban a formális paraméter elé **Var** -t írunk)

Az aktuális paraméter címe kerül át a formális paraméterhez, ha változik a formális paraméter, akkor változik az aktuális is. Ezáltal egyaránt használhatjuk *be- és kimenő paraméterként* is [1].

Az aktuális paraméter csak változó lehet.

e, Lokális és globális változók [4]

Egy eljárásban deklarált változókat ezen eljárás lokális változóinak nevezzük. Ezek a program más részein nem ismertek. (Így különböző eljárásokban előfordulhatnak azonos nevű változók, amelyeknek azonban semmi közük egymáshoz.) A lokális változókhoz (az eljárás paramétereihez hasonlóan) a rendszer a veremben rendel tárterületet, dinamikus módon, azaz csak akkor van címe a változóknak, ha az eljáráson van a vezérlés. A lokális változók értéke az eljárás két hívása között elvész.

Egy eljárásra nézve globális változó egy őt tartalmazó eljárásban vagy a főprogramban deklarált változó. Ezt az eljárás ismeri, hacsak nem deklaráltunk egy vele azonos nevű lokális változót vagy az eljárásnak nincs egy vele azonos nevű paramétere. Ekkor a lokális változó "eltakarja" a globálisat. A globális változó értéke természetesen nem vész el. (Abban az esetben használhatunk egy a lokálissal azonos nevű globális változót, ha az a főprogram változója. Ekkor a program nevével kell minősítenünk a globális változót: *programnév.változónév*.)

Lokális és globális változók – példaprogram [1]

```
program Pelda;

var a, b, c: integer;
   x, y, z: real;
```

```
procedure p1(x:real);          {a p1 nem ismeri, nem használhatja a főprogram x
változóját}
  var a: array[1..10]of real;
      s: string;               {lokális változó}
  begin
    {globális változók: b, c, y, z}
  end;

procedure p2;
  begin
    {imeri a főprogram összes változóját, mint globális változót, nem ismer a p1 s
változóját}
  end;

begin

end.
```

f, Információ csere [5]

Összefoglalva elmonhatjuk, hogy egy alprogram kétféle módon kommunikálhat az őt hívó programegységgel,

- a paramétereken keresztül,
- a globális változók segítségével.

Kommunikáció paraméterek illetve globális változók segítségével - példaprogram

1. A program egy globális változóban (tömbben) tárolja az adatokat, melyeket az eljárások feldolgoznak **[1]**.

```
program Globalis;

uses Crt;
type TOszt=array[1..50] of string;
var Osztaly: TOszt;
    Letszam: integer;
    c: char;

procedure Beiras;
  var i: integer;
  begin
    ClrScr;
    Write('Hány név lesz: ');
    ReadLn(Letszam);
    for i := 1 to Letszam do
```

```
    begin
      Write('Az ',i,'. név: ');
      ReadLn(Osztaly[i])
    end
  end;

procedure Rendezes;
  var i, j: integer;
    seged: string;
  begin
    for i := 1 to Letszam-1 do
      for j := i + 1 to Letszam do
        if Osztaly[i] > Osztaly[j] then
          begin
            seged := Osztaly[i];
            Osztaly[i] := Osztaly[j];
            Osztaly[j] := seged
          end
  end;

procedure Listazas;
  var i: integer;
  begin
    ClrScr;
    for i:=1 to Letszam do
      WriteLn(Osztaly[i]);
    ReadKey
  end;

begin
  repeat
    ClrScr;
    WriteLn('1. Nevek beírása');
    WriteLn('2. Névsorba rendezés');
    WriteLn('3. Listázás');
    WriteLn('4. Vége');
    repeat c := ReadKey until c in ['1'..'4', #27];
    case c of
      '1': Beiras;
      '2': Rendezes;
      '3': Listazas;
    end;
  until (c = '4') or (c = #27)
```

end.

2. Az egyes eljárások több adatsorral (tömbbel) is dolgoznak, paramétereken keresztül kapják meg a feldolgozandó tömböt, illetve adják vissza a főprogramnak **[1]**.

```
program Parameterek;
uses Crt;
type TNaplo = array[1..50] of real;
var A_oszt, B_oszt: Tnaplo;
    A_letsz, B_letsz: integer;
procedure Beolvas(var Oszt: Tnaplo; var Letsz: integer);
  var i: integer;
  begin
    Write('Hány tanuló van: ');
    readln(Letsz);
    for i := 1 to Letsz do
      begin
        Write('Az ',i,'. átlaga: ');
        ReadLn(Oszt[i])
      end;
    WriteLn;
  end;
function Atlag(Oszt: Tnaplo; Letsz: integer): real;
  var i: integer;
      sum: real;
  begin
    sum := 0;
    for i := 1 to Letsz do
      sum := sum + Oszt[i];
    Atlag:= sum / Letsz
  end;
begin
  ClrScr;
  WriteLn('Irja be az A osztály tanulóinak átlagait: ');
  Beolvas(A_oszt, A_letsz);
  WriteLn('Irja be az B osztály tanulóinak átlagait: ');
  Beolvas(B_oszt, B_letsz);
  WriteLn('Az A osztály átlaga: ', Atlag(A_oszt, A_letsz):4:2 );
  WriteLn( 'A B osztály átlaga: ', Atlag(B_oszt, B_letsz):4:2 );
  ReadKey;
end.
```

Függvény - Function

A függvény feladata egy érték előállítása. Ezt az értéket a függvény nevéhez rendeljük, a függvény törzsében kell szerepelni legalább egy értékadó utasításnak, amelyben a függvény neve a baloldalon áll. (Vigyázzunk, ha a jobboldalon szerepeltetjük a függvény nevét, akkor az már rekurziót jelent [2].

A függvényt egy kifejezésben hívhatjuk meg, pl. egy értékadó utasítás jobboldalán.

Szerkezete megegyezik az eljárásával azzal a különbséggel, hogy még meg kell határoznunk a viszatérési érték típusát is. Így a függvény feje:

FUNCTION *azonosító* [**(** *formális paraméter lista* **)**]**:** *típusazonosító***;**

ahol a típusazonosító csak csak sorszámozott, valós, karakterlánc vagy mutató lehet.

Pl.

```
function Tangens(Alfa: real): real;
begin
  if cos(Alfa) <> 0 then
    Tangens := Sin(Alfa) / Cos(Alfa)
end;
```

Rekurzió

Ha egy alprogram saját magát meghívja, akkor rekurzióról beszélünk. Megkülönböztethetünk közvetlen és közvetetten rekurziót.

A rekurzió alkalmazásának egyik területe, amikor úgy oldunk meg egy problémát, hogy visszavezetjük egy egyszerűbb esetre, majd ezt addig folytatjuk, míg el nem jutunk a triviális esetig. A módszer a matematikai indukción alapszik [3].

A megoldás lépései:

1. Megkeressük azt a legegyszerűbb esetet, ahol a megoldás már magától értetődő - triviális eset. Ekkor áll le a rekurzív hívások sorozata.
2. Megvizsgáljuk, hogy ismételt egyszrűsítésekkel hogyan juthatunk el a triviális esethez. (Az általános esetet visszavezetjük az eggyel egyszerűbbre.)

Példák [1]:

1. Faktoriális számítás

- Triviális eset: 1! = 1
- Egyszerűsítés: N! = N*(N-1)!

Ezzel a problémát megoldottuk, már csak kódolnunk kell.

```
program Faktorialis;
uses Crt;
var N: integer;
```

```
function  FaktIter(N: integer): longint;  {Iterációs megoldás}
var i: integer;
    fakt: longint;
begin
  fakt := 1;
  for i: = 2 to N do fakt := fakt * i;
  FaktIter := Fakt
end;

function FaktRek(N: integer): longint;    {Rekurzív megoldás}
begin
  if N = 1 then
    FaktRek := 1                        {Triviális eset}
  else
    FaktRek := n * FaktRek(n-1)         {Visszavezetés az egyszerűbbre}
end;

begin
  ReadLn(N);
  WriteLn(FaktIter(N));
  WriteLn(FaktRek(N));
  ReadKey
end.
```

Megj.: Bár a feladat kitűnő példa a rekurzív algoritmusra, az iterációs (ciklussal történő) megodás jobb, mivel az ismételt függvényhívások időigényesek.

2. Fibonacci sorozat (1, 1, 2, 3, 5, 8, 13...) N. eleme
- Triviális eset: az első és a második elem értéke 1.
- Egyszerűsítés: az N. elem az N-1 - edik és az N-2 - dik elemek összege **[1]**.

```
program Fibon_Rek;
uses Crt;
var n: byte;

function Fibo(n: byte): integer;
begin
  if (n = 0) or ( n = 1) then
    Fibo := 1                   {Triviális eset}
  else
    Fibo := Fibo(n-1) + Fibo(n-2)  {Visszavezetés az egyszerűbbre}
end;

begin
  ClrScr;
```

```
  ReadLn(n);
  WriteLn(Fibo(n));
  ReadKey
end.
```

Állománykezelés

A programok a bemeneti adataikat nem csak a billentyűzetről, hanem a háttértárolókon lévő állományokból is kaphatják, valamint kimeneti adataikat a képernyőn történő megjelenítés mellett állományokban is tárolhatják. A Pascal nyelvben három összetett típus és az ezekhez kapcsolódó szabványos eljárások és függvények valósítják meg az állományok kezelését [4].

Típusos állomány

Deklarálása: **FILE OF** *alaptípus*

Összetett típus, fizikailag egy lemezes állomány. Egyforma méretű elemekből (komponensekből) áll. Az elemek számának csak a lemez mérete szab határt.

A típusos állományból való olvasás illetve az állományba való írás egysége a komponens.

Az elemekhez a rendszer sorszámot rendel 0-tól kezdődően. Az elérés szekvenciálisa (Read, Write) vagy a komponensek sorszáma szerint direkt módon történhet (az állomány mutató mozgatásával) [5].

A program mindig egy logikai állományt kezel, melyet hozzá kell rendelnünk egy fizikai állományhoz (Assign), majd használat előtt meg kell nyitnunk. A Rewrite eljárás létrehozza, és megnyitja a logikai fájlhoz rendelt fizikai állomány. Ha a fizikai fájl már létezett, akkor törli annak tartalmát. A Reset eljárással egy már létező állományt nyithatunk meg. Ekkor az állománymutató az 0. komponensre áll. (Ezért ezt az eljárást használhatjuk egy nyitott állomány elejére való ugrásra is.) Használat után a Close eljárással zárjuk le fájlunkat! [1]

A típusos állományból a Read eljárás olvas be változókba adatokat. Ügyeljünk arra, hogy a változó típusa egyezzen meg a fájl alaptípusával! Beolvasás után az állomány mutató automatikusan a következő komponensre lép (szekvenciális elérés). Egy változó (vagy kifejezés) értékét a Write eljárással írhatjuk ki egy fájlba. Hasonlóan az olvasáshoz a változó típusának meg kell egyeznie a fájl elemeinek a típusával, valamint az eljárás után az állomány mutató továbblép. Ha az állomány mutató a fájl végén (az utolsó elem mögött) áll, akkor az Eof függvény értéke True. Nézzünk egy példát a fájl szekvenciális feldolgozására:

```
  Reset(f)
  while not Eof(f) do
    begin
      Read(f,v);
      {a v változóban lévő adat feldolgozása}
    end;
```

Az állomány mutató direkt pozicionálását a Seek eljárás valósítja meg. A FilePos függvénnyel lekérdezhetjük az aktuális pozíciót, a FileSize függvény pedig az állomány elemeinek a számát (méretét) adja vissza.

Az I/O műveletek során nagy a hibalehetőség (pl. a lemezegység, fájl nem elérhető). Az esetleges futási hibákat tudnunk kell kezelni, ha megbízhatóan működő programot szeretnénk írni. Ha az I/O műveletek ellenőrzése aktív (ez az alapértelmezés), akkor programunk futási hibával leáll egy I/O hiba esetén. Ezért I/O műveletek ellenőrzését inaktívvá kell tennünk a {$I-} fordítási direktívával a kényes műveletek esetén. A művelet után az esetleges hiba kódját az IOResult függvénnyel kérdezhetjük le [12]. Erre egy példa:

```
Assign(f, 'adatok.dat');
{$I-}
Reset(f);                {megpróbáljuk megnyitni a fájlt}
{$I+}
if IOResult <> 0 then     {ha hiba történt, tehát a fájl nem létezik, }
  Rewrite(f);             {akkor létrehozzuk az állományt}
```

A Truncate eljárással levághatjuk a fájl komponenseit az aktuális pozíciótól kezdődően.

Lezárt állományokra használhatjuk a Rename valamint az Erase eljárásokat a fájlok átnevezésére illetve törlésére.

Példa [1]:

1. A program egy bolt árucikkeinek adatait (név, kód, ár) tárolja és kezeli egy állományban.

```
 program aruk;
uses crt;
type TAru = record        {A fájl alaptípusa.}
       kod: string;
       nev: string[15];
       ar: real;
       t: boolean;      {Ez a mező jelzi, hogy e rekord törölt-e (logikai törlés).}
     end;

var bolt: file of TAru;
   aru: TAru;
   mkod: string;
   mvalasz: char;

{Megkeres egy adott kódú rekordot az állományban.}
function Van(kodja: string): boolean;
var talalt: boolean;
begin
```

```
  seek(bolt,0);
  talalt := false;
  while not Eof(bolt) and not talalt do
    begin
      read(bolt, aru);
      if (aru.kod = mkod) and not aru.t then
         talalt := true;
     end;
  van := talalt;
end;

{Egy rekord felvitele az állományba.}
procedure Bevitel;
begin
  ClrScr;
  WriteLn('Kerem a kodot!'); ReadLn(mkod);
  if not Van(mkod) then
    begin
      Seek(bolt, filesize(bolt));          {Pozicionálás a fájl végére.}
      WriteLn('Kerem az aru nevet!');
      ReadLn(aru.nev);
      WriteLn('Kerem az aru arat!');
      ReadLn(aru.ar);
      aru.t := false;
      aru.kod := mkod;
      Write(bolt, aru);
    end
  else
    begin
      WriteLn('Mar van ilyen kod!');
      ReadKey
    end
end;

{Egy rekord módosítása a fájlban.}
procedure Modosit;
begin
  ClrScr;
  WriteLn('Kerem az aru kodjat!');
  ReadLn(mkod);
  if Van(mkod) then
    begin
      Seek(bolt, FilePos(bolt) - 1);
```

```
      WriteLn('Kerem az aru nevet!');
      ReadLn(aru.nev);
      WriteLn('Kerem az aru arat!');
      ReadLn(aru.ar);
      aru.t := false;
      aru.kod := mkod;
      Write(bolt, aru);
    end
  else
    begin
      Writeln('Nincs ilyen aru!');
      ReadKey
    end;
end;

{Egy rekord logikai törlése: a t mezőt True értékűre állítja,
 az ilyen rekordokat a program nem létezőnek tekinti.
 Fizikai törlés kilépéskor.}
procedure Torles;
begin
  ClrScr;
  WriteLn('Kerem az aru kodjat!');
  ReadLn(mkod);
  if Van(mkod) then
    begin
      Seek(bolt, FilePos(bolt) - 1);
      aru.t := true;
      Write(bolt, aru);
    end
  else
    begin
      WriteLn('Nincs ilyen aru!');
      ReadKey
    end
end;

{A fájl tartalmának kiírása a képernyőre.}
procedure Lista;
begin
  ClrScr;
  Seek(bolt, 0);
  while not Eof(bolt) do
    begin
```

```
      Read(bolt, aru);
      if aru.t = false then
        begin
          Write(aru.kod);
          GotoXy(30, wherey); write(aru.nev);
          GotoXy(60, wherey); writeln(aru.ar:10:0);
        end;
    end;
   ReadKey
end;

{Fizikai törlés: azon rekordok átmásolása egy új állományba, melyek nincsenek
logikailag törölve.
 A régi állomány törlése, az új fájl átnevezése a régi nevére.}
procedure Surites;
var ujfile: file of TAru;
begin
  Assign(ujfile, 'ujfile');
  Rewrite(ujfile);
  Seek(bolt, 0);
  while not Eof(bolt) do
    begin
      Read(bolt, aru);
      if aru.t = false then write(ujfile, aru);
     end;
  Close(bolt);
  Erase(bolt);
  Close(ujfile);
  Rename(ujfile, 'bolt');
end;

{Főprogram, menü.}
begin
  clrscr;
  Assign(bolt, 'bolt');
  {$I-}
  Reset(bolt);
  {$I+}
  if IOResult <> 0 then Rewrite(bolt);
  repeat
    ClrScr;
    WriteLn('1. Adatbevitel');
    WriteLn('2. Modositas');
```

```
    WriteLn('3. Torles');
    WriteLn('4. Listazas');
    WriteLn('5. Vege');
    WriteLn('Valassz!');
    repeat mvalasz := readkey until mvalasz in['1'..'5'];
    case mvalasz of
      '1': bevitel;
      '2': modosit;
      '3': torles;
      '4': lista;
      '5': surites;
    end;
  until mvalasz = '5';
end.
```

Szöveges állomány - Text

Deklarálása: **TEXT**

A Pascal programban szöveges állományként kezelhetjük az egyszerű ASCII szövegeket. (Például a .pas kiterjesztésű forrásprogramjainkat.) A szöveges állomány változó hosszúságú sorokból áll, melyeket a sorvégjel zár le (CR/LF). Az állományt az állományvégjel zárja(^Z). Az Eoln illetve az Eof függvény értéke True, ha az aktuális pozíció egy sorvégjelen vagy az állomány végén áll. A SeekEoln illetve a SeekEof függvények az állomány következő TAB szóköz illetve TAB szóköz és sorvégjel karaktereit átugorva tájékoztatnak arról, hogy sorvégjelen illetve az állomány végén állunk-e [1].

A szöveges állományt csak szekvenciálisan érhetjük el. Az állomány csak olvasásra vagy csak írásra lehet megnyitni. Az állományból olvasni a Read, ReadLn, illetve írni a Write, Writeln eljárásokkal tudunk. Ha az eljárásoknak a fájl azonosító paraméterét elhagyjuk, akkor az olvasás / írás az alapértelmezett input / output szöveges állományból / -ba történik, ami a billentyűzet illetve a monitor. Szöveges állományból (azaz a billentyűzetről is) olvashatunk egész, valós, karakteres és sztring típusú változokba adatokat. Az állományba az előbbi típusokon kívül még logikai értéket is kiírathatunk.

Az fizikai állományhoz az Assign eljárással rendelhetünk egy Text típusú változót, azaz a logikai állományt. A Rewrite eljárás csak írásra nyitja meg a szöveges állományt, ha nem létezett létrehozza, egyébként törli a tartalmát. A Reset eljárással csak olvasásra nyithatunk meg egy már létező fájlt. Az Append eljárás egy létező állományt nyit meg írásra, és az állománymutató a fájl végére állítja. Az állományt a Close eljárással zárhatjuk be [3].

Az I/O műveletek hibakódját az IOResult függvény adja vissza (bővebben ld. Típusos állományok).

Lezárt állományokra használhatjuk a Rename valamint az Erase eljárásokat a fájlok átnevezésére illetve törlésére.

A Fluss és a SetTextBuf eljárásokkal az írás, olvasás során a rendszer által használt átmeneti tárolóhoz (pufferhez) férhetünk hozzá.

Példa [1]:

1. A doga.txt állományban egy feladatsor van, kérdések és válaszok felváltva egymás után. Minden kérdés illetve válasz új sorban kezdődik. A kérdések egy számjeggyel kezdődnek, és kérdőjellel fejeződnek be. Készítsünk két új szöveges állományt úgy, hogy az egyik csak a kérdéseket, a másik pedig csak a válaszokat tartalmazza.

```
program Doga;
var f, k, v: text;
   s: string;
   kerd: boolean;
begin
  Assign(f, 'doga.txt');
  Assign(k, 'kerd.txt');
  Assign(v, 'val.txt');
  Reset(f);
  Rewrite(k);
  Rewrite(v);
  while not Eof(f) do
    begin
      ReadLn(f, s);                          {Egy sor beolvasása a dolgozatból}
      if s[1] in ['1'..'9'] then kerd := true;       {A sor első karaktere számjegy-e
(kérdés)}
      if kerd then WriteLn(k, s) else WriteLn(v, s);  {Kiírás a megfelelő állományba}
      if s[Length(s)] = '?' then kerd := false;       {Vége-e a kérdésnek}
    end;
  Close(f);
  Close(k);
  Close(v)
end.
```

Típusnélküli állomány

Deklarálása: **FILE**

Általában gyors adatmozgatás vagy ismeretlen állomány esetén használjuk. Hasonló a típusos állományhoz, de az elemeinek nem a típusa, hanem a hossza a lényeges. A komponensek hosszát a fájl megnyitásakor adhatjuk meg (Reset, Rewrite), az alapértelmezés 128 bájt. Az állomány írható, olvasható, az elérés szekvenciálisan (BlockRead, BlockWrite eljárásokkal) vagy az elemek sorszáma szerint direkt módon történhet [4].

További függvények, eljárások: Assign, Close, Eof, Erase, FilePos, FileSize, IOResult, Rename, Seek, Truncate.

Példa:

1. Tördeljünk szét egy állományt egy kilobájt hosszúságú kisebb állományokra! **[1]**

```
program Tordel;
uses Crt;
var forras, cel: file;
   n, maradek: integer;
   s: string;
   t: array[1..1024]of byte;
begin
  Assign(forras, 'nagyfajl.arj');
  Reset(forras,1);                          {Megnyitás egy bájt elemhosszúsággal}
  n := 0;
  while not Eof(forras) do
    begin
      inc(n);
      str(n,s);
      Assign(cel, 'kisfajl.'+s);              {A cél állomány hozzárendelése, a
kiterjesztés a sorszám}
      Rewrite(cel,1);
      maradek := FileSize(forras)-FilePos(forras);  {A forrás még át nem másolt
részének a hossza}
      if maradek >= 1024 then
        begin
          BlockRead(forras, t, 1024);
          BlockWrite(cel, t, 1024);
        end
      else
        begin
          BlockRead(forras, t, maradek);
          BlockWrite(cel, t, maradek);
        end;
      Close(cel)
    end;
  Close(forras)
end.
```

Karakteres képernyő kezelése - a CRT unit

A Crt egység a karakteres képernyő, a billentyűzet valamint a hangszóró kezelését segítő függvényeket, eljárásokat tartalmazza. Mint az egységek többsége, a Crt unit is definiál konstansokat, változókat.

Színek:

A karakteres képernyő tartalma megtalálható az ún. képernyő memóriában. Itt egy karaktert két bájton tárol el a rendszer, melyek a karakter ASCII kódja (1 bájt) valamint a karakter attribútuma (1 bájt). Ez utóbbi a színinformációt hordozza, az alábbi módon:

7 6 5 4 3 2 1 0

V R G B I R G B

A 0.-3. bit a karakter tintaszínét határozza meg, R, G, B az additív színkeverés három alapszíne, I pedig az intenzitás. Például 0100 - piros, 1100 - világospiros, 0101 - lila. A 4.-6. bitek a karakter háttérszínét kódolják. Ha a 7. bit (V) egyes, akkor a karakter villog.

A fentiekből következik, hogy összesen 16 tinta- és 8 háttérszínt használhatunk. A színek kódjait könnyen kiszámolhatjuk, ezeket a megfelelő eljárásokban használhatjuk, de a könnyebb megjegyezhetőség kedvéért a Crt unit az alábbi szín konstansokat definiáljam [12].

Tinta- és háttérszínek:			**További tintaszínek:**		
Balck	0	Fekete	**DarkGray**	8	Sötétszürke
Blue	1	Kék	**LightBlue**	9	Világoskék
Green	2	Zöld	**LightGreen**	10	Világoszöld
Cyan	3	Türkiz	**LightCyan**	11	Világostürkiz
Red	4	Piros	**LightRed**	12	Világospiros
Magenta	5	Lila	**LightMagenta**	13	Világoslila
Brown	6	Barna	**Yellow**	14	Sárga
LightGray	7	Világosszürke	**White**	15	Fehér

Blink 128 Villogás

Pl: *TextColor(Lightred+Blink)*, ezzel egyenértékű: *TextColor(12 + 128)* vagy *TextColor(140)*.

Fontosabb eljárások, függvények:

Képernyőkezelés:

Függvények:

WhereX - visszaadja a vízszintes koordinátát

WhereY – visszaadja a függőleges koordinátát

Eljárások:

TextBackground - a háttérszínt állítja be.

TextColor - a szöveg színtét állítja be.

ClrScr - törli a képernyőt.

ClrEol - kurzorpozíciótól kezdve a sor végéig törli a karaktereket.

DelLine - törli a kurzort tartalmazó sort.

InsLine - egy sort helyez a kurzor pozícióba.

GotoXY - a szöveges képernyő x.oszlopába és y.sorába viszi a kursort.

Window - a képernyőn kijelöl egy ablakot, azaz amit itt beállítottál akkora területre tudsz továbbra írni. Ezután az 1,1 koordináta az ablak bal felső sarka lesz.

NormVideo - beállítja a normál karakter fényerejét.

TextMode - szöveges mód beállítása.

Billentyűzetkezelés:

Függvények:

KeyPressed - True értékkel tér vissza, ha a gomb megnyomása megtörtént.

ReadKey - beolvass egy karaktert a billentyűzetről anélkül, hogy a képernyőn megjelenne.

Hang, késleltetés:

Eljárások:

Sound - bekapcsolja a hanggenerátort.

Delay - megállítja a program végrehajtását n milliszekundumig.

NoSound - kikapcsolja a hanggenerátort.

Példa [1]:

1. Mozgassunk egy téglalapot (egy kis képernyőt) benne egy szöveggel a képenyőn a kurzormozgató billentyűk segítségével!

```
program CrtPl;
uses Crt;
var  x1, x2, y1, y2: byte;
    c: char;
begin
  x1:=35; y1:=12; x2:=45; y2:=16;
  repeat
    {A régi ablak törlése}
    TextBackground(black);
    ClrScr;
    {Új ablak és a szöveg megjelenítése}
    Window(x1, y1, x2, y2);
    TextBackground(blue);
    TextColor(red);
```

```
  ClrScr;
  GotoXY(3, 3);
  WriteLn('szoveg');
  {Várakozás egy billentyű leütésére}
  c := ReadKey;
  {Ha a billentyűzetnek két bájtos kódja van (az első bájt #0), a második bájt
beolvasása.
   Ilyenek a kurzormozgató billentyűk.}
  if c = #0 then
   begin
    c := ReadKey;
    case c of
     #72: begin Dec(y1); Dec(y2) end; {Felfele nyíl}
     #80: begin Inc(y1); Inc(y2) end; {Lefele nyíl}
     #77: begin Inc(x1); Inc(x2) end; {Jobbra nyíl}
     #75: begin Dec(x1); Dec(x2) end; {Balra nyíl}
    end;
   end
  {Kilépés ESC-re}
 until c = #27;
 NormVideo;          {Eredeti színek visszaállítása}
 ClrScr
end.
```

A Turbo Pascal grafikája - a GRAPH unit

Ha rajzolni szeretnénk a képernyőre, akkor azt át kell kapcsolnunk grafikus üzemmódba. A grafikus képernyőn külön-külön hozzáférhetünk az egyes képpontokhoz. Ismernünk kell (illetve bizonyos határok között meghatározhatjuk) a képernyőnk felbontását (a VGA üzemmódban a legnagyobb felbontás 640x480) valamint azt, hogy hány színt használhatunk (az előbb említett felbontásnál 16 színt). A (0, 0) képpont a képernyő bal felső sarkában található [12].

Az egység fontosabb eljárásai, függvényei, típusai, konstansai:

A grafikus képrnyő inicializálása (átkapcsolás karakteres képernyőről grafikusra), bezárása:

Eljárások: InitGraph, DetectGraph, CloseGraph, stb.

Függvények: GraphResult, stb.

Konstansok: grafikus meghajtók (Pl. Detect = 0, CGA = 1 stb.); grafikus üzemmódok (pl. VGALo, VGAMed, VGAHi stb.)

Pl. [1]:

```
uses Graph;
var Meghajto, Uzemmod: integer;
```

```
  begin
    Meghajto := Detect; {Automatikusan beállítja grafikus üzemmódot a legnagyobb felbontással.}
    InitGraph(Meghajto, Uzemmod, 'C:\TP70\BGI');  {Inicializálás}
    If GraphResult <> 0 then
      begin
        WriteLn('Grafikus hiba!');  {Nem sikerült az inicializálás, kilépés a programból}
        ReadLn;
        Halt
      end;
    ...{Grafika használata}
    CloseGraph    {Grafikus képernyő bezárása}
  end
```

Színek:

Konstansok: 16 színű üzemmódban megegyeznek a Crt unit konstansaival.
Eljárások: SetColor, SetBkColor, stb.
Függvények: GetColor, GetBkColor, stb.

Rajz:

Típusok: LineSettingsType (ld. GetLineSettings), stb.
Konstansok: vonalstílus (pl. SolidLn, DottedLn, stb. ld. SetLineStyle), vonalvastagság (NormWidth. ThickWidth ld. ld. SetLineStyle), rajzolási mód (pl. CopyPut, XorPut, stb. ld. SetWritMode)
Eljárások: PutPixel, Line, LineTo, LineRel, Circle, Rectangle,SetLineStyle,GetLineSettings, SetWriteMode, stb.

Kitöltött rajz:

Típusok: FillSettingsType (ld. GetFillSettings), stb.
Konstansok: kitöltési stílus (pl. SolidFill, LineFill, stb. ld. SetFillStyle)
Eljárások: Bar, Bar3D, FillEllipse, FloodFill, SetFillStyle, GetFillSettings, stb.

Szöveg:

Típusok: TextSettingsType (ld. GetTextSettings), stb.
Konstansok: betűtípus, szövegállás, szövegigazítás
Eljárások: OutText, OutTextXY, SetTextStyle, GetTextSettings stb.

Kurzor:
Függvények: GetX, GetY, stb.
Eljárások: MoveTo, MoveRel, stb.

Egyéb:
Kép mentése egy változóba, visszatöltése: ImageSize, GetImage, PutImage.

Példa:

```
PROGRAM grafik;
 Uses CRT,GRAPH;
  VAR videokartya, Grafikusmod:integer;
begin
  videokartya:=Detect;
   InitGraph(videokartya,Grafikusmod,'C:\tp\bgi');
  SetBkColor(5);
  SetLineStyle(3, 3, 1);
  SetFillStyle(HatchFill, 1);
  Bar(300, 300, 350, 350);
  Line(10, 10, 200, 200);
  Circle(250, 250, 200);
readkey;
  CloseGraph;
end.
```

Mutatók

Típusos mutató

Deklarálása: *azonosító*: ^*alaptípus*

Pl. Var p1, p2: ^real;

A mutató egy memóriacímet (4 bájtos: szegmens, ofszet) tartalmaz, egy *alaptípusú* változóra mutat. A mutatóhoz futás közben rendelhetünk memóriacímet és így tárterületet (dinamikus adattárolás).

A New eljárás a heap memóriában foglalterületet a mutatott változó számára, a Dispose eljárás pedig felszabadítja a lefoglalt területet. Így lehetővé válik, hogy egy nagy helyfoglalású adatstruktúra számára csak akkor kössünk le memóriát, amikor használjuk a változót. Nézzünk erre egy példát:

```
  type TTomb = array[1..1000]of real;
  var t: TTomb;  {A rendszer már a program idításakor lefoglal 6000 bájtot az
adatszegmensben}
      pt: ^TTomb;{A rendszer a program idításakor csak 4 bájtot foglal le a mutató
számára}
  begin
    ...
    New(pt);     {A heap-ben létrejön a mutatott változó}
    ...          {Használhatom a pt által mutatott változót (pt^)}
    Dispose(pt)  {A terület felszabadul a heap-ben}
  end.
```

A mutatót ráirányíthatjuk egy az alaptípusával megegyező típusú változóra a @ operátorral vagy az Addr függvénnyel. Pl. pt := @t; vagy pt := Addr(t).
A Ptr függvénnyel a mutatónak egy tetszőleges memóriacímet adhatunk értékül.

A negyedik lehetőség arra, hogy egy mutatóhoz egy memóriacímet rendeljünk: értékadó utasítás egy vele azonos alaptípusú mutatóval.

Hivatkozás a mutatott változóra: *mutató-azonosító^*
(pl.: pt^[12] := 1)

Mutató típusú konstans: Nil.
A Nil nem mutat sehová. (Pl. láncolt lista végének a jelzésére használhatjuk).

Műveletek:
Címe: @
Egyenlőség vizsgálat: =, <>

További szabványos eljárások, függvények:
Eljárások: Mark, Release
Függvények: MaxAvail, MemAvail, Ofs, Seg

Láncolt listák

A különböző típusú láncolt listák nagyon fontos adatszerkezetek a számítástechnikában. Az egyes adatelemek (rekordok) közötti csatolást mutatókkal valósíthatjuk meg, így a rekordnak van egy mutató típusú mezője, melynek alaptípusa maga a rekord. Az alábbi példában figyeljük meg, hogy a mutató típus deklarálásánál olyan azonosítót használunk, amely a programunkban csak később szerepel. (A Pascal ebben az egy esetben engedi ezt meg.) [12]

```
type Mutato = ^Adatelem;
     Adatelem = record
       Adat: real;
       Kovetkezo: Mutato;
     end;
var Elso, Uj, Aktualis: Mutato;
```

Példák [1]:
1. Fűzzük fel láncolt listára a billentyűzetről beolvasott számokat. Írassuk ki a listát!

```
program Lista1;
uses Crt;
type TMutato = ^TAdatElem;
    TAdatElem = record
      Adat: integer;
      Kovetkezo: TMutato;
    end;
var  Elso, Aktualis, Uj: TMutato;
    a: integer;
begin
  ClrScr;
```

```
  ReadLn(a);
  Elso := nil;
  while a <> 0 do
    begin
      New(Uj);                    {Az Uj mutatóhoz memóriaterület rendelése}
      Uj^.Adat := a;
      Uj^.Kovetkezo := nil;
      if Elso = nil then
        Elso:=uj                  {Első rekord, az Elso mutat az Uj-ra}
      else
        Aktualis^.Kovetkezo := Uj;  {Az Aktualis-hoz csatoljuk az Uj-at}
      Aktualis := Uj;             {Az Aktualis léptetése}
      ReadLn(a)
  end;
  {A lista megjelenítése}
  WriteLn;
  Aktualis := Elso;
  while Aktualis <> nil do
    begin
      WriteLn(Aktualis^.Adat);
      Aktualis := Aktualis^.Kovetkezo
    end;
  ReadKey
end.
```

2. Fűzzük fel rendezett láncolt listára a billentyűzetről beolvasott számokat. Írassuk ki a listát!

```
program Lista2;
uses Crt;
type TMutato = ^TAdatElem;
     TAdatElem = record
       Adat: integer;
       Kovetkezo: TMutato;
     end;
var  Elso, Aktualis, Uj, Elozo: TMutato;
     Szam: integer;
begin
  ClrScr;
  Elso := nil;
  ReadLn(Szam);
  while Szam <> 0 do
    begin
      New(Uj);
```

```
    Uj^.Adat := Szam;
    Uj^.Kovetkezo := nil;
    {Az Uj rekord helyének a megkeresése a rendezett listában}
    Aktualis := Elso;
    while (Aktualis <> nil) and (Aktualis^.Adat < Uj^.Adat) do
      begin
        Elozo := Aktualis;
        Aktualis := Aktualis^.Kovetkezo;
      end;
    {Az Uj rekord beillesztése a listába}
    if Aktualis = Elso then   {Az Elso elem lesz}
      Elso := Uj
    else                      {Az Elozo mögé, az Aktualis elé}
      Elozo^.Kovetkezo := Uj;
    Uj^.Kovetkezo := Aktualis;
    ReadLn(Szam)
   end;
  {A lista megjelenítése}
  WriteLn;
  Aktualis := Elso;
  while Aktualis<>nil do
   begin
    WriteLn(Aktualis^.Adat);
    Aktualis := Aktualis^.Kovetkezo;
   end;
  readkey
end.
```

Típusnélküli mutató - Pointer

Deklarálása: POINTER

Egy négy bájtos memóriacímet tartalmaz. A mutatott változónak nincs típusa. A típusnélküli műveleteknél használjuk (pl. adatmozgatás).

A GetMem eljárással foglalhatunk le egy megadott méretű területet a heap-ben a mutatónk számára. Ha a memóriaterületre már nincs szükségünk a FreeMem eljárással azt felszabadíthatjuk [1].

A mutatót ráirányíthatjuk akármilyen címre vagy változóra a @ operátorral vagy az Addr függvénnyel. Pl. p := @tomb1; vagy p := Addr(tomb1) [8].

A Ptr függvénnyel a mutatónak egy tetszőleges memóriacímet adhatunk értékül.

Egy mutató értékadó utasítással egy másik mutató címét is felveheti.

A mutatott változóhoz rendelhetünk típust: *típus***(***mutató-azonosító***^)**.

Hivatkozás a mutatott változóra: *mutató-azonosító*^

Mutató típusú konstans: Nil.

Műveletek:

Címe: @

Egyenlőség vizsgálat: =, <>

További szabványos függvények: MaxAvail, MemAvail, Ofs, Seg.

Példa [1]:

1. Mentsünk el egy garfikát egy típus nélküli állományba, majd olvassuk vissza!

```
program KepMent;
uses Crt, Graph;
var f: file;          {A típusnélküli állomány, amelybe mentjük a képet}
   kep: pointer;     {A kép memóriába való elmentéséhez szükséges mutató}
   d, m: integer;
   meret: word;
   grd, grm, a, b, c, px, py, py1, i: integer;
   x, y: real;
begin
 {Grafika kirajzolása (két függvény ábrázolása)}
 d := detect;
 InitGraph(d, m, 'c:\bp\bgi');
 a := 1;
 b := -2;
 c := -3;
 for i := 0 to 640 do PutPixel(i,240,14);
 for i := 0 to 64 do begin PutPixel(10*i,241,14); PutPixel(10*i,239,14) end;
 for i := 0 to 480 do PutPixel(320,i,14);
 for i := 0 to 48 do begin PutPixel(319,10*i,14); PutPixel(321,10*i,14) end;
 for i := -320 to 320 do
  begin
    x := i/10;
    y := 5*Sin(x/4);
    px := Round(10*x + 320);
    py := Round(-10*y + 240);
    py1 := Round(-10*x + 240);
    if (py < 480) and (py > 0) then PutPixel(px,py,15);
    if (py1 < 480) and (py1 > 0) then PutPixel(px,py1,15)
  end;

 Assign(f, 'kepment.dat');
 Rewrite(f);
 {A képet négy részletben tudjuk elmenteni}
 meret := ImageSize(0, 0, 319, 239); {A negyedkép mérete}
 GetMem(kep, meret);              {Helyfoglalás a típusnélküli mutatónak a heap-ben}
 GetImage(0, 0, 319, 239, kep^);     {Negyedkép elmentése a mutatott területre}
```

```
  BlockWrite(f, kep^, meret div 128); {A memóriaterület elmentése a fájlba}
  GetImage(320, 0, 639, 239, kep^);
  BlockWrite(f, kep^, meret div 128);
  GetImage(0, 240, 319, 479, kep^);
  BlockWrite(f, kep^, meret div 128);
  GetImage(320, 240, 639, 479, kep^);
  BlockWrite(f, kep^, meret div 128);
  FreeMem(kep, meret);              {A memóriaterület felszabadítása}

  {A kép visszaolvasása}
  ReadKey;
  ClearDevice;
  ReadKey;
  Reset(f);
  GetMem(kep, meret);               {Helyfoglalás a típusnélküli mutatónak a heap-ben}
  BlockRead(f, kep^, meret div 128);  {Adatmozgatás a fájlból a lefoglalt
memóriaterületre}
  PutImage(0, 0, kep^, copyput);      {A negyedkép kirajzolása}
  BlockRead(f, kep^, meret div 128);
  PutImage(320, 0, kep^, copyput);
  BlockRead(f, kep^, meret div 128);
  PutImage(0, 240, kep^, copyput);
  BlockRead(f, kep^, meret div 128);
  PutImage(320, 240, kep^, copyput);
  FreeMem(kep, meret);              {A memóriaterület felszabadítása}

  ReadKey;
end.
```

Saját unit készítése

Az egységek (unitok) előre lefordított programmodulok. Általában egy adott területhez tartozó eljárásokat, függvényeket tartalmaznak, illetve deklarálják az általuk használt konstansokat, típusokat, változókat. Mivel a kódszegmens maximálisan 64 kB lehet, így programunk nagysága is korlátozott. Ha elértük a határt (kb. 2-3000 programsor), akkor programunk egyes részeit saját unitokban helyezhetjük el, melyek külön-külön szintén 64 kB méretűek lehetnek [1].

Az egység felépítése:

Egységfej:

Unit *azonosító*;

Az azonosítót kell megadnunk a Uses kulcsszó után abban a programban vagy egységben, ahol a unitot használni szeretnénk, továbbá az azonosító legyen a neve az elmentett forrásnyelvű unitnak.

Illesztő rész:

INTERFACE
[**USES** *azonosító* [**,***azonosító*...]**;**]

Továbbá globális deklarációk (konstansok, típusok, címkék, változók, eljárások, függvények), melyeket az egységet használó programokban illetve egységekben is elérhetünk. Az eljárásoknak, függvényeknek itt csak a fejlécei szerepelnek.

Kifejtő rész:

IMPLEMENTATION
[**USES** *azonosító* [**,***azonosító*...]**;**]

Továbbá egység hatáskörű deklarációk (konstansok, típusok, címkék, változók, eljárások, függvények), melyeket csak ebben az egységben érhetünk el. Itt fejtjük ki az Interface részben deklarált eljárásokat, függvényeket is [11].

Inicializáló rész:

[**BEGIN**
[*utasítás* [**;** *utasítás*...]]
END.

A főprogram első utasítása előtt egyszer végrehajtódik, elhagyható.

Ha több egység egymást kölcsönösen használja, akkor mindegyikben a többi egység nevét az implementációs rész Uses kulcsszava után kell megadni.

Ha az egységet elkészítettük, .pas kiterjesztéssel metjük lemezre. Az egységet le kell fordítanunk (a fordítást lemezre kérjük) [12]. A lefordított egység kiterjesztése .tpu lesz.

Példa [1]:
Az egérkezelést megvalósító rutinokat lehelyezhetjük egy unitban.

```
unit EgerUnit;
interface
  procedure Init;
  procedure Be;
  procedure Ki;
  function LeBal: boolean;
  function LeJobb: boolean;
  function XKoor: integer;
  function YKoor: integer;

implementation
uses Dos;
```

```
var  r: registers;
procedure Init;
  begin
    r.ax := 0;
    Intr($33, r);
  end;
procedure Be;
  begin
    r.ax := 1;
    Intr($33, r);
  end;
procedure Ki;
  begin
    r.ax := 2;
    Intr($33, r);
  end;
function LeBal: boolean;
  begin
    r.ax := 3;
    Intr($33, r);
    LeBal := r.bx=1;
  end;
function LeJobb: boolean;
  begin
    r.ax := 3;
    Intr($33, r);
    LeJobb := r.bx=2;
  end;
function XKoor: integer;
  begin
    r.ax := 3;
    Intr($33, r);
    XKoor := r.cx;
  end;
function YKoor: integer;
  begin
    r.ax := 3;
    Intr($33, r);
    YKoor := r.dx;
  end;
end.
```

Kérdések

1. A Turbo Pascal környezete.
2. A program általános felépítése Turbo Pascalban. Az azonosító fogalma és használata.
3. Adat típusok a Turbo Pascalban. Egyszerű szabványos adattípusok a Turbo Pascalban.
4. Változók és állandók. Konstansok és változók leírására.
5. Szabványos eljárások használata a Turbo Pascalban.
6. Turbo Pascal nyel operátorai. Az operátor fogalma. Az operátorok típusai. Egyszerű operátorok.
7. Aritmetikai és logikai kifejezések. Bejegyzések szabályai. Összetett parancsok. Logikai zárójelek.
8. Szabványos egyszerű adattípusok be- és kimenetének szerkesztése. Formázott adatkimenet.
9. Turbo Pascal vezérlőnyelvi konstrukciók.
10. Összetett vagy strukturált Turbo Pascal operátorok. If-Then-Else, Case szelektor.
11. Ciklikus operátorok While-Do, Repeat-Until.
12. Összetett statikus adatstruktúrák.
13. Felsorolt típus és intervallum típus.
14. Felsorolt típus tulajdonságai.
15. Felsorolt típus és intervallum típusú állandók.
16. Tömbök fogalma. Tömbök leírásának szintaktikája. Egy dimenziós tömb elemeinek elérése.
17. Tárterület foglalás. A FOR ciklus a tömbök ki- és bevitelénél.
18. Karakterlánc. A karakterlánc - String típus. A szabványos eljárások használata a String típus adatok feldolgozásánál.
19. Összetett adattípusok.
20. Két dimenziós tömbök. Két dimenziós töm elemének az elérése. Tárterület foglalás. Két dimenziós tömbök használata. Tömb típusú állandók bejegyzésének szintaktikája.
21. Record típus. A Record típus szintaktikája. Mezőhivatkozás. A With parancs. Record típust felhasználó algoritmusok.
22. Összetett adatstruktúrák használata.
23. Saját összetett adatstruktúrák létrehozása a tömb és a Record típusok segítségével.
24. Alprogramok.
25. Eljárásokon alapuló programozási elvek.

26. Alprogram fogalma.
27. Alprogramok fajtái a Turbo Pascalban.
28. Eljárások szintaktikája. Eljárás meghívása.
29. Függvények szintaktikája.
30. Függvények meghívása.
31. Alprogramok paraméterei.
32. Formális és aktuális pataméterek. Adat csere.
33. Globális és lokális paraméterek fogalma és használata.
34. Alprogramok egymásba ágazódása.
35. Rekurzió. Rekurzív alprogramok leírása és használata.
36. Dinamikus változók és mutatok.
37. Dinamikus változók szintaktikája és használata.
38. Eljárások: GetMem, FreeMem, New, Dispose.
39. Dinamikus tömbök. A dinamikus tömbök kezelésére szolgáló szerkezet és eljárások leírása.
40. Memória lefoglalása egy dinamikus tömbhöz. Hozzáférés az elemekhez.
41. Kapcsolódó adatszerkezetek.
42. Láncolt listák.
43. Láncolt listák létrehozása és használata.
44. Állományok fogalma. Állományok típusai.
45. Állomány típusú változók használata.
46. Állományok megnyitása és bezárása.
47. Szöveges állomány – Text.
48. Adatcsere az állomány és a program között.
49. Típusos állományok tulajdonságai, használatuk (adatok be illesztése, áthelyezése, törlése).
50. Típusnélküli állományok használata.
51. Adatok mentése típusnélküli állományokban.
52. Saját unit készítése.
53. Unit fogalma.

II Fejezet

Az objektum orientált programozás alapjai (objektum, metódus, egtségbezárás, öröklődés)

Az objektum orientált programozás (Object Oriented Programming = OOP) három legfontosabb tulajdonsága:

Egységbezárás (encapsulation). Az adatstruktúrákat és az azokat kezelő metódusokat egy egységként kezeljük és elzárjuk őket a külvilág elől. Az így kapott egységeket **objektumoknak** nevezzük.

Öröklés (inheritance). A meglévő objektumokból levezetett újabb objektumok öröklik a definiálásukhoz használt alapobjektumok adatstruktúráit és metódusait, de lehetőség van új adatok definiálására illetve az egyes metódusok átdefiniálására illetve azonos néven történő újradefiniálásra (redefine, override).

Többrétegűség (polymorphism). Egy adott metódus azonosítója közös lehet egy adott objektumhierarchián belül, ugyanakkor a hierarchia minden egyes objektumában a tevékenységeket végrehajtó metódus implementációja az adott objektumra nézve specifikus lehet (pl: virtuális metódusok).

Objektumorientált programnyelvek:C++, Object Pascal, Java.

Hozzáférési jogok:

nyilvános: public

saját: private.

védett: protected.

publikált: published.

Az Object Pascalban az objektum felhasználói típusként (CLASS osztály) jelenik meg, mellyel változókat, objektum példányokat (instance) hozhatunk létre. A **CLASS** típus példányai dinamikusan jönnek létre és minden új típusnak van elődje.

Általában minden osztály adatmezőket (field), metódusokat (methods) és jellemzőket (properties)tartalmaz.

Adatmező:az osztály minden obj.példányában megtalálható, kezelése és deklarálása a rekordmezőével megegyező.

Metódus:az objektumon végzendő műveleteket definiáló eljárások és függvények. Azonos osztályhoz tartozó objektumpéldányok a metódusokat közösen használják. A Self paraméter jelöli, hogy éppen melyik példány adatmezőin kell műveletet végezni.

Kiemelt metódusok: **Constructor:** az objektum létrehozásával, inicializálásával kapcsolatos műveletek megoldása.

Destructor: Az objektum megszüntetésével kapcsolatos műveletekre.

Osztály metódus: Az objektumpéldány helyett az osztályon fejti ki hatását, innen nem elérhetők az adatmezők és a jellemzők.

Jellemzo: Az osztály névvel ellátott attribútuma, melyre csak az olvasás és/vagy az írás műveletét definiáljuk. A Delphi által támogatott komponens orientált programfejlkesztés alapját a jellemzők képezik.

Komponensek: Olyan objektumok, melyeknek jellemzőik tervezési és futási időben egyaránt elérhetőek.

Adatrejtés elve: Az objektum adatmezői és metódusai alaphelyzetben korlátozás nélkül elérhetők, ez ellentmond az OOP adarejtési(data hiding)elvének. Általános szabály: az objektum mezőit csak a metódusok felhasználásával érhetjük el. Az Object Pascalban ennek a szabálynak úgy szerezhetünk érvényt, ha az objektum definicióját a unit inteface részébe tesszük, és a private (belső), illetve public(külső) kulcsszavak segítségével kijelöljük az objektum részeit. Az objektum protected kulcsszóval deklarált része private elérésű a külvilág számára, de ha saját osztályt származtatunk a védett elemekkel rendelkező osztályból, akkor public elérésűvé válnak az összetevők.Az osztályon belül a fenti kulcsszavakkal megjelölt részek teszőleges számban és sorrendben helyezkedhetnek el, de megkötés, hogy az adatok után a metódusok következnek.

Az objektum példányai: Adott osztálytípussal több objektum példányt is létrehozhatunk. A példányok saját adatterülettel rendelkeznek, de a metódusokat közösen használják. Az aktuális objektumpéldány címét a Self paraméter adja, mely minden objektum utolsó nem látható paramétere. Ezért metódus paramétereként nem használhatjuk a Self nevet, de metóduson belül a Self használható a mezőhivatkozásban, vagy objektumok címének lekérdezésére.

Dinamikus helyfoglalású objektumpéldányok: Az objektum számára memóriaterületet constructorral foglalunk és destruktorral szabadítjuk fel a foglalt területet. Osztály objektum esetén a hagyományos new és dispose memóriakezelő eljárások nem használhatók.

Objektumok hierarchiája: Object Pascalban a származtatott osztálynak csak egyetlen közvetlen őse lehet. Itt az egyedi osztályok helyett az egymásra épülő osztályok hierarchiája biztosítja az objektumorientált megközelítés előnyeit.

Osztályok: os osztály(ancestor), amibol az újat származtatjuk, származtatott osztály(descendant)

Tobject osztály: Ez az osztály előre definiált, ez minden osztály közös őse. Ha az osztály deklarációjában elmarad az ős megadása, akkor az új tipus automatikusan TObject lesz.

Sokoldalúság az osztály hierarchiában: Egy származtatott objektum tulajdonságainak megváltoztatását virtuális és dinamikus metódusokon keresztül tudjuk megvalósítani. Tehát attól függően, hogy a program futása során mely részén vagyunk az osztály hierarchiának, azonos hivatkozás esetén más-,más metódus kerül végrehajtásra, azaz a program futása során dől el, hogy melyik metódust kell aktivizálni. Ez a jelenség a késői kötés, late binding, míg a fordítás során megvalósított kötés az early bilding. A virtuális metódusokat tartalmazó osztályok esetén a fordító virtuális metódustáblákat készíit.(VMT Virtual Method Table) Ez a táblázat az alapja a késői hivatkozás feloldásnak., mert a virtuális metódusok a VMT tábla alapján kerülnek meghívásra. A virtuális metódusok hívása gyorsabb, de sok memóriát foglalnak VMT táblázatok. A dinamikus metódusok adattáblái láncot alkotnak és csak az adott osztályban definiált dinamikus metódusokról tárolnak

információt és a metódus belépési címét rendszerrutin keresi meg.

Osztályoperátorok: Az is operátort dinamikus típusellenőrzésre használjuk, segítségével megtudhatjuk, hogy egy adott típusú objektum a megadott oszályhoz tartozik-e vagy sem. Az as operátort típuskonverzió kijelölésére használjuk.

Üzenetkezelés: Üzenetkezelő metódusok segítségével mi magunk is fogadhatunk és küldhetünk üzeneteket. Üzenetek fogadása Message direktívával deklarált üzenetfogadó metódussal. Egy adott objektumnak az objektum Dispatch metódusával küldhetünk üzenetet.

Az objektum-orientált programozás (OOP) a 90-es évek uralkodó stílusirányzata, s egyre inkább felváltja a - lassan már elavulttá váló, de ugyanakkor még klasszikusnak is számító - struktúrált programozást. Az objektum-orientált programozás jobban megközelíti, utánozza a valóságot, és jobban igazodik a tárgyakhoz. Minden valóságos tárgyat nemcsak alakja, elhelyezkedése jellemez, (Tehát nem csak a rá jellemzô adatok -méretek-.) hanem az is, hogyan viselkednek bizonyos körülmények között. Így a tárgyakat minden jellemzőivel együtt komplex egészként tekintjük. (Az-az, az objektum az adatok és jellemzőjük komlpexe, elválaszthatatlan egésze.) Amikor egy objektumot deklarálunk akkor írjuk le szerkezetét a mezőkkel, másrészt azokat a szubrutinokat, amelyek leírják az objektum viselkedését. Az első magasszintű programozási nyelv mely ezt tökéletesen támogatja, ez a: Borland Pascal. Az objektumokat a Pascal nyelv az 5.5-ös verziótól támogatja. Az 5.5-ös verzió még nem követte az objektum-orientált programozás elméletét, majd 6.0-ás verzió kezde el kiegészíteni ezeket a hiányosságokat. Az jelenlegi (7.0) verzió - szinte - tökéletesen igazodik az elmélethez, mind: az öröklés, a profilizmus, a zártság, a sokoldalúság, és az adatrejtés elvét tekintve. Ezenkívül a Borland Pascal nyelvhez tartozik - kiegészítésként - a Turbo Vision mely segít egy objektum-orientált felhasználói felület kialakításában.

Az Objektumokról: Egy objektum négy fő részből áll ezek: adatelemek, szerkezeti összefüggések, szelekciós műveletek, konstrukciós műveletek

Egy objektum négy fő tulajdonsága:

1. Adat és kód kombinációja
2. Öröklés
3. Profilizmus
4. Zártság

objektum = adat + kód (Ettôl objektum, az objektum; mert e kettônek elválaszthatatlan egészén értjük az objektumot!)

Az objektum egyik alkotóeleme az adat, vagy adatszerkezet. Ez a rekordhoz hasonlóan deklarált adatokat jelent. E részben tulajdonképpen a valóságot ábrázoljuk. (Úgymond: a tárgy méreteit) A másik a kód, amelyen olyan eljárások és függvények összességét értjük, amelyek leírják az objektum viselkedésmódját. Szintaktikája a rekordéval - majdnem teljesen - megegyezik, a különbség annyi, hogy metódusokat - procedure-ákat, és function-öket - is deklarál(hat)unk.

Öröklés: (Egy egyszeru evolúciós példán keresztül.) A kétéltűekből származnak a hüllők. (A biológia mai állása szerint.) A hüllők ugyanúgy rendelkeznek gerinccel,

lábakkal, tüdővel, mint a kétéltűek.

De rendelkeznek új tulajdonságokkal is pl: nincs szükség a vízre a peterakáshoz, -mint a kétéltűeknél- az egyedfejlődés első szakaszában. Az objektumot leírva ez így néz ki:

```
TKeteltuek = Object(THalak)
agassag, Súly: Type; {Ez maga az adat}
procedure Tulajdonsagok(); {ez a kód}
function Tulajdonsagok(): Type;
{esetleg még új adatok}
AzObjektumraJellemzőUjValtozó(k): Type;
end;
THullok = Object(TKeteltuek)
{Az örökölt metódusokat tartalmazza}
AzObjektumraJellemzoUjValtozo(k):Type
procedure Szaporodas;
{nem szükségeli a vizet a petelerakás,hiszen már lágy héjú tojást rak}
procdeure EgyedfejlodesElsoSzakasza;
{egyéb új metódusok}
end;
```

Profilizmus: Amint láttuk az új objektum (jelen esetben az új osztály) tartalmazza a régi metódus(oka)t de helyette új utasítás sorozattal látjuk el. Így egy származtatott objektum tartalmazhatja (használhatja) ugyanazt a metódust, de nem használhatja ugyanazt az objektumra jellemzo változók azonosítóit (neveit).(Tehát: ugyanolyan nevu procedure-át, vagy function-t deklarálhatunk (azért, hogy felülírjuk a régit), de ugyanolyan nevu változót nem. (Mivel azt már az ős objektumban már deklaráltuk. De gondolkodjunk el rajta, hogy nem véletlenül nem lehet a változót újra deklarálni: hiszen az objektum minden körülmény között ugyanolyan feltételekmellett fog létezni. (Illetve, így is gondolkodhatunk -ez egy durvább megközelítés - : az utód súlyát sohasem fogjuk méter-ben mérni, hanem mindig kilógramm-ban.))

Zártság: (A példát folytatva) Mivel a hüllok utódaik a kétéltueknek minden tulajdoságukat öröklik így a kétéltuekre jellemzo szaporodást, egyedfejlodést is, de rejtve marad mivel helyette új szaporodás, ill. egyedfejlodés típus jött létre. Ugyanígy az objektumoknál, hiszen a Profilizmus megengedte ugyanazt a metódust, (itt: életfunkciót) 'kicserélni' egy új metódusra. !!!DE NE feledjük el, úgy ahogy a hülloknél a saját szaporodás, ill. egyedfejlodés az elsodleges, (azaz az új életfunkciónál a saját életfunkcióját 'hajtja' végre és véletlenül sem az örököltet) ugyanúgy az objektumoknál is, ahol szintén, a saját új metódusok az elsodlegesek. Ezután a zártság két szálon fut tovább amit nem érdemes biológiai példán keresztül modellezni. Az egyik szál - az öröklésre vonatkoztatva - a Statikus szál, a másik a Virtuális.(Ezt a két különbözô megoldási módot a metódusokra kell érteni, hiszen az adatok úgymond állandóak.)

Statikus metódusok: az ilyen metódusok az öröklésekor, csupán kicserélik az elôd metódusát az újra, nincs hatással az objektum más részeire - így nem változik meg

teljesen annak tulajdonsága - . Gondolok itt az objektum más részében elhelyezkedô esetleg ôt meghívó más metódusokra, akik nem az újat hanem a régit fogják meghívni, a statikus megoldás következménye képen.

Virtuális metódusok: Ilyen típusú metódusokkal lehet, megoldani az öröklés folyamaton keresztül a sokoldalúságot. Ez azt jelenti, hogy nem csak a régi metódust cseréli ki az újra, hanem az egész objektumot 'átnézve' a régi metódusra mutató összes hivatkozást átírja az új metódusra mutatóvá. Ezáltal megváltozik az egész objektum tulajdonsága, és az öröklés folyamatra nézve sokoldalúvá válik.

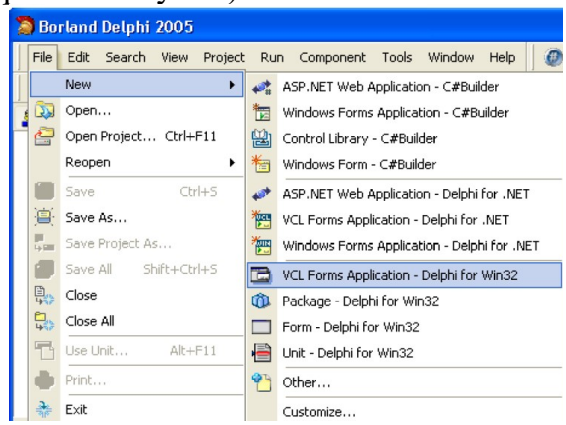
Borland Pascalban leírás szintjén annyi a különbség, hogy a metódus után odabiggyesztjük a virtual kulcsszót. (Figyelem, ha egy objektumban már használtuk a virtual kulcsszót akkor annak utódaiban is kötelezô.)

TValami = Object*{jellemzo változók}*
constructor Init;
procedure Valami(…); virtual;
function Valami(…): Type; virtual;*{egyéb metódusok}*
end;

Ha egy objektumban használunk virtuális metódusokat akkor használunk kell konstruktort. (Ha az objektumban csak statikus metódusok vannak) akkor nem kötelezô.) Ez egyenértéku a procedure-ával, használata azért kötelezô, mert ez hozza létre, a VirtuálisMetódusTáblát. Ha nem hozzuk létre a VMT-t akkor programunk megállhat - kiakadhat -, lefagyhat, de leginkább újraindítja számítógépünket.

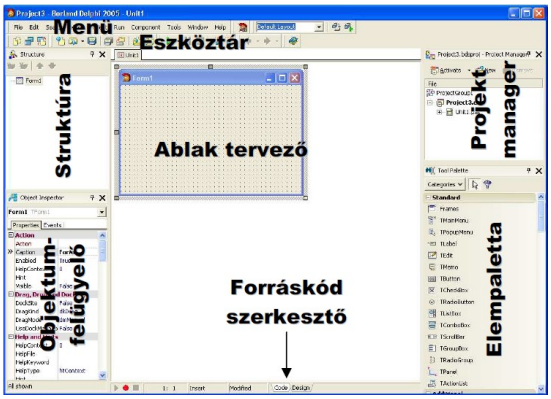
Az integrált fejlesztői környezet

A *Delhi* elindítása után új alkalmazás létrehozásához válasszukki a **File – New – VCL Form application - Delphi for Win32**menüpontot. (VCL = Visual Component Library = Vizuáliskomponenskönyvtár)



Itt láthatjuk, hogy *Delphi 2005*-ben nem csak *Delphi Win32*alkalmazást, de *C#*, illetve *.Net* alkalmazásokat is létrehozhatunk. Miebben a könyvben csak *Delphi Win32* alkalmazásokat fogunk létrehozni.

Miután létrehoztunk egy új alkalmazást, az alábbi ábrához hasonlót láthatunk. Nézzük meg részletesebben milyen részekből áll a*Delphi* fejlesztői környezete:



Menü: A különféle beállítások, programfuttatás, segítség, keresés, stb. Megvalósítását végezhetjük el itt.

Eszköztár: A menüből is meghívható funkciók gyors elérését teszik lehetővé. Ha az egérkurzort valamelyik ikonfölé visszük, akkor egy buborékban tájékoztatást kapunk az adott funkcióról.

Ablaktervező: A leendő programunk formáját tervezhetjük meg itt aránylag egyszerű módon. Megváltoztathatjuk az ablak (form) méretét, komponenseket (nyomógombokat, címkéket, képeket, stb.) helyezhetünk el rajta.

Elempaletta: Itt választhatjuk ki az alkalmazásunk építőelemeit (komponenseket), pl. nyomógombokat, bevitelimezőket, stb.

Objektumfelügyelő: Ez a *Delphi* egyik legfontosabb része. Segítségével beállíthatjuk az ablakunkon elhelyezett komponensek tulajdonságait (properties) és reakcióit az eseményekre (events).

TIPP: Az objektum felügyelőben a tulajdonságok és az események kategóriák szerint vannak sorbarendezve. Ezt át állíthatjuk, ha rákattintunk az egérjobbgombjával az objektum felügyelő tetszőleges mezőjére és kiválasztjuk az „Arrange – byName"menüpontot. Hasonlóan az „Arrange – by Category" segítségévelvisszaállíthatjuk a kategóriák szerinti elrendezést.

Forráskód szerkesztő: A *Delphi*nek az a része, ahova magát aforráskódot (programot) írjuk. Ezt az ablakot kezdetben nem látjuk, azablak alján levő „code" fül segítségével jeleníthetjük meg. Ha visszaszeretnénk menni az ablakunk (form) tervezéséhez, ugyanott klikkeljünka „design" fülre.

Struktúra: Itt láthatjuk az alkalmazásunk ablakán (form) levőkomponensek hierarchikus elrendezését.

Project manager: A *Delphi*ben mindig egy complex rendszerben (projektben) dolgozunk. Minden egyes alkalmazásunk egyprojektből áll. Egy projekt tetszőleges mennyiségű fájlt használhat. Ezeka fájlok lehetnek programfájlok (unit), a hozzájuk tartozó ablakok (form),az ablakon levő komponensek elrendezését tartalmazó fájlok,adatállományok, kép- és hangfájlok. Azt, hogy a projektünkhöz milyenfájlok kapcsolódnak és melyik fájl melyik fájlhoz tartozik, a projectmanager-ben láthatjuk. Kezdetben a projektünkhöz két fájl kötődik – egyprogramkódot tartalmazó fájl (.pas kiterjesztésű) és egy olyan fájl, amelyaz alkalmazásunk ablakán levő komponensek elrendezését, kezdetibeállításait tartalmazza (.dfm kiterjesztésű).

A projekt fájlfelépítése

Vizsgáljuk meg, hogyan néz ki a projektünk fájlfelépítése. Ha megnézzük a mappánkat, ahova a projektet mentettük, több állományt találhatunk benne. Elsősorban nézzük meg, melyik állományokat kell átmásolnunk, ha a forráskódot szeretnénk más számítógépre átvinni:

***.DPR** Delphi Project. Minden projektnek létezik egyetlen ilyen fő forrásállománya. Ez elsősorban létrehozza az alkalmazás ablakait, majd sikeres létrehozás után elindítja az alkalmazást.

***.BDSPROJ** Borland Development Studio Project fájl. Minden projekthez egyetlen ilyen állomány tartozik. A projekt különféle beállításait tartalmazza.

***.PAS** Unit forráskód. Ez tartalmazza az egyes modulok programkódját. Egy projektnek egy vagy több ilyen állománya lehet. Gyakorlatilag az alkalmazás minden egyes ablakához tartozik egy ilyen állomány, de ezeken kívül a projekt még tartalmazhat további ilyen állományokat (modulokat) is, melyekhez ablak nem tartozik.

***.DFM** Delphi Form. Formleírás. Azokhoz a modulhoz, melyekhez tartoznak ablakok, léteznek ilyen kiterjesztésű állományok is. Ezek az állományok az ablak és a rajta levő komponensek listáját és tulajdonságait tartalmazzák, tehát mindent, amit az ablak tervezőben és az objektum felügyelőben beállítottunk (a komponensek elrendezését, méreteit,feliratait, egyéb tulajdonságait és a komponensek egyes eseményeihez tartozó eljárások neveit is).

***.RES** Resource. Windows erőforrásfájl. Az alkalmazásunk ikonját tartalmazza.

A többi, mappában található állományt nem szükséges átmásolnunk, ezen állományok többségét a *Delphi* a fenti állományokból hozta létre automatikusan a projekt fordításakor. Ezek közül számunkra a legfontosabb az ***.EXE** kiterjesztésű állomány. Ha alkalmazásunkat más gépre szeretnénk átvinni és futtatni (a forráskód nélkül), elég ezt az állományt átmásolnunk és futtatnunk (az adott gépben nem szükséges hogy legyen telepítve *Delphi)*. Természetesen, ha a programunk forráskódját meg szeretnénk nézni, ill. szeretnénk benne valamit javítani, majd újra fordítani, nem elég ez az egyetlen állomány, szükséges hozzá az összes fent említett állomány is.

A forráskódok áttekintése

Részletesen áttekintjük, milyen programkódokat hozott létre a *Delphi*a program megírásakor.

Az ablak forráskódja (.pas)

Amikor megtervezzük az alkalmazásunk ablakát és elhelyezzük rajta az egyes komponenseket, a *Delphi* automatikusan kigenerál hozzá egy forráskódot. Nézzük meg most ennek a **unitl.pas** állománynak a szerkezetét:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;
type
TForm1 = class(TForm)
Button1: TButton;
Label1: TLabel;
procedure Button1Click(Sender: TObject);
private { Private declarations }
public { Public declarations }
end;
var
Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
      Application.Terminate;
end;
end.
```

A **unit Unit1;** a modulunk nevét adja meg. Ezt követően észrevehetjük, hogy a unit két részre van bontva. Az első része az **interface** kulcsszóval kezdődik (csatlakozási vagy publikus felület), a második az **implementation** (kivitelezési vagy implementációs rész).

Az **interface** részben fel vannak sorolva azok a típusok, változók, melyeket a unitban használunk, és amelyeket szeretnénk hogy más unitból, programból is elérhetők legyenek (miután ott megadjuk a **uses Unit1;** sort).

Az **implementation** részben egyrészt a feljebb felsorolt eljárások, függvények megvalósítását írjuk le, tehát azt, mit is tegyen az adott eljárás vagy függvény. Másrészt ide írhatjuk azokat a további változókat, eljárásokat, függvényeket is, melyeket csak a mi unit-unkon belül szeretnénk használni.

Nézzük meg részletesebben, mi van a programunk **interface** részében. A **uses** parancs után fel vannak sorolva azok a modulok, melyek szükségesek a mi modulunk futtatásához.

A **type** parancs után a **TForm1** típusú osztály definícióját látjuk. Ez valójában a mi főablakunk típusa. Láthatjuk, hogy **TForm** típusú osztályból lett létrehozva. Ezek után a **TForm1** osztály **private** (magán - csak az osztályon belül használható) és **public** (nyilvános - az osztályon kívülről is elérhető) változók, eljárások definíciója következhet.

A **var** kulcsszó után egyetlen változónk van deklarálva, ez a **Forml** objektum, ami valójában a mi alkalmazásunk főablaka.

Az **implementation** részben találunk egy **{$R *.dfm}** sort. A **$R** egy külső resource fájl beolvasását jelzi. A ***.dfm** most nem azt jelzi, hogy az összes **.dfm** végződésű állományt olvassa be, hanem itt a * csak a mi unitunk nevét helyettesíti, tehát csak a **unit1.dfm** állomány beolvasására kerül sor. Ez a fájl tartalmazza a főablakunk és a rajta található komponensek kezdeti beállításait.

Végül a **begin..end** közötti részben program kódot írhatunk be.

Megjegyzés: egy alkalmazáson belül több **.pas** végződésű állomány is lehet. Alkalmazásunk minden egyes ablakához tartozó forráskód egy ilyen külön modulban található. Ezen kívül az alkalmazásunk tartalmazhat még további ilyen modulokat is, melyekhez ablak (form) nem tartozik.

Alkalmazás projekt fájlja (.dpr)

Valójában ez az állomány nem mást, mint egy hagyományos Pascal fájl **.dpr** kiterjesztéssel:

```
program elso;
uses
Forms, Unit1 in 'Unit1.pas' {Form1};
{$R * .res}
begin
   Application.Initialize;
   Application.CreateForm(TForm1, Form1);
   Application.Run;
end.
```

Láthatjuk, hogy ez a program használja ez előbb elemzett **unitl.pas** modult - tehát azt a modult, amely az alkalmazásunk főablakát tartalmazza. Ha az alkalmazásunk több ablakot tartalmazna, itt lenne felsorolva az összes többi ablakhoz tartozó modul (unit) is.

A **{$R *.res}** sor most az **elso.res** állomány csatolását jelzi. Ez az állomány tartalmazza az alkalmazásunk ikonját.

A begin…end közötti főprogram inicializálja az alkalmazást, létrehozza a ablakot, majd elindítja az alkalmazást.

Komponensek tulajdonságai

Minden komponensnek vannak tulajdonságaik (melyek valójában az adott komponens osztályának attribútumai). A tulajdonságok nem csak a komponens külalakját határozzák meg, de a viselkedését is. Sok tulajdonság közös több

komponensnél is, de vannak olyan egyedi tulajdonságok is, melyek csak egy-egy komponensnél találhatók meg.

Az objektum felügyelőben a komponenseknek csak azokat a tulajdonságait találjuk meg, melyek hozzáférhetők a tervezés alatt. Ezen kívül léteznek még úgynevezett **run-time** tulajdonságok is, melyek csak az alkalmazás futása alatt érhetők el.

Továbbá megkülönböztetünk még **read-only** (csak olvasni lehet) és **write-only** (csak írni lehet) tulajdonságokat. Ezek a tulajdonságok általában csak a program futásakor érhetők el.

Most csak a közös tulajdonságokat soroljuk fel, amelyek minden komponensnél megtalálhatók, a többi „egyedi" tulajdonságot az egyes komponenseknél fogjuk külön tárgyalni.

Ha többet szeretnénk tudni valamelyik tulajdonságról, klikkeljünk rá az adott tulajdonságra az objektum felügyelőben, majd nyomjuk meg az **F1** funkcióbillentyűt. Ennek hatására megjelenik a kijelölt tulajdonságra vonatkozó súgó.

A komponens neve és felirata

Minden komponensnek a *Delphiben* van neve **(Name** tulajdonság). Ha a komponens nevét nem állítjuk be, a *Delphi* automatikusan beállít neki egy nevet, amely a komponens típusából (pl. Button) és egy sorszámból áll, pl. Button5. A komponens nevének egyedinek kell lennie a tulajdonosán belül. A komponens neve egy azonosító, amellyel az alkalmazásban a komponensre hivatkozni tudunk.

A névvel ellentétben a komponens felirata **(Caption** tulajdonság) bármilyen lehet, tartalmazhat szóközöket, és lehet ugyanolyan is, mint egy másik komponensé. A felirat például az ablak tetején jelenik meg a címsorban (Form komponensnél), vagy egyenesen rajta a komponensen (Button). Felirattal nem lehet ellátni olyan komponenseket, melyeknél ennek nincs értelme (pl. görgetősávnak nincs felirata).

A felirat segítségével lehet beállítani a komponens gyors elérését is a felhasználó számára. Ha a komponens feliratában valamelyik betű elé **&** jelet teszünk, akkor ez a betű a feliratban alá lesz húzva, és a felhasználó ezt a komponenst kiválaszthatja az **Alt + aláhúzott betű** billentyűkombináció segítségével. Ha a feliratban az & jelet szeretnénk megjeleníteni, a megadásánál meg kell azt dupláznunk (&&).

A komponens mérete és elhelyezkedése

A komponens elhelyezkedését a **Left** (bal szélétől) és **Top** (tetejétől) tulajdonságok adják meg. A tulajdonságok a koordinátákat nem az egész képernyőhöz viszonyítva tartalmazzák, hanem a tulajdonoshoz (szülőhöz) viszonyítva. Ha például egy nyomógombot helyezünk el közvetlenül az ablakunkon (formon), akkor a tulajdonosa az ablak (form) és ennek bal felső sarkához képest van megadva a nyomógomb elhelyezkedése (Left és Top tulajdonsága).

A komponens méretét a **Width** (szélesség) és **Height** (magasság) tulajdonsága határozza meg. Hasonlóan a Left és Top tulajdonságokhoz az értékük képpontokban (pixelekben) van megadva.

Néhány komponensnél beállíthatjuk, hogy a komponens mindig az ablak (form) valamelyik részéhez illeszkedjen (ragadjon). Ezt az **Align** tulajdonság segítségével tehetjük meg. Ennek megadásával a komponenst nem fogjuk tudni onnan leválasztani, az ablak átméretezésénél is ott marad az ablak teljes szélességében (ill. magasságában).

Az **Anchors** tulajdonság segítségével megadhatjuk, hogy a komponens a Form melyik széléhez (vagy széleihez) illeszkedjen.

Az utolsó mérettel és elhelyezkedéssel kapcsolatos érdekes tulajdonság a **Constrains.** Ennek a tulajdonságnak négy altulajdonsága van, melyek segítségével megadhatjuk a komponens lehetséges minimális és maximális méretét. Ha például beállítjuk ezt a tulajdonságot egy alkalmazás ablakánál, akkor az ablakot az alkalmazás futtatásakor nem lehet majd a megadott méretnél kisebbre, illetve nagyobbra méretezni.

A komponens engedélyezése és láthatósága

A komponens engedélyezését az **Enabled** tulajdonság segítségével tudjuk beállítani. Alapértelmezésben ez mindig igaz (true). Ha átállítjuk hamisra (false), tervezési módban nem történik látszólag semmi, de az alkalmazás futásakor a komponens „szürke" lesz és nem reagál majd a rákattintásra.

A másik hasonló tulajdonság a **Visible.** Segítségével beállíthatjuk, hogy a komponens látható legyen-e az alkalmazás futásakor. Az alapértelmezett értéke ennek a tulajdonságnak is igaz (true). Tervezési időben itt sem fogunk látni különbséget ha átállítjuk hamisra (false), csak az alkalmazás futtatásakor vehetjük majd észre hogy a komponens nem látható.

Megjegyzés: Ha a **Visible** tulajdonság értéke igaz egy komponensnél, az még nem jelenti feltétlenül azt, hogy a komponensünk látható a képernyőn. Ha ugyanis a komponens tulajdonosának (tehát amin a komponens van, pl. TPanel, TForm, stb.) a **Visible** tulajdonsága hamis, akkor sem a tulajdonos, sem a rajta levő komponensek nem láthatók. Ezért létezik a komponenseknek egy **Showing** tulajdonsága, amely egy run-time (csak futási időben elérhető) és read-only (csak olvasható) típusú tulajdonság. Ennek a tulajdonságnak az értéke megadja, hogy a komponensünk valóban látható-e a képernyőn.

A komponens „Tag" tulajdonsága

A **Tag** tulajdonság (lefordítva: hozzáfűzött cédula, jel) a komponensek egy különös tulajdonsága. Ennek a tulajdonságnak a beállítása semmilyen hatással nem hat ki a komponens működésére vagy külalakjára. Ez csupán egy kiegészítő memóriaterület, ahol különféle felhasználói adatok tárolhatók. Alapállapotban ebben a tulajdonságban egy LongInt típusú értéket tárolhatunk. Szükség esetén áttipizálással bármilyen más 4 bájt hosszúságú értéket írhatunk bele (pl. mutatót, karaktereket, stb.).

A komponens színe és betűtípusa

A komponensek **Color** (szín) és **Font** (betűtípus) tulajdonságaik segítségével beállíthatjuk a komponensek háttérszínét, ill. a komponenseken megjelenő feliratok betűtípusát (ha a komponensen megjeleníthető felirat).

A **Color** tulajdonság értékét megadhatjuk előre definiált konstansok segítségével: **clXXX** formában. Az XXX helyére vagy a szín nevét írhatjuk angolul (pl. **cIRed, cIGreen, cIBlue,** stb.), vagy a *Windows* által definiált, a rendszerelemekre használt színek neveit (pl. **cIBtnFace, clWindow,** stb.).

A **Font** tulajdonság értéke egy TFont típus lehet. A TFont osztály egyes elemeit beállíthatjuk az objektum felügyelőben, ha a **Font** mellett rákattintunk a „+" jelre.

A legtöbb komponens tartalmaz egy **ParentColor** (szülő színe) és egy **ParentFont** (szülő betűtípusa) tulajdonságot is. Ezekkel beállíthatjuk, hogy a komponens a tulajdonosának (ami leggyakrabban az alkalmazás ablaka – Form) a színét és betűtípusát használja. Így be tudjuk egyszerre állítani az ablakunkon levő összes komponens színét és betűtípusát a Form-unk **Font** és **Color** tulajdonságainak beállításával.

A komponens lebegő súgója

A komponens **Hint** (javaslat, buborék súgó) tulajdonságának köszönhetően az objektum felett egérrel elhaladva egy sárga téglalapban információt közölhetünk a felhasználóval (ha pl. megnyomja a gombot, akkor mi fog történni). A kiírandó segítséget a komponens **Hint** tulajdonságához kell hozzárendelnünk.

A komponens **ShowHint** (javaslatot megjelenít) tulajdonságával megadható, hogy ez a segítség megjelenjen-e a felhasználónak.

A **ParentShowHint** tulajdonsággal meghatározhatjuk, hogy a komponenshez a javaslat akkor jelenjen meg, ha a komponens tulajdonosának (ami általában a Form) a ShowHint tulajdonsága igaz. Így egyetlen tulajdonság átállításával (a Form ShowHint tulajdonságával) beállíthatjuk, hogy az ablak összes komponensére megjelenjen-e a buborék súgó vagy nem.

Az egérmutató beállítása

Sok komponens rendelkezik **Cursor** (egérmutató) tulajdonsággal. Ennek segítségével beállíthatjuk, hogy az egérmutatónak milyen alakja legyen, ha az adott komponens felett áll. Lehetséges értékek: **crHourGlass** (homokóra), **crCross** (kereszt), **crHelp** (nyíl kérdőjellel), **crUpArrow** (felfelé mutató nyíl), stb.

Tabulátor

Ha az alkalmazásunknak több komponense van, jó ha intelligensen működik a komponensek kiválasztása a TAB billentyű lenyomásakor. Azt, hogy a TAB billentyű megnyomásakor milyen sorrendben legyenek aktívak a komponensek a**TabOrder** (TAB sorrend) tulajdonság segítségével állíthatjuk be. Ide egy számot kell beírnunk, amely azt jelenti, hányadik lesz a komponens a sorrendben. A számozás 0-tól kezdődik.

A **TabStop** (TAB álljon meg) tulajdonság segítségével beállíthatjuk, hogy az adott komponensre el lehet-e jutni a tabulátor segítségével (ha a TabStop értéke igaz, akkor lehet, ha hamis, akkor nem lehet – a tabulátor nem áll meg a komponensen, hanem a sorban következőre ugrik át).

Események

A legtöbb komponensnél nem elég, ha csak a tulajdonságait állítjuk be. Sokszor szükségünk van rá, hogy az adott komponens valamilyen tevékenységet végezzen ha pl. rákattintunk egérrel, megnyomunk egy billentyűt, mozgatjuk felette az egeret, stb. Erre szolgálnak az események. Ahhoz, hogy egy eseményre a komponens úgy reagáljon, ahogy azt mi szeretnénk, meg kell írnunk az eseményhez tartozó eljárás programkódját.

Hasonlóan, ahogy a komponenseknek vannak olyan **tulajdonságaik,** amelyek szinte minden komponensnél megtalálhatók, vannak olyan **események** is, melyek majdnem minden komponensnél előfordulnak. Ezek közül a legfontosabbak a következők:

Komponensek eseményei:

Esemény	Mikor következik be	Megjegyzés
OnChange	Ha a komponens vagy annak tartalma megváltozik (pl. a szöveg az Edit komponensben).	Gyakran használatos az Edit és Memo komponenseknél. Összefügg a **Modified** tulajdonsággal (*run-time*, *read-only),* amely megadja, hogy a komponens tartalma megváltozott-e.
OnClick	A komponensre kattintáskor az egér bal gombjával.	Ez az egyik leggyakrabban használt esemény. Ez az esemény nem csak egérkattintáskor, hanem Enter, ill. Space billentyűk megnyomásakor is bekövetkezik, ha a komponens aktív (pl. egy aktív nyomógomb esetében).
OnDblClick	A komponensre duplakattintáskor az egér bal gombjával.	Duplakattintáskor az első klikkelésnél **OnClick** esemény következik be, majd ha rövid időn belül (ahogy a *Windowsban* be van állítva) érkezik második klikkelés is, akkor bekövetkezik az **OnDblClick** esemény.
OnEnter	Amikor a komponens aktiválva lett.	Itt nem az ablak (form) aktiválásáról van szó, amikor az egyik ablakból átmegyünk a másikba, hanem a komponens aktiválásáról, például ha Edit komponensbe kattintunk.
OnExit	Amikor a komponens deaktiválva lett.	Az előző esemény ellentettje. Például akkor következik be, ha befejeztük a

		bevitelt az Edit komponensbe és máshova kattintunk.
OnKeyDown	Amikor akomponens aktív és a felhasználó lenyom egy billentyűt.	Felhasználhatjuk az eljárás **Key** paraméterét, amely megadja a lenyomott billentyű virtuális kódját (virtual key codes). Továbbá a **Shift** paraméter (amely egy halmaz típusú) segítségével meghatározhatjuk, hogy le volt-e nyomva az Alt, Shift, vagy Ctrl billentyű (ssAlt, ssShift, ssCtrl). Megjegyzés: Ha azt szeretnénk, hogy a lenyomott billentyűt a Form kapja meg (méghozzá a komponens előtt), és ne az éppen aktív komponens, akkor a Form **KeyPreview** tulajdonságát át kell állítanunk igazra (true).
OnKeyPress	Amikor akomponens aktív és a felhasználó lenyom egy billentyűt.	A különbség ez előző eljárástól, hogy itt a **Key** paraméter char típusú, amely a lenyomott billentyűt ASCII jelét (betűt, számot, írásjelet) tartalmazza. Ez az esemény csak olyan billentyű lenyomásakor következik be, amelynek van ASCII kódja (tehát nem Shift, F1 és hasonlók).
OnKeyUp	Amikor akomponens aktív és a felhasználó felenged egy billentyűt.	A gomb felengedésénél jön létre, **Key** és **Shift** paramétere hasonló, mint az **OnKeyDown** eseménynél.
OnMouseDown	Amikor a felhasználó lenyomja valamelyik egérgombot.	Általában ez annak a komponensnek az eseménye, amely éppen az egérmutató alatt van.
OnMouseMove	Amikor a felhasználó megmozdítja az egeret a komponensen.	Hasonlóan az előzőhöz, annak a komponensnek az eseménye, amely éppen az egérmutató alatt van.
OnMouseUp	Amikor a felhasználó felengedi valamelyik egérgombot.	Ha több egérgomb van lenyomva, akkor mindegyik felengedésénél létrejön ez az eljárás.

Ablak (form) eseményei:

Esemény	Mikor következik be	Megjegyzés
OnActivate	Amikor az ablak aktívvá válik.	Akkor van generálva ez az eljárás, ha a felhasználó egy másik ablakból (vagy alkalmazásból) erre az ablakra klikkel.

OnDeactivate	Amikor az ablak inaktívvá válik.	Ha a felhasználó egy másik ablakra (vagy alkalmazásra) klikkel, tehát elhagyja a mi ablakunkat.
OnCloseQuery, OnClose	Ha az ablakot bezárjuk (Alt-F4, X a jobb felső sarokban, rendszermenü segítségével, programból a Close eljárással, stb.).	Az ablak bezárásakor először az **OnCloseQuery** esemény következik be, utána az **OnClose.** Az első esemény szolgálhat megerősítésre (pl. „Biztos hogy kilépsz?") vagy az adatok elmentésének figyelmeztetésére. Az alkalmazás bezárásának elkerülésére még az **OnClose** eseménynél is van lehetőségünk. Itt a paraméterben megadhatjuk azt is, hogy az ablakunk bezárás helyett csak elrejtve vagy minimalizálva legyen.
OnCreate, OnDestroy	Az ablak létrehozásakor ill. megszüntetésekor.	Az **OnCreate** esemény kezelésében lehetőségünk van dinamikusan létrehozni objektumok, melyeket ne felejtsünk el megszüntetni az **OnDestroy** eljárás kezelésében.
OnShow, OnHide	Az ablakmegmutatásakor, ill. elrejtésekor.	Ezek az eljárások szorosan összefüggenek az ablak **Visible** tulajdonságával.

Látható ablakok (melynek a visible tulajdonságuk igaz) létrehozásakor az események bekövetkezéseinek a sorrendje a következő: **OnCreate, OnShow, OnActivate, OnPaint.**

Hibakeresés

A *Delphi* tartalmaz sok eszközzel ellátott **integrált debugger**-t a hibák megkeresésére. A leggyakrabban használtakat.

A programunkban előforduló hibákat durván két csoportra oszthatjuk:

- olyan hibákra, melyeket a fordító kijelez (ide tartoznak a szintaktikai hibák – elírt parancsok, és a szemantikai hibák – parancsok logikailag rossz sorrendben használata),
- és olyan hibákra melyeket a fordító nem jelzi (logikai hibák).

Olyan a hibák esetén, melyeket a **fordító kijelez,** a program nem fut le, a kurzor pedig mindig a hibás sorban áll és megjelenik egy hibaüzenet. Ha rákattintunk a hibaüzenetre és megnyomjuk az F1 funkcióbillentyűt, elolvashatjuk a hiba részletes leírását.

Nehezebb azonban megtalálni az olyan hibákat, melyeket a **fordító nem jelez.** Az ilyen hibáknál a program elindul és mi abban a meggyőződésben élünk, hogy a

programunk hiba nélkül fut. Némely esetben azonban előfordulhat, hogy például a számítások eredményeként, nem a helyes eredményt kapjuk. Ilyenkor használhatjuk a hibakeresésre szolgáló eszközöket, melyeket a menüben a **Run** alatt találunk. Ezek közül a leggyakrabban használtak:

 Trace Into lépegetés

Ezzel az eszközzel lépegetni tudunk soronként az alkalmazásunkban. Egyszerűbben az **F7** funkcióbillentyűvel indíthatjuk el, illetve léphetünk tovább a következő sorra. Ha alprogram hívásához érünk, beleugrunk az alprogramba és ott is soronként lépegethetünk tovább.

 **Step Over** lépegetés

Hasonló az előző eszközhöz annyi különbséggel, hogy ha alprogram hívásához érünk, nem ugrunk bele az alprogramba, hanem azt egy blokként (egy lépésben) végzi el a program. Egyszerűbben az **F8** funkcióbillentyűvel érhetjük el.

 Run to Cursor

Ha ráállunk a kurzorral valamelyik sorra a forráskódban és ezzel (vagy egyszerűbben az **F4** funkcióbillentyűvel) indítjuk el, a program hagyományos módon elindul és fut mindaddig, amíg ahhoz a sorhoz nem ér, ahol a kurzorunk áll. Itt leáll a program, és innen lépegethetünk tovább a fent említett eszközökkel.

 **Breakpoints** (Add Breakpoint – Source Breakpoint...)

A breakpoint (megszakítás pontja) segítségével a *Delphinek* megadhatjuk, hogy a programunk melyik során álljon meg.

Gyakorlatban: ráállunk valamelyik sorra a forráskódban, kiválasztjuk a menüből a **Run – Add Breakpoint – SourceBreakpoint...** menüpontot, majd „Ok" (vagy rákattintunk a sor elején a kék körre - ). Ekkor a kijelölt sor háttere átszíneződik, és a sor előtt egy piros kör jelenik meg ( i. Ez jelenti azt, hogy a program ebben a sorban le fog állni. A programot utána elindítjuk a **Run - Run** (vagy F9) segítségével. Ha a program a futása során breakpointhoz ér, leáll. Innen lépegethetünk tovább egyesével az első két eszköz segítségével (F7, F8), vagy futtathatjuk tovább a programot a **Run – Run** (vagy F9) segítségével. Egy programban több breakpointot is elhelyezhetünk.

A breakpointot a forráskódban a sor elején található piros körre () kattintva szüntethetjük meg.

 **Watch** (Add Watch...)

A program lépegetése közben ennek az eszköznek a segítségével megfigyelhetjük az egyes változók értékét.

A változók értékeit a **Watch List** ablakban követhetjük nyomon (ez az ablak automatikusan megjelenik a program indításakor, de ha mégsem jelenne meg a View – Debug Windows – Watches menüvel hívhatjuk elő).

Új változót vagy kifejezést a **Run - Add Watch. (CTRL+F5)** menüpont segítségével adhatunk a megfigyelt változók közé (a Watch List-be).

Gyakorlatban ezt úgy használhatjuk, hogy kijelölünk a programban Breakpointot,

ahonnan a változókat figyelni szeretnénk, vagy odaállunk a kurzorral és elindítjuk a programot a „Run to Cursor" segítségével. Majd az „Add Watch." (vagy CTRL+F5) segítségével beállítjuk a figyelni kívánt változókat és elkezdünk lépegetni a „Trace Into" ill. a „Step Over" segítségével. Közben figyelhetjük a kiválasztott változók értékeit.

 Evalute / Modify

Ennek az eszköznek a segítségével nem csak megfigyelhetjük, de **meg is változtathatjuk** a kifejezések, változók vagy tulajdonságok értékeit. Ez egy nagyon hasznos eszköz, ha arra vagyunk kíváncsiak, hogyan viselkedne a program, ha például az „i" változóban nem 7, hanem 1500 lenne. Ezt az eszközt egyszerűbben a **CTRL+F7** funkcióbillentyűvel hívhatjuk elő.

 Program Reset

Előfordulhat, hogy a programunk lefagy, vagy csak egyszerűen olyan helyzetbe kerülünk, hogy a programunk futását le szeretnénk állítani, majd elölről futatni. Ebben az esetben lehet segítségünkre a **Run - Program Reset** menüpont (vagy **CTRL+F2).**

Forráskód külalakja

Az alábbi javaslatok betartásával olvasható és áttekinthető forráskódot tudunk majd írni:

- **Nagy és kisbetűk**– a Delphi (Pascal) nem case-sensitive programozási nyelv. Ennek ellenére jó, ha a nagy és kisbetűk használatában rendet tartunk és követünk valamilyen logikát. Például a „btnKilepesClick" sokkal áttekinthetőbb, mint a „btnkilepesclick" vagy a „BTNKILEPESCLICK").
- **Megjegyzések**– hasznos megjegyzések gyakori használatával az alkalmazásunkban sok időt és problémát spórolhatunk meg magunknak a jövőben. Megjegyzést a forráskódba tehetünk {kapcsos zárójelek} között vagy két törtvonal // segítségével az egysoros megjegyzés elején.
- **Bekezdések, üres sorok, szóközök**– ne spóroljunk az üres sorokkal és a bekezdésekkel (beljebb írásokkal), szóközökkel a programunkban. Ezek megfelelő használatával programunk sokkal áttekinthetőbb lesz.

Információk bevitele

Az információkat a programunkba komponensek segítségével vihetjük be. Ebben a fejezetben veszünk néhány példát ezek használatára és megismerkedünk az egyes komponensek további (egyéni) tulajdonságaival, eseményeivel és metódusaival.

A következő komponensek nem csak információk bevitele de adatok megjelenítésére egyaránt alkalmasak.

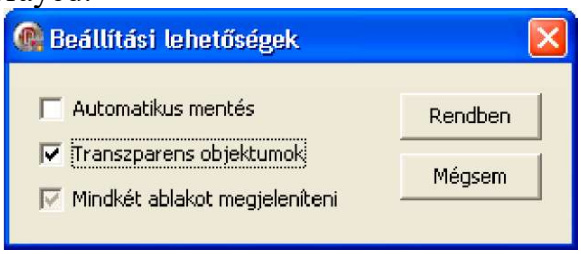
Jelölőnégyzet használata - CheckBox

Előnye, hogy állandóan látható ki van-e jelölve, egyetlen kattintással kijelölhető, áttekinthető és a felhasználónak nem ugrálnak elő állandóan ablakok (mint ahogy az

történne üzenetablakok használatánál).

A **CheckBox** fontosabb tulajdonságai:

- **AllowGrayed**– ha ennek a tulajdonságnak az értéke igaz (true), akkor a jelölőnégyzetnek három lehetséges értéke lehet: Checked (kipipálva), Unchecked (nincs kipipálva), Grayed (szürke). Egyébként csak két értéke lehet.
- **Caption**– felirat, amely a jelölőnégyzet mellett szerepeljen.
- **Checked**– megadja hogy ki van-e pipálva a jelölőnégyzet (true) vagy nincs kipipálva (false).
- **State**– hasonló az előzőhöz, de ennek értéke háromféle lehet: cbChecked, cbUnchecked, cbGrayed.



Nagyon egyszerűen olvashatjuk ki a jelölőnégyzetek értékeit. Például meg szeretnénk tudni, hogy a felhasználó kijelölte-e az első jelölőnégyzetet (és ha igen, akkor el szeretnénk végezni valamilyen műveletet). Ezt a következőképpen tehetjük meg:

if CheckBoxl.Checked = true then ...

Vagy használhatjuk ugyanezt „rövidített" változatban:

if CheckBoxl.Checked then ...

Az adatok felhasználótól való beolvasásán kívül természetesen nagyon jól használható a CheckBox logikai értékek megjelenítésére is.

Választógomb - RádioButton

Ez a komponens általában csoportokban fordul elő, mivel itt a csoporton belül mindig csak egyet lehet kiválasztani. Az alkalmazás indításakor megoldható, hogy a csoporton belül egy gomb se legyen kiválasztva, de ez csak ritkán szokott előfordulni. Ha már ki van választva egy, kiválaszthatunk egy másikat, de nem szüntethetjük meg a kiválasztást, tehát egyet mindenképpen ki kell választanunk. Az egyik legfontosabb tulajdonsága:

Checked – értéke (true/false) megadja, hogy a gomb ki van-e választva.

A RadioButton komponensek szinte mindig valamilyen logikai csoportot összekapcsoló komponensen vannak rajta (GroupBox, Panel komponenseken), de lehet közvetlenül a Form-on (ablakunkon) is. Több RadioButton komponens használata helyett azonban jobb használni inkább egy RadioGroup komponens.

Választógomb csoport - RadioGroup

A RadioGroup komponens legfontosabb tulajdonságai:

Items – értéke TStrings típus lehet. Ennek segítségével adhatjuk meg, milyen választógombok szerepeljenek a komponensünkön (tehát miből lehessen választani). Az egyes lehetőségek neveit külön-külön sorban adjuk meg. A „karikákat" a RadioGroup komponens kirakja automatikusan mindegyik sor elé.

Columns – segítségével megadhatjuk, hogy a választási választógombok mennyi oszlopban legyenek megjelenítve.

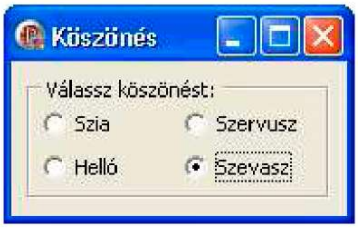
ItemIndex – ennek a tulajdonságnak az értéke tartalmazza, hogy melyik választógomb van kiválasztva. Ha értéke -1 akkor egyik sincs kiválasztva, ha 0 akkor az első, ha 1 akkor a második, stb. választógomb van kijelölve (tehát a számozás 0-tól kezdődik).

Azt, hogy melyik nyomógomb van kiválasztva, az **ItemIndex** tulajdonság tesztelésével vizsgálhatjuk:

if RadioGroupl.ItemIndex = 0 then...

Ez a tesztelés azonban nagyon sok programozónak nem a legmegfelelőbb, mivel meg kell jegyezni, melyik lehetőséghez melyik szám tartozik. Ezen könnyíthetünk, ha például a számok helyett konstansokat definiálunk (const SZIA = 0; HELLO = 1; ...) és ezek segítségével teszteljük a feltételt:

if RadioGroupl.ItemIndex = SZIA then...



Beolvasás „üzenetablak" segítségével

Egysoros szöveg beolvasásához használhatunk egy újabb „üzenetablakot", az **InputBox**-ot: function InputBox(const ACaption, APrompt,ADefault: string): string;
Az egyes paraméterek leírása:

- **ACaption:** a dialógusablak felirata,
- **APrompt:** a dialógusablakban megjelenő szöveg,
- **ADefault:** a beviteli mezőben megjelenő kezdeti szöveg. Például:
  nev := InputBox('Név megadása','Kérlek add meg a neved:', '');

Természetesen ennél az eszköznél sokkal jobban felhasználható egysoros szöveg bevitelére az Edit komponens.

Egysorosszövegbevitelidoboz - Edit

Az Edit komponensnek sok specifikus tulajdonsága van, melyek segítségével az egysoros bevitelt korlátozhatjuk, vagy formátozhatjuk (például megadhatjuk egyetlen tulajdonság beállításával, hogy a bevitt szöveg helyett csillagok vagy más jelek jelenjenek meg - így használhatjuk jelszó bevitelére is).

Az Edit komponens legfontosabb tulajdonságai:

Text – ez a tulajdonság tartalmazza a beviteli mezőben megjelenő szöveget. Segítségével kiolvashatjuk vagy beállíthatjuk az Edit komponensben levő szöveget.

MaxLength – az Edit-be megadható szöveg maximális hossza. Segítségével beállíthatjuk milyen hosszú szöveget adhat meg a felhasználó.

Modified – annak megállapítására szolgáló tulajdonság, hogy a beviteli mezőben történt-e változás.

AutoSelect – segítségével beállíthatjuk, hogy a beviteli mezőbe lépéskor (kattintáskor) ki legyen-e jelölve az egész szöveg. Ezt az szerint adjuk meg, hogy a felhasználó a mező értékét előreláthatóan új értékkel fogja helyettesíteni vagy csak az ott levő értéket fogja módosítani.

ReadOnly – meghatározza, hogy a felhasználó megváltoztathatja-e a beviteli mezőben levő értéket.

PasswordChar – ennek a tulajdonságnak az értéke egy karakter lehet, amely meg fog jelenni a bevitelnél a bevitt karakterek helyett. Jelszó bevitelénél használhatjuk.

Az Edit komponens több metódussal is rendelkezik, melyekközül az egyik:

CopyToClipboard – vágólapra másolja a beviteli mezőben levő szöveget.

Többsoros szöveg beviteli doboz - Memo

A Memo komponens az Edit-hez hasonló, de ebbe többsoros szöveget olvashatunk be.A Memo legfontosabb tulajdonságai:

Alignment – a sorok igazításának beállítására használható tulajdonság. A sorokat igazíthatjuk balra, jobbra vagy középre.

ScrollBars – tulajdonsággal megadhatjuk, hogy a komponensen a vízszintes, a függőleges vagy mindkettő görgetősáv megjelenjen-e, esetleg ne legyen egyik görgetősáv se a komponensen.

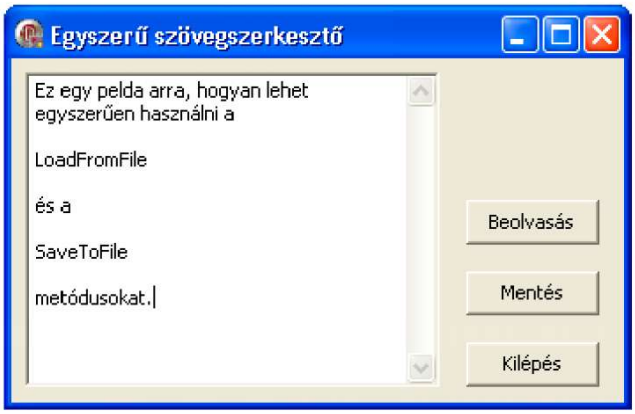
WordWrap – igaz (true) értéke az automatikus sortördelést jelenti (ha ez a tulajdonság igaz, nem lehet „bekapcsolva" a vízszintes görgetősáv, mivel ez a kettő tulajdonság kölcsönösen kizárja egymást).

WantTabs, WantEnter – a Tab és Enter billentyűknek minden komponenseknél más funkciójuk van. A tabulátor megnyomására általában a következő komponens lesz aktív a Form-on. Ha szeretnénk, hogy a felhasználó a Memo komponensben használni tudja a Tab és Enter billentyűket, tehát hogy tudjon tabulátorokat (bekezdéseket) és új sorokat létrehozni a szövegben, ezeknek a tulajdonságoknak az értékeit igazra (true) kell állítanunk. Ha ezt nem tesszük meg, a felhasználó akkor is tud létrehozni bekezdéseket és új sorokat a Ctrl+Tab és Ctrl+Enter billentyűkombinációk segítségével.

Text – hasonló tulajdonság, mint az Edit komponensnél. A Memo komponensben levő szöveget tartalmazza.

Lines – segítségével a Memo-ba beírt szöveg egyes soraival tudunk dolgozni. Ez a tulajdonság TString típusú,melynek sok hasznos altulajdonsága és metódusa van. Például a Lines (TString) **LoadFromFile** és **SaveToFile** metódusaival be tudunk olvasni a merevlemezről ill. el tudunk menteni a merevlemezre szöveget.A Lines tulajdonság használatát mutatja be a következő példa is:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
        Memo1.Lines.LoadFromFile('c:\autoexec.bat');
                ShowMessage('A 6. sor: ' + Memo1.Lines[5]);
end;
```



Görgetősáv – ScrollBar

Gyakori probléma a számok bevitele a programba. Számok bevitelére használhatjuk a már említett Edit vagy Memo komponenseket is. Most azt mutatjuk be, hogyan vihetünk be számokat a **ScrollBar** komponens segítségével.

A ScrollBar legfontosabb tulajdonságai:

Kind – meghatározza, hogy a komponens vízszintesen (sbHorizontal) vagy függőlegesen (sbVertical) helyezkedjen-e el. Az újonnan létrehozott ScrollBar kezdeti beállítása mindig vízszintes.

Min, Max – meghatározza a határértékeket.

Position – meghatározza a csúszka aktuális pozícióját (aktuális értéket).

SmallChange – az eltolás mértéke, ha a görgetősáv szélein levő nyilakra kattintunk, vagy az értéket a billentyűzeten levő nyilak segítségével állítjuk be.

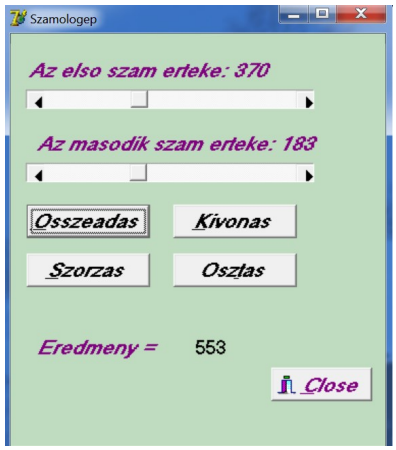
LargeChange – az eltolás mértéke, ha a görgetősáv sávjában kattintunk valahova, vagy a PageUp, PageDown billentyűket nyomjuk meg.

Ha meg szeretnénk határozni azt az értéket, amelyet a felhasználó beállított a görgetősávon, azt legegyszerűbben az OnChange eseményben vizsgálhatjuk.

Ha igazán szép görgetősávot akarunk létrehozni, akkor a görgetősáv mellé (vagy elé) tegyünk egy címkét (Label) is, amely folyamatosan a csúszka aktuális pozíciójának értékét mutatja.

Példaprogram

Készítsünk csúszkás számológépet! A kért számot egy-egy vízszintes görgetősáv tologatásával lehessen bevinni, majd a megfelelő nyomógombra (feliratuk: *Összeadás, Kivonás, Szorzás, Osztás)* való kattintáskor jelenjen meg egy címkében az eredmény.



A Form-ra helyezzünk el két ScrollBar-t, még négy nyomógombot is (*Összeadás, Kivonás, Szorzás, Osztás*).

Most az egyes nyomógombok OnClick eseményeit fogjukkezelni:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(ScrollBar1.position + ScrollBar2.position);
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
   Label1.Caption := IntToStr(ScrollBar1.position - ScrollBar2.position);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
   Label1.Caption := IntToStr(ScrollBar1.position * ScrollBar2.position);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
Label1.Caption := FloatToStr((ScrollBar1.Position)/(ScrollBar2.Position));
end;

procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
Label2.Caption := 'Az elso szam erteke: ' + IntToStr(ScrollBar1.Position);
end;

procedure TForm1.ScrollBar2Change(Sender: TObject);
begin
      Label3.Caption := 'Az masodik szam erteke: ' + IntToStr(ScrollBar2.Position);
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
      Form1.Close;
end;

end.
```

Ebben a programkódban használtuk az **IntToStr**ill. **StrToInt** függvényeket. Ezek egész számot alakítanak át szöveggé ill. fordítva az (**FloatToStr** ill. **StrToFloat** hasonlo cask valos számt alakítanak át szöveggé és vissza). Szintaxisuk:

```
function IntToStr(Value: integer): string;
function StrToInt(const S: string): integer;
```

Szám bevitele - SpinEdit segítségével

A SpinEdit komponens szintén számok bevitelére szolgál. A számot megadhatjuk billentyűzet segítségével és egér segítségével is a komponens szélén levő fel-le nyilakra kattintva.

Legfontosabb tulajdonságai:

Value – meghatározza a beadott (kiválasztott) értéket.

MinValue – meghatározza a minimum értéket, amit a felhasználó megadhat a SpinEdit-be.

MaxValue – segítségével megadhatjuk a maximum értéket, amit a felhasználó megadhat a SpinEditbe.

Increment – megadja, hogy a jobb szélén levő nyilakra kattintva mennyivel növekedjen ill. csökkenjen a SpinEdit aktuális értéke.

Listadoboz - ListBox

A klasszikus listadoboz (ListBox) az egyik leggyakrabban hasznát kimeneti komponens.

Legfontosabb tulajdonságai:

Columns – oszlopok száma, melyekben az adatok meg lesznek jelenítve.

Items – a legfontosabb tulajdonság, a lista egyes elemeit tartalmazza. Ez is TString típusú, hasonlóan a Memo komponens Lines tulajdonságához, és mint olyannak, rengeteg hasznos metódusa van.

ItemIndex – az éppen kiválasztott elem sorszáma. A számozás 0-tól kezdődik. Ha nincs kiválasztva egyik eleme sem a listának, akkor az ItemIndex értéke -1.

MultiSelect – egyszerre több érték (elem) kiválasztását engedélyezi (true) ill. tiltja (false). Több elem kiválasztásánál azt, hogy melyik elemek vannak kiválasztva a ListBox **Selected** tulajdonságával vizsgálhatjuk, amely egy 0 indextől kezdődő tömb (pl. a Selected[0] igaz, ha az első elem van kiválasztva, a Selected[1] igaz, ha a második elem van kiválasztva, stb.).

SelCount – kiválasztott elemek darabszámát tartalmazza (ha a MultiSelect értéke igaz).

Sorted – megadja, hogy a lista elemei legyenek-e rendezve ábécé sorrendben. Ha értéke igaz (true), új elem hozzáadásánál a listához automatikusan rendezve kerül a listadobozba.

Az **Items** tulajdonságnak sok hasznos metódusa van. Ezek közül a leggyakrabban használt metódusok:

- **Add** – a lista végére új elemet rak be.
- **Clear** – a ListBox összes elemét törli.
- **Delete** – kitöröl egy kiválasztott elemet a listában.
- **Equals** – teszteli, hogy két lista tartalma egyenlő-e. False értéket ad vissza, ha a két lista különbözik a hosszában (elemek számában), más elemeket tartalmaznak, vagy ha más sorrendben tartalmazzák az elemeket.
- **Insert** – új elemet szúr be a listába a megadott helyre.
- **LoadFromFile** – beolvassa a lista elemeit egy szöveges állományból. A sikertelen beolvasást a kivételek segítségével kezelhetjük, melyekről később lesz szó.
- **Move** – egy adott elemet a listában egy másik (új) helyre helyez át.
- **SaveToFile** – elmenti a lista elemeit egy szöveges állományba. A lista minden eleme egy új sorban lesz a fájlban. A sikertelen mentést a kivételek segítségével kezelhetjük, melyről bővebben későbbi fejezetekben lesz szó.

Példaprogram

A Form-ra helyezzünk el egy ListBox-ot, melynek Items tulajdonságába adjunk meg néhány nevet. Rakjunk a Form-ra még pár nyomógombot is (Rendezd, Adj hozzá, Töröld, Töröd mind, Olvasd be, Mentsd el, Kilépés).

Az egyes nyomógombok OnClick eseményeit fogjukkezelni:

Adj hozzá nyomógomb - egy InputBox segítségével beolvasunk egy nevet, melyet a lista végéhez adunk:

```
procedure TForm1.Button2Click(Sender: TObject);
var s:string;
begin
   s := InputBox('Adj hozzá','Kérlek add meg a nevet:', '');
   if s<>'' thenListBox1.Items.Add(s);
end;
```

Rendezd nyomógomb:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   ListBoxl.Sorted := true;
end;
```

Töröld nyomógomb - törli a lista kijelölt elemét:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
   ListBox1.Items.Delete(ListBox1.ItemIndex);
end;
```

Töröld mind nyomógomb - törli a lista összes elemét:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
   ListBox1.Clear;
end;
```

Megjegyzés: A lista törlését itt a ListBoxI.Clear metódussal végeztük el. Ez ugyanúgy kitörli a lista elemét, mint a ListBoxI.Items.Clear metódus, de az elemek

törlésen kívül további „tisztító" műveleteket is elvégez. Gyakorlatilag a két metódus közti különbség a ComboBox komponensnél látható: a ComboBox.Clearkitörli a teljes listát, a ComboBox.Items.Clear kitörli szintén a listát, de az utolsó kiválasztott érték a beviteli mezőben marad!

Mentsd el nyomógomb:

```
procedure TForm1.Button6Click(Sender: TObject);
begin
  ListBoxl.Items.SaveToFile('nevsor.txt');
end;
```

Olvasd be nyomógomb:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  ListBoxl.Items.LoadFromFile('nevsor.txt');
end;
```

Láthatjuk, hogy az utóbbi két metódus saját maga megnyitja a fájlt, beolvassa / menti az adatokat, majd bezárja a fájlt.

Kilépés nyomógomb:

```
procedure TForm1.Button7Click(Sender: TObject);
begin
  Forml.Close;
end;
```

Ezekből a példákból is jól látható, hogy a ListBox komponens felhasználására nagyon sok lehetőség van. Azonban a ListBox komponens használatának is lehet hátránya: az egyik hátránya, hogy az alkalmazás ablakán állandóan ott van és sok helyet foglal el. Másik hátránya, hogy ha a ListBox-ot bemeneti komponensként használjuk, a felhasználó csak a listában szereplő értékek közül választhat. Természetesen van amikor nekünk ez így megfelel, de előfordulhat, hogy a felhasználónak több szabadságot szeretnénk adni a választásnál (például saját érték beírására). Ezekre adhat megoldást a ComboBox komponens.

Kombinált lista - ComboBox

Ennek a komponensnek a formája a képernyőn nagyon hasonlít az Edit komponenséhez, ugyanis a felhasználó sok esetben írhat bele saját szöveget is. Hasonlít azonban a ListBox komponenshez is, mivel a jobb szélén levő nyílra kattintva (vagy Alt + lefelé nyíl, vagy Alt + felfelé nyíl) megjelenik (legördül) egy lista, amelyből a felhasználó választhat.

Mivel a ComboBox tulajdonságai, metódusai és használata sok mindenben megegyezik (vagy nagyon hasonlít) a ListBox-al, ezért nem vesszük át mind még egyszer, helyette inkább kiegészítjük őket továbbiakkal:

Style – ez a tulajdonság nem csak a ComboBox külalakját adja meg, de komponens viselkedését és a felhasználói bemenetek lehetőségét is. Értéke lehet:

- **csDropDown:** tipikus ComboBox, amely megjeleníti a listát, de közvetlen szöveg bevitelt is lehetővé tesz.

- **csDropDownList:** szövegbevitelt nem tesz lehetővé. Valamelyik betű (billentyű) megnyomásakor az első olyan elemre ugrik a listában, amely ezzel a betűvel kezdődik.
- **csSimple:** a közvetlen szövegbevitelt is lehetővé teszi, a lista közvetlenül a beviteli mező alatt van megjelenítve (állandóan). A megjelenített lista méretét a komponens Height tulajdonsága határozza meg.
- **csOwnerDrawFixed:** kép megjelenítését teszi lehetővé a listában. A lista összes elemének a magassága azonos, melyet az ItemHeight tulajdonság határoz meg.
- **csOwnerDrawVariable:** hasonló az előzőhöz, de az egyes elemeknek a listában különböző magasságuk (méretük) lehet.

StringGrid komponens

Ez nem olyan gyakran használt komponens, mint a lista. Segítségével szöveges adatokat jeleníthetünk meg táblázatban.

Helyezzünk el a Form-on egy StringGrid komponenst és két nyomógombot. Az első nyomógomb OnClick eseményébe írjuk be az alábbi programrészt:

```
procedure TForm1.Button1Click(Sender: TObject);
var i,j,k:integer;
begin
    k := 0;
      with StringGrid1 do
            for i:=1 to ColCount-1 do for j:=1 to RowCount-1 do
     begin
       k := k + 1;
            Cells [i,j] := IntToStr(k) ;
     end;
```

A forráskódban StringGrid komponens több tulajdonságával is megismerkedhettünk:

ColCount – oszlopok számát határozza meg (fix oszlopokkal együtt).
RowCount – hasonló az előzőhöz, csak ez a sorok számát határozza meg.
Cells – az egész táblázat mátrixa.

A StringGrid komponens további tulajdonságai, melyek a példában nem szerepelnek:

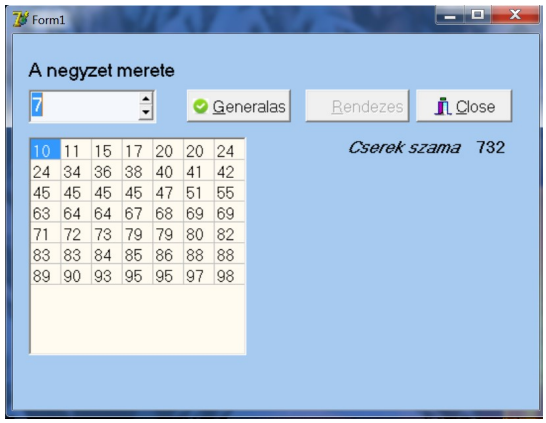
FixedCols – a rögzített (fix) oszlopok száma.
FixedRows – a rögzített (fix) sorok száma.
FixedColor – a rögzített oszlopok és sorok háttérszíne.
GridLineWidth – az egyes cellák közti vonal vastagsága.

Végül még nézzük meg a StringGrid komponens néhány érdekes metódusát:

- **MouseToCell:** az X, Y koordinátákhoz meghatározza a táblázat sorát és oszlopát.
- **CellRect:** a megadott cella képernyő-koordinátáit adja meg pixelekben.

Példaprogram

A következő feladat szemlélteti a StringGrid komponens használatát: Készítsünk egy alkalmazást, melyben egy SpinEdit komponens segítségével beállíthatjuk a StringGrid komponens méretét 3x3-tól 10x10-ig. A programunk továbbá tartalmazzon két nyomógombot. Az első nyomógomb generáljon véletlen számokat a StringGrid-be, a második nyomógomb pedig rendezze ezeket a számokat növekvő sorrendbe.



Miután a szükséges komponenseket elhelyeztük a Form-on, állítsuk be a komponensek alábbi tulajdonságait az Objektum Inspector- ban (vagy írjuk be a a Form OnCreate eseményébe):

```
Labell .Caption := 'A négyzet mérete';
StringGridl .DefaultColWidth := 30;
StringGridl.DefaultRowHeight := 30;
StringGridl .FixedCols := 0;
StringGridl.FixedRows := 0;
StringGridl.ScrollBars := ssNone;
SpinEditl.EditorEnabled := False;
SpinEditl .MaxValue := 10;
SpinEditl .MinValue := 3;
SpinEditl .Value := 5;
```

```
      Buttonl.Caption := 'Generálás';
      Button2.Caption := 'Rendezés';
      Button2.Enabled := False;
```
Majd írjuk meg az egyes komponensek eseményeihez tartozó programrészeket:
```
procedure TForml.SpinEditlChange(Sender: TObject);
       var i,j:integer;
begin
                                   // toroljuk a cellak tartalmat
 for i:=0 to 9 do
    for j:=0 to 9 do StringGridl.Cells[i,j]:='';
                              // a rendezes gombot nem elerhetove tesszuk
    Button2.Enabled := false;
                                 // beallitjuk a sorok es oszlopok szamat
    StringGrid1.ColCount := SpinEdit1.Value;
    StringGrid1.RowCount := SpinEdit1.Value;
    { beallitjuk a StringGrid szelesseget es magassagat minden cella 31 szeles(magas)
              a vonalai egyutt  az egyik szelen + 3 a StringGrig keretenek
                                   szelessege(magassaga)}
 StringGrid1.Width := 31 * SpinEdit1.Value + 3;
 StringGrid1.Height := 31 * SpinEdit1.Value + 3;
 end;
procedure TForml.ButtonlClick(Sender: TObject);
var
   i,j:integer;
begin
    {a cellakba 10-99 kozotti veletlen szamokat generalunk az oszlopok(sorok) 0-tol
                                   vannak szamozva !!}
 for i:=0 to StringGrid1.ColCount-1 do for j:=0 to StringGrid1.RowCount-1 do
 StringGrid1.Cells[i,j] :=IntToStr(random(90)+10);
                              // a rendezes gombot elerhetove tesszuk
 Button2.Enabled := true;
end;
procedure TForm1.Button2Click(Sender: TObject);
var i,j,ei,ej:integer;
s:string;
csere:boolean;
begin
  {a StringGrid1-el dolgozunk. Az alabbi sor kiadasaval nem kell mindig megadnunk
   hogy pl. StringGrid1.Cells[i,j], helyette eleg a Cells[i,j] a with parancson belul.}
 with StringGrid1 do repeat
  ei := 0;
     {elozo cella sorindexe ej := 0; elozo cella oszlopindexe csere := false;azt jelzi,
                                   volt-e csere(false=nem volt) }
```

```
  for j:=0 to RowCount-1 do for i:=0 to ColCount-1 do begin
                        // osszehasonlitjuk az aktualis cellat az elozovel
        if StrToInt(Cells[i,j])<StrToInt(Cells [ei,ej] ) then begin
        s := Cells [i,j] ;
        Cells [i,j] := Cells[ei,ej] ;
        Cells[ei,ej] := s;
        csere := true;
        end;
                            // beallitjuk az elozo cellat az aktualis cellars
        ei := i;
        ej := j;
        end;
until not csere;
{addig megyunk vegig az egesz StringGrid-en, amig igaz nem lesz, hogy csere=false;
  a rendezes gombot nem elerhetove tesszuk, mivel mar rendezve vannak a szamok}
Button2.Enabled := false;
end;
procedure TForml.FormCreate(Sender: TObject);
begin
                // a program inditasakor beallitjuk a veletlenszam generatort
randomize;
end;
```

A StringGrid komponens nem csak adatok megjelenítésére, de adatok bevitelére is használható. Ahhoz, hogy a program futása közben a felhasználó tudjon beírni közvetlenül is adatokat a komponensbe, át kell állítanunk a **StringGrid.Options** tulajdonságának **goEditing** altulajdonságát true értékre.

Időzítő - Timer

Gyakran szükségünk lehet bizonyos időközönként (intervallumonként) megszakítani a program normális futását, elvégezni valamilyen rövid műveletet, majd visszatérni a program normális futásához – ezt a Timer komponens segítségével tehetjük meg. Időzítővel tudjuk megoldani például mozgó szöveg (folyamatosan körbe futó szöveg) kiírását is.

A Timer komponens nem sok tulajdonsággal rendelkezik, pontosabban egy specifikus tulajdonsága van:

Interval – meghatározza azt az időintervallumot (milliszekundumokban), amely eltelte után újra és újra bekövetkezik az OnTimer esemény.

Az **OnTimer** eseménybe írhatjuk azt a kódot, amelyet periodikusan végre akarunk hajtani. Például a már említett körbe futó szöveg így oldható meg a segítségével:

Példaprogram

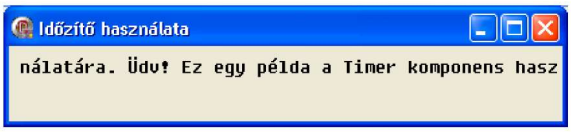
```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
```

```
Label1.Caption := RightStr(Label1.Caption,Length(Label1.Caption)-1) +
LeftStr(Label1.Caption, 1);
end;
```

Mivel a programunk használ két függvényt: RightStr és LeftStr, amelyek az StrUtils unitban találhatók, ki kell egészítenünk programunk uses részét ezzel a unittal:

```
uses
    ..., Strutils;
```

A RightStr függvény az első paraméterként megadott szöveg jobb, a LeftStr a szöveg bal részéből ad vissza a második paraméterben megadott mennyiségű karaktert.



Gauge, ProgressBar komponensek

Egy hosszabb folyamat állapotát jelezhetjük ezeknek a komponenseknek (és a Timer komponens) segítségével.

Nézzük először a **Gauge** komponens fontos tulajdonságait:

MinValue – minimális értéke a sávnak (default: 0).

MaxValue – maximális értéke a sávnak (default: 100).

Progress – aktuális értéke a sávnak.

Például: ha a MinValue = 0 és MaxValue = 200, akkor a Progress = 20 érték 10%-nak felel meg, amely a komponensen is megjelenik.

A Gauge komponens további tulajdonságai:

ForeColor – a kitöltés színe.

Kind – a komponens külalakját határozza meg. Lehetséges értékek: gkHorizontalBar (vízszintes sáv), gkVerticalBar (függőleges sáv), gkNeedle („analóg sebességmérő óra"), gpPie („kalács" - kör formájú alak, melyben a szelet nagyobbodik), gkText (csak a százalékot jeleníti meg, nem szemlélteti semmilyen grafikus elemmel).

Példaprogram

A Form-ra helyezzünk el egy Gauge komponenst és egy Timer komponens. A Timer komponens Interval tulajdonságát állítsuk be 100 milliszekundumra (tehát másodpercenként 10-szer fog bekövetkezni az OnTimer eseménye). Az OnTimer eseményhez a következő programrészt rendeljük hozzá:

```
procedure TForm1.Timer1Timer(Sender: TObject);
```

```
begin
   Gauge1.Progress := Gauge1.Progress + 1;
end;
```

A **ProgressBar** komponens hasonló a Gauge-hoz, csak más a külalakja és mások a tulajdonságainak a nevei: a MinValue, MaxValue és Progress helyett **Min, Max** és **Position** tulajdonságai vannak.

Vizuális komponensek

Ebben a fejezetben főleg olyan további komponenseket sorolunk fel, melyek grafikailag szebbé, érdekesebbé tehetik alkalmazásunkat.

Kép használata - Image

Kép megjelenítését teszi lehetővé. Ezen kívül jól használható rajzprogram készítésére is, mivel tartalmaz egy vásznat **(Canvas** objektum), melyre bármit kirajzolhatunk. Az image komponens leggyakrabban használt tulajdonságai:

Picture – megjelenítendő kép.

Stretch – ha értéke igaz (true), akkor az egész képet széthúzva jeleníti meg a komponensen. Tehát ha nagyobb a kép, akkor lekicsinyíti a komponens méretére, ha kisebb, akkor felnagyítja.

Proportional– ha értéke igaz (true), akkor betartja a szélesség és magasság arányát, tehát nem torzul a kép.

Transparent– bitmap (BMP) kép esetében, ha a transparent tulajdonság értéke igaz (true), akkor a háttérszínt átlátszóvá teszi (csak akkor működik, ha a stretch tulajdonság hamis - false). Háttérszínnek a *Delphi* a bitmap bal alsó sarkában levő pont színét veszi.

Példaprogram

Készítsünk egy egyszerű képernyővédőt, melyben egy kép fog körbe-körbe járni a képernyőn, a másik kép egyszer körbe megy a monitoron utána meg pattog a képernyö alján. Ez mindaddig történjen, amíg nem nyomunk le egy billentyűt vagy nem mozdítjuk meg az egeret.

Mielőtt belekezdenénk a programunk készítésébe, rajzoljuk meg Paint-ban (vagy más rajzoló programban) a léggömböt, melynek mérete legyen 200 x 200 pixel. Ezt a rajzot BMP fájlformátumban mentsük el!

A **Form**-unkra tegyünk két képet **(Image)** és egy időzítőt **(Timer).** A képek lesznek maguk a rajzaink, az időzítő pedig ezt a képet fogja mozgatni (minden tízedik ezredmásodpercben egy képponttal jobbra teszi).

Állítsuk be a komponensek tulajdonságait az Objektum felügyelőben:

- Form1.WindowState **:=** wsMaximized;

Ezzel az ablakunk kezdetben maximalizált állapotban lesz.

- Form1.BorderStyle **:=** bsNone;
- Image1.Picture – ehhez a tulajdonsághoz rendelhetjük hozzá az elmentett

képünket (BMP), amely a léggömböt ábrázolja.

- Image1.Width := 200; Image1.Height := 200;

Meghatározzuk a képünk méretét. Mi esetünkben ez 200 x 200 képpont.

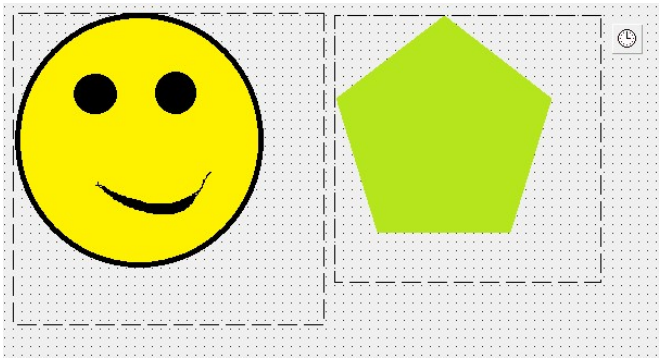
- Image1.Transparent := true;

Megadjuk, hogy képünk háttere átlátszó legyen.

- Timer1.Interval := 10;

Megadjuk, hogy az időzítőnél 10 ezred-másodpercenként következzen be az OnTimer esemény.

Ezzel megadtuk a fontosabb tulajdonságokat. Alkalmazásunk tervezési fázisban most így néz ki:



Ezek után nekiláthatunk az események kezelésének, tehát a programkód megírásának.

```
var
    Form1: TForm1;
    egerpoz1, egerpoz2: TPoint;
```

A **Forml - OnCreate** eseményében beállítjuk véletlen helyre a képet, továbbá az ablakunk hátterének a színét feketére és az egérkurzort kikapcsoljuk:

```
procedure TForm1.FormCreate(Sender: TObject);
  begin
    randomize;
    Imagel.Top := Random(Screen.Height-200);
    Imagel.Left := Random(Screen.Width-200);
    Forml.Color := clBlack;
    Forml.Cursor := crNone;
    Form1.DoubleBuffered := true;
    GetCursorPos(egerpoz1);
  end;
```

Itt azért a **Screen** objektumot használtuk és nem a Form1-et, mert ennek az eljárásnak a meghívásakor az alkalmazásunk még nincs maximalizálva, így nem kapnánk meg az egész képernyő méretét. A Screen (képernyő) objektum segítségével meghatározható a képernyő felbontása úgy, ahogy azt a fenti programrészben is tettük.

Most beállítjuk, hogy a léggömb mozogjon, tehát növeljük a **Timerl – OnTimer** eseményében a kép Left tulajdonságát 5-gyel.

```
procedure TForm1.Timer1Timer(Sender: TObject);
var elmozdulas:integer;
begin
    Image1.Left := Image1.Left + 5;
        begin
          if Image1.Left > Form1.Width-200 then
            begin
                Image1.Top :=  Image1.Top+10;
                Image1.Left :=Image1.Left-5;
            end;
            begin
                if Image1.Top > Form1.Height-200 then
                Image1.Left :=Image1.Left-10;
            end;
            begin
                if Image1.Left < 0 then
                Image1.Top :=  Image1.Top-1;
            end ;
        end;
    Image2.Left := Image2.Left + 5;
        begin
            if Image2.Left > Form1.Width-240 then
            begin
                Image2.Top :=  Image2.Top+10;
                Image2.Left :=Image2.Left-5;
            end;
            begin
                if Image2.Top > Form1.Height-240 then
                Image2.Left :=Image2.Left-10;
            end;
            begin
                if Image2.Left < 5 then
            begin
                Image2.Top :=  Image2.Top-10;
                Image2.Left := Image2.Left-5;
            end;
                if Image2.Top < 5 then
            begin
                Image2.Left :=Image2.Left+10;
            end;
        end;
    end;
    GetCursorPos(egerpoz2);
```

```
elmozdulas := round(sqrt (sqr (egerpoz1.x - egerpoz2.x) + sqr(egerpoz1.y - egerpoz2.y)));
      if elmozdulas > 10 then
      Form1.Close;
  end;
```

Most beállítjuk, hogy bármelyik billentyű megnyomására a program bezáródjon. Ezt a **Forml** – **OnKeyDown** eseményében tehetjük meg:

```
procedure TForm1.FormKeyDown(Sender: TObject;var Key: Word; Shift: TShiftState);
      begin
        Form1.Close;
      end;
end.
```

Futtassuk le az alkalmazásunkat. Próbáljunk meg kilépni az egér mozgatásával. Képernyővédőnk megpróbálhatjuk magunk továbbfejleszteni például úgy, hogy több kép legyen, a méretük változzon, esetleg ne csak vízszintesen haladjon a kép, hanem közben emelkedjen, vagy süllyedjen is.

Választóvonal – Bevel

Ez talán az egyik legegyszerűbb, mégis egy elegáns komponens az alkalmazásunk szebbé tételére. Segítségével a komponenseket vizuálisan elválaszthatjuk egymástól. A Bevel komponenssel tehetünk az alkalmazásunkba egyszerű vonalat vagy keretet (amely lehet benyomódva vagy kiemelkedve is). Fontos megemlíteni, hogy a Bevel komponens nem lesz a ráhelyezett komponensek tulajdonosa (mint pl. a Panel komponensnél), csak vizuálisan jelenik meg a programban (tehát nem csoportosíthatunk vele pl. RadioButton-okat sem logikai csoportokba)!

A komponensnek két fontos tulajdonsága van:

Shape – meghatározza, hogyan nézzen ki a Bevel komponens. Lehetséges értékei:

- **bsBottomLine** – a komponens alján egy vízszintes vonal jelenik meg.
- **bsBox** – doboz (keret), amely belseje attól függően, hogy a komponens Style tulajdonsága mire van állítva vagy beljebb lesz vagy kijjebb lesz az ablakon (Form-on).
- **bsFrame** – keret körbe. A keret belseje egy szintben van az ablakkal (Form-mal).
- **bsLeftLine** – függőleges vonal a komponens bal oldalán.
- **bsRightLine** – függőleges vonal a komponens jobb oldalán.
- **bsSpacer** – üres (nem lesz látható semmi).
- **bsTopLine** – vízszintes vonal a komponens tetején.

Style – meghatározza, hogy az alakzat beljebb lesz (bsLower) vagy kijjebb lesz (bsRaised) az ablakon. Természetesen ez csak azoknál az alakzatoknál (Shape tulajdonság) használható, melyeknél van értelme.

Ennek a komponensnek nincs egyetlen eseménye sem.

Alakzat - Shape

Egyszerű geometriai alakzatok, melyekkel a programunkat szépíthetjük. Hasonlóan az előző komponenshez, nem lesz ez a komponens sem a ráhelyezett komponens tulajdonosa, csak vizuális célt szolgál. Fontosabb tulajdonságai:

Shape – az alakzat formája (stCircle, stEllipse, stRectangle, stRoundRect, stRoundSquare, stSquare).

Brush – kitöltés színe és stílusa.

Pen – körvonal színe, vastagsága és stílusa.

Grafikus nyomógomb – BitBtn

Ha a hagyományos nyomógombtól (Button) egy kicsit eltérőt szeretnénk kialakítani, használhatjuk a BitBtn komponenst. Ez nagyon hasonló a Button komponenshez, de BitBtn nyomógombon képet is el lehet helyezni, így bármilyen egyedi külalakú gombot létrehozhatunk. Itt most a BitBtn komponensnek csak a Button-nál nem található, további fontos tulajdonságait fogjuk felsorolni:

Glyph – a gombon megjelenítendő kép (bitmap). Ha azt szeretnénk, hogy a saját képünk hasonló legyen az előre definiáltakhoz, akkor 18 x 18 képpontnyi méretű képet használjunk.

Kind – néhány előre definiált kép közül választhatunk, lehetséges értékei: bkCustom, bkYes, bkNo, bkOK, bkCancel, bkAbort, bkRetry, bkIgnore, bkAll, bkClose, bkHelp. Ha a bkCustom-ot választjuk, saját képet adhatunk meg (a Glyph tulajdonságban).

Layout – meghatározza, hogy a kép hol jelenjen meg a nyomógombon (bal oldalon, jobb oldalon, fent, lent).

Margin – meghatározza a kép és a gomb bal széle közti távolságot pixelekben. Ha értéke -1, akkor a képet a szöveggel együtt középre igazítja.

Spacing – meghatározza a kép és a felirat közti távolságot pixelekben.

NumGlyphs – megadja, hogy a bitmap hány képet tartalmaz. A bitmap-nek teljesítenie kell két feltételt: minden képnek ugyanolyan méretűnek kell lennie és a képeknek sorban, egymás után kell elhelyezkedniük. A BitBtn komponens ezek után egy ikont jelenít meg ezek közül a képek közül a BitBtn állapotától függően:

- Up – a nem lenyomott gombnál jelenik meg, és a többi állapotnál akkor, ha nincs más kép,
- Disabled – akkor jelenik meg, ha a gombot nem lehet kiválasztani,
- Clicked – ha a gomb éppen meg lett nyomva (rá lett klikkelve egérrel),
- Down – ha a gomb tartósan lenyomott állapotban van (ez az állapot a BitBtn komponensnél nem következik be, ennek az állapotnak csak a SpeedButton gombnál van jelentősége, melyet a következő részben írunk le).

Eszköztár gomb – SpeedButton

Az eddig tárgyalt gombok, a sima Button, a BitBnt gombok nem használhatóak eszköztár gombnak. Gondoljuk végig például Word-ben a szöveg igazítását:

A szöveg igazításánál a gombok közül egy gomb állandóan lenyomott állapotban van. Ezt az eddig már tárgyalt, többi fajta nyomógomb nem tudja megvalósítani. A kölcsönös kizárás lényege ilyen esetben, hogy az egy csoportba tartozó gombok közül, mindig csak egy lehet lenyomva, s a másik lenyomásakor az előző felenged. Külön megadható az is, hogy legalább egynek mindig lenyomva kell-e lennie, vagy egy csoporton belül mindegyik lehet-e felengedett állapotban.

A SpeedButton komponens nagyon hasonló a BitBtn komponenshez, melyet az előző fejezetben tárgyaltunk. Hasonlóan a BitBtn komponenshez ez is tartalmazhat több képet, mindegyik állapotához egy-egy képet. Itt jelentősége lesz a negyedik – down (tartósan lenyomva) állapotnak is.

Legfontosabb tulajdonságai:

Glyph – a gombon elhelyezkedő kép. Négy különböző képet helyezhetünk el a gomb állapotának megfelelően.

GroupIndex – ennek a tulajdonságnak a segítségével csoportosíthatóak a gombok. Az egy csoportba tartozó gomboknak azonos GroupIndex-el kell rendelkezniük. Amennyiben a GroupIndex = 0 akkor a gomb BitBtn gombként fog viselkedni, tehát nem fog „benyomva" maradni. Ha GroupIndex-nek 0-nál nagyobb számot adunk, akkor létrejön a csoportosítás.

Down – ez a tulajdonság adja meg, hogy a gomb lenyomott állapotban van-e (true esetén lenyomott állapotban van). Csak 0-tól különböző GroupIndex esetén használható.

AllowAllUp – ez a tulajdonság határozza meg, hogy az egy csoportba tartozó gombok közül lehet-e mind egyszerre felengedve (true), vagy az egyik gombnak mindenképen muszáj benyomva maradnia (false).

A SpeedButton-okat szinte mindig egy Panel komponensen helyezzük el, eszköztárat alakítva ki belőlük.

Kép lista - ImageList

Ennek a komponensnek a célja, több ugyanolyan méretű kép tárolása. Ezek a tárolt képek (bitmapek, ikonok...) indexek segítségével érhetők el. A komponenst tervezési fázisban egy kis négyzet jelképezi, amely az alkalmazás futása alatt nem látható (hasonlóan pl. a Timer komponenshez).

Az ImageList komponenst hatékonyan ki tudjuk használni több kép vagy ikon tárolására. Az összes kép az ImageList-ben úgy van reprezentálva, mint egy darab széles bitmap. Ezt felhasználhatjuk a BitBtn, SpeedButton komponenseknél, de hasonlóan felhasználható további komponenseknél is (pl. ToolBar).

Mindegyik ListBox-ban tárolt kép index segítségével érhető el, melynek értéke 0-tól N-1-ig lehet (N darab kép esetében). Például: az ImageList1.GetBitmap(0,

Image1.Picture.Bitmap); metódus az ImageList1-ben tárolt 0 indexű képet állítja be az Image1 Bitmap-jának.

Az alkalmazás tervezési fázisában a képek megadásához az Image List Editor eszközt használhatjuk, melyet a komponensre kattintva jobb egérgombbal (vagy dupla kattintással) hívhatunk elő.

Az ImageList fontosabb tulajdonságai:

Width – a komponensben tárolandó képek szélessége.

Height –a komponensben tárolandó képek magassága.

Eszköztár – ToolBar

A ToolBar komponens segítségével szintén előállíthatunk eszköztárt. Az előbbi fejezetekben leírt eszköztár kialakításával (panel és SpeedButton komponensekből) szemben a ToolBar komponens valamivel több lehetőséggel rendelkezik, továbbá más a tervezési fázisa is. Szorosan együttműködik az ImageList komponenssel, melyet az előző részben ismerhettünk meg. Az eszköztár kialakítását ToolBar komponens segítségével egy példán ismertetjük:

1. A Form-on elhelyezünk egy ToolBar és egy ImageList komponenst.
2. Jobb egérgombbal rákattintunk az ImageList-re és kiválasztjuk az Image List Editor-t.
3. Az Editor-ban az Add gomb megnyomásával hozzáadjuk az ImageList-hez a szükséges képeket (bitmapot - BMP és ikont - ICO választhatunk). A képek hozzáadása után az OK gomb megnyomásával bezárjuk az Editor-t.
4. Az egér bal gombjával rákattintunk a ToolBar komponensre, majd az Object Inspector-ban beállítjuk az **Images** tulajdonságot az ImageList komponensünkre.
5. Jobb egérgombbal rákattintunk a ToolBar komponensre, majd kiválasztjuk a „New Button" menüpontot. Ezt a lépést megismételjük annyiszor, ahány gombot akarunk elhelyezni. Ne felejtsünk el néha berakni „New Separator"-t is, hogy a gombok áttekinthetően legyenek elrendezve.
6. Az előző lépés ismétlésénél a gombokhoz automatikusan hozzá lettek rendelve az ImageList-ben levő ikonok. Ha ez a kezdeti beállítás nekünk nem felel meg, nem probléma megváltoztatni őket a ToolBar nyomógombjainak (tehát a **ToolButton** objektumoknak) az **ImageIndex** tulajdonságainak átállításával.

Ennek a komponensnek a segítségével nagyon egyszerűen és rugalmasan kialakíthatunk eszköztárakat. A ToolBar komponens fontosabb tulajdonságait:

DisabledImages – segítségével meghatározhatjuk, hogy a nyomógombokon milyen képek jelenjenek meg, ha a nyomógomb nem elérhető.

HotImages – segítségével meghatározhatjuk, hogy a nyomógombokon milyen képek jelenjenek meg, ha a nyomógomb aktív (ki van választva).

A ToolBars-on elhelyezkedő nyomógombok, választóvonalak (ToolButton objektumok) fontosabb tulajdonságai:

Style - meghatározza a nyomógombok stílusát és viselkedését is. Lehetséges értékek:

- tbsButton:nyomógomb**,**
- **tbsDivider:** látható függőleges választóvonal,
- **tbsDropDown:** legördülő választék - menü (pl. *Word* betűszínének kiválasztása az eszköztárban),
- **tbsCheck:** nyomógomb két lehetséges (lenyomott ill. felengedett) állapottal,
- **tbsSeparator:** üres hely (választósáv).

Grouped – meghatározza, hogy a gombok olyan logikai csoportok bavannak-e osztva, melyekben mindig cask egy gomb lehet lenyomva (pl.*Word*sorigazításai). A gombok logikaicsoportokbavaló osztását az határozza meg, hogyan vannak elválasztva (tbsDivider, tbsSeparatos stílusú ToolButton-okkal). Ennek a tulajdonságnak cask a tbsCheckstílusú nyomógomboknál (ToolButton-oknál) van értelme.

MenuItem – segítségével a gombokhoz a főmenü egyes menüpontjait lehet hozzárendelni.

Állapotsáv – StatusBar

Az alkalmazásunk ablakának alján az állapotsáv (StatusBar) segítségével tudunk kiíratni a felhasználónak különféle információkat. Például egy grafikus editorban kiírathatjuk ide az egér koordinátáit, a kijelölt rész koordinátáit és méretét, a vonalvastagságot, az aktuális betűtípust, stb. Ha StatusBar komponenst rakunk az alkalmazásunkba, az automatikusan az ablak aljához „tapad", mivel az Align tulajdonsága alapértelmezésben erre van állítva. Legfontosabb tulajdonsága:

Panels – tervezési fázisban egy editor segítségével megadhatjuk hány részre legyen szétosztva, pontosabban hány részből álljon az állapotsávunk. Az egyes részeket indexek segítségével érhetjük el. Minden egyes rész egy új, TStatusPanel típusú objektum. A StatusPanel fontosabb tulajdonságai:

- **Width** – meghatározza a szélességét, azonban az utolsó StatusPanel szélessége mindig az alkalmazásunk ablakának szélességétől függ, mivel az utolsó a maradék részt tölti ki.
- **Text** – a StatusPanelon megjelenítendő szöveget tartalmazza.
- **Alignment** – a szöveg igazítását határozza meg a StatusPanelen belül.

Az alkalmazás futási idejében ha meg szeretnénk jelentetni valamit a StatusBar első StatusPanel-jén (ez a 0. indexű), például a betűtípust, azt a következő módon tehetjük meg:

```
StatusBar1.Panels[0].Text := 'Times New Roman';
```

Könyvjelzők – TabControl, PageControl

A *Delphiben* könyvjelzőkkel kétféle képpen dolgozhatunk. Vagy TabControl vagy PageControl segítségével. Első látásra nem látunk a két komponens között különbséget, mégis mindkét komponens működése eltérő.

A **TabControl** csak a könyvjelzők definiálására szolgál. A felső részében megjeleníti a könyvjelzőket, de saját maga nem tartalmaz semmilyen lapokat: ha elhelyezünk egy komponenst valamelyik „lapon" (bár fizikailag nincs egyetlen lapja sem), mindegyik „lapon" látható lesz. Ebből adódik, hogy a lapok közötti átváltást nekünk kell beprogramoznunk (átváltáskor nekünk kell a rajta levő komponenseket láthatóvá ill. láthatatlanná tenni).

A **PageControl** az előzővel ellentétben már tartalmaz lapokat is. Mindegyik lapja tartalmazhat saját komponenseket. Ha valamelyik lapra elhelyezünk egy komponenst, az a többi lapon nem lesz látható, tehát fizikailag is csak az adott lapon lesz rajta.

Ezekből a különbségekből adódik a két komponenssel való eltérő munka és a két komponens eltérő kialakítása is a tervezési fázisban.

TabControl

A TabControl-nál a könyvjelzőket (füleket) a **Tabs** tulajdonság segítségével adhatjuk meg, amely a már ismert TString típusú. Így tervezési időben használhatjuk a String List Editor-t. Továbbá kihasználhatjuk az összes metódust, amelyet a TString típusnál megismertünk. A **TabIndex** tulajdonság meghatározza az aktuális könyvjelzőt.

A TabControl egyik fontosabb eseménye az **OnChanging,** amely akkor következik be, ha a felhasználó át szeretne váltani másik fülre (könyvjelzőre). Ebben az eljárásban megakadályozhatjuk az átváltást is, ha például a felhasználó nem megfelelő értékeket adott meg - erre az AllowChange paraméter szolgál.

PageControl

A PageControl-nál az egyes lapok fizikailag új komponensek (TabSheet). Igaz, hogy ez egy kicsit bonyolítja a tervezését, viszont itt mindegyik lapra külön-külön komponenseket rakhatunk.

A PageControl-nál nincs editorunk, amelyben be tudnánk állítani az egyes könyvjelzőket (füleket). A tervezési fázisban úgy tehetünk bele új lapot, hogy a PageControl komponensre jobb egérkattintással rákattintunk és kiválasztjuk a New Page menüpontot. Minden létrehozott lappal úgy tudunk dolgozni, mint egy külön komponenssel.

A kiválasztott lapot az **ActivePage** tulajdonság határozza meg. Ez egy TTabSheet típusú tulajdonság, tehát egyenesen az adott lapot használja, nem az indexét vagy más „mutatót". A következő vagy előző lapra való átmenésre a programban elég meghívni a PageControl **SelectNextPage** metódusát.

Formázható szövegdoboz – RichEdit

A többi szövegdoboztól a legfőbb eltérés, hogy itt a szöveg formázható. A Memo-hoz hasonló tulajdonságokkal és metódusokkal rendelkezik. További előnye, hogy beolvassa, vagy elmenti RTF állományba a formázott szöveget. Pl. *Wordpad-del* előállíthatunk egy RTF állományt, s azt beolvastathatjuk *Delphibe.*

Tulajdonságai hasonlóak a Memo komponens tulajdonságaihoz, ezért itt csak néhány további fontosabb tulajdonságát említjük meg:

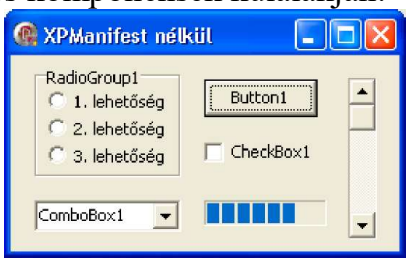
- **Lines** – a Memo-hoz hasonlóan épül fel, TStrings a típusa. A TStrings típusnak van olyan metódusa, mely fájlból olvassa be a szöveget, ill. oda ki tudja menteni. Itt a RichEdit esetében van egy automatikus konverzió, hogy ne az RTF fájl sima szöveges változatát lássuk, hanem a megformázott szöveget. Így nekünk itt is csak a beolvasással vagy a fájlba írással kell törődnünk.
- **PlainText** – igaz (true) érték esetén a szöveget sima TXT állományba menti el, hamis (false) értéknél a mentés RFT fájlba történik.
  Példa egy RTF állomány beolvasására:

```
RichEdit1.Lines.LoadFromFile('c:\delphi.rtf');
```

XPManifest komponens

Ha azt szeretnénk, hogy az alkalmazásunknak, melyet létrehozunk, elegáns kinézete legyen, mint más modern Windows XP / Windows Vista / Windows 7 alatti alkalmazásoknak, használhatjuk az XPManifest komponenst.

Ennek a komponensnek a használata nagyon egyszerű, elég elhelyezni bárhova az alkalmazásunkban (futási időben nem látható). A különbség a program futása alatt mindjárt látható lesz az egyes komponensek külalakján:



Ha el szeretnénk távolítani az XPManifest komponenst az alkalmazásunkból, nem elég kiszedni az alkalmazás ablakából, a teljes eltávolításhoz ki kell törölnünk a programunk uses részéből is az **XPMan** unitot.

Menük létrehozása

Az alkalmazásunkban kétfajta menüt hozhatunk létre: főmenüt és lokális (popup) menüt. Nézzük ezeket sorban.

Főmenü - MainMenu

A főmenü az alkalmazásunk ablakának legtetején helyezkedik el. Mielőtt belekezdenénk a menük létrehozásába a *Delphiben,* nézzük meg milyen követelményeknek kell megfelelnie a főmenünek. Természetesen ezek csak javaslatok, nem kötelező őket betartani, de ajánlott.

A főmenü menüpontjai lehetnek:

- **Parancsok** – azok a menüpontok, melyek valamilyen parancsot hajtanak végre, cselekményt indítanak el.
- **Beállítások** – olyan menüpontok, amelyek segítségével a program valamilyen beállításának ki vagy bekapcsolása lehetséges. Ezeknél a menüpontoknál bekapcsolt állapotban egy „pipa" (vagy pont) van a menüpont bal oldalán.
- **Dialógusok** – menüpontok, melyek hatására egy új ablak (dialógusablak) jelenik meg. Az ilyen menüpontoknál a nevük (feliratuk) után három pont van. Ezt a három pontot a név után mi írjuk be. A három pont kirakása a menüpontban nem kötelező, ez nélkül is működik, de ajánlott ezt az elvet betartanunk.
- **Almenüt megnyitó menüpontok** – olyan menüpont, mely egy mélyebb szinten levő almenüt nyit meg. Az ilyen menüpont a jobb szélén egy kis háromszöggel van megjelölve.

Menüpontot, mely valamilyen parancsot végrehajt, tehetünk a főmenü sávjába is (amely mindig látható az ablak tetején). Ez azonban nem ajánlatos, mivel a felhasználó általában ezekre klikkelve egy menü megnyílását várja el (melyből aztán választhat), nem azonnal egy parancs lefutását. Így ez nagyon zavaró lehet.

Másik dolog, mely a felhasználót zavarhatja, ha a menüből egy almenü nyílik meg, abból egy újabb almenü, stb. Legjobb, ha a menü legfelső szintjére klikkelve megnyílik egy olyan választék, melyből már nem nyílik meg további, alacsonyabb szintű almenü, csak kivételes esetekben. Almenük helyett inkább használjunk a

menüben vízszintes választóvonalakat.

A felhasználó számára másik, nagyon zavaró eset lehet, ha a menüben megváltoznak a menüpontok nevei. Néha ez jól jöhet, pl. ha rákattintunk a „Táblázat megjelenítése" menüpontra, akkor az megváltozhat „Táblázat eltüntetése" menüpontra, de nagyon sok esetben megzavarhatjuk vele a felhasználót (főleg ha a megváltozott név nincs logikai összefüggésben az előzővel, pl. „Táblázat megjelenítése" után ha megjelenne „Lista megjelenítése" ugyanabban a menüpontban).

További zavaró eset lehet, ha a menüben eltűnnek és megjelennek menüpontok. A felhasználók többsége csak a menüpont helyzetét jegyzi meg, nem a pontos nevüket. A menüpontok eltüntetése (visible) helyett használjuk inkább a menüpontok engedélyezésének tiltását (enabled). Így a menüpont a helyén marad, csak „szürke" lesz, nem lehet rákattintani.

A menüpontokat próbáljuk valamilyen logikailag összefüggő csoportokban elrendezni. Használjunk a csoportok között vízszintes választóvonalakat, de azért vigyázzunk, hogy ezt se vigyük túlzásba. Egy összefüggő csoportban jó, ha nincs több 5-6 menüpontnál.

Fontos, hogy betartsuk a menü standard struktúráját, melyek minden alkalmazásban hasonlóak, és melyekhez a felhasználók már hozzászoktak. A menüsávot Fájl, Szerkesztés, Nézet, menüpontokkal kezdjük, a végén legyenek a Beállítások, Eszközök, Ablak, Súgó menüpontok. Az almenüknél is próbáljuk meg betartani a standard elrendezést, pl. a File menüpont alatt legyen az Új, Megnyitás, Mentés, Mentés másként, ,.., Nyomtató beállítása, Nyomtatás, Kilépés menüpontok. Hasonlóan a Szerkesztés alatt legyen a Visszavonás, Kivágás, Másolás, stb.

Tartsuk be a megszokott billentyűkombinációkat is, pl. a Ctrl+C a másolás, Ctrl+V a beillesztés legyen, stb. Nem nagyon örülnének a felhasználók, ha pl. a Crtl+C megnyomásakor befejeződne a program. A másik fajta billentyűkombinációk, melyekre szintén próbáljunk meg odafigyelni: az Alt+betű típusúak, melyeknél, ha lehet, az adott betűre kezdődő menüpont nyíljon meg.

Ezzel összefügg a menü nyelve is. Ha magyar programot készítünk, használjunk benne a menük neveinek (feliratainak) megadásakor is magyar szavakat. Ha külföldön is szeretnénk a programunkat terjeszteni, készítsünk külön egy angol változatot. A billentyűkombinációkat azonban nem ajánlatos lefordítani! Pl. a Ctrl+C maradjon Ctrl+C, ne változtassuk meg pl. Crtl+M-re (mint másolás). A megváltoztatásukkal több kárt érnénk el, mint hasznot.

Ez után a rövid bevezető után nézzük, hogyan készíthetünk a *Delphiben* menüt. Helyezzünk el az ablakunkon bárhova egy **MainMenu** komponenst. A komponens az alkalmazásunkban egy kis négyzettel lesz jelezve, mely természetesen futási időben nem látható. Ha erre a kis négyzetre duplán rákattintunk, megnyílik a Menu Designer, amely segítségével könnyen kialakíthatjuk a főmenünket.

Klikkeljünk az új menüpont helyére, majd adjuk meg a menüpont feliratát **(Caption** tulajdonság). Észrevehetjük, hogy minden egyes menüpontnál vannak külön tulajdonságok. Csak rajtunk múlik, hogy itt beállítjuk-e a **Name** tulajdonságot

is (pl. mnuFajl, mnuSzerkesztes), vagy hagyjuk azt, amit a *Delphi* automatikusan hozzárendelt. A menünk a Menu Designer-ben hasonlóan néz ki, mint ahogy ki fog nézni az alkalmazásunkban, azzal a különbséggel, hogy itt láthatjuk azokat a menüpontokat is, melyeknek a **Visible** tulajdonságuk hamis (false).

Minden menüpontnak egyetlen fontos eseménye van, az **OnClick** esemény. Ebben adhatjuk meg azokat a parancsokat, melyeket végre akarunk hajtani, ha a felhasználó rákattint a menüpontra.

Ha valamelyik menüpontból egy új almenüt szeretnénk megnyitni, kattintsunk rá a tervezési időben jobb egérgombbal és válasszuk ki a „Create Submenu"-t.

Ha a menüpontokat el szeretnénk választani egymástól egy vízszintes vonallal, hozzunk létre oda egy új menüpontot és adjunk meg a **Caption** tulajdonságnak egy kötőjelet (-). A menüben ez a menüpont egy vízszintes választóvonalként fog megjelenni.

Ha szeretnénk, hogy a menüpontot az Alt+betű billentyű kombinációval is el lehessen érni, a **Caption** tulajdonságban a betű elé tegyünk egy „and" (&) jelet. Például: &File, &Szerkesztés, stb.

A menüpontok fontosabb tulajdonságaik:

Checked – meghatározza, hogy a menüpont ki legyen-e jelölve, pontosabban hogy mellette (a bal oldalán) legyen-e pipa (jelölőpont).

Enabled – meghatározza, hogy a menüpont engedélyezve van-e, vagy szürke és nem lehet rákattintani.

GroupIndex – a menüpontok logikai csoportokba való osztását lehet vele megoldani (az ugyanabba a csoportba tartozó menüpontoknak a GroupIndex-e egyforma, 0-nál nagyobb szám).

Radioltem – segítségével meghatározható, hogy az egy csoportba tartozó menüpontok közül egyszerre csak egy lehet-e kiválasztva (kipipálva). Ha néhány menüpontnak ugyanazt a GroupIndex-et állítjuk be (0-nál nagyobb) és a Radioltem értékét igazra (true) állítjuk, akkor a menüpontok közül egyszerre mindig csak egy lehet kiválasztva. A menüpontra kattintva nekünk kell beállítani a programban a Checked tulajdonságot true-ra, nem jelölődik be automatikusan a menüpontra klikkelve.

ShortCut – a menüpont billentyűkombinációját határozza meg. Tervezési időben a billentyűkombinációt az Object Inspector-ban egyszerűen kiválaszthatjuk, futási időben a következő példa mutatja, hogyan állíthatjuk be például a Ctrl+C kombinációt:

```
MainMenuItmMasolas.ShortCut :=ShortCut(Word('C'), [ssCtrl]);
```

A rövidítés (billentyűkombináció) a menüpont mellé (jobb oldalára) automatikusan kiíródik.

Visible – meghatározza, hogy látható-e a menüpont.

Már szó volt arról, hogy a menüpontokat nem jó gyakran változatni, eltüntetni és megjeleníteni. Mégis előfordulhat, hogy a menüpontokat vagy az egész menüt meg szeretnénk változtatni (például váltás a nyelvi verziók között, vagy ha van egy kezdő és egy haladó felhasználónak készített menünk). Ebben az esetben létrehozunk két

menüt (két MainMenu komponenst teszünk a Form-ra) és a Form **Menu** tulajdonságában beállítjuk azt, amelyiket éppen használni szeretnénk.

A menüpontok sok metódussal rendelkeznek. Egyik közülük pl. az **Add** metódus, melynek segítségével futási időben is adhatunk új menüpontot a menü végére. Például:

```
procedure TForm1.Button1Click(Sender: TObject);
var mnltmUj: TMenultem;
begin
  mnltmUj := TMenultem.Create(Self); mnltmUj.Caption := 'Új menüpont';
  mnltmFile.Add(mnltmUj);
end;
```

Ezzel kapcsolatban felsoroljuk, hogy milyen lehetőségeink vannak, ha futási időben szeretnénk új menüpontokat berakni a menübe:

- Berakhatjuk az új menüpontokat a fent említett módon (hasonlóan az **Add**-hoz létezik **Insert** metódus is, amely segítségével beszúrhatunk menüt máshova is, nem csak a végére). A problémánk itt az új menüpont **OnClick** eseményéhez tartozó eljárás megadásánál lehet.
- Létrehozhatunk több menüt (több MainMenu komponenst) és a futási időben ezeket cserélgethetjük (a Form Menu tulajdonságának segítségével).
- Létrehozhatunk egy „nagy" menüt, melybe mindent belerakunk és a menüpontok **Visible** tulajdonságainak segítségével állítgathatjuk, hogy melyek legyenek láthatók.

Lokális (popup) menü – PopupMenu

Manapság már szinte nem létezik olyan alkalmazás, amely ne tartalmazna lokális (popup) menüt. Ezek a menük általában a jobb egérgombbal kattintáskor jelennek meg. Az popup menük varázsa abban rejlik, hogy pontosan azokat a menüpontokat tartalmazza, amelyre az adott pillanatban szükségünk lehet (az aktuális komponenshez vonatkozik).

A *Delphi*ben popup menüt a **PopupMenu** komponens segítségével hozhatunk létre. Az ilyen menü létrehozása nagyon hasonlít a MainMenu létrehozásához, ezért itt ezt nem részletezzük.

Amivel itt foglalkozni fogunk, az az, hogy hogyan lehet bebiztosítani, hogy mindig a megfelelő popup menü jelenjen meg. A megjelenítendő popup menüt a Form-on levő komponensekhez tudjuk külön-külön beállítani, mégpedig a komponensek **PopupMenu** tulajdonságával (egy alkalmazásban természetesen több PopupMenu-nk is lehet).

A PopupMenu a MainMenu-hoz képest egy új tulajdonsággal, egy új metódussal és egy új eseménnyel is rendelkezik. Az új tulajdonsága az **AutoPopup.** Ha ennek értéke igaz (true), akkor a menü automatikusan megjelenik a komponensre kattintáskor a jobb egérgombbal, ahogy azt megszoktuk más programokban. Ha a tulajdonság értéke hamis (false), akkor a menü nem jelenik meg automatikusan, hanem azt a programkódban a **Popup** metódus segítségével jeleníthetjük meg. A

PopupMenu komponens új eseménye az **OnPopup.** Ez az esemény pontosan az előtt következik be, mielőtt megjelenne a popup menü. Itt tehát még letesztelhetünk valamilyen beállításokat, majd azok szerint beállíthatjuk a menüpontokat.

Grafika, rajzolás, szöveg kiírása

A *Delphiben* van egy alapobjektum a rajzolásra - a vászon (TCanvas osztály). Képzeljük el az ablakunkat (Form) úgy, mint egy üres területet, amelyen vászon van. Erre a vászonra **(Canvas)** rajzolhatunk, hasonlóan, mint ahogy a festőművészek is rajzolnak a vászonra.

Canvas objektuma sok grafikus komponensnek van. Vászon van a Form-on, de ugyanúgy megtalálható további komponensekben is, mint pl. az Image-ben és a TBitmap osztályban. Ne feledjük, hogy a vászon nem egy különálló komponens, hanem csak némely komponensnek része. A következő felsorolásból megismerhetjük a vászon legfontosabb tulajdonságait:

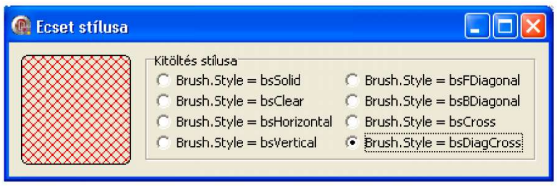
- **Brush** – ecset. A Brush tulajdonság beállításával változik az alakzatok kitöltésének színe és mintája. Az ecsetnek vannak további tulajdonságai is: **Bitmap** (az ecset mintáját definiáló bitkép), **Color** (szín) és **Style** (stílus).
- **Font** – betű. A Font tulajdonságnak is vannak altulajdonságai: **Color, Charset** (karakterkészlet), **Name** (betűtípus neve), **Size** (méret), **Style** (stílus), stb.
- **Pen** – toll. A vászon tollának típusát adja meg. Altulajdonságai: **Color** (szín), **Style** (stílus), **Width** (vonalvastagság), **Mode** (toll rajzolási módja).
- **PenPos** – toll aktuális pozíciója. Ezt a tulajdonságot írni és olvasni is lehet.

Pixels – a pixelek színe. A tulajdonság értékét olvasni és írni is lehet, így rajzolhatunk a vászonra pontonként.

Ecset stílusa

Az első program ebben a fejezetben bemutatja, hogyan lehet a vászonra kirajzolni egyszerű geometriai alakzatot megadott kitöltési stílussal.

Példaprogram



Az egyes események kezelésének programkódja:

```
procedure TForml.FormCreate(Sender: TObject);
begin
    RadioGroupl.Columns := 2;
    RadioGroupl.ItemIndex := 0;
end;
procedure TForml.FormPaint(Sender: TObject);
begin
  Canvas.Brush.Style := TBrushStyle(RadioGroupl.ItemIndex);
```

```
  Canvas.Brush.Color := clRed;
  Canvas.RoundRect(10,10,100,100,10,10);
end;
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  Repaint;
end;
```

Maga a kirajzolás az OnPaint eseményben történik. Ez az esemény mindig akkor következik be, ha szükséges átrajzolni az ablakot (pl. ha az ablak el volt takarva másik ablakkal, ha indult az alkalmazás, ha minimalizálás után lett visszaállítva, stb.).

Miután a felhasználó rákattint valamelyik választógombra (RadioGroup), kikényszerítjük az ablak átrajzolását a **Repaint** metódus segítségével.

Az alkalmazásban beállítjuk a **Canvas.Brush.Color** és a **Canvas.Brush.Style** tulajdonságok segítségével az ecsetet. Az ecset stílusának beállításánál az ItemIndex aktuális értékét átalakítjuk (áttipizáljuk) TBrushStyle típusra, így rögtön hozzárendelhetjük a **Brush.Style** tulajdonsághoz.

A négyzet kirajzolását a **RoundRect** (lekerekített sarkú téglalap) metódus segítségével biztosítjuk be. Megpróbálhatunk más alakzatokat is kirajzolni, pl. a **Rectangle** (téglalap), **Pie** (körszelet), **Polygon, Polyline, Chord,** stb. segítségével.

Szöveg grafikus kiírása

A vászonra nem csak írhatunk, de rajzolhatunk is. A következő alkalmazás egyrészt szemlélteti a szöveg kiírását grafikus módban, másrészt megmutatja, hogyan dolgozhatunk a FontDialog komponenssel (erről bővebben a későbbi fejezetekben lesz szó). Miután a felhasználó ennek a dialógusablaknak a segítségével választ betűtípust, a kiválasztott betűtípussal kiírunk egy szöveget a Form-ra. A FontDialog komponensen kívül szükségünk lesz még egy Button komponensre. Az alkalmazásban kezelni fogjuk a Button komponens OnClick eseményét és a Form OnPaint eseményét.

Példaprogram

Az események eljárásaihoz tartozó programkódok:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if FontDialog1.Execute then
    Canvas.Font.Assign(FontDialog1.Font);
```

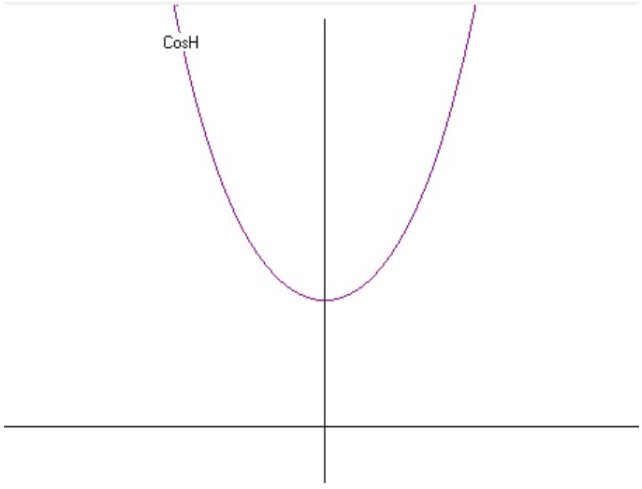
```
     Repaint;
end;
procedure TForm1.FormPaint(Sender: TObject);
begin
   Canvas.TextOut(20,50,'Teszt szöveg');
end;
```

A betűtípus kiválasztása után a vászon **Font.Assign** metódusát használjuk, amely az egyik betűtípus összes atribútumát átmásolja a másikba.

A betűtípus változtatásakor meghívjuk az ablak átrajzolására szolgáló **Repaint** metódust.

Az OnPaint eseményben kiírjuk a szöveget a **TextOut** metódus segítségével, melynek első két paramétere a szöveg koordinátáit jelentik.

Példaprogram




```
procedure TForm1.Button1Click(Sender: TObject);
var i : integer;
begin
        with Image1.Canvas do
          begin
          MoveTo(300,10);
          LineTo(300,600);
          MoveTo(10,300);
          LineTo(600,300);
          Pen.Color := clPurple;
          MoveTo(195,0);
          for i:=-300 to 600 do LineTo(i+300,
          round(300-((exp(i*Pi/180)+exp((-i)*Pi/180))/2)*90));
          Image1.Canvas.TextOut(185,20,'CosH');
          end;
end;
```

Hibák a program futásakor, kivételek kezelése

Amint sejtjük, hogy a program egy adott részében előfordulhat hiba a futása közben, ezt a hibát megfelelően kezelnünk kell még akkor is, ha a hiba csak nagyon ritkán, kis valószínűséggel fordul elő.

Létrehozunk egy eljárást, amely segítségével szemléltetni fogjuk az egyes hibakezelési lehetőségeket. Hozzunk létre egy ablakot (Form), tegyünk rá egy nyomógombot (Button), majd a nyomógomb OnClick eseményének kezelésébe írjuk be az alábbi programrészt:

```
procedure Form1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
  begin
  a := 0;
  b := 0;
  c := a div b;
  Button1.Caption := IntToStr(c);
  end;
```

A program kiszámolja, majd beírja a nyomógomb feliratába két szám egész részű hányadosát (div).

Ebben a programban nincs kezelve semmilyen hiba. Mivel az a, b változókba 0-t tettünk a program elején, nyilvánvaló, hogy a hiba bekövetkezik (osztás nullával). Ez a hiba egy kivételt eredményez, melyet a div művelet generál. Ha a programot lefuttatjuk és megnyomjuk a nyomógombot, láthatjuk az üzenetet a kivételről. Még ha mi nem is kezeltük a hibát, láthatjuk, hogy a kivétel kezelve van. Hogy miért van ez így, erről a későbbiekben lesz szó.

Hibák kezelése hagyományos módon

A hiba kezelése hagyományos módon általában feltételek segítségével történik, melyekben valamilyen változók, hibakódok, függvények és eljárások visszaadási értékeit figyeljük. Ennek a módszernek a hátrányai egyszerűen megfogalmazva a következők:

- a hibakódokat meg kell jegyeznünk,
- minden függvény az eredménytelenséget másképp reprezentálja - false értéked ad vissza, 0-t, -1-et, stb.,
- az eljárásokban a hiba valamelyik paraméterben van megadva, esetleg valamilyen globális paraméterben.

Nézzük meg hogyan kezelnénk hagyományos módszerekkel a hibát az előző programunkban.

Példaprogram

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
```

```
  begin
       a := 0;
       b := 0;
       if b<>0 then
              begin
                     c := a div b;
                     Button1.Caption := IntToStr(c);
              end
      else
              begin
                     ShowMessage('Nullával nem lehet osztani!');
              Button1.Caption := 'Hiba';
             end;
end;
```

Hibák kezelése kivételek segítségével

Most ugyanebben az eljárásban a hibát kivételek segítségével fogjuk kezelni. Nem baj, ha még nem ismerjük a kivételek használatának pontos szintaxisát, ebben a példában csak azt mutatjuk be, hogyan fog az eljárásunk kinézni.

Példaprogram

```
procedure TForml.ButtonlClick(Sender: TObject); var
  a, b, c: Integer; begin a := 0; b := 0;
  try
      c := a div b;
      Buttonl.Caption := IntToStr(c);
  except
      on EdivByZero do begin
        ShowMessage('Nullával nem lehet osztani!'); Buttonl.Caption := 'Hiba'; end;
  end;
end;
```

Ez program kiírja a hibát, majd a nyomógomb feliratában megjeleníti a „Hiba" szót. Ha a program elején a b változó értékét megváltoztatjuk nem nulla számra, akkor a program az osztás eredményét megjeleníti a nyomógomb feliratában.

Ez az egyszerű példa bemutatja a munkát a kivételekkel. Az kivételek egész mechanizmusa négy kulcsszón alapszik:

- **try**– a védett kód elejét jelzi, tehát azt a programrészt, amelyben előreláthatóan bekövetkezhet a hiba,
- **except**– a védett kód végét jelenti, a kivételek kezelésére szolgáló parancsokat tartalmazza a következő formában: on *kivétel_típpusa* do *parancsok* else *parancsok*
- **finally**– annak a programrésznek az elejét jelzi, amely minden esetben végrehajtódik, akár bekövetkezett a kivétel, akár nem. Ezt a részt általában a lefoglalt memória felszabadítására, megnyitott fájlok bezárására használjuk.

- **raise**– kivétel előhívására szolgál. Még ha úgy is tűnik, hogy a kivételt értelmetlen dolog kézileg előhívni, mégis néha hasznos lehet.

Mielőtt konkrét példákon megmutatnánk a kivételek használatát, elmondunk néhány dolgot a kivételekről és a program állapotáról. Ha bekövetkezik valamilyen kivétel, akkor kerestetik egy „kezelő eljárás", amely a kivételt kezeli. Ha az adott programrészben nincs ilyen eljárás, akkor a kivétel „feljebb vivődik" mindaddig, amíg valaki nem foglalkozik vele. Extrém esetekben ezt maga a *Delphi* kezeli, ezért van az, hogy végül minden kivétel kezelve lesz. Fontos, hogy a kivétel kezelése után a program a „kivételt kezelő eljárás" után fog folytatódni, nem a kivételt okozó programkód után.

Nézzük meg most részletesebben a **finally** részt. Ezt a részt olyan tevékenységek elvégzésére használjuk, amelyet el akarunk végezni minden esetben, akár a kivétel bekövetkezik, akár nem. Ilyen pl. a memória felszabadítása.

Nézzünk most példát a kivételek kezelésére a finally blokk nélkül (a memória felszabadítása helyett most az ablak feliratát fogjuk megváltoztatni „Szia"-ra).

```
procedure Form1.Button1Click(Sender: TObject);
vara, b, c: integer;
begin
a := 0;
b := 0;
    try
        c := a div b;
    Button1.Caption := IntToStr(c);
    Form1.Caption := 'Szia';
    except
    on EdivByZero do begin
        ShowMessage('Nullával nem lehet osztani!'); Button1.Caption := 'Hiba'; end;
    end;
end;
```

Ha bekövetkezik a 0-val való osztás, soha nem lesz végrehajtva az ablak feliratának beállítása „Szia"-ra. A megoldás a **finally** blokk használata lehet:

```
procedure Form1.Button1Click(Sender: TObject); var
    a, b, c: Integer;
    begin
    a := 0;
     b := 0;
    try
    c := a div b;
    Button1.Caption := IntToStr(c);
    finally
    Form1.Caption := 'Szia';
    end;
end;
```

Most már biztosak lehetünk benne, hogy az ablak feliratának megváltoztatása (memóriatisztogatás) minden esetben megtörténik. Sajnos azonban most nincs lekezelve a kivételünk, ami végett az egészet tettük. A megoldás: kombináljuk (egymásba ágyazzuk) a **finally** és az **except** blokkot.

```
procedure TForm1.Button1Click(Sender: TObject); var
   a, b, c: Integer; begin a := 0; b := 0;
   try try
      c := a div b;
      Button1.Caption := IntToStr(c);
   except
      on EdivByZero do begin
         ShowMessage('Nullával nem lehet osztani!'); Button1.Caption := 'Hiba';
         end;
finally
      Forml.Caption := 'Szia'; end;
end;
```

Except blokk szintaxisa

Az **except** rész több felhasználási lehetőséget is ad:

```
try
   {parancsok}
except
   on {kivétel_típusa} do
      {ennek a kivételnek a kezelése} on {kivétel_típusa} do
      {ennek a kivételnek a kezelése}
   else
      {bármilyen más kivétel kezelése}
end;
```

Láthatjuk, hogy az else részben bármilyen más kivételt kezelhetünk, melyet előre nem vártunk. Az ismeretlen kivételek kezelésénél azonban legyünk maximálisan óvatosak. Általában legjobb az ismeretlen kivételeket nem kezelni, így a *Delphire* hagyni. Az sem jó ötlet, ha a kivételt kezeljük pl. egy MessageBox-al, majd újból előhívjuk, mivel ebben az esetben a felhasználó kétszer lesz figyelmeztetve: egyszer a saját MessageBox-unkkal, egyszer pedig a Delhi MessageBox-ával. Tehát a kivételt vagy kezeljük, vagy figyelmen kívül hagyjuk, így a standard kezelése következik be.

Ha a kivételt kezeljük, lehetőségünk van például egy új kivétel meghívására megadott hibaszöveggel:

```
raise EConvertError.
Create ('Nem lehet konvertálni!') ;
```

Műveletek fájlokkal

A fájlok támogatását a *Delphiben* három pontba lehet szétosztani:

- az Object Pascal-ból eredő fájltámogatásra. Ennek az alap kulcsszava a File.
- a vizuális komponenskönyvtár fájltámogatása, amelyben metódusok segítségével lehet adatokat beolvasni ill. elmenteni (pl. LoadFromFile, SaveToFile metódusok)
- fájltámogatás adatbázis formátumokhoz. Ez csak a Delphi Professional változatától érhető el, ezzel nem fogunk foglalkozni ebben a fejezetben.

Fájltámogatás az ObjectPascal-ban

A fájlokkal való munkát az Object Pascalban egy példa segítségével említjük meg. Hozzunk létre egy ablakot (Form), melyen helyezzünk el egy Button és egy Memo komponenst. A gomb megnyomásakor az aktuális könyvtárban található DATA.TXT fájl tartalmát beolvassa a program a Memo komponensbe.

```
procedure TForm1.Button1Click(Sender: TObject);
var
   fajl: TextFile;
   sor: String;
begin
   AssignFile(fajl,'data.txt');
   Reset (fajl) ;
   while not Eof(fajl) do
   begin
   ReadLn(fajl,sor);
   Memol.Lines.Add(sor);
   end;
   CloseFile(fajl);
end;
```

Lehet, hogy a Pascal-ból megszoktuk a Text (szöveg fájl típusa), Assign (fájl hozzárendelése), Close (fájl bezárása) parancsokat. Ezek a *Delphiben* **TextFile, AssignFile** és **CloseFile** parancsokkal vannak helyettesítve. Ennek az oka az, hogy a *Delphi*ben az eredeti parancsok máshol vannak használva (pl. a Text több komponens tulajdonsága, pl. Edit, Memo). Az eredeti parancsszavak is a *Delphiben* továbbra is megmaradtak, de a System modullal lehet csak őket használni. Pl. az Assign(F) helyett a System.Assign(F) parancsot használhatjuk.

Ha más, nem szöveges fájlal szeretnénk dolgozni, hanem valamilyen típusos állománnyal, akkor használhatjuk a file of <tipus> formát a deklaráláshoz, pl. file of Integer.

Fájlokkal kapcsolatos leggyakrabban használt parancsok:

- **AssignFile(fájl, fizikai_név)**– a fájl változóhoz egy fizikai fájl hozzákapcsolása a merevlemezen,
- **Reset(fájl)** – fájl megnyitása olvasásra,
- **Rewrite(fájl)**– fájl megnyitása írásra,
- **Read(fájl, változó)**– egy adat olvasása fájlból,
- **Write(fájl, változó)**– egy adat írása fájlba,

- **ReadLn(fájl, szöveg)** – sor olvasása szöveges (txt) fájlból,
- **WriteLn(fájl, szöveg)**– sor írása szöveges (txt) fájlba,
- **Seek(fájl, pozíció)**– a mutató beállítása a megadott helyre a típusos fájlban. A pozíció értéke 0-tól számolódik (0-első adat elé állítja be a mutatót, 1-második adat elé, 2-harmadi adat elé, stb.),
- **CloseFile(fájl) -** állomány bezárása.

Fájltámogatás a Delphiben

Sokszor nem akarunk foglalkozni a „hosszadalmas" Object Pascalból eredő fájltámogatással, hanem helyette egy rövid, egyszerű megoldást szeretnénk használni. Erre is van lehetőségünk a *Delphiben.* A legismertebb metódusok a **LoadFromFile** és a **SaveToFile,** melyek adatokat beolvasnak (megjelenítenek) ill. elmentenek a fájlba egyetlen parancssor beírásával. Ezek a metódusok elérhetők pl. a TString, TPicture, TBitmap osztályokban, ahogy további osztályokban is.

Változtassuk meg az előző példánkat a LoadFromFile metódus használatával.

```
procedure TForml.ButtonlClick(Sender: TObject);
begin
    Memol.Lines.LoadFromFile('data.txt');
end;
```

Láthatjuk, hogy ez így mennyivel egyszerűbb. Nem kell deklarálnunk változókat, megnyitni, bezárni az állományt, hozzárendelni a külső fájl a változónkhoz.

Hibák a fájlokkal való munka során

A *Delphi* bármilyen I/O hiba esetében **ElnOutError** kivételt generál. A hiba pontosabb definíciója az **ErrorCode** lokális változóban szerepel, melynek értéke a következők lehetnek:

ErrorCode	Jelentése
2	File not found
3	Invalid file name
4	Too many open files
5	Access denied
100	Disk read error - ha pl. a fájl végéről akarunk olvasni (eof)
101	Disk write error - ha pl. teli lemezre akarunk írni
102	File not assigned - ha pl. nem volt meghívva az Assign
103	File not open - ha pl. olyan fájlból akarunk dolgozni, amely nem volt megnyitva Reset, Rewrite, Append segítségével
104	File not open for input - ha olyan fájlból akarunk olvasni, amely írásra volt megnyitva
105	File not open for output - ha olyan fájlba akarunk írni, amely olvasásra volt megnyitva
106	Invalid numeric format - ha pl. nem számot akarunk beolvasni szöveges fájlból szám típusú változóba

A kivételek standard kezelése természetesen képes a kivételt kezelni, ha azt nem tesszük meg a programunkban.

A következő példa bemutatja a kivételek kezelését a fájl beolvasásakor a Memo komponensbe.

Példaprogram

```
Procedure TForm1.Button1Click(Sender: TObject);
var
    fajl: TextFile;
    sor: String;
begin
    AssignFile(fajl,'data.txt');
try
    Reset (fajl) ;
    try
        while not Eof(fajl) do
        begin
            ReadLn(fajl,sor);
            Memol.Lines.Add(sor);
        end;
    finally
        CloseFile(fajl);
```

```
        end;
    except
                                        on E:EInOutError do case E.ErrorCode of
        2: ShowMessage('Nincs meg a fájl!');
        103: ShowMessage('A fájl nem volt megnyitva!');
        else
        ShowMessage('Hiba: ' + E.Message);
        end;
    end;
end;
```

Ebben a példában kezeltük a hibákat a fájl megnyitásánál és a fájlból való olvasáskor is.

Képzeljük el, hogy egy programban több helyen is dolgozunk az állományokkal. Leírni mindenhova ugyanazt a programrészt a kivételek kezelésére unalmas és hosszadalmas lehet. Szerencsére ez fölösleges is, mivel használhatjuk a **TApplication** objektum **OnException** eseményét, melybe beírjuk a kivételeket kezelő programrészt „egyszer s mindenkorra". Ezt egy egyszerű példán szemléltetjük, melyben bármilyen nem kezelt kivétel esetében a program leáll.

Példaprogram

```
Procedure TForm1.FormCreate(Sender: TObject);
begin
      Application.OnException := AppException;
end;
Procedure TForm1.AppException(Sender: TObject; E: Exception);
begin
      Application.ShowException(E); // hibaüzenet
Application.Terminate;// programleállás
end;
```

További fájlokkal kapcsolatos parancsok

Csak röviden megemlítjük a *Delphi* további metódusait, melyekkel a fájlok és a mappák összefüggnek:

- **FileExist**(név) – értéke true, ha a megadott nevű állomány létezik.
- **DeleteFile**(név) – kitörli a megadott nevű állományt és true értéket ad vissza, ha a törlés sikeres volt.
- **RenameFile**(régi_név, újnév)– átnevezi a fájlt és true értéket ad vissza, ha az átnevezés sikeres volt.
- **ChangeFileExt**(név, kiterjesztés)– megváltoztatja a fájl kiterjesztését és visszaadja az fájl új nevét.
- **ExtractFileName**(teljes_név) – A teljes útvonallal megadott fájlnévből kiszedi és visszaadja csak a fájl nevét.
- **ExtractFileExt**(név) – Az adott fájl kiterjesztését adja vissza.

További fájlokkal és mappákkal kapcsolatos függvények találhatók a **SysUtils** modulban.

Standard dialógusablakok

Mielőtt belekezdenénk a standard dialógusablakok ismertetésébe, nézzünk meg még néhány komponenst, melyek az állományokkal, mappákkal való munkát segítik. Ezek a komponensek nem dolgoznak közvetlenül a fájlokkal, könyvtárakkal, hanem csak a neveiket jelenítik meg, választási lehetőséget adnak, stb.

Az alábbi komponensek igaz, egy kicsit régebbiek, de lehet őket jól használni. Ezek a Win 3.1 kategória alatt találhatók:

- **FileListBox** – ez egy speciális ListBox az aktuális könyvtárban található összes fájl kiírására.
- **DirectoryListBox** – szintén egy speciális ListBox, amely az adott meghajtón levő könyvtárszerkezet megjelenítésére szolgál.
- **DriveComboBox** – egy speciális legördülő lista, amely számítógépről elérhető meghajtók listáját tartalmazza.
- **FilterComboBox** – a fájlok maszkjainak megjelenítésére szolgáló legördülő lista.

Sok alkalmazásnak hasonló igénye van: fájlokat megnyitnak meg, mentenek el, keresnek valamilyen kifejezést, nyomtatnak, színt (betű, háttér) választanak, stb. Ezért léteznek úgynevezett „common dialog"-usok, tehát előre definiált **standard dialógusablakok.**

Előnyei:

- ugyanaz az ablak jelenik meg a felhasználónak minden alkalmazásnál (pl. mentésnél, megnyitásnál, stb.), így könnyen tudja kezelni,
- a programozónak is egyszerűbb ezeket használnia, mint sajátot készíteni.

Hátrányai:

- egy bizonyos határokon túl nem lehet őket változtatni, úgy kell használnunk őket, ahogy kinéznek, még ha a dialógusablak néhány funkcióját nem is használjuk vagy hiányzik valamilyen funkció,
- ezek standard Windows dialógusablakok, ezért ha pl. magyar nyelvű programunkat angol Windows alatt futtatjuk, keveredik a két nyelv – a programunk magyar marad, de a dialógusablakok angolul lesznek.

A *Delphi*ben a következő dialógusablakokkal találkozhatunk:

Dialógusablak neve	Modális?	Jelentése
OpenDialog	igen	Az állomány kiválasztása megnyitáshoz.
SaveDialog	igen	Az állomány megadása mentéshez.
OpenPictureDialog	igen	Az állomány kiválasztása megnyitáshoz, tartalmazza a kiválasztott kép előnézetét is.
SavePictureDialog	igen	Az állomány megadása mentéshez, tartalmazza a kép előnézetét is.
FontDialog	igen	A betűtípus kiválasztására szolgáló dialógusablak.
ColorDialog	igen	Szín kiválasztására szolgáló dialógusablak.
PrintDialog	igen	A nyomtatandó dokumentum nyomtatóra való küldéséhez szolgáló dialógusablak.
PrinterSetupDialog	igen	A nyomtató (nyomtatás) beállítására szolgáló dialógusablak.
FindDialog	nem	Szövegben egy kifejezés keresésére szolgáló dialógusablak.
ReplaceDialog	nem	Szövegben egy kifejezés keresésére és kicserélésére szolgáló dialógusablak.

Megjegyzés: A modális ablak olyan ablak, amelyből nem lehet átkapcsolni az alkalmazás másik ablakába, csak a modális ablak bezárása után.

A dialógusablakok tervezési időben egy kis négyzettel vannak szemléltetve, amely futási időben nem látszódik. A dialógusablakot az **Execute** metódussal lehet megnyitni. Ez a metódus true értéket ad vissza, ha a felhasználó az OK gombbal zárta be, false értéket pedig ha a felhasználó a dialógusablakból a Cancel gombbal vagy a jobb felső sarokban levő X-szel lépett ki. A PrinterSetupDialog kivételével mindegyik dialógusnak van **Options** tulajdonsága, amely több true/false altulajdonságból áll.

OpenDialog, SaveDialog

Ez a két dialógusablak annyira hasonlít egymásra, hogy egyszerre vesszük át őket.

Tulajdonságok:

- **DefaultExt** – kiterjesztés, amely automatikusan hozzá lesz téve a fájl nevéhez, ha a felhasználó nem ad meg.
- **FileName** – a kiválasztott állomány teljes nevét (útvonallal együtt) tartalmazza. Lehet megadni is mint bemeneti érték.
- **Files** – read-only, run-time tulajdonság, amely a kiválasztott állomány (állományok) teljes nevét (neveit) tartalmazza útvonallal együtt.
- **Filter** – az állományok szűrése oldható meg a segítségével (megadott maszk alapján, pl. *.txt, *.doc, stb.). A program tervezési fázisában ez a tulajdonság a Filter Editor segítségével adható meg.

- **FilterIndex** – melyik legyen a „default" maszk a dialógusablak megnyitásakor.
- **InitialDir** – kezdeti könyvtár (mappa).
- **Options** – beállítási lehetőségek (logikai értékek), melyekkel a dialógusablak külalakját lehet módosítani:
  ofOverwritePrompt – megjelenik a figyelmeztetés, ha a felhasználó létező fájlt akar felülírni.
  ofHideReadOnly– nem jelenik meg a „megnyitás csak olvasásra" lehetőség a dialógusablakon.
  ofShowHelp – megjelenik a „Help" nyomógomb. Ha nincs súgónk hozzá, akkor ezt ajánlott letiltani.
  ofAllowMultiSelect – lehetőséget ad több állomány kiválasztására egyszerre. A kiválasztott fájlokat a TString típusú **Files** tulajdonságban kapjuk vissza.
  ofEnableSizing – lehetőséget ad a felhasználónak változtatni a dialógusablak méretét.
  ofOldStyleDialog – „régi stílusú" dialógusablakot jelenít meg.
- **Title** - a dialógusablak felirata (Caption helyett).

Események:

A legfontosabb események az **OnShow** és az **OnClose,** melyek a dialógusablak megnyitásakor ill. bezárásakor következnek be. Hasznos esemény lehet még az **OnCanClose,** melyben a dialógusablak bezárását lehet megakadályozni. Továbbá használhatjuk még az **OnFolderChange, OnSelectionChange, OnTypeChange** eseményeket is, melyek akkor következnek be, ha a felhasználó megváltoztatja a mappát, fájlok kijelölését ill. a fájlok maszkját (szűrő).

Metódusok:

Legfontosabb metódus az **Execute,** mellyel a dialógusablakot megjelentetjük.

Példa: a kiválasztott állomány nevét a Form1 feliratába szeretnénk kiírni, majd megállapítani, hogy az állomány csak olvasásra lett-e megnyitva.

```
if OpenDialog1.Execute then
   begin
   Form1.Caption := OpenDialog1.Filename;
         if ofReadOnly in OpenDialog1.Options then ShowMessage('Csak olvasásra
     megnyitott.');
   end;
```

OpenPictureDialog, SavePictureDialog

Hasonló az előzőkhöz, de a dialógusablak része a kép előnézetét mutató felület is. Ezt az előnézetet azonban csak akkor láthatjuk, ha a képet felismeri a TPicture osztály, tehát ha a kép .bmp, .ico, .wmf, .emf típusú.

FontDialog

Ez a dialógusablak biztosan mindenki számára ismerős a szövegszerkesztő programokból.

Tulajdonságai:

- **Device** – meghatározza melyik berendezés számára van a betűtípus (fdScreen, fdPrinter, fdBoth).
- **Font** – bemeneti és kimeneti információk a betűtípusról (bemeneti lehet pl. az aktuális betűtípus).
- **MinFontSize, MaxFontSize** – meg lehet segítségükkel határozni milyen értékek között választhat a felhasználó betűméretet. Szükséges hozzá még engedélyezni a **fdLimitSize**-ot az **Options** tulajdonságban. Ha értéknek a 0-t hagyjuk, akkor a választás nem lesz korlátozva.
- **Options** – különféle beállítási lehetőségek:
  fdAnsiOnly – csak szöveges betűtípusokat jelenít meg, tehát kiszűri pl. a Symbols, Wingdings, stb. betűtípusokat.
  fdApplyButton – megjeleníti az „Alkalmaz" nyomógombot.
  fdEffects – a betűstílusok beállításának lehetőségét jeleníti meg a dialógusablakban (pl. aláhúzott, stb.)
  fdTrueTypeOnly – csak a True-Type betűtípusokat jeleníti meg.
  fdForceFontExist – ajánlott értékét true-ra állítani, különben a felhasználó megadhat olyan nevű betűtípust is, amely nem létezik.

Események:

Az előző dialógusokhoz képest van egy új eseménye, az **OnApply,** amely akkor következik be, ha a felhasználó megnyomja az „Alkalmaz" nyomógombot.

Metódusok:

Legfontosabb metódusa az **Execute.**

Nézzünk egy konkrét példát a FontDialog használatára. A példánkban a Memo komponens betűtípusát szeretnénk beállítani.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if FontDialog1.Execute then
    Memo1.Font := FontDialog1.Font;
end;
```

ColorDialog

Legfontosabb tulajdonsága a **Color,** melyben megadható és melyből kiolvasható a konkrét szín. A **CustomColor** tulajdonság segítségével definiálhatunk 16 felhasználói színt. A definiáláshoz klasszikus String List Editor-t használhatunk, melyben a formátum a következő:

ColorX=AABBCC,

ahol X helyére A..P betűket írhatunk, az AA, BB, CC pedig a szín egyes összetevőit jelölik hexadecimális számrendszerbe.

Az **Options** tulajdonság altulajdonságai:

- **cdFullOpen** – az egész dialógusablak megnyitását eredményezi (tehát a felhasználói színeket is).
- **cdPreventFullOpen** – megakadályozza (tiltja) a felhasználói színek részének

megnyitását a dialógusablakban.

PrinterSetupDialog, PrintDialog

A **PrinterSetupDialog** a nyomtató beállításait megjelenítő dialógusablakot nyit meg. Ennek nincs semmi különösebb eseménye vagy tulajdonsága. Amit ezzel a dialógusablakkal kapcsolatban szükséges megtennünk, az csak annyi, hogy megnyitjuk az **Execute** metódussal. A dialógusablak formája szorosan összefügg a beinstallált nyomtató típusával.

A **PrintDialog** komponensnek már van néhány paramétere. Be lehet állítani kezdeti értékeket vagy ki lehet olvasni beállított értékeket, melyek megadhatják például: a másolatok számát **(Copies** tulajdonság), a leválogatás módját **(Collage** tulajdonság). Szintén be lehet határolni (korlátozni) a kinyomtatandó oldalak számát **(MinPage, MaxPage).**

FindDialog, ReplaceDialog

A **FineDialog** és **ReplaceDialog** ablakoknál szintén csak a dialógusablakokról van szó, amely kizárólag az adatok bevitelére szolgál, magát a keresést, cserét sajnos nekünk kell teljes mértékben beprogramoznunk.

A szöveget, amelyet keresnünk kell a **FindText** tulajdonságban kapjuk meg. A ReplaceDialog-nak van még egy **ReplaceText** tulajdonsága is.

Az **Options** tulajdonság lehetőségei:

- **frDown** – keresés iránya, true értéke azt jelenti, hogy az alapértelmezett = lefelé.
- **frMatchCase** – meg legyenek-e különböztetve a kis és nagybetűk.
- **frWholeWord** – csak egész szavak legyenek keresve.

Ha ezek valamelyikét egyáltalán nem szeretnénk a felhasználónak megjeleníteni a dialógusablakban, használjuk a Hide-al kezdődő lehetőségeket (pl. **frHideMatchCase,** stb.). Van lehetőség arra is, hogy ezek a lehetőségek a felhasználónak megjelenjenek, de le legyenek tiltva. Ehhez a Disable szóval kezdődő lehetőségeket használhatjuk (pl. **DisableMatchCase).**

Események:

A FindDialog **OnFind** eseménnyel rendelkezik, amely akkor következik be, ha a felhasználó a „Find Next" nyomógombra kattintott. A ReplaceDialog még rendelkezik egy **OnReplace** eseménnyel is, amely a „Replace" nyomógomb megnyomásakor következik be.

Metódusok:

Ezek a dialógusablakok nem modálisak, tehát a képernyőn maradhatnak többszöri keresés / csere után is. Az **Execute** metóduson kívül rendelkezésre áll még az **CloseDialog** metódus is, amely bezárja a dialógusablakot.

Több ablak (Form) használata

Az alkalmazások készítésénél egy idő után eljön az a pillanat, amikor már nem elég egy ablak az alkalmazás elkészítéséhez. Szükségünk lehet további ablakokra, amelyek segítségével például valamilyen adatokat viszünk be a programba, beállításokat állítunk be, stb.

Tehát az alkalmazásnak lesz egy fő ablaka (ez jelenik meg rögtön a program indításakor) és lehetnek további ablakai (pl. bevitelre, beállítások megadására, stb.). Ezek az ablakok két féle módon jeleníthetők meg:

- modális ablakként: az így megjelenített ablakból nem tudunk átlépni az alkalmazás másik ablakába.
- nem modális ablakként: az ilyen ablakból át lehet lépni az alkalmazás másik ablakába.

Alkalmazás két ablakkal (modális ablak)

Készítsünk egy programot! Az alkalmazás két ablakot fog tartalmazni: egy fő ablakot és egy segédablakot, amely segítségével adatot viszünk be a programba. A segédablak modálisan lesz megjelenítve, tehát nem lehet majd belőle átkapcsolni az alkalmazás másik (fő) ablakába.

Alkalmazásunkban a következő komponenseket fogjuk használni: Label, 2 x Button (Form1-en) és Label, Edit, 2 x Button (Form2-n).

Az alkalmazás létrehozása után **(File – New – VCL Form Applications – Delphi for Win32)** az alkalmazásunknak egy ablaka (Forml) van. Mivel a mi alkalmazásunk két ablakot fog tartalmazni, a második ablakot külön be kell raknunk az alkalmazásba. Ehhez válasszuk a **File – New – Form – Delphi for Win32** menüpontot. Az alkalmazásba bekerül a Form2 ablak és a hozzá tartozó program modul is – Unit2.pas. Ahhoz, hogy az egyik modulból (unitból) tudjuk használni a másikat (tehát hogy az egyik modulból elérhetők legyenek a másikban levő komponensek), egy kicsit változtatnunk kell a forráskódokon (Unitl.pas, Unit2.pas) a következő képpen:

1. Az első modul (Uniti .pas) **uses** részét egészítsük ki a Unit2-vel:
   UsesWindows,Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,Dialogs,Unit2;
2. A második modulban keressük meg az **implementation** részt és ez alá írjuk be:
   uses Unit1;

Ezzel bebiztosítottuk, hogy mindkét modulban fogunk tudni dolgozni a másik modulban definiált komponensekkel.

A második ablakot a *Delphi* automatikusan létrehozza a program indításakor. Mi azonban most megmutatjuk, hogyan tudjuk kikapcsolni az ablak automatikus létrehozását, majd hogyan tudjuk a program futása alatt mi létrehozni a második ablakot (Form2).

Ehhez tehát előbb kapcsoljuk ki a Form2 automatikus létrehozását a program indításakor a következő képpen: nyissuk meg a **Project - Options** menüpontot. Ha nincs kiválasztva, válasszuk ki itt a **Forms** részt. Majd a Form2-t az **Auto-create**

forms listából helyezzük át az **Available forms** listába. A Form létrehozását a programban így akkor hajthatjuk végre, amikor szükségünk lesz rá. Ezt a nyomógomb megnyomásakor fogjuk megtenni:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Form2: TForm2;
    begin
    Form2 := TForm2.Create(Self);
    try
       if Form2.ShowModal = mrOk then
          Label1.Caption := Form2.Edit1.Text;
    finally
       Form2.Free;
       end;
  end;
```

Észrevehetjük, hogy az alkalmazás futása alatt ha megnyitjuk a második Form-ot, az alkalmazásból át lehet kapcsolni másik alkalmazásba, de ugyanennek az alkalmazásnak a főablakába nem tudunk visszamenni, amíg a modális ablakot nem zárjuk be.

Nézzük meg részletesen először a Uniti .pas forráskódját:

```
unit Unit1;
   procedure TForm1.Button1Click(Sender: TObject);
   var Form2: TForm2;
   begin
   Form2 := TForm2.Create(Self);
   try
         if Form2.ShowModal = mrOk then
       Label1.Caption := Form2.Edit1.Text;
       finally
          Form2.Free;
         end;
   end;
   procedure TForm1.Button2Click(Sender: TObject);
   begin
     Close;
```

```
end;
end.
```

A Unit2.pas forráskódja pedig így néz ki:

```
unit Unit2;
procedure TForm2.Button1Click(Sender: TObject);
begin
    ModalResult := mrOk;
end;
procedure TForm2.Button2Click(Sender: TObject);
begin
    ModalResult := mrCancel;
end;
end.
```

A megnyitott modális ablakot a **ModalResult** tulajdonság beállításával zárjuk be. A beállított értéket megkapjuk a Form2.ShowModal függvény hívásánál visszatérési értékként.

Ablakok, melyekből át lehet kapcsolni másik ablakokba (nem modális ablak)

Készítsünk el az előző alkalmazáshoz hasonló alkalmazást annyi különbséggel, hogy a segédablak nem modálisan lesz megjelenítve, tehát a felhasználó átkapcsolhat a főablakba a segédablak bezárása nélkül.

Példaprogram

Az eltérés az előző példától a modális és nem modális ablakok közötti különbségekből adódik. Míg a modális ablakot csak az ablakbezárásával hagyhatjuk el, a nem modális ablaknál ez nem igaz. Az ablakot ezért nem hozhatjuk létre ugyanúgy mint az előző példában: elsősorban azért nem, mert a felhasználó belőle átléphet az alkalmazás fő ablakába és megpróbálhatja újra létrehozni a segédablakot annak ellenére, hogy az már létezik. Ezért mielőtt létrehoznánk a segédablakot tesztelnünk kell, hogy az már létezik-e, és ha igen, akkor csak aktiválnunk kell. A másik „problémánk" az, hogy a **ShowModal** metódus meghívásánál a program „várakozik" addig, amíg a modális ablakot nem zárjuk be, míg a nem modális ablak létrehozásánál, a **Show** metódus meghívásánál a program fut tovább.

Ahhoz, hogy az ablak létezését bármikor tesztelhessük szükségünk lesz egy globális változóra (Form2). Ezt tehát a program elején kell deklarálnunk, nem valamelyik eljárásban.

Most megváltoztatjuk a második ablak megnyitására szolgáló eljárást:

```
procedure TForm1.Button1Click(Sender: TObject); begin
  if Form2 = nil then
    Form2 := TForm2.Create(Self); Form2.Show; end;
```

Megoldásra maradt még két problémánk: hogyan zárjuk be a második ablakot és hogyan biztosítsuk be a második ablakban megadott érték átadását. Mindkét tevékenységet a második ablakban (Form2) fogjuk végrehajtani. Ne felejtsük el, hogy a **Close** metódus a második ablaknál (Form2) csak az ablak elrejtését eredményezi, nem a felszabadítását a memóriából. A Form2 továbbra is a memóriában vanés

bármikor előhívhatjuk. Ha azt akarjuk, hogy a Form2 a bezárása után a memóriában ne foglalja a helyet, fel kell szabadítanunk.

Nézzük most meg a két forráskódot, először a Unit1-et:

```
unit Unit1;
var
Form1: TForm1; Form2: TForm2;
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
   if Form2 = nil then
   Form2 := TForm2.Create(Self); Form2.Show;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin  Close;
end;
end.
```

Majd nézzük meg a Unit2 modult is:

```
unit Unit2;
procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1.Label1.Caption := Edit1.Text;
  Close;
end;
procedure TForm2.Button2Click(Sender: TObject);
begin Close;
end;
procedure TForm2.FormClose(Sender: TObject;
var Action: TCloseAction);
begin
  Action := caFree;
end;
procedure TForm2.FormDestroy(Sender: TObject);
begin
  Form2 = nil;
end;
end.
```

A második Form nyomógombjainak **OnClick** eseményeibe a **Close** metódust használtuk, mellyel bezárjuk az ablakot.

Az ablak bezárásánál bekövetkezik az **OnClose** esemény. Ennek az eseménynek a kezelésében beállítottuk az **Action** paramétert **caFree**-re, ami az ablak felszabadítását eredményezi a memóriából.

A felszabadításkor bekövetkezik az **OnDestroy** esemény, melynek kezelésében

beállítottuk a Form2 mutató értékét nil-re.

A Form2 értékét az első unitban teszteljük, amikor megpróbáljuk létrehozni a második ablakot.

Kérdések

1. Az objektum orientált programozás.
2. A projekt fájlfelépítése.
3. Komponensek tulajdonságai.
4. Események.
5. Hibakeresés.
6. Jelölőnégyzet használata – CheckBox.
7. Választógomb – RádioButton.
8. Választógomb csoport – RadioGroup.
9. Egysorosszövegbevitelidoboz – Edit.
10. Többsoros szöveg beviteli doboz – Memo.
11. Görgetősáv – ScrollBar.
12. Szám bevitele – SpinEdit.
13. Listadoboz – ListBox.
14. Kombinált lista – ComboBox.
15. StringGrid komponens.
16. Időzítő – Timer.
17. Gauge, ProgressBar komponensek.
18. Kép használata – Image.
19. Választóvonal – Bevel.
20. Grafikus nyomógomb – BitBtn.
21. Eszköztár gomb – SpeedButton.
22. Kép lista – ImageList.
23. Eszköztár – ToolBar.
24. Könyvjelzők – TabControl, PageControl.
25. Formázható szövegdoboz – RichEdit.
26. Főmenü – MainMenu.
27. Lokális (popup) menü – PopupMenu.
28. Grafika, rajzolás, szöveg kiírása.
29. Hibák kezelése kivételek segítségével.
30. Fájltámogatás az ObjectPascal.
31. Fájltámogatás a Delphiben.
32. Standard dialógusablakok.
33. Több ablak (Form) használata.

Gyakorlati feladatok

I. Egy elágazó struktúrával rendelkezö folyamatábrát szerkeszen az összetett függvény értékéi kiszámítására, majd készítsen egy programot, amely az algoritmust realizálja (a függvény argumentum értékét a billentyűzetről kell megadni) [6], [7], [9], [10].

Változat	Függvény	Változat	Függvény
1	$y=\begin{cases} x-2 & x>2.5 \\ 1+x^2 & 0\le x\le 2.5 \\ x\ln\lvert\cos(x)\rvert & x<0 \end{cases}$	2	$y=\begin{cases} \sin(2.3x-1) & x>2.5 \\ 1-3\ln\lvert 1-x\rvert & 0<=x<=2.5 \\ \dfrac{x^2}{2-x} & x<0 \end{cases}$
3	$y=\begin{cases} \sqrt{(tg(x^2-1))} & x>1 \\ -2x & 0\le x\le 1 \\ e^{\cos(x)} & x<0 \end{cases}$	4	$y=\begin{cases} x^2-3+2.5x^2 & x>12.5 \\ e^x+5+\cos(0.001x) & 0\le x\le 12.5 \\ x^2 & x<0 \end{cases}$
5	$y=\begin{cases} 1+\sqrt{\lvert\cos(x)\rvert} & x>1 \\ x+1 & -0.5\le x\le 1 \\ 1-x^2 & x<-0.5 \end{cases}$	6	$y=\begin{cases} 2.5\cdot x^3+6\cdot x^2-30 & x>1.5 \\ x+1 & 0\le x\le 1.5 \\ x & x<0 \end{cases}$
7	$y=\begin{cases} 1+x & x>14.5 \\ e^{-x} & 3\le x\le 14.5 \\ \cos(x) & x<3 \end{cases}$	8	$y=\begin{cases} \ln\lvert 1+x\rvert & x>3.8 \\ e^{-x} & 2.8\le x\le 3.8 \\ \cos(x) & x<2.8 \end{cases}$
9	$y=\begin{cases} 1+\sqrt{\cos(x)} & x>4 \\ x+1 & 0\le x\le 4 \\ 1-x^2 & x<0 \end{cases}$	10	$y=\begin{cases} e^{-(x+8)} & x>3.61 \\ 1 & 0\le x\le 3.61 \\ \dfrac{x}{5} & x<0 \end{cases}$
11	$y=\begin{cases} x & x>1.5 \\ 2x^2\sqrt{\lvert\cos(2x)\rvert} & 0\le x\le 1.5 \\ e^{-\cos(3x)} & x<0 \end{cases}$	12	$y=\begin{cases} 1-\sqrt{\cos(2x)} & x>2.5 \\ x^2-x & 1\le x\le 2.5 \\ 1+x^2 & x<1 \end{cases}$
13	$y=\begin{cases} 2x & x>4.5 \\ 1-\ln\lvert 1-x^2\rvert & 0\le x\le 4.5 \\ e^{-x} & x<0 \end{cases}$	14	$y=\begin{cases} \sqrt{\ln(x^2-1)} & x>2 \\ -2x^3 & 0\le x\le 2 \\ e^{\sin(x)} & x<0 \end{cases}$
15	$y=\begin{cases} 1-\dfrac{2x^3}{1-x^2} & x>3.5 \\ \sqrt{\cos(2x-1)} & 0\le x\le 3.5 \\ e^{-\cos(2x)} & x<0 \end{cases}$	16	$y=\begin{cases} x+1 & x>2.5 \\ 1-x^5 & 0\le x\le 2.5 \\ x+\ln\lvert\sin(x)\rvert & x<0 \end{cases}$
17	$y=\begin{cases} x-2 & x>2.5 \\ 1+x^2 & 0\le x\le 2.5 \\ x\ln\lvert\cos(x)\rvert & x<0 \end{cases}$	18	$y=\begin{cases} 1+3x & x>4.5 \\ e^{-2x} & 1\le x\le 4.5 \\ \cos(2x) & x<1 \end{cases}$
19	$y=\begin{cases} \sqrt{\lvert tg(x^2-1)\rvert} & x>4 \\ -2x & 0\le x\le 4 \\ e^{\cos(x)} & x<0 \end{cases}$	20	$y=\begin{cases} e^{(x+2)} & x>1 \\ 1-2x & -1\le x\le 1 \\ -\dfrac{2x^3-3}{5} & x<-1 \end{cases}$

II. Hozzon létre egy folyamatábrát és egy programot az egyes feladatok végrehajtására a case operátor segítségével. Minden esetben gondoskodjon a forrásadatok helyességének ellenőrzéséről. Helytelen adatok beírásakor hibaüzenetet kell megjeleníteni [6] ,[7], [9], [10].

1. Hónapszámot adjuk meg (január 1. - január, 2. - február, ...). Nyomtassa ki a megfelelő évszak nevét ("tél", "tavasz" stb.).
2. Tekintettel a hónap (1 - január 2 - február, ...). Nyomtatás a napok száma az adott hónapban nem szökőév (azaz a február 28 nap).
3. Adott egy egész szám a 0 és 9 közötti tartományban van. Nyomtasson egy karakterláncot - a megfelelő szám nevének magyar nyelvű neve (0 - „nulla”, 1 - „egy”, 2 - „kettő”, ...).
4. Adott egy egész szám 1-től 5-ig terjed. Vezese ki - a megfelelő értékelés verbális leírása (1 - „rossz”, 2 - „nem kielégítő”, 3 - „kielégítő”, 4 - „jó”, 5 - „kiváló”).
5. A számok számtani műveletei a következőképpen vannak számozva: 1 - összeadás, 2 - kivonás, 3 - szorzás, 4 - osztás. Figyelembe véve a művelet számát és két számot A és B (A, B ≠0). Hajtsa végre a megadott műveletet a számokon és jelenítse meg az eredményt.
6. A hossz mértékegységei a következőképpen vannak számozva: 1 - deciméter, 2 - kilométer, 3 - méter, 4 - milliméter, 5 - centiméter. Tekintettel a hosszúság egységeinek számára és az L szegmens hosszára ezekben az egységekben (valós szám). Nyomtassa ki ennek a szegmensnek a hosszát méterben.
7. A tömegegységek a következőképpen vannak számozva: 1 kilogramm, 2 milligramm, 3 gramm, 4 tonna, 5 mázsa. Tekintettel a tömegegységek számára és az M testtömeg értékére ezekben az egységekben (valós szám). Nyomtassa ki az adott test tömegét kilogrammban.
8. Állítson össze egy olyan programot, amely a személy életkora szerint (egész számként beírva a billentyűzetről) meghatározza a korosztályhoz tartozó tartozást: 0-13 - fiú; 14-től 20-ig - fiatal férfi; 21-től 70-ig - férfi; 70 év felett - egy idős ember.
9. A lokátor egy adott pontra van irányítva („É” északra, „Ny” nyugatra, „D” délre, „K” keletre) és három parancs egyikét képes elvégezni: -1 - forduljon balra, 2 - forduljon jobbra, 3 - 180 fokos fordulat. A C karakter - a helymeghatározó kezdeti tájolása és az N szám - megadott parancs. A parancs futtatása után nyomtassa ki a lokátor tájolását.
10. A kör elemei a következőképpen vannak számozva: 1 - sugár (R), 2 - átmérő (D), 3 - hossz (L), 4 - kör terület (S). Figyelembe véve ezen elemek számát és jelentését. Nyomtassa ki a kör többi elemének értékeit (ugyanabban a sorrendben). A π értékhez használja a Pi állandót.
11. Adott egy egyenlő szárú derékszögű háromszög, amelynek számozva vannak a következő értékei: 1 - befogó (a), 2 - átfogója (c), 3 - magasság, az átfogóra

(h), 4 - a terület (S). Adva van ezen elemek egyike. Nyomtassa ki ennek a háromszögnek a többi elemét (ugyanabban a sorrendben).

12. Két egész számot adunk: D (nap) és M (hónap), amelyek meghatározzák a nem szökő év helyes dátumát. Nyomtassa ki a megadott dátumot megelőző dátum D és M értékeit (például megadva D = 1 M = 1, kimenet D = 31 M = 12; megadva D = 1 M = 3, kimenet D = 28 M = 2; megadva D = 15) M = 12-t kell levonni D = 14 M = 12).
13. Két egész számot adunk: D (nap) és M (hónap), amelyek meghatározzák a nem szökő év helyes dátumát. Nyomtassa ki a megadott időpontot követő dátum D és M értékeit (például megadva D = 1 M = 1, kimenet D = 2 M = 1; megadva D = 31 M = 12, kimenet D = 1 M = 1; megadott D = 28 M = 2 le kell vonni D = 1 M = 3).
14. Adott egy egész szám a 20-69 tartományban, hogy meghatározzák a életkor (években). Nyomtasson egy sort - a megadott életkor szóbeli leírását, biztosítva a szám helyes összehangolását az „év” szóval, például: 20 - „húsz év”, 33 - „harminchárom év”, 41 - „negyven egy év”.
15. Egy egész szám a 100 és 999 közötti tartományban van. Nyomtasson egy sort - ennek a számnak a verbális leírása, például: 256 - "kétszázötven hat", 804 - "nyolcszáznégy".
16. Készítsen egy programot, hogy meghatározza a hónapokban levő napok számát, ha megadva van: N-hónapszám - egész szám 1-től 12-ig, A egész szám 1-ha a szökőév és 0 egyébként.
17. Hozzon létre egy programot, amely kiszámítja a geometriai ábra területét. Az ábra típusát a (c) szimbólum határozza meg: O - kör, T - egyenlő szárú derékszögű háromszög és K - négyzet. A szimbólum után beírt egész szám meghatározza a megfelelő elemet a terület kiszámításához (a körnek a sugára, a háromszögnek a befogó hossza, egy négyzetnek az oldal hossza).
18. Állitson össze olyan programot, amely a hónap sorszáma alapján meghatározza, mely évszakhoz tartozik.
19. Állitson össze olyan programot, amely kinyomtatja az évfolyamot a félév száma szerint, amelyhez a megadott félév tartozik (1. és 2. félév - 1 év, 3. és 4. félév - 2 év, stb.).
20. Adva van egy n – egész szám, amely megfelel a geometriai ábra szögeinek. Készítsen egy programot, amely az megadott n szám szerint kinyomtatja az ábra nevét (például n = 3 - háromszög , n = 5 - ötszög, n> 8 - sokszög). Ha 2-nél kisebb számot ír be, hibaüzenet jelenyen meg.

III. Egy elágazó struktúrával rendelkezö folyamatábrát szerkeszen az összetett függvény értékei kiszámítására, majd készítsen egy programot, amely az algoritmust realizálja (a függvény argumentum értékét az alábi táblázatböl vegye) [6] ,[7], [9], [10].

Valtozato	Függvény	Az argumentum tartománya	Lépték
1	$y = \begin{cases} x - 2 & x > 2.5 \\ 1 + x^2 & 0 \le x \le 2.5 \\ x \ln\lvert\cos(x)\rvert & x < 0 \end{cases}$	[-π; π]	π/10
2	$y = \begin{cases} \sin(2.3x - 1) & x > 2.5 \\ 1 - 3\ln\lvert 1 - x\rvert & 0 <= x <= 2.5 \\ \frac{x^2}{2 - x} & x < 0 \end{cases}$	[-π/5; 9π/5]	π/3
3	$y = \begin{cases} \sqrt{(tg(x^2 - 1))} & x > 1 \\ -2x & 0 \le x \le 1 \\ e^{\cos(x)} & x < 0 \end{cases}$	[-1; 1.5]	0.5
4	$y = \begin{cases} x^2 - 3 + 2.5x^2 & x>12.5 \\ e^x + 5 + \cos(0.001x) & 0 \le x \le 12.5 \\ x^2 & x < 0 \end{cases}$	[-5; 10]	0.55
5	$y = \begin{cases} 1 + \sqrt{\lvert\cos(x)\rvert} & x >1 \\ x + 1 & -0.5 \le x \le 1 \\ 1 - x^2 & x < -0.5 \end{cases}$	[-1.5; 1.5]	0.25
6	$y = \begin{cases} 2.5 \cdot x^3 + 6 \cdot x^2 - 30 & x > 1.5 \\ x + 1 & 0 \le x \le 1.5 \\ x & x < 0 \end{cases}$	[-2; 3]	0.5
7	$y = \begin{cases} 1 + x & x > 14.5 \\ e^{-x} & 3 \le x \le 14.5 \\ \cos(x) & x <3 \end{cases}$	[-1; 15]	1
8	$y = \begin{cases} \ln\lvert 1 + x\rvert & x > 3.8 \\ e^{-x} & 2.8 \le x \le 3.8 \\ \cos(x) & x < 2.8 \end{cases}$	[0; 5]	0.5
9	$y = \begin{cases} 1 + \sqrt{\cos(x)} & x > 4 \\ x + 1 & 0 \le x \le 4 \\ 1 - x^2 & x < 0 \end{cases}$	[-1;4.5]	0.25

10	$y = \begin{cases} e^{-(x+8)} & x > 3.61 \\ 1 & 0 \le x \le 3.61 \\ \frac{x}{5} & x < 0 \end{cases}$	[-π; 2π]	π/5
11	$y = \begin{cases} x & x > 1.5 \\ 2x^2\sqrt{\lvert\cos(2x)\rvert} & 0 \le x \le 1.5 \\ e^{-\cos(3x)} & x < 0 \end{cases}$	[-1; 3]	0.5
12	$y = \begin{cases} 1-\sqrt{\cos(2x)} & x > 2.5 \\ x^2 - x & 1 \le x \le 2.5 \\ 1+x^2 & x < 1 \end{cases}$	[0; 3]	0.3
13	$y = \begin{cases} 2x & x > 4.5 \\ 1-\ln\lvert 1-x^2\rvert & 0 \le x \le 4.5 \\ e^{-x} & x < 0 \end{cases}$	[-0.5; 5]	0.5
14	$y = \begin{cases} \sqrt{\ln(x^2-1)} & x > 2 \\ -2x^3 & 0 \le x \le 2 \\ e^{\sin(x)} & x < 0 \end{cases}$	[-π/2; π]	π/5
15	$y = \begin{cases} 1-\frac{2x^3}{1-x^2} & x > 3.5 \\ \sqrt{\cos(2x-1)} & 0 \le x \le 3.5 \\ e^{-\cos(2x)} & x < 0 \end{cases}$	[-0.5; 4.5]	0.25
16	$y = \begin{cases} x+1 & x > 2.5 \\ 1-x^5 & 0 \le x \le 2.5 \\ x+\ln\lvert\sin(x)\rvert & x < 0 \end{cases}$	[-π; 2π]	π/5
17	$y = \begin{cases} x-2 & x > 2.5 \\ 1+x^2 & 0 \le x \le 2.5 \\ x\ln\lvert\cos(x)\rvert & x < 0 \end{cases}$	[-π/2; 2π]	π/4
18	$y = \begin{cases} 1+3x & x > 4.5 \\ e^{-2x} & 1 \le x \le 4.5 \\ \cos(2x) & x < 1 \end{cases}$	[-π/2; 2π]	π/5
19	$y = \begin{cases} \sqrt{\lvert tg(x^2-1)\rvert} & x > 4 \\ -2x & 0 \le x \le 4 \\ e^{\cos(x)} & x < 0 \end{cases}$	[-2; 5]	0.5
20	$y = \begin{cases} e^{(x+2)} & x > 1 \\ 1-2x & -1 \le x \le 1 \\ -\frac{2x^3-3}{5} & x < -1 \end{cases}$	[0.8; 2.5]	0.1

IV. Hozzon létre egy programot, amely egész számokat tartalmazó egydimenziós tömbök adatait dolgozza fel. A feldolgozás során használja az elemek permutációit új tömbök létrehozása nélkül. A forrás tömb kitöltése a véletlen számok generátorával végezze. A forrás és a feldolgozott tömböt jelenítse meg [6] ,[7], [9], [10].

1. Korrigálja az A = (a_1, a_2, ..., a_n) tömböt úgy, hogy a tömb elejére írja azt a csoportot, amely a legtöbb egymást követő pozitív elemet tartalmazza. A tömb elemeit a billentyűzetről adja be.
2. Az A = (a_1, a_2, ..., a_n) tömb minden nullával egyenlő elemét helyezze át közvetlenül a tömb maximális eleme után. A tömb elemeit a billentyűzetről adja be.
3. Az A = (a_1, a_2, ..., a_n) tömb minden negatív elemét helyezze át a tömb végébe.
4. Az A = (a_1, a_2, ..., a_n) tömbben távolítsa el a pozitív elemek utolsó csoportját. A csoporton ugyanazon jelű egymást követő elemeinek értünk, amelyek száma legalább 2 vagy több.
5. Az A = (a_1, a_2, ..., a_n) tömbben helyezze át azokat a pozitív elemeket, amelyek a minimális pozitív elem előtt találhatók, a tömb végébe.
6. Az A = (a_1, a_2, ..., a_n) tömbben távolítsa el az egymást követő negatív elemeket amelyek a tömb minimális eleme után találhatók.
7. Az A = (a_1, a_2, ..., a_n) tömbben távolítsa el az összes negatív elemet amelyek a tömb minimális eleme előtt találhatók.
8. Az A = (a_1, a_2, ..., a_n) tömbben távolítson el minden elemet, amely kisebb, mint a tömb azon eleme amely maximálistól balra található.
9. 9.Az A = (a_1, a_2, ..., a_n) tömbbe illesszen be egy új elemet a tömb negatív elemei közül a legnagyobb után.
10. Az A = (a_1, a_2, ..., a_n) tömbben távolítson el minden elemet a minimális pozitív és a maximális negatív elem között.
11. Az A = (a_1, a_2, ..., a_n) tömbben távolítson el minden pozitív elemet, amelyek páros sorszámmal rendelkeznek és a tömb minimális elemét követik.
12. Az A = (a_1, a_2, ..., a_n) tömbben az összes pozitív elemet, a második pozitívtól kezdve, helyezze át a tömb végébe.
13. Az A = (a_1, a_2, ..., a_n) egydimenziós tömbben cserélje ki a legnagyobb számú egyenlő elemet tartalmazó csoportot ennek a tömbnek a maximális elemére. A beállítás után a tömb kevesebb elemet tartalmazhat, mint korábban. Írja be a tömb elemeit a billentyűzetről.
14. Az A = (a_1, a_2, ..., a_n) egydimenziós tömbben helyezze át az elemek azon csoportját, amelyek a tömb legtöbb egymást követő negatív elemét tartalmazza a tömb végébe. Írja be a tömb elemeit a billentyűzetről.

15. Az A = (a_1, a_2, ..., a_n) egydimenziós tömbben helyezze át az összes negatív elemet, amelynek páratlan sorszámmal rendelkeznek a tömb végébe, vagyis tegye az utolsó elemek helyére.
16. Egydimenziós tömbben A = (a_1, a_2, ..., a_n) cserélje ki az összes elemcsoportot, amely több, mint 3 egymást követő negatív elemet tartalmaz a maximális elemmel. Írja be a tömb elemeit a billentyűzetről.
17. Egy egydimenziós tömbben A = (a_1, a_2, ..., a_n) minden pozitív elemet amely páros sorozatszámmal rendelkezik helyezze át a tömb elejére.
18. Az A = (a_1, a_2, ..., a_n) egydimenziós tömbben cserélje ki a legnagyobb számú egyenlő elemet tartalmazó csoportot ennek a tömbnek a maximális elemére. Írja be a tömb elemeit a billentyűzetről.
19. Az A = (a_1, a_2, ..., a_n) egydimenziós tömbben távolítsa el az összes negatív elemet, amelyek a pozitív elemek között helyezkednek el.
20. Egydimenziós A = (a_1, a_2, ..., a_n) tömbből távolítsa el azt a csoportot, amelyben a legtöbb egymást követő pozitív elem van. Írja be a tömb elemeit a billentyűzetről.

V. Hozzon létre egy programot, amely egész számokat tartalmazó mátrixok adatait dolgozza fel. A feldolgozás során használja az elemek permutációit új tömbök létrehozása nélkül. A forrás tömb kitöltése a véletlen számok generátorával végezze. A forrás és a feldolgozott tömböt jelenítse meg [6] ,[7], [9], [10].

1. Adott egy A($n \times (n + 1)$) mátrix és két egydimenziós tömb X = ($x_1, ..., x_{n+1}$) és Y = ($y_1, ..., y_{n+1}$), valamint a két természetes szám p és q. Hozzon létre egy új ($n + 1) \times (n + 2$) méretű mátrixot úgy, hogy a p számú sor után beilleszt egy új sort az A mátrixba $x_1, x_2, ..., x_{n+1}$ elemekkel, majd beilleszt a q számú oszlop után egy új oszlopot $y_1, y_2, ..., y_{n+1}$ elemekkel.
2. Adott A = ($a_1, a_2, ..., a_{10}$) tömb és a B($n \times n$) mátrix. Cserélje ki nullákra a mátrix azon elemeit, amelyek számára az indexek összege páros és megegyeznek az A tömb elemeivel.
3. Adott A = ($a_1, a_2, ..., a_{10}$) tömb és a B($n \times n$) mátrix. A mátrix első sorának elemei növekvő sorrendben vannak rendezve. Szerkesszen egy új, $n \times (n + 1)$ méretű mátrixot az A tömb elemeinek új oszlopának beillesztésével az eredeti mátrixba úgy, hogy a mátrix első sorának rendezése maragyon meg.
4. Adott egy A($n \times m$) mátrix. Szerkesszen egy új mátrixot, amelyet a következő kapuk: az oszlopok átrendezésével: az első az utolsó, a második az utolsó előtti stb.
5. Adott egy A($n \times m$) mátrix, valamint p és q egész számot. Alakítsa át az A mátrixot úgy, hogy az eredeti p sorszámmal rendelkező sor kövesse az eredeti q sorszámút, megőrizve a fennmaradó sorok sorrendjét.
6. Adott egy A($n \times n$) mátrix. Keresse meg és nyomtassa ki azt a mátrix azon sorát, amely a legtöbb páros számot tartalmazza
7. Adott egy A($n \times m$) mátrix. Rendezze át a mátrix sorait következő képen: a sorok átrendezésével - az első az utolsóval, a második az utolsó előttivel stb. A mátrix megengedett transzformációját két sor és két oszlop permutációjának nevezzük.
8. Adott egy n rendű négyzetmátrix. Engedélyezett transzformációk segítségével érje le, hogy a mátrix legkisebb abszolút értékel rendelkező eleme a mátrix jobb alsó sarkában kerüljön.
9. Adott egy A($n \times m$) mátrix. Alakítsa át a mátrixot oly módon, hogy törölje azt a sort és oszlopot, amelyeknek a metszéspontjában a legnagyobb abszolút értékű elem található.
10. Adott egy A($n \times n$) mátrix, amelynek nincsenek megegyező elemei. Keresse meg a legnagyobb elemet a fő- és az oldalátlón, és cserélje meg őket.
11. Alakítson ki egy egy dimenziós tömböt azokból a negatív elemekből amelyek abban a sorban találhatók ahol az A($n \times n$) mátrix legkisebb eleme.
12. Alakítson ki egy egy dimenziós tömböt azokból a pozitív elemekből amelyek abban a sorban találhatók ahol az A($n \times n$) mátrix legnagyobb eleme.

13. Az A ($2 \times n$) mátrix segítségével n pontot definiálunk a síkban úgy, hogy $a_{1,j}$, $a_{2,j}$ a j pont koordinátái. A pontokat szegmensek páronként kötik össze. Keresse meg a legnagyobb szegmens hosszát, és jelenítse meg annak koordinátáit.
14. Adott egy A($n \times n$) mátrix. Számítsa ki az $x_1x_n + x_2x_{n-1} + ... + x_nx_1$ értéket, ahol x_k a mátrix k oszlopának legnagyobb értéke.
15. Adott egy A($n \times n$) mátrix. Keresse meg a sor- és oszlopszámot, amelyek metszéspontjában van egy mátrix legnagyobb eleme található.
16. Távolítsa el az A($m \times n$) mátrixból a legnagyobb mennyiségű nullát tartalmazó sort.
17. Adott egy ($n \times m$) mátrix. Javítsa ki ezt a mátrixot azáltal, hogy törli a sort és az oszlopot, amelyeknek a metszéspontjában található a legmagasabb abszolút értékű elem.
18. Adott egy ($n \times n$) mátrix. Szerkesszen egy egydimenziós tömböt e mátrix pozitív elemeiről, amelyek a fő átló felett találhatók.
19. Adott egy ($n \times n$) mátrix. Szerkesszen egy egydimenziós tömböt e mátrix negatív elemeiről, amelyek a fő átló alatt találhatók.
20. Adott egy egész számokat tartalmazó ($n \times n$) mátrix. Keresse meg a mátrix minden sorában a legkisebb elemeket, és számítsa ki a páros számok mennyiségét közöttük.

Pót feladatok

1. Számok bekérése, sorrendben való kiíratása.
2. Osztás adott pontossággal.
3. Maximumelem kiválasztás tömbből.
4. Buborékrendezés.
5. Lotto – legtöbbször kisorsolt elem.
6. Fájlkezelés – a rejtö.txt fájl tartalmát átmásolni egy másik fájlba.
7. Menürajzolás.
8. Vonal, kör, rúd kirajzoltatása.
9. Egy pontba érintkező koncentrikus körök kirajzolása.
10. Mátrix szorzása (tükrözése).
11. Bekért személyi adatok fájlba mentése.
12. Alakzatok kirajzolása bekért paraméterek szerint.
13. Írjunk programot, amely beolvas két természetes számot, majd kiírja a két szám hányadosát és maradékát az alábbi formában. A program az adatok beolvasása után hagyjon ki egy üres sort.
14. Készítsünk programot, amely bekér egy egész számot, majd mindaddig kér be további egész számokat, amíg nem adjuk meg a 0-t. A program határozza meg és írja ki a beadott egész számok közül a legnagyobbat.
15. Készítsünk programot, amely ki fogja kérdezni a matematikát (két szám összeadását, kivonását és szorzását az <1,10> intervallumból). A két számot és a műveletet a számítógép véletlenszerűen válassza ki. A program akkor fejeződjön be, ha a felhasználó 10 példát kiszámolt helyesen. Rossz válasz esetén kérdezze újra ugyanazt a példát. A program végén írjuk ki az eredményességet százalékokban.
16. Készítsünk programot, amely bekér egész számokat mindaddig, amíg nem adjuk meg a 0-t. A program határozza meg és írja ki a beadott egész számok közül a legkisebbet és a legnagyobbat. (A 0-t ne számítsa bele a beadott számokba, ez csak a bevitel végét jelzi.) A számok beolvasását a 0 végjelig repeat .. until ciklus segítségével valósítsuk meg!
17. Olvassunk be egész számokat 0 végjelig egy maximum 100 elemű tömbbe (a tömböt 100 eleműre deklaráljuk, de csak az elejéből használjunk annyi elemet, amennyit a felhasználó a nulla végjelig beír). A beolvasás után írjuk ki a számokat a beolvasás sorrendjében majd fordítva.
18. Olvassunk be egész számokat 0 végjelig egy maximum 100 elemű tömbbe (a tömböt 100 eleműre deklaráljuk, de csak az elejéből használjunk annyi elemet, amennyit a felhasználó a nulla végjelig beír).
    - Írjuk ki a számokat a beolvasás sorrendjében.
    - Írjuk ki az elemek közül a legkisebbet és a legnagyobbat, tömbindexükkel együtt.
    - Írjuk ki az elemeket fordított sorrendben.

19. Olvassunk be egész számokat egy 20 elemű tömbbe, majd kérjünk be egy egész számot. Keressük meg a tömbben az első ilyen egész számot, majd írjuk ki a tömbindexét. Ha a tömbben nincs ilyen szám, írjuk ki, hogy a beolvasott szám nincs a tömbben.
20. Állítsunk elő egy 50 elemű tömböt véletlen egész számokból (0-tól 90-ig terjedő számok legyenek).
    - Írjuk ki a kigenerált tömböt a képernyőre.
    - Számítsuk ki az elemek összegét és számtani középértékét.
    - Olvassunk be egy 0 és 9 közötti egész számot, majd határozzuk meg, hogy a tömbben ez a szám hányszor fordul elő.
21. Olvassunk be egy N egész számot (1<=N<=10), majd egy NxN-es kétdimenziós tömbbe generáljunk véletlen egész számokat 10-tol 99-ig. (A tömböt 10x10-esre deklaráljuk, de ennek csak az NxN-es részét használjuk.)
22. Keressünk olyan p prímeket, amelyekre a) p+10 b) p+14 c) p^2+2 is prímszám!
23. Mersenne-prímnek nevezzük a 2^p-1 alakú prímszámot, ha p prím. Keressünk ilyen alakú összetett számot!
24. Fermat-féle számoknak szokás nevezni a 2^{2n+}1 alakú számokat. Keressük meg közülük a prímszámokat!
25. Négyesiker prímeknek nevezzük azokat a számokat, amelyekre p, p+2, p+6 és p+8 is prímszám. Keressünk adott intervallumban ilyen négyesikreket! Miért nem szerepel a sorban a p+4?
26. Keressük meg az összes olyan n természetes számot, amelyekre az n+1, n+3, n+7, n+9, n+13 és n+15 számok mindegyike prímszám! Ha az utolsó feltételt elhagyjuk, akkor találunk-e további számot?
27. Keressünk olyan n természetes számot, amelyre n^2-3 osztható 1-nél nagyobb négyzetszámmal!
28. Keressünk olyan természetes számot, ami után legalább 12 összetett szám következik!
29. Keressünk olyan természetes számpárokat, amelyeknek az euklideszi algoritmusuk 2,3,... lépésből áll!
30. Keressünk olyan természetes számpárt, amelynek legkisebb közös többszöröse egy adott d számmal nagyobb a legnagyobb közös osztónál! Milyen szám lehet a d?
31. Keressük meg adott számig a legtöbb osztójú természetes számot!
32. Erősen összetett számnak nevezzük azokat a természetes számokat, amelyek osztói száma több, mint bármely náluk kisebb természetes szám osztói száma. Készítsünk adott N-ig erősen összetett számot kereső programot!
33. Határozzuk meg adott intervallumban, hogy a számok hány százaléka esetében kisebb a valódi osztók összege a számnál!
34. Suryanarayana nevezte el nagyon tökéletesnek azt a természetes számot, amelyre teljesül, hogy az osztóinak összegének osztóinak összege a szám kétszerese. Keressünk ilyen számokat!

35. Egy számot k-szorosan tökéletesnek nevezünk, ha a nála kisebb pozitív osztóinak összege a szám k-szorosa. Keressünk 2, 3, 4, 5-szörösen tökéletes számokat adott intervallumban! (Ez ideig nem találtak 7-nél nagyobb tökéletességű számot!)

Tárgymutató

Irodalom

1. http://zeus.nyf.hu/~akos/pascal/pasframe.htm
2. http://prog.ide.sk/pas.php
3. http://progizas123.atw.hu/programozas.htm
4. http://ep128.hu/Ep_Konyv/Szabvanyos_Pascal.htm
5. https://studfile.net/preview/412168/
6. Angster Erzsébet: **Programozás tankönyv I.- II**.Akadémia nyomda, Martonvásár, 1999.
7. Baga Edit: **Delphi másképp** Akadémia nyomda, Martonvásár, 1999.
8. Benkő Tiborné: **Programozási feladatok és algoritmusok Delphi rendszerben** Computer Books, Budapest, 2002.
9. Ray Lischner: **Delphi kézikönyv** Kossuth Kiadó, 2001.
10. Paul Kimmel: **Delphi 6 fejlesztők kézikönyve** Panem Kft. , Budapest, 2002.
11. Dr. Fercsik János: **Programozás – Delphi** Dunaújvárosi Főiskola - jegyzet, 2001.
12. Marco Cantú: **Delphi 7 mesteri szinten I. –II.** Kiskapu Kft, 2003.

Гарнітура Times New Roman. Папір офсетний.
Формат видання 60х84/16.
Умовн. друк. арк. 3,72 Наклад:100. Зам. №00.
Видавництво ПП «АУТДОР – ШАРК»
88000, м. Ужгород, Україна
пл. Жупанатська, 15/1.

Свідоцтво про внесення суб'єкта видавничої справи
до державного реєстру видавців, виготовників
і розповсюджувачів видавничої продукції.
Серія 3т № 40 від 29 жовтня 2012 року.