Perl programozási nyelv
munkapéldány

Dr. Schuster György

2013. május 9.

Tartalomjegyzék

1. Bevezetés (Olvasd el, hasznos!)

A Perl (Practical Extraction and Report Language) nyelvet a kilencvenes évek elején Larry Wall fejlesztette ki kimondottan rendszeradminisztrációs feladatokra. A UNIX és a UNIX jellegű rendszerek [1] számos eseménynaplót készítenek, amelyek elengedhetetlenek a rendszer követéséhez.
Larry célja az volt, hogy ezek a naponta keletkező rendszerbejegyzések követhetőek legyenek. Bár a rendszer jelentős segítséget nyújt és számos jól használható szűrőt biztosít a rendszergazda számára, de a shell programozása nehézkes és rugalmatlan.
A shell másik problémája az, hogy tisztán interpreteres végrehajtású, ami azt jelenti, hogy a végrehajtás során az értelmező sorról-sorra végigolvassa a sorokat, értelmezi őket és végrehajtja.
Ebben az időben már legalább tíz éve rendelkezésre állt a C programozási nyelv, ami természetesen alkalmas ilyen jellegű feladatok ellátására.
A C problémája az, hogy túlságosan aprólékos, ezáltal egy jellegzetes shell szűrési feladatot csak nagyon sok programozással tudunk megoldani. Nagy előnye azonban, hogy compiler-es jellegű nyelv, tehát a forrásszövget a futtatás előtt lefordítják bináris állományra és ezt bináris állományt futtatják.
A perl egyesíti a két módszer jó tulajdonságait, biztosítja a magasszintű szűrés lehetőségét és hatékony, jól használható programozási struktúrákat ad a programozó kezébe. Ráadásul a perl nem interpreteres futtatású, hanem egy átmeneti megoldást használ. A programot forrásban tárolja, de a futtatás előtt lefordítja bináris állománnyá és ezt a bináris állományt futtatja, tehát nem soronként értelmez. Ezzel két problémát is megold, nevezetesen:

1. relatíve gyorsan fut,
2. a későn jelentkező szintaktikai hibák problémáját kikerüli. [2].

Ejtsünk néhány szót a perl általános jellemzőiről!
A perl a C jellegű programnyelvek családjába tartozik, ilyen jellegű még a JAVA, a JAVASCRIPT, a PHP és a PIKE (továbbá számos más nyelv). Ez azt jelenti, hogy az operátorok és utasítások nagy része megegyezik a C operátoraival és utasításaival és — ami nagyon lényeges —, hogy különbséget tesz a kis és nagybetűk között.
Természetesen a perl számos más szolgáltatást is nyújt.
A C nyelvről a szakirodalom azt állítja, hogy egy rendkívül jól írható nyelv, de az olvashatósága nehézkes, ez a kifejezés a perl nyelvre fokozottan érvényes.
Larry Wall ezt a nyelvet nem tudományos alapokon fejlesztette — hasonlóan Kernigham-hoz és Ritchie-hez — , hanem a UNIX szűrőkből kiválogatta a számára tetsző, illetőleg hasznos komponenseket. Ennek egyenes következménye, hogy a nyelv nem teljesen konzekvens, és néha nehezen követhető, de![3] Az is ennek az elvnek a következménye, hogy egy feladatot legalább kétféleképpen lehet megoldani.
A nehéz olvashatóság kiküszöbölésére célszerű a programot geometriailag jól tagolni és kellő sűrűséggel ellátni megjegyzésekkel, hogy a megírást követő héten is képesek legyünk megérteni, mit csináltunk.
A jegyzet felépítése a következő:

[1] Az ilyen jellegű operációs rendszereket UN*X rendszereknek nevezik.
[2] A későn jelentkező szintaktikai hibák problémája azt jelenti, hogy a program futása során akár órák múlva jelentkeznek szintaktikai hibák, mivel az adott programrészre akkor kerül rá a vezérlés. A compiler-es megoldások esetén ez már fordítás időben kiderül.
[3] A perlben mindig lesz ”de”.

- az első négy fejezet a perl alapjaival foglalkozik, illetőleg a hibakeresés elősegítő debug üzemmódot ismerteti,
- az ötödik fejezet röviden a perl objektum orientált használatát mutatja be,

2. Szintaktikai elemek

A perl programozási nyelv, mint minden programozási nyelv szintaktikai elemekből áll, ezek a következők:

- változótípusok,
- operátorok,
- utasítások,
- szubrutinok,
- reguláris kifejezések,
- referenciák
- csomagok és modulok.

Sajnálatos módon ezeket a szintaktikai elemeket nem lehet egymástól teljesen függetlenül kezelni, így időnként sajnos még ismeretlen anyagra is hivatkozni kell. A jegyzetben ezt úgy hidaljuk át, hogy az adott helyen egy rövid magyarázatot adunk és hivatkozunk arra az oldalra, ahol a részletes leírás található.

2.1. Változótípusok

A perl öt változótípust különböztet meg, ezek:

- skalár, jele `$`,
- tömb, jele `@`,
- asszociációs tömb, másnéven hash, jele %,
- szubrutin, jele `&`,
- globális, jele `*`.

Megjegyzés: A perl nyelvben a változókat nem kell, de lehet előre deklarálni. Ez részben megkönnyíti a programozó dolgát, azomban a kész program öndokumentálását nem segíti elő.

2.1.1. Skalár

A **skalár** változótípus általános, egy egységként kezelhető adatelemek tárolására szolgál. Ez azt jelenti, hogy tárolhatunk benne numerikus, szöveges és változócím jellegű értékeket.
A következő programrészlet a skalár kezelését mutatja be.

```
1  $a=5;                  # Az $a skalar erteke 5 lesz
2  $a="Szoveges ertek\n"; # Az $a skalar erteke a szoveg es egy soremeles
3  $a='Szoveges ertek\n'; # Az $a skalar erteke a szoveg es egy \n
```

A fenti programrészlet 2. és 3. sora csak annyiban különbözik, hogy a szövegrészletet a 2. sorban idézőjlek veszik körül, míg a 3. sorban aposztrofok. A második sorban a lévő szöveget egy soremelés karakter, a harmadik sorban lévő szöveget a \ és az n karakter zárja le.
Ha tehát a 2. sorban definiált `$a` változót iratjuk ki a képernyőre a következő képet látjuk:

```
Szoveges ertek
_
```

A 3. sor kiiratása esetén a képernyő tartalma:

```
Szoveges ertek\n_
```

A _karakter a cursort jelképezi.
A kiiratást a `print $a;` kifejezéssel tehetjük meg.

2.1.2. Tömb

A **tömb** adattípus arra szolgál, hogy benne skalárokat tároljunk. A tömb adattípust a @ jel jelöli. A tömböt a skalárhoz hasonlóan nem kell deklarálni, aminek az a következménye, hogy a tömb mérete a program futása során tetszőlegesen változtatható.
Ha a tömb n elemet tartalmaz, akkor az indexelés tartománya $0, \ldots, n-1$. Tehát egy tetszőleges n elemű `@t` tömb esetén az első elem a `$t[0]`, az utolsó elem a `$t[n-1]`.
A következő program egy tömb inicializálását és listázását mutatja be.

```
1  @a=("Humpty","Dumpty","sat","on","the","wall");
2  foreach $k (@a)
3     {
4     print "$k\n";
5     }
```

Az 1. sorban az `@a` tömböt inicializáljuk. Az egyenlőségjel jobboldalán a `()` operátorpár között adjuk meg a tömb elemeit. A példában a szövegrészleteket adunk meg.
A 2. sorban egy perl utasítást láthatunk, amely a tömböt végigolvassa, és az egyes tömbelemeket sorra a `@k` változóba teszi, amit a `print "$k\n";` kifejezéssel kiiratjuk a képernyőre. A `foreach` leírását a 23. oldalon olvashatjuk.
A mennyiben az `@a` tömb 3. elemét szeretnénk elérni pl. kiiratni, akkor `print $a[3];` kifejezést kell alkalmaznunk.
Figyeljük meg, hogy egy tömbelemet skalárként iratunk ki.
A tömb egyetlen `print` utasítással is kiírható, azonban ilyenkor a kiiratás tagolatlan. A megfelelő kifejezés `print @a;`. Az előző példa tömbjét így kiiratva a képernyőn a következőt láthatjuk:

```
HumptyDumptysatonthewall_
```

Mivel a tömböt a program elején nem deklaráljuk ezért a méretét sem tudjuk mindig, azonban lehetőség van a tömb utolsó elemének indexének lekérdezésére:

```
print $#a;
```

A kiiratott érték skalár, ezért van a kifejezés elején a $ karakter. A tömb elemeinek száma $#a+1.
A tömb elemeinek száma kiírható a következő módon is:

```
$n=@t;
print $n;
```

A tömb a futás során az indexek segítségével bővíthető, a bővítésnek nem kell folytonosnak lenni. Ez azt jelenti, hogy a példánkban az utolsó index értéke 5 és minden további nélkül adhatunk értéket a tömb 7. elemének (nem feleddjük0-tól számolunk), anélkül, hogy a 6. elemet inicializáltuk volna. Természetesen a 6. elemet később még feltölthetjük.
Egy feltöltött tömböt nagyon egyszerűen újra skalárokra bonthatunk a következő módon.

```
1  @t=(1,2,3,4,5);          # A tomb inicializalasa
2  ($a,$b,$c,$d,$e)=@t;     # A skalarokra bontas
```

A példa 1. sorában a @t tömböt értékekkel töltjük fel. A második sorban a ezt tömböt a kifejezés baloldalán, a zárójelben található skalárokba bontjuk szét. Tehát az $a értéke 1, $b értéke 2, $c értéke 3, $d értéke 4 és $e értéke 5 lesz [4].
Ha a balodalon kevesebb skalár van, mint a tömb elemeinek száma, akkor a baloldali változók feltöltése a tömb elejéről történik. A fennmaradó elemek értéke üres sztring lesz.
Érdekesség, hogy a perl ismeri a negatív indexeket. A negatív indexek hátulról indexelik a tömböt, tehát ha az előző példában látható @t tömböt nézzük, akkor a következő megfeleltetést tehetjük:
$t[4] ≡ $t[-1]
$t[3] ≡ $t[-2]
$t[2] ≡ $t[-3]
$t[1] ≡ $t[-4]
$t[0] ≡ $t[-5]

2.1.3. Aszociációs tömb

Az aszociációs tömb, vagy másnéven hash tömb (röviden hash), egy egyszerű táblázatok jellegű adatszerkezet létrehozására szolgál olymódon, hogy kulcs—érték párokat hoz létre, ahol a kulcs és az érték egy—egy skalár.

```
%h=(1,"egy",2,"ketto",3,"harom",4,"negy");
```

A ¸ h hash kulcsai az 1, 2, 3 és a 4, a kulcsokhoz tarozó értékek az egy, a ketto, a harom és a negy.
Egy érték elérése a kulcs segítségével (esetünkben legyen ez a ketto) a $h{2} kifejezéssel történik.
A hash feltöltés nagy hash esetében áttekinthetetlen. Ezért létrehoztak egy jobban átlátható inicializálást.

```
%h=(  1 => "egy",
      2 => "ketto",
      3 => "harom",
      4 => "negy" );
```

[4] Első pillanatra nagyon furcsa, a kifejezés olyan, mintha megsértené a balérték szabályt.

A két módszer teljesen egyenértékű, de a második sokkal jobban áttekinthető (és kommentezhető!).
A hash természetesen utólag tetszőlegesen bővíthető. Az előző példánkban szereplő hash-hez az alábbi módon fűzhetük egy új táblázatelemet:

```
$h{5}="ot";
```

Ekkor van egy új elemük, amelyre az 5 értékkel hivatkozhatunk.

2.1.4. Szubrutin

A szubrutinok alkalmazása más programozási nyelvekhez hasonlóan a perl nyelvben is lehetséges, azonban a perl a szubrutint is adattípusnak tekinti. A szubrutin adattípus jelölése a & jellel történik.
A szubrutinok szerkezetéről és alkalmazásáról a 28. oldalon találhatunk leírást.

2.1.5. Globális

A globális adattípus jele a `*`. Ez a típus mára már elavult, a perl korábbi verzióiban arra használták, hogy tömböket adjanak át szubrutinoknak.
A globális adattípus tulajdonképpen minden adattípust képvisel, tehát átadhatunk neki skalárt, tömböt és hash-t.
A perl 5.0 verziója felett a globális adattípus helyett a referenciákat használják. A refernciákról a 34. oldalon lehet olvasni.

2.2. Operátorok

Az operátorok a kifejezésekben a műveleti jeleknek felenek meg. Egyes irodalmi feldolgozások a gyári függvények egy részét is operátornak nevezik. Ebben a jegyzetben azokat a kifejezéseket, amelyek névvel rendelkeznek nem soroljuk az operátorok közé, ez alól csak néhány kivételt teszük.

Definíció: Egy operátort **unárisnak** nevezzük, ha az általa kijelölt műveletnek csak egyetlen operandusa van.

Definíció: Egy operátor **bináris** vagy **binér**, ha az általa kijelölt műveletben két operandus szerepel.

A perl nyelvben a következő operátor csoportok vannak:

1. elválasztó operátorok,
2. aritmetikai operátorok,
3. sztring, tömb operátorok,
4. relációs operátorok,
5. logikai operátorok,
6. értékadó operátorok,
7. bit operátorok,

8. címképző és indirekciós operátorok,

9. a reguláris kifejezés operátorai.

2.2.1. Elválasztó operátorok

Az elválasztó operátorok a következők:

```
{}  []  ()  ,  =>  ;
```

{} operátorpár az un. blokk operátorok. Egy logikailag összetartozó utasításcsoportot foghatunk segítségével össze. Ez lehet egy ciklus törzse, egy igaz vagy hamis ág (Ezt az operátorpárt sokkal többet használjuk, mint a C nyelvben, mert ez minden blokkot kezelő utasítás esetén is kötelező.). Nézzük a következő példát!

```
1  if($a == 5)
2     {
3     print "Ot";
4     }
```

A fenti programrészlet első sorában egy `if` utasítás található. Ha az `if` argumentumában szereplő kifejezés igaz, akkor a {} operátorok közötti részt hajtja végre a program. Az `if` utasítás leírása a 19. oldalon található.

`()` operátorpár két szerepet tölthet be, ezek[5]:

1. A matematikai kifejezések végrehajtási sorrendjét változtatja meg.
2. A tömbök és hash-ek inicializálásánal használjuk, lásd a 2.1.2. és a 2.1.3. fejezetet. A `()` operátorpárt a tömb→skalár átalakításnál is használjuk.

`,` operátor szintén több szerepet tölt be. Ezek:

1. Inicializálásnál az elemek elválasztását végzi. Például:

   ```
   @a=(1,2,3,4,5);
   ```

2. Kifejezéseket választ el, ekkor azonban több lehetőség lehetséges.

   (a) `$a=(5,6);`
   Az eredmény 6.

   (b) `$a=5,6;`
   Az eredmény 5.

   A 2a példa eredménye azért különbözik a 2b eredményétől, mert a zárójel egy egységgé fogja össze a kifejezést, míg a második esetben két különálló kifejezésünk van, valahogy így:

   ```
   $a=5;
   6;
   ```

[5]Néhány utasítás szintakszisához (helyesírásához) hozzátartozik a zárójelpár, ilyen például az `if`. Ekkor a `()` nem operátor.

=> operátor a vessző operátor szinonímája. A hash tömböknél szoktuk használni (lásd 9. oldalon). A program jobban olvasható lesz.

; operátor a kifejezések lezárására szolgál. Minden kifejezést ezzel zárunk le. Használata kötelező.

2.2.2. Aritmetikai operátorok

A perl aritmetikai operátorai a következők:

```
+  -  *  /  %  **  -  ++  --
```

+ - * / operátorok hagyományos aritmetikai műveleteket jelölnek ki.

% operátor a modulóképzés operátora. Ezzel a művlettel egy egész osztás maradékát kaphatjuk meg, lásd a következő programrészletet!

```
$a=29;
$b=3;
print $a%$b;
```

A programrészlet a 2-es értéket írja ki.

****** operátor a hatványozás operátora.

```
$a=2;
$b=3;
print $a**$b;
```

A kiírt érték 8 ($2^3 = 8$), tehát a balodali operandus a hatvány alapja, a jobboldali a kitevő. Az oprátor segítségével tört és negatív hatványokat is lehet képezni.

- operátor az egy operandusú vagy unáris minusz. Egy számot tartalmazó skalár változó előjelét változtatja meg.

```
$a=5;
$b=-$a;
```

A `$a` értéke természetesen marad 5, `$b` értéke -5 lesz.

++ inkrementáló operátor. Az operátor egy számot tartalmazó skalár értékét növeli egy értékkel. Az operátor használható a változó előtt és után is. A különbség a két módszer között a következő két programrészlet alapján látható.

1. ```
   $a=5;
   $b=$a++;
   print "$b";
   ```
   A kiírt éték 5.
   ```

2. ```
   $a=5;
   $b=++$a;
   print "$b";
   ```
   A kiírt éték 6.

A fenti ”jelenség” magyarázata az 1. példára a következő:

- A program fut, megtalálja az értékadó operátort (=).
- Milyen értéket adjon át? Megtalálja az `$a` változót és értékét átadja a `$b`-nek.
- Fut tovább, megtalálja a `++` operátort. Mit kell inkrementálni? Utána nem áll semmi (nem is állhat), tehát visszalép és `$a`-t növeli.

A 2. példa magyarázata:

- A program fut, megtalálja az értékadó operátort (=).
- Megtalálja a `++` operátort. Ez operátor, tehát az értéke nem adható át. A program tovább fut.
- Megtalálja a `$a`-t és inkrementálja.
- Visszalép és az inkrementált értéket átadja `$b`-nek.

-- operátor a dekrementáló operátor, az adott változó értékét egyel csökkenti. Minden másban működése megegyezik a A **++** operátoréval.

2.2.3. Sztring, tömb operátorok

Ez a fejezet a szakírók köreiben vitát válthat ki, mert néhány — általunk beépített függvényként kezelt — szubrutin hívást mások operátoroknak tekintenek. Az általunk ebbe a csoportba sorolt operátorok:

```
.  x  ..  [ , ]
```

. operátor sztring elemek összefűzésére szolgál.

```
$a="Hello ";
$b="vilag";
$c=$a.$b."\n";
print $c;
```

A programrészlet a következő kiírást jeleníti meg a képernyőn.

```
Hello vilag
_
```

x operátor az un. sokszorozó operátor, amely egy sztringrészletet vagy egy tömbelemet sokszoroz meg. Az első program a sztringelem sokszorozását, a második a tömbelem sokszorozását mutatja be.

1. ```
   $a="ha"x5;
   print $a;
   ```
   A képernyőn `hahahahaha` jelenik meg.

2.
```
@vigyor=("ha")x5;
foreach $k (@vigyor)
   {
   print $k." ";
   }
```

A `@vigyor` tömbben ötször szerepel a `ha` elem. Így a `foreach` (leírás a 23. oldalon található) utasítás által kilistázott tömb a képernyőn a következő: `ha ha ha ha ha`.

`..` az un. tartomány operátor. Használata tömbök kezdeti inicializálásánál célszerű.

```
@t=(1..10);
foreach $k (@t)
   {
   print $k." ";
   }
```

A képernyőn a `1 2 3 4 5 6 7 8 9 10` kiírás jelenik meg.

Az operátor használható szöveges inicializálásra is.

```
@t=(a..f);        # vagy @t=("a".."f");
foreach $k (@t)
   {
   print $k." ";
   }
```

A képernyőn `a b c d e f` látható.

Az operátor bal oldali operandusa numerikusan kisebb kell, hogy legyen, mint a jobb oldali. Ha szöveges értékekkel dolgozunk a bal oldali sztringnek előbb kell lennie ABC sorrendben, mint jobboldalinal, továbbá a bal oldali sztring hossza kisebb, vagy egyenlő a jobb oldaliéval.

`[ , ]` az úgynevezett tömbszelet operátor[6] ennek segítségével mintegy kötegelve tudjuk a tömb elemit kezelni.

```
@t=(0..9);
print @t[2,3];
```

A képernyőn `23` jelenik meg. Nem csak két elemű tömb szeletet lehet előállítani, hanem nagyobbat is.

```
@t=(0..9);
print @t[2,6,9];
```

A kiirás `269`. Ez az operátor nagyon jól használaható a tömb egyes elemeinek felcserélésénél.

[6] Ez kicsit több, mint egy operátor, inkább kifejezés.

```
@t=(0..9);
@t[2,6]=@t[6,2];
print @t;
```

A képernyőn `0163452789` jelenik meg, tehát a 2. és a 6. elem felcserélődött (Ne feledjük, az indexelés 0-tól kezdődik.).

2.2.4. Relációs operátorok

A relációs operátorok skalár változók összehasonlítására szolgálnak. Minden olyan kifejezés, amelynek értéke 0 vagy ures sztring, az hamisnak minősül, ellenkező esetben a kifejezés igaz.
A relációs operátorok:

`<  <=  ==  !=  >=  >  <=>  lt  le  eq  ne  ge  gt  cmp`

Ebbe az operátor csoportba sorolhatók a fájlteszt operátorok is, ezekről a 47. oldalon beszélünk.
Az előző felsorolásból láthatjuk, hogy két jól megkülönböztethető operátor csoport van. A matematikai jelek a numerikus, a szöveges operátorok a sztringek összehasonlítását végzik.

<	`lt`	kisebb ,
<=	`le`	kisebb vagy egyenlő ,
==	`eq`	egyenlő ,
!=	`ne`	nem egyenlő
>=	`ge`	nagyobb vagy egyenlő,
>	`gt`	nagyobb,
<=>	`cmp`	lexikografikus összehasonlító operátorok .

Az első oszlopban látható operátorok a numerikus, a második oszlopban lévő operátorok a szöveges összehasonlítást végzik.

<=> **`cmp`** operátorok a két változót úgy hasonlítják össze, hogy az összehasonlítás eredménye számszerű. Ha a bal oldali skalár kisebb, mint a jobboldali az eredmény -1, ha egyenlő, akkor 0, ha nagyobb az eredmény 1.

A perl érdekes tulajdonsága, hogy két skalár összehasonlításának eredménye függhet attól, hogy numerikusan vagy szövegesen hasonlítjuk őket egymáshoz.

```
$a=2;
$b=10;
if($a<$b)
   {
   print "Kisebb\n";
   }
else
   {
   print "Nagyobb\n";
   }
```

```
$a=2;
$b=10;
if($a lt $b)
   {
   print "Kisebb\n";
   }
else
   {
   print "Nagyobb\n";
   }
```

A bal oldali programrészlet kiírja a `Kisebb` feliratot. A jobb oldali programrészlet kiírja a `Nagyobb` szöveget.

A magyarázat nagyon egyszerű, az nyilvánvaló, hogy a 2 kisebb, mint a 10, azonban a 10 ABC sorrendben előbb van, mint a 2.

2.2.5. Logikai operátorok

A logikai operátorok relációs jellegű kifejezések összekapcsolására szolgálnak, ha összetett feltételeket akarunk alkalmazni.
Az operátorok:

`&&  and  ||  or  !  not`

`&& and` operátorok két relációs kifejezés és kapcsolatát hozza létre. A két operátor működése azonos, azonban a `&&` precedenciája[7] magasabb.

Ha egy összetett logikai kifejezést készítünk az és operátor segítségével, akkor a végrehajtás során a perl csak addig dolgozza fel a kifejezést, amíg az eredmény nem biztos.

```
... 1.kifejezes && 2.kifejezes && 3.kifejezes ...
```

Ha az `1.kifejezes` hamis, akkor már a többit kifejezést nem vizsgálja, mivel az eredmény már biztosan hamis. Ha `1.kifejezes` igaz, akkor megvizsgálja `2.kifejezes`-t. Ha `2.kifejezes` hamis, a feldolgozás befejeződik, ha igaz, akkor a `3.kifejezes`-t is feldolgozza.

A hagyományos magasszintű programozási nyelvekben a logikai operátorokat csak ciklus és elágazás utasításokban használjuk, azonban a perl-ben egymást követő, logikai értékkel visszatérő kifejezések összefűzésére használhatjuk.
Példa erre az a perl szkript, amely ezt a jegyzetet fordítja `.tex` állományból `.ps` állományra.

```
system("latex perl.tex") && system("latex perl.tex") && dvips -f perl.dvi >perl.ps;
```

Ha az első fordítás igaz, akkor elindítja a másodikat, ha nem kilep. Ha a második igaz, akkor elindítja a `.dvi`→`.ps` konverziót.

|| **`or`** két relációs jellegű kifejezés között vagy kapcsolatot hoz létre. Erra az operátorra is igaz, hogy egy összetett kifejezést addig hajt végre, amíg az eredmény nem biztos.

```
... 1.kifejezes || 2.kifejezes || 3.kifejezes ...
```

Ha az `1.kifejezes` igaz a többi már nem kerül feldolgozásra.

`xor` két kifejezés között kizáró vagy kapcsolatot hoz létre.

`! not` egy kifejezés logikai értékét negálja.

[7] A precedencia az operátorok végrehajtási sorrendjét jelenti, ha egy operátor precedenciája magas, akkor előbb hajtódik végre, lásd a szorzás és az összeadás példáját.

2.2.6. Értékadó operátorok

Az operátorcsopot már maga meghatározza a feladatkörét, segítségükkel egy vagy több változónak értéket lehet adni.
Az operátorok:
`= ?:` rekurzív csoport

= operátor a klasszikus értékadó operátor a bal odalon lévő kifejezésnek adja át a jobb oldalon lévő kifejezés által szolgáltatott értéket.

?: az úgynevezett feltételes értékadó utasítás, amely három kifejezésből áll.

```
bal oldali kifejezes = 1.kifejezes ? 2.kifejez\'es : 3.kifejezes
```

Az `1.kifejezes` kiértékelése relációs jellegű. Ha igaznak minősül a `2.kifejezés` által szolgáltatott érték lesz az átadott, ha hamisnak a `3.kifejezés` által szolgáltatott érték lesz az átadott érték. Így lehet például a nagyobbat kiválasztani két változó közül.

```
$c=($a>$b)?$a:$b;
```

rekurzív csoport Ez az operátorcsoport minden bináris operátor esetén használható, ahol `$a=$a + $b;` jellegű, tehát rekurzív értékadás történik. Ekkor írhatjuk, hogy `$a+=$b;`.

Néhány példa.

```
$a=$a+$b      $a+=$b
$a=$a-$b      $a-=$b
$a=$a&&$b     $a&&=$b
```

2.2.7. Bit operátorok

Ez az operátor csoport kimondottan bináris jellegű adatok kezelésére szolgál, skalárok közötti bit szintű logikai műveleteket ír elő.
Az operátorok:

`~ & | ^ << >>`

~ unáris bitenkénti invertálás operátor. Segítségével egy numerikus skalár minden bitjét negáljuk. A következő példával meghatározható, hogy hány biten ábrázol a perl egy egész numerikus skalárt.

```
$a=0;
print ~$a;
```

A kiírt érték 4294967295. Ezt hármas csoportokra bontva $4\ 294\ 967\ 295$, ami $2^{32} - 1$, ez 32 bitet jelent.

& két numerikus skalár bitenkénti és kapcsolatát képezi. Például (a ”fölöslges” bitek elhagyásával).

```
$a=5;           ... 0101
$b=6;         & ... 0110
$c=$a&$b;       ... 0100
```

| két numerikus skalár bitenkénti vagy kapcsolatát képezi.

^ két numerikus skalár bitenkénti kizáró vagy kapcsolatát képezi.

<< Egy numerikus egész értékű skalárt adott számban balra léptet. Az első operandus a léptetett változó, a második operandus a lépésszámot adja meg.

```
$a=1;
$b=$a<<3;
print $b;
```

A kiírt érték 8. Ha egy léptetett bit eléri a legfelső bitet, a következő lépésre belép alul a legalsó biten[8].

>> Egy numerikus egész értékű skalárt adott számban jobbra léptet. Az első operandus a léptetett változó, a második operandus a lépésszámot adja meg.

Ha egy léptetett bit eléri a legalsó bitpozíciót, akkor a következő lépésre elvész.

2.2.8. Címképző és indirekciós operátorok

A jegyzet elején említettük, hogy a perl az úgynevezett C jellegű nyelvek közé tartozik. A C-nek nagyon nagy erőssége a mutatók kezelése a perl is tud mutatókat (perl terminológiával referenciákat) kezelni, amelyek részletes ismertetése a 34. oldalon található.
A címképző és indirekciós operátorok a következők:

```
\  $  ->
```

címképző operátor. Egy változó címét képezi le és tölti egy skalárba. A változó bármilyen típusú lehet, skalár, tömb, hash, szubrutin és globális.

$ indirekciós operátor. Egy skalárban tárolt címen található változó értékéhez biztosít hozzáférést.

```
1  $a=5;
2  $b=\$a;
3  $c=$$b;
```

Az első sorban az `$a` skalárnak adunk értéket. A második sorban a `$b` skalárba töltjük az `$a` címét. A harmadik sorban a `$c` skalár értéke a `$b` skalár által címzett memóriarekesz értéke lesz, tehát az `$a` értéke.

Az első `$` jelenti azt, hogy skalárral van dolgunk, a második az indirekciós operátor. Ha nem skalárral dolgozunk, hanem például tömbbel, akkor a következőképpen kell eljárnunk.

[8] Szóval ez attól függ, már találkoztunk olyan perl-el, ahol a bit csak kilépett.

```
1 @a=(1..5);
2 $b=\@a;
3 @c=@$b;
```

A második sorban a `$b`-ben az `@a` tömb címe kerül. A harmadik sorban a `@c` tömböt a `$b`-ben tárolt című tömbbel töltjük fel. Figyeljük meg, hogy a típusazonosító elöl van.

Részletesebb leírást a referenciáknál találhatunk a 34. oldalon.

$->$ indirekt mezőhozzáférés operátor. Esetünkben (még nem tartunk az objektum orientált programozásnál) egy tömb címét tartalmazó skálár segítségével érhetjük el a címzett tömb egy elemét. Lássuk a példát!

```
@a=(1..5);
$b=\@a;
print $b->[2];
```

A programrészlet 3-at ír ki.

2.2.9. Reguláris kifejezések operátorai

A reguláris kifejezéseket később tárgyaljuk a 30. oldalon. A jegyzet ezen pontján nem lenne értelme az ide tartozó operátorokat ismertetni.

2.3. Utasítások

A perl utasításai a következő két csoportba sorolhatók:

1. elágazás utasítások,
2. ciklus utasítások.

2.3.1. Elágazás utasítások

Ezek az utasítások a programok futását módosítják különböző feltételektől függően (kivétel a `return`). Az csoport utasításai:

`if unless else elsif return goto`

`if` a klasszikus elágazás utasítás. Ha az argumentumában lévő kifejezés igaz az program az `if` utáni blokkra ugrik.

```
1 if(kifejezes)
2    {
3    igaz ag
4    }
```

Az 1. sorban található kifejezés igaz, akkor a {} operátorok közötti terület, jelen esetben a 3. sor hajtódik végre, ellenkező esetben a program ezt a területet kihagyja. Lásd még `else`!

A {} operátorok alkamazása kötelező! Ellentétben a C nyelvvel, ahol csak több sor esetén kellene használni.

Az `if` használható az úgynevezett ”egyszer” utasítás formátumban is, ekkor egyetlen kifejezést lehet feltételessé tenni úgy, hogy a feltéetelt a az utasítás után írjuk.

```
print "Hello\n" if($a eq "igaz");
```

Ha `$a` egyenlő az `"igaz"`-zal a `hello` felirat megjelenik a képernyőn.

`unless` utasítás pont ellentétesen viselkedik, mint az előzőekben látott `if`, ha az utasítás argumentumában található kifejezés hamis, akkor ugrik a program az utasítást követő blokkra. Lásd a következő példát.

```
1  unless(kifejezes)
2     {
3     hamis ag
4     }
```

A 2. és 4. sorban található {} operátorpár alkalmazása kötelező.

Az `unless` is használható ”egyszer” utasításként.

```
print "Hello\n" unless($a eq "hamis");
```

Ha `$a` nem egyenlő a hamissal a `hello` felirat megjelenik a képernyőn.

`else` a klasszikus ”másik ág” utasítás. Csak az `if` és az `unless` esetleg `elsif` utasítás után használható utasítás. Ha az elágazást `if` vagy `unless` után a program nem a következő blokkra ugrik, akkor lehetőség van az `else` használatára, de nem kötelező. Az `else`-nek közvetlenül kell az előző blokkot követnie, köztük más utasítás nem lehet.

```
1  if(kifejezes)          unless(kifejezes)
2     {                      {
3     igaz ag                hamis ag
4     }                      }
5  else                   else
6     {                      {
7     hamis ag               igaz ag
8     }                      }
```

`elsif` egy `else { if(){...} }` szerkezetet vált ki. Programrészlet az alkalmazására.

```
1  if($a<5)
2     {
3     print "Kisebb, mint 5\n";
4     }
5  elsif($a<8)
6     {
7     print "Kisebb, mint 8, de nagyobb, mint 4\n";
8     }
```

Az 1. sorban található `if` csak az 5-nél kisebb értékeket ”engedi át”, ha `$a` nagyobb, mint négy, akkor a vezérlés az 5. sorra kerül, itt egy újabb vizsgálat történik, aminek eredményeként az vagy az 6.—8. sorok közti blokkra kerül a vezérlés, vagy a program fut tovább.

`return` értékvisszaadó utasítás szubrutinok esetén. Részletes leírása 25. oldalon található.

`goto` feltétel nélküli ugrás utasítás. Az utasítás az őt követő cimkére ugratja a program futását.

```
CIMKE:  ...
          ⋮
        goto CIMKE;
```

A cimkének mindig betűvel kell kezdődnie, ami lehet nagy és kisbetű. A cimkét `:` karakter zárja le, ami nem része a cimkének.

A `goto` utasítás a programozás nagyjai szerint kerülendő, mert nagyon áttekinthetetlenné válik a kód. Véleményem szerint akkor használjuk, ha már más megoldás nincs.

Sajnos a perl nyelv nem ismeri a C-ben széleskörben alkalmazott `switch|case` szerkezetet.

2.3.2. Ciklus utasítások

A perl ebben is számos utasítást kínál a programozónak. Az utasítások:

`while until do for foreach last next redo`

`while` klasszikus elöltesztelő ciklus utasítás. A ciklus addig fut, amíg az utasítás argumnetumában található kifejezés igaz.

```
1  while(kifejezes)
2     {
3     ciklustorzs
4     }
```

A 2. és a 4. sorban látható {} operátorpár itt is kötelező, hasonlóan az elágazás utasításaihoz.

A `while` utasítás használható ”egyszer” utasításként [9]

[9] Hmm, elég furcsa elnevezés egy ciklusnál

```
$a=0;
print $a++."\n" while($a < 10);
```

A ciklus kiírja a számokat 0—9-ig.

Figyelem ez nem hátultesztelő ciklus, ha a ciklusfeltétel nem igaz, akkor egyszer sem fut le az utasítás.

until a `while` utasítás ellentéte. Az utasítás ugyanúgy elöltesztelő ciklus létrehozására szolgál, azomban a ciklus addig fut, amíg az utasítás argumentumában lévő kifejezés hamis. Használata:

```
until(kifejezes)
   {
   ciklustorzs
   }
```

Az `until` is használható egyszer utasításként. Írjuk át a `while` példáját `until`-ra!

```
$a=0;
print $a++."\n" until($a == 10);
```

do segéd utasítás hátultesztelő ciklusok létrehozására, illetve egyes esetekben blokkvégrehajtást ír elő.

```
do                      do
   {                       {
   ciklustorzs             ciklustorzs
   }                       }
while(kifejezes);       until(kifejezes);
```

Ha a `while` utasítást használjuk, akkor a ciklus addig fut, amíg a kifejezés igaz. Ha az `until` a ciklus lezáró utasítás, akkor a ciklus addig fut, amíg a kifejezés hamis.

A blokkvégrehajtást előíró szerepére az egyik legjobb példa a `switch` utasítás szimulálása.

```
1  SWITCH: {
2          $a eq "egy"   && do { $b=1; last SWITCH; };
3          $a eq "ketto" && do { $b=2; last SWITCH; };
4          $a eq "harom" && do { $b=3; last SWITCH; };
5          $a eq "negy"  && do { $b=4; last SWITCH; };
6          $b=-1;
7          }
```

A fenti programrészlet 1. sorában a `SWITCH:` cimkét láthatjuk az megnevezi az egész szerkezetet (a cimkékről a `goto` utasításnál a 21. oldalon olvashatunk részletesen.).

A 2.—5. sorokban az esetek kezelése történik. Vizsgáljuk a 2. sort.

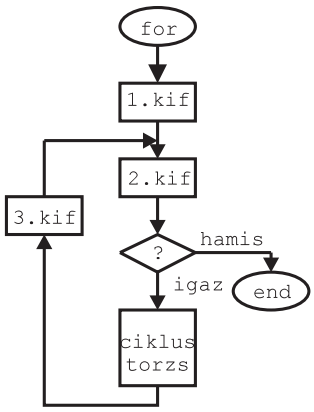
```
$a eq "egy"   && do { $b=1; last SWITCH; };
```

Ha a `$a eq "egy"` kifjezés igaz, akkor az && operátor a sor további feldolgozását írja elő. A `do` utasítás viszont az blokkot hajtja végre. A `last SWITCH;` hatására a program elhagyja az egész szerkezetet.

Ha a `last SWITCH;` kifejezést elhagyjuk a `$b` értéke -1 lesz. Ennek oka az, hogy a program folytatja a szerkezet végrehajtását, tehát sorra veszi a sorokat, ahol viszont a feltételek `$a` szerint hamisak. Az egyetlen olyan sor, amelyben nincs feltétel a 6., tehát itt a `$b` értéket kap.

`for` klasszikus számláló ciklus utasítás. Az utasítás argumentuma három különálló részből áll. Ezek:

- Az első kifejezés csak egyszer fut le, ezért kiváló a ciklus kezdeti értékadására.
- A második kifejezés relációs jellegű, ha igaz, akkor, a ciklus fut, ellenkező esetben a ciklus végetér.
- A harmadik kifejezés a ciklus törzs lefutása után hajtódik végre.



2.2. ábra: A `for` folyamatábrája

A formátuma:

```
for(1.kif ; 2.kif ; 3.kif)
  {
  ciklustorzs
  }
```

Az argumentumban lévő kifejezéseket `;` karakterekkel választjuk el.

A `for` három kifejezésébe bármit be lehet írni — hasonlóan a C nyelvhez — semmilyen megkötés nincs.

`foreach` tömbolvasó ciklus. Egy megadott tömb elemit olvassa végig. Használatára példa:

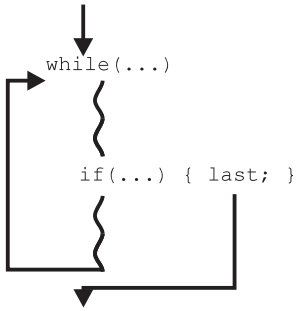
```
foreach $k (@t)
  {
  print $k;
  }
```

A ciklus a `@t` elemeit olvassa végig és ezeket sorra a `$k` változóba teszi, amit a `print` kilistáz.

A skalár használata nem kötelező, ekkor az aktuális tömbelemet az úgynevezett default változóba teszi, amelynek neve `$_` (részletes leírást lásd **??**). Így az előző példa `$k` nélkül:

```
foreach (@t)
  {
  print $_;
  }
```

`last` ciklus elhagyó utasítás. Ha a program a futása során egy `last` utasítással találkozik, az éppen aktuális ciklust elhagyja.
Ha a program két (vagy több) egymásba ágyazott ciklus belső ciklusában talál `last` utasítást, akkor csak a belső ciklusból ugrik ki, nem az összessből.
Ha azonban valamelyik ciklus fej elé cimkét teszünk és az utasításban erre a cimkére hivatkozunk, akkor a megcímzett ciklusból lép ki a program.

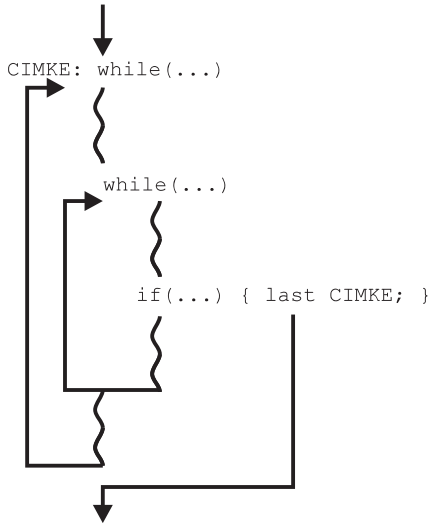


2.3. ábra: A `last` futási gráfja

Nézzük a következő példát!

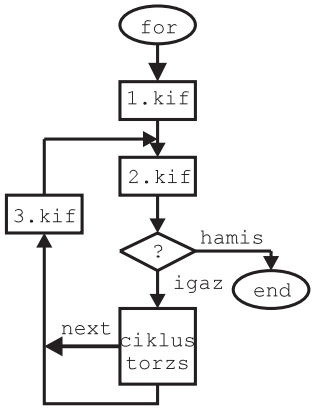
```
1  CIMKE: while(1)
2           {
3           while(1)
4             {
5             $a=<>;chomp($a);
6             if($a eq "q") { last CIMKE; }
7             }
8           }
```

Az 1. sorban a `while` elé betettük a `CIMKE:` cimkét, majd 3. sorban egy újabb végtelen ciklust hoztunk létre, amelyben a 5. sorban a konzolról olvasunk be értéket (lásd fájlkezelés 41. oldalon.). Ha a 6. sorban lévő `if` úgy dönt, hogy a `$a`-ban egy ”q” karakter van, akkor köszönhetően a `last CIMKE;` utasításnak, a program kilép mindkét ciklusból.
A `last` elletétben a 2.3. és 2.4. ábrával nem csak a `while` utasítás esetén használható.



2.4. ábra: A `last` futási gráfja cimkével

next a ciklus újrakezdését előíró utasítás. Ha a progra a futása során egy `next` utasítást talál a ciklust elölről kezdi, úgy, hogy a feltételeket megvizsgálja.
Hátultesztelő ciklusok esetén nem alkalmazható.
A `next` előírja, hogy a ciklust a program elölről kezdje. Kérdés azonban az, hogy mi történik a `for` utasítás 3. kifejezésével, végrehajtásra kerül, vagy nem?
A 2.5 ábrán ez követhető. A **3. kifejezés végrehajtásra kerül!**



2.5. ábra: A `for` esetén a `next`

A `next` esetén is használhatók a címkék. Egymásbaágyazott ciklusok esetén, ha megcimkézünk egy ciklust és a program valamelyik belső ciklusban talál egy `next` utasítást cimkével, akkor nem az aktuális ciklus kezdi elölről, hanem a megcikézettet. Példa:

```
CIMKE: while(1)
         {
         print ">";
         while(1)
           {
           print ":";
           $a=<>;chomp($a);
           if($a eq "n") { next CIMKE; }
           }
         }
```

Ha a ”n”-t ütünk be, akkor a program nem csak a ”:”-ot írja ki, hanem a ”>” karakter is.

redo ciklus újrakezdését előíró utasítás, a ciklus feltétel vizsgálata nélkül. Ha a program a `redo` utasításra fut, akkor a ciklus törzs végrehajtását elölről kezdi anélkül, hogy figyelembe venné a ciklus fejben előírt feltételeket.

Hasonlóan a `last` és a `next` utasításokhoz a `redo` esetén is lehet cimkéket használni. Ekkor a megcímzett ciklust kezdi elölről a program.

2.4. Szubrutinok, típusokhoz rendelt függvények

Ennek a fejezetnek az első felében az adattípusokhoz rendelt függvények ismertetése történik, a második részben az általunk írt szubrutinok tulajdonságai, és a változók hatásköri kérdéseit tárgyaljuk.
A skalároknak, a szubrutinoknak és a globális változóknak nincsennek olyan szorosan a típushoz tartozó függvényei, mint a tömböknek és a hash-eknek.

2.4.1. Tömb függvények

push A kérdéses tömb végére egy skalárt illeszt. Használata:

```
push(@t,$a);
```

Ha a hívás előtt a `@t` tartalma `("a","b","c")` és a `$a` értéke `"k"`, akkor a `push` hívása után a `@t` `("a","b","c","k")`.

pop A kérdéses tömb utolsó elemét kiveszi a tömbből és egy skalárban adja vissza. Használata:

```
$a=pop(@t);
```

Ha a hívás előtt a `@t` tartalma `("a","b","c","k")`, akkor a `pop` hívása után a `@t` `("a","b","c")` és a `$a` skalár értéke `"k"`.

shift A kérdéses tömb első elemét kiveszi a tömbből és egy skalárban adja vissza. Használata:

```
$a=shift(@t);
```

Ha a hívás előtt a `@t` tartalma `("a","b","c")` értéke `"k"`, akkor a `pop` hívása után a `@t` `("b","c")` és a `$a` skalár értéke `"a"`.

unshift A kérdéses tömb elejére egy skalárt illeszt. Használata:

```
unshift(@t,$a);
```

Ha a hívás előtt a `@t` tartalma `("a","b","c")` és a `$a` értéke `"k"`, akkor a `unshift` hívása után a `@t` `("k","a","b","c")`.

Megjegyzés: A `push-pop` és a `unshift-shift` függvénypárokkal verem (stack) adatszerkezetet lehet létrehozni. Ha a A `push-shift` és a `unshift-pop` utasításpárokat használjuk, akkor csővezetéket (fifo-t) készíthetünk.

qw az úgynevezett `q` függvények egyike. A függvény argumentumába beírt és ”whitespace”[10] karakterekkel elválasztott sztringeket egy tömbe helyezi. Használata:

```
@t=qw(Tojas Tobias a falra ult);
```

A `$t[0]` értéke `Tojas`, a `$t[1]` értéke `Tobias` lesz, és így tovább.

2.4.2. Hash függvényei

keys A kérdéses hash kulcsait egy tömbbe helyezi. Példa a használatára:

```
%h=(1=>'a',2=>'b',3=>'c');
@t=keys(%h);
foreach (@t)
  {
  print "$_ ";
  }
```

[10] Szóköz és tabulátor.

A programrészlet kiírja a képernyőre a kulcsokat `1 2 3`.

A hash elemeinek számát is meg lehet határozni a `keys` segítségével.

```
@t=keys(%h);
print $#t;
```

`values` A kérdéses hash ”értékeit” teszi egy tömbbe. Példa a használatára:

```
%h=(1=>’a’,2=>’b’,3=>’c’);
@t=values(%h);
foreach (@t)
  {
  print "$_ ";
  }
```

A programrészlet kiírja a képernyőre az értékeket `a b c`.

`each` A kérdéses hash értékeit és kulcsait egyenként, de egymás után beteszi két valtozóba. Példa a használatra:

```
1  %h=(1=>’a’,2=>’b’,3=>’c’);
2  ($a,$b)=each(%h);
3  print "$a $b\n";
4  ($a,$b)=each(%h);
5  print "$a $b\n";
6  ($a,$b)=each(%h);
7  print "$a $b\n";
```

A programrészlet sorra kiírja a képernyőre a kulcsokat és az értékeket.

```
1 a
2 b
3 b
_
```

`delete` Egy elemet töröl a hash-ből. Használata:

```
delete($h{2});
```

A programrészlet a `2=>"b"` kulcs-érték párt törli.

`exists` Egy kulcs-érték pár meglétét vizsgálja, ha a kérdéses bejegyzés létezik, akkor a visszatérési érték igaz, ellenkező esetben hamis. Használata:

```
exists($h{2});
```

2.4.3. Szubrutinok

A felhasználó által létrehozott szubrutinok rendkívül rugalmassá teszik a perl nyelvet. A C függvényeihez hasonlítva a perl szubrutinjait láthatjuk, hogy használatuk kicsit egyszerűbb.
A perl szubrutinjainak tulajdonságai:

- A szubrutint nem kell deklarálni (de lehet).
- A szubrutin fejléce a **`sub`** kulcszóval kezdődik.
- A szubrutin törzsét a {} operátorpár fogja közre.
- A szubrutin a paramétereit a `@_` tömbön keresztül kapja meg.
- A szubrutin a visszadott értéket a **`return`** utasítással adja vissza. Ha a `return`-t paraméterek nélkül hívjuk meg, akkor a program a szubrutinból visszaadott érték nélkül tér vissza.

Nézzünk példaként egy szubrutint, amely kiszámítja két szám legnagyobb közös osztóját [11].

```
1  sub euklid
2     {
3     ($a,$b)=@_;
4     do
5        {
6        $c=$a%$b;
7        $a=$b;$b=$c;
8        }
9     while($c!=0);
10    return $a;
11    }
```

A szubrutin 3. sorában a `@_` default tömb átadja a paraméterlistán átadott paramétereket az `$a` és `$b` skalároknak az első paraméter az `$a`-ba a második a `$b`-ba kerül. Az eredményt a 10. sorban található `return` segítségével adjuk vissza a hívónak.
A szubrutin hívása a következő módon történik:

```
$lnko=&euklid(192,45);
```

Ennek a szubrutinnak az a baja, hogy nem teljesen általános felhasználású a `$a, $b` és `$c` változó, ha más programrészletben felhasználásra kerül, akkor értéke megváltozik.
Ezeknek a változóknak a hatásköre globális [12].

Definíció: Egy változó globális hatáskörrel rendelkezik, ha a program teljes területén látható.

Az általános felhasználáshoz a szubrutinnak olyan változókra van szüksége, amelyek csak a saját kódterületén, vagyis a rutin {} operátorai között látszanak. Ezt a hatáskört **lexikus** hatáskörnek nevezzük.

[11] Euklideszi algoritmus.
[12] Nem globális típusú, hanem globális hatáskörű, vagy láthatóságú.

Definíció: Egy változó lexikus hatáskörrel rendelkezik, ha csak és kizárólag abban a szubrutinban látszik, ahol létrejött.
A lexikus hatáskört a `my` kulcsszóval jelöljük ki.

Példa a használatra a rekurzív faktoriális számító szubrutin.

```
1  sub fact
2     {
3     my $a=$_[0];
4     if($a<=1)  return 1;
5     $a=$a*fact($a-1);
6     return $a;
7     }
```

A szubrutin működésének magyarázata:

- A 3. sorban az átadott paraméter értékét a `$a` változónak, ennek a változónak a hatásköre lexikus.
- A 4. sorban ha `$a` értéke 0, vagy egy a rutin 1 értékkel tér vissza, mivel $0! =$, illetve $1! = 1$.
- Az 5. sor a szubrutin lelke, a tulajdonképpeni rekurzió.

  Ha a szubrutint meghívjuk 3-mal, akkor az 5. sorban a 3 értékét megszorzzuk a szubrutin második meghívásának visszatérési értékével, ennek a hívsnak a paramétere 2. A második hívás során az 5. sorban a 2-es értéket a rutin harmadik meghívásának visszatérési értékével szorozzuk. A harmadik hívás paramétere 1, ennek a visszatérési értéke 1. A visszatérés után ezzel az 1 értékkel szorozzuk a 2-őt, tehát a második hívás vissztérési értéke 2. Ezzel a 2-vel szorozzuk a 3-at, így a rutin visszatérési értéke 6.

A harmadik hatáskör a **lokális**.

Definíció: Egy változó hatásköre lokális, ha a változó abban a szubrutinban látható, ahol létrejött, illetve azokban a szubrutinokban, amelyeket a kérdéses szubrutinból hívunk meg.
A lokális hatáskört a `local` kulcsszóval jelöljük ki.

A hatásköri kérdések tisztázására nézzük a következő három példaprogramot.

1. példa

```
$a="fo\n";
&rut1();
print $a;

sub rut1
 {
 $a="r1\n";
 print $a;
 &rut2();  }

sub rut2
 {
 print $a;
 }
```

2. példa

```
$a="fo\n";
&rut1();
print $a;

sub rut1
 {
 my $a="r1\n";
 print $a;
 &rut2();  }

sub rut2
 {
 print $a;
 }
```

3. példa

```
$a="fo\n";
&rut1();
print $a;

sub rut1
 {
 local $a="r1\n";
 print $a;
 &rut2();  }

sub rut2
 {
 print $a;
 }
```

A 1. példa esetén nem használunk hatásköri módósítot, ezért a képernyőn a

```
r1
r1
r1
```

felirat jelenik meg. A program első sorában az `$a` változó értéket kap, azonban a `&rut1()` szubrutin ezt az értéket felülírja a saját értékével, és ezt az új értéket írja ki a `&rut1()` a `&rut2()` és a főprogram is.

A 2. példa esetén a `&rut1()` szubrutinban a `$a` változót lexikussá tesszük, így a képernyő a következő lesz:

```
r1
fo
fo
```

Mivel a lexikus változó csak az adott rutinra vonatkozik, ezért az `r1` üzenetet csak `&rut1()` írja ki. A `&rut2()` és a főprogram a `fo` szöveget írja ki.

A 3.példa esetén a `&rut1()` szubrutinban a `$a` változót lokálissá tesszük. Ekkor a képernyő a következőképpen fog kinézni:

```
r1
r1
fo
```

A local hatására a `&rut1()` szubrutinban a `$a` változó lokális lesz, tehát látszani fog a `&rut1()` és a `&rut2()` szubrutinban, de a főprogram `$a` változója nem változik meg.

Abban az esetben, ha egy változót létrehozunk egy szubrutinban, akár `my`, akár `local` módosítással, akkor az adott helyen egy új tárolóterület jön létre [13].

Ha az adott hatáskörben azonos néven különböző hozzáférésű változó szerepel, akkor mindig a legszigorúbban módosított változó látszik.

2.5. Reguláris kifejezések

A reguláris kifejezések az úgynevezett mintaillesztésre szolgálnak. Ez azt jelenti, hogy egy skalárban tárolt sztringet vizsgálunk abból a szempontból, hogy tartalmaznak-e adott rész-sztringeket, adott számban, adott helyen. Egyes szolgáltatások lehetővé teszik, hogy bizonyos feltételeknek megfelelő rész-sztringekkel műveleteket végezzünk.

A reguláris kifejezések minden esetben a `//` jelpár között állnak. Ha a reguláris kifejezés az úgynevezett default változóra vonatkozik igaz állításként, akkor további operátor haszálata nem szükséges. Példa a használatra:

```
if(/kifejezes/)
```

[13] Vagyis egy új változó.

```
{
    ⋮
}
```

Ha a hamis állítást vizsgáljuk, akkor csak a reguláris kifejezés nyitó `/` jele elé kell egy `!`-et írnunk.

2.5.1. A reguláris kifejezések operátorai

Az reguláris kifejezések operátorai a következők:

`=~  !~  s///  tr///`

Itt is találkozunk azzal a már ismert problémával, hogy különböző szakírók néhány függvényt ebben a témakörben is operátornak vesznek[14].

A reguláris kifejezések belső operátorai:

`^  $  [...]  [^...]  |  .  *  +  ?  {n}  {n,}  {n,m}  ()`

A belső operátorok működését a `=~` operátor ismertetésénél adjuk meg.

`=~` ponált mintailesztő operátor. Ha az operátort követő kifejezés illeszkedik az operátor bal oldalán lévő skalárra, akkor a kifejezés igaz lesz. Nézzük a következő példát!

```
1  @t=qw(aaa aab aba abb baa bab bbb);
2  foreach $k (@t)
3     {
4     if($k=~/a/)
5        {
6        print "$k ";
7        }
8     }
```

A képernyőre a következő kiírás kerül: `aaa aab aba abb baa bab bba_`, tehát minden olyan skalár kiíródik, amelyben `a` karakter van.

`^` a sor elejére illesztő operátor. Ha az előző példa 4. sorát kicseréljük a következő sorra:
`if($k=~/^a/)`,
akkor a képernyőre azok a skalárok kerülnek kiiratásra, amelyek az `a` karakterrel kezdődnek, vagyis az `aaa aab aba abb` sztringeket.

`$` a sor végére illesztő operátor. Figyelem az operátort nem a minta elé, hanem mögé kell tenni, mert ha elé tesszük, akkor skalárként, vagy indirekcióként értelmezi.

Cseréljük ki a negyedik sort a következőre:
`if($k=~/a$/)`
A képernyőn azok a skalárok lesznek, amelyek `a` karakterre végződnek, vagyis `aaa aba baa bba` sztringek.

[14] Igen, egy kicsit ”rendetlen” a perl, de nagyon hatékony.

[...] karakterosztály operátor. Megadhatók azon karakterek, illetve tartományuk, amelyekre a kifejezés illeszkedik. Peldák a használatára:

```
if($k=~/[abc]/)
```

ez olyan sztringekre illeszkedik, amelyek tartalmazzák vagy az a vagy a b vagy a c karaktereket. Ez a kifejezés másként is megadható:

```
if($k=~/[a-c]/)
```

Ekkor tartományt adtuk meg.

Az egész alfanumerikus tartomány megadása a következő módon adható meg:

```
if($k=~/[a-zA-Z0-9_]/)
```

Ebben a példában a tartományok kombinálását láthattuk.

[^...]

^... a karakterosztály negálása. Azokra a sztringekre illeszkedik, amelyekben megadott karakterosztály elemein kívül más elemek is vannak.

A jól definiálható karakterosztályok megadásának egyszerűsítésére metakaraktereket definiáltak, ezek a teljesség igénye nélkül:

\w	alfanumerikus és az _karakter
\W	nem alfanumerikus
\d	számjegy
\D	nem számjegy
\s	szóköz jellegű karakter
\S	nem szóköz jellegű karakter
\l	kisbetű
\u	nagybetű

| alternatívák elválasztására szolgál. Használata:

```
1  @t=qw(aaa aab bee ebc ddd eee eec bbb);
2  foreach $k (@t)
3     {
4     if($k=~/a|c|d/)
5        {
6        print "$k ";
7        }
8     }
```

A képernyőre kiírt szöveg:

```
aaa aab ebc ddd eec
```

. általános helyettesítő operátor, bármilyen karakterre illeszkedik a \n-en kívül. Ha az előző példa 4. sorát a

```
if($k=~/./)
```

kifejezésre cseréljük, akkor program az összes skalárt kiírja.

Hogy ennek mi az értleme, következő példán érthető meg. Cseréljük ki a 4. sort

```
if($k=~/b./)
```

Ekkor kiírja az összes olyan skalárt, amelyben van b karakter, de nem az utolsó helyen,

vagyis

```
bee ebc bbb
```

a kiírt szöveg.

Tehát a `.` operátor a 4. sorban előírja azt, hogy a `b`-t egy karakternek mindenképpen követnie kell.

$*$ az előtte lévő karakter 0, vagy több előfordulását írja elő. Ha egy egyszerű `if`-es kifejezésben használjuk, mint az előző példában, akkor minden mintára illeszkedni fog.

```
if($k=~/a*/)
```

Minden olyan mintára illeszkedik, ahol 0 vagy több `a` betű fordul elő.

Megjegyzés: Ennek így első pillantra nincs sok értelme, de a későbbiekben láthatjuk, hogy nagyon jól használhatjuk.

$+$ az előtte lévő karakter 1 vagy több előfordulására illeszkedik.

? az előtte lévő karakter 0 vagy 1 előfordulására illeszkedik.

{**n**} pontosan az előtte lévő karakter `n` elemű előfordulására illeszkedik.

Ha `if`-es kifejezésben használjuk érdekes eredményt kapunk. A példa ezt mutatja be.

```
@t=qw(bab baab baaab baaaab baaaaab);
foreach $k (@t)
   {
   if($k=~/ba{3}b/)
      {
      print "$k ";
      }
   }
```

A program kiírja a `baaab` szöveget.

{**n,m**} az előtte lévő karakter minimum `n`, de maximum `m` számú előfordulására illeszkedik.

{**n,**} az előtte lévő karakter minimum `n`, számú előfordulására illeszkedik.

{**,m**} az előtte lévő karakter maximum `m`, számú előfordulására illeszkedik[15].

() csoportosító operátor. Azokat a sztring részleteket rögzíti, amelyeket a zárójelekben helyezünk el. Használata a szövegfeldolgozásban igen széleskörű. A következő példa egy alkalmazást mutat be.

```
1  $k="abcdefghijk"
2  ($a,$b)=$k=~/ab(.)de(.)/;
3  print $a.$b."\n";
```

A program ki fogja írni, hogy `cf`.

A 2. sorban a `$a`, `$b` skalárok felveszik azokat a sztring részleteket, amelyek az adott helyen vannak.

[15] A tapasztalatok azt mutatták, hogy ez a szolgáltatás nem minden perlben működik. Ekkor alkalmazzuk a {`0,m`} kifejezést.

`!~` negált mintaillesztő operátor. Működése pontosan a negáltja a `=~` operátornak akkor, ha `if`-es kifejezésben használjuk.

`s///` helyettesítés operátor. Az operátor szerkezete a következő:

```
s/1.kifejezes/2.kifejezes/eloirasok
```

Az operátor 1. kifejezést keresi a szövegben, ha megtalálja, akkor kicseréli a 2. kifejezésre figyelembe véve az előírásokat.

Használatára nézzük a következő példát:

```
$a="abcdefg";
$a=~s/cd/xxx/;
print "$a\n";
```

A kiírt szöveg `abxxxefg` . Vagyis a `cd` szövegrészletet az operátor `xxx`-re cserélte.

Az előírások a következők:

`i`	nem tesz különbséget a kis és nagybetűk között,
`m`	a sztringet úgy kezeli, mint többsoros szöveget,
`e`	a kifejezésben vegrehajtandó kód van,
`s`	a szöveget egyetlen sorként kezeli,
`x`	ne vegye figyelembe a szóközjellegű karaktereket,
`g`	minden minta előfordulásra elvégzi a cserét,
`c`	a `g` módosító esetén sikertelen keresés esetén ne nullázza a mintát,
`o`	az operátor a kifejezést csak egyszer fordítsa le, ne minden végrehajtás esetén.

`tr///` fordítás operátor. Az operátor megvizsgálja a sztringet egy megadott karakterlistát keresve, ha megtalálja, akkor a megadottra kicseréli. Az operátor szerkezete:

```
tr/KeresettLista/CsereLista/eloirasok
```

Az előírások a következők lehetnek:

`c`	mindazon karaktereket cseréli, amelyek nem felelnek meg a `KeresettLista`-nak,
`d`	törli az összes meghatározott karaktert,
`s`	kicseréli a többszörösen azonos kimenő karaktereket egyetlen karakterré,

A `tr///` operátor tipikus alkalmazása a kisbetű $\rightleftharpoons$ nagybetű konverzió.

```
$a=~tr/a-z/A-Z/;    # kisbetu->nagybetu
$a=~tr/A-Z/a-z/;    # nagybetu->kisbetu
```

2.6. Referenciák

A referenciák tulajdonképpen mutatók. Bármilyen típusú változónak képezhetők a referenciái, de ennél több szolgáltatást is képesek nyújtani. Segítségükkel képezhetünk például többdimenziós tömböket, szervezhetünk hash-eket tömbbe, illetőleg lehetőségünk van szimbólikus referenciák használatára.

2.6.1. Normál referencia, és indirekció

A normál referenciák egy változó címét tartalmazzák. A változó típusa tetszőleges lehet. A címképzésről a 18. oldalon már beszéltünk. Most egy példát mutatunk be a normál referenciák használatára.

A példában az általános másodfokú egyenlet megoldását mutatjuk be, ahol a feladatot megoldó szubrutin hagyományos módon visszaadja a megoldás jellegét, és referenciák segítségével a gyököket, illetőleg a gyökök részeit.

```
1  $r=&masod($A,$b,$C,\$X1,\$X2);
2  if($r)  { print "$X1 $X2\n"; }
3  else    { print "$X1+-j$X2\n"; }
4
5  sub masod
6     {
7     my $a=$_[0];my $b=$_[1];my $c=$_[2];  # parameterek
8     my $x1=$_[3];my $x2=$_[4];            # eredmeny(ek) cime
9     my $d=$b**2-4*$a*$c;                  # diszkriminans
10    if($d>=0)                             # valos gyokok
11       {
12       $$x1=(-$b+sqrt($d))/(2*$a);
13       $$x2=(-$b-sqrt($d))/(2*$a);
14       return 1;                          # jelzes, valos gyokok
15       }
16    $$x1=-$b/(2*$a);                      # valos resz
17    $$x2=sqrt(-$d)/(2*$a);                # kepzestes resz
18    return 0;
19    }
```

Az 1. sorban meghívjuk a `&masod` szubrutint öt paraméterrel, amelyek értelmezése a következő: $ax^2 + bx + c$, ahol a hívás `$A` paramétere az egyenlet a paraméterének, a `$B`, a b és a `$C` a c-nek felel meg.

Az utolsó két paraméter két skalár változó címe, ezekbe a változókba fogjuk az eredményt indirekt módon belerakni.

A program 8. sorában a lexikus `$x1` és `$x2` változónak átadjuk az előzőekben említett címeket. Ezt a lépést meg kell tennünk, mert a `$_[3]` `$_[4]` változók nem használhatók közvetlenül indirekcióra.

Amennyiben az egyenlet megoldása valós, akkor az eredmény a 12. és 13. sorban keletkezik.

A 12. sor balodala szavakban elolvasva a következő: **Az `$x1` skalár által címzett memóriarekesz tartalma legyen** . . . . Ezt nevezzük indirekciónak, mivel az ertéket nem közvetlenül, tehát direkt módon, hanem címen keresztűl, tehát közvetve adjuk át.

Hasonló történik a 13., a 16. és a 17. sorban. A szubrutin a 14. és a 18. sorban egy jelzőértéket ad vissza, amivel jelzi a hívó programnak, hogy a hogyan kezelje az eredményt. Vagyis a 2. sorban található kiiratást vagy a 3. sorban lévő kiiratást alkalmazza.

2.6.2. Szimbólikus referencia

Tekintsük a következő programrészletet:

```
1  $szam=5;
```

```
2  $ref="szam";
3  print $$ref."\n";
```

A program kiírja az 5-ös értéket. Tehát egy változóra a nevével és nem a címével hivatkoztunk. Ezt nevezzük szimbólikus hivatkozásnak és a `$ref` változó egy szimbólikus referencia.
Természetesen nem csak skalárhoz rendelhetünk szimbólikus referenciát, hanem tetszőleges típushoz.

2.6.3. Névtelen tömbök

Tömböket létre tudunk úgy is hozni, hogy nem nevet adunk meg a tömbnek, hanem egy referenciát rendelünk hozzá. Ezért nevezik ezeket a tömböket névtelen tömböknek. A következő példa egy egydimenziós névtelen tömböt hoz létre.

```
$t=[ "nulla","egy","ketto","harom" ];
```

Vegyük észre, hogy a változó egy skalár. Természetesen, hiszen ez egy mutató.
Egy adott elemhez való hozzáférés a következő módón történik:

```
print $t->[2];
```

A programrészlet, az előző inicializálást figyelembe véve, kiírja: `ketto` .
A tömböknél említettük, hogy csak egydimenziós tömböket lehet létrehozni. Mivel a tömb skalárokból áll, és a névtelen tömb skalárhoz van rendelve, így lehet többdimenziós tömböket készíteni.

```
@t=( [ 1,2,3 ],
     [ 4,5,6 ],
     [ 7,8,9 ] );
```

Ha az 1, 2 indexű elemet szeretnénk elérni, akkor a `$t[1]->[2]` kifejezést kell alkalmazni.
Hozzunk létre kétdimenziós névtelen tömböt.

```
$t=[ [ 1,2,3 ],
     [ 4,5,6 ],
     [ 7,8,9 ] ];
```

Ha az ugyanazt az elemet szeretnénk elérni, mint az előbb, akkor a `$t->[1][2]` kifejezést kell használnunk.

2.7. Csomagok

A perl nyelv lehetőséget biztosít különböző láthatósági körök létrehozására. Ezt biztosítják az úgynevezett csomagok.
Minden perl program tulajdonképpen legalább egy csomagot tartalmaz. Ez a csomag az un. default csomag a `main`.
Egy csomag kezdetét—kivétel a default csomag—a `package` kulcsszó jelzi, amelyet a csomag neve követ. Valahogy így: `package elso;`
A csomagokra a következő szabályok vonatkoznak:

- Minden változó és hivatkozási név az aktuális csomag szimbólum táblájában kerül tárolásra.

- Egy változóra, amely az aktuális csomagban jött létre, a változó egyszerű nevével hivatkozhatunk. Ha egy másik csomag változójára hivatkozunk, akkor a változó nevét a csomag nevével kell kiegészítenünk.
- Az alapértelmezett csomag a `main`.

Nézzük a következő példát.

```
1  $x="Alapertelmezett\n";
2  package elso;
3  $x="Elso\n";
4  package masodik;
5  $x="Masodik\n";
6  print $main::x.$elso::x.$x;
```

A programrészlet kiírja a képernyőre:

```
Alapertelmezett
Elso
Masodik
```

Az 1. sorban az `$x` változó a `main` csomaghoz tartozik. A 2. sorban egy csomagot definiálunk, amelyben a 3. sorban szintén egy `$x` skalárt hozunk létre. A 4. sorban létrehozzuk a következő csomagot, amelyben újfent definiálunk egy `$x`-t.
A 6. sor `print`-je először kiírja a `main` `$x`-ét, majd az `elso` `$x`-ét, végül az aktuális `$x`-t, amely a `masodik` csomaghoz tartozó változó.
FIgyeljük meg, hogy a csomaghoz tartozást a `::` operátorral definiáljuk. Ezt az operátort **hozzáférés definiáló operátornak (HDO)** nevezzük[16].
A csomagok használatára az OOP-ben látunk még példákat.

2.8. Modulok

A perl számos előre definiált függvényt biztosít számunkra, azonban speciális feladtatok megoldására nekünk kell megírni a szubrutinjainka. Sokszor jó lenne ezeket a programrészleteket olyan formátumban eltenni, hogy könnyen felhasználhassuk a későbbiekben. A modulok erre adnak lehetőséget.
A modul egy perl nyelven megírt program, amely általában globális változókat és szubrutinokat tartalmaz, általában önálló futásra nem alkalmas.
A modult egy **igaz utasítás** zárja le. Az igaz utasítás: `1;` .
A modul beolvasásának egyik módja a `require` utasítás. Használata:

```
require 'modul.pl';
```

Ebben az esetben a beolvasandó modul a `modul.pl`. A kérdés az, hogy hol helyezkedhet el ezek a modulok?

[16] `::` operátornak a szakirodalomban számos elnevezése található például: hiavtkozás operátor, scope operator, scope resulation operator, namespace operator.

A perlnek van egy rendszerváltozója az `@INC` tömb, amelyben modulok elérési útja található. Általában a az `@INC` tartalmazza a `.` sztringet, amely az aktuális könyvtár, így ha a modult ugyanabba a könyvtárba helyezzük, ahova programot, akkor az elérhető lesz.
A modul beolvasásának másik módja a `use` utasítás használata. A `use` feltételezi, hogy a modul ”kiterjesztese” `.pm`, ezért ezt nem kell a modul nevéhez hozzátenni. Alapesetben[17] a használata:

```
use modul;
```

A fenti kifejezés a `modul.pm` nevű modult olvassa be.

[17] Alapesetben, mert a `use` többet tud, erről az objektumorientál perl alkalmazásoknál ejtünk néhány szót.

3. IO kezelés

Az IO kezelés a UNIX terminológia szerint alapvetően a fájlkezelést jelenti. Ez jelentheti a ”közönséges” fájlok kezelését, a karakteres be- és kimenetet, a csővezetékek kezelését és a hálózati forgalom egyes szolgáltatásait. Ebben a fejezetben beszélünk a fájlteszt operátorkról is, amelyekről a relációs operátorknál már említést tettünk.

3.1. Fájlkezelés

3.1.1. A fájl megnyitása és lezárása

Ahhoz, hogy egy fájlhoz hozzáférjünk, vagyis írhassuk, olvashassuk, létrehozzuk, hozzáfűzzünk, illetőleg a fájlon egyéb műveleteket végezzünk a fájlt megfelelő módon meg kell nyitnunk.
Vannak speciális fájlok, amelyeket nem kell külön megnyitni. Ezek az úgynevezett standard fájlok. A perl három ilyet ismer, ezek:

- `STDIN` a standard bemenet. Alapértelmezésben a billentyűzet.
- `STDOUT` a standard kimenet. Alapértelmezésben a képernyő.
- `STDERR` a standard hibajelzés kimenete. Alapértelmezésben a képernyő.

A standard fájlok átirányíthatók[18].
A fájl megnyitásához az `open` függvényt használjuk. A függvény alakja a következő:

```
open(FILE,MODE.PATH);
```

`FILE` az úgynevezett fájl kezelő (file handler) ezzel a speciális változóval azonosítjuk a fájlt.

`MODE` az a mód, ahogy a fájlt megnyitjuk.

`PATH` a fájl elérési útja[19].

Tegyük fel, hogy a következőkben a megnyitandó fájlunk az aktuális könyvtárban lévő `file.txt` lesz.
A fájl megnyitási módja a következő lehet:

- olvasásra nyitás:

  ```
  open(FILE,"<file.txt");
  ```

- írásra nyitás:

  ```
  open(FILE,">file.txt");
  ```

  ez a megnyitási mód a fájlt az elejéről írja, vagyis ha már a fájl létezett a hosszát 0-ra levágja és újra kezdi írni. Ha a fájl a megnitás előtt nem létezett, akkor létrehozza.

[18] Tehát alkalmazhatók az átirányítás operátorai: >, >>, <, |.
[19] Figyeljük meg a . operátort, összefűztük a megnyitási módot a fájl nevével.

- hozzáfűzésre nyitás:

  ```
  open(FILE,">>file.txt");
  ```

  a már létező fájlt úgy nyitja meg írásra, hogy a fájl pozíció mutatót[20]a fájl végére állítja. Ha a fájl nem létezik, akkor a létrehozza.

- írásra—olvasásra nyitás:

  ```
  open(FILE,"+<file.txt");
  ```

  A fájlt írásra és olvasásra nyitja meg, de nem vágja le nullára a hosszát. Ha a fájl nem létezik nem hozza létre.

- írásra—olvasásra—levágásra nyitás:

  ```
  open(FILE,"+>file.txt");
  ```

  A fájlt írásra és olvasásra nyitja meg, de levágja nullára a hosszát. Ha a fájl nem létezik létrehozza.

- hozzáfűzésre és olvasásra nyitás:

  ```
  open(FILE,"+>>file.txt");
  ```

  A fájlt hozzáfűzésre és olvasásra nyitja meg. Ha a fájl nem létezik, akkor létrehozza.

A file lezárása a `close` függvénnyel történhet. Használata:

```
close(FILE);
```

3.1.2. Hibakezelés

Ha egy fájl megnyitása sikeres, akkor minden rendben van, dolgozhatunk az állománnyal. Azonban, ha a megnyitás sikertelen, akkor kezelnünk kell a hibát. Erre szolgál a hibakezelés. Erre a célra a perl két utasítást használ, ezek a `warn` és a `die`.
A `warn` használata a következő:

```
open(FILE,"<file.txt")||warn "sikertelen file nyitas\n";
```

Ha a fájlmegnyitás sikertelen volt, akkor az open hamis értékkel tér vissza, így a vezérlés a `warn` utasításra kerül, amely kiírja a hibaüzenetet a képernyőre, majd továbbengedi a programot. A || operátor helyett használható az `or` is.
A `die` használata tejesen azonos az előzőekben látott `warn` használatával. A különbség az, hogy a `die` a hiba kiírása után leállítja a programot. Példa a `die` használatára:

```
open(FILE,"<file.txt")||die "sikertelen file nyitas\n";
```

Megjegyzés: Ha már egy kicsit tapasztaltabbak vagyunk, akkor rájövünk arra, hogy ezzel a logikával bármilyen utasítást, vagy szubrutint meghívhatunk hibakezelési célból. Gondoljunk csak bele, ha grafikus képernyőn dolgozunk milyen furcsa egy karakteres üzenet, ha egyáltalán látható.

[20]A fájl pzíció mutató az a változó, amely megmutatja, hogy a fájlban melyik pozíción történik a közvetkező művelet. A fájl pozíció mutató értéke olvasás és írás eseten növekszik akkora értékkel, ahány bájtot olvastunk be, illetve írtunk ki.

3.1.3. Fájlok olvasása és írása

A fájlok olvasásánál megkülönböztetjük a binárisan kezelt állományok olvasását és írását a szöveges állományok olvasásától és írásától. Természetesen vannak olyan függvények, amelyek mindkétféle állomány kezelésénél használhatók.
Elsőként nézzük a szöveges fájlkezelés függvényeit.
Amennyiben megnyitottunk egy szöveges fájlt olvasásra, amely a megszokott `file.txt` névre hallgat és amelyet a `FILE` kezelővel azonosítunk, és ebből egy sornyi információt akarunk beolvasni egy skalár változóba, akkor ezt a következő módon tehetjük meg:

```
$sor=<FILE>;
```

A `$sor` változóban az éppen aktuális sor bekerül. **Ne feledjük** a soremelés karakter ott van a sor végén, amit a `chomp($sor);` kifejezéssel távolíthatunk el.
Amennyiben az olvasandó fájl a standard input, akkor egy sor beolvasása a következő két módon történhet:

```
$sor=<STDIN>;                $sor=<>;
```

A bal és a jobboldali kifejezés teljesen azonos értékű.
A fájl karakterenkénti olvasását a `getc` függvény végzi. A függvény használata:

```
$chr=getc(FILE);
```

A `$chr` skalárba kerül az a karakter, amelyre a fájl pozíció mutató mutatott az olvasás előtt.
A `getc` függvény természetesen bináris állományok esetén is használható, ekkor egy bájtnyi információt olvas be.
A fájlba írás a már ismert `print` függvénnyel történik. A függvény használata:

```
print FILE kiirando
```

Figyeljük meg, hogy a `FILE` fájlkezelő után semmilyen elválasztó karakter sincs, csak egy szóköz. Ez nagyon lényeges, mert ha nem így használjuk a függvényt a kiírás nem működik.
Amennyiben a standard kimenetre írunk, akkor a `print` függvényt a már ismert módon használjuk.
Ellentétben az eddigiekkel most két teljes programot ismertetünk. Az első program minden sor végére egy ”kocsivissza-soremelés” szekvenciát tesz a `UNIX` rendszerekben szokásos soremelés helyett.

```
1  #!/usr/bin/perl
2  while(<>)
3     {
4     chomp($_);
5     print "$_\r\n";
6     }
```

Az 1. sor `UNIX` rendszerek esetén a futtató program elérési útját adja meg. Elképzelhető, hogy másik rendszerben ez a sor eltér az ittenitől.
A 2. sor egy ciklus fej, amíg a `<>` olvas a standard benetről addig a ciklus fut. Ha fájl végét olvas a ciklus végetér.

A 4. sorban a `chomp` levágja a soremelést a default skalárról.

Az 5. sorban kiírja a soremelés nélküli defalut skalárt úgy, hogy hozzáfűz egy kocsivissza és egy soremelés karaktert.

Ez a program lehetővé teszi, hogy egy `UNIX` környezetben készített szövegfájlt egy hagyományos `DOS` környezetben működő nyomtatóval kinyomtathassunk. Legyen a nyomtatandó fájlunk `fájl.txt`, a nyomtató port `/dev/lp0`, a perl program neve `todos.pl`.

```
cat file.txt | ./todos.pl >/dev/lp0
```

A másik program a tabulátor karaktereket cseréli ki megfelelő mennyiségű szóközre úgy, hogy a listázott állomány ugyanúgy nézzen ki, mint az eredeti. A példában a hármas tabulációs lépést használtuk.

Legyen a program neve `tabkill.pl`

```
1  #!/usr/bin/perl
2  $i=0;
3  while(1)
4     {
5     $c=getc(STDIN);
6     if($c eq "")  {  last; }
7     if($c eq "\n")
8        {
9        $i=0;
10       print $c;
11       next;
12       }
13    if($c eq "\t")
14       {
15       print " ";
16       $i++;
17       for(;$i%3;$i++)
18          {
19          print " ";
20          }
21       next;
22       }
23    print $c;
24    $i++;
25    }
```

A példában két skalár változót használunk. A `$i` a változó pozíció számláló, a `$c` az átmeneti tároló. A standard inputról ide kerül beolvasott karakter, amelyet a tovúbbiakban feldolgozunk.

A program 1. sorát már az előző példából ismerjük.

A 2. sorban a karakter számlálónak kezdőértéket adunk.

A 3. sor egy végtelen ciklus feje.

Az 5. sorban a `$c` skalárba egy karaktert belovasunk a standard bemenetről.
A 6. sorban megvizsgáljuk, hogy a beolvasott karakter üres-e- Ha igen a `last` utasítással elhagyjuk a ciklust és kilépünk a programból.
A 7. sorban azt vizsgájuk, hogy a beolvasott karakter soremelés-e.
Ha igen, akkor 9. sorban a karakter számlálót nullázzuk (persze a következő sor elölről kezdődik).
A 10. sorban kiírjuk a soremelést a standard kimenetre.
A 11. sorban a `next` utasítással elölről kezdjük a ciklust.
A 13. sorban azt vizsgáljuk, hogy a beolvasott karakter tabulátor-e.
Ha igen a 15. sorban kiírunk egy szóközt, majd a 16. sorban növeljük a karakterszámlálót.
Erre azért van szükség mert, ha a kurzor éppen tabulációs pozíción áll (jelen esetben hárommal osztható karakterpozíción áll), akkor egy újabb tabulátor beolvasása nem mozdítaná tovább a kurzort.
A 17. sorban látható `for` utasítás első kifejezését nem használjuk ki. A második kifejezés azt vizsgálja, hogy a karakterpozíció osztható-e hárommal. Ha osztható, akkor nincs maradék, tehát az eredmény nulla, ami azt jelenti, hogy a második kifejezés ekkor hamis, következésképpen a `for` ciklus befejeződik.
A 19. sorban kiírunk egy szóközt, ami a `for` harmadik kifejezésével együtt egy karakterpozíciót jelent a kimeneten.
A 21. sorban a `next` segítségével újból kezdjük a `while` ciklust.
A 23. sorra akkor kerül a program, ha a `$c` nem üres, soremelés vagy tabulátor. Ekkor a beolvasott karaktert változatlanul kiírjuk a standard kimenetre.
A 24. sorban növeljük a karakterpozíció számlálót.
Tegyük fel, hogy a ”formázandó” állományunk neve `progi.c`, akkor a `tabkill.pl` használata:

```
cat progi.c | ./tabkill.pl >progi.txt
```

A formázott állomány a `progi.txt`.
Sok esetben a fájlban tárolt információ nem szöveges jellegű[21], hanem bináris jellegű. Ekkor a `<...>` nem használható. A `getc` igen, de ez a függvény csak egy karaktert olvas be, ami nehézségeket okozhat. Amennyiben az operációs rendszer különbséget tesz a bináris és a text módú fájlkezelés között, akkor alkalmazni kell a **`binmode`** függvényt. Ilyen operációs rendszerek például a `WINDOWS x`-ek. A `UNIX` jellegű rendszerek esetén ez a függvény hatástalan.

Megjegyzés: Ha egy operációs rendszer különbséget tesz a szöveges és a bináris fájlkezelés között, akkor szöveges módban beolvasáskor egy kocsivissza—soremelés szekvencia esetén csak a soremelés kerül beolvasásra. Amikor a fájlba irunk, akkor egy soremelés kiírásakor a fájlba egy kocsivissza—soremelés kerül.

Bináris módban a rendszer minden karaktert változatlanul átvisz a fájl és a ”külvilág” között.

A `binmode` használata:

```
binmode(HANDLE,MODE);
```

`HANDLE` A jól ismert fájl kezelő.

`MODE` A mód leírására szolgáló paraméter, amely lehet:

[21] Ettől még persze lehet olvashat, értelmes szöveg.

- `":raw"` vaggyis bináris,
- `":crlf"` vagyis szöveges.

A `binmode` függvényt az `open` után kell használni, de még az előtt, hogy adatot mozgatnánk a fájlba vagy a fájlból.
A meghatározott hosszúságú adatblokkok olvasására a **`sysread`** függvény szolgál. Használata:

```
sysread(HANDLE,SKALAR,LENGTH);
sysread(HANDLE,SKALAR,LENGTH,OFFSET);
```

`HANDLE` a fájlkezelő,

`SKALAR` a skalár, ahova a fájlból az adatokat olvassuk,

`LENGTH` a beolvasandó adat hossza bájtokban,

`OFFSET` megadja, hogy a beolvasott adat a skalár hányadik bájtján kezdődjön, ha a skalár üres, akkor a beolvasást elölről kezdi. Ha az `OFFSET` értéke nagyobb, mint a skalárban tárol adat hossza, akkor a szükséges területet a skalárban 0 értékű bájtokkal tölti fel. A negatív `OFFSET` a skálár végétől számítja a betöltési pozíciót.

Nézzünk egy példát a `sysread` használatára. Legyen egy `text.txt` fájlt, amelyne tartalma a következő:

```
0123456789abcdefghijklmnopqrstuvw
```

A `sysread`-ot használó program neve `sr.pl`, listája:

```
1  #!/usr/bin/perl
2  sysread(STDIN,$a,10);
3  print "$a\n";
4  sysread(STDIN,$a,5,5);
5  print "$a\n";
6  sysread(STDIN,$a,5,-3);
7  print "$a\n";
```

A program használata:

```
cat text.txt | ./sr.pl
```

A 2. sor beolvassa a fájl első tíz bájtját (a standard input közbeiktatásával).
A 3. sor kiírja `0123456789`
A 4. sor az `$a` ötödik bájtjától olvas be öt bájtnyi adatot.
Az 5. sor kiírja `01234abcde`
A 6. sor az `$a` végétől számított hármadik bájtra olvas be öt bájtot.
A 7. sor által kiírt szöveg `01234abfghij`

Megjegyzés: Azt ne felejtsük el, hogy minden olvasás a beolvasott bájtok számával növeli a fájl pozíció mutató értékét.

A meghatározott hosszúságú adatok kiírására a **`syswrite`** függvény szolgál. A `syswrite` használata:

```
syswrite(HANDLE,SKALAR,LENGTH);
syswrite(HANDLE,SKALAR,LENGTH,OFFSET);
syswrite(HANDLE,SKALAR);
```

HANDLE a fájlkezelő,

SKALAR a skalár, ahonnan az adatokat a fájlba írjuk,

LENGTH a kiírandó adat hossza bájtokban. Ha a hossz nagyobb, mint a skalár hossza, akkor csak annyi bájtot ír ki, amennyi lehetséges. Ha a `LENGTH` nincs specifikálva, akkor a teljes skalárt kiírja a fájlba.

OFFSET megadja, hogy a kiírt adat a skalár hányadik bájtjától kezdődjön, ha a skalár üres, akkor csak és kizárólag 0 értékű `OFFSET` használható. A negatív `OFFSET` a skálár végétől számítja a kiírási pozíciót.

3.1.4. Mozgás a fájlban

A fájl írása és olvasása a fájl pozíció mutatót csak előre, tehát növekvő irányba módosítja. Néha szükségünk lehet arra, hogy egyrészt tudjuk a fájl pozíció mutató értékét, másrészt ezt az értéket változtatni tudjuk.
Az aktuális fájl pozíciót a **`tell`** függvény adja vissza. Használata:

```
$p=tell(HANDLE);
```

A fájl pozíció változtatásához a **`seek`** függvényt használhatjuk. A `seek` használata:

```
seek(HANDLE,POSITION,WHENCE);
```

HANDLE a fájlkezelő,

POSITION az az érték, amellyel a fájl pozíció mutató változni fog,

WHENCE azt mondja meg, hogy honnan számítjuk a változtatást. Ez lehet:

- **0** a fájl kezdetétől,
- **1** az aktuális pozíciótól. Ekkor a negatív értékek a fájl eleje felé, a pozitívak a fájl vége felé mozdítják a mutatót.
- **2** a fájl végétől számít az elmozdulás a negatív értékek itt is a fájl eleje felé történő elmozdulást írják elő.

3.1.5. Csővezetékek nyitása

A `UNIX` rendszerek igen jól használható eszköze a ”csővezeték”[22], amelynek segítségével egy program standard kimenetét összekapcsolhatjuk egy másik program standard bemenetével.
A perl lehetővé teszi, hogy ne csak fájlokat, hanem csővezetékeket is megnyithassunk, így programunk adatait a programon belül átirányíthatjuk egy másik promramra úgy, hogy ez az átirányítás nem jelenik meg parancssorban. Ennek hatalmas jelentősége van biztonsági szempontból.

[22] Angolul pipe.

Csővezeték megnyitása olymódon, hogy a `|`a **program neve előtt van**. Tulajdonképpen ez a ”normális” módja a csővezeték megnyitásának. Az általános formája:

```
open(HANDLE,"|PROGRAM");
```

`HANDLE` a csővezeték kezelője, ennek segítségével hivatkozunk a csővezetékre,

`PROGRAM` a program, amire küldjük az adatokat.

Vegyünk egy hoszabb példát. Először készítsük egy programot, ami a standard bemenetre érkező adatokat egy fájlba menti, amelynek neve `ir.pl`. A program listája magyarázat nélkül:

```
#!/usr/bin/perl
open(FILE,">reg.txt");
while(<>)
   {
   print FILE $_;
   }
close(FILE);
```

A csővezetéket használó program listája:

```
1  #!/usr/bin/perl
2  open(PIPE,"|./ir.pl");
3  print PIPE "Humpty Dumpty sat on the wall\n";
4  print PIPE "Humpty Dumpty had a great fall\n";
5  print PIPE "All the King’s horses\n";
6  print PIPE "All the kings men\n";
7  print PIPE "Couldn’t put Humpty together again\n";
8  close(PIPE);
```

A program második sorában megnyitjuk a csővezetéket, ezután a 3.-tól a 7. sorig erre a megnyitott csővezetékre írjuk ki a szöveget. A 8. sorban lezárjuk a csővezetéket.
A csővezeték használatának másik módja az, amikor a `|`a **program neve után van**. Ez egy érdekes kérdés, mert tulajdonképpen itt nem történik más, mint elindítunk egy programot, aminek a standard kimenete az indító programra van irányítva. A programok programból történő indításáról az 55. oldalon részletesen beszélünk.
A használatra egy nagyon egyszerű példát mutatunk be, az `ls -al` parancs kimenetét vesszük át programunkkal.

```
1  #!/usr/bin/perl
2  open(PIPE,"ls -al|");
3  while(<PIPE>)
4     {
5     print $_;
6     }
7  close(PIPE);
```

A 2. sorban megnyitjuk az csővezetéket és egyben elindítjuk az `ls -al` parancsot.
A 3.-5. sorokban lévő ciklus kilistázza az `ls -al` által küldött sorokat. Ha az adatok elfogytak a standard input küld egy fájl vége karaktert, amire a ciklus befejeződik.
A 8. sorban lezárjuk a csővezetéket.

3.2. Fájl teszt operátorok

A relációs operátoroknál említettük, hogy vannak olyan operátorok, amelyek a fájlok tesztelésével kapcsolatosak. Ezeket az operátorokat ugyanúgy használjuk, mint a ”hagyományos” relációs operátorokat, vagyis egy állítás igazságát (`if`), esetleg hamisságát (`unless`) vizsgáljuk.
Használata `if`-es kifejezésben:

```
if(OPERATOR FILE) {...}
```

`FILE` a fájl neve (esetlegesen elérési úttal),

`OPERATOR` az fájl tesztelő operátor. Ezek:

- **`-r`** a fájl olvasható a felhasználó és csoportja számára,
- **`-w`** a fájl írható a felhasználó és csoportja számára,
- **`-x`** a fájl végrehajtható a felhasználó és csoportja számára,
- **`-o`** a felhasználó birtokolja a fájlt,
- **`-R`** a fájl olvasható valódi felhasználó és csoportja számára,
- **`-W`** a fájl írható valódi felhasználó és csoportja számára,
- **`-X`** a fájl végrehajtható valódi felhasználó és csoportja számára,
- **`-O`** a fájlt valódi felhasználó birtokolja,
- **`-e`** a fájl létezik,
- **`-z`** a fájl mérete 0,
- **`-s`** a fájl mérete nem 0,
- **`-f`** a fájl ”közönséges” fájl,
- **`-d`** a fájl katalógus[23],
- **`-l`** a fájl szimbólikus link,
- **`-p`** a fájl elnevezett csővezeték[24],
- **`-S`** a fájl egy foglalat[25],
- **`-b`** a fájl egy blokkos készülék,
- **`-c`** a fájl egy karakteres készülék,
- **`-t`** a fájl egy terminál számára nyitott,
- **`-u`** a fájl setuid tulajdonságú,

[23] Directory
[24] Named pipe
[25] Socket

-g a fájl setgui tulajdonságú,

-k a fájl sticky tulajdonságú,

-T a fájl text fájl,

-B a fájl bináris fájl,

-M a fájl kora napokban a program indulásakor,

-A a fájl hozzáférési ideje napokban,

-C a fájl inod módosításának ideje napokban.

Megjegyzés: Ennek a jegyzetnek nem célja a UNIX rendszerek működésének ismertetése, ezért számos fogalmat nem magyarázunk meg.

Egy példa segítségével tegyük világosabbá a fájltesztelő operátorok használatát. Nézzük meg, hogy a jól ismert `text.txt` állományunk olvasható-e.

```
#!/usr/bin/perl
if(-r "text.txt")
   {
   print "text.txt is readable\n";
   }
```

Ha a fájl olvasható a program ezt kiírja.
A fájl jellemzőit nemcsak a fájl tesztelő operátorkkal kérdezhetjük le, hanem a **stat** függvénnyel is. A `stat` az eredményt egy listába rakja, amelyet később lekérdezhetünk. Sajnos a lista mérete operációs-rendszer függő. Nézzük a `text.txt` tesztelését a `stat` segítségével.

```
@t=stat("text.txt");
```

A `@t` tömb tartalma `Debian Woody LINUX` disztribúció esetén:

$t[0] a fájlrendszer egységszáma,

$t[1] az fájl inod száma,

$t[2] a fájl típusa és az engedélyek

$t[3] a fájl linkelési száma,

$t[4] a felhasználói azonosító szám,

$t[5] a csoport azonosító szám,

$t[6] készülék azonosító (csak speciális fájlok esetén),

$t[7] a fájl mérete bájtokban,

$t[8] a fájlhoz történő utolsó hozzáférés időpontja másodpercekben 1977 január 1. 0 óra 0 perc 0 másodperctől számolva (epoc).

$t[9] a fájl utolsó módosítása másodpercben epoc-tól számítva.

$t[10] az inod változásának ideje másodpercekben epoc-tól számítva,

$t[11] az aktuális blokkméret,

$t[12] a fájl által lefoglalt blokkok száma.

Nézzük meg, hogy mik is a hozzáférési jogai a `text.txt` fájlnak. Erre természetesen több lehetőség van.
Egyenlőre még nem ismerjük azt a módszert, amellyel bináris adatokat konvertálhatunk, ezért egy bitvizsgálatot használó megoldást ismertetünk.

```
#!/usr/bin/perl
@t=stat("text.txt");
# Others' rights
if($t[2] & 1) { print "ox\n"; }
if($t[2] & 2) { print "ow\n"; }
if($t[2] & 4) { print "or\n"; }
# Group's rights
if($t[2] & 8) { print "gx\n"; }
if($t[2] & 16) { print "gw\n"; }
if($t[2] & 32) { print "gr\n"; }
# User's (owner's) rights
if($t[2] & 64) { print "ux\n"; }
if($t[2] & 128){ print "uw\n"; }
if($t[2] & 256){ print "ur\n"; }
```

A 2. sorban a `@t` tömbbe töltjük a a fájl jellemzőit. A `$t[2]` változó fogja tartalmazni a jogokat[26]. A példában szereplő `text.txt` fájl jogait `UNIX` környezetben az `ls -al text.txt` paranccsal nézhetjük meg. Esetünkben az eredmény:

```
-rw-r--r--    1 hal      hal            37 JUL 17 18:03 text.txt
```

Tehát a tulajdonos olvashatja és írhatja, a tulajdonos csoportja olvashatja, míg a többiek szintén csak olvashatják az állományt.
A program által kiírt lista a jobboldalon látható.

```
or
gr
ur
uw
```

3.3. `C` jellegű fájl kezelés

A perl lehetővé teszi, hogy a `C` nyelvben megszokott `IO` kezelő függvényeket használjuk. Ezek közül az egyik legfontosabb függvény a `C` nyelvből jól ismert `printf` függvény. Használata:

```
printf FILE FORMAT,PARAMETRES;
printf FORMAT,PARAMETRES;
```

[26] Csak az alapvető jogokat vizsgáljuk.

FILE a fájlkezelő.

FORMAT a formátum sztring. Ez a sztring határozza meg, hogy a következő mezőt a függvény hogyan értelmezze. A formátum sztring tartalmazhat **formátum specifikátort** és tetszőleges szöveget. A tesztszőleges szöveg a képernyőre kerül.

A formátum specifikátor a helyének megfelelő változó kiírásának módját határozza meg. A specifikátor szerkezete:

```
%[flags][width][.prec][mod]type
```

A `[]`-ekben található paraméterek nem kötelezőek. Ezek valamilyen módon módosítják a kiírást. Látható, hogy a formátum specifikátorban két "alkotórész" kötelező a `%` jel és a `type` típusazonosító.

`flags` speciális kiírási formákat írnak elő, jelentésük:

`szóköz`	a pozzitív számok előtt az előjel helyére egy szóközt tesz,
`+`	a pozitív számok elé kiteszi a `+` előjelet,
`-`	a kiírást a rendelkezésre álló hely bal oldalára igazítja,
`0`	a jobbra igazítás esetén `0`-ákkal tölti fel a rendelkezésre álló helyet,
#	az oktális számok elé egy `0`-t a hexadecimális számok elé egy `0x`-et tesz.

`width` a minimális kiírási szélességet írja elő. Ha a kiírandó adat nagyobb helyet igényel, mint amit a `width` előír, akkor a teljes adatot kiírja. Ha a `width` nagyobb, mint a kiírandó adat hossza, akkor a kiírandó adatot a rendelkezésre álló hely jobboldalára igazítva írja ki, ha a `flags` nem ír elő mást.

`.prec` értelmezése a kiírandó adat jellegétől[27] függ a következő módon:

lebegőpontos	a tizedes jegyek számát adja meg,
egész jellegű	a kiírás minimális hosszát adja meg,
sztring	a kiírás maximális hosszát adja meg.

`mod` megadja, hogy a kiírt adat milyen pontosságú, ezek:

`l`	hosszú egész kiírását írja elő,
`h`	rövid egész kiírását írja elő,
`V`	perl típusú egész kiírását írja elő,
`v`	egy sztring értelmezése egész vektorként.

`type` a típus leíró. Ezek:

[27] Más nyelven azt írnánk, hogy a típusától, de a perlben a típus értelmezése más, mint mondjuk a C-ben.

`c`	a paraméterben adott számot karakterként írja ki,
`s`	a paramétert sztringként írja ki,
`d, i`	a paramétert egészként írja ki,
`u`	a paramétert előjeltelen, decimális egészként írja ki,
`o`	a paramétert előjeltelen, oktális egészként írja ki,
`x, X`	a paramétert előjeltelen, hexadecimális egészként írja ki,ha `x`, akkor a kilenc feletti számok kisbetűvel kerülnek kiírásra, `X`, akkor a kilenc feletti számok nagybetűvel kerülnek kiírásra,
`b`	a paramétert előjeltelen, bináris egészként írja ki,
`f, F`	a paramétert `1234.5678` lebegőpontos formában írja ki,
`e, E`	a paramétert lebegőpontos formában írja ki, ha `e`, akkor a kiírás formája `123e45`, `E`, akkor a kiírás formája `123E45`,
`g, G`	a paramétert lebegőpontosan írja ki, az `f` és az `e, E` formák közül a rövidebbet választja,
`p`	a perl változó címét írja ki hexadecimális formában,
`D`	`ld`-nek felel meg,
`U`	`lu`-nak felel meg,
`O`	`lo`-nak felel meg.

PARAMETERS a kiírandó paraméterek listája. Ezek a paraméterek skalár változók.

Nos láthatjuk, hogy a `printf` függvény eléggé bonyolult. Nézzünk egy példát a formátumsztring és a kiirandó változók használatára.

```
$a=5;
$b=6;
printf("Az $a erteke %i, a $b erteke %i\n",$a,$b);
```

A képernyőre kiírt szöveg:

```
Az $a erteke 5, a $b erteke 6
```

–

Nézzük részletesen a formátumsztringet.

- A ”`Az $a erteke `” szöveg a perl számára nem értelmezhető formátum specifikátornak, így ez a szöveg változtatás nélkül kiírásra kerül a képernyőre.
- A ”`%i`” az első formátunspecifkátor. A perl megkeresi az első paramétert a paraméterlistából és a típus azonosítónak megfelelően kiírja.
- A ”`, a $b erteke`” szintén szöveg, kiírja.
- A második ”`%i`” formátum specifikátor helyére a második paraméter értéke kerül, jelen esetben a `$b`.
- Végül a `\n` kerül sorra, ami a egy újsort ír elő a képernyőn.

Az olvasó joggal teheti fel a kérdést, hogy mi értelme van egy ilyen ”borzasztóan” bonyolult függvény használatának. Az előző példát a `print` segítségével is könnyedén megoldhatjuk. Valóban. Valóban? Nézzük azt a példát, ahol a fájl hozzáférési jogait szerettük volna kiírni (49. oldal).

```
#!/usr/bin/perl
@t=stat("text.txt");
printf("The rights of text.txt are %.4o\n",$t[2]&0xfff);
```

A jogokat oktális számrendszerben írjuk ki. a `0xfff` egy maszk, ami csak a jogokat jelentő biteket ”engedi át”, a fájl többi jellemzőit ”kitakarja”. A kiírt információ esetünkben:

```
The rights of text.txt are 0644
```

–

Ha az `C` összes függvényét használni szeretnénk használni, akkor a `POSIX` mudult kell használnunk. Erről a **??**. fejezetben beszélünk.

3.4. Bináris adatok kezelése

Ez a fejezet igazából nem tartozik az `IO` funkciók fogalomkörébe, de a bináris adatokat általában akkor kezelünk, ha `IO` műveletet végeztünk, vagy fogunk végezni. A bináris adatok kezelésére két függvényt használhatunk, ezek a `pack` és az `unpack`.

`pack` a változók listáját egyetlen sztringgé rakja össze. Használata:

```
SKALAR=pack(TEMPLATE,LIST);
```

`SKALAR` a skalár változó, ahova az ”öszepakolt” adatok kerülnek.

`TEMPLATE` formátumsztring. Ez határozza meg, hogy az ”összepakolás” hogyan történjen (lásd a táblázatot az `unpack` magyarázata utan).

`LIST` az a skalár paraméterlista, amit össze szeretnénk ”pakolni”.

`unpack` a `pack` utasítás ellentéte. Egy skalár sztringből egy listát készít[28]. Használata:

```
LIST=unpack(TEMPLATE, EXPR);
```

`LIST` ahova az eredmény kerül.

`TEMPLATE` a formátumsztring, amely meghatározza, hogy a ”kicsomagolás” hogyan történjen.

`EXPR` az a skalár, amelyet ”ki akarunk csomagolni”.

A formátumsztring specifikátorai egyszerűbbek, mint a korábban látott `printf` függvény formátum specifikátorai.
A specifikátorok a következők:

`a`	ASCII sztring null karakterekkel feltöltve a fentmaradó hely `pack` esetén, ha szükséges,
`A`	ASCII sztring szóközökkel feltöltve a fentmaradó hely `pack` esetén, `unpack` esetén az záró null és szóköz karakterektől megfosztva,
`b`	bitsztring az `LSB` van elöl,
`B`	bitsztring az `MSB` van elöl,

[28]—ami általában egy tömb—

c	előjeles karakter,
C	előjeltelen karakter,
d	kétszeres pontosságú lebegőpontos szám,
f	lebegőpontos szám,
h	hexa sztring az LSD van legelöl,
H	hexa sztring az MSD van legelöl,
i	előjeles egész,
I	előjel nélküli egész,
L	előjel nélküli hosszú egész,
n	rövid egész a magsabb helyiérték van elöl,
N	hosszú egész a magsabb helyiérték van elöl,
p	sztringre mutató pointer,
P	struktúrára mutató pointer,
s	előjeles rövid egész,
S	előjel nélküli rövid egész,
v	rövid egész a kisebb helyiérték van elöl,
V	hosszú egész a kisebb helyiérték van elöl,
u	uu kódolt sztring,
x	null bájt,
X	egy bájtot vissza,
@	nullával feltöltött a megadott pozícióig pack esetén.

Nézzünk egy egyszerű példát a pack és unpack használatára.

```
1  #!/usr/bin/perl
2  $a=pack("vv",5,6);
3  ($b,$c)=unpack("vv",$a);
4  print "$b,$c\n";
```

A példa 2. sorában az 5 és a 6 értékeket, mint egészeket a $a skalárba csomagolja.

A 3. sorban ezt a becsomagolt értéket kicsomagolja a $b és a $c skalárba.

A 4. sor kiírja az eredményt.

Amennyiben sztringekkel dolgozunk, akkor nyilvánvaló, hogy a sztrig mérete változhat, így adott méretű sztringet becsomagolni csak a hossz megadásával lehet.

```
1  #!/usr/bin/perl
2  $a="Ez egy szoveg";
3  $b=pack("A32v",$a,8);
4  ($c,$d)=unpack("A32v",$b);
5  print "$c,$d\n";
```

A 3. sorban a sztringnek előírjuk a 32 bájt hosszúságot. Mivel az egész mérete állandó ezt a paramétert nem kell hosszal paraméterrel ellátni.

A 4. sorban a becsomagolt adatot kicsomagoljuk. Itt is meg kellett adni a sztring hosszát.

3.5. Néhány gondolat

Ennek a fejezetnek a végén úgy véljük, hogy egy néhány gondolatot még el kell mondanunk a perl `IO` kezeléséről.
Az eddig közölt ismertek csak az alapokat tartalmazzák, a perl `IO` kezelése egy külön jegyzet témája lehetne, de címszavakban felsoroljuk, milyen témaköröket nem érintettünk.

- hálózati kommunikáció kezelése,
- párhuzamos folyamatok és szálak közötti kommunikáció,
- szemaforok kezelése,
- megszakítások és események ”elkapása”.

4. Programok indítása programból

Az operációs rendszerek számos nagyon hasznos szolgáltatással és segédprogrammal támogatják a felhasználót. A programok ezeket a szolgáltatásokat szintén elérhetik, de ez idönként nehézkes. Ezért szinte minden programnyelv lehetővé teszi azt, hogy más programokat illetőleg operációsrendszer parancsokat programból elindíthassunk.

4.1. A `system`, az `exec` függvények és a ‘‘

A perl esetén a legegyszerűbb programindítás a **`system`** függvénnyel történhet. Használata:

```
system(COMMAND);
```

`COMMAND` operációs rendszer parancs, a parancssor vagy az elindítandó program az esetleges paramétereivel.

Példa a `system` használatára:

```
system("ls -al|grep perl.tex > result.txt");
```

A `system` argumentumába kerülhet skalár változó is.
A `system` a meghívásakor felfüggeszti a hívó program futását és elindítja az argumentumban lévő parancsot. Ha az sikeres volt, akkor igaz értékkel tér vissza, ha nem hamissal.
Egy másik programindító függvény az **`exec`** kezelése azonos a `system` kezelésével azonban az `exec` által elindított program nem tér vissza a hívóhoz.
Ha a program indítása sikeres volt a az indító program befejeződik, ellenkező esetben fut tovább. Nézzünk erre egy rövid példát.

```
1  #!/usr/bin/perl
2  print "Launch script\n";
3  exec("prog1");
4  print "prog1 doesn\'t work\n!";
```

A 2. sorban a program udvariasan bemutatkozik.
A 3. sorban elindítja `prog1` programot.
A 4. sorra csak akkor kerül a vezérlés, ha a `prog1` indítása nem sikerült.
Mint azt az előbb említettük az operációs rendszer szolgáltatások indítása nagyon hasznos lehet, de számos esetben nem csak arra az információra van szükségünk, hogy a szolgálatás hiba nálkül lefutott, vagy nem futott le, hanem az általa szolgáltatott adatokra is. Ekkor használjuk a **‘‘** (más néven `backtick`) operátort[29].
A `backtick` oprátor az elindított program a standard kimenetre kiírt adatait adja át egy skalárnak. Használata:

```
SKALAR=‘COMMAND‘;
```

Nézzünk erre egy egyszerű példát. A következő programrészletben egy skalárnak átadjuk az aktuális könyvtár bejegyzéseit hosszú formában.

[29] Sokat vitatkoztunk, hogy ez a két jel valójában mi. Abban maradtunk, hogy operátor.

```
$dir=‘ls -al‘;
```

A `$dir` változó tartalmazza a könyvtár tartalmát, ha szükség van rá a változót a továbbiakban fel lehet dolgozni.

4.2. A `fork`

A `fork` jellegét tekintve más, mint az előző három eljárás. A fork lehetővé teszi, hogy egy önállóan futó folyamatot (programot) tudjunk indítani, miközben az indító program tovább fut. Az indító folyamatot **szülő folyamatnak** az indított folyamatot **gyerek folyamatnak** nevezik.
A `fork` mehívásakor az operációs rendszer másolatot készít a futó folyamtról, vagyis a futó programot újra elindítja. A függvény visszatérési érteke a szülő esetén a gyerek `PID`-jével (**P**rocess **Id**entfication number)[30] tér vissza. Gyerek folyamat esetén a visszatérési érték 0. A program szülő vagy gyerek voltát a programnak kell eldöntenie. Nézzünk erre egy példát.

```
1  #!/usr/bin/perl
2  if($pid=fork())
3     {
4     print "Parent. The child pid=$pid\n";
5     }
6  else
7     {
8     print "Child\n";
9     }
```

A 2. sorban a `$pid` változó megkapja a `fork` visszadott értékét. Ha a folyamat szülő, akkor a gyerek `PID`-jével tér vissza, ami nem 0, következésképpen igaz.
Ha ez gyerek folyamat, akkor az `$pid` értéke 0, vagyis hamis, tehát a program a 6. sorban folytatódik.
A szülönek azonban meg kell várnia a gyerek folyamat befejeződését ellenkező esetben a gyermek folyamat meghatározatlan állapotba kerülhet. Erre a célra a szülő folyamat egy `wait` függvényhívást használhat.

`wait` a gyerek processz befejeződésére vár és visszatér a befejezett processz `PID`-jével, ha nem volt gyerek processz a visszatérési érték -1.

A `wait` függvény használatának megértéséhez vegyük a következő példát.

```
1  #!/usr/bin/perl
2  if($pid=fork())
3     {
4     print "Parent. The child pid=$pid\n";
5     wait();
6     }
7  else
```

[30] folyamtazonosító szám

```
8       {
9       print "Child\n";
10      }
```

Ez a program az előző példától abban különbözik, hogy az ötödik sorba egy `wait` függvényt beszúrtunk. Ez a sor akkor kerül feldolgozásra, ha a kérdéses program szülőként fut. A szülő most — ellentétben az előző programmal — megvárja, hogy a gyerek befejezze a működését, csak azután lép ki.

Megjegyzés: Ha a `wait` nélküli programot futtatjuk, a program ”elakad[31]”, míg a `wait` használatával ez nem következik be.

A `fork` a programozó számára egy rendkívül rugalmas eszközt biztosít. Például nagyon egyszerű ugyanolyan jellegű feladatotokat azonos programmal elvégezhetjük gyakorlatilag egy időben, anélkül, hogy a programunkat erre külön fel kellene készíteni. Tipikusan ilyen jellegű feladatok a szerver alkalmazások, ahol esetenként akár több tíz klienssel is foglakozni kell.

[31] Nem fagy le, mert egy `CTRL+C`-vel kil lehet lépni belőle.

5. A ”DEBUG” üzemmód

Ha a megoldandó feladat bonyolultsága egy adott szintet meghalad, akkor a gondos tervezés ellenére[32] is könnyen előfordulhat, hogy a megírt forráskód szemantikai hibákat tartalmaz, ekkor elkerülhetetlen, hogy valamilyen hibakereső eszközzel kövessük a program futását. A korunkban divatos integrált fejlesztőrendszerek mindegyike biztosít ilyen eszközt, de a manapság ”fapadosnak” minősített környezetek is rendelkeztek valamilyen hasonló eszközzel. A perl szintén biztosít számunkra ilyen ”fapados”, de hatékony eszközt.

5.1. Általános jellemzők

A perl debuggere nem egy külön program, mint az megszokott más rendszerek esetén, hanem egy olyan üzemmód, amely a fordítót arra utasítja, hogy a információkat osszon meg a felhasználóval.
A fordító először lefordítja a programot a debugger számára, amely aztán értelmezi ezt a kódot.
A program debug üzemmód megnyitásához a programot `-d` kapcsolóval kell futtatni. Emellett célszerű a `-w` kapcsolata is, amely engedélyezi a warningokat[33], amelyek a fordítás során a veszélyes kifejezésekre figyelmeztet.
A `-d` kapcsoló használata parancssorból:

```
perl -d prg.pl
```

és a szkripten belülről az első sor:

```
#!/usr/bin/perl -d -w
```

A debug bekapcslása után megjelenik a debug promptja.

```
main::(./prg.pl:3):  $i=0;
  DB<1>
```

A prompt előtt az első végrehajtható sor tartalmát látjuk, amely ebben az esetben a `prg.pl` program `main` moduljának 3. sora, melynek tartalma `$i=0;`.
A kilistázott sor még nem került végrehajtásra.

5.2. A debug üzemmód parancsai

A debug üzemmód karakteres parancsokkal kezelhető. Ebben a felsorolásban csak a legfontosabb parancsokat ismertetjük helyhiány miatt. Amennyiben részletesebb leírást keresünk, akkor UNIX — LINUX környezetben a `perldebug`, `perldebtut` man lapokat célszerű megtekinteni.
A parancsok leírásánál a nem kötelező részeket `[]` jelekkel jelezzük.
A parancsok:

Help parancsok :

h help parancs. Többoldalas helpet kaphatunk a képernyőre. Ha lapozni szeretnénk a helpet a `|h` parancs kiadásával a rendszer aktuális lapozójába (pager) irányítja át a help kimenetét[34].

[32] ”A szoftver mérnöki tevékenység eredménye, és mint ilyen tervezést igényel.” (Végző műszeres hallgató szakdolgozata.)
[33] Talán a figyelmezetetés kifejezés lehetne megfelelő, de a szakmai szleng nem használja.
[34] Az aktuális lapozó általában a `less` vagy a `more`

Megjegyzés: A pager használata azonos minden parancs esetén.

h h kéthasábos rövidített helpet kapunk.

Futtató parancsok :

s [kif] egylépéses üzemmód. A programot a következő kifejezés elejéig hajtja végre, ha a végrehajtott kifejezés szubrutin hívás belép a szubrutinba.

Ha az `kif` kifejezés nem üres, akkor az adott sorban ez végrehajtásra kerül. Ha az `kif` egy szubrutin, akkor a progam belép a szubrutinba és megáll a rutin első kifejezésén. Példa az `kif` használatára.

```
main::(./prg.pl:3):  $i=0;
  DB<1> s $i=1;
```

n [kif] egylépéses üzemmód. A programot a következő kifejezés elejéig hajtja végre. A szubrutin hivásokat végrehajtja, mint egy utasítást. Tehát nem lépteti le a rutint, hanem egyben végrehajtja.

Ha az `kif` kifejezés nem üres, akkor az adott sorban ez végrehajtásra kerül. Ha az `kif` egy szubrutin, akkor a progam belép a szubrutinba és végrehajtja és visszatérés után a övetkező kifejezésen áll meg.

<ENTER> az utolsó `s` vagy `n` parancs ismétlése.

r futtatás az aktuális szubrutin végéig. A visszatérési értéket kiírja, ha a `PrintRet` opció be van állítva. Ez az alapértelmezés.

c [sor|sub] a program folytatása. Lehetőség van egy adott sorban, vagy egy adott szubrutinban egy egyszeri töréspont elhelyezésére.

a sor [parans] egy parancsot ír elő az adott sor végrehajtása előtt. Ha nem adjuk meg a sor számát a következő végrehajtandó elé szúrja be a parancsot. A debugger a következő lépéseket hajtja végre a parancs hatására:

1. ellenőrzi, hogy van-e töréspont az adott sorban,
2. kiírja a sort, ha szükséges, lásd léptetés,
3. végrehajtja a kijelölt parancsot,
4. ha szükséges átadja a vezérlést a felhasználónak, lásd töréspont,
5. végrehajtja a sort.

a [sor] törli a kijelőlt sorra előírt parancsot.

A törli az összes előírt parancsot.

Kereső és listázó parancsok :

/minta az adott minta keresése a programban előre.

?minta az adott minta keresése visszafelé.

l a program sorainak listazása egy képernyőablak méretben.

l kezd+n a program `n+1` sorának listázása a `kezd` sortól kezdve.

l kezd-veg a program sorainak listázása a `kezd` sortól a `veg` sorig.

l sor az adott sor listázása.

l sub a megadott szubrutin sorainak listájának első képernyője.

- az előző képernyő listája.

w [sor] az aktuális vagy a kijelölt sor körüli sorok listája.

S [regkif] listázza azon szubrutinok nevét, amelyek a reguláris kifejezésnek megfelelnek.

p kif kiírja a `kif` kifejezést. A parancs a perl `print` függvényét használja.

x kif végrehajtja a `kif` kifejezést és kiírja az eredményt. Az egymásba ágyazott kifejezéseket rekurzív módon listázza.

Töréspont kezelő parancsok :

b [sor][felt] töréspont elhelyezése `sor` által megadott sorban.

Ha a `felt` feltétel meg van adva, akkor a töréspontonon a program csak a feltétel teljesülése esetén áll meg. Például:

```
b 5 $i<5
```

Ha nincs megadva a `sor` paraméter, a töréspont az éppen végrehajtandó sor elé kerül.

b sub [felt] töréspontot helyez a `sub` által megnevezett szubrutin első sora elé. A `felt` feltétel kezelését lásd az előző pontnál.

Amennyiben a szubrutin neve referencia a feltétel kezelése nem támogatott.

b postpone sub [felt] töréspontot helyez el a megnevezett szubrutin első sora mögé.

b load fajlnev töréspontot helyez el a `fajlnev`-vel azonosított fájl első végrehajtandó sora elé.

A fájl elérési útjának szerepelnie kell az `%INC` hash-ben.

b compile sub a `sub` által megadott szubrutin lefordítása után töréspontot helyez el a rutin első sora elé.

d [sor] az adott sorban lévő töréspontot törli.

D az összes installált töréspontot törli.

L listázza az összes töréspontot.

Watch kezelés :

W kif egy globális ”watch” kifejezést állít be[35],

W az összes ”watch” kifejezés törlése,

V [csomag[valt]] `csomag` csomagban a `valt` változó[k] értékét listázza,

X [valt] az aktuális csomagban listázza a `valt` változó[k] értékét listázza,

[35] A watch egy változó értékét figyeli, ha a program valahol megáll, mert töréspontot talál, vagy a programot léptetjük, a kiválasztott változó értéke kiratható.

Debug vezérlő parancsok :

- **q** kilépés a debuggerből,
- **R** a debugger újraindítása, a beállítások, mint watch, töréspont, stb. esetleg elveszhetnek.

6. Objektum orientát programozás perlben

A fejezet szigorúan a perl objektum orientált programozásával (OOP) foglalkozik. A fogalmakat csak olyan mélységig tárgyaljuk, amennyire ezek az adott környezethez szükségesek. Nem megyünk bele az OOP elméletébe, módszertanába és a különböző absztrakciós szintekbe. Viszont elegendő ismeretet szeretnénk közölni a már előre megírt objektum ”csomagok” használatához.

6.1. Alapvető ismeretek, definíciók

A perl OOP filozófiájának megértéséhez néhány fogalmat célszerű tisztázni. Ezért a következő definíciók okvetlenül szükségesek[36].
Az objektum orientált módszertan három olyan tulajdonságot alkalmaz, amelyet az előző módszertanok nem alkalmaztak. Ezek:

1. öröklődés. Ami azt jelenti, hogy az objektum osztályok egymás tulajdonságait öröklik.
2. egységbezárás. Az objektum tartalmazza a hozzá tartozó adatokat és metódusokat.
3. többalakúság. Azonos néven több függvényt lehet megvalósítani.

Ezeknek egy részét a perl nyelv is alkalmazza, ha nem is a klasszikus értelemben.

Megjegyzés: *A 2002-es perl konferencián elhangzott, hogy a perl egy olyan OOP nyelv, amely nem is objektum orientált.*

Definíció: Osztálynak nevezzük az egy adott problémacsoporthoz rendelt szubrutinok halmazát és az ezekhez leírt változó területet.

Az osztályt úgy is tekinthetjük, mint egy általános leírást, amely meghatározza az osztályhoz tartozó objektumok általános alakját.

Definíció: Az objektum az osztály egy példánya, vagyis adott memória területet foglal és hivatkozni lehet rá.

Definíció: Metódusnak nevezzük az objetumhoz rendelt valamely függvényt.

Definíció: Osztályhoz rendelt metódus az a függvény, amely az osztályhoz tartozó összes objektumra vonatkoznak.

Definíció: Objektumhoz rendelt metódus az a függvény, amely nem az összes osztályhoz tartozó objektumra, hanem csak egy jól meghatározott objektumra jellemző.

Definíció: Konstruktor az a metódus, amely egy objektumot hoz létre. A konstruktor lehet bármely függvény, de az esetek többségében ez a metódus a `new` függvény.

[36] Tényleg szükségesek, bocs.

Definíció: Destruktor az metódus, amely egy korábban létrehozott objektumot megszünteti.

A destruktor meghívása egy `DESTROY` nevű függvény meghívásával történik.

6.2. Az OOP tulajdonságai perlben

A perl nyelvben az objektum egy olyan referencia, amely ”tudja”, hogy hol helyezkedik el a tárban. Ez a referencia célszerűen egy aszociációs tömb, amely ezek után az objektum saját szimbólum táblájaként szolgál.
Az objektum osztály egy közönséges `package`. Egyetlen utalás nincs arra, hogy ez egy objektum osztály. Nézzünk egy példát!

```
1  package RECTANGLE;
2  sub new
3     {
4     my ($type,$w,$h)=@_;
5     my $self={};
6     $self->{width}=$w;
7     $self->{height}=$h;
8     return bless($self, $type);
9     }
10 sub area
11    {
12    my $self=shift;
13    my $area=$self->{width}*$self->{height};
14    return $area;
15    }
```

Ez tehát egy osztály definíció, amelyben a téglalap (`RECTANGLE`) osztály írjuk le. Az osztály belső változói a szélesség `width` és a mágasság `height`, metódusai a `new`, amely most konstruktorként szerepel és az `area` függvény, amely a téglalap területét számítja ki.
Nézzük végig a fenti programrészletet.

- Az első sorban megadjuk az osztály nevét (vagy típusát ahogy tetszik).
- A negyedik sorban a `$type` lexikus változó fogja az osztály nevét tartalmazni.
- Az ötödik sorban a `$self` változó megkapja a package szimbólum táblájaként szereplő hash címét.
- A hatodik és a hetedik sorban az átadott inicializáló változók értékei kerülnek a szimbólum táblába.
- A nyolcadik sorban a `bless` függvénnyel készítjük el az objektumot[37]. Ezek után a `return` segítségével kilépünk a `new` függvényből.

[37] Néha ovasható, hogy az objektumot megáldjuk.

- Az `area` függvény a tizenkettedik sorban a `$self` lexikus változó fogja az osztály nevét tartalmazni — a default tömbből sifteljük —. A függvény többi része már semmi újat nem tartalmaz.

Az elkészített osztályt úgy használhatjuk, hogy objetumot készítünk belőle, vagyis példányosítjuk. A következő programrészlet az előbb megírt `RECTANGLE` osztály használatára mutat be példát.

```
16  package main;
17  $obj=RECTANGLE->new(5,10);
18  print 'Area of $obj='.$obj->area()."\n";
```

- Az tizenhatodik sor egy új csomagot nyit meg, és lezárja az előzőt.
- Az `$obj` változó lesz annak objektumnak a címe, amelynek felépítése megfelel a `RECTANGLE` osztálynak, továbbá átadtunk neki két paramétert.
- A tizennyolcadik sorban használjuk az objektum `area` függvényét és az eredményt rögtön ki is iratjuk a képernyőre.

A `bless` függvény használata a következő:

```
bless REF, CLASSNAME
bless REF
```

`REF` ez annak az objektumnak a referenciája, amelyet létrehoztunk a `CLASSNAME` package-ben,

`CLASSNAME` a package neve.

Figyelmeztetés! Az osztály nevét nem célszerű csupa kisbetűvel írni, mert a perl csak kisbetűs csomagneveket használ, ezért a névütközések elkerülésére javasolt a kis—nagy betüs csomagnevek használata[38].

Amennyiben a `CLASSNAME` paraméter hiányzik, akkor a perl az aktuális package nevét tekinti az osztály nevének.
Amint látjuk a hash használatával az OOP egyik alaptulajdonságát, az egységbezárást sikerült biztosítanunk. Ha egy objektumot létrehoztunk, akkor az a belső változóit tárolja. A példánkban a `width` és a `height` értékeket nem kellett megadnunk az `area` függvénynek, mert azt már a konstruktorban elintéztük.
A perl biztosítja az öröklődést is erre a célra az `@ISA` tömböt használja fel, de csak a metódusok öröklését biztosítja. Ha egy metódust használunk akkor a következő mechnizmus indul el:

1. az adott package-ben keres,
2. az `@ISA` tömbben felsorolt csomagokban keres,
3. az `AUTOLOAD` nevű függvény hívásával próbálja a hiányzó metódust megtalálni,
4. végül egy `UNIVERSAL` nevű modulban keres.

[38] Én mindig a nagybetűs neveket használok.

Természetesen csak akkor lép a köetkező lépésre, ha az előző nem teljesült.

Az `@ISA` egy tömb ezért menet közben is lehet módosítani, így az öröklődési sor menetközben változtatható[39].

Természetesen szükség van a változó területek öröklésére is, nézzünk erre egy példát!

```
1  package PARENT;
2  sub new
3     {
4     my $type=shift;
5     my $self={};
6     $self->{'a'}='A';
7     return bless($self,$type);
8     }
9  package CHILD;
10 sub new
11    {
12    my $type=shift;
13    my $self=PARENT->new();
14    $self->{'b'}='B';
15    return bless($self,$type);
16    }
17 package main;
18 $ch=CHILD->new();
19 print 'a='.$ch->{'a'}."\n";
20 print 'b='.$ch->{'b'}."\n";
```

A példa két dolgot is bemutat, az első az ős osztály változóterületének használatát.

A második lényeges dolog, hogy a példából látható, hogy a leszármazott konstruktorának lefutásakor az ős konstruktora automatikusan nem fut le, hanem a tizenharmadik sorban nekünk kell ezt meghívni. Ekkor kapjuk meg azt a referenciát, amely már az ősben is szerepelt.

Az OOP harmadik tulajdonságát a többalakúságot a perl közvetlenül nem támogatja.

Az osztályhoz rendelt metódusok létrehozása a következő módon történik:

```
package SOMETHING;
sub iam
   {
   my $name=@_;
   print "My name is: $name\n";
   }
```

Hívása:

```
package main;
SOMETHING->iam();
```

[39] Hmm, szép, de veszélyes.

A következő példa azt mutatja be, hogy ugyanezt a feladatot hogyan oldhatjuk meg objektumhoz rendelt metódussal.

```
package SOMETHING;
sub new
   {
my $type=shift;
   my $self={};
   bless($self);
   return ($self,$type);
   }
sub iam
   {
   my $self=shift;
   print $self."\n";
   }
```

Hívása:

```
package main;
$obj0=SOMETHING->new();
$obj1=SOMETHING->new();
$obj0->iam();
$obj1->iam();
```

A kiírt eredmény nem igazán emészthető, mivel címeket kapunk vissza. Az azonban jól látható, hogy a két cím nem azonos.
A továbbiakban kész objektum osztályok használatát mutatjuk be.

7. TK

A TK egy olyan objektum osztály, amely lehetővé teszi, hogy az adott környezetben grafikus felületet hozzunk létre. A TK eredetileg nem a perl, hanem a TCL nyelv része volt. A fejlesztők úgy gondolták, hogy a perl hatékonyságát célszerű lenne összeházasítani a TCL/TK grafikus képességeivel.
A perl/TK a következő elemeket tartalmazza:

MainWindow a grafikus alkalmazása alapja, a fő ablak,

Button nyomógomb,

Canvas a rajz terület (a festővászon),

Checkbutton jelölőnégyzet,

Entry szövegbeviteli mező,

Frame összefüggő terület,

Label felirat,

Listbox lista készítő objektum,

Menu menü objektum,

Menubutton a menülécben egy menü létrehozása,

Message üzenet kiírása a képernyőre,

Radiobutton rádiógomb létrehozása,

Scale csúszka,

Scrollbar gördítősávot létrehozó objektum,

Text szerkeszthető szövegmezőt létrehozó objektum,

Toplevel új ablak létrehozása.

A TK használatához be kell töltenünk a `Tk.pm` modult, tehát a program elején szerepelnie kell a következő sornak:

```
use Tk;
```

Ezután már használhatjuk a `Tk.pm` modul objektumait.

7.1. `MainWindow`

Ahhoz, hogy egy grafikus alkalmazást hozzunk létre először egy ablakot kell létrehoznunk. Ez egy `MainWindow` típusú objektum létrehozását jelenti, ami ezek után a grafikus elemek — más néven (és pontatlanul) grafikus widgetek — gyökere lesz[40]. Alapesetben ez egy nagyon egyszerű lépés.

```
$mw=MainWindow->new();
```

Azonban ez nem elegendő, a grafikus felületet létrehozó programrészlet egy `MainLoop()` függvénnyel kell lezárni. Tehát általános esetben egy grafikus felhasználást a következő felépítésű:

```
use Tk;
    ⋮
$mw=MainWindow->new();
    ⋮
további grafikus elemek
    ⋮
MainLoop();
    ⋮
további programrészek
    ⋮
```

A `MainWindow` használata:

```
$mw=MainWindow->new();
```

Az `MainWindow` típusú objektum paraméterezése az objektum függvényeivel történik.
Ezek közül a legfontosabbak:

`minsize` az ablak minimális méretét határozza meg. Használata:

```
$mw=MainWindow->new();
$mw->minsize(qw(50 40));
```

Ez egy 50×40 pixel minimális méretű ablakot ír elő.

`maxsize` az ablak maximális méretét határozza meg.

`title` az ablak nevét határozza meg, amely az ablakkezelőtől függően a fejlécben jelenik meg.

`configure` a további konfigurációt állítja be. Ezeket a paramétereket a függvénynek hash formátumban adjuk át.

`-background` az ablak háttérszínét adja meg. Ezt lehet névvel, ha ez lehetséges, de lehet hexa értékekkel is.

`-forground` az ablak előtérszínét adja meg névvel, vagy értékkel.

`-borderwidth` az ablak keretének vastagságát adja meg pixelben.

[40] Oszlatandó a homályt Szóval az összes többi komponens ebből származik, neki nincs szülő objektuma.

-width az ablak szélességét adja meg.

-height az ablak magasságát adja meg.

-relief az ablak felület besüllyesztését, illetőleg kiemelését írja elő. Ha ez a paraméter

'raised' kiemelkedő ablakfelület,

'sunken' besüllyesztett ablakfelület,

'flat' szintben lévő ablakfelület,

'ridge' kiugró ablakkeret,

'groove' "bevésett" ablakkeret,

'solid' sima ablakkeret.

Nézzük példának a következő ablak konfigurációt:

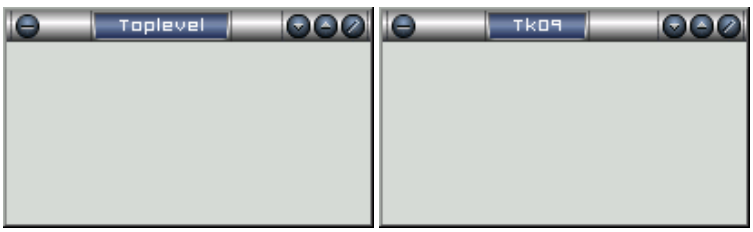
```
 $mw=MainWindow->new();
$mw->maxsize(qw(200 80));
$mw->tile("Hello");
$mw->configure(-bordewidth=>3,
               -background=>'darkgray',
               -foreground=>'white',
               -relief=>'ridge');
```

7.2. ábra: MainWindow

7.2. `Toplevel`

A `Toplevel` widget egy új ablak, amely a `MainWindow` objektumból származik. A következő programrészlet egy ilyen objektum létrehozását mutatja be.

```
$mw=new MainWindow();
$mw->maxsize(qw(200 100));
$tl=$mw->Toplevel();
$tl->maxsize(qw(200 100));
MainLoop();
```



7.3. ábra: Toplevel

7.3. `Frame`

A `Frame` objektum arra szolgál, hogy egy ablakon belül egy önálló keretet hozzunk létre. A `Frame` az ablak — esetleg egy másik keret — leszármazottja A következő programrészlet erre mutat be példát.

```
1  $mw=MainWindow->new();
2  $mw->minsize(qw(200 100));
```

```
3
4  $fr=$mw->Frame();
5
6  $fr->configure( -borderwidth => 3,
7                  -height      => 50,
8                  -relief      => 'raise');
9
10 $fr->pack( -side => 'bottom',
11            -fill => 'x');
12
13 MainLoop();
```

Az első sorban létrehozunk egy ablakot.
A negyedik sorban létrehozunk egy `Frame` objektumot, amely az első sorban létrehozott `MainWindow` típusú objektum leszármazottja.
A hatodik sorban konfiguráljuk az `$fr` keretet.
A tizedik sorban a `pack` függvénnyel a rendezzük az ablakot és kirakjuk a keretet.
A `pack` függvénnyel a **??** fejezetben foglalkozunk
A létrehozott `Frame` úgy viselkedik, mint egy ablak, lehetőséget nyújt arra, hogy a keretbe új widgeteket helyezhetünk el.

7.4. ábra: Frame

A `Frame` konfiguráló paraméterei:

`-background` `-borderwidth` `-height` `-width` `-relief` lásd `MainWindow`.

`-colormap` specifikálja azt a szín térképet, amit a keret a továbbiakban használ.

`-container` logikai változó. Ha ennek a paraméternek az értéke igaz, akkor keret úgynevezett konténerként használható. Ez azt jelenti, hogy a keretben beágyazott alkalmazás futtatható.

7.4. `Button`

Nyomógomb widgetet helyez el az ablakban, vagy a keretben. A gomb úgy biztosítja a kívánt szolgáltatást, hogy meghív egy szubrutint. Ez a szubrutin végzi el az adott feladatot. A következő programrészlet erre mutat példát.

```
1  $mw=MainWindow->new();
2  $mw->minsize(qw(80 40));
3
4  $b=$mw->Button(-text    => 'exit',
5                 -command => sub { exit; }
6                )->pack();
7  MainLoop();
```

A program az oldalt látható gombot hozza létre az ablakban.
Az ötödik sorban egy úgynevezett névtelen szubrutint hívunk meg. Ez a szubrutin általában nagyon rövid. Tulajdonképpen ez a rutin lehet hosszabb, de akkor a program olvashatósága romlik[41], illetőleg más helyről nem érhetjük el.
Amennyiben nem akarunk névtelen szubrutint használni, akkor a `-command` paraméternek a meghívandó szubrutinnak a címét vagyis a referenciáját kell átadnunk.

7.5. ábra: Button

A `Button` widget paraméterei a következők:

`-borderwidth` `-background` `-foreground` `-relief` `-height` `-width` lásd `MainWindow`

`-activebackground` a gomb háttere erre a színre vált, ha a cursor a gomb felett van,

`-activeforeground` a gomb felirata erre a színre vált, ha a cursor a gomb felett van,

`-padx` az x irányban plusz területet ad meg az ”optimálishoz” képest,

`-pady` az y irányban plusz területet ad meg az ”optimálishoz” képest,

`-state` a gomb státuszát lehet parancssorból beállítani. Ezek lehetnek:

- **`normál`** a gomb default állapota,
- **`active`** a gomb ugyanabba az állapotba kerül, mintha a cursor felette lenne,
- **`disabled`** a gomb nem aktiválható

`-command` a gomb által hívott szubrutin címe, vagy egy névtelen függvény, lásd az alfejezet elején látható példát.

`-image` a gombra helyezhető képet adja meg.

`-bitmap` a gombra helyezhető bittérképes képet adja meg.

A gomb függvényei a következők:

`flash` a rutin meghívása után a gomb néhányszor színt vált az aktív és a normál háttér színeit felhasználva.

`invoke` a `-command` opcióban megadott szubrutint hívja meg.

7.5. `Checkbutton`

A `Checkbutton` widget egy jelölő négyzetet tesz ki a képernyőre. A négyzet állapotát egy változóban adja vissza.
A következő programrészlet a `Checkbutton` használatát mutatja be.

```
$cb=$mw->Chechkbutton( -variable => \$a);
$cb->pack();
```

A widget paraméterei a következők:

-borderwidth -background -foreground -relief -height -width lásd a `MainWindow` leírásánál.

-activebackground -command -state lásd a `Button` leírásánál.

-variable a widget állapotát visszaadó változó referenciája.

-onvalue a kijelölt változó értéke bekapcsolt jelölőnégyzet esetén.

-offvalue a kijelölt változó értéke kikapcsolt jelölőnégyzet esetén.

-indicatoron logikai változó. Igaz esetben a jelölőnégyzet hagyományos, hamis esetben a jelölőnégyzet megjelenése az ablakkezelőtől függ.

A `Checkbutton` widget függvényei a következők:

deselect a widgetet kikapcsolt állapotba helyezi.

select a widgetet bekapcsolt állapotba helyezi.

toggle a widget állapotát megváltoztatja. Ha az ki volt kapcsolva, akkor bekapcsolja, a bekapcsolt állapotot kikapcsoltra állítja.

flash lásd a `Button` leírásánál.

invoke lásd a `Button` leírásánál.

7.6. Radiobutton

A `Radiobutton` widget egy (értsd egy darab) rádiógombot hoz létre. Ha több gombból álló csoportot akarunk létrehozni, akkor ezeket az egyedi gombokat kell összekapcsolnunk. Ez az összekapcsolás a kijelölt változón keresztül történik, lásd a következő programot.

```
#!/usr/bin/perl
use Tk;
$a;
$mw=MainWindow->new();
$mw->minsize(qw(80 40));
$r1=$mw->Radiobutton(-variable=> \$a,
                     -value   => 1)->pack();
$r2=$mw->Radiobutton(-variable=> \$a,
                     -value   => 2)->pack();
$r3=$mw->Radiobutton(-variable=> \$a,
                     -value   => 3)->pack();
$r4=$mw->Radiobutton(-variable=> \$a,
                     -value   => 4)->pack();
$r1->select();
MainLoop();
```

A listából látható, hogy az `$a` az a változó, amelyen keresztül az egy csoportba tartozó gombokat összekapcsoljuk.
Láthatjuk, hogy a program futása során mindig csak egy gomb kerülhet kiválasztott állapotba. Ekkor az `$a` értéke az adott gombnak megfelelő értéket veszi fel.
Az `$r1` gombot a `select` függvénnyel kiválasztott állapotúra állítjuk. Erre azért van szükség, mert kiinduláskor az összes gomb ”kiválasztatlan” állapotban lenne.


7.6. ábra: Radiobutton

A `Radiobutton` widget paraméterei és függvényei megegyeznek a `Checkbutton` paramétereivel és függvényeivel.

7.7. `Label`

Ez a widget egy feliratot tesz az ablak megfelelő helyére.

```
$l=$mw->Label( -text => ’Hello’ );
$l->pack();
```

A label paraméterei:

`-borderwidth` `-background` `-foreground` `-relief` `-height` `-width` lásd `MainWindow` widget.

7.8. `Entry`

Az `Entry` widget szerepe az, hogy szöveges információt adhassunk át a kezelői felületről a program számára.

`-borderwidth -background -selectbackground -foreground -relief` lásd a `MainWindow` leírásánál.

`-width` a szövegbeviteli mező szélességét adja meg karakterekben. A fizikai mezőszélességet az alkalmazott karakterkészlet méretének átlagos szélessége határozza meg.

`-show` ha ez a paraméter be van állíva, akkor az szövegbeviteli mező a megadott karakterrel lesz feltöltve. Például:
`$e=$mw->Entry(-width=>5,show=>’*’);`


7.7. ábra:
. . . `-show=>’*’` . . .

`-state` a widget státuszát határozza meg. Ez lehet:

`normal` normál üzemmód,

`disabled` tiltott üzemmód A widget nem reagál a vezérlésre, még akkor sem, ha ki van választva.

Az `Entry` widget legfontosabb metódusai:

`get` visszaadja a szövegbeviteli mező tartalmát.

```
$a=$e->get();     # Az $e az Entry widget
```

configure a widget konfigurálását végzi a fent olvasható opciók segítségével.

delete(first,last) a szövegmező egy tartományát törli, a `first` paraméter adja meg azt az első törlendő karakter indexét. A `last` paraméter annak a karakternek az indexét adja meg, amely közvetlenül az utolsó törölt karakter után van.

A legelső karakter indexe 0.

insert(index,string) az adott index pozíciótól beszúrja a megadott sztringet.

7.9. Menubutton

A `Menubutton` widget legördülő menü létrehozását teszi lehetővé. A widget a képernyőn egy gombot jelenít meg, amelyet, ha elnyomunk egy menü ugrik elő. A widget paraméterei:

-borderwidth -background -foreground -selectbackground -foreground -relief lásd a `MainWindow` leírásánál.

-state lásd `Button` leírását.

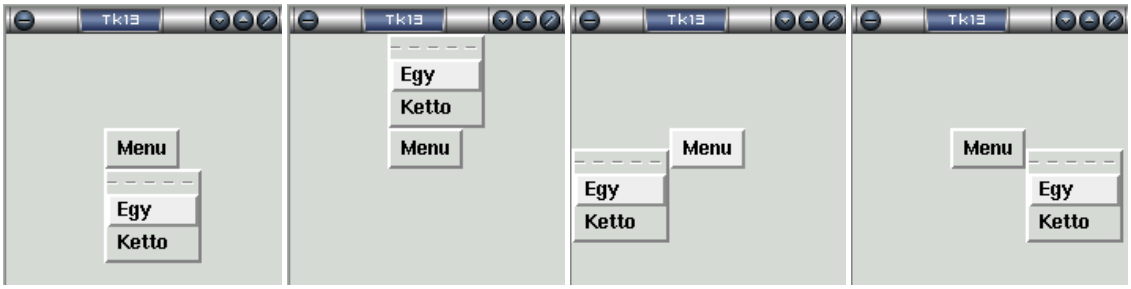
-direction a menü elhelyezését írja elő. A lehetőségek:

below a menüt a gomb alá helyezi, ha lehetséges,

above a menüt a gomb fölé helyezi, ha lehetséges,

left a menüt a gomb bal oldalára helyezi, ha lehetséges,

right a menüt a gomb jobb oldalára helyezi, ha lehetséges.



7.8. ábra: `Menubutton` elrendezések `below, above, left, right`.

A widget függvényei:

command meghatározza a gomb által hívott menü egy elemét. A `command` úgy viselkedik, mint egy önálló widget, ezek közül a legfontosabb paraméterek:

-background -selectbackground -foreground -selectforeground

-command annak a szubrutinnak a referenciáját adja meg, amelyet az adott menüpont hív.

A következő program a 7.8. ábrán látható menüt valósítja meg.

```
1  #!/usr/bin/perl
2
```

```
3  use Tk;
4
5  $mw=MainWindow->new();
6  $mw->maxsize(qw(175 160));
7
8  $m=$mw->Menubutton( -text => 'Menu',
9                      -relief => 'raise',
10                     -direction => 'below'
11                     )->place(-x => 62, -y => 60);
12 $m->command( -label => 'Egy',
13             -command => \&d );
14 $m->command( -label => 'Ketto',
15             -command => \&d );
16 MainLoop();
17
18 sub d
19    {
20    my $a=pack('C',7);
21    print $a;
22    }
```

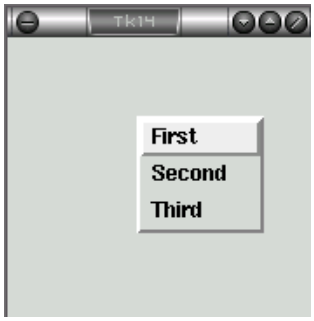
A 12. és a 14. sorban hozunk létre egy - egy menüpontot, amelyek a 18. sorban lévő `d` nevű szubrutint hívják meg. Természetesen nem csak azonos függvényt hívhatunk.
A szubrutin csak egy rövid hangjelzést ad.

7.10. `Menu`

A `Menu` widget menü létrehozását teszi lehetővé.
A `Menu` widget egy új un. Toplevel ablakot nyit, tehát szinte minden olyan szolgáltatást tud, amit egy "normális" ablak tud. Miután ismertetése szinte parttalan lenne ezért inkább egy példát nézünk végig részletesen.

```
1  #!/usr/bin/perl
2  use Tk;
3  $mw=MainWindow->new();
4  $mw->maxsize(qw(175 160));
5
6  $m=$mw->Menu( -tearoff => 0,
7                -menuitems =>
8                [
9                 [ Button => "First",  -command => \&d ],
10                [ Button => "Second", -command => \&d ],
11                [ Button => "Third",  -command => \&d ]
```

```
12              ]);
13
14  $mw->bind("<Button-3>" => sub
15                            {
16                            $m->Popup( -popover   => "cursor",
17                                       -popanchor => 'nw');
18                            } );
19  MainLoop();
20
21  sub d
22     {
23     my $a=pack('C',7);
24     print $a;
25     }
```



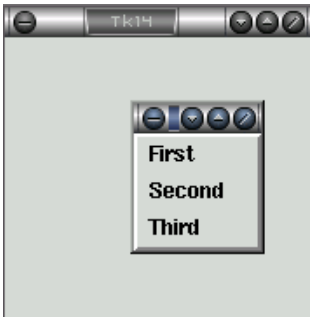
7.9. ábra: Popup menü

A példa egy popup menüt hoz létre, amely az egér jobb gombjának lenyomására előugrik.
A cursor további mozgatásával az egér gombját nyomva tartva válogathatunk a bejegyzések között. Mikor elengedjük az egér jobb gombját az a menüpont kerül kiválasztásra, amely felett a cursor áll.
A példa hasonlóan `Menubotton` példában látottakhoz szintén ugyanazt a szubrutint hívja. Ennek oka csak és kizárólag hely-takarékosság, a valóságban nincs sok értelme.

A 6. sorban hozzuk létre a menü widgetet és beállítjuk a menü viselkedését. Jelen esetben ez `tearoff => 0`, ez azt jelenti, hogy a felugró menü nem mozdítható el a helyéről.
A 7. sorban beállítjuk a létrehozandó menüpontokat beágyazott névtelen tömbök segítségével (2.6.3. fejezet).
A 14. sorban egy eseményt kötünk a menü megjelenéséhez. A későbbiekben az eseménykezelésről lesz szó.
A 16.-17. sorban az esemény meghívja az `$m` objektum `Popup` metódusát.
Változtassuk meg a 6. sort az alábbi módon. Ekkor a menü "leszakíthatóvá" válik. Ha leszakítjuk, akkor egy új ablak jön létre, amely a menü gombjait tárolja.

```
6   $m=$mw->Menu( -tearoff => 0,
7                 -menuitems =>
8                 [
```

7.10. ábra: ”Leszakítható” popup menü



7.11. ábra: ”Leszakított” popup menü

7.12. ábra: Menü rádiógombbal

A 7.10. ábra a leszakítható menüt mutatja. A menü felső sorában látható a ”perforáció”, ha e fölött engedjük el az egér jobb gombját a menü leszakad. Az eredményt a 7.11. ábrán látjuk.

Ennek a widgetnek érdekessége az is, hogy nem csak gombokat adhatunk meg, hanem más elemeket is. Példaként ágyazzunk a menübe rádiógombokat. A menü a 7.12. árán látható.

A program a következő:

```
6  $m=$mw->Menu( -tearoff => 1,
7                -menuitems =>
8                [
9                 [ Button => "First",  -command => \&d ],
10                [ Button => "Second", -command => \&d ],
11                [ Button => "Third",  -command => \&d ],
12                [ Radiobutton => "Hipp", -variable => \$a, -value => 1 ],
13                [ Radiobutton => "Hopp", -variable => \$a, -value => 2 ]
14               ]);
```

A 12. és a 13. sor hozza létre a két rádiógombot[42].

A menü másik gyakori megjelenési formája a pull-down menü Ilyet már tudtunk a `Menubutton` widgettel is készíteni.

Nézzük a következő példát!

```
1  #!/usr/bin/perl
2  use Tk;
3  $mw=MainWindow->new();
4  $mw->maxsize(qw(175 190));
5  $mw->configure(-menu => my $mb = $mw->Menu);
6  my $file = $mb ->cascade(-label => ’File’);
7  $file->command(-label       => ’New’,
8                 -accelerator => ’Ctrl-n’,
9                 -underline   => 0,
10                -command     => \&d  );
```

[42]Mindenesetre érdekes és főleg szokatlan.

```
11  $file->command(-label       => 'Open',
12                  -accelerator => 'Ctrl-o',
13                  -underline   => 0,
14                  -command     => \&d );
15  $file->command(-label => 'Save',
16                  -accelerator => 'Ctrl-s',
17                  -underline   => 0,
18                  -command     => \&d );
19  $file->command(-label       => 'Save as',
20                  -accelerator => 'Ctrl-a',
21                  -underline   => 1,
22                  -command     => \&d );
23  $file->separator;
24  $file->command(-label       => 'Exit',
25                  -accelerator => 'Ctrl-x',
26                  -underline   => 1,
27                  -command     => \&d );
28  MainLoop();
29
30  sub d
31     {
32     print "\a";
33     }
```

A példa létrehoz egy egyszerű file menüt. Természetesen ez csak a képernyő kezelése, nem a teljes feladat kidolgozása.
Az első öt sort már nagyon jól ismerjük, ezért itt csak azt említjük meg, hogy grafikai okokból magasabb az ablak, mint az előzőekben.
Az ötödik sorban létrehozunk egy menüt, amelynek a neve `$mb` (menubar) lesz.
A hatodik sorban ezt kaszkádosítjuk.

Megjegyzés: Szóval az `$mb` egy menü, amely a fő menüpontokat tárolja ezt kell kaszkádosítani, tehát a fő menüpontokhoz újabb menüket kell rendelni, hogy az alfunkciókat elérhessük.

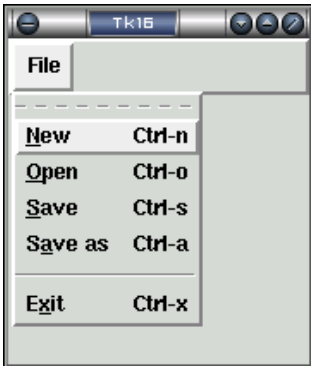
A hetedik sorban megkezdjük az almenüpontok definiálását a `-command` paraméter segítségével. Ugyanebben a sorban található a `-label`, ami megadja a menüpont nevét.
A nyolcadik sorban megadjuk, hogy a gyorsító billentyű-t, ami a New menüpont hívásához tartozik. Ez jelen esetben a `Ctrl-c` (kontrol c).
A kilencedik sorban megadjuk, hogy melyik karaktert húzzuk alá a menüpont nevében. Ez most jelen esetben a nulladik, tehát a `'N'`-t.
A tizedik sorban lévő `-command` paraméter megadja az un. callback[43] rutin címét.

[43] Ez az a szubrutin, amit meghívunk a gombbal.

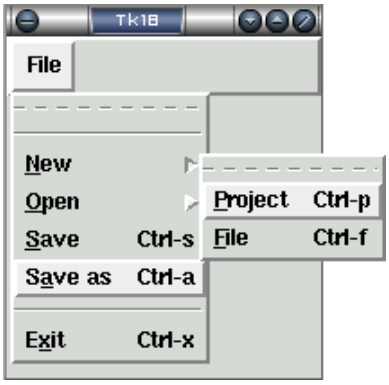
Megjegyzés: Itt az egyszerűség kedvéért szintén ugyanazt a `&d` függvény hívjuk, kivétel az `Exit` menüpont.

A huszonharmadik sorban található egy `separator` bejegyzés, ez egy vízszintes vonalat húz a menüpontok közé.

7.13. ábra: Pull-down menü menüléce



7.14. ábra: A ”lehúzott” menü



7.15. ábra: A kaszkádosított menü

A menü természetesen további menükre bontható a `cascade` funkció használatával. A következő programrészlet a 7.15. ábrán látható menüszerkezet létrehozására szolgál.

```
my $new=$file->cascade(-label       => ’New’,
                       -accelerator => ’Ctrl-n’,
                       -underline   => 0 );

                                  ⋮

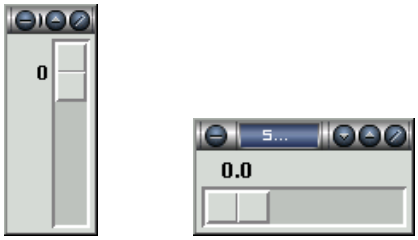
$new->command(-label       => ’Project’,
              -accelerator => ’Ctrl-p’,
              -underline   => 0,
              -command     => \&d );
$new->command(-label       => ’File’,
              -accelerator => ’Ctrl-f’,
              -underline   => 0,
              -command     => \&d );

                                  ⋮
```

7.11. `Scale`

A `Scale` widget egy csúszkát hoz létre a képernyőn. A csúszka lehet függőleges és vízszintes irányú. A csúszka helyzete egy előre konfigurált változóban jelenik meg.
Az alapvető paraméterek megegyeznek a korábbiakban leírtakkal. A widget speciális paraméterei a következők:

-orient ez a paraméter adja meg, hogy a csúszka vízszintes, vagy függőleges legyen, ha értéke ”h”, akkor vízszintes, ha ”v”, akkor függőleges.



7.16. ábra: Függőleges és vízszintes elrendezés

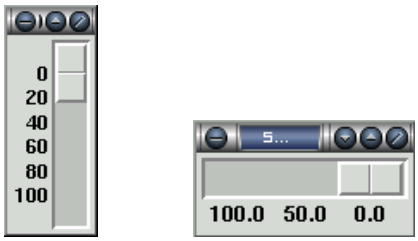
-width a csúszkapálya szélességét adja meg (pixelben).

-length a widget hosszát adja meg.

-sliderlength a csúszka (és nem a widget) hosszát adja meg képernyő egységekben (pixelekben).

-showvalue egy logikai változó, amely meghatározza, hogy a változó értékét jelezze-e a képernyőn, vagy ne.

-tickinterval egy pozitív szám, amely megadja, hogy a csúszka melletti, vagy alatti skála mekkora felbontással szerepeljen. Ha a változó értéke 0 a widget nem készít skálát.



7.17. ábra: `Tickinterval` függőleges és vízszintes elrendezés esetén

-sliderrelief megadja, hogy a csúszka milyen jellegű legyen a `MainWindow - relief`-nél használt paraméterek használhatók.

-troughcolor a csuszkapálya színét adja meg.

-state a widget állapotát határozza meg. Ez lehet `normal`[44] és `disabled`[45].

-digits a változó értének kiiratásánaál a számjegyek minimális számát határozza meg a tizedespont-tal együtt.

-label szöveges cimkét ad a widgetnek. Függőleges orientáció esetén a program a csúszka jobb felső sarka mellé, vízszintes orientáció esetén a csúszka bal felső sarka fölé helyezi a szöveget.

[44] Vagyis tudjuk használni.
[45] Ott van a képernyőn, de használata tiltott.

-resolution a widget felbontását adja meg. Ha ez az érték nagyobb, mint 0, akkor a változó értéke ezzel az értékkel változik és az eredményt a program kerekíti. Ha az érték kisebb, mint 0, akkor kerekítés nem történik.

-variable annak a változónak a referenciáját adja meg, ahova a widget az értéket helyezi.

-from a változó kezdeti értéke. Függőleges orientáció esetén a csúszka fenti helyzetének felel meg. Vízszintes esetben a csúszka baloldali helyzete a kezdő érték.

-to a változó végértéke.

Megjegyzés: Ha a kezdeti érték nagyobb, mint a végérték a csúszka visszafelé fog számolni.

-command megadja annak a szubrutinnak a referenciáját, amelyet akkor szeretnénk meghívni, amikor a változó értéke megváltozik.

7.12. Listbox

A Listbox widget egy választási listát hoz létre a képernyőn. Paraméterézése hasonló módon történik a többi widgethez. Néhány paraméternek speciális jelentése van. Ezek:

-height a widget magasságát adja meg sorban.

-width a widget kívánt szélességét adja meg karakter számban.

-selectmode a lista elemek kiválasztásának módját adja meg. Ez lehet:

single ekkor az egér segítségével kiválaszthatunk egy és csak egy elemet a listából, ha cursor a kiválasztott elem felett van és megnyomjuk az egér bal gombját.

browse mint az előző pont, de ha nyomva tartjuk az egér gombját és mozgatjuk a cursort a kiválasztás módosul.

multiple egyszerre több elemet választhatunk ki. Ha egy elemen kattintunk az változik[46] anélkül, hogy a többi elemet ez a művelet befolyásolná.

extended itt is egyszerre több elem választható ki úgy, hogy az egér bal gombját nyomva húzzuk a cursort. Ha bárhova újra kattintunk a widgeten belül a kiválasztás áthelyeződik oda és az összefüggő tartomány megszünik.

Ha a kontrol billentyűt nyomva tartjuk létre hozhatunk nem összefüggő tartományt.

A Listbox legfontosabb függvényei a következők:

insert adott sorszámú elem elé szúr be új elemeket. A függvény formális leírása:

```
$listbox->insert(index, elment1, element2, ....);
```

Ezzel a függvénnyel tölthetjük fel a listát.

[46] Ha ki volt választva, akkor a kiválasztás megszünik, ha nem, akkor most ki lett választva.

curselection a kiválasztott listaelem számát adja meg, ha több elem került kiválasztásra, akkor egy listát (tömb) kapunk. Ha nincs kiválasztott elem a visszatérési érték üres string.

A legfelső elem sorszáma 0.

delete a listából töröl egy vagy több elemet. A függvény formális leírása:

```
$listbox->delete(first,last);
```

`first` a törlendő tartomány első eleme, `last` a törlendő tartomány utolsó eleme.

Ha nem adunk meg utolsó elemet, akkor csak a kezdő index által jelölt elemet töröljük a listából.

selectionSet a függvény kiválasztott állapotba hoz egy listaelemet

selectionClear a függvény első és második paramétere közötti elemek esetleges kiválasztott állapota megszűnik.

A `Listbox` widgetet általában a `Srollbar` widgettel együtt használjuk. Ellenkező esetben használata nehézkes.
A következő példa bemutatja, hogy hogyan lehet inicializálni egy `Listbox` widgetet.

```
1  #!/usr/bin/perl
2  use Tk;
3  $mw=MainWindow->new();
4  $mw->maxsize(qw(60 160));
5  $lb=$mw->Listbox( -relief  => 'sunken',
6                    -width   => -1,
7                    -height  => 10,
8                    -selectmode => 'extended',
9                    -setgrid => 'yes')->place(-x => 5, -y => 5);
10 my @i=qw(one two three four five six seven eight nine ten);
11 $lb->insert('end',@i);
                      ⋮
```

Az első négy sor számunkra már jól ismert. Az ötödik sorban léterhozzuk a `Listbox` objektumot paraméterezve.
A hatodik sorban a `width` paraméter értéke -1. Ez azt jelenti, hogy a widget szélessége a legszélesebb listaelemmel legyen azonos.
A tizedik sorban inicializálunk egy tömböt. A tömb elemei tartalmazzák a lista elemeit.
A tizenegyedik sorban a tömböt beszúrjuk a már létrehozott listába.

7.13. Optionmenu

Az `Optionmenu` widgetet hasonlít az előző `Listbox` és a korábban ismertetett `Menubutton` widgetekre. Szintén listát lehet vele kezelni, de alap állapotában csak egy gombot látunk a felületen, amely az aktuális kiválasztott elemet mutatja.
Az `Optionmenu` widget paraméterei:

-options a kiválasztható menüpontok listáját tartalmazó tömb címét adja meg.

-command a változáskor meghívandó szubrutin címét adja meg.

-variable annak a skalárnak a címét adja meg, amelyben a kiválasztott menüpont neve kerül.

Az widget utólag módosítható a `addOptions` függvénnyel.
A következő példa bemutatja a widget használatát.

```
1  #!/usr/bin/perl
2  use Tk;
3  $mw=MainWindow->new();
4  $mw->maxsize(qw(90 160));
5  $op=$mw->Optionmenu( -options  => [qw(one two three four)],
6                       -command  => \&rut,
7                       -variable => \$v)->place( -x => 5,-y => 5);
8  $op->addOptions([five=>'five'],[six=>'six']);
9  MainLoop();
10 sub rut
11    {
12    print "$v\n";
13    }
```

A nyolcadik sorban láthatjuk az `addOptions` függvényt. Ellentétben az ötödik sorban lévő `-option`-ban definiált menüpontokkal itt megadható más visszaadott érték is mint a menüpont neve.[47].
Ha használjuk az `addOptions` függvényt akkor, induláskor a függvényben első megadott pont lesz a kiválasztott (most a 'five').

7.14. `Text`

A `Text` widget egy szövegmezőt kezel[48]. Mivel ez a widget nagyon sok aprólékosan paraméterezhető, ezért csak a legfontosabb tulajdonságait ismertetjük.
A `Text` gyakorlatilag egy kis szövegszerkesztő, minden olyan szerkesztési feladat elvégezhető, amit egy karakter típusú szöveg szerkesztővel meg lehet tenni.
A legfontosabb paraméterek:

-width a widget szélessége karakterben mérve.

-height a widget magassága sorszámban mérve.

-spacing1 pixelben megadja a paragrafus előtt tartott távolságot.

-spacing2 pixelben megadja a sorok közötti távolságot.

-spacing3 pixelben megadja a paragrafus után tartott távolságot.

[47] Nekem ez így logikátlan.
[48] A `Text` man page 1700 sor.

-state értéke lehet:

normal ekkor a widget engedélyezett,

disable , ekkor a widget nem engedélyezett.

-tabs a szövegmezőben a tabulátorok jellegét és helyét adja meg. A jelleg lehet:

left balra ütköztetett,

right jobbra ütköztetett,

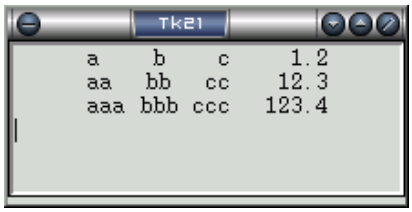
center középre ütköztetett,

numeric tizedespontra ütköztetett.

A `tabs` egy tömb referenciát vár paraméterként, ahol elöl a pozíció, majd a tabulátor jellege következik. Például:

```
-tabs => [qw(1c left 2c center 3c right 4c numeric)]
```

A 7.18. ábra ezt a beállítást mutatja



7.18. ábra: Tabulátorok

-wrap ez a paraméter a sortörés kezelését határozza meg. A lehetőségek:

none a sort nem töri el, a beíráskor eltolja a sort,

word szóhatáron töri el a sort úgy, hogy a teljes szó az új sorba kerül,

char ahol véget ér a sor a widgeten, ott eltöri a sort (a szó közepén is). Ez az alapállapot.

A widget legfontosabb függvényei:

Contents egy skalárban visszaadja a widget tartalmát.

get a függvény paraméterében szereplő index értékek segítségével részleteket emelhetünk ki a szövegből. Használatát a következő program részletekkel mutatjuk be.

```
$r=$text->get('1.0');
```

Az `$r` skalárba az első sor első karakterét teszi.

```
$r=$text->get('1.0','1.5');
```

Az `$r` skalárba az első sor első öt karakterét teszi.

```
$r=$text->get('1.0','1.0 lineend');
```

Az `$r` skalárba a teljes első sort teszi.

```
@r=$text->get('1.0','1.0 lineend','2.0');
```

A tömb `$r`[0] eleme az első sort, a `$r`[1] eleme a második sor első karakterét tartalmazza.

```
@r=$text->get('1.0','','2.0','2.0 lineend');
```

A tömb `$r`[0] eleme az első sor első karakterét, a `$r`[1] a második sort tartalmazza.

```
@r=$text->get('1.0','1.0 lineend','2.0','2.0 lineend');
```

A tömb `$r`[0] eleme az első sort, a `$r`[1] a második sort tartalmazza.

A fenti példákban láthatunk szöveges index paramétereket. Ezek közül a legfontosabbak:

`linestart` az aktuális sor elejére állítja az indexet,

`lineend` az aktuális sor vége utáni karakterre állítja az indexet[49] (ezt láttuk),

`wordstart` az index annak a szónak az elejére áll, amely az aktuális indexet tartalmazza,

`wordend` a szó utáni karakterre állítja az indexet

`delete` tartományt töröl a bevitt szövegből. A tartomány beviteli módja megegyezik a `get` függvény ismertetett móddal.

`insert` sztringet. szúr be a szövegbe. Használatát példákon mutatjuk be.

```
$text->insert('1.0','Hello');
```

A legelső sor legelejére beszúrja a ”Hello” szöveget.

```
$text->insert('1.0','Hello','2.5', Bye);
```

A második sor ötödik eleme elé beszúrja a ”Bye” szöveget.

Tetszőleges számú index—szöveg párt használhatunk. A szöveges index paraméterek itt is használhatók.

[49] A záró index által címzett karakter már nincs benne a tartományban.

7.14.1. `Tag`

A tag egy sztring, amely a szövegmező egy karakterét vagy tartományát esetleg több tartományát határozza meg. A tageknek három szerepe van, ezek:

1. a tag által kijelölt szövegrészlet megjelenését lehet külön definiálni,
2. a taghez külön eseményeket lehet definiálni,
3. a szelekció egyedi kezelését teszi lehetővé. Ez azt jelenti, hogy olyan függvények esetén, ahol indexek segítségével határoztunk meg tartományt, megadhatunk taget is. Például:

```
$r=$text->get($text->tagRanges($tag));
```

A tag definiálására több lehetőség van:

`tagAdd` tag hozzáadása:

```
$text->tagAdd(TAG_NAME,INDEX1,[INDEX2,...]);
```

`tagConfigure` tag beszúrása:

```
$text->tagConfigure(TAG_NAME,CONFIGURATION);
$text->insert(TAG_NAME,INDEX);
```

A tag törlése:

`tagDelete` függvény:

```
$text->tagDelete(TAG_NAME);
```

A függvény az adott tag-re vonatkozó összes információt törli.

A tagek prioritással rendelkeznek, melynek szabályai:

- Általános szabály, hogy bármely tag magasabb prioritással rendelkezik, mint az általános szöveg[50].
- A később definiált tag magasabb prioritással rendelkezik.

A prioritás megváltoztatható:

`tagRaise` növeli az argumentumában megadott tag prioritását.

`tagLower` csökkenti az argumentumában megadott tag prioritását.

[50] Ez logikus, hiszen a taget átkonfiguráljuk, ha az alap szöveg magasabb prioritással rendelkezne, akkor nem lenne értelme a tagnek

7.14.2. `Mark`

A mark célja az, hogy a szövegben egy helyet megjelöljön. Azonban míg a tag egy tartományt jelöl meg a mark azt a helyet jelöli, ahova egy ilyen tartományt szeretnénk beszúrni.
A mark függvényei:

`markSet` egy markot állít be.

```
$text->markSet(MarkName=> INDEX);
```

A mark az `INDEX` által megadott karakter elé kerül.

`markUnset` a megadott nevű markot, vagy markokat törli.

`markGravity` a mark beszúrás utáni helyzetét határozza meg.

```
$text->markGravity(MarkName=> left|right);
```

A paraméterek:

`left` a beszúrás esetén a mark a beszúrt intervallum bal oldalán marad,

`right` a beszúrás esetén a mark a beszúrt intervallum jobb oldalán marad.

`markNames` a függvény egy listával (tömb) tér vissza, amelyben a beállított mark-ok neve van.

`markPrevious, markNext` az adott indexhez viszonyítva az előző—következő mark nevét adja vissza.

7.14.3. `Window`

A window lehetővé teszi, hogy egy widgetet ágyazzunk be a `Text` widget belsejébe. Nézzük a következő példát.

```
$buttom=$text->Button(text => "Hello");
$text->window('create', 'end' ,-window=>$button);
```

A példa első sorában létre hoztunk egy gombot. A második sorban egy már előzőleg létrehozott `Text` típusú widgetben létre hozunk egy `window`-t, amely most a gomb.
Az `end` paraméter itt egy index érték.

7.19. ábra: Szövegbe ”ágyazott” gomb

A 7.19. ábrán láthatjuk, hogy a gomb úgy viselkedik, mint egy speciális betű. Még akár le is lehet törölni.
A `window` paraméterei:

`-align` ha a beágyazott widget magassága kisebb, mint szöveg magassága, akkor megadhatjuk, hogy a widget a szöveghez képest hol legyen. Ez lehet: `top` — felső él, `center` — középvonal és `baseline` — alapvonal.

`-create` a `window`-hoz rendel egy függvényt, ami akkor kerül meghívásra, amikor a widget létrejön. illetve megszűnik.

`-stretch` ha a beágyazott widget magassága kisebb, mint szöveg magassága, akkor a widgetet függőlegesen a szöveg magasságára széthúzza.

`-window` megadja a beágyazott widgetet lásd 7.19. ábra példáját.

A `window` függvényei:

`windowCreate` létrehoz egy `window`-t. A 7.19. ábra példájának második sora ezzel a függvénnyel:

```
$text->windowCreate('end',-window=>$button);
```

`windowConfigure` a függvény `window` paramétereit változtatja. A függvény első paramétere egy index, amely azonosítja a beágyazott widgetet. Ezt követi egy lista `-opcio=>paraméter` formátumban.

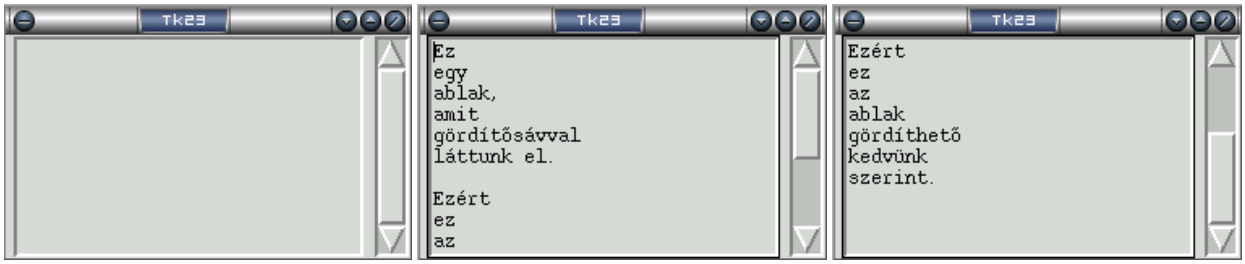
`windowNames` `Text` widgetbe ágyazott `window`-k nevét tartalmazó listával tér vissza.

7.15. `Scrollbar`

A `Scrollbar` widgetet a `Listbox`, a `Text` és a `Canvas` widgetekhez használhatjuk, ha ezek mérete meghaladja a rendelkezésre álló helyet.
A gördítősávot értelemszerűen használhatjuk vízszintesen és használhatjuk függőlegesen. Nézzünk egy példát.

```
1  #!/usr/bin/perl
2  use Tk;
3  $mw=MainWindow->new();
4  $mw->minsize(qw(250 90));
5  $text=$mw->Text(-width=>30,-height=>10);
6  $scroll=$mw->Scrollbar( -command => ['yview',$text]);
7  $text->configure( -yscrollcommand=> ['set',$scroll]);
8  $scroll->pack(-side=>'right',-fill=>'y');
9  $text->pack();
10 MainLoop();
```

Az első öt sor már ismert, létrehozunk egy ablakot és egy `Text` widgetet.
A hatodik sorban létrehozunk a `Scrollbar`-t. A `-command` paraméterben szereplő névtelen tömben adjuk meg, hogy a gördítősáv függőleges legyen, és itt adjuk meg azt is, hogy melyik widgethez lesz ez rendelve.
A hetedik sorban a szövegmezőt konfiguráljuk úgy, hogy függőlegesen gördíthető legyen és itt adjuk meg azt is, hogy melyik gördítősávval. Erre az `-yscrollcommand` paraméterei szolgálnak.
A nyolcadik és kilencedik sorban elhelyezzük a gördítősávot és a szövegmezőt.



7.20. ábra: Szövegmező gördítősávval

A `Scrollbar` widget paraméterei:

`-command` annak a függvénynek a referenciáját adja meg, amelyet akkor kerül meghívásra, ha a gördítősávhoz rendelt widget megváltozik a gördítősáv elmozdulásának hatására.

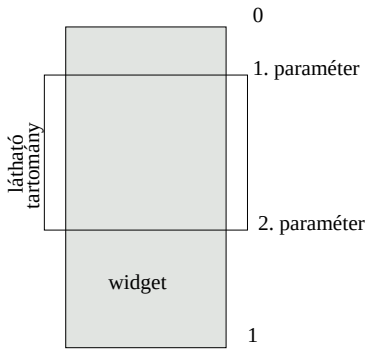
A `Scrollbar` widget függvényei:

`set` ezt a függvényt a gördítősávhoz rendelt widget hívja meg. A függvénynek két paramétere van, amelyek 0 és 1 között lehetnek. Az első paraméter megadja, hogy a gördítősáv hol álljon, ha a widget elején vagyunk. A második megadja, hogy a widget végét úgy tekintse, mintha az csak az adott része lenne.

```
$scroll->set(0.3,0.7);
```

Tehát a `Scrollbar` 30%-on áll, ha a widget tetején vagyunk, és a csúszka hossza akkora, mintha a widget vége csak az össz. hossz 70%-a lenne.

`get` egy tömbben visszaadja a widget látható tartományát.



7.21. ábra: A `get` értelmezése

activate a gördítősáv egy elemét aktiválja A megadható elemek:

arrow1 a felső, vagy bal oldali nyíl,

arrow2 az alsó, vagy jobb oldali nyíl,

slider a csúszka.

Ha nem adunk meg paramétert, akkor a függvény az éppen aktív elem nevével tér vissza. Ha egy elem sem aktív, a visszatérési érték üres sztring.

fraction egy tört értékkel tér vissza, amely megadja, hogy a csúszka hol áll. Vízszintes gördítősáv esetén 0 a bal oldali, függőleges esetben a legfelső pozíciónak felel meg.

identify megadja annak az elemnek a nevét (pl. arrow1), amely a függvény argumentumában megadott x, y paraméterek ”alatt” van.

A `Scrollbar` widgettel kapcsolatos, a görgetett widget függvényei.
A függvényt tehát nem a gördítősávhoz, hanem a kezelendő widgethez rendeljük. Például:

```
$text->yviewMoveto(0.5);
```

xviewMoveto x irányban a widgetet elmozdítja úgy, hogy a látható ablak a paraméterben megadott törtnek feleljen meg. A 0 érték a bal oldalnak, az 1 a jobb oldalnak, a 0.5 a widget közepének.

yviewMoveto mint az előző, csak függőlegesen.

xviewScroll megadja azt, hogy a felső és alsó nyílra kattintás esetén a widget hogyan és mennyit mozduljon el x irányban. Példa:

```
$text->yviewScroll(1,"page");
```

Ez azt jelenti, ha a felső nyílra kattintunk a widget egy oldalt ugrik előre. Az ugrás ”dimenziója” lehet `"unit"`, ez pixelt, vagy `"page"`, ez lapot jelent.

yviewScroll mint az előző, csak függőlegesen.

7.16. `Scrolled` függvény

Az előzőekben láthattuk, hogy a `Scrollbar` widget kezelése nem túl egyszerű. A perl-tk-ban van azonban más lehetőség is, hogy egy widgetet ellássunk gördítősávokkal. Erre szolgál a `Scrolled` függvény.
Használata sokkal egyszerűbb, mint a `Scrollbar` használata. Nézzük ezt egy példán:

```
1 #!/usr/bin/perl
2 use Tk;
3 $mw=MainWindow->new();
4 $mw->minsize(qw(250 90));
5 $text=$mw->Scrolled("Text",-scrollbars=>’e’,-width=>30,-height=>10);
6 $text->pack();
7 MainLoop();
```

Láthatjuk, hogy az ötödik sor megcsinál mindent, ami a `Text` widgethez kell és még meghatározza a gördítősáv helyét is a `-scrollbars` paraméterrel.

`-scrollbars` paraméter megadja a gördítősáv vagy gödítősávok helyét. Minden widget maximum két gördítősávval rendelkezhet. A `-scrollbars` paraméterei:

`w` bal oldalt függőleges,

`n` fent vízszintes,

`e` jobb oldalt függőleges,

`s` alul vízszintes,

`nw` felül és bal oldalt,

`sw` alul és bal oldalt,

`ne` felül és bal oldalt,

`se` alul és bal oldalt

`center` ugyanaz, mint az `ne`.

Ezek után a gördítősávok ugyanúgy paraméterezhetők, mint azt az előző fejezetben láttuk.

7.17. `Canvas`

A `Canvas` widget a rajzvászon, ez biztosítja azt a felületet az ablakban, amelyre a rajzelemek elhelyezhetők.
A vásznon elhelyezhető elemek:

Line vonal rajzolása,

Arc körív rajzolása,

Bitmap bittérképes kép elhelyezése,

Image kép elhelyezése,

Oval kör, vagy ellipszis rajzolása,

Poligon sokszög rajzolása,

Rectangle téglalap rajzolása,

Text szöveg kezelése,

Grid háló rajzolása,

Group más elemek csoportosítására szolgáló elem.

A vászonon a $(0,0)$ koordinátájú pont a bal felső sarokban található. Az x-korrdináta jobbra, az y-koordináta lefelé növekszik.
A rajzvászon konfigurálására mindazon opciók használhatók, amelyekkel az eddig felsorolt widgeteknél találkoztunk.
A leggyakrabban a méretet meghatározó `-width` és `-height`, továbbá a háttérszínt megadó `-backround` opciókat használjuk.
A rajzvászon sok esetben nem fér el az ablakban, ekkor görgetni kell. Definiálni tudjuk azt a területet, amely görgethető. Erre a `-Scrollregion` opció használható. Lássuk példaként a következő program részletet!

```
    :
$canvas=$mw->Canvas(-width=>500,-height=>500)->pack();
$canvas->configure(-scrollregion => [ $canvas->bbox("all") ]);
    :
```

A példában a görgethető terület a teljes rajzvászon. Ebben az esetben célszerűbb a `Scrolled` (90 oldal) függvényt használni.

```
    :
$canvas = $mw->Scrolled('Canvas',-width=>500,-height=>500)->pack();
    :
```

Ha nem akarjuk a teljes területet görgetni, akkor a kérdéses területet befoglaló téglalapot kell megadnunk.

```
    :
$canvas->configure(-scrollregion => [ 5,20,35,50 ]);
    :
```

Az $(5,20)$ a téglalap bal-felső sarkának, a $(35,50)$ a téglalap jobb-alsó sarkának koordinátái.
A görgetés lépésnagysága állítható:

-xscrollincrement a vízszintes görgetés lépésnagyságát adja meg. Ha ez az érték 0, akkor a görgetés normál lépésnagysággal történik.

-yscrollincrement a függőleges görgetés lépésnagyságát adja meg.

A következőkben a festővásznon elhelyezhető elemeket mutatjuk be.

7.17.1. `Line`

A `Line` widget vonal, illetve vonal sorozat rajzolását teszi lehetővé a festővásznon. A következő programrészlet egy ferde vonalat hoz létre.

```
        :
$line=$canvas->createLine(20,20,80,80);
        :
```

Ha a koordinátapárok felsorolását egy következő koordinátapárral folytatjuk, akkor az első vonal utolsó pontjától folytatja a vonalat a megadott koordinátához. Lásd a következő példát:

```
        :
$line=$canvas->createLine(20,20,80,20,80,80);
        :
```

Ha további koordinátapárokat adunk a paraméter sorhoz, akkor a újabb szakaszokat rajzolunk. Lényeges, hogy a koordináták száma páros legyen. Amennyiben a vonalsorozatot meg akarjuk szakítani, akkor új vonalat kell létrehoznunk

A `Line` widget paraméterei:

`-width` megadja a vonal vastagságát. A vastagság mértéke lehet pixelben, mm-ben vagy cm-ben. Normális esetben az alapértelmezett vastagság 1 pixel.

`-arrow` megadja, hogy a vonal melyik végére helyezzen nyilat. Értéke lehet:

`none` ne tegyen nyilat a vonalra (alapértelmezett),

`first` a vonal(sorozat) kezdőpontjára tegyen nyilat,

`last` a vonal(sorozat) végpontjára tegyen nyilat,

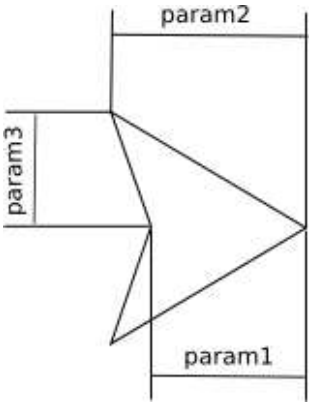
`both` a vonal(sorozat) mindkét végpontjára tegyen nyilat.

Lásd 7.23. ábra!

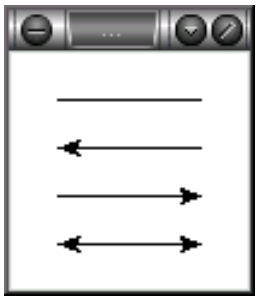
`-arrowshape` a vonal végekre teendő nyíl alakját határozza meg. Megadási módja:

```
        :
$line=$canvas->createLine(20,20,80,80,
                          -arrow => 'last',
                          -arrowshape => [ 10,12,5 ]);
        :
```

Az `-arrowshape` paramétereit, mint láthatjuk egy névtelen tömb segítségével adhatjuk meg. a paraméterek értelmezése a 7.22 ábrán látható.



7.22. ábra: A nyíl paraméterei



7.23. ábra: Nyilak helye

-capstyle a vonal végének ”formáját” határozza meg, ezeknek a paramétereknek természetesen a vastagabb vonalak esetén van jelentősége.

A paraméter értéke lehet:

butt a vonal végén — az adott koordinátán — a vonalat merőlegesen levágja (alapértelmezett).

projecting itt is merőleges levágással fejezi be a vonalat, de nem az adott koordinátán. A vonalat fél vonalvastagsággal tovább húzza.

round a vonal végét lekerekíti és fél vonalvastagsággal tovább húzza.

-joinstyle a paraméter a vonalak találkozásának módját határozza meg. Ennek a paraméternek akkor van értelme, ha legalább két vonalat rajzolunk egy `createLine` hívással és a vonalak vastagok.

Értéke lehet:

bevel a vonalak külső pontjait a találkozásnál egyenesen összeköti,

miter a vonalak találkozása szögletes (alapértelmezett),

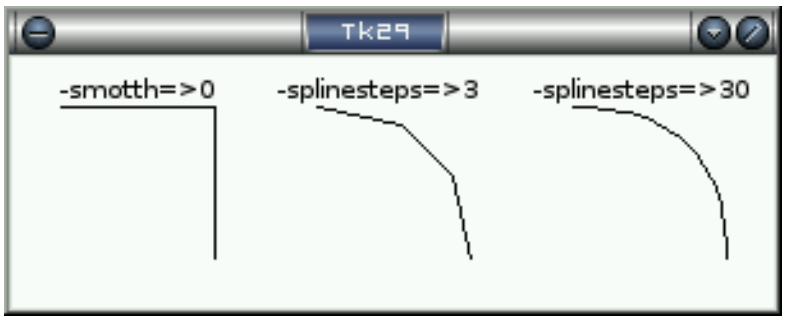
round a vonalak találkozása lekerekített.

7.24. ábra: `-capstyle` változatok

7.25. ábra: `-joinstyle` változatok

-smooth logikai változó, értéke lehet 0 vagy 1. Ha 1 a vonalak találkozását Bézier interpolációval lekerekíti. 0 esetén a vonalak találkozása szögletes.

-splinesteps ha a `-smooth` be van kapcsolva az interpoláció lépéseinek számát adja meg.



7.26. ábra: `-smooth` és a `-splinesteps` hatása

-stipple egy bittérképpel meghatározza a vonal mintázatát. A bittérkép lehet valamelyik alapértelmezett — a rendszerrel együtt ”szállított” bittérkép, vagy egy bittérképet tartalmazó fájl.

A bittérkép azon pixeljei, amelyeknek értéke 1 a vonal beállított színének felelnek meg, a 0 értékű bitek átlátszóak.

-tags a létrehozott vonalhoz rendelhetünk egy, vagy több nevet, amelynek segítségével egyszerűbben hivatkozhatunk a létrehozott objektumot.

```
        :
$line=$canvas->createLine(... ,-tags => [ "vonal", "tengely" ]);
        :
```

7.17.2. Arc

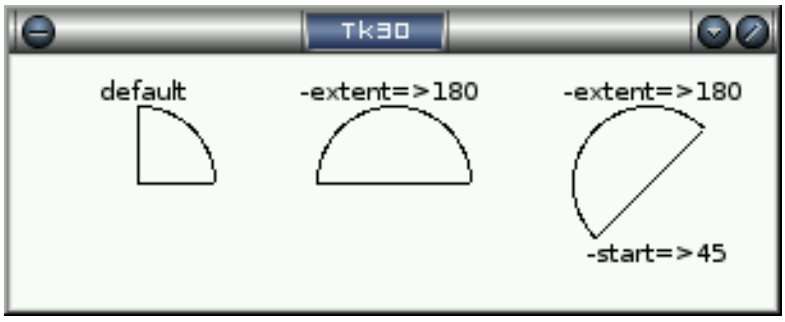
Az `Arc` elem egy körívet rajzol a vászonra. Az ívet a `createArc` függvénnyel hozhatjuk létre.
A körív paraméterezése eltér a megszokottaktól, mert nem a középpontot és a sugarat adjuk meg, hanem a körívet tartalmazó téglalap bal felső és jobb alsó koordinátáját Ez lehetővé teszi, hogy ellipszis ívet hozzunk létre. A következő példa egy negyedkört rajzol a képernyőre.

```
$arc=$canvas->createArc(20,20,80,80);
```

Az `Arc` paraméterei:

-extent az ív szögtartományát határozza meg. A paraméter tartománya $[-360 \ldots 360]$. A default érték 90, vagyis 90^o.

-start meghatározza az ív kezdő szögét.



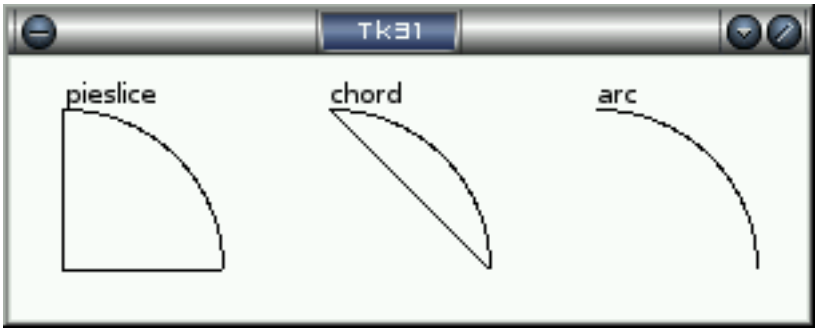
7.27. ábra: Az `-extent` és a `-start` hatása

-style az ív stílusát határozza meg, ez lehet:

pielsice tortaszelet szerű síkidomot hoz létre (alapértelmezett),

chord az ív két végpontját egy egyenessel köti össze,

arc csak az ívet rajzolja meg nem készít síkidomot.



7.28. ábra: Az `Arc` stílusai

`-width` meghatározza a vonal vastagságát.

`-outline` meghatározza a vonal színét.

`-outlinestipple` lásd `Line -stipple` 95 oldalon.

`-fill` meghatározza a kitöltés színét.

`-stipple` egy bittérképpel meghatározza a kitöltés mintázatát. A bittérkép lehet valamelyik alapértelmezett — a rendszerrel együtt ”szállított” bittérkép, vagy egy bittérképet tartalmazó fájl.

`-tags` lásd `Line -tags` 95 oldalon.

7.17.3. `Rectangle`

Az `Rectangle` elem egy téglalapot rajzol a vászonra. Az ívet a `createRectangle` függvénnyel hozhatjuk létre. A téglalap létrehozását a következő példa mutatja:

```
$canvas->createRectangle(10,20,30,40,-fill=>'blue',-width=>2);
```

Paraméterei:

az első két paraméter megadja a téglalap bal felső sarkának koordinátáit (a példában $x = 10$ és $y = 20$),

a harmadik és negyedik koordináta a téglalap jobb alsó sarkának koordinátáit adja meg (a példában $x = 30$ $y = 40$),

`-fill` meghatározza a kitöltés színét,

`-width` meghatározza a vonal vastagságát.

7.17.4. `Widget`

A `canvas` widgetbe elhelyezhetünk tetszőleget widgetet. Azonban ez nem a legegyszerűbb eljárás. A következő példa mutatja, hogyan helyezhető el egy gomb a vásznon.

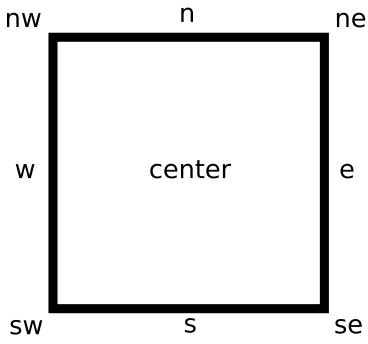
```
$b=$c->Button(-text=>"Gomb",-command=>sub {print "Gomb\n";});
$w=$c->createWindow(0,0,-window=>$b);
```

Az első lépés a kívánt widget létrehozása, majd egy `Window` létrehozása, amelyen letesszük a gombot. Az ablak első két paramétere a gomb pozíciója a vásznon.
A paraméterezés célszerűen a `Window`-n keresztül történik. Paraméterei:

`-anchor` Azt a pontot adja meg a gombon, amelyre a koordináták vonatkoznak. A lehetőségek:

n felső oldal,
e jobb oldal,
s alsó oldal,
w alsó oldal,
ne jobb felső sarok,
nw bal felső sarok,
se jobb alsó sarok,
sw bal alsó sarok,

Az alapértelmezés a widget közepe.



7.29. ábra: Az `anchor` paraméterek

`-height` a kérdéses widget magasságát adja meg,

`-width` a kérdéses widget szélességét adja meg.

A mennyiben nem adjuk meg a szélességet és vagy a magasságot, akkor a kérdéses widget saját természetes méreteit tartja meg.

7.17.5. `Grid`

A `Grid` lehetővé teszi számunkra, hogy a vászonra egy hálózatot rajzoljunk (mint például egy oszcilloszkóp képernyője).
A legegyszerűbb grid létrehozása a következő módon történik:

```
$c=$mw->Canvas(-width=>190,-height=>190,
               -bg => 'gray')->place(-x=>4,-y=>4);
$c->createGrid(0,0,10,10);
```

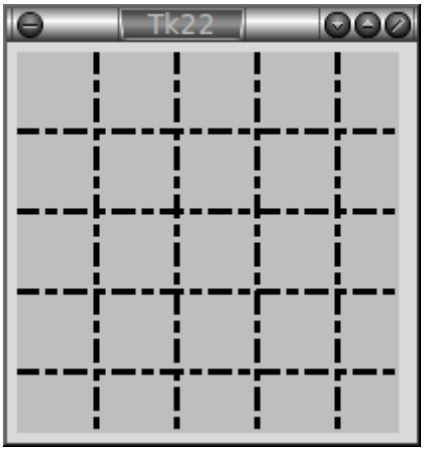
7.30. ábra: Egyszerű `grid`

Paraméterei:

`-lines` logikai változó, ha iga, akkor a háló nem pontokból, hanem vonalakból áll,

`-dash` a vonal stílusát adja meg, például ha pontozott vonalat szeretnénk, akkor a megoldás `-dash=>'-.'`,

`-width` a vonal vastagságát adja meg.



7.31. ábra: `-dash=>'-.'` és `-width=>3`

7.17.6. `Group`

A `Group` "elemmel" több képernyő elemet tudunk összekapcsolni. Ezeket a későbbiekben a Tk értelemszerűen egy egységként kezeli. A következő programrészlet egy magyar zászlót készít a vásznon és a három színt képviselő téglalapokat egy csoportként kezeli.

```
$c=$mw->Canvas(-width=>190,-height=>190,
               -bg => 'gray')->place(-x=>4,-y=>4);

$p=$c->createRectangle(20,20,80,30,-fill=>'red');
$f=$c->createRectangle(20,30,80,40,-fill=>'white');
$z=$c->createRectangle(20,40,80,50,-fill=>'green');

$group=$c->createGroup(0,0,-members=>[$p, $f, $z]);
```

7.17.7. Elemek konfigurálása a vásznon

A megrajzolt elemek tulajdonságai utólag megváltoztathatóak. Ezáltal mozgó alakzatokat tudunk létrehozni. Így a `perl-tk` segítségével változó eseményeket nagyon könnyen figyelemmel tudunk kísérni
A következő példa egy téglalapot rajzol a vászonra és 0.1 másodpercenként egy pixellel növeli a magasságát. Ha eléri az 50 pixel magasságot a színe zöldről pirosra vált 100 pixel magasság után nem nő tovább.

```
1  #!/usr/bin/perl
2  use Tk;
3
```

```
4  $mw=MainWindow->new();
5  $mw->geometry("300x200");
6  $c=$mw->Canvas(-width=>190,-height=>190,
7                 -bg => 'gray')->place(-x=>4,-y=>4);
8  $i=0;
9  $b=$c->createRectangle(100,100,120,100,-fill=>'green');
10
11 $mw->repeat(100,\&rut);
12 MainLoop();
13
14 sub rut
15   {
16   if($i>=100) { return; }
17   $c->coords($b,100,100,120,100-$i);
18   if($i>=50)
19     {
20     $c->itemconfigure($b,-fill=>'red');
21     }
22   $i++;
23   }
```

A 9. sorban létrehozzuk a téglalapot, amelynek magassága 0 pixel és a színe zöld. A 11. sorban a `repeat` függvénnyel 0.1 másodpercenként meghívjuk a `rut` szubrutint, amely az `$i` változó értékének megfelelően változtatja a téglalap magasságát és színét.
A magasság változtatása a 17. sorban található `coords` függvénnyel történik.
A színt a 20. sorban változtatja az `itemconfigure` függvény.
Amennyiben egy kérdéses elem koordinátáit szeretnénk lekérni szintén a `coords` függvényt hívhatjuk meg az alábbi módon:

```
($x1,$y1,$x2,$y2)=$c->coords($b);
```

Ha egyéb paramétereket is le szeretnénk kérdezni, akkor azt a kérdéses paraméter megadásával az `itemcget` függvénnyel kaphatjuk meg.

```
$color=$c->itemcget($b,-fill);
```

Az adott elem mozgatása a vásznon a `move` függvénnyel történik.

```
$c->move($b,50,50);
```

7.17.8. Eseménykezelés a vásznon

Az eseménykezelésről a későbbiekben lesz még szó. Ez az alfejezet kimondottan a vászonnal kapcsolatos eseménykezeléssel foglakozik.
A következő programrészlet a cursor koordinátáit írja ki az indító konzolra az egér bal gombjának megnyomására.

```
$c->CanvasBind("<Button-1>",[\&rut, Ev('x'), Ev('y')]);
MainLoop();

sub rut
  {
  print "$_[1] $_[2]\n";
  }
```

7.17.9. A vászon mentése fájlba

Erre a célra az úgynevezett `postcript` "elem" szolgál.

```
$ps=$c->postscript();
$c->postscript(-file=>'out.ps');
```

Paraméterei:

`-colormode` ezzel lehet a postscript "nyomtatás" módját beállítani. Ez lehet: "color", "gray", vagy "mono",

`-file` a mentés helye,

`-height` a "nyomtatandó" terület magassága,

`-width` a "nyomtatandó" terület szélessége,

`-pageanchor` hol értelmezze a megadott pozíció paramétereket. Ez lehet: "n", "e", "s", "w", vagy "center"

`-x` megadja a "nyomtatás" baloldali koordinátáját,

`-y` megadja a "nyomtatás" felső koordinátáját,

`-x` megadja a "nyomtatási" terület baloldali koordinátáját,

`-y` megadja a "nyomtatási" terület felső koordinátáját.

7.17.10. `tag`-ek

A `tag` egyfajta csoportot jelent. Minden a vásznon létrehozott elem egy `tag`. Ezeket az elemeket tovább is lehet csoportosítani[51] erre szolgál a `Canvas addtag` függvénye.
A `tag`-et úgy hozzuk létre, hogy a megvalósított elemeket valamilyen logikai kifejezéssel egymáshoz rendeljük. A logikai kifejezés lehet `&&` és, `||` vagy, `^` kizáró vagy és a `!` nem. Továbbá ezek kombinációi. Két speciális `tag` van. Ezek a `current` ez akkor létezik, ha az egér cursor a kérdéses elem felett van. A `all` az összes elemet jelenti a vászon felett.
Egy `tag` az `addtag` függvénnyel bővíthető, illetőleg hozható létre. Egy egyszerű összerendelés a következő módon történik:
Tételezzük fel, hogy a vásznon három elem van, a nevük `$x`, `$y` és `$z`.

[51] Persze, különben a `tag` fogalom értelmetlen lenne.

```
$canvas->addtag('newtag','$x||$y');
```

A `newtag` az `$x` és az `$y` elemekből áll.

```
$canvas->addtag('newtag','$x^$y');
```

A `newtag` az `$x` és az `$y` elemekből valamelyikéből áll, de egyszerre a kettőből nem.
Az `addtag` függvény rendelkezik néhány opcióval az opciók használata a következő:

```
addtag("tagname","option",tag/id,...);
```

Az opciók:

`"above"` közvetlenül a kérdéses `tag` fölötti `tag`-et, vagy elemet adja hozzá az új `tag`-hez,

`"below"` közvetlenül a kérdéses `tag` alatti `tag`-et, vagy elemet adja hozzá az új `tag`-hez,

`"closest"` megkeresi az adott `tag`-hez legközelebbi elemet és hozzáadja. Megadhatjuk azt is, hogy egy adott helytől keresse a legközelebbi elemet, adott sugárral:

```
$canvas->addtag("tagname","closest",30,40,10);
```

Vagyis az itemhez adja hozzá azt az elemet, vagy `tag`-et, ami a 10 pixel sugáron belül van.

`"enclosed"` egy adott területen belül található elemeket, vagy `tag`-okat adja hozzá a kérdéses `tag`-hoz:

```
$canvas->addtag("tagname","enclosed",10,10,40,40);
```

A kérdéses elem(ek)nek, vagy `tag`-(ek)nek teljesen benne kell lenniük a kijelölt területben.

`"overlapping"` hasonló az előző ponthoz, de ebben az esetben elegendő, ha a kérdéses elem(ek)nek egy része van a kijelölt területben.

`"withtag"` az aktuális `tag`-et hozzárendeli egy másik `tag`-hez.

A `tag`-ek keresésére szolgála a `find` függvény. A `find` visszaadja a `canvas`-on lévő elem(ek) sorszámát. A következő program erre mutat példát:

```
#!/usr/bin/perl

use Tk;
$mw=MainWindow->new();
$mw->geometry("200x200");
$c=$mw->Canvas(-width=>190,-height=>190,-bg=>'gray')->pack();

$b0=$c->createRectangle(10,10,40,40,-fill=>'blue');
$b1=$c->createRectangle(110,110,140,140,-fill=>'blue');
@r=$c->find("overlapping",15,15,40,40);
```

```
print "$r[0] $r[1]\n";

MainLoop();
```

A `print` az `$r[0]` írja ki. Ennek az értéke 1 lesz.
Ha a koordináta sor `115,115,140,140`, akkor a kiírt érték természetesen 2.
A `find` paraméterezése azonos az `addtag` paraméterezésével, természetesen a `"newtag"` nélkül. Vagyis:

```
$c->find("above", tag/id);
$c->find("all");
$c->find("below", tag/id);
$c->find("closest", x, y [ , additional_area ] [ , tag/id ]);
$c->find("enclosed", x1, y1, x2, y2);
$c->find("overlapping", x1, y1, x2, y2);
$cs->find("withtag", tag/id);
```

Ha arra vagyunk kíváncsiak, hogy egy `tag`-ben milyen elemek vannak a `gettags` függvényt használjuk:

```
@list=$c->gettags(tag/id);
```

A lista a felhasznált `tag`-ek listáját adja vissza. Ha nem talált semmit akkor a visszaadott érték üres sztring.

7.18. Elrendezés kezelők

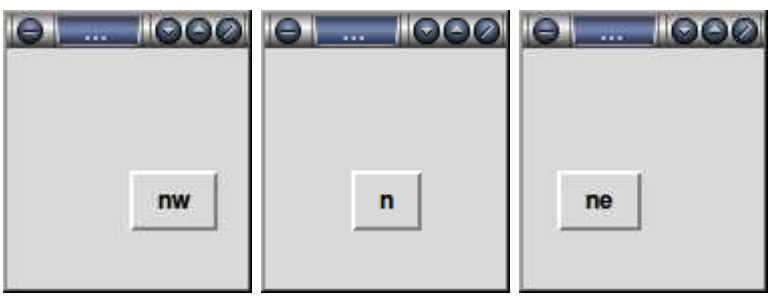
Az elrendezés kezelők feladata az adott területen (`Window`,) (`Toplevel`, (`Frame`, a widgetek elhelyezése különböző szempontok szerint. A perl-tk négy ilyen vezérlőt ismer, ezek: `place`, `pack`, `grid`, `form`.

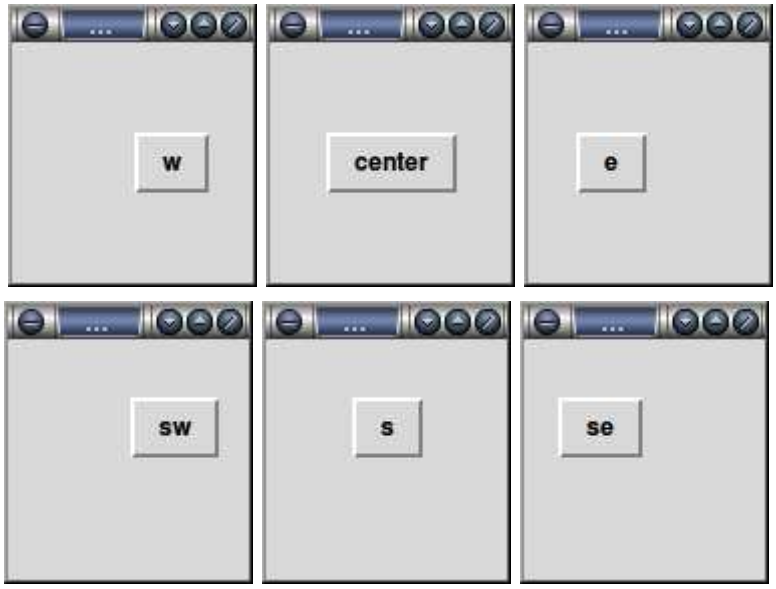
7.18.1. `place`

A `place` megmondja az adott szülő widgeten a kérdéses widget helyét és egyéb geometriai jellemzőit.
A place paraméterei:

`-anchor` megadja, hogy a kérdéses widget melyik oldalát, vagy sarkát, esetleg közepét kell figyelembe venni a lehelyezésnél.

Ez lehet `'n'`, `'e'`, `'w'`, `'s'`, `'nw'`, `'ne'`, `'sw'`, `'se'`, `'center'`, lásd 7.29. ábra.





7.32. ábra: Az `-anchor` értelmezése

A default az `nw`. Ha csak egybetűs `anchor` paraméterünk van a másik ”irány” mindig a középvonal.

`-bordermode` megadja, hogy a widget kerete beleszámít-e a kérdéses koordinátákba. Ennek megfelelően a paraméter lehet: `'inside'`, `'outside'`, `'ignore'`.

`-x` megadja a widget x koordinátáját a szülő widgeten (`-anchor` szerint),

`-y` megadja a widget y koordinátáját a szülő widgeten (`-anchor` szerint),

`-height` meghatározza a widget abszolút magasságát. Ezt az értéket akkor sem lehet meghaladni, ha a widget egyébként magasabb lenne.

`-width` meghatározza a widget abszolút szélességét Hasonlóan az előző paraméterhez ez az érték sem haladható meg.

`-relx` a szülő widget szélességéhez viszonyítva helyezi el a kérdéses widgetet,

`-rely` a szülő widget magasságához viszonyítva helyezi el a kérdéses widgetet,

`-relheight` a szülő widget magasságához viszonyítva adja meg a widget magasságát,

`-relwidth` a szülő widget szélességéhez viszonyítva adja meg a widget szélességét,

7.18.2. `pack`

A `pack` elrendezés kezelő célja a kérdéses widget optimális elhelyezése az adott területen. Használatához nem kell ismerni a szülő widget paramétereit és a kérdéses koordinátákat.
A `pack` legegyszerűbb használata a következő példán látható:

```
$mw->Checkbutton(-text=>"elso")->pack();
$mw->Checkbutton(-text=>"masodik")->pack();
$mw->Checkbutton(-text=>"harmadik")->pack();
$mw->Button(-text=>"Kilep",
            -command=>sub {exit;})->pack();
```

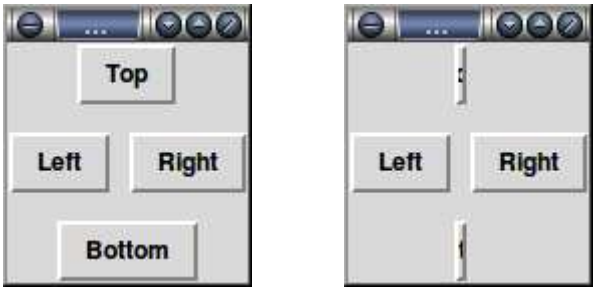
7.33. ábra: Egyszerű `pack`

Az igazat megvallva ez az elrendezés meglehetősen ”ronda”. A hasonló problémák elkerülésére a `pack` számos paraméterrel rendelkezik. Ezek a paraméterek a következők:

`-side` megadja, hogy a kérdéses widget a szülő widgeten belül melyik oldalra kerüljön. Ez lehet: `'left'`, `'right'`, `'top'`, `'bottom'`. Alapértelmezett helyzet a `top`.

`-fill` előírja, hogy a widget a rendelkezésre álló helyet (un. allokációs terület) milyen irányba töltse ki. Ez lehet: `'none'`, `'x'`, `'y'`, `'both'`, az alapértelmezett kitöltés a `'none'`.

Megjegyzés: itt érdemes megemlíteni, hogy a widgetek létrehozási sorrendjét figyelembe venni, mert az eredmény furcsa lehet.



7.34. ábra: Jó sorrend (Top, Bottom, Left, Right), rossz sorrend Left, Right, Top, Bottom)

`-expand` logikai érték, amely előírja, hogy a rendelkezésre álló helyet a widget töltse ki a kérdéses ablakban (`Top`, `MainWindow`), vagy keretben `Frame`. Ha használjuk a `-fill` opciót, akkor ki is tölti a helyet. Ha nem, akkor az allokációs terület közepére teszi a kérdéses widgetet.

7.35. ábra: A négy gomb `-fill`-lel, `-fill,-expand`-al és `-expand, -fill` nélkül

`-anchor` megadja, hogy a kérdéses widget melyik oldalát, vagy sarkát, esetleg közepét kell figyelembe venni a lehelyezésnél.

Ez lehet `'n'`, `'e'`, `'w'`, `'s'`, `'nw'`, `'ne'`, `'sw'`, `'se'`, `'center'`, lásd 7.29. ábra.

-after megadja, hogy a kérdéses widgetet melyik widget után tegye le. Formája:

```
$widget->pack(-after=>elozowidget);
```

-before megadja, hogy a kérdéses widgetet melyik widget elé tegye le. Formája:

```
$widget->pack(-before=>kovetkezowidget);
```

-in abban az esetben, ha a widgetet nem a szülő widgetben szeretnénk elhelyezni, akkor ezzel a paraméterrel adhatjuk meg a bennfoglaló widgetet. Formája:

```
$widget->pack(-in=>masikwidget);
```

-ipadx megnöveli a widget méretét vízszintesen a megadott mértékkel,

-ipady megnöveli a widget méretét függőlegesen a megadott mértékkel,

-padx addig növeli a widget értékét vízszintesen, amíg az allokációs terület szélétől a távolsága a megadott érték,

-pady addig növeli a widget értékét függőlegesen, amíg az allokációs terület szélétől a távolsága a megadott érték,

7.18.3. grid

A `grid` elrendezés vezérlő a szülő widgetet sorokra és oszlopokra osztja és az így keletkezett cellákba rakja le az elhelyezendő elemeket, lásd a következő példát!

```
$mw->Button(-text=>'1')->grid(
         $mw->Button(-text=>'2'),
         $mw->Button(-text=>'3'));
$mw->Button(-text=>'4')->grid(
         $mw->Button(-text=>'5'),
         $mw->Button(-text=>'6'));
$mw->Button(-text=>'7')->grid(
         $mw->Button(-text=>'8'),
         $mw->Button(-text=>'9'));
```

7.36. ábra: Gombok

A `grid` opciói:

"-" speciális karakter, amely előírja egy cellaelem összevonását. Egy `"-"` elem eggyel növeli az oszlop szélességét az adott helyen. Ez nem jelenti azt, hogy a kérdéses widget kitölti ezt a helyet (részletezést lásd később). A következő példa mutatja be a hatását.

```
$mw->Button(-text=>'1')->grid(
        $mw->Button(-text=>'2'),
        $mw->Button(-text=>'3'));
$mw->Button(-text=>'4')->grid(
        $mw->Button(-text=>'5'),
        "-");
$mw->Button(-text=>'7')->grid(
        $mw->Button(-text=>'8'),
        $mw->Button(-text=>'9'));
```

7.37. ábra: `"-"` hatása

`"x"` speciális karakter, amely előírja az oszlopszámláló eggyel történő növelését. Lásd következő példa!

```
$mw->Button(-text=>'1')->grid(
        $mw->Button(-text=>'2'),
        $mw->Button(-text=>'3'));
$mw->Button(-text=>'4')->grid(
        "x");
        $mw->Button(-text=>'6'),
$mw->Button(-text=>'7')->grid(
        $mw->Button(-text=>'8'),
        $mw->Button(-text=>'9'));
```

7.38. ábra: `"x"` hatása

`-column` megadja, hogy a kérdéses widgetet melyik oszlopba tegye,

`-row` megadja, hogy a kérdéses widgetet melyik sorba tegye,

`-columnspan` megadja a leteendő widget szélességét oszlopokban értelmezve a `-column` értéktől kezdődően,

`-rowspan` megadja a leteendő widget magasságát sorokban értelmezve a `-row` értéktől kezdődően,

`-sticky` egy sztringet vár, amely megmondja, hogy a leteendő widget a cella melyik oldalára "ragadjon". Ez lehet: `n`, `w`, `s` és `e`, illetve ezek kombinációi.

```
$mw->Button(-text=>'1')->grid(
        $mw->Button(-text=>'2'),
        $mw->Button(-text=>'3'));
$mw->Button(-text=>'5')->grid(
        $mw->Button(-text=>'6',
        -sticky=>'nwse'),
        "-");
$mw->Button(-text=>'7')->grid(
        $mw->Button(-text=>'8'),
        $mw->Button(-text=>'9'));
```

7.39. ábra: `"-sticky"` hatása

`-in` megadja azt a widgetet, amelybe a lerakandó widgetek kerülnek a szülő helyett,

`-ipadx` megnöveli a widget méretét vízszintesen a megadott mértékkel,

`-ipady` megnöveli a widget méretét függőlegesen a megadott mértékkel,

-padx addig növeli a widget értékét vízszintesen, amíg a rendelkezésre álló terület szélétől a távolsága a megadott érték,

-pady addig növeli a widget értékét függőlegesen, amíg a rendelkezésre álló terület szélétől a távolsága a megadott érték.

7.18.4. `form`

A `form` elrendezés vezérlő eltér az eddig tárgyalt eszközöktől. Kicsit azt lehetne mondani, hogy a `form` olyan, mint minden előző elrendezés vezérlő egyfajta keveréke. A `form` paraméterei:

-top megadja, hogy a kérdéses widgetet a szülő widget felső élétől mérve százalékosan milyen magasra tegye. A felső él a 0% a szülő widget alja a 100%. Ellentétben a `place`-el itt nem a pontos helyet adjuk meg pixelben.

Erre a legjobb példa az, hogy ha a több widgetet helyezünk el egy ablakban és az ablakot utólag megnyújtjuk, akkor a widgetek az adott irányban tartják az arányt. Ekkor a widgetek mérete nem változik.

-bottom működése hasonló a `-top`-hoz, csak a szülő widget aljától értelmezi a lerakást,

-left működése hasonló a `-top`-hoz, csak a szülő widget bal oldali élétől értelmezi a lerakást,

-right működése hasonló a `-top`-hoz, csak a szülő widget jobb oldali élétől értelmezi a lerakást,

-topspring, -bottomspring, -leftspring, -rightspring ezek a paraméterek egyfajta prioritást fejeznek ki az elhelyezés során. Úgy kell ezeket elképzelni, mint olyan rugókat, amelyek az adott oldalhoz vannak rögzítve. Amennyiben ezek közül a paraméterek közül egyiket sem alkalmazzuk, akkor úgy tekinti a rendszer, mintha az elhelyezendő elem minden oldalhoz egyformán lenne kifeszítve.

Amelyik rugó ”értéke” nagyobb, annak prioritása magasabb az elhelyezést tekintve.

-padx addig növeli a widget értékét vízszintesen, amíg az allokációs terület szélétől a távolsága a megadott érték,

-pady addig növeli a widget értékét függőlegesen, amíg az allokációs terület szélétől a távolsága a megadott érték,

-fill meghatározza azt az irányt, amelybe a widgetet szét kell húzni, ha `spring`-et használtuk A lehetséges állapotok: `'x'`, `'y'`, `'both'` és `'none'`. Alapértelmezése nincs.

-in meghatározza azt a widgetet, amelybe elhelyezi a kérdéses elemet a szülő widget helyett.

7.19. Eseménykezelés

A perl-tk fel van készítve aszinkron események kezelésére, mint például egy egér gomb lenyomására, egy billentyű lenyomására, vagy egy objektum ütközésére a képernyőn.
Az esemény kezelés azt jelenti, hogy az esemény bekövetkezésekor a program meghív egy előre megadott szubrutint, amely — várhatóan — a kívánt műveletsort elvégzi.
Nézzük a következő példaprogramot:

```
1  #!/usr/bin/perl
2  use Tk;
3  $mw=MainWindow->new();
4  $mw->bind('<KeyPressed-a>',\&rut)
5  MainLoop();
6  sub rut
7     {
8     print '.';
9     }
```

A program egy 'a' karakter leütésére egy pontot rajzol ki az indító konzolra. Ha tehát a fő ablak aktív és a kívánt esemény megtörténik — itt az 'a' lenyomása —, akkor a kérdéses szubrutin meghívásra kerül. Az esemény kiválasztása és a szubrutin megadása a 4. sorban látható.

7.19.1. Események és módosítók

A legfontosabb kezelt események a következők:

`ButtonPress` vagy `Press` a kérdéses egérgombot lenyomták. A gomb lehet 1 2 3 4 5. Példa: `<ButtonPress-1>`

`ButtonRelease` a megfelelő egérgombot felengedték,

`Configure` a kérdéses widget mérete, vagy pozíciója a képernyőn megváltozott,

`Destroy` a widgetet törölték,

`Enter` a cursor a widget területe fölé került,

`FocusIn` a widgetre került a fókusz,

`FocusOut` a widget már nem kiválasztott,

`KeyPress` vagy `Key` egy billentyű lenyomásra került a klaviatúrán. Példa: `<KeyPress-a>`. Ha nem adunk meg semmilyen billentyűt, akkor bármelyik billentyű lenyomására hívja a kezelő függvényt.

`KeyRelease` egy billentyű felengedésre került a klaviatúrán,

`Motion` a cursor mozog a widget felett,

`MouseWheel` az egér mozog a widget felett,

`Leave` a cursor elhagyta a widget területét,

`Map` a widget láthatóvá vált a képernyőn,

`Unmap` a widget nem látható a képernyőn.

Amennyiben az eseményeket módosítani akarjuk ezt a következő módon tehetjük meg:

```
<modifier-modifier-type>
```

A kérdéses módosító lehet:[52]

`Alt` az alt billentyű,

`Control` a control billentyű,

`Shift` a shift billentyűk, `Meta`, `Mod3`], `Lock`

`Double` az egér gombjának kétszeri lenyomása,

`Triple` az egér gombjának háromszori lenyomása.

7.19.2. Paraméter átadás a ”callback” rutinnak

Az eseménykezelésnél gyakran előfordul, hogy a meghívott szubrutinnak valamilyen paramétert szeretnénk átadni például azért, hogy azonosítani tudjuk az esemény kiváltóját. Ebben az esetben az eljárás a következő:

```
$widget->bind(<event>,[\&rutin, parameter]);
```

Ha ezt esemény kezelőnél használjuk, akkor a paraméter a második átadott változó lesz, esetleg érték, mert az első a kérdéses widget azonosítója (egy memória cím).
Amennyiben ez egy widget `-command` paramétere után jön, akkor a kérdéses változó, vagy érték az első átadott paraméter lesz a hívott szubrutinba.

7.19.3. Esemény struktúra

Ez a megoldás egy másik fajta azonosítása a hívónak, vagy a hívást kiváltó eseménynek. Ebben az esetben a hívó widget `XEvent` függvényét használjuk a következő példa alapján:

```
1  $mw->bind(<KeyPress>,\&rut);
2  sub rut
3    {
4    my ($w)=@_;
5    my $e=$w->XEvent();
6    print "chr=$e->K code=$e->N\n";
7    }
```

Az 1. sorban egy szabványos hozzárendelést látunk.
A 4. sorban az átadott paraméterek közül az elsőt levesszük ez a hívó widget címe.
Az 5. sorban az `$e` skalárba beletesszük az esemény azonosítót az `XEvent` függvény segítségével. A `$e` skalár az esemény struktúra címét tartalmazza.
A 6. sorban kiíratjuk az esemény kiváltójának karakteres kódját (`$e->K`) és a decimális kódját (`$e->N`).
Ezáltal könnyebben azonosíthatjuk a szubrutin hívóját és a hívást kiváltó eseményt.

[52] A `Meta`, `Mod3`] és a `Lock` billentyű a normál PC-s klaviatúrákon általában nem szerepel. Néhány alkalmazásban a `Meta` billentyű a baloldali `Alt`, ezt azonban a perl-tk nekem sima `Alt` gombként kezeli, `Meta`-ként nem.

7.19.4. A `Canvas` eseményei

A `Canvas` saját eseménykezelővel rendelkezik, ezért `Canvas` esetén a hozzárendelés a következő módon történik:

```
$canvas->CanvasBind('<Event>',\&callback);
```

Ha a `Canvas`-on belüli objektumra akarunk definiálni egy eseményt, akkor a következő eljárást célszerű használni:

```
$c=$mw->Canvas(-width=>100,-height=>100)->pack();
$b=$c->createRectangle(10,10,40,40,-fill=>'blue');
$c->bind($b,'<1>',\&rut);
```

Ez a programrészlet a kék téglalap fölötti a baloldali egérkattintásra hívja meg a `rut` szubrutint.[53]
Amennyiben `tag`-re akarunk eseményt definiálnunk, akkor a `$b` helyére a `tag` nevét kell írnunk.

7.19.5. Fájl esemény

Egy rendszerhívás blokkolhatja a `perl-tk` működését. Ez azt jelenti, hogy egy például egy fájl olvasására esetleg írására való várakozás megállíthatja a rendszer futását és a programunk megfagy mindaddig, amíg a kérdéses esemény meg nem történik.
A megoldás kétféle lehet:

1. a kérdéses rendszerhívást úgynevezett nem blokkoló módon tesszük meg,
2. a `perl-tk` fájl esemény kezelőjét használjuk.

Az első megoldás mindig működik, akkor is ha nem használjuk a grafikus felületet. Erre az `fcntl` függvényt használjuk. Használata a következő példán látható:

```
$flags=fcntl(FILE,F_GETFL,0);
fcntl(FILE,F_SETFL,$flags|O_NONBLOCK);
```

Az első sorban lekérdezzük az adott állomány (a `FILE` azonosítja) beállításait, ezt fogja a `$flags` skalár tartalmazni.
A második sorban a már lekérdezett beállításokra felmaszkoljuk az `O_NONBLOCK` jelzőbitet és ezt át is adjuk az állománynak.

```
$a=<FILE>;
```

Ekkor a fent látható sor olvasásnál a program nem áll meg, hanem továbbfut és a visszaadott érték üres sztring lesz. A beolvasás akkor történik meg, ha a fájlban egy soremelés karakter lezárja a sort, vagy a kérdéses fájlt lezárták.
A második megoldás esetén az ablakhoz egy fájl eseményt kötünk. Ez lehet: a fájl olvasható lett, vagy a fájl írható lett. Ha az esemény bekövetkezik, akkor a meghívja a megjelölt szubrutint, amely olvas az állományból. A program nem áll meg, mert a rutin meghívásának feltétele volt, hogy a fájlba legyen kiolvasható tartalom. A következő program erre példa:

[53] Igen ez így működik, az irodalom erre is a `CanvasBind`-et írja, de az nem működik kipróbáltam.

```
1  #!/usr/bin/perl
2
3  use Tk;
4  $mw=MainWindow->new();
5  $mw->title("Ablak");
6  $mw->geometry("200x200+250+250");
7
8  $mw->fileevent('STDIN','readable',\&rut);
9  $b=$mw->Button()->pack();
10 MainLoop();
11 sub rut
12   {
13   $a=<>;
14   print $a;
15   }
```

A 8. sorban rendeljük hozzá az `STDIN` olvashatóságához a `rut` szubrutin meghívását. Ha van beolvasható adat, akkor a szubrutin a meghívása után ezt kiolvassa, majd kiírja az indító terminálra.
A gomb csak arra kellett, hogy láthatóan a program nem ”fagyott be”.
A `fileevent`-nél az események lehetnek:

`'readable'` a fájl olvashatóvá vált,

`'writable'` a fájl írhatóvá vált.

7.20. Összetett widgetek

A `perl-tk` előre elkészítve számos összetett widgetet tartalmaz. Ezek ismertetése bőven meghaladja ennek a jegyzetnek a tartalmát, ezért csak felsoroljuk őket:

`Dialog` igen, nem döntést tesz lehetővé és lőre lekészíthető, mert van `show` függvénye,

`messageBox` lásd az előző widgetet, de létrehozásakor azonnal megjelenik,

`DialogBox` beállítható dialógus doboz. Elhelyezhetők rajta gombok, beviteli mezők. Példa: login és password bekérés.

`chooseColor` színkiválasztást tesz lehetővé,

`getOpenFile` egy fájlnév kiválasztását teszi lehetővé grafikusan,

`getSaveFile` mint az előző, csak a neve más,

`Ballon` helyzetérzékeny help szövegek kiírására szolgáló widget,

`BrowseEntry` keresési feltétel bevitelére szolgáló widget,

`LabFrame` olyan `Frame`-et hoz létre, amelynek a kiválasztott élére szöveget lehet írni,

`NoteBook` több oldalnyi szöveges információ rendezett megjelenítésére szolgáló widget,

`Pane` görgethető `Frame`,

`ProgressBar` előrehaladást jelző widget (bitkolbász).

8. Folyamatok közötti kommunikáció (IPC)

Az IPC betűszó az angol **I**nter**p**rocess **C**ommunication rövidítésből ered, ami a futó folyamatok közötti kommunikációt jelenti.
Ebbe a fogalomkörbe tartozik az egy gépen belüli információ csere, de ide tartozik a gépek között megvalósított, hálozaton keresztül történő kommunikáció is. Formái a tejesség igénye nélkül:

1. foglalatok (sockets),
2. csővezetékek (pipes),
3. megosztott memória (shared memory),
4. jelzések (signals),
5. szemaforok,
6. feltételes változók,
7. mutex-ek.

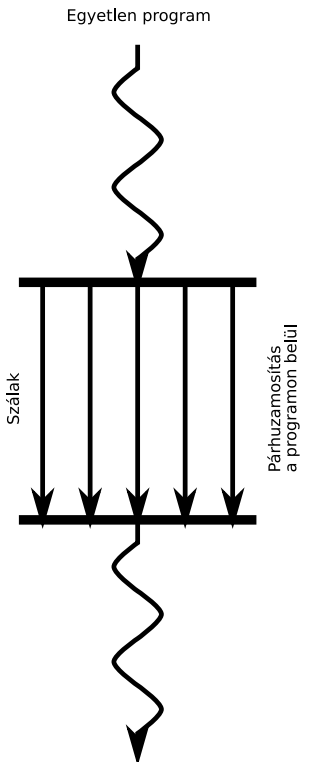
8.1. Konkurrens programozás

Mielőtt a folyamatok közötti kommunikációról szót ejtenénk egy pár alapvető fogalmat tisztázni kell.

Definíció: Folyamatnak tekinthetünk minden egyes futó programot — az általa lefoglalt memóriával és egyéb eroforrásokkal együtt. (Gyakran használják hasonló értelemben a taszk és a processz elnevezést is.)

Definíció: Multi taszkos operációs rendszer képes látszólag, vagy valóságban több folyamatot futtatni azonos időben.

Amennyiben a redszereben egy valós processzor van, akkor egyértelmű, hogy azonos időben csak egy folyamat futhat. Azonban az operációs rendszer megosztva erőforrásait, adott ütemezési stratégiával azt a látsztot kelti, mintha az elindított folyamatok egyidejűleg futnának.
Ha a rendszer több, mint egy fizikai processzort tartalmaz lehetőség van valós multitaszkingra, tehát a processzorok számával megegyező folyamatok egyidejű futtatására.
Megjegyzés: szinte mindig nagyobb a folyamatok száma, mint a fizikai processzorok száma, ezért az ütemezési stratégia akkor is fontos, ha a rendszer több processzoros.



Definíció: A multi threading (szál) program futtatás olyan párhuzamosítást jelent, amelyben egy folyamat rendelkezik több jól elkülöníthető részekkel, amelyek képesek valós, vagy látszólagos párhuzamos futásra.

Tehát a multi threading program szintű, mígy a multi tasking operációs rendszer szintű párhuzamosítás.

8.2. Szálak a perlben

A szálak kezelése a perl-ben rendkívül egyszerű.

8.2.1. Szál létrehozása

A következő példa egy szál létrehozását mutatja be:

```
1  #!/usr/bin/perl
2  use Thread;
3  $th1=Thread->new(\&thr1);
4  $th1->join();
5  sub
6    {
7    print "Child\n";
8    }
```

2. sorban a szálkezeléshez szükséges modul kerül betöltésre,

3. sorban a létrehozzuk a szálat. A `new` helyett írhatunk `create`-t is.

4. sorban a fő szál arra vár, hogy a gyerek szál befejezze a futását,

7. sorban a gyerek szál kiírja, hogy ő ki.

8.2.2. További szál függvények

Osztály metódusok

`new,create` mint azt az előző példában láttuk, ezen függvények egyike teszi lehetővé a szál létrehozását. Amennyiben értéket szeretnénk adni a futó szálnak induláskor, akkor ezt a következő módon tehetjük meg:

```
$retval=Thread->new(\&thread_function,$arg1,$arg2,...);
```

vagy

```
$thread_object=Thread->new(\&thread_function,@parlist);
```

A szál létrehozása történhet névtelen függvényként is:

```
$$thread_object=Thread->new(sub {...},@parlist);
```

`list` megadja az aktuálisan futó szálak listáját.

```
@th_list=Thread->list();
```

self egy szál a saját szál objektumát a `self` függvény meghívásával kaphatja meg.

```
$this_thread=Thread->self();
```

Ha a szál azonosítóra van szükségünk, akkor:

```
$this_thread_id=Thread->self->tid();
```

kifejezést használjuk.

yield a kérdéses szál átadja a processzort más szálaknak.

```
$this_thread_id=Thread->yield();
```

Objektum metódusok

join a szülő vár a gyerek szálra, hogy befejezze a futását. Ez megakadályozza azt, hogy a szülő kilépjen anélkül, hogy a gyerek befejezte volna a futását. A `join` esetén a gyerek üzenetet küldhet a szülőnek a következő módon:

```
$retval=th1->join();
```

vagy

```
@retlist=th1->join();
```

detache amennyiben nincs szükségünk egy szál visszatérési értékére, de azt meg akarjuk szüntetni a `detache` ezt megteszi.

eval ha a szál valamilyen várátlan hibával kilép, akkor ez nem okvetlenül jelenti a teljes program azonnali leállását. Ilyen esetekben célszerű az `eval` függvény használata.

```
1  $retval = $th1->eval();
2  if($@)
3    {
4    warn "thread hiba: $@";
5    }
6  else
7    {
8    print "Thread vissza: $retval\n";
9    }
```

1. az `eval` hívása a `join` helyett,

2. ha a `$` tömb nem üres hiba keletkezett,

6. ha a tömb üres nincs hiba.

[`tid`] a perl a szálak azonosítására egy úgynevezett ”therad identification number” nevű változót használ. Ennek értékét adja vissza a `tid` függvény.

`equal` segítségével eldönthető, hogy két szál azonos-e. Kezelése:

```
$th1->equal($th2);
```

8.3. Jelzések

A jezések (signal) arra szolgálnak, hogy egy futó program egy másik futó programnak valamilyen jelzésértékű üzenetet tudjon adni.
Vegyük példának a következő helyzetet. Adott egy folyamat, amely valamilyen számítást végez és adott egy másik folyamat, amely ennek a számításnak a befejeztére var (most az eredmény átküldése és további feldolgozása lényegtelen). Az első folyamat az említett másodiknak egy üzenetet küld akkor, amikor a számítási feladat befejeződött. Ekkor a második folyamat a jelzés hatására egy művelet sort elkezd.
Első lépésként foglalkozzunk a jelzés küldésével. Erre a `kill` függvény szolgál.

```
$COUNT=kill SIGNAL, PRLIST
```

A `SIGNAL` a jelzés megnevezése, vagy száma. A lehetséges jelzések:

`0` a ’0’ egy olyan jelzés, ami nem kerül elküldésre, viszont a rendszer ellenőrzi, hogy a kérdéses folyamat létezik-e (lásd visszatérési érték).

`SIGABRT` felhívja a fogadó folyamatot, hogy az összes megnyitott állományt lezárja és futását fejezze be. Ez a jelzés nem blokkolható.

`SIGARLM` riasztás jelzés, legyakrabban alvó folyamatoknak küldik, hogy ”felébresszék azokat”.

`SIGFPE` ez a jelzés akkor kerül adásra, ha valamilyen aritmetikai hiba keletkezik számítások közben.

`SIGHUP` a folyamat futásának felfüggesztését kéri a jelzés küldője. Ezt a jelzést el lehet kapni. Általában akkor kapja a folyamat, ha a terminálját bezárták.

`SIGILL` ezt a jelzést valamilyen illegális művelet végrahajtása után szokta a rendszer küldeni. Ezt a jelzést el lehet kapni.

`SIGINT` ez a jelzés ”megaszkítás” jelzésre vonatkozik. Például ha adott egy futó program és az indító terminájában `<CTRL-C>` leütése történt. Ez a jelzés elkalpható.

`SIGKILL` erre a jelzésre a futó program azonnal befejezi működését. Ez a jelzés nem kapható el.

`SIGPIPE` ezt a jelzést a rendszer akkor küldi a folyamatnak, ha az olyan csővezetékre akar írni, amleynek a másik oldalán nincs másik folyamat. Ez a jelzés is elkapható.

`SIGQUIT` ez a jelzés akkor keletkezik, ha felhasználó kilépés karaktert (`CTRL-\`) küldött.

SIGSEGV ezt a jelzést a rendszer akkor küldi, ha a kérdéses folyamat illegális memória hozzáférést valósít meg (Segmentation Fault).

SIGTERM a jelzés hatására a kérdéses folyamat befejezi futását. Ez a jelzés elkapható.

SIGUSER1 a felhasználó által definiált és felhasznált jelzés. El kell kapni.

SIGUSER2 a felhasználó által definiált és felhasznált jelzés. El kell kapni.

A `PRLIST` azoknak a folyamatoknak a PID-jét tartalmazza, akiknek a jelzést elküldjük (PID **P**rocess **Id**entification number). A `PRLIST` lehet sima felsorolás, vagy tömb.
A kimeneten a `$COUNT` skalár változóban azoknak a processzeknek a számát kapjuk meg, ahányan megkapták a jelzést.
A jelzés elkapása, vagy lekezelésére a perl egy egyszerű hash-tömböt használ. A következő példa azt mutatja meg, hogyan kapjunk el egy `SIGINT` jelzést.

```
$SIG{INT}=\&int_fgv;
sub int_fgv
   {
   printf("Hehe\n");
   }
```

Ha a kérdéses jelzést nem akarjuk elkapni, csak figyelmen kivül hagyni, akkor használjuk a:

```
 $SIGINT='IGNORE';
```

kifejezést (az `INT` persze lehet más is, kivétel az el nem kaphatók).
Ha vissza akarjuk a program futása során állítani az eredeti kezelőt, akkor:

```
 $SIGINT='DEFAULT';
```

kifejezést kell használni.
A perl lehetővé teszi, hogy a jelzéseket ideiglenesen is átirányítsuk egy szubrutin futásának idejére. Ekkor célszerűen a kérdéses hash elemet `local`-nak kell definiálni.

```
sub here_is_no
   {
   local $SIG{INT}='IGNORE';
   printf("Hehe\n");
   }
```

Amikor a program belép a szubrutinba a jelzés hatástalan lesz. A szubrutin elhagyása után az előzőleg beállított módon fog reagálni.
A kezelő szubrutin itt is lehet névtelen.

```
   $SIG{INT}=sub { printf "Hehe\n"}
```

8.4. Csővezetékek

8.5. Foglalatok (sockets)

Ez a részfejezet a hálózati kapcsolatok programozásával foglalkozik. A UNIX jellegű rendszerek ezt a feladatot — éppúgy, mint a feladatok legtöbbjét — igen kultúráltan egy fájl interfészen keresztül valósítja meg.

8.5.1. Egy kis hálózatelmélet

Ez sajnos elengedhetetlen ahhoz, hogy értsük, hogy mi is történik egy kapcsolat felvételnél.
A foglalatok segítségével egy kliens-szerver kapcsolatot lehet felépíteni.
A kliens felépítése (a hálózati szempontokat figyelembe véve) nagyon egyszerű. Feladata csak annyi, hogy létrehozzon egy foglalatot és ezzel csatlakozzon a szerverhez.
A szerver a kliensnél bonyolultabb. Itt kétfajta foglalatot használunk. A szerver mikor még nem kapcsolódtak hozzá figyeli a hálozatot ez a **szerversocket** segítségével végzi, ha egy kliens kérelme befut, akkor az adott kliens felé is kiépít egy un. **kliensocketet**. Az adatcsere klienssocketen keresztül történik a szervesockettől függtelenül. A szerversocket pedig várhat (nem biztos, hogy teszi) a következő kliensre.

8.5.2. Egyszerű TCP kliens

A következő példa a legegyszerűbb TCP klienst mutatja be.

```
1  #!/usr/bin/perl
2  use Socket;
3  $proto=getprotobyname('tcp');
4  socket(SOCKET,PF_INET,SOCK_STREAM,$proto);
5  $addr=inet_aton('localhost');
6  $port=15000;
7  $paddr=sockaddr_in($port,$addr);
8  connect(SOCKET,$paddr) or die("Nem sikerult!\n");
9  print <SOCKET>;
10 close(SOCKET);
```

A program működése a következő:

2. A socket létrehozásához a `Socket` modult kell használnunk.

3. sorban mgehatározzuk a protokoll típusát. Ez jelen esetben TCP lesz.

4. sorban létrehozunk egy `SOCKET` nevű fájl leírót, amely:

   `PF_INET` internet típusú,

   `SOCK_STREAM` sorrendtartó, meggbízható kapcsolatú,

   `$proto` változóban megadott protokollal rendelkező

   kapcsolat lesz.

5. sorban meghatározzuk a szerver IP címét.

6. sorban megadjuk a kapcsolat prort számát. Ez most 15000.[54]

7. sorban az IP címből és a port számából egy pakolt változót állítunk elő.

8. sorban megkísérreljük a kapcsolódást a szerverhez. Ha siketelen kilépünk a programból.

9. sorban kiírjuk a terminálra, amit a hálózaton kaptunk.

10. sorban lezárjuk a kapcsolatot.

8.5.3. Egyszerű TCP szerver

```
1  #!/usr/bin/perl
2  use Socket;
3  my $proto = getprotobyname(tcp);
4  socket(SOCKET, AF_INET, SOCK_STREAM, $proto) or die "Socket nem!";
5  $port=15000;
6  $paddr = sockaddr_in($port, INADDR_ANY);
7  bind(SOCKET, $paddr) or die "Bind nem!";
8  listen(SOCKET,5) or die "Listen nem!";
9  while(1)
10    {
11    if(accept(CLIENT, SOCKET))
12      {
13      print CLIENT "Hello!\n";
14      close CLIENT;
15      }
16    }
```

3. sorban a protokolt határozzuk meg. Miután ez a szerver alkalmazás az előző kliens számára készül a protokoll it is TCP.

4. sorban megadjuk a port számát ez szintén 15000.

5. sorban létrehozzuk a pakolt formátumú cím leírót. Ne feledjük el azt, hogy ez egy szerver alkalmazás, ezért nem konkrét IP címet adunk meg, hanem egy tartományt. Ezt az `INADR_ANY` ”címmel” helyettesítjük, amely szerint a szervert tetszőleges (na jó a domainon belüli) IP címről el lehet érni.

6. sorban létrehozzuk a szerversocketet.

7. sorban jelezzük a kernelnek, hogy kapcsolatokat szeretnénk fogadni. jelen esetben maximálisan 5 darabot (ezt jelenti az 5).

[54] A port száma mindenképpen 1024 fölötti legyen, mert az ez alatti értékűek biztosan a rendszerhez vannak rendelve. Ez a 15000-s érték nagyon jó bevált.

11. sorban figyeljük, hogy létrejött-e a kapcsolat, ekkor az `accept` függvény létrehoz egy klienssocketet, amelyen keresztül a későbbiekben az adatforgalom történik.

13. sorban elküldünk egy ”Hello!” üzenetet a kliensnek.

14. sorban bezárjuk a klienssocketet.

Egy gyakorlati tanács: nagyon neház úgy programot éleszteni, hogy a szerver és a kliens is egyidejűleg készül és hiba esetén nem tudjuk melyik nem működik. A megoldás az, hogy először a szervert célszerű megírni és ezt a telnet programmal lehet tesztelni. A szükséges parancs esetünkben:

```
telnet 127.0.0.1 15000
```

ahol a 127.0.0.1 IP cím az úgynevezett localhost, vagyis az aktuális gép.

8.5.4. Kliens az `IO::Socket` modullal

Ez a módszer kicsit egyszerűbb, átláthatóbb, mint az előbbi ”sima” megoldás, de talán a ”sima” kicsit jobban beállítható.

```
1  use IO::Socket;
2  $socket = IO::Socket::INET->new(PeerAddr => $remote_host,
3                                  PeerPort => $remote_port,
4                                  Proto    => "tcp",
5                                  Type     => SOCK_STREAM)
6            or die "Hiba: $@\n";
7  print $socket "Hello\n";
8  $answer = <$socket>;
9  close($socket);
```

1. betöltjük a megfelelő modult,

2. létrehozzuka a megfelelő foglalatot. A sor további résében megadjuk a szerver IP címét.

3. megadjuk a port számát,

4. meghatározzuk a protokollt,

5. megadjuk a kapcsolat típusát,

6. hiba esetén üzenetet küldünk és kilépünk a programból,

7. a szerver felé üzenetet küldünk,

8. megkapjuk a választ a szervertől,

9. lezárjuk a kapcsolatot.

Ha a megnyitás sikertelen, akkor a `$socket` skalár értéke nem definiált.

8.5.5. Szerver az `IO::Socket` modullal

```
1  use IO::Socket;
2  $server = IO::Socket::INET->new(LocalPort => $server_port,
3                                  Type      => SOCK_STREAM,
4                                  Reuse     => 1,
5                                  Listen    => SOMAXCONN)
6      or die "Nem sikerult $server_port : $@\n";
7  $client = $server->accept())
     ⋮
8  close($server);
```

2 létrehozzuk a foglalatot. A szerver esetén nincs megadva IP cím, mert ezt a kliensnek kell tudnia. A szerver nem kezdeményez kapcsolatot a klienssel.

4 ha a `Reuse` opció igazra van állítva és program szabálytalanul áll le, akkor újraindításnál hibajelzést ad, hogy a foglalatot már használjuk.

5 a `Listen` paraméter meghatározza, hogy hány klienst várakoztat a szerver miközben ők a szerverrel kapcsolatot kívánnak létesíteni,

6 a hibakezelés,

7 a kliens foglalat létrehozása,

⋮ ide írjuk a hálózati kommunikációt (ez most kimaradt),

8 a kliens foglalat lezárása.

8.5.6. Szerver `fork`-al

Egy szerverhez egyidejüleg több kliens is csatlakozhat az eddigi programjaink csak egy kliens csatlakozását engedték meg. Persze ezt a problémát meg lehet oldani sokféle módon, de a leginkább elterjett megoldás a `fork` használata.
A `fork` függvény a futó processzből egy másolatot készít.

```
1  #!/usr/bin/perl
2  use IO::Socket;
3  use POSIX qw(:sys_wait_h);
4  socket(SERVER,PF_INET,SOCK_STREAM,getprotobyname(' tcp'));
5  setsockopt(SERVER,SOL_SOCKET,SO_REUSADDR,1);
6  $my_addr=sockaddr_in("15000",INADDR_ANY);
7  bind (SERVER,$my_addr) or die "bind hiba\n";
8  listen (SERVER,SOMAXCONN) or die "listen hiba\n";
9  $SIG{CHLD}= \&REAPER;
```

```
10  print ("Kliensre varunk\n");
11  while ($client_addr = accept(CLIENT,SERVER))
12    {
13    $pid = fork;
14    die "fork hiba: $!" unless defined $pid;
15    print "Pid: $pid\n";
16    if ($pid == 0 )
17      {                                          # Ez mar a gyerek
18      $sor=<CLIENT>;
19      print CLIENT "Ezt kuldted: $sor";
20      exit;
21      }
22    }
23  close (SERVER);
24  sub REAPER
25    {
26    until(-1==waitpid(-1,WNOHANG)){};
27    $SIG{CHLD}=\&REAPER;
28    }
```

A program a 11. sorig gyakorlatilag megegyezik az első TCP szerver alkamazással, kivéve a 9. sort. A fő eltérés a 13. sorban történik.

9. sorban egy jelzéskezelőt állítunk be a `CHLD` jelzésre. Ez a jelzést majd a gyerek folyamat adja, ha befejezi működését és kilép.

13. a program egy másolatot készít magából, ha a 11. sorban látható `accept` függvény elfogad egy kapcsolódási kérelmet.

16. sorban döntjük el, hogy az éppen aktuális folyamat szülő-e, vagy gyerek. Ha szülő a `$pid` értéke nem nulla, hanem a létrehozott gyerep azonosítója. Ha gyerek folyamat, akkor ez a skalár nulla értékű.

18.-20. sorokban történik meg a kliens kezelése. Egy sort olvasunk be a kliensről, majd egyszerűen visszaküldjük és megszüntetjük a kapcsolatot.

24.-28. sorok egy jelzéskezelő rutint takarnak, ami a gyerek processzek kilépési jelzéseit kezeli le. Ha ezt a jelzéskezelőt nem használjuk a gyerek folyamat zombivá válik és fölöslegesen terheli a gépet.